Scalable Algorithms for Global Snapshots in Distributed Systems

Rahul Garg
IBM India Research Lab
Block-I, IIT Delhi,
Hauz Khas-16
New Delhi, India
grahul@in.ibm.com

Vijay K. Garg*
ECE Department
The University of Texas
at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu

Yogish Sabharwal
IBM India Research Lab
Block-I, IIT Delhi,
Hauz Khas-16
New Delhi, India
ysabharwal@in.ibm.com

ABSTRACT

Existing algorithms for global snapshots in distributed systems are not scalable when the underlying topology is complete. In a network with N processors, these algorithms require $O(N)$ space and $O(N)$ messages per processor. As a result, these algorithms are not efficient in large systems when the logical topology of the communication layer such as MPI is complete. In this paper, we propose three algorithms for global snapshot: a grid-based, a tree-based and a centralized algorithm. The grid-based algorithm uses $O(N)$ space but only $O(\sqrt{N})$ messages per processor. The tree-based algorithm requires only $O(1)$ space and $O(\log N \log w)$ messages per processor where w is the average number of messages in transit per processor. The centralized algorithm requires only $O(1)$ space and $O(\log w)$ messages per processor. We also have a matching lower bound for this problem. Our algorithms have applications in checkpointing, detecting stable predicates and implementing synchronizers. We have implemented our algorithms on top of the MPI library on the Blue Gene/L supercomputer. Our experiments confirm that the proposed algorithms significantly reduce the message and space complexity of a global snapshot.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*snapshot, stable predicates*; D.4.5 [**Operating Systems**]: Reliability—*Checkpointing*

General Terms

Algorithms

Keywords

Checkpointing, Global Snapshot Algorithms, Fault tolerance, Stable Predicates, Blue Gene/L

1. INTRODUCTION

Computing the global snapshot of a system is a fundamental problem in distributed computing. It has applications in fault-tolerance of long-running programs by providing an intermediate *checkpoint* of the system. In case of a failure, the system can restart from the checkpoint instead of the beginning of the program.

Global snapshots are also useful in monitoring stable properties of the system. A property is stable if, once it becomes true, it stays true. Some examples of stable properties are termination, deadlock, loss-of-a-token etc. By repeatedly computing the global snapshot and evaluating the property on the computed snapshot, one can detect any stable property.

All the existing global snapshot algorithms for non-FIFO channels [9, 10], require at least one message and one integer to be stored for every channel in the system. In a system with N processors and a completely connected topology, this translates to $O(N)$ messages and $O(N)$ space per processor. In massively parallel computers, this overhead can be quite significant. For example, the IBM Blue Gene/L computer has $64K$ processors. Furthermore, at the application level a process at any processor may send message to any other processor. Thus, the topology at the application level is that of a completely connected graph. Using existing algorithms on such a system would result in $64K$ messages per processor per snapshot. In this paper, we propose three algorithms with different characteristics that make them much more scalable than existing algorithms. We call these algorithms grid-based, tree-based and centralized algorithms.

The grid-based algorithm assumes a logical grid-like structure of the system. It requires $O(\sqrt{N})$ messages per processor with each message of size $O(\sqrt{N})$ integers. For Blue Gene/L, this algorithm would use 256 messages per node instead of $64K$ messages. Although each message of our algorithm is bigger than existing algorithms, the total overhead of communication and computation is significantly reduced by using fewer messages. The space complexity of the grid-based algorithm is similar to existing algorithms. It uses $O(N)$ space at every processor.

Our tree-based algorithm reduces the space complexity of

[*] This work was performed when the author was visiting IBM India Research Lab

ICS06, June 28-30, Cairns, Queensland, Australia.

the global snapshot problem. It uses a constant number of variables at each processor and $O(\log N \log w)$ messages per processor where w is the average number of in-transit messages when the snapshot is taken. A crucial difference between the earlier algorithms and our tree-based algorithm is that earlier algorithms relied on informing each process the number of in-transit messages. The tree-based algorithm avoids this step to reduce the space complexity at each process. The tree-based algorithm uses $O(N \log N \log w)$ messages in all.

The total message complexity of the system can be further reduced to $O(N \log w)$ messages by using a centralized algorithm. The disadvantage of the centralized algorithm is that it may require a single node to process as many as $O(N \log w)$ messages.

We also have a matching lower bound on the message complexity of any global snapshot algorithm that is based on detecting when all in-transit messages have been received. We show that detecting whether W messages have been received in a system with N processes requires at least W control messages when W is at most N and $\Omega(N \log W/N)$ when W is greater than N.

The characteristics of the algorithms proposed in this paper are summarized in Figure 1. The algorithm CLM refers to Chandy and Lamport's algorithm with Mattern's modification for non-FIFO channels. It requires $O(N^2)$ messages in all when the underlying topology is completely connected. The centralized algorithm has the least message complexity; however, it requires the coordinator node to process $O(N \log w)$ messages.

