An Efficient Checkpointing Method for Multicomputers with Wormhole Routing

Kai Li* Jeffrey F. Naughton† James S. Plank*

In *International Journal of Parallel Programming*, Vol. 20, No. 3, June, 1992, pp 159-180

Abstract

Efficient checkpointing and resumption of multicomputer applications is essential if multicomputers are to support time-sharing and the automatic resumption of jobs after a system failure. We present a checkpointing scheme that is transparent, imposes overhead only during checkpoints, requires minimal message logging, and allows for quick resumption of execution from a checkpointed image. Furthermore, the checkpointing algorithm allows each processor p to continue running the application being checkpointed except during the time that p is actively taking a local snapshot, and requires no global stop or freeze of the multicomputer. Since checkpointing multicomputer applications poses requirements different from those posed by checkpointing general distributed systems, existing distributed checkpointing schemes are inadequate for multicomputer checkpointing. Our checkpointing scheme makes use of special properties of wormhole routing networks to satisfy this new set of requirements.

Keywords: Checkpointing, fault-tolerance, multicomputers, wormhole routing.

1 Introduction

Recently, multicomputer systems[1] with large numbers of fast processors, high-speed interconnects, and concurrent I/O systems have become commercially available. While the raw hardware on these machines is powerful, multicomputers are still more difficult to use than uniprocessors and shared memory multiprocessors. Two important factors making multicomputers difficult or at least inconvenient to use are

*Department of Computer Science, Princeton University, Princeton, NJ 08544.

†Department of Computer Science, University of Wisconsin at Madison, Madison, WI 53706

[1] As defined by Athas and Seitz, a multicomputer is an ensemble of "small programmable computers, called nodes, connected by a message-passing network [2]."

- There is no facility for automatic resumption from failure points for long-running jobs. Because there is no resumption facility, if the multicomputer has to be taken down, perhaps because of a failure, perhaps for routine maintenance, all currently active jobs must be terminated. After the system is brought back up, these terminated jobs must be restarted from the beginning. This is a serious drawback, since it is precisely the long-running jobs that require the resources of a multicomputer.

- There is no way to start a job, then suspend it because a higher priority job has arrived, then resume it after the higher priority job has terminated. This makes sharing a multicomputer difficult. Once a job has been allocated the multicomputer, or some portion of the multicomputer (e.g., a subcube in a hypercube), the job must run to completion or be aborted.

To remove these difficulties requires a checkpoint/resume capability.

Since a multicomputer can be viewed as a distributed system, schemes presented in the distributed checkpointing literature can be used for multicomputer checkpointing. However, multicomputers present a set of opportunities and requirements significantly different from those in general distributed systems. The special opportunities arise because multicomputers have a lower degree of node autonomy than do distributed systems, and because existing multicomputer interconnection networks use deterministic, deadlock-free routing algorithms. The predominant special requirement of multicomputer checkpointing is that both checkpointing and recovery must be fast. In fact, it is primarily this requirement that renders previous distributed system checkpointing algorithms insufficient for multicomputer checkpointing.

Previous distributed checkpointing schemes fall into four broad categories: distributed database checkpointing schemes, pessimistic checkpointing schemes, optimistic checkpointing schemes, and Chandy-Lamport algorithms. The problem of taking distributed database checkpoints has been well-studied. However, distributed database checkpointing schemes, such as those presented in [11, 18, 17], require that the checkpointed computation consist of a sequence of transactions. Since multicomputer applications are general purpose parallel programs, no such computational model can be assumed.

Pessimistic schemes, such as those presented in [3, 20], require that every message be logged synchronously as it is sent or received. The advantage of pessimistic checkpointing is quick recovery — the synchronous logging of messages makes recovery from a checkpointed state trivial. However, the overhead of synchronously logging every message makes pessimistic schemes inappropriate for multicomputer application checkpointing. (The name "pessimistic" comes from the fact that

these schemes optimize recovery at the expense of normal execution, implicitly assuming frequent failures.)

Optimistic schemes, such as those presented in [12, 24], allow asynchronous logging of messages, and hence can be implemented to cause less overhead than the pessimistic schemes during normal operation. The name "optimistic" comes from the fact that these schemes optimize normal operation at the expense of recovery, implicitly assuming few failures. One of the strengths of the optimistic schemes is that they allow a large degree of autonomy among the nodes participating in the checkpointing. In particular, each node can choose to checkpoint its internal state and buffered messages at any time, independent of the other nodes. This autonomy does not come for free — it is a source of much of the complexity in the recovery algorithm. The optimistic schemes require rollback and need to replay their logs in order to discover and restore a consistent state. Rolling back and replaying can be time-consuming so that existing optimistic recovery methods are not appropriate for fast job swapping on multicomputers.

The fourth class of checkpointing schemes [23, 14, 13, 1, 5] give up this node autonomy and attempt to strike a balance between the low overhead of optimistic schemes and the simple recovery of pessimistic schemes. In all of these algorithms, (which we call *CL* algorithms as they stem from a paper by Chandy and Lamport [4]), there are identifiable global snapshots, and each processor knows when a global snapshot begins and ends. This allows for quick recovery from checkpointed images, since there is no need for a complex distributed algorithm to determine from which checkpointed state each processor should resume.

The CL algorithms view a distributed system as a set of processes with FIFO channels for communication. While their method seems to be the most natural for multicomputer checkpointing, it fails for the following reason: For the algorithm to work correctly, each processor must send a special marker message to all other processors to which it has a channel. If the network is fully connected, this means $O(n^2)$ messages.

While the processors in current multicomputers are not physically fully connected, the "wormhole" or "virtual cut-through" routing networks used in most multicomputers enforce the illusion of full connectivity at the processor level. That is, if a message from processor p_1 to processor p_2 passes through the router at the node for processor p_3, there is no way for any of p_1, p_2, or p_3 to detect this fact. As far as the three processors are concerned, the message was sent over a direct link between p_1 and p_2.

