We omit the proof of this, as it may be found in [CL85]. The crux of the proof is that any cross-cut message must arrive at a node after it has checkpointed, but before it receives m_{CL} from the sender. Thus they are all identified and logged properly.

One way to measure the complexity of this algorithm is to count the number of extra messages sent. If the distributed system has n nodes and m channels, then the CL algorithm sends $O(m)$ m_{CL} messages, and $O(n)$ m_S messages. For the system to be connected, m must be at least $n-1$; therefore the CL algorithm sends $O(m)$ extra messages.

While the CL algorithm seems to be ideal for multicomputer checkpointing, it fails for the following reason: If the network being checkpointed is fully connected, the algorithm sends $O(n^2)$ extra messages, which is unacceptable for large values of n. While the processors in current multicomputers are not physically fully connected, the "wormhole" or "virtual cut-through" routing networks used in multicomputers enforces 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 CL checkpointing algorithm on a multicomputer will have message complexity $O(n^2)$, making it unacceptably slow when n scales to large machines. Of all the variants of the Chandy-Lamport algorithm, only two attempt to reduce this number of messages. The former, by Venkatesan [Ven89], 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 applica-

tion 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.

Cristian and Jahanain [CJ91] give a consistent checkpointing algorithm that uses timing constraints to avoid sending the $O(n^2)$ messages. For their algorithm to work, the time skews between processors must be bounded, as must message transmission time. That way, the processors can check their local clocks to determine whether a message is potentially a cross-cut message without broadcasting m_{CL} messages. The authors provide no implementation for this algorithm, but state that their timing requirements are reasonable.

2.6 The Network Sweeping Algorithm

In this section, we present a checkpointing algorithm that greatly reduces the number of marker messages that a processor must send without imposing timing constraints on the machine's architecture. We call this the *Network Sweeping (NS)* algorithm as it uses properties of wormhole routing to "sweep" the interconnection network of a multicomputer, and checkpoint in fewer than $O(n^2)$ messages. To state a bound on the number of messages sent by the checkpointing algorithm, we define two functions:

$l(n)$ = 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/860, $l(n)$ is $O(n \log n)$; on mesh-connected multicomputers (such as the Intel Delta and 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

Venkatesan's algorithm [Ven89]. As a side note, Venkatesan proves that his 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 his proof.

The NS algorithm uses three types of extra messages to take a checkpoint:

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. Bidirectional links can be viewed as two unidirectional links). We discuss how to make this determination below in Section 2.6.2. Until then, the above definition suffices for our explanation.

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

The algorithm works as follows: P_c initiates global checkpoint i by broadcasting an m_{CL} message with a sequence bit of (i mod 2) to all other processors, and then immediately taking its local checkpoint. All processors p then perform the following:

- If p receives m_{CL} or m_{NS} and has not 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 $(i \bmod 2)$ 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 its 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 $(i \bmod 2)$.

- When p receives m_S from p_c, then it knows that checkpoint i is over. It may now delete old checkpoints. [1]

[1] One might note that it is 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 $i+1$.

To give a bit of intuition about how this algorithm works, we present two examples.

Example 1 In this example, we treat the system in Figure 2.5 as a multicomputer, instead of a distributed system. Thus, p_c can send a message directly to p_2: Even though the message passes through the routing hardware of p_1, the processor p_1 has no knowledge of it. Although this multicomputer is extremely simple, it is sufficient to illustrate the details of the algorithm.

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 2.7. We detail them below:

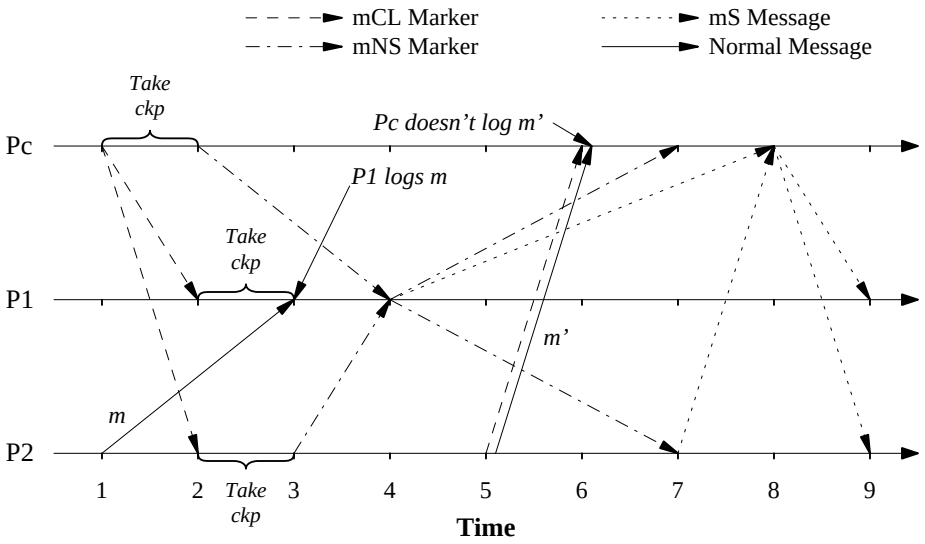

Figure 2.7: Time line of Example 1.

