CHECKPOINTING MULTICOMPUTER APPLICATIONS

Kai Li
Jeffrey F. Naughton
James S. Plank

CS-TR-315-91

April 1991

Checkpointing Multicomputer Applications

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

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. 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 multicomputer interconnection networks to satisfy this new set of requirements.

1 Introduction

Recently, multicomputer systems 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

- 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 the job after the higher priority job has terminated. This makes sharing

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

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

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, multicomputer checkpointing presents a set of opportunities and requirements significantly different from those present in general distributed systems. The special opportunities arise because multicomputers have a lower degree of node autonomy than do distributed systems, and 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 [Gra81, ML83, LS83], 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 [BBG83, PP83], 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 [JZ89, SY85], allow asynchronous logging of messages, and hence can be implemented to cause less overhead than the pessimistic schemes during normal operation. However, the optimistic schemes require complex recovery algorithms in order to discover and restore a consistent state. (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.

While node autonomy is important in general distributed systems, it is less important within a single parallel application such as a multicomputer application. By giving up on this node autonomy, our algorithm achieves both the low overhead of optimistic schemes and the simple recovery of pessimistic schemes. More specifically, in our checkpointing scheme there are identifiable global snapshots, and every processor knows approximately when a global snapshot begins and ends. This is what 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.

Finally, a host of checkpointing algorithms [SK86, LY87, KT87, Ahu89, CT90] have stemmed from a paper by Chandy and Lamport [CL85]. They view a distributed system as a set of processes with FIFO channels for communication. A global snapshot of a distributed system consists of a set of states for processes and a set of states for channels. Their method suffers on multicomputers for the following reason: At some point in the algorithm, processor p must be certain that there are no outstanding messages from a previous checkpoint interval bound for p but not yet received by p. The technique laid out by Chandy and Lamport for verifying that no more messages are outstanding requires $O(n^2)$ messages in multicomputers of n processors, which renders their checkpointing algorithm unacceptably slow when n scales to large machines. An improvement to this technique has been suggested in [Ven89], but it degrades to $O(n^2)$ messages as the interval between checkpoints grows larger. Thus, it is unacceptable for the suspending and resuming jobs in a shared environment.

A main contribution of this paper is a new algorithm, based on the deadlock-free routing algorithms employed by multicomputers. This algorithm guarantees that there are no outstanding messages with far fewer than $O(n^2)$ messages — $O(n \log n)$ for the hypercube connected multicomputers such as the Intel iPSC/2, and $O(n)$ for the next generation of mesh-connected multicomputers.

2 The Algorithms

The principal difficulty in taking a distributed checkpoint lies in determining and saving a consistent global state. In multicomputer applications, this reduces to ensuring that the processes involved in the checkpoint agree about which communications have and have not taken place.

A standard way to visualize the problem to view the individual processes as progressing down parallel, disjoint timelines as the computation proceeds. A communication between process p_1

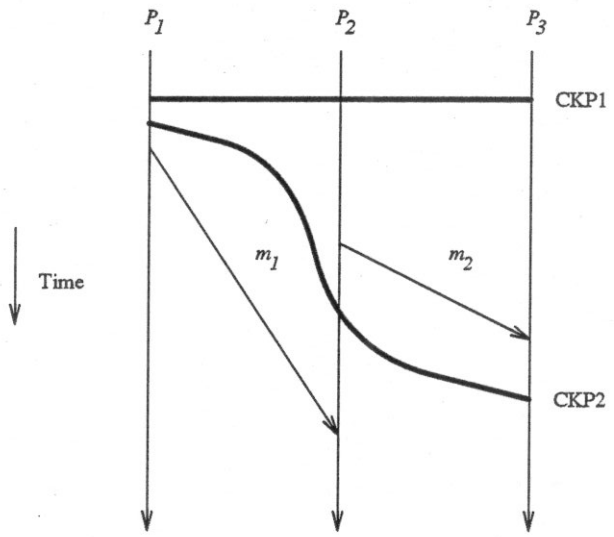

Figure 1: Two consistent checkpoints.

and process p_2 is represented by an arrow from the timeline for p_1 to the timeline for p_2, with the tail at the point in p_1's timeline where the message was sent, and the head at the point in p_2's timeline where the message was received. A possible checkpoint can be represented as a horizontal line cutting these timelines. For example, Figure 1 shows two possible checkpoints (labeled CKP1 and CKP2) of a computation involving the three processes P_1, P_2, and P_3. In the figure there are two messages: m_1 from P_1 to P_2, and m_2 from P_2 to P_3.