This virtual full connectivity means that an implementation of the Chandy and Lamport checkpointing algorithm on a multicomputer will have message complexity $O(n^2)$, which will render the

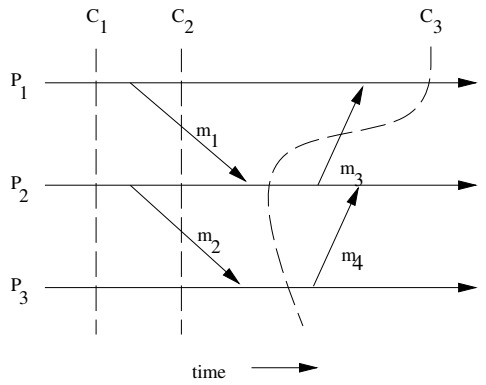

Figure 1: A sample distributed system.

scheme unacceptably slow when n scales to large machines. An improvement to the Chandy and Lamport technique has been suggested in [25], but it degrades to $O(n^2)$ messages as the interval between checkpoints grows larger.

A main contribution of this paper is a new checkpointing algorithm, which stems from the CL algorithms, but is also based on the deadlock-free routing algorithms employed by multicomputers. We refer to this algorithm as the *Network Sweeping algorithm*, since it conceptually "sweeps" through the interconnection network, enabling processes to checkpoint without sending messages to all other processes. The Network Sweeping algorithm requires far fewer than $O(n^2)$ messages — $O(n \log n)$ for hypercube-connected multicomputers such as the Intel iPSC/2 and iPSC/860, and $O(n)$ for mesh-connected multicomputers. These bounds hold in spite of the virtual full connectivity provided by the multicomputer interconnection networks.

2 Checkpointing Nomenclature

The standard way of visualizing a distributed group of processes is to use horizontal lines to denote the relative progress of the processes over time. A message between processes is denoted by an arrow from the point at which the sending process sends the message to the point at which the receiving process receives it. A *cut* is represented by a dotted line that crosses each process's time line exactly once. A message is said to cross the cut line from left to right if its sending point is to the left of the cut line, and its receiving point is to the right of the cut line. Crossing from right to left is defined similarly. A *consistent cut* is a cut in which no messages cross the cut from right to left. Other cuts are called *inconsistent*.

Figure 1 shows a distributed system with three processes, four messages, and three cuts. The

first two cuts (C_1 and C_2) are consistent as no messages cross them from right to left. The last (C_3) is inconsistent due to message m_3.

Checkpoints are related to cuts as follows: A distributed checkpoint consists of *process state* and *message state*. The process state is created from individual *local checkpoints* of each process, and the message state is a *log* of messages from one process to another. Given a cut C, one can construct a distributed checkpoint by having each process p take a local checkpoint at the point where the C's cut line crosses p's time line, and by logging each message that crosses the cut line from left to right. A distributed checkpoint is restarted by starting each process from its local checkpoint, and then resending the messages in the log.

It is well-known [13, 5] that checkpoints can only be made from consistent cuts. Inconsistent cuts lead to the recording of spurious messages in the process state of the checkpoint. Most checkpointing algorithms concentrate on finding consistent cuts for the taking of local checkpoints, and on discovering which messages comprise the message log.

A series of checkpoints divides the execution of the checkpointed program into distinct intervals that can be visualized as being the spaces between the checkpoints' cuts. Further, checkpoints divide all messages into two types: *Cross-cut* messages, which start in one checkpointing interval, and end in another, and *non-cross-cut* messages, which are contained in a single checkpoint interval. For example, messages m_1, m_2 and m_3 in Figure 1 are cross-cut messages; m_4 is a non-cross-cut message.

Finally, we make the assumption that there is one process per processor in the multicomputer. This makes it possible to ignore the distinction between processor and process, so for clarity of exposition we refer to messages passing between processors rather than processes.

3 The Chandy-Lamport Checkpointing Algorithm

The original Chandy-Lamport checkpointing algorithm has all processors periodically define a cut and make a distributed checkpoint. Its main assumption is that the channels between processors enforce a FIFO ordering among messages. The algorithm works as follows: Some processor p_0 starts the checkpoint by taking its local checkpoint, and then broadcasting a special *marker* message m_{CL} to all processors to which it has a channel. All processors p then perform the following:

- If p receives m_{CL} and hasn't taken its local checkpoint yet, then it takes its local checkpoint, and broadcasts m_{CL} to all processors to which it has a channel.

- Afterwards, if p receives a non-marker message m along a channel c, and it has not yet received m_{CL} on c, then m is a cross-cut message, and must be logged.
- When all processors have received m_{CL} on all their incoming channels, then there are no more cross-cut messages to be logged: The checkpoint is finished.

This algorithm performs three major functions: First, it ensures that the local checkpoints will define a consistent cut, as any message that would cross the cut from right to left would have to bypass an m_{CL} message, which is illegal given the assumption of FIFO channels. Second, it correctly identifies cross-cut messages: These are the ones received by a processor p on channel c after p has checkpointed, but before p has received m_{CL} on c. Third, it provides notification to each p that p will receive no more cross-cut messages: This is when p has received m_{CL} on every incoming channel.

As stated before, the problem with this algorithm when applied to multicomputers with wormhole routing is that at the process level, the processors are fully connected. (In the preceding terminology, every processor p has a channel to every other processor.) Thus, every processor must send m_{CL} to every other processor, leading to $O(n^2)$ messages on multicomputers with n processors. Of all the variants of the Chandy-Lamport algorithm, only [25] attempts to reduce this number of messages. It optimizes the CL algorithm by having each processor remember to which processors it has sent messages in the previous checkpoint interval. Then it only sends m_{CL} to those processors. (The algorithm uses additional acknowledgement and synchronization messages to ensure correctness without sending the full n^2 messages.) This is indeed an optimization, however its worst case is still $O(n^2)$ messages, and it depends upon how many pairs of processors exchange messages between cuts. Unless the patterns of communication in the application program are very restrictive, as the size of checkpoint intervals grows, the number of m_{CL} messages that a processor must send will also grow.

