For example, suppose processors P_1 and P_2 do not synchronize with each other when they checkpoint. Then the following scenario could occur:

Time	P_1	P_2
1	Save CPU State	
2	Read x	
3		Write x
4		Save CPU State
5		Read x

The resulting checkpoint will be invalid regardless of when the state of memory is saved. If it is saved at or before timestep 2, then P_2 will read the wrong value of x upon recovery. If it is saved after timestep 2, then P_1 will read the wrong value of x upon recovery. Thus, all the processors must agree on the state of memory that is saved to disk.

This leads to a very simple algorithm for checkpointing a multiprocessor:

- Freeze all the processors. This can be done with a system call, if the operating system provides one. Alternatively, it may be effected through interprocessor interrupts, or by setting the page protection bits of the address space to "no access" and getting control of the processors in the page fault handlers.
- Save the state of the processors to disk.
- Save the state of the memory to disk. Note that if the processors have write-back caches or relaxed consistency models, then the processors should first flush their caches to make sure that the view of memory seen by each processor is the same as the contents of memory saved to disk.
- Unfreeze the processors.

We call this *sequential checkpointing*, as it is a totally sequential algorithm. It is illustrated in Figure 2.1. Note that by freezing all the processors, they are synchronized with one another and with the state of memory.

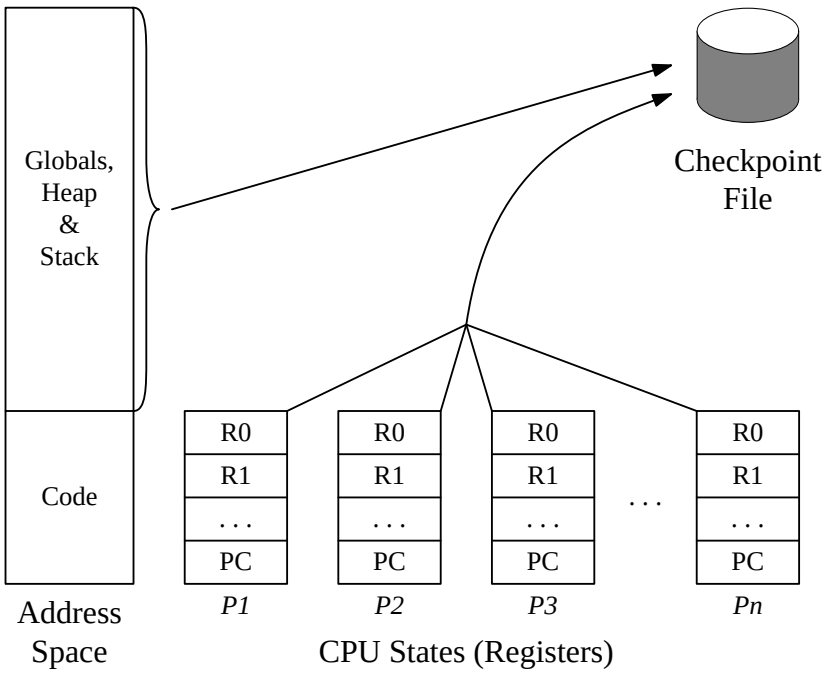

Figure 2.1: Sequential Checkpointing Algorithm for Multiprocessors.

The sequential checkpointing algorithm is clearly optimal in terms of overall checkpoint time, as it is limited solely by the duration of the disk writes plus the time to synchronize the processors. However, its checkpoint overhead is equal to the checkpoint time, giving it zero concurrency. Moreover, the latency is also equal to the checkpoint time, giving it bad real-time behavior. Thus, there are several ways in which it can be improved. These are discussed in Chapter 3.

Recovery from this checkpoint is straightforward:

- Freeze all the processors.
- Read the state of memory from disk
- Read the state of the processors from disk.
- Unfreeze the processors.

The processors continue running from the recovery point.