A moment's thought reveals that a checkpoint is a consistent state if and only if the line for the checkpoint crosses no message arrows. The reason for this is simple: if an arrow lies entirely above the checkpoint line, then all processes involved in the communication agree in the checkpointed image that the message was sent. If an arrow lies entirely below the checkpoint line, then all processes involved agree in the checkpoint that the message has not been sent. The two checkpoints in Figure 1 are both consistent; in CKP1, neither m_1 nor m_2 has been sent or received; in CKP2, m_2 has been sent and received, while m_1 has been neither sent nor received.

Conversely, if a message arrow starts above a checkpoint line and crosses the checkpoint line, then in the checkpointed image the sender thinks the message has been sent, while the receiver thinks that it has not, clearly an inconsistent state. Similarly, if the message arrow starts below the checkpoint line and terminates above the line, the sender thinks the message has not been sent while the receiver thinks that it has, again an inconsistent state. Figure 2 shows a checkpoint CKP that is not consistent for two reasons: m_1 has been sent but not received, while m_2 has been received but not sent.

In our checkpointing algorithms, there are explicitly identifiable system checkpoints, and the

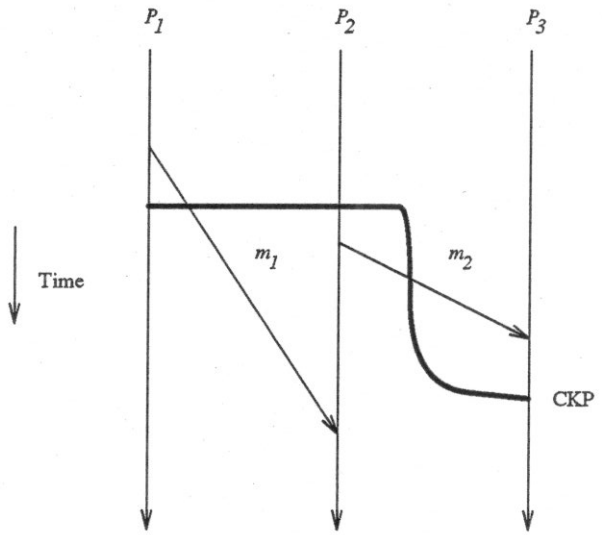

Figure 2: An inconsistent checkpoint.

nodes involved in a checkpoint are coordinated to the extent that each node understands that a particular local checkpoint corresponds to a particular system checkpoint. (A *local checkpoint* is simply saving the local state of the process to disk.) This is in contrast to the optimistic schemes [JZ89, SY85] in which there are no explicit system checkpoints.

We define two algorithms for checkpointing, each of which consists logically of two parts, *checkpoint coordination* and *enforcement of message consistency.* Checkpoint coordination makes sure that all processes take part in the checkpoint, and that each process knows when global checkpoints begin and end. Enforcement of message consistency ensures that messages like m_1 in Figure 2 are recorded as part of the receiving process's local checkpoint, and that messages like m_2 are eliminated entirely. Each part is explained in further detail below.

3 Checkpoint Coordination

For checkpoint coordination, we designate a special process p_c to be the coordinating process. Note that this is a departure from most standard distributed checkpointing algorithms, which typically assume no such coordinating process. While the requirement of a coordinating process is unreasonable for distributed systems in general, for multicomputers it is entirely reasonable, since at any given time the entire machine is coordinating to execute a single application. P_c goes through the following steps to coordinate each checkpoint:

1. P_c checkpoints itself.

2. P_c broadcasts a "start checkpoint" message to all other processes.

3. When p_c determines that all the processes have checkpointed, and that all necessary messages have been logged, it is free to start a new checkpoint.

Steps 1 and 2 are straightforward. Step 3, however, requires further explanation. The challenge here is to accomplish step 3 without flooding the interconnection network with control messages. The difficulty is the following: How can process p_i ever be sure that there is no pending message en route to p_i from process p_j? The simplest solution is the one suggested by Chandy and Lamport [CL85]: After taking a local checkpoint, process p_i sends a "local state checkpoint complete" message to every process p_j such that p_i can communicate directly with p_j. When p_j has received a "local state checkpoint complete" message from every process p_k such that p_k can send a message to p_j, then p_j knows that there are no outstanding messages bound for p_j from the previous checkpointing interval. At this point p_j can complete the checkpointing of its message state, and send p_c a "message state checkpoint complete" message. When p_c has received a "message state checkpoint complete" message from each process, the global checkpoint is guaranteed to be completed.