Algorithm	Message Complexity	Message Size	Space
CLM	$O(N^2)$	$O(1)$	$O(N)$
Grid-based	$O(N^{3/2})$	$O(\sqrt{N})$	$O(N)$
Tree-based	$O(N \log N \log w)$	$O(1)$	$O(1)$
Centralized	$O(N \log w)$	$O(1)$	$O(1)$

Notation: N: Number of processes, w: Average Number of in-transit messages/process

Figure 1: Summary of Global Snapshot Algorithms

This paper is organized as follows. In Section 2, we briefly describe the problem and the existing work. Section 3, 4 and 5 discuss the grid-based algorithm, the tree-based algorithm and the centralized algorithm respectively. Section 6 describes our implementation of the global snapshot algorithm on the Blue Gene/L computer and performance analysis of the algorithms proposed in the paper. Section 7 describes other applications of techniques described in the paper. Section 8 provides concluding remarks.

2. MODEL AND BACKGROUND

We model a distributed system as an asynchronous message-passing system without any shared memory or a global clock. A *distributed program* consists of a set of N processes denoted by $\{P_1, P_2, ..., P_N\}$ and a set of unidirectional channels. A channel connects two processes. Thus the topology of a distributed system can be viewed as a directed graph in which vertices represent the processes and the edges represent the channels. A channel is assumed to have infinite buffer and to be error-free. We do not make any assumptions on the ordering of messages. Any message sent on the channel may experience arbitrary but finite delay. The state of the channel at any point is defined to be the sequence of messages sent along that channel but not received.

A *computation* is defined as a tuple $(E, \rightarrow)$ where E is the set of all events that are generated during the execution, and $\rightarrow$ is the happened-before relation [8] on E. We define a *consistent cut*, or a global snapshot, as any subset $F \subseteq E$ such that

$$f \in F \wedge e \rightarrow f \Rightarrow e \in F.$$

The definition and the first algorithm to compute a consistent global snapshot for a system with FIFO channels was given by Chandy and Lamport in [3]. This elegant algorithm has the property that it does not freeze the underlying computation during global snapshot computation. Thus, the underlying application is not stopped from sending or receiving any messages when the snapshot algorithm is in progress. We will restrict ourselves to algorithms with this property.

Chandy and Lamport's algorithm, which works only for FIFO channels, uses the concept of *color* of messages and processes. Each process is either white or red. Intuitively, the computed global snapshot corresponds to the state of the system just before the processes turn red. All processes are initially white. After recording the local state, a process turns red. One or more processes initiate the snapshot algorithm by recording their local state and turning red. Once a process turns red, it is required to send a special message called *marker* along all its outgoing channels before it sends out any message. A process is required to turn red on receiving a marker if it has not already done so. Since channels are FIFO, the above mentioned rule guarantees that no white process ever receives a message sent by a red process. This property ensures that the global snapshot recorded is consistent.

It is easy to define the state of the channels for the snapshot. These are the messages sent by a white process received by a red process. To record the state of the channel from P_i to P_j, the process P_j starts recording all messages it receives from P_i after turning red. Since P_i sends a marker to P_j on turning red, the arrival of the marker at P_j from P_i indicates that there will not be any further white messages from P_i sent to P_j. It can, therefore, stop recording messages once it has received the marker. When a process has received a marker on all its incoming channels, it has finished recording all incoming channels and its component of the snapshot algorithm is finished.

Chandy and Lamport's algorithm sends exactly one overhead message (marker) along each link. Although the algorithm is efficient for sparse networks it is not scalable for large dense networks. The message complexity of this algorithm is $O(N^2)$.

Mattern's algorithm is an extension of Chandy and Lamport's algorithm to remove the assumption that channels are FIFO [9]. In absence of the FIFO property, the marker message cannot be used to distinguish between white and red messages. Therefore, Mattern's algorithm includes the color in all the outgoing messages for any process besides sending the marker. Further, even after P_i gets a red message from P_j or the marker, it cannot be sure that it will not receive a white message on that channel. A white message may arrive later than a red message due to the overtaking of messages. To solve this problem Mattern's algorithm includes in the marker the total number of white messages sent by that process along that channel. The receiver keeps track of the total number of white messages received and knows that all white messages have been received when this count equals the count included in the marker. The message complexity of this algorithm is also $O(N^2)$.

A different protocol has been implemented more recently by Schulz, Bronevetsky, Fernandes, Marques, Pingali and Stodghill in [10]. This algorithm also uses a message count per channel that indicates the number of white messages sent on the channel. Spezialetti and Kearns have given efficient algorithms to disseminate a global snapshot to processes initiating the snapshot computation [11]. Bouge [2] has given an efficient algorithm for repeated computation of snapshots for synchronous computations. The reader is referred to a [7] for a survey of global snapshot algorithms.

3. GRID-BASED ALGORITHM

The grid-based algorithm reduces the message complexity of sending the white message count by making use of the following two observations. First, a process does not need a separate count of the white messages sent to it by a specific process. All it needs is the total number of white messages that have been sent to it. Whereas earlier algorithms communicate to every process the number of white messages sent *on every channel*, our algorithm communicates only the total number of white messages sent to a process. The second observation is that a process can use one message to send out information about the number of messages it has sent for multiple processes. We use this observation to reduce the number of messages at the expense of increasing the size of messages.

