EFFICIENT CHECKPOINTING ON MIMD ARCHITECTURES

James Steven Plank

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCEINCE

June 1993

Abstract

Presented here are efficient algorithms for checkpointing on MIMD architectures. These algorithms have been implemented on two representative machines: a shared-memory multiprocessor, and a message-passing multicomputer. The algorithms and implementations are evaluated according to three speed metrics: checkpoint time, overhead, and latency.

Checkpointing is important as a general means of software fault-tolerance. It is also the backbone of certain program control utilities, such as job-swapping, process migration, and playback debugging. We employ several techniques to minimize the invasiveness of the checkpointer on the target program. Such techniques are main memory checkpointing, copy-on-write, buffering, compression, and the elimination of bottlenecks and extra control messages.

The major result of this dissertation is that we can implement efficient checkpointing on MIMD architectures, thereby enhancing the usability of such machines.

Acknowledgements

I've been fortunate to work under the guidance of Kai Li. Kai has taught me many things about computer science and academia, and I have much to learn from his example of how to advise students. I am also indebted to Jeff Naughton for getting me interested in checkpointing, and being a collaborator in research.

I am indebted to several people for helping me finish this dissertation. I wish to thank Anne Rogers for being a reader, and Andrew Appel, David Dobkin and Pat Hanrahan for attending the defense. Melissa Lawson has been a lifesaver in administrivia. I'd also like to thank Jack Dongarra for providing me with space to work and access to machines, and Brad Vander Zanden for his generosity with his time.

Several people have donated their time and programs to help test checkpointing. I thank Michael Vose, Viktor Eijkhout, Jack Dongarra, Elmootazbellah Elnozahy, Willy Zwaenepoel, Susan Ostrochov, Al Geist and Steve Moulton for providing test programs for checkpointing. Karin Petersen, Matt Blumrich and Liviu Iftode have been invaluable in fixing and rebooting broken hypercubes while I'm 700 miles away.

There are many people who have made Princeton an enjoyable place to live and work. Adam Buchsbaum has provided me with more ways to waste time than I ever really needed. Norman Ramsey has shared with me a love of power tools, not-so-popular music, and writing non-thesis-oriented software. Adam, Mary Fernandez, Norbert Schlenker, Norah McCloy and Karin Petersen have provided me with many evenings of good food, card playing, and entertainment. Matt Blumrich and Mark Greenstreet have been solid and entertaining officemates for the duration of my stay in Princeton. Finally, I'd like to thank Stefanos Damianakis, Joe Studholme, Logan Fox, Ray Miller, Brad Vander Zanden, Bill Rosener, Gavin O'Brien, Derek Martin and especially David Dobkin and Geter Hicks for knowing the importance of devoting many hours of one's week to the playing of basketball.

I'd like to thank my parents and my sisters for providing me with love and emotional support all my life. Most of all, I thank my wife, Heather, for everything that has made my life happy since we've been together.

My last three years have been funded by an AT&T fellowship. This work has also been funded by NSF grants CCR-9020893 and CCR-8814265, DARPA and ONR contract N00014-91-J-4039, Intel Supercomputer Systems Division, and Digital Equipment Systems Research Center and External Research. I am grateful to Jack Dongarra, Al Demers and Xerox PARC for summer employment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, supercomputer systems with large numbers of processors, high-speed and scalable interconnects, and concurrent I/O systems have become commercially available. While the raw processing power of these machines is impressive, their general usability and scale is limited by the need for fault-tolerance. The combination of more and more components in a single system greatly reduces the system's mean time to failure. For example, if the mean time to failure of a microprocessor is F, then the mean time to failure of a system of n microprocessors is $\frac{F}{n}$. With such a large combination of components, hardware fault-tolerance is infeasible. Thus, supercomputers are greatly in need of software fault-tolerance.

This dissertation explores general software fault-tolerance on *Multiple-Instruction-Multiple-Data (MIMD)* architectures. Specifically, it presents algorithms for *checkpointing* on MIMD architectures, and evaluates implementations of these algorithms with respect to their speed and invasiveness. The main result of this dissertation is that we can implement efficient checkpointers on MIMD architectures, thereby adding fault-tolerance and job-swapping where there was none previously.