Such a method is correct and works effectively for distributed systems that are sparsely connected. For multicomputers with "wormhole routing" interconnections, every processor can send messages directly to every other processor. This means that on a multicomputer with n processors, the preceding solution requires $O(n^2)$ messages, and hence cannot be scaled to large configurations.

We have developed a new technique for determining when there are no outstanding messages in much less than $O(n^2)$ messages. The method is somewhat intricate; to avoid complicating the overview of our checkpointing algorithm we have moved an explanation of our new technique, including a proof of the general case and examples of how it applies to hypercube and mesh interconnects, to Section 6.

4 Enforcement of Message Consistency

The inconsistencies that arise from messages come in two flavors: Those that are recorded as sent but not received (e.g. m_2 in Figure 2) and those that are recorded as received but not sent (e.g. m_1). These inconsistencies can be eliminated if at all times the following two properties are guaranteed:

1. If a process p_j has not taken its local state checkpoint corresponding to system checkpoint k, then p_j does not "receive" any messages sent by a process p_i after p_i has taken its local state checkpoint for checkpoint k.

2. If a process p_j has taken its local state checkpoint for checkpoint k, then p_j appends to its message checkpoint state any message m sent by any other process p_i such that m was sent before p_i took its local state checkpoint for checkpoint k. Upon failure, these checkpointed messages can be "replayed" so that they will not be lost.

In short, these properties ensure that messages like m_1 in Figure 2 will be saved for replay, and that messages like m_2 will never occur. The two algorithms that we describe differ in the specifics of how they enforce these two properties.

4.1 Tag Bit Toggling

The first algorithm, which we call Tag Bit Toggling, uses an extra bit in each message to guarantee the above two properties. This extra bit is called the *tag bit.* (The use of a tag bit in distributed checkpointing has been proposed previously in [LY87].) In this scheme the tag bit toggles between two values, and at all times a process can examine the tag bit of an incoming message to determine if the message is from the current checkpoint interval or the previous checkpoint interval (the algorithm guarantees that there are no other possibilities.)

Suppose all processes have completed checkpoint $k-1$, but have not started checkpoint k. Furthermore, assume without loss of generality that the tag bit value for messages in the interval between checkpoints $k-1$ and k is zero. At some point p_c determines that a checkpoint needs to be taken, and sends "begin checkpoint" messages to all processes. Each process then observes the following protocol to ensure the above message consistency properties.

- If process p_i sends a message m before it takes its local state checkpoint for checkpoint k, then it sets m's tag bit to 0. If it sends m after taking its local state checkpoint, then it sets m's tag bit to 1.
- If process p_j has not taken its local state checkpoint for checkpoint k, and it receives a message with a tag bit of 1, then before processing that message, it must take its local state checkpoint.
- If process p_j has taken its local state checkpoint for checkpoint k, and it receives a message with a tag bit of 0, then it logs that message as part of its message state checkpoint.
- When the processes determine that global checkpoint k is finished, the function of the tag bit is flipped —- p_i continues to send tag bits of 1 until it takes its next local state checkpoint, for checkpoint $k+1$, after which it sends tag bits of 0.

The disadvantage of this approach is its per-message overhead.

4.2 Checkpoint Demarcation Messages

The second algorithm eliminates the per-message overhead of the tag bit. Instead, it uses *checkpoint demarcation messages*, which are variants of the marker messages described by Chandy and Lamport [CL85]. For this algorithm, each process maintains two bit vectors SENT and RECEIVED which contain n entries each (where n is the total number of processes). Moreover, there exists a special demarcation marker message M which every process recognizes. Suppose again that processes have completed global checkpoint $k-1$, but have not started checkpoint k. Then they observe the following protocol:

- If process p_j has not taken its local checkpoint for checkpoint k and it receives M from p_i, then before processing M, it must take its local checkpoint, clear the vectors SENT and RECEIVED, and then set RECEIVED[i] to 1.
- If process p_j has taken its local state checkpoint for checkpoint k, and RECEIVED[i] is 0, then if it receives any message other than M from p_i, it logs that message as part of its local state checkpoint. If it receives M from p_i, then it sets RECEIVED[i] to 1.
- If process p_i has taken its local checkpoint for checkpoint k, and SENT[j] is 0, then before it may send a message to p_j, it must send M to p_j, and set SENT[j] to 1.
- When checkpoint coordination determines that the global checkpoint is finished, then all bits of SENT and RECEIVED are set to 1.