Our algorithm uses the notion of color of processes similar to Chandy and Lamport's algorithm. A process may be white, red, or black. It is white if it has not recorded its local state. It is red if it has recorded its local state but not the state of incoming channels, and black if it has recorded its local state and all the transit messages. Initially all processes are white.

There are three main components in our algorithm. The first component assumes that there is a pre-defined spanning tree in the system. It uses the spanning tree to broadcast the fact that one or more processes want to take the global snapshot. It ensures that all processes in the system turn red with $O(N)$ messages by using the spanning tree. Furthermore, by including the color of the sending process in all messages, the algorithm also guarantee that no white process ever acts on a red message. On receiving a red message a white process immediately turns red. This component is fairly standard, but for completeness sake it is shown in Figure 2.

The second component is required for processes to determine the total number of white messages it is supposed to receive. This component assumes a 2D grid structure to compute the total number of white messages sent to a specific process. We will use $P_{i,j}$ to denote the process at the co-ordinate (i, j) of the grid. For simplicity, we will assume that the grid is a perfect square, although the algorithm can be applied to a grid with any number of rows and columns. Each process $P_{r,c}$ maintains a matrix $whiteSent[i, j]$ which records the number of white messages it sent to process $P_{i,j}$. Further, any process $P_{r,c}$ will also compute $rowCount[j]$ which is the total number of white messages sent by processes in row r to the process $P_{c,j}$. Note that the vector $rowCount$ has size $\sqrt{N}$ and is maintained by each process. Further, all diagonal processes in the grid $P_{c,c}$ maintain an additional vector $totalCount$ such that $totalCount[j]$ equals the total number of white messages sent by all processes to the process $P_{c,j}$.

```
initiate() // enabled only if (color=white)
    take local checkpoint;
    color = red;
    send "init" to all processes connected by tree edges;

On receiving "init" message or a red message on edge e
    if (color = white)
        take local checkpoint;
        color = red;
        send "init" to all tree edges except e
```

Figure 2: Common Initiation Component

This component is shown in Figure 3. In step 1, each process $P_{r,c}$ sends i^{th} row of $whiteSent$ to $P_{r,i}$. Note that this message has $O(\sqrt{N})$ integers and $O(\sqrt{N})$ such messages are sent by each process. In step 2, $P_{r,c}$ receives c^{th} row from all processes in row r. It maintains a cumulative sum of all the vectors received. When it has received values from all the processes in its row, $rowCount[j]$ is equal to the number of white messages sent to $P_{c,j}$ by processes in row r. It then sends $rowCount[.]$ to $P_{c,c}$. In step 3, a process participates only if it is a diagonal processor in the grid. Nodes that correspond to $P_{i,i}$ are responsible for computing the $totalCount$ for all processes in row i. Once $P_{i,i}$ has calculated $totalCount$ for each process in its row, it informs that process with a message of size $O(1)$.

The message complexity of this component is $O(\sqrt{N})$ messages sent/received per node, each of size $O(\sqrt{N})$. This component results in every process knowing $whiteCount$, the total number of white messages that have been sent to the process.

```
$P_{r,c}$::

Step 1: for i:= 0 to numRows-1 do
    send row i of the matrix whiteSent to $P_{r,i}$;

Step 2: receive row c from all processes in row r;
    let rowcount be the cumulative count;
    send rowcount to $P_{c,c}$;

Step 3: if (r=c)
    receive rowcount from all processes $P_{i,c}$ in column c;
    let totalcount be the cumulative count;
    send totalcount[j] to $P_{c,j}$;
```

Figure 3: The Grid-based Algorithm to Compute whiteCount

The third component is responsible for turning all processes black and detecting when that has happened. A process keeps track of all the white messages it has received in the counter *whiteReceived*. When this count equals *whiteCount*, the process knows that it has recorded all possible in-transit messages and turns black. When the entire tree has turned black, the snapshot algorithm is complete. Detecting that the entire tree is black can be performed with $O(N)$ messages by using standard convergecast on the spanning tree [6].

Based on the preceding discussion, we have the following Theorem.

THEOREM 1. *There exists a global snapshot algorithm that requires $O(\sqrt{N})$ messages per node where each message contains $O(\sqrt{N})$ integers.*

Remark: In the above discussion, we assumed that the grid was $\sqrt{N} \times \sqrt{N}$. The algorithm can be generalized to any grid of size $numR \times numC$. When the number of rows is less than the number of columns, the algorithm does not need any change. When the number of rows is greater than the number of columns, then a process $P_{r,c}$ sends $whiteSent[i,.]$ to $P_{r,i \ mod \ numcols}$ instead of sending it to $P_{r,i}$. The algorithm uses $O(numR)$ messages each of size $O(numC)$. Note that when the grid is N by 1, the algorithm reduces to a completely distributed algorithm with $O(N^2)$ messages of size $O(1)$. The other extreme case is when the grid is 1 by N. This case reduces to the centralized algorithm with $O(N)$ messages each of size $O(N)$.