4 The Network Sweeping Algorithm

In this section, we present our checkpointing algorithm that greatly reduces the number of marker messages that a processor must send. We call this the *Network Sweeping* algorithm as it uses properties of wormhole routing to "sweep" the interconnection network of a multicomputer, and checkpoint in less than $O(n^2)$ messages. To state a bound on the number of messages sent by the checkpointing algorithm, we define two functions:

$$l(n) \quad = \quad \text{the number of physical communication links in the multicomputer}$$

$pp(n)$ = the number of distinct processor pairs (p_i, p_j) such that p_i sends p_j an application program message while the checkpoint is in progress.

On a hypercube like the Intel iPSC/2 or iPSC/860, $l(n)$ is $O(n \log n)$; on the mesh-connected multicomputers (such as the Intel Paragon), $l(n)$ is $O(n)$. The function $pp(n)$ depends upon the behavior of the application program. However, it is critical to note that $pp(n)$ depends on the number of pairs of processors that communicate while the checkpoint *is actually in progress* (i.e. during the time interval bounded by the first processor beginning its local checkpoint and last processor learning that the checkpoint is over), rather than the number of pairs of processors that communicate between cuts as in [25]. As a side note, in [25], the authors prove that their scheme achieves the theoretical lower bound on the number of marker messages that must be exchanged to take a checkpoint. While our scheme uses fewer messages than that lower bound, this is not a contradiction, since our scheme makes stronger assumptions about the interconnection network and routing scheme than those used in the proof in [25].

As stated before, the Chandy-Lamport algorithm achieves three functions (determining a consistent cut, identifying cross-cut messages, and guaranteeing that there are no more cross-cut messages) with just one type of marker message (m_{CL}). Our algorithm differs in that it uses three types of markers:

1. The first type (which we still call m_{CL}) is like the m_{CL} of the CL algorithms as it is used to define a consistent cut and help define the cross-cut messages. It differs, however, as it need not be sent to all neighboring processors, and as it contains a one-bit sequence number. The m_{CL} messages can be sent between arbitrary pairs of processors in the multicomputer.

2. The second type of marker message (which we call m_{NS}, for "Network Sweeping") is used by a processor to determine when there are no more cross-cut messages for the processor to log. The m_{NS} messages are sent between pairs of processors connected by a physical link when it has been determined that no more cross-cut messages will be sent along that link. (We consider links to be unidirectional. We view bidirectional links as two unidirectional links). We discuss how to make this determination below in section 5. Until then, the above definition suffices for our explanation.

3. The third type of message (which we call m_S for "Stop Message") is used by all processors to determine when the checkpoint is over.

The combination of the three types of messages achieves the same three functions as the CL algorithms above, using a smaller overall number of markers, yet still allowing the individual processors

to perform useful work while the checkpoint is being taken.

Our algorithm works as follows: To oversee the taking of the checkpoint, we assign one processor p_c to be the *coordinating* processor. This processor is responsible for starting the checkpoint, and for knowing when the checkpoint is over. In order to initiate global checkpoint i, p_c first takes a local checkpoint, then broadcasts m_{CL} with sequence bit (i mod 2) to all other processors. All processors p then perform the following:

- If p receives m_{CL} or m_{NS} and hasn't taken its local checkpoint yet, then p takes its local checkpoint.
- After taking its local checkpoint, p may start sending m_{NS} messages. As stated above, it only sends m_{NS} to p' if p and p' are connected by a physical link. It does so when it is sure that no more cross-cut messages can be sent along that link. When p has sent m_{NS} on all its outgoing links, and has received m_{NS} on all its incoming links, then we say that p has *finished sweeping.*
- After taking its local checkpoint and until the checkpoint is over, if p sends a non-marker message to p' and p has not sent m_{CL} to p', then it must send m_{CL} with a sequence bit of (i mod 2) to p' before sending m.
- After taking its local checkpoint and until it is finished sweeping, if p receives a non-marker message m from processor p', and p has not yet received m_{CL} from p', then m is a cross-cut message: P logs m.
- When p has finished sweeping, then it knows that it will receive no more cross-cut messages. Therefore, p commits all of its checkpointing information (its local checkpoint and message log) to disk. Afterwards, it notifies p_c by sending it m_S.
- When p_c receives m_S from all processors, checkpoint i is over. It broadcasts m_S back to all processors to let them know. After sending m_S, p_c is free to start the next checkpoint.
- Before p finishes sweeping, if p receives a m_{CL} message with a sequence bit of $((i+1) \bmod 2)$, then it ignores the m_{CL}. Similarly, after it finishes sweeping p should ignore any m_{CL} with a sequence bit of (i mod 2).
- When p receives m_S from p_c, then it knows that checkpoint i is over. It stops sending m_{CL} messages. [2]

Figure 2: A simple multicomputer.

To give a bit of intuition about how this algorithm works, we present an example in which the multicomputer of Figure 2 takes a checkpoint. Although this multicomputer is extremely simple, it is sufficient to illustrate our points.

Example 1 In this example, we assume that all processors have finished taking checkpoint 0. We show how they take checkpoint 1, where there is one cross-cut message m and one non-cross-cut message m'. The events are shown graphically in Figure 3. We detail them below:

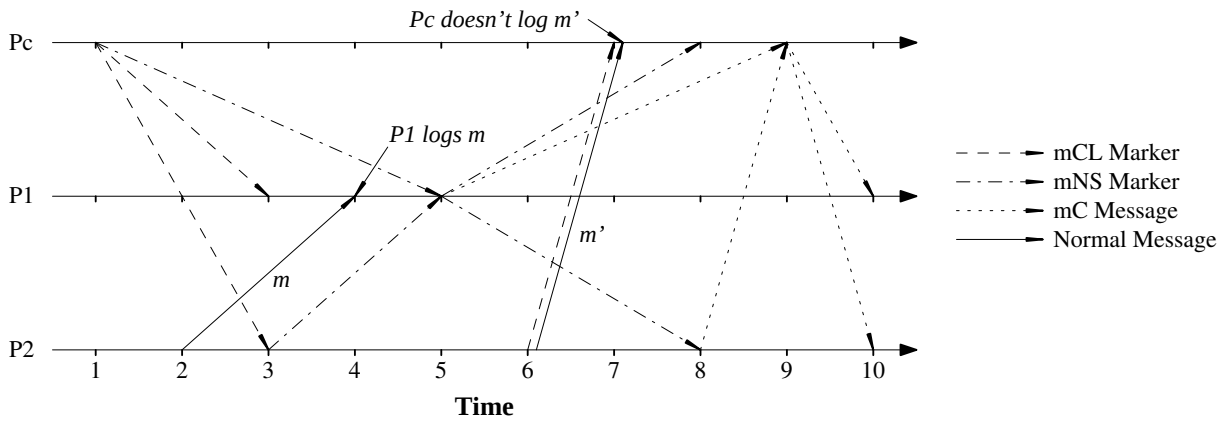

Figure 3: Time line of Example 1.

1. At time 1, p_c decides that it wants to start checkpoint 1. Thus it checkpoints itself, and broadcasts m_{CL} with a sequence bit of 1 to the other two processors. After it checkpoints itself, p_c can't send any more cross-cut messages. Therefore, its link to p_1 won't receive any more cross-cut messages: p_c sends m_{NS} to p_1.

2. At time 2, p_2 has not yet received m_{CL}. Its application program sends the message m to p_1.

3. At time 3, p_1 and p_2 both receive m_{CL} from p_c. Thus they checkpoint themselves. Now, since it can't send any more cross-cut messages, p_2 knows that its link to p_1 will not receive any more cross-cut messages: It sends m_{NS} to p_1.

[2]One might note that it's possible for p to receive a m_{CL} or m_{NS} message from checkpoint $i+1$ before receiving m_S from checkpoint i. If this happens, p can simply use the m_{CL} or m_{NS} to flag both the end of checkpoint i and the beginning of checkpoint $i+1$. It then ignores the m_S when it arrives. (There can be no confusion between the m_S from checkpoint i and the one from checkpoint $i+1$: The former m_S must be received by p before p finishes sweeping for checkpoint $i+1$. The latter must be received afterwards).

4. At time 4, p_1 receives m from p_2. Since it hasn't received m_{CL} from p_1, it knows that m is a cross-cut message: p_1 logs m.

5. At time 5, the two m_{NS} messages bound for p_1 arrive. Since p_1 has checkpointed, and since no cross-cut messages will come through its incoming links, p_1 knows that no cross-cut messages will be sent through its outgoing links. Therefore, p_1 sends m_{NS} out both of those links. Afterwards, it is finished sweeping. It notifies p_c by sending it m_S.

6. Next, at time 6, p_2's application program has resumed, and wants to send the message m' to p_c. It must precede m' with m_{CL}.

7. At time 7, p_c receives the m_{CL} and m' from p_2. Because of the m_{CL} marker, p_c knows that m' is not a cross-cut message. Therefore p_c doesn't log m'.

8. At time 8, p_c and p_2 both receive the m_{NS} messages from p_1. Thus they are finished sweeping. P_2 notifies p_c of this by sending m_S to p_c.

9. At time 9, p_c receives the m_S messages from p_1 and p_2. Checkpoint 1 is over. P_c then broadcasts m_S back to p_1 and p_2 to let them know.

10. At time 10, p_1 and p_2 receive m_S from p_c. Thus they know that checkpoint 1 is over. After this point, they stop sending m_{CL} messages.

Thus, the checkpoint is correctly taken, and the one cross-cut message is successfully logged. Note that the processors' decisions about when to send m_{NS} messages depend on the underlying interconnection network. We formalize how the processors make these decisions in section 5 below. □

The total number of messages that this algorithm uses is as follows:

$O(n)$	for the initial broadcast of m_{CL} messages by p_c
$pp(n)$	for the other m_{CL} messages
$O(l(n))$	for the m_{NS} messages
$O(n)$	for the m_S messages

Since $l(n)$ must be at least $n-1$ for the multicomputer to be connected, the total number of messages can be simplified to $O(l(n)+pp(n))$. We later describe an optimization that gets rid of the $pp(n)$ term, and thus checkpoints with only $O(l(n))$ extra messages.

We provide the following lemmas to prove the algorithm's correctness, and to further illustrate some of the details of the algorithm.

Lemma 1 *Our Network Sweeping algorithm defines a consistent cut.*

Proof: The only way this can be false is if there is a message m crossing the checkpoint's cut from right to left. In other words, it is sent by a processor p after p checkpoints, and received by p' before p' checkpoints. Now, assume that p has checkpointed itself, but that p' has not. Since p is in the interval between checkpointing and receiving m_S from p_c (p_c can't send m_S until after p' checkpoints) p must precede m with m_{CL}. When p' receives m_{CL}, it checkpoints itself. Therefore, when it receives m, it will have checkpointed itself: m cannot cross the cut from right to left, and the cut is consistent. □

This next lemma is to provide justification for the sequence bit in the m_{CL} messages. One might notice that a processor p can receive m_{CL} messages well after receiving its final m_S message from p_c. This lemma shows that these m_{CL} messages are properly ignored, even if another checkpoint is started soon after the current one finishes.

Lemma 2 *No message can cross more than one cut.*

Proof: Suppose processor p sends the message m to p', and m crosses more than one cut. Since all cuts are consistent, this means that p sends m in checkpoint interval i, and p' receives m in interval $i+j$ where $j > 1$. This also means that p' receives it along a link *after* receiving m_{NS} for checkpoint $i+1$ on that link. This is a contradiction from the definition of m_{NS} messages. Therefore it is not possible for such an m to exist. □

The way this lemma ties in with the sequence bit is as follows: Suppose that processor p has just taken its local checkpoint for global checkpoint i but p hasn't finished sweeping. Then p will ignore m_{CL} messages with sequence bits of $((i+1) \bmod 2)$. Since no message (including markers) can cross more than one cut, the m_{CL} messages that p doesn't ignore are those from checkpoint interval i. These are exactly the ones that p needs to decide whether or not to log a non-marker message.