Checkpointing, also known as "backward error recovery," is an important means of software fault-tolerance as it is the only known mechanism that can tolerate unanticipated faults—faults that are not envisioned in the design of

a system [AL81]. We present algorithms and optimizations for checkpointing the two types of MIMD architectures: Shared-memory *multiprocessors*, and message-passing *multicomputers*. All algorithms and optimizations are implemented on a representative multiprocessor and multicomputer. We discuss the implementations and their results in detail.

1.1 MIMD Architectures

This dissertation studies software fault tolerance for two kinds of MIMD architectures: multiprocessors and multicomputers.

1.1.1 Shared-Memory Multiprocessors

Figure 1.1 shows the basic architecture of multiprocessors. There are n processors in such a machine, each of which is connected to a bus or interconnection network. All processors can access all memory modules in the architecture.

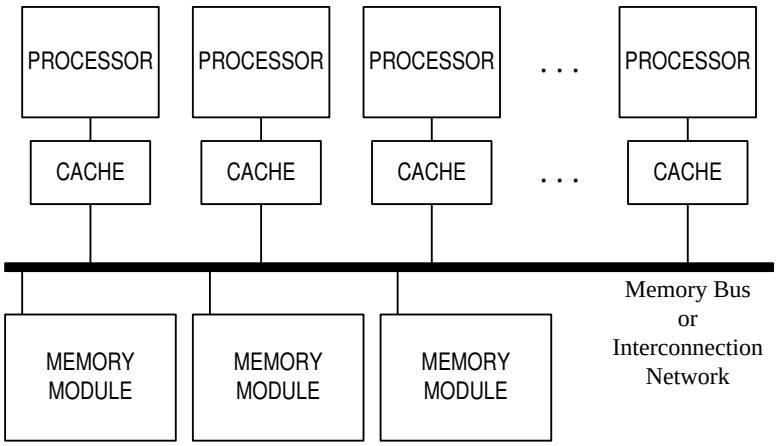

Figure 1.1: Basic Architecture of a Shared-Memory Multiprocessor.

In most multiprocessors, each processor has a local cache enabling it to make fast memory references. These caches have extra hardware between them and the bus or interconnection network that ensures that local cache entries do not become stale when a single memory location is present in several

processors' caches. These are known as *snoopy caches*. The problem of keeping snoopy caches coherent is known as the *cache coherence* problem, and has a rich history in parallel architecture research [AB86].

Examples of multiprocessors are the Sequent Balance and Symmetry, the DEC-SRC Firefly, the SGI Powerstation, the BBN Butterfly, the Stanford DASH and MIT's Alewife. The first four have snoopy caches connected to a shared bus. Because the bus becomes a bottleneck, these multiprocessors are limited in the number of processors that they can handle. The machine with this design containing the largest number of processors is the Symmetry, with 32 processors.

On the BBN Butterfly, processors do not have caches. Instead, each processor is directly connected a local memory that is a small portion of the global address space. To access the rest of the global memory, a processor must issue a memory request over an interconnection network that forwards the request to another processor's local memory, which then services that request. This protocol allows for Butterflies to be built with a larger number of processors, the most being 256.

On the last two multiprocessors, instead of having a single, shared bus, there is an interconnection network which attaches the processors to the shared memory. Cache coherence is maintained by *directories*, which keep track of the memory locations contained in the processors' caches. The directories are responsible for sending out invalidation messages to other processors when possible coherence problems are encountered. Directory schemes are often combined with *relaxed consistency* models for building large scale multiprocessors [AH90, GLL+90]. With relaxed consistency models, processors' caches are allowed to be inconsistent with memory and with other processors' caches until executing synchronizations or fences. This reduces the degree of false sharing and the number of invalidation messages.