4. TREE-BASED ALGORITHM

The grid-based algorithm requires every process to maintain the number of white messages sent to any process. This information requires $O(N)$ integers to be maintained at every process. This overhead may be prohibitive for high performance computing applications. We now show an algorithm that reduces this overhead to $O(1)$ integers.

In the tree-based algorithm each process does not maintain individual counts for the number of white messages sent. Instead, it maintains the *deficit* which denotes the total number of white messages sent minus the total number of white messages received. Note that the deficit for a process may be negative. Clearly, the total number of in-transit messages that correspond to the global checkpoint equals the sum of all deficits when the processes turn from white to red. Our algorithm is based on computing this number and detecting when all these messages have been delivered.

The tree-based algorithm consists of three components. The first component for initiation is same as the grid-based algorithm. The second component corresponds to computation of the total deficit in the network. This can be accomplished easily using a convergecast in $O(N)$ messages. Let W be the total deficit computed using the second component. The third and final component corresponds to detecting when all in-transit messages have been received. We call this problem the *distributed message counting problem.* The *distributed message counting problem* can be defined as follows. There are W in-transit messages that are destined for N processes. The problem is to detect when all W messages have been received. We are interested in algorithms that are efficient for even large values of W.

We abstract the total deficit as tokens that are distributed in a network. Each token represents a pending message for which the destination is unknown. Whenever a message is received a token is consumed. The goal is to detect when the total number of tokens reduces to zero. A simple algorithm in which a coordinator maintains all the tokens can solve this problem in $O(W)$ messages. Whenever a process receives a white message it simply informs the coordinator who consumes a token.

We now show an algorithm that uses $O(N \log N \log(W/N))$ messages. When W is much larger than N, this algorithm will outperform the coordinator based algorithm. The algorithm works based on rounds. There are at most $\lceil \log W/N \rceil$ rounds. Initially, we divide the total number of tokens W equally among all processes. The maximum number of tokens any process has in the first round is $w = \lceil \log W/N \rceil$. Let w_k be the maximum number of tokens any process has at round k. Our algorithm ensures that $w_{k+1} \leq w_k/2$. Thus the maximum number of tokens owned by a process goes down by a factor of two in every round.

We now describe the algorithm at round k. We use three colors — green, yellow and orange — to label all the processes. A process is orange if it has no tokens and it has received one or more in-transit messages. These in-transit messages have not consumed tokens and such processes would initiate messages to search for tokens. A process that does not have this problem is either yellow or green. A process p_i with r_i tokens is considered green if it has strictly greater than $w_k/2$ tokens and yellow otherwise. Intuitively, green processes are rich, yellow are debt-free, and orange are in debt (and therefore poor).

We organize the processes in a perfect binary tree. The degree and the height of the tree is used only in analyzing the message complexity. The algorithm is correct for any spanning tree.

Our algorithm ensures the following properties:

(I1) A yellow process cannot have a green child.
(I2) The root is green, and
(I3) Any orange node eventually becomes yellow.

It is sufficient to describe the protocol when an in-transit message arrives. This event is detected as arrival of a white message at a red process. For every white message, a token must be consumed. No action is required at the destination to maintain the invariants if consuming a token does not change the color of the process. After all, the invariants are specified only using the color of the process. We now consider two possible transitions that can happen when the color gets changed. The pseudo-code for the algorithm is shown in Figure 4.

First, consider the case when the color changes from green to yellow. This transition can violate invariant (I1) if any of its children is green. To maintain the invariant, the process sends a “swap” message with its tokens to its left child (if any). If the left child is green, it replies with an “accept” message and performs the swap operation. Otherwise, it replies with a “reject” message. When a process receives a “reject” message, it tries the swap operation with the other child. If none of the children is green then the process knows that it does not violate (I1). If this node is root, then it also knows that the entire tree does not have any green node. In this case, it initiates a global “reset” operation that takes

```
On turning from green to yellow
    if any child green
        send (“swap”, tokens) to that child;
    else if root node
        reset for the next round

On turning from yellow to orange
    send (“split”, tokens) to the nearest green ancestor;
```

Figure 4: The Tree-based Algorithm for Distributed Message Counting

the algorithm to the next round.

Note that when a child performs a swap operation, it turns yellow and may violate (I1). So after the swap operation, the child, in turn, may initiate its own swap message. The total number of "swap" "accept" and "reject" messages is $O(\log N)$ because they traverse one path down the tree.

Now consider the case when a node changes from yellow to orange. The process sends a "split" message to its parent. Our algorithm will allow only a green node to split. If the parent is green, it splits its tokens with the requesting process. Otherwise, this message is forwarded to its parent. The message is guaranteed to find a green node due to (I2). When a green node splits its token, it is guaranteed to become yellow. This can now fire up the rule for turning from green to yellow.

The "reset" operation is performed as follows. The root requests all nodes to send their tokens to the root. Once the root has received messages from all processes, it calculates the total number of tokens (in-transit messages) and recalculates w_k for the next round.

By using the above algorithm, and repeatedly halving w_k, we are guaranteed to reach the case when w_k is 1. At that point, we continue with one more round with a process to be green if it has the token, yellow if it has no token and orange if it has no token but a pending in-transit message. When the root detects that the the entire tree has turned yellow, it can signal the end of the global snapshot algorithm.