Suppose now that p has finished sweeping for checkpoint i, but hasn't started checkpoint $i+1$. The only m_{CL} messages that p cares about are those from interval $i+1$. Since p ignores m_{CL} markers with bits of $(i \bmod 2)$ and since no message can cross more than one cut, the m_{CL} marker that p doesn't ignore does indeed come from checkpointing interval $i+1$.

The next two lemmas show that the Network Sweeping algorithm correctly logs messages.

Lemma 3 *All cross-cut messages are logged.*

Proof: For a message m to be a cross-cut message, it must be sent by p before p starts checkpoint i, and received by p' after p' starts checkpoint i. By the definition of the m_{NS} messages, m must be received by p' before p' finishes sweeping. Thus, p' will log m if and only if it doesn't receive a m_{CL} message from p prior to receiving m. As stated above, the only m_{CL} messages that aren't ignored are those from checkpoint i. Since p sends m before starting checkpoint i, p' won't receive any such m_{CL} message before receiving m, and thus it will recognize m as a cross-cut message. M will be properly logged. □

Lemma 4 *No non-cross-cut messages are logged.*

Proof: Non-cross-cut messages come in two varieties: Those sent by p after p starts checkpointing and before p stops checkpointing, and those sent by p after p stops the current checkpoint and before p starts the next checkpoint. In either case, the messages are not logged. If p sends m before receiving m_S from p_c, then m must be preceded by an m_{CL} message. If m is received by p' before it is finished sweeping, then p' knows from the receipt of the m_{CL} marker that m is a non-cross-cut message. If it receives m after it is finished sweeping, then the m_{CL} message doesn't matter: p' already knows that m is a non-cross-cut message. In either case, p' doesn't log m. If p sends m after receiving m_S from p_c, then it is guaranteed that p' has finished sweeping: Again, p' will recognize m as a non-cross-cut message, and will not log it. □

Therefore, these lemmas show that our Network Sweeping algorithm is correct: It defines a consistent cut, and correctly identifies and logs the cross-cut messages. Moreover, the lemmas provide some justification for various parts of the algorithm: The sequence bit, the m_{NS} markers and the m_S messages are all necessary for the proofs.

5 Sending the m_{NS} Messages on Wormhole Routing Networks

Above we state that a processor p sends a m_{NS} marker along a link when it determines that no more cross-cut messages will be sent along that link. However, we don't state specifically how p makes this determination. The example in Figure 2 was simple enough to provide an *ad hoc* solution, but here we consider the general case.

We are aided by the fact that while the interconnects on wormhole-routing multicomputers provide the illusion of full connectivity, in reality each of the $O(n^2)$ paths between processors corresponds to a route through some subset of the physical links of the system. Furthermore, with the exception of the original store-and-forward hypercube multicomputers, all of the multicomputer interconnection networks of which we are aware satisfy the following three conditions:

1. Routing in the system is deterministic, using a deadlock-free routing algorithm as defined by Dally and Seitz [9].

2. Allocation of links along a route is non-preemptive; that is, once a message is allocated a link along its route, no other message travels along the same link until the message is on the next link in the route (or until the message is at the destination processor).

3. Queueing for incoming messages in the system is strictly FIFO.

We call an interconnection network that satisfies these conditions a *well-behaved* interconnection network. Many existing and proposed multicomputers are well-behaved. Among these are Intel's iPSC/2, iPSC/860, Delta, Sigma and Paragon machines, Stanford's DASH [15], and MIT's J-Machine [8]. The first two of these use hypercube interconnections, while the last four are two and three-dimensional meshes.

For well-behaved interconnection networks, we present a method for sending the m_{NS} messages. We prove that the method is guaranteed to start and terminate, and that it does what it promises: When m_{NS} is received on a link, no more cross-cut messages will be received on that link. Afterwards, we display how the method works for hypercubes and two-dimensional, toroidal meshes. Since a constant number of m_{NS} messages are sent per physical link of the interconnection network, the total number of m_{NS} messages is $O(l(n))$.

Method for General Well-behaved Interconnection Networks

Dally and Seitz [9] define an interconnection network I to be a strongly connected, directed graph $I = G(N, C)$, where N is the set of nodes, and C is the set of channels, or physical links that connect the nodes. A routing function $f : C \times N \rightarrow C$ maps the current channel c_i upon which a message resides, and its destination node n_d, to the next channel c_j that it will take on its way to n_d. Thus, f is a non-adaptive, memoryless routing function. It's non-adaptive because it always yields the same path from one node to another, and it's memoryless because the function works from channel to channel, retaining no knowledge of the source node once it's underway.

To prove a routing function deadlock-free, they define a *channel dependency graph* $D = G(C, E)$, where C is the same set of channels described above, and E is a set of edges determined by f as follows:

$$E = \{(c_i, c_j) \mid f(c_i, n_d) = c_j \text{ for some } n_d \in N\}$$

They show that f is deadlock-free if and only if D has no cycles.

We use the results of Dally and Seitz to define how to send the m_{NS} marker messages. It is assumed that each processor p_i resides on node n_i. Moreover, if the interconnection network contains the channel $c = (n_i, n_j)$, then the path of a message from p_i to p_j is routed through c. Finally, when we say "p_i sends m_{NS} on channel c," we mean that if $c = (n_i, n_j)$, then p_i sends m_{NS} to p_j, thereby using channel c. The method is quite simple:

- Assume that p_i has checkpointed itself.
- For all outgoing channels c_o from p_i, if there is no c_j such that $(c_j, c_o) \in E$, then p_i sends the marker m_{NS} on channel c_o.
- For all other outgoing channels c_o from p_i, when m_{NS} markers have been received from all channels c_j such that $(c_j, c_o) \in E$, p_i sends m_{NS} on channel c_o.