1. At time 1, p_c decides that it wants to start checkpoint 1. Thus it broadcasts m_{CL} with a sequence bit of 1 to the other two processors, and then

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) We show how this can occur in Example 2. .

takes its local checkpoint. When it finishes its local checkpoint, p_c cannot send any more cross-cut messages. Therefore, since its link to p_1 will not receive any more cross-cut messages, it sends m_{NS} to p_1.

2. Also at time 1, p_2's application program sends the message m to p_1.

3. At time 2, p_1 and p_2 both receive m_{CL} from p_c. Thus they checkpoint themselves. After checkpointing itself, processor p_2 cannot send any more cross-cut messages. Therefore, it sends m_{NS} to p_1. Similarly, after checkpointing itself, p_1 cannot send any more cross-cut messages; however, it cannot send any m_{NS} messages over its outgoing links, since it does not know whether its link to p_2 will receive cross-cut messages from p_c, and vice versa.

4. At time 3, p_1 receives m from p_2. Since it has not received m_{CL} from p_1, it knows that m is a cross-cut message: p_1 logs m.

5. At time 4, the two m_{NS} messages bound for p_1 arrive. Now, p_1 knows that no cross-cut messages will be sent through its outgoing links, and it 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 5, 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 6, 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 does not log m'.

8. At time 7, 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 it m_S.

9. At time 8, 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 9, p_1 and p_2 receive m_S from p_c. Thus they know that checkpoint 1 is over. They may now delete their local checkpoints and message logs for checkpoint 0.

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 2.6.2 below. □

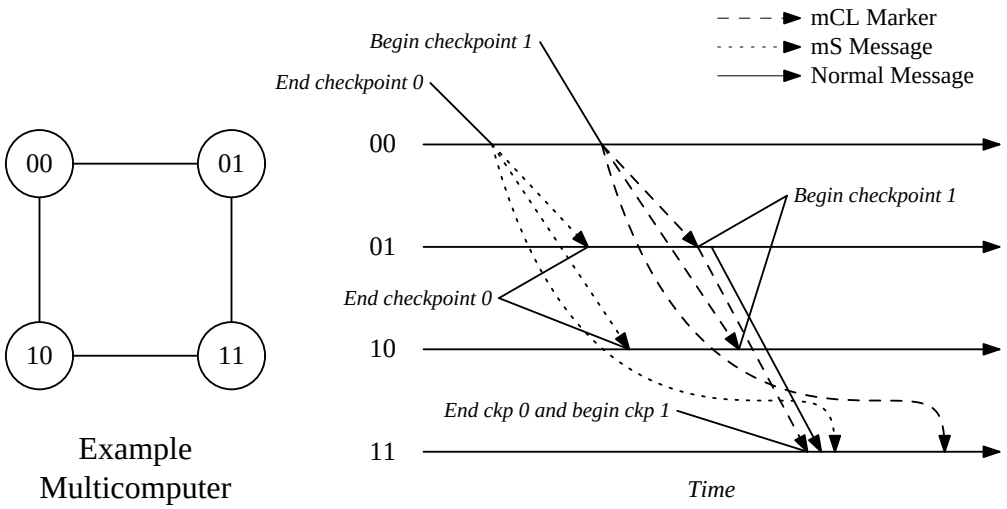

Figure 2.8: Time line of Example 2.

Example 2 This second example shows how an m_S message from one checkpoint can be bypassed by an m_{CL} message from a subsequent checkpoint. It assumes the four-processor multicomputer shown in Figure 2.8, where messages from processor 00 to processor 11 pass through the routing hardware of processor 10. The example focuses on what could happen at the end of checkpoint 0 and the beginning of checkpoint 1. Processor 00 broadcasts the m_S message to end checkpoint 0, and because of contention for the link from 10 to 11, it takes a long time for the m_S message to reach processor 11. Soon after ending checkpoint 0, processor 00 begins checkpoint 1 by broadcasting m_{CL} messages with sequence bits set to 1. Soon after it begins checkpointing,

processor 01 sends a normal message to processor 11, which it must precede with an m_{CL} message whose sequence bit is set to 1. As this message takes a different path than the m_S message, it might reach processor 11 before the m_S message. As described above, processor 11 takes this as a signal to end checkpoint 0 and start checkpoint 1. It then ignores the m_S message from processor 00. □

The total number of messages that the NS 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.

2.6.1 Proof of Correctness

The following theorems prove the algorithm's correctness, and further illustrate more details of the algorithm.

Theorem 2.6.1 *The 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 cannot 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 theorem provides 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 theorem shows that these m_{CL} messages are properly ignored, even if another checkpoint is started soon after the current one finishes.

Theorem 2.6.2 *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 theorem 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 has not 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 does not 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 has not 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 does not ignore does indeed come from checkpointing interval $i + 1$.

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

Theorem 2.6.3 *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 does not receive a m_{CL} message from p prior to receiving m. As stated above, the only m_{CL} messages that are not ignored are those from checkpoint i. Since p sends m before starting checkpoint i, p' will not 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. □

Theorem 2.6.4 *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 does not matter: p' already knows that m is a non-cross-cut message. In either case, p' does not 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 theorems show that the Network Sweeping algorithm is correct: It defines a consistent cut, and correctly identifies and logs the cross-cut messages. Moreover, the theorems 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.