We now show the following claim.

THEOREM 2. *The algorithm uses $O(N \log N \log(W/N))$ messages.*

PROOF. Initially no node has more that W/N tokens. Let w_k be the maximum number of tokens any process has at round k. We show that $w_{k+1} \leq w_k/2$. The new round begins only when the root turns yellow and both its children are yellow. This implies that no node in the tree has more than $w_k/2$ tokens. Thus, the root node can start the next round with $w_{k+1} = w_k/2$. This argument shows that there are at most $\log(W/N)$ rounds.

In each round, there can be two types of transitions — from green to yellow and from yellow to orange. The yellow to orange transition results in splitting operations. There can be at most N splits because each split turns a green node into yellow. Thus, there can be at most N splits. Each split takes at most $\log N$ messages before it succeeds in finding a green node. Every split is followed by a transition of a node from green to yellow. This transition also takes at most $\log N$ messages.

The number of transitions from green to yellow is also bounded by N. Thus, each round requires $O(N \log N)$ messages. □

5. CENTRALIZED ALGORITHM

We now give an algorithm that uses $O(N \log(W/N))$ messages. The algorithm is similar to the tree-based algorithm, except that it uses a different method for solving the *distributed message counting problem.*

```
If not tail process
    On turning from green to yellow
        send (“swap”,tokens) to tail;
    On turning from yellow to orange
        send (“split”, tokens) to tail;
    On receiving “swap” request
        retain $w_k/2$ tokens,
        send accept message with remaining tokens to
            requestor;
    On receiving “split” request
        send accept message with half the tokens to
            requestor;

else if tail process
    On turning from green to yellow
        If $head \neq$ tail
            send (“swap”, tokens) to $head$;
            $head = head - 1$;
        else
            reset for the next round;
    On receiving “swap” or “split” request
        If $head \neq$ tail
            forward request to $head$;
            $head = head - 1$;
        else
            reset for the next round;
```

Figure 5: The Centralized Algorithm for Distributed Message Counting

The centralized algorithm is very similar to the tree-based algorithm in that it works in rounds, tokens are distributed and maintained as in the tree-based algorithm and processes have colors — green, yellow and orange with the same significance. The difference lies in the way the control messages are exchanged and processed.

The processes are organized as follows. All the green nodes are organized as a list. There is a fixed process that represents the tail of the list. This node always remains green. Let the processes have ranks $0, 1, .., N - 1$. Let the tail be the process with rank 0. The tail maintains the list using only one variable, *head*. The variable *head* stores the first green node in the list. When *head* equals h, then logically, the list of green nodes is $\{h, h-1, .., 0\}$. The algorithm ensures that whenever there is need to remove a green node from the list, only *head* is removed from the list. The list can then be quickly updated simply by decrementing *head*. At the start of every round, *head* is initialized to $N - 1$.

We will show that by arranging the processes in such a manner, only a constant number of message exchanges occur when the process of a colour changes, unlike in the tree-based algorithm, where as many as $\log n$ message exchanges could be required. The disadvantage, however, is that all the processes must interact with the process representing the tail of the list, leading to centralization of message exchanges.

We now describe the protocol when an in-transit (white) message arrives. We now consider the two possible transitions that can happen when the colour gets changed. The pseudo-code for the algorithm is shown in figure 5.

First, consider the case when the color changes from green to yellow. Since nodes are only removed from the head of the green list, this process borrows all except $w_k/2$ tokens from *head*, so that it remains green and *head* turns yellow instead. In order to accomplish this, it sends a "swap" request to the tail. The tail forwards this request to the *head* and removes *head* from the green list, since the *head* is guaranteed to become yellow. The *head*, on receiving the message sends all except $w_k/2$ tokens to the requesting process. If the requestor is the same as *head*, then the tail sends back a reject to the *head*, and removes it from the green list. The *head* on receiving the reject, turns yellow. Note that if the *head* is same as tail, that is there is no other green node, then the tail knows that all the processes have turned yellow. In this case, it initiates a global "reset" operation that takes the algorithm to the next round.

Now consider the case when a node changes from yellow to orange. Since nodes are only removed from the head of the green list, this process borrows half the tokens from *head*. In order to accomplish this, the process sends a "split" message to the tail. The tail forwards this request to the *head* and removes *head* from the green list, since the *head* is guaranteed to become yellow. The *head*, on receiving the message splits its tokens with the requesting process. In case there is no other green node other than the tail, the tail initiates a global "reset" operation that takes the algorithm to the next round.

The "reset" operation is performed as in the case of the tree-based algorithm. The root requests all nodes to send their tokens to the root. Once the root has received messages from all processes, it calculates the total number of tokens (in-transit messages) and recalculates w_k for the next round.

Since there are at most $\log(W/N)$ rounds and in each round there are at most $O(N)$ splits/swaps with $O(1)$ message overhead, we get the following claim.