The following lemmas prove the method's correctness:

Lemma 5 *Until m_{NS} messages have been sent on all channels, there is always some processor that may send a m_{NS} message.*

Proof: Since D is a graph with no cycles, we can number the channels from 1 to $|C|$ in such a way that if $(c_i, c_j) \in E$, then $i < j$. Now, since there is no c_i such that $(c_i, c_1) \in E$, a m_{NS} marker can instantly be sent on c_1. The proof proceeds by induction. Suppose that m_{NS} has been sent on channels c_1 through c_k, but not on channel c_{k+1}. Consider all c_i such that $(c_i, c_{k+1}) \in E$. From our numbering, $i < k + 1$, so m_{NS} has been sent out all channels c_i. As f is deadlock-free, the markers are eventually received, and m_{NS} can be sent out c_{k+1}. Therefore, there is always a channel that is ready to receive a m_{NS} marker. □

Lemma 6 *When m_{NS} is received on a channel c, it is guaranteed that no more cross-cut messages will be sent or received along c.*

Proof: Suppose p sends a cross-cut message m to p'. That is, m is sent before p has taken its local checkpoint, and received after p' has taken its local checkpoint. Further, let $c_1, \ldots, c_l$ be the route that m takes; in other words, $f(c_{k-1}, n_d) = c_k$ for $1 < k \leq l$. We show that m always travels along a channel c_i before m_{NS} can be sent on c_i.

Since allocation of a route is non-preemptive, and channels are FIFO, m will be put on channel c_1 before p can send out another message on channel c_1. In particular, m is put on c_1 before p can put m_{NS} on c_1.

We now show that m is put on all channels $c_1, \ldots, c_l$ before any processor can put m_{NS} there. This is done by induction. First, as noted above, m is put on c_1 before m_{NS} can be put there. Next, suppose that for all $k < l$, m travels through c_k before a m_{NS} message can be put there. Before m_{NS} can be put there, m must be able to allocate c_{k+1}. Therefore m is put on channel c_{k+1} before m_{NS} is put on channel c_k. Now, since $f(c_k, n_d) = c_{k+1}$, $(c_k, c_{k+1}) \in E$, and m_{NS} will not be sent on c_{k+1} until after it has been received on c_k. As stated above, m_{NS} won't be received on c_k until after m has been put on c_{k+1}. Thus, m_{NS} won't be put on c_{k+1} until after m has been put there.

Therefore it is impossible for a cross-cut message to travel on a channel c after m_{NS} has been put there. Thus, when m_{NS} is received along a channel, it is guaranteed that no more cross-cut messages will be use that channel. □

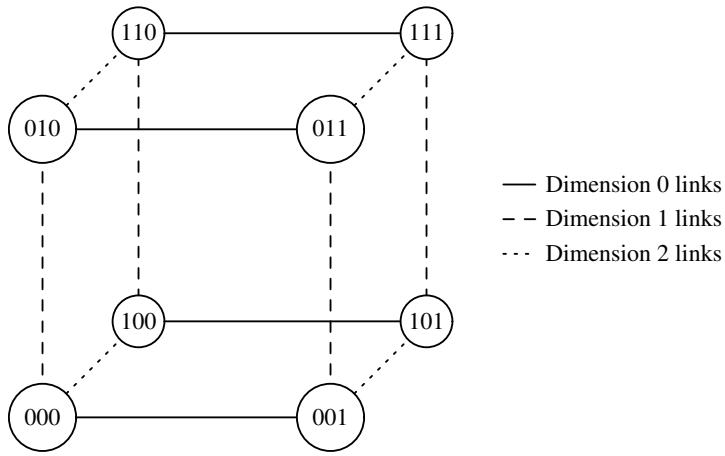

Figure 4: A 3-dimensional hypercube

Hypercube Interconnections

The Intel iPSC/2 and iPSC/860 [19] are multicomputers with hypercube interconnection networks that are well-behaved. Each process is associated with a node on a hypercube. In a d-dimensional hypercube, nodes are numbered 0 through $2^d - 1$. If the binary representation of nodes i and j differ by exactly their k-th bit, then they are physically connected by a link that is said to be in dimension k. As an example, Figure 4 shows a 3-dimensional hypercube with its links.

Each link is bidirectional, and thus can be divided into two unidirectional channels. The iPSC/2 and iPSC/860 both use the *e-cube* routing function, which is deadlock-free. It specifies that the only channels used are those in the dimensions in that the binary representation of the source and destination nodes differ. These channels are used in increasing order of dimension. For example in

Figure 4, a message from node 000 to node 111 will pass through the channels $(000, 001), (001, 011)$, then $(011, 111)$.

To put this in the notation of Dally and Seitz, let $c_i = (n_a, n_b)$. Then $f(c_i, n_d) = c_o$, where c_o is the channel emanating from n_b in the lowest dimension in that the binary representations of n_b and n_d differ. For example, in Figure 4, $f((000, 001), 111) = (001, 011)$.

The method for sending m_{NS} markers therefore works as follows:

- After processor p takes its local checkpoint, it sends m_{NS} on its outgoing channel c_0 of dimension 0. This is because c_0 always starts a message route: There are no c_j such that $(c_j, c_0) \in E$.
- P sends m_{NS} on its outgoing channel c_k of dimension k only after it has received m_{NS} on each incoming channel c_j of dimension $j < k$. This is because for each of those c_j, $(c_j, c_k) \in E$.

This method results in two m_{NS} markers sent per physical link. As each node has $\log n$ links incident to it, the total number of m_{NS} messages is $n \log n$, which is $O(l(n))$ for the hypercube.