The advantage of this method is that it does not add any overhead to existing messages, and it may be easier to implement (no need to add a bit to the message format). The method is superior to the scheme proposed by Venkatesan [Ven89]: In this scheme, each process remembers whether or not it has sent a message out on each of its channels since the previous checkpoint. If it has not, then it need not send a marker through that channel. Thus, if checkpoints are recorded at a frequent rate, the number of markers should be reduced. However, with this algorithm, the number of markers becomes a function of the time between checkpoints, which can be large. Our method only adds extra messages during the time a global checkpoint is being taken.

5 Recovery

Recovering from a system checkpoint made from these algorithms is fast and simple. Each process needs to retain only its previous two checkpoints and message logs. As part of its local state, p_c retains the number of the last completed system checkpoint. Upon recovery, p_c

broadcasts that number to all the other processes, which have retained their local checkpoint corresponding to that system checkpoint, as one of their last two checkpoints. The processes then recover to the state of that local checkpoint, and then replay the messages from their message logs. The recovery is then complete.

It is efficient as far as space is concerned, because for fault-tolerance, only the previous two checkpoints and message logs need be retained. This is to guard against a failure while a checkpoint is being taken. For job suspension and resumption, only the current checkpoint and message log need be retained. As far as time is concerned, the recovery is efficient, as all processes may recover in parallel – the only extra comunication is the initial broadcast of the system checkpoint number by p_c.

6 Determining Quiescence

In this section we address a key problem in our checkpointing algorithm: How a processor p in a multicomputer can determine that there are no more messages from the previous checkpoint interval that are bound for p but have not yet been received by p. The problem is complicated by the fact that multicomputer interconnects provide the illusion of complete connectivity. That is, while a given processor may only have physical connections to a few processors (for example, each node is connected to $\log n$ neighbors in an n-node hypercube), at the process level, messages can be sent between any pair of processors. Furthermore, if a message m is sent between processors p_i and p_j, and the message traverses the route $p_i p_k p_j$, there is no way for p_k to know whether or not m has been sent.

If we view the multicomputer as being completely connected, then there are n^2 links in the system, and an apparent lower bound on the number of messages to ensure no outstanding messages is n^2. However, while the interconnect provides the illusion of full connectivity, in reality each of these n^2 links 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 [DS87], and

2. Allocation of a route is non-preemptive; that is, once a message is allocated a route, no other message travels along the same channels until the message has been completely transmitted, and

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

We will call an interconnection network that satisfies these conditions a *well-behaved* interconnection network.

For well-behaved interconnection networks, we can determine that there are no outstanding messages in far fewer than n^2 messages. For example, in the current generation of hypercube connected multicomputers (e.g., the Intel iPSC/2 or iPSC/860), our algorithm takes $O(n \log n)$ messages; on the next generation of mesh-connected multicomputers, our algorithm takes only $O(n)$ messages. The next three subsections consider our algorithm for the general case, for the case of hypercube interconnects, and for the case of mesh interconnects.

6.1 General Case

Dally and Seitz [DS87] 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 which 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 an algorithm which lets every node determine that there are no pending messages destined for it which were sent before the sender checkpointed. It is assumed that each process 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 a message on channel c," we mean that if $c = (n_i, n_j)$, then p_i sends a message to p_j, thereby using channel c. The algorithm is defined as follows:

- 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 a "local state checkpoint complete" (LSCC) message on channel c_o.
- For all other outgoing channels c_o from p_i, when LSCC messages have been received from all channels c_j such that $(c_j, c_o) \in E$, p_i sends an LSCC message on channel c_o.

- When p_i has sent LSCC messages on all of its outgoing channels, and has received LSCC's on all its incoming channels, then it can send its "message state checkpoint complete" (MSCC) message to p_c.

The following theorems prove that the algorithm is correct:

Theorem 1 *Given a deadlock-free routing function, the above algorithm will terminate.*

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 $(c_i, c_1) \in E$, an LSCC message can instantly be sent on c_1. The proof proceeds by induction. Suppose that LSCC messages have 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 LSCC messages have been sent out all channels c_i. As f is deadlock-free, the messages are eventually received, and an LSCC message is sent out c_{k+1}. Therefore, LSCC messages are sent and received on all channels, prompting all the processes to send MSCC messages to p_c. Thus, the algorithm terminates. □

Theorem 2 *After sending its MSCC message, no process p_i will receive a message from p_j which was sent before p_j checkpointed itself, provided that all processes receive messages in a FIFO order from their incoming channels.*

Proof: Suppose p_j sends a message m to p_i before p_j has taken its local state checkpoint. Further, let $c_1, \ldots, c_l$ be the route that m takes; that is, $f(c_{k-1}, n_d) = c_k$ for $1 < k \leq l$. Since allocation of a route is non-preemptive, m will be received by p_i before p_j can send out another message on channel c_1. In particular, p_i receives m before p_j sends LSCC on c_1.