THEOREM 3. *The centralized algorithm uses $O(N\log(W/N))$ messages.*

6. LOWER BOUND

In this section we show that any algorithm must exchange at least $\Omega(N \log W/N)$ point-to-point control messages to detect termination if $W > N$ and W control messages otherwise.

LEMMA 1. *Let W be the number of messages remaining to be delivered. Then there exists a processor p_W that generates a control message on receiving at most $\lceil W/N \rceil$ messages. Therefore any algorithm for termination detection can be forced to exchange a control message on delivery of at most $\lceil W/N \rceil$ messages (by sending these messages to p_W).*

PROOF. Suppose there is no such processor. Then the adversary could send $W/N + 1$ messages to $W \bmod N$ processors and W/N messages to the remaining $N-(W \bmod N)$ processors. The algorithm would not exchange any control message and therefore would not be able to detect termination. The adversary can therefore force a control message to be generated by sending $\lceil W/N \rceil$ messages to p_W. □

Let W be the remaining messages to be delivered. Now, consider the algorithm of the adversary described in figure 6. We define one *round* to be one iteration of the loop of the adversary.

```
$W$ := Sum of deficits at all processes;
while ( $W > 0$ ) do
    deliver $\lceil W/N \rceil$ messages to processor $p_W$
        (as determined in lemma 1) ;
    $W := W - \lceil W/N \rceil$ ;
```

Figure 6: Algorithm of the adversary

LEMMA 2. *Any termination detection algorithm must exchange at least r control messages where r is the number of rounds executed by the adversary.*

PROOF. Using lemma 1, one control message is forced to be generated by the termination detection algorithm on execution of one round of the adversary's algorithm. The statement follows. □

THEOREM 4. *The number of control messages exchanged by any algorithm to detect termination is*

$$\Omega(N\log(W/N)) \quad \textit{if} \quad W \geq N \textit{ and}$$
$$\Omega(W) \quad \textit{if} \quad W < N$$

PROOF. We will show that the adversary executes W rounds when $W < N$ and $\Omega(N\log(W/N))$ rounds when $W \geq N$.

Consider the case when $W < N$. Only one message is delivered in every round until W becomes 0. Therefore the adversary can execute W rounds.

Now consider the case when $W \geq N$. Let $k = \lfloor \log W/N \rfloor$, i.e., $2^k N \leq W < 2^{k+1}N$. We claim that the adversary executes at least $(k+1)N/4$ rounds. The proof is by induction.

- **Base Case ($k = 0$):** $N \leq W < 2N$
  Two messages are delivered in every round until W becomes $< N$. Thereafter one message is delivered in every round until W becomes 0. Therefore at least $N > N/4$ rounds are executed by the adversary.

- **Induction Case :** $2^k N \leq W < 2^{k+1}N$
  The number of messages delivered in each round is $\lceil W/n \rceil$ until $W < 2^k N$. Thereafter, the number of messages delivered in each round decreases. Therefore the number of messages delivered in any round is no more than 2^{k+1}. Consider the first $N/4$ rounds. The number of messages delivered in these $N/4$ rounds is no more than $2^{k+1}N/4$. Therefore the number of messages remaining to be delivered after $N/4$ rounds is at least $W - 2^{k+1}N/4 \geq 2^{k-1}N$. If the number of messages remaining to be delivered after these $N/4$ rounds is $\geq 2^k N$, then the adversary keeps executing rounds and delivering messages until $2^{k-1}N \leq W < 2^k N$. Now, using the induction hypothesis, the adversary

executes at least $kN/4 + N/4 = (k+1)N/4$ rounds before termination. Substituting the value of k, the adversary executes at least $N(1 + \log W/N)/4$ rounds.

The theorem follows from the discussions above and lemma 2. □

Note that the distributed message counting problem is a special case of the termination detection problem where one tries to detect when all processes are passive and all channels are empty [5]. If one considers only those messages that are sent before checkpoint and defines a process to be passive after checkpoint, our problem is reduced to that of termination detection. It is well known that termination detection may require at least as many messages as the application messages in the worst case[4]. The main difference between the general termination detection problem and distributed message counting problem is that once a process turns passive it can never turn active (even when it receives a message). This difference explains the reduced message complexity.

7. EXPERIMENTAL RESULTS

To compare the performance of the above three algorithms, we wrote a micro-benchmark to simulate application behavior. The benchmark takes two parameters W and M. In the first phase, all the processors send W messages to randomly selected nodes without receiving any message. This is followed by M iterations of a loop in which every processor sends one message to a randomly selected destination and attempts to receives one message using a non-blocking call. Finally every process sends a "finish" message to every other process. After receiving $N - 1$ finish messages, the processes proceed to the last phase where they wait for the checkpointing algorithm to complete.

The checkpointing was initiated at a time picked uniformly at random by every processor from a range T_1 to T_2. In one set of experiments we set T_1 to 950 milliseconds and T_2 to 1050 milliseconds. The mean checkpoint initiation time was 1 second. In another set of experiments we set T_1 to 285 milliseconds and T_2 to 315 milliseconds. The mean checkpoint initiation time was 300 milliseconds.

The checkpoint library resides in a layer between the application and MPI library. It uses a dedicated MPI tag for checkpoint messages. It intercepts all the MPI function calls and processes the checkpoint messages according to one of the three algorithms described earlier.

For the grid based algorithm, the grid is constructed by logically organizing the processors into m rows and n columns ($n = m$ if the number of nodes is a perfect square and $n = 2m$ otherwise). The processors are logically organized in the grid by row major ordering of their ranks which range from 0 to $N-1$. In the convergecast tree the root has rank 0. In general, for a node with rank i, its left child has rank $2i + 1$, its right child has rank $2i + 2$ (and its parent has rank $\lfloor (i+1)/2 \rfloor - 1$). For the tree based algorithm, the convergecast tree is used as the checkpointing algorithm tree as well.

We implemented these algorithms and the benchmark on the Blue Gene/L supercomputer. We ran the benchmark on Blue Gene/L partitions of 32, 64, 128, 256 and 512 nodes. We analyzed the algorithms for the fixed values of 40000 for W, and 50000 for M.

7.1 Results

We recorded the total latency, message sizes and counts, initital deficit and number of rounds for the three algorithms. We now report the performance of the algorithms on these metrics.

7.1.1 *Total Latency*

The total latency is the time elapsed (in microseconds) from when the earliest processor determines that checkpointing has been initiated to when the last processor determines that checkpointing has been completed.

The total latencies (in microseconds) observed for the three checkpointing algorithms when the application was executed with $W = 40000$, $M = 50000$ and mean checkpoint timeout of 1 second, are shown in Table 1.

With a mean checkpoint timeout of 1 second, we observed that all the white messages were sent before the checkpointing algorithm started. Therefore the latency observed reflects the time taken by the checkpointing algorithm from initiation to completion.

The maximum time during active processing is spent in MPI calls. For the grid algorithm, the latencies are very small compared to the tree and centralized algorithms. This is because the number of messages generated is small up to 512 nodes and most of these are processed (sent and received) in parallel on different nodes. There is little difference in the latencies of the tree and centralized algorithms.

Since the maximum time in processing is spent in MPI calls, we calculated the maximum messages processed (sent + received) by any node for different values of N. For the centralized algorithm, the tail node is almost always busy sending/receiving messages to/from other nodes. Therefore, we expected the latency to be proportional to the number of messages being processed by the tail node. For this, we calculated the average time for processing a message by dividing the latency by the sum of the messages sent and received (assuming send and receive MPI calls take roughly the same time). Our hypothesis was confirmed by the consistency observed for time per message. For the grid algorithm, we expected similar results since the diagonal elements of the grid are almost always busy processing messages. However, the number of messages sent and received are too small to account for noise (initiation over convergecast tree, computations, etc.) to make any meaningful inferences. For the tree based algorithm, the message exchange is more dynamic in nature and there is no single node that is almost always busy making it difficult to analyze the latency numbers.

7.1.2 *Message Sizes and Message Counts*

For the grid based algorithms, the messages exchanged in steps 1 and 2 contain one integer (4 bytes) for every column in the grid layout. Therefore the maximum message size is $32 + 4\cdot$(number of columns). The average message size also increases with the increase in the number of columns. Most messages are of type 1 or 2. Type 3 messages only carry a single integer. For the machine sizes of 32, 128 and 512 nodes, the average message sizes for the grid based algorithm were 53.70, 82.70 and 144.97 bytes respectively.

For the tree and centralized algorithms, message sizes are not dependent on the number of nodes in the system. Every checkpointing message has a fixed 32 byte header that contains some checkpointing algorithm specific information such as *message type*, *source*, *destination*, *round number*.

Some messages carry an additional information element (such as requestor or tokens) that take up an additional 4 bytes. Therefore, the maximum message size observed is 36 bytes while the minimum is 32 bytes. The average message size for these algorithms is found to be between 35.95 and 35.98 bytes.

The minimum, maximum and average send message counts observed for the three checkpointing algorithms when the application was executed with $W = 40000$, $M = 50000$ and mean checkpoint timeout of 1 second, are shown in Table 2. Similarly, the receive counts are shown in Table 3.

The minimum and maximum messages sent and received for the grid based algorithm is deterministic and follows from the communication pattern of the algorithm.

Even though the number of messages received and sent for the grid algorithm is small compared to the tree and centralized algorithms, this number increases at a much faster rate with an increase in the number of nodes. Comparing the message counts for the tree and centralized algorithm, observe that the average number of messages sent/received for the centralized algorithm is less than that of the tree based algorithm, however, the maximum message counts are much higher. These observations are in agreement with the theoretical analysis; even though the message count complexity of the centralized algorithm is lower, it suffers from centralization in communication at the tail node.

7.1.3 Optimized Implementation

From the above results, it is evident that the tree-based algorithm outperforms the other two algorithms. We performed a small optimization on the tree and centralized algorithms. We modified these algorithms, so that on the initiation of a round the initial deficit is computed after receiving all the pending white messages. We collected data on 512 nodes using this optimization to analyze the improvements.

The initial deficit counts and number of rounds for the tree and centralized algorithms (both optimized and unoptimized) for mean checkpoint timeout values of 300 ms and 1 second, with $W = 40000$ and $M = 50000$ on 512 nodes are shown in table 4.