Mesh Interconnections

As a second example for sending m_{NS} messages, we consider a toroidal mesh. The next generation multicomputers, including the Intel Paragon use toroidal mesh interconnection networks. In these interconnection networks, the nodes are laid out on a grid, and each node has two outgoing links: one to its right, and one going up. To be more formal, in a $R \times C$ toroidal mesh, $I = G(N, L)$, where:

$$\begin{aligned} N &= \{n_{ij} \mid i < R \text{ and } j < C\} \\ L &= \{(n_{ij}, n_{ij'}) \mid j' = (j+1) \bmod C\} \\ &\quad \cup \{(n_{ij}, n_{i'j}) \mid i' = (i+1) \bmod R\} \end{aligned}$$

As before, for the purposes of the deadlock-free routing function, we split each link into two channels. This time we call them *high* and *low* channels, for reasons discussed below. To be more clear, we give each channel a 4-digit identification number "*dhrc*," where

d = channel's dimension: 1 if it is of the form $(n_{ij}, n_{i'j})$, 0 otherwise.

h = 1 if it's a high channel, 0 if it's a low channel.

r = row number of its source node.

c = column number of its source node.

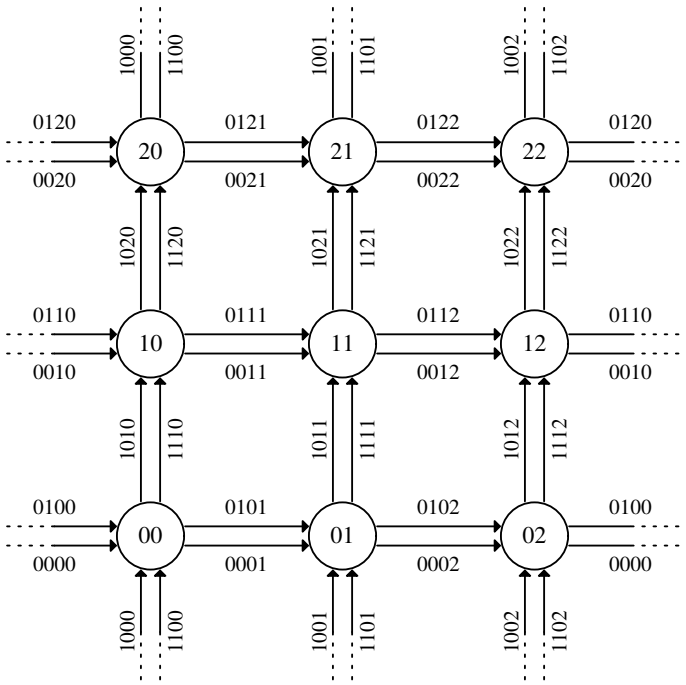

Figure 5: A 3×3 toroidal mesh

Figure 5 shows an example of a 3×3 toroidal mesh.

Deadlock-free routing on the mesh is similar to the routing on the hypercube. The message first gets routed to the correct column, and then to the correct row. When routing to the correct column, low channels are used when the number of the current column is greater than the number of the desired column, and when routing to the correct row, low channels are used when the number of the current row is greater than the number of the desired row. We formally define f in terms of the current node n_{rc} and the destination node n_{ij} as follows:

$$f(n_{rc}, n_{ij}) = \begin{cases} c_{00rc} \text{ if } \ c > j \\ c_{01rc} \text{ if } \ c < j \\ c_{10rc} \text{ if } c = j \text{ and } r > j \\ c_{11rc} \text{ if } c = j \text{ and } r < j \end{cases}$$

The definition of f in terms of the current channel and destination node is slightly more complex, but follows directly. We omit it for the sake of clarity.

The first thing to notice about this function is that channels with identification numbers of the form c_{00r0}, $c_{01r(C-1)}$, c_{100c}, and $c_{11(R-1)c}$ never get used. Thus, they can be removed from the network. Next, a moment's thought shows that in terms of the channel identification numbers, this function is strictly increasing. In other words, if $(c_i, c_j) \in E$, then $i < j$. Therefore, the channel

dependency graph has no cycles, and the routing function is deadlock-free.

The method for sending the m_{NS} messages works in a manner similar to the hypercube example: P_{ij} sends a m_{NS} message on outgoing channel c_{dhij} only after it has received m_{NS} messages from each incoming channel $c_{d'h'i'j'}$ where $d'h'i'j' < dhij$.

Thus, unlike the hypercube method, where every processor can start sending m_{NS} messages after checkpointing, on a mesh only the processors p_{i1} can start by sending m_{NS} on their channel c_{00i1}. Like the hypercube algorithm, however, it sends approximately two m_{NS} messages per physical link, which results in $4n$ total m_{NS} messages.

K-ary d-cubes and Express Cubes

A k-ary d-cube is a general class of interconnection network topologies in which there are k^d nodes. These nodes are laid out in d dimensions, with k sets of nodes in each dimension. Channels connect neighboring nodes that differ in one dimension. Both hypercubes and toroidal meshes are examples of k-ary d-cubes: A d-dimensional hypercube is a binary d-cube, and a k-by-k toroidal mesh is a k-ary 2-cube.

General k-ary d-cubes are important because Dally has shown that multicomputers built using low-dimensional k-ary d-cubes as their interconnection networks give optimally low latency and high throughput [6]. Accordingly, many new multicomputers, such as the Paragon, Stanford's DASH [15], the J-Machine [8], the Ametek 2010 [21] and the Mosaic [22], have been designed to be k-ary 2 and 3-cubes.

Since all of these machines use wormhole routing and are well-behaved, they too can be checkpointed with our Network Sweeping algorithm. We don't include the details here, but they follow as an extension of the mesh example above.

Finally, Dally has proposed *Express Cubes* as an optimization to standard k-ary d-cubes [7]. Express cubes use extra *express channels* for messages to jump over a number of nodes in each dimension so that the node-latency of a message's transmission is reduced. If the space allows, multiple express channels can be added hierarchically to each dimension to give more opportunity for greater throughput. He shows that express cubes both lower latency and increase throughput. Again, multicomputers built with express channels are well-behaved, and thus can be checkpointed with the Network Sweeping algorithm. The extension here is simply to have m_{NS} messages sent along express channels as well as the normal channels.

6 Recovery

Recovery in our algorithm is the same as recovery in all the CL algorithms: Each processor reads its process state from its local checkpoint, and messages from the message log are replayed. This replaying can take one of two forms: Either each processor can resend each of its messages which was logged by the receiving processor, or each processor can rebuild its queue of messages from its own message log. The latter form is superior since it involves no extra communication between processors, and no reading of other processors' message logs.