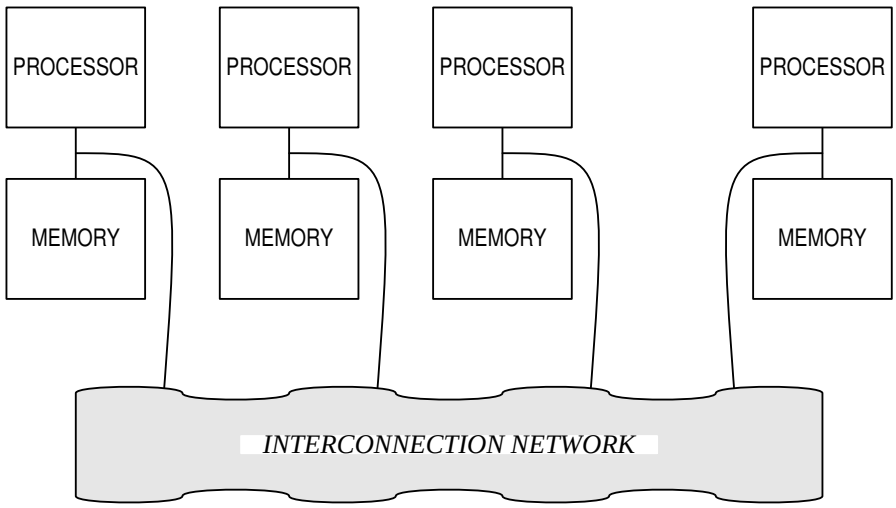

Figure 1.2: Basic Architecture of a Message-Passing Multicomputer.

1.1.2 Message-Passing Multicomputers

Figure 1.2 shows the basic architecture of multicomputers. Multicomputers consist of n processors, each with a local memory that no other processor can reference. The processors communicate with one another by sending messages over an interconnection network.

One of the most important parts of a multicomputer is the interconnection network. In general, machine designers try to design multicomputers with interconnects that are scalable, high-bandwidth and low-latency. Popular interconnection networks include hypercubes, two and three-dimensional meshes, and fat trees. With these scalable interconnection networks, multicomputers can be built with many more processors than multiprocessors. Table 1.1 gives a listing of example multicomputers with their interconnection network topologies and maximum number of processors.

Machine	Topology	Maximum Number of Processors	
		Currently Built	Design Limit
Intel iPSC/2	Hypercube	128	128
Intel iPSC/860	Hypercube	128	128
Intel Delta	2-d Mesh	570	570
Intel Paragon	2-d Mesh	500	1000
MIT J-Machine	3-d Mesh	1024	4096
CalTech Mosaic	3-d Mesh	1024	4096
Thinking Machines CM-5	Fat Tree	1024	65536

Table 1.1: Example multicomputers

1.2 Checkpointing

To checkpoint a program for fault-tolerance, one must save enough information to stable storage that after a machine fails and its volatile state is lost, one can reconstruct it from the information in stable storage. By "stable storage," we mean a medium that does not lose its values when it loses power. In this dissertation, we assume it means magnetic disks. Other types of stable storage are magnetic tapes and drums, and non-volatile RAM.

As pictured in Figure 1.3, a typical checkpointer periodically interrupts the execution of a program (that we call the *target*), and checkpoints its state to disk. After a failure, the target program's state is restored from the most recent checkpoint. This attempts to minimize the amount of work lost due to the failure.

1.2.1 Uses of Checkpointing

Besides fault-tolerance, checkpointing is a useful tool for other tasks. Some operating systems, like Intel's NX/2, have no facilities for *job-swapping*, and thus are difficult to share, especially with respect to long-running jobs. With a checkpointing tool, one can swap a program off a machine by checkpointing