When the mean checkpoint timeout is 1 s, checkpointing is initiated after all the data messages have been sent and received. Therefore, in the optimized versions of the algorithms, the initial deficits are straightaway computed as 0 and the algorithms terminate in the first round.

When the mean checkpoint timeout is 300 ms, checkpointing is initiated while the data messages are being exchanged. The initial deficit counts are lower as messages that have already been received are not included in the deficit. It can be observed that the optimized algorithms terminated in fewer number of rounds. Actually, the optimized algorithms always terminated in the first round reset that occurs after all the processes receive all white messages destined for them.

8. OTHER APPLICATIONS

One of the key ingredients in the tree-based and the centralized algorithm is an efficient solution of a problem that we call *distributed message counting* problem. The solution of this problem also has applications in implementation of synchronizers [1]. A synchronizer is a layer of software that allows simulation of a synchronous network on asynchronous networks. The mechanism gives each process a logical abstraction of a pulse. A process can start the next pulse if all messages that have been sent to it in the last pulse have been received.

An implementation of a synchronizer using distributed message counting is as follows. In each pulse, the root computes the total deficit using broadcast and convergecast on the tree. It then uses distributed message counting to detect when all messages in the current pulse have been received. Finally, a new pulse is generated using broadcast.

Assuming a perfect binary tree, the β synchronizer uses $O(W + N)$ messages. Our algorithm uses $O(N \log N \log(W/N))$ messages which is is significantly smaller than $O(W + N)$ when $W >> N$.

9. CONCLUSIONS

We have propose three scalable algorithms for global snapshot: a grid-based, a tree-based and a centralized algorithm. We have also implemented our algorithms on top of the MPI library on the Blue Gene/L supercomputer. Our experiments confirm that the proposed algorithms significantly reduce the message and space complexity of a global snapshot.

10. REFERENCES

[1] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, Oct. 1985.

[2] L. Bouge. Repeated snapshots in distributed systems with synchronous communication and their implementation in CSP. *Theoretical Computer Science*, 49:145–169, 1987.

[3] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

[4] K. M. Chandy and J. Misra. How processes learn. In R. Strong, editor, *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–214, Minaki, ON, Canada, Aug. 1985.

[5] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(4):1–4, Aug. 1980.

[6] V. K. Garg. *Concurrent and Distributed Computing in Java*. Wiley & Sons, 2004.

[7] A. D. Kshemkalyani, M. Raynal, and M. Singhal. An introduction to snapshot algorithms in distributed computing. *Distributed Systems Engineering*, 2(4):224–233, Dec. 1995.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[9] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, pages 423–434, Aug. 1993.

[10] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In *SC'2004 Conference CD*, Pittsburgh, PA, Nov. 2004.

[11] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proc. of the 6th International Conference on Distributed Computing Systems*, pages 382–388, 1986.

		Grid			Tree	Centralized		
N	Initial Deficit	Latency	Max Msgs	Time/ Msg	Latency	Latency	Max Msgs	Time/ Msg
32	2880992	104	23	4.52	6192	5053	1548	3.26
64	5764032	157	31	5.06	11232	9108	3109	2.93
128	11536256	190	48	3.96	19191	15765	5738	2.75
256	23105280	293	64	4.58	36921	34641	12290	2.82
512	46341632	572	96	5.96	50578	69574	24506	2.84

Table 1: Total latencies (microseconds) for the checkpointing algorithms on different number of nodes

	Grid				Tree			Centralized		
N	Layout	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
32	4 ×8	4	12	5.88	34	485	149.41	29	576	80.16
64	8 ×8	8	16	9.80	37	813	193.25	28	991	78.03
128	8 ×16	8	24	10.09	38	1884	252.68	28	1992	80.72
256	16 ×16	16	32	17.94	35	3603	269.91	27	3894	81.91
512	16 ×32	16	48	18.13	31	3434	190.36	23	6557	69.66

Table 2: Sent Message Counts for the checkpointing algorithms on different number of nodes

	Grid				Tree			Centralized		
N	Layout	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
32	4 ×8	1	12	5.88	48	429	149.41	44	592	80.16
64	8 ×8	8	15	9.80	51	799	193.25	43	1012	78.03
128	8 ×16	1	24	10.09	53	2120	252.68	42	2100	80.72
256	16 ×16	16	32	17.94	49	4053	269.91	42	4085	81.91
512	16 ×32	1	48	18.13	43	3314	190.36	35	7193	69.66

Table 3: Receive Message Counts for the checkpointing algorithms on different number of nodes

	Mean Checkpoint Timeout=300 ms		Mean Checkpoint Timeout=1 s	
Algorithm	Initial Deficit	No. of Rounds	Initial Deficit	No. of Rounds
Tree	30318186	13	46341632	16
Tree (Opt)	20469063	5	0	1
Centralized	30203345	13	46341632	9
Centralized (Opt)	20467177	7	0	1

Table 4: Initial Deficit Counts and number of rounds taken for unoptimized and optimized versions of tree and centralized algorithms