2.2 Checkpointing Multicomputers

The rest of this chapter describes algorithms for checkpointing and restarting parallel programs on multicomputers. Unlike multiprocessors, multicomputers have more of a problem synchronizing to define a valid recovery point. As described above, multiprocessors all share a global memory, and can synchronize around it. Multicomputers do not have a globally shared resource, and thus they do not have a convenient mechanism through which to synchronize. Consistent checkpointing algorithms give mechanisms through which processors can synchronize and take a valid checkpoint.

Consistent checkpointing has been well researched for distributed systems. While multicomputers bear a great deal of similarity to distributed systems, there are a few differences that render many consistent checkpointing algorithms too costly for use on multicomputers with a lot of processors. In Section 2.6 we present a new algorithm called *Network Sweeping*, which is based on the properties of *wormhole routing*, the kind of routing most often employed in today's multicomputers. For massively parallel multicomputers, the Network Sweeping algorithm should outperform the other algorithms presented.

2.3 Consistent Checkpointing Nomenclature

The standard way of visualizing a distributed group of processors is to use horizontal lines that denote the relative progress of the processors over time. A message between processors is denoted by an arrow from the point at which the sending processor sends the message to the point at which the receiving processor 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*.

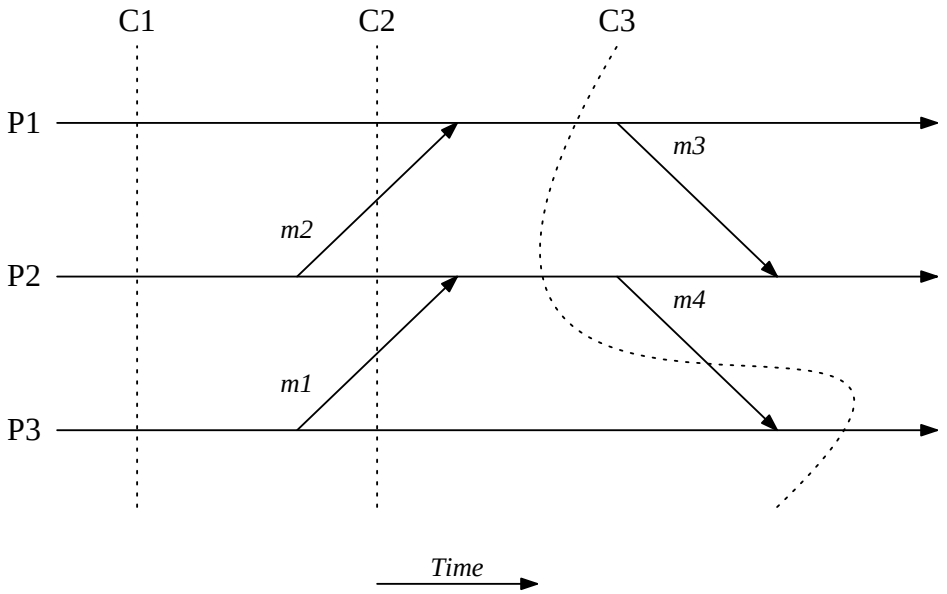

Figure 2.2: A Sample Distributed System.

Figure 2.2 shows a distributed system with three processors, 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_4.

Checkpoints are related to cuts as follows: A distributed checkpoint consists of *processor state* and *message state*. The processor state is created from individual *local checkpoints* of each processor (that is, a uniprocessor checkpoint as described in Chapter 1), and the message state is a *log* of messages from one processor to another. Given a cut C, one can construct a distributed checkpoint by having each processor 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 processor from its local checkpoint, and then resending the messages in the log.

It is well-known [KT87, CT90] that checkpoints can only be made from consistent cuts. Inconsistent cuts lead to the recording of spurious messages in the processor state of the checkpoint. For example, suppose a global checkpoint is taken from cut C_3 in Figure 2.2. Then the processor states recorded

in P_3's local checkpoint will be contingent on having received m_4, but P_2's checkpoint will not record having sent it. When the processors recover, there's no guarantee that P_2 will execute to a point where it resends m_4. If it does, then perhaps P_3 can ignore the new message, having already recorded the old one's receipt. However, if it does not resend m_4, then P_3's state will be invalid as it relies upon a message that is never sent. Thus, inconsistent cuts lead to invalid distributed checkpoints.