7 Final Analysis and Optimizations

Our algorithm for checkpointing a multicomputer gives an overhead of $O(l(n) + pp(n))$ extra messages. As stated before, $pp(n)$ is equal to the number of distinct pairs of processors that communicate during the checkpoint. In the worst case, this number is $O(n^2)$, but this worst case is only realized if every processor sends an application program message to every other processor in the multicomputer while the checkpoint is in progress (this is as opposed to, or rather is a subset of [25], where the worst case is realized when every processor sends a message to every other processor during an entire checkpointing interval).

As a potential optimization (similar to the one in [14]), we can eliminate the need for extra m_{CL} markers if we simply attach an extra bit to each normal message in the system (on some multicomputers, like the iPSC/2, this bit comes for free, as there is an extra bit in message headers). Then, for messages coming from even checkpointing intervals, we set this bit to 0 and for those coming from odd intervals, we set it to 1. Now there is no need for m_{CL} messages after p_c's initial broadcast, because a processor can tell if it needs to checkpoint or if it needs to log an incoming message simply by looking at this extra bit.

Thus with this method, there is no possibility of degradation to $O(n^2)$ markers – instead, the number of extra messages is merely $O(l(n))$, which is $O(n \log n)$ for hypercubes and $O(n)$ for k-ary 2 and 3-cubes. This optimization does not come for free, however. In some multicomputers, there might not be any free bits in message headers. This combined with the fact that every receive operation must check this extra bit means that the regular processing of messages will be slowed down even in the absence of checkpointing. Thus, the degree of optimization (or degradation) depends on how much the multicomputer is slowed down by this extra bit, as well as on the frequency of checkpointing and the behavior of the program being checkpointed.

The performance of this algorithm should be sufficient to provide fault-tolerance for long-running programs without incurring a significant overhead. However, the latency of taking local checkpoints may be too large for implementing fine-grain time-sharing. This latency can be reduced by using a multicomputer's address translation hardware to overlap the most time-consuming operations of the checkpoint with the running of the target program [16]. Such a method allows a processor to save its checkpointed address space to disk concurrently with the target program and with the sending of marker messages. If checkpoints are taken frequently, this latency can be further reduced by saving only the pages that have been changed since the last checkpoint [10]. Similar techniques can be used to decrease the latency of recovery.

8 Conclusion

We have designed a new distributed checkpointing algorithm, specialized for the demands and opportunities that arise in checkpointing programs on multicomputers with wormhole routing interconnection networks. The checkpointing algorithm is unique in the distributed checkpointing literature in that it is highly efficient both when taking checkpoints and when recovering from checkpointed images. The algorithm achieves its high performance by making use of assumptions not satisfied in general by distributed systems, but satisfied by the routing hardware in most of the existing and proposed multicomputers of which we are aware. We are currently implementing the algorithm on the Intel iPSC/2 hypercube.

Acknowledgements

Kai Li was supported in part by NSF grant CCR-9020893, by DARPA and ONR under contracts N00014-91-J-4039, and by Intel Supercomputer Systems Division. Jeffrey Naughton was supported in part by NSF grant IRI-8909795 and by NSF Presidential Young Investigator Award. James Plank was supported in part by AT&T fellowship.

The authors would like to thank Jack Dongarra for his help and support.

References

[1] Mohan Ahuja. Repeated global snapshots in asynchronous distributed systems. Technical Report OSU-CISRC-8/89 TR40, Ohio State University, August 1989.

[2] William C. Athas and Charles L. Seitz. Multicomputers: Muessage-passing concurrent computers. *Computer*, 21(8):9–24, August 1988.

[3] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 90–99, Atlanta, Georgia, October 1983.

[4] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):3–75, February 1985.

[5] Carol Critchlow and Kim Taylor. The inhibition spectrum and the achievement of causal consistency. Technical Report TR 90-1101, Cornell University, February 1990.

[6] William J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[7] William J. Dally. Express cubes: Improving the performance of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 40(9):1016–1023, September 1991.

[8] William J. Dally. Fine-grain concurrent computing. *Research Directions in Computer Science: An MIT Perspective*, pages 127–154, 1991.

[9] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[10] S. Feldman and C. Brown. Igor: A system for program debugging via reversible execution. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):112–123, Jan 1989.

[11] Jim N. Gray *et. al.* The recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

[12] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Computer Systems Research at Rice University: Annual Report 1988-1989*, pages 83–102, 1989.

[13] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[14] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, May 1987.

[15] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *The 17th Annual International Symposium on Computer Architecture*, pages 148–159, Seattle, Washington, May 1990.

[16] Kai Li, Jeffrey F. Naughton, and James S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, Seattle, Washington, Mar 1990.

[17] Barbara H. Liskov and Robert W. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, july 1983.

[18] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings of the ACM Symposium of Distributed Computing*, pages 76–80, Montreal, august 1983.

[19] Steven F. Nugent. The iPSC/2 direct-connect communications technology. In *Proceedings of the Third Hypercube Conference*. ACM, November 1988.

[20] Michael L. Powell and David L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, pages 100–109, october 1983.

[21] C. L. Seitz *et al.* The architecture and programming of the Ametek series 2010 multicomputer. In *Proceedings of the Third Hypercube Conference*, pages 33–37. ACM, January 1988.

[22] C. L. Seitz *et al.* Submicron systems architecture project semiannual technical report. Technical Report TR-88-18, Caltech Computer Science Tech. Rep., November 1988.

[23] Madalene Spezialetti and Phil Kearns. Efficient distributed snapshots. In *Proceedings of The Sixth International Conference on Distributed Computing Systems*, pages 382–388, Cambridge, Massachusetts, May 1986. IEEE Computer Society.

[24] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, pages 204–226, August 1985.

[25] S. Venkatesan. Message-optimal incremental snapshots. In *Proceedings of The Ninth International Conference on Distributed Computing Systems*, pages 53–60, Newport Beach, California, June 1989. IEEE Computer Society.