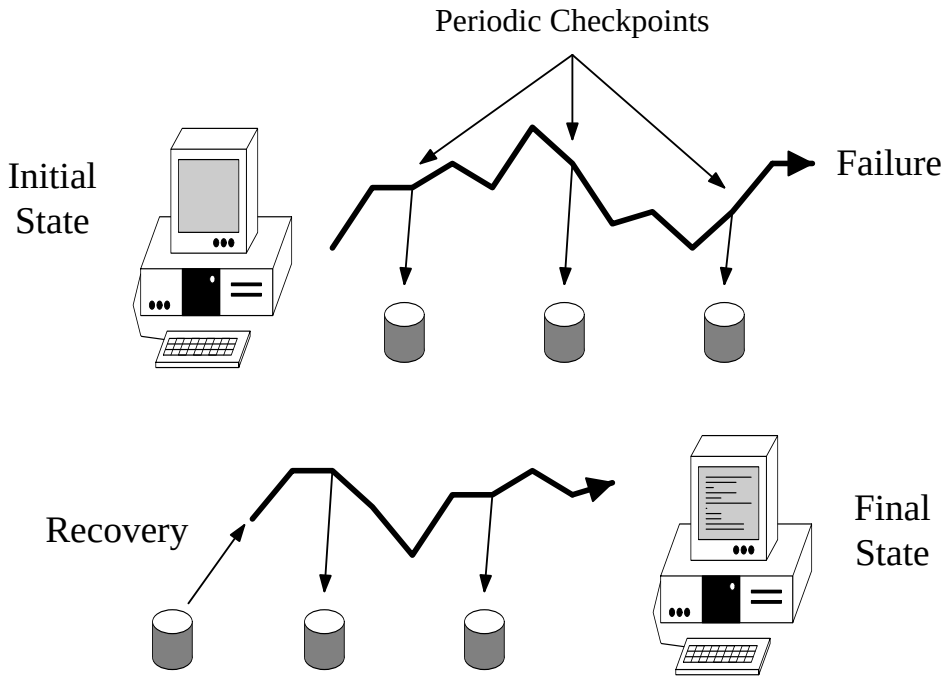

Figure 1.3: Periodic Checkpoints for Fault-Tolerance.

the program, and then killing its process. Swapping on is simply a matter of recovering.

Few operating systems provide generic *process migration*. With a checkpointer, one can migrate a process from one machine to another of the same type by checkpointing on the first machine, and then recovering on the second machine. In fact, checkpointing for fault-tolerance and job-swapping both fall under the category of process-migration: They migrate to the same machine at a different point in time.

An extension of processes migration is *program migration*. The difference is that a program might consist of multiple processes on the same or on different machines. With a checkpointing tool, one can checkpoint all the processes in such a program, and thus migrate the program to a different set of machines, or to a different point in time.

Finally, checkpointing can be useful as a tool for debugging. The information contained in a checkpoint is the same as a typical core file. If several checkpoints are maintained instead of just the most recent one, users can replay their programs back to any point in time to regenerate and discover error conditions. This is called *playback debugging*. Playback debugging is especially useful in parallel and distributed programming, as these programs contain more non-determinism than sequential programs, and thus error conditions may not be regenerated by simply restarting a program from the beginning. Playback debugging is also useful for debugging long-running programs, as users may not have ample time to restart their program and regenerate bugs.

1.2.2 How Checkpoints Are Taken

On a uniprocessor, checkpointing is straightforward. The point at which the target program is interrupted for checkpointing is called the *recovery point*. It is to this point that the program is restored upon recovery. The volatile state of a standard uniprocessor consists of the state of memory, and the state of the machine's registers. Upon checkpointing, enough information must be saved to disk so that upon recovery, the state of the registers and memory can be reconstructed, and the target program can proceed from the recovery point.

Figure 1.4 shows what information is saved in a typical uniprocessor checkpoint. It consists of the registers, and all of a process's address space except for the code. This is because on most machines the code is read-only, and is already saved on disk as part of the target's executable file. None of the operating system's address space is checkpointed: To do so, the checkpointer would have to be part of the operating system—otherwise it would not have permission to read or write the kernel address space. Instead, the checkpointer must save what information it can extract from the kernel's state, and attempt to reconstruct it upon recovery.

For example, on a Unix-like operating system, one can almost make the file system fully recoverable in the following way: At checkpoint time, for each open file, flush the file to disk, and save the current value of its `seek` pointer,

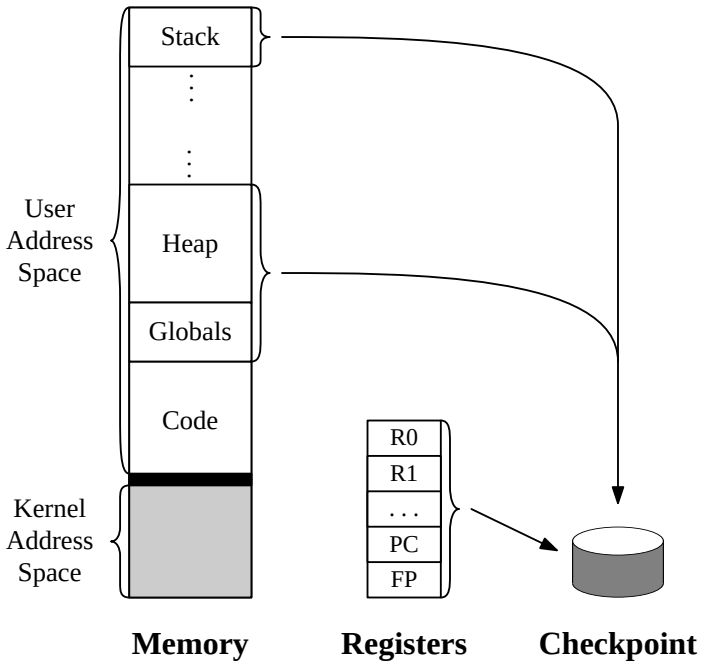

Figure 1.4: A Typical Uniprocessor Checkpoint.

as well as enough information to reopen the file later. Upon recovery, reopen each file and reset its `seek` pointer to the saved value. If the files are not deleted between checkpointing and recovery, and each file is either read-only, write-only, or sequentially written read-write, then all the files are guaranteed to be recoverable. Only randomly-written read-write files are non-recoverable. Figures 1.5 and 1.6 illustrate how this works.

In Figure 1.5, there are two files: File A is write-only, and File B is read-only. When the processor checkpoints, the `seek` pointers are saved with the checkpoint. When the processor fails, the `seek` pointers have been updated, as have the contents of File A. When the processor recovers, the `seek` pointers of Files A and B are reset to their values at the time of the checkpoint. As File B is read-only, its values are unchanged from the time of checkpointing. File A's values have changed, but since the file is write-only, the processor will eventually repeat the writes that have gotten it to its current state. Thus, the files are recoverable.

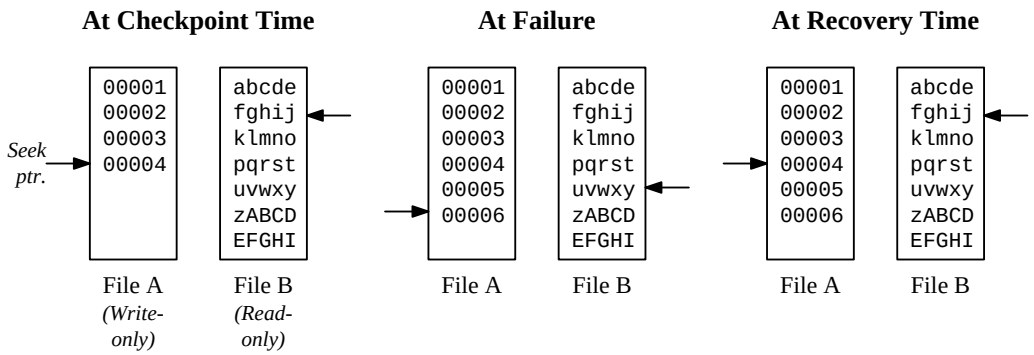

Figure 1.5: Example of Recoverable Files.

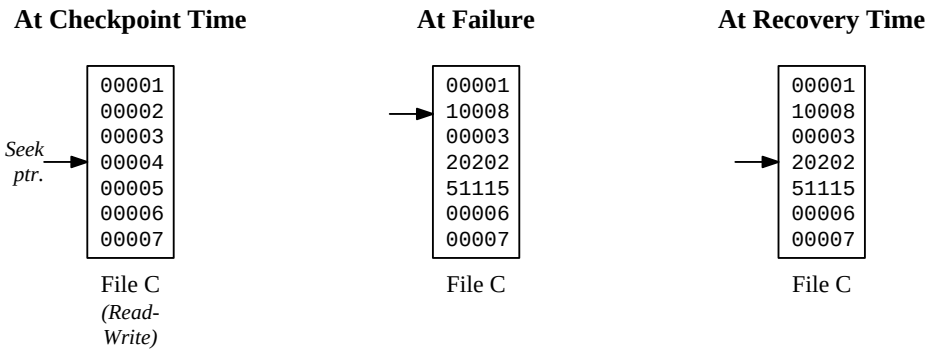

Figure 1.6: Example of Non-recoverable Files.

Figure 1.6 shows an example of a non-recoverable, randomly-written read-write file. After checkpointing and saving the value of the `seek` pointer, the processor skips around in the file, and performs reads and writes. Upon recovery, the file is reopened and the `seek` pointer is reset, but the contents of the file at the `seek` pointer have changed. If the processor reads this value, then it will receive `20202` instead of `00004`. Files of this sort cannot be recoverable by this kind of mechanism.

Other operating systems constructs are also not recoverable. For example, the value returned by the *getpid()* system call might change over time in the presence of checkpointing and recovery. Moreover, sockets and pipes are

non-recoverable. For programs to be checkpointable without making changes to the operating system, they must restrict themselves to be bereft of any reliance on non-recoverable kernel information. We consider such restrictions to be reasonable. Other checkpointers [LS92, EJZ92] have also made these assumptions.

1.3 Speed of Checkpointing

1.3.1 Speed Metrics

One of the most important properties of a checkpointer is its speed: Users are not likely to employ a tool that significantly slows down their programs, even if doing so renders them vulnerable to failures. Thus, when designing a checkpointer, emphasis must be placed on the speed of the checkpointing algorithms. For checkpointing, the measurement of speed can be divided into three metrics: checkpoint time, checkpoint overhead, and latency. Each is explained below.

Checkpoint Time

Checkpoint time is the total time that it takes for the checkpointer to take a checkpoint. It is a measure of how long it takes the checkpointer to run, and also determines how many checkpoints can be taken over a period of time. Checkpoint time is limited by the speed of writing to stable storage. For example, if a checkpoint file is four megabytes long, then the checkpoint time must be at least as long as it takes to write four megabytes to disk. As a rule of thumb, fast disk drives write at a speed of one or two megabytes per second; therefore checkpoints of programs with large writable address spaces must take at least a few seconds to complete.

Checkpoint Overhead

Checkpoint overhead is the total time added to the application program as a result of the checkpoint. Note that this can be less than the checkpoint time if, for example, the writing to stable storage can be overlapped with the execution of the target program. Checkpoint overhead is a measure of how invasive the checkpointer is, and thus is related to the amount of concurrency that a checkpointing algorithm has. Specifically, we define concurrency as:

$$concurrency = \left(1 - \frac{overhead}{checkpoint\ time}\right) * 100\%.$$

One of the goals of our checkpointing algorithms is to maximize concurrency, and therefore decrease the checkpoint overhead as much as possible.

Latency

Latency is a measure of real-time behavior. We define it to be the maximum amount of time that the target program is interrupted as a result of the checkpointer. Decreasing latency is important, especially if the target program is interactive. A checkpointer will be successful if users do not perceive it as freezing their program for large chunks of time. Another goal of our algorithms, especially those concerning multiprocessor workstations where a user is more likely to be at a console watching his or her program, is that the latency be small—the checkpointer should only interrupt the target for small, fixed periods of time, such as a tenth of a second.

1.3.2 Incremental vs. Non-incremental Checkpoints

The best way for a checkpointer to reduce its checkpoint time is to reduce the amount of information saved in a checkpoint. One way to reduce this amount was suggested by Feldman and Brown for playback debugging [FB89]. Instead of taking a complete checkpoint at periodic intervals, they take *incremental checkpoints*. That is, only the state that has changed since the previous checkpoint is recorded. To recover from an incremental checkpoint, one must

recreate the recovery point from a number of incremental checkpoints, rather than just the most recent one. The assumption made with incremental checkpointing is that programs tend to exhibit locality of reference, and that the state changed between checkpoints is a small fraction of the total volatile state of the program.

In our algorithms, we take full instead