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_4 in Figure 2.2 are cross-cut messages; m_3 is a non-cross-cut message.

2.4 The Sync-and-Stop Algorithm

The simplest consistent checkpointing algorithm is like the multiprocessor checkpointing algorithm of Section 2.1. It basically shuts down the multicomputer to define a consistent cut and take a global checkpoint. We call it the *Sync-and-Stop (SNS)* algorithm, as the processors first synchronize to define a consistent cut, and then stop to take their local checkpoints. Specifically, the algorithm works as follows:

- One processor is designated to be p_c, the *coordinating processor*. It is this processor that starts and stops checkpoints.
- When p_c decides it is time to take a checkpoint, it stops running the target program and broadcasts a special *marker* message, m_{SNS}, to all the other processors.
- When a processor p receives m_{SNS}, it stops running its target program. Next, it must wait for all messages that the target has sent to be received.

How a processor does this is dependent on the system. For example, in some multicomputers, like the Intel ones, there is a system call that allows the sending processor to test whether messages have been received by the receiving processor. As another example, in a system where all messages are acknowledged, the processor must merely wait for all acknowledgements to come in.

- After all of p's messages have been received, p sends m_{SNS} back to p_c.
- When p_c has received m_{SNS} from all other processors, p_c rebroadcasts m_{SNS} and saves its local checkpoint to disk.
- When p receives this second m_{SNS} from p_c, it saves its local checkpoint to disk. After completing the local checkpoint, p sends m_{SNS} back to p_c.
- When p_c has received this last m_{SNS} from all other processors, the checkpoint is complete. It notifies the other processors by broadcasting m_{SNS} a final time.
- When p receives this final m_{SNS} message, it performs its final checkpointing activities: It may delete previous checkpoints and restart the target program.

Figure 2.3 shows a time-line representation of the SNS algorithm. Here processor p_2 sends a message m to p_1, and has to wait for p_1 to receive m before it may send its first m_{SNS} back to p_c. Note how the algorithm works to define C, the cut corresponding to the local checkpoints. As no messages cross this cut, it is consistent, and thus the checkpoint is valid. As a matter of fact, all cuts produced by the SNS algorithm are consistent because there are never any messages that cross them. Theorem 2.4.1 proves this property:

Theorem 2.4.1 *No message can cross a cut defined by the SNS algorithm.*

Proof: For the sake of clarity, we assume that p_c sends m_{SNS} messages to itself during broadcasts—thus when we refer to "any processor p" below, p can

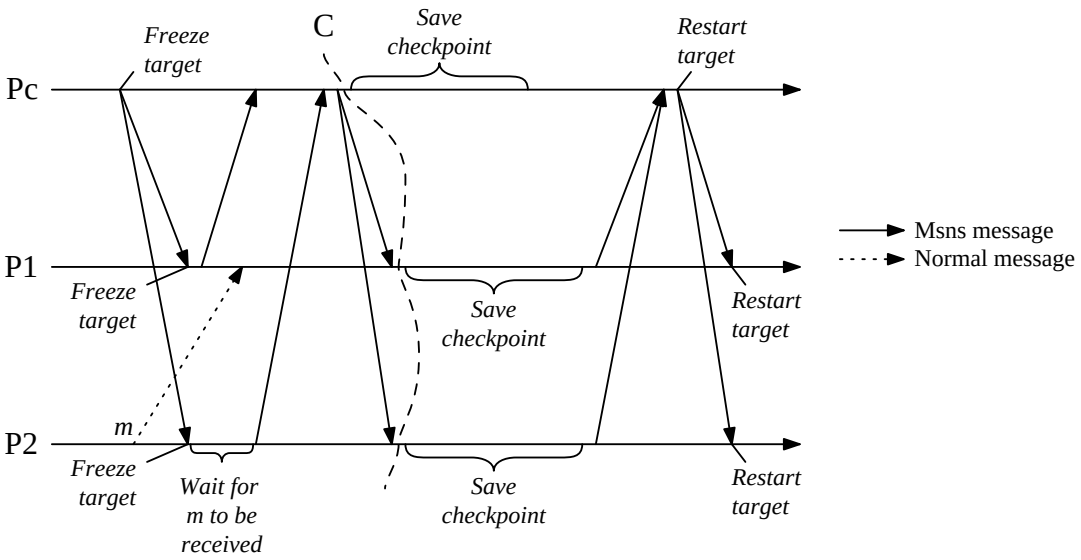

Figure 2.3: The Sync-and-Stop algorithm at work.

be p_c. We also assume that all messages sent between any two processors are received in FIFO order.