We now show that LSCC is received on all channels $c_1, \ldots, c_l$ after p_i has received m. This is done by induction on l. First, as noted above, LSCC is received on c_1 after p_i has received m. Next, suppose that for all $k < l$, LSCC is received on c_k after p_i has received m. Since $f(c_{l-1}, n_d) = c_l$, $(c_{l-1}, c_l) \in E$, and the LSCC message will not be sent on c_l until it has been received on c_{l-1}. Thus, LSCC is also received on c_l after p_i receives m.

Therefore, p_i receives LSCC on c_l after receiving m, and cannot send MSCC until after it has received all messages which were sent before their sender checkpointed. □

6.2 Hypercube Interconnections

The Intel iPSC/2 [Nug88] is a multicomputer with a hypercube interconnection network that is well-behaved. Each process is associated with a node on a hypercube. In an N-dimensional hypercube, nodes are numbered 0 through $2^N - 1$. If the binary representation of nodes i and

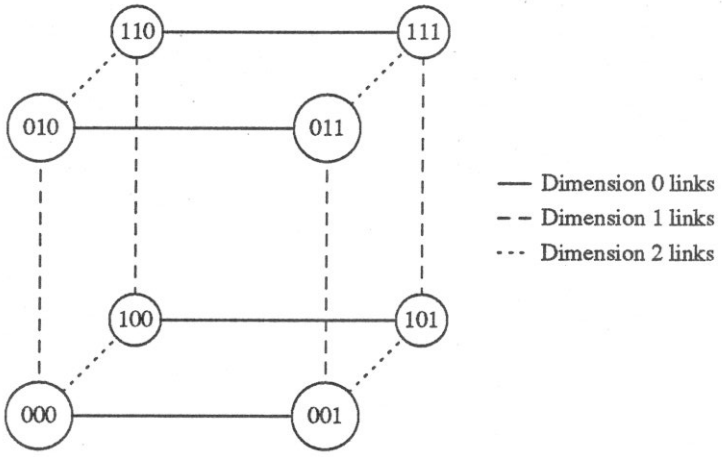

Figure 3: A 3-dimensional hypercube

j differ by exactly their k-th bit, then they are physically connected by a link which is said to be in dimension k. As an example, Figure 3 shows a 3-dimensional hypercube with its links.

Each link is bidirectional, and thus can be divided into two uni-directional channels. The iPSC/2 uses the *e-cube* routing function, which is deadlock-free. It specifies that the only channels used are those in the dimensions in which the binary representation of the source and destination nodes differ. These channels are used in increasing order of dimension. For example, in Figure 3, a message from node 000 to node 111 will use the channels $(000, 001), (001, 011)$, and $(011, 111)$.

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 which the binary representations of n_b and n_d differ. For example, in Figure 3, $f((000, 001), 111) = (001, 011)$.

The algorithm for sending LSCC messages, therefore, works as follows:

- After process p_i takes its local state checkpoint, it sends LSCC 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_i sends LSCC on its outgoing channel c_k of dimension k only after it has received LSCC on each incoming channel c_j of dimension $j < k$. This is because for each of those c_j, $(c_j, c_k) \in E$.
- When LSCC has been received on all incoming channels and sent on all outgoing channels, p_i sends MSCC to p_c.

Thus, this algorithm results in two LSCC messages sent per physical link. As each node