For any processor p, p's target program is frozen from the receipt of the first m_{SNS} message until the receipt of the final m_{SNS} message. Thus, any message m that would potentially cross the cut has to originate either before its sender receives the first m_{SNS} message or after its sender receives the last m_{SNS} message. Consider the first case (which is the one illustrated in Figure 2.3). If p sends m before it receives the first m_{SNS} message, then p must wait for m to be received before it may send the second m_{SNS} back to p_c. As p_c must wait for this message before its rebroadcast of m_{SNS}, p_c must wait for m to be received before rebroadcasting m_{SNS}. As no processor can start checkpointing before receiving this rebroadcasted m_{SNS} message, all processors must start checkpointing after m is received: m cannot cross the cut.

In the second case, p sends m after it receives its last m_{SNS} message. Since p_c waits for all processors to finish checkpointing before it sends the last m_{SNS} message, all processors must be finished checkpointing with p sends m. Thus,

m is received after the receiving processor is finished checkpointing, and m cannot cross the cut. Therefore, in the SNS algorithm, no messages may cross the cut. □

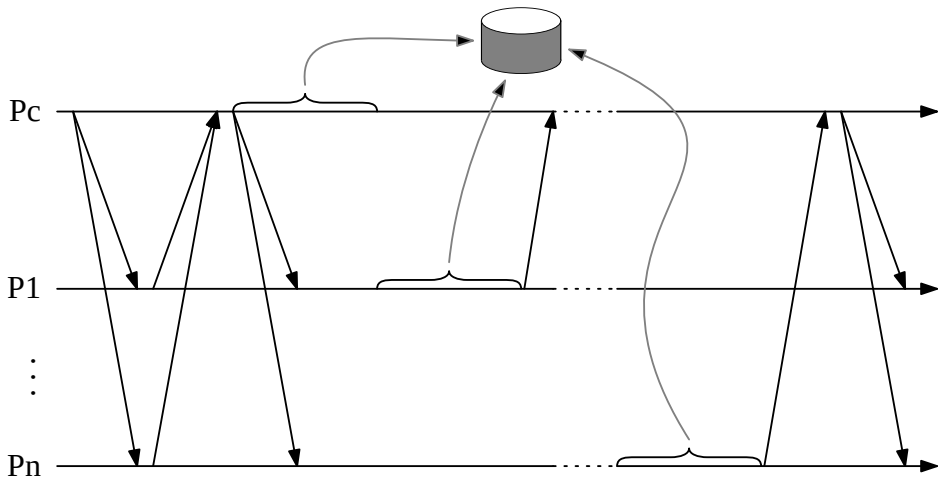

Figure 2.4: The Sync-and-Stop algorithm serializing at the disk.

Again like the sequential checkpointing algorithm for multiprocessors, the SNS algorithm is an algorithm where the overhead of checkpointing is very large. Not only are processors halted to write their checkpoints to disk, but they must also halt while other processors write their checkpoints to disk. This adds a large amount of needless overhead to the checkpointer. As an example, consider the scenario in Figure 2.4. Here there is only one disk that n processors share. Therefore when they write their checkpoint to disk, they end up serializing, and if it takes time t for one processor to write its checkpoint to disk, then it takes at least time nt, all of which is overhead, for the checkpoint to complete. Obviously, this algorithm can be improved to impose less overhead. Some of the optimizations of Chapter 3 help to decrease this overhead. The next two algorithms in this chapter should drastically improve the overhead as well.

2.5 The Chandy-Lamport Algorithm

The *Chandy-Lamport (CL)* [CL85] checkpointing algorithm solves the major problem with the SNS algorithm, which is the freezing of all processors. Instead, it is more liberal in the types of consistent cuts that it allows. Such cuts can include cross-cut messages, which must be logged as part of the global checkpoint.