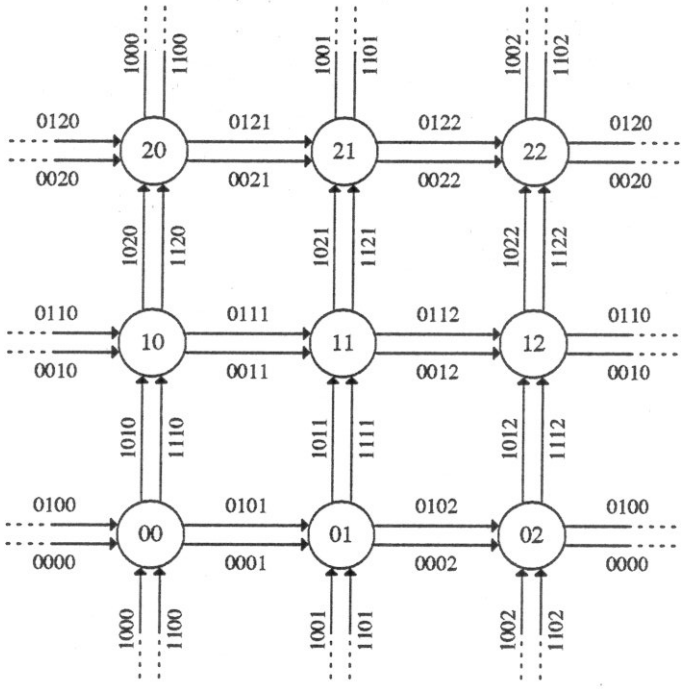

Figure 4: A 3 × 3 toroidal mesh

has $\log n$ links incident to it, this results in $n \log n$ LSCC messages, a significant improvement over $O(n^2)$.

6.3 Mesh Interconnections

As a second example for this algorithm, consider a toroidal mesh. (The next generation multicomputers, including the Intel Touchstone Delta and Sigma machines will use toroidal mesh interconnection networks.) In this interconnection network, 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:

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

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 = The 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, 1 if it's a low channel

r = The row number of its destination node

c = The column number of its destination node

Figure 4 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. To be formal:

$$f(c_{dhrc}, n_{ij}) = \begin{cases} c_{00r((c+1)} \text{ if } c > j \text{ and } c \neq C-1 \\ c_{01r((c+1) \bmod C)} \text{ if } (c+1) \bmod C \leq j \\ c_{10(r+1)c} \text{ if } c = j \text{ and } r > i \text{ and } r \neq R-1 \\ c_{11((r+1) \bmod R)c} \text{ if } c = j \text{ and } (r+1) \bmod R \leq i \end{cases}$$

The first thing to notice about this function is that channels with identification numbers of the form c_{00r0}, c_{01r1}, c_{100c}, and c_{111c} 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 algorithm for sending LSCC messages now works in a manner similar to the hypercube example:

- P_{ij} sends an LSCC message on outgoing channel c_{dhij} only after it has received LSCC messages from each incoming channel $c_{d'h'rc}$ where $d'h'rc < dhij$.
- When LSCC has been received on all incoming channels and sent on all outgoing channels, p_{ij} sends MSCC to p_c.

Thus, unlike the hypercube algorithm, where every process can start sending LSCC messages after checkpointing, in this algorithm only the processes p_{i0} can start, by sending LSCC on channel c_{00i1}. Like the hypercube algorithm, however, it sends approximately two LSCC messages per physical link, which results in $4n$ LSCC messages.

Therefore, by using properties of an interconnection network, and its deadlock-free routing function, our algorithm only sends $O(m)$ extra messages, where m is the number of physical links in the network. This is a significant savings over the $O(n^2)$ messages which occur when the properties of the network are ignored.

7 Conclusion

We have designed a new distributed checkpointing algorithm, specialized for the demands and opportunities that arise in checkpointing multicomputer programs. 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 all of the existing and proposed multicomputers of which we are aware. We are currently implementing the algorithm on the Intel iPSC/2 hypercube. It is our hope that through checkpointing our algorithm will greatly reduce the chances of lost work due to system failures, and will also for the first time enable efficient time-sharing of multicomputers.

Acknowledgements

Kai Li was supported in part by NSF grant CCR-8814265 and Intel Supercomputer Systems Division. James Plank was supported in part by AT&T fellowship.

References

[Ahu89] Mohan Ahuja. Repeated global snapshots in asynchronous distributed systems. Technical Report OSU-CISRC-8/89 TR40, Ohio State University, August 1989.

[BBG83] 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.

[CL85] 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.

[CT90] Carol Critchlow and Kim Taylor. The inhibition spectrum and the achievement of causal consistency. Technical Report TR 90-1101, Cornell University, February 1990.

[DS87] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[Gra81] Jim N. Gray et. al. The recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

[JZ89] 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.

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[LS83] 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.

[LY87] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, May 1987.

[ML83] 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.

[Nug88] Steven F. Nugent. The iPSC/2 direct-connect communications technology. In *Proceedings of the Third Hypercube Conference*. Association for Computing Machinery (ACM), 1988.

[PP83] 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.

[SK86] 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.

[SY85] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, pages 204–226, August 1985.

[Ven89] 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.