The Chandy-Lamport algorithm is defined in terms of distributed systems. That is, a system has processors that are connected by *channels*. A processor p_i can send a message directly to p_j only if p_i and p_j are connected by a channel. Otherwise, the message has to go through an intermediate process. For example, in the distributed system pictured in Figure 2.5, p_c can only send messages directly to p_1. All messages to p_2 must pass through p_1. The algorithm assumes that all channels are FIFO.

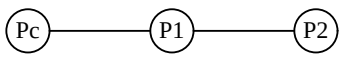

Figure 2.5: A simple distributed system.

The Chandy-Lamport algorithm works as follows: Some processor p_c starts the checkpoint by broadcasting a special message m_{CL} to all processors to which it has a channel. Immediately after this broadcast, it takes a local checkpoint. All processors p then perform the following:

- If p receives m_{CL} and has not taken its local checkpoint yet, then it broadcasts m_{CL} to all processors to which it has a channel, and takes its local checkpoint immediately after the broadcast.

- 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 local check-

point is finished. Each processor notifies p_c at this point by sending it a *stop* message, m_S (if p is not directly connected to p_c by a channel, then it sends m_S via intermediate processors, which may save on bandwidth by combining them with their own m_S messages).

- When p_c receives all the m_S messages, then it broadcasts m_S back to the other nodes (again, if p is not directly connected to p_c, then the broadcast is done via intermediate nodes).
- When p receives this last m_S, then the checkpoint is over. It may now delete old checkpoints.

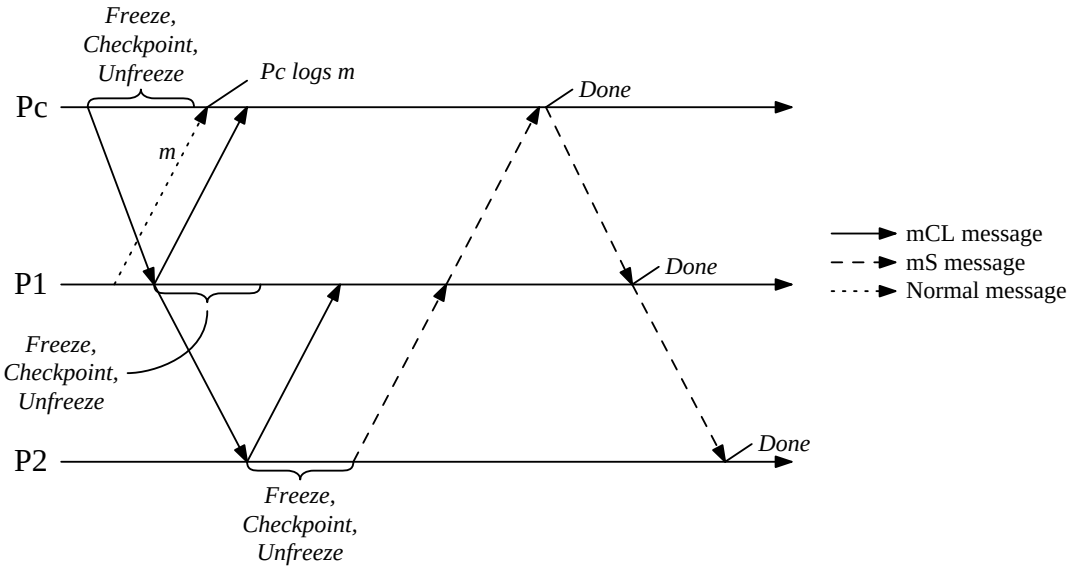

Figure 2.6: The CL Algorithm at work.

Figure 2.6 shows an example of the CL algorithm checkpointing the distributed system of Figure 2.5. Note that m is a cross-cut message, and is logged properly.

The Chandy-Lamport algorithm is correct because it defines a consistent cut, and takes a complete global checkpoint corresponding to that cut. Specifically, it logs all cross-cut messages, and does not log non-cross-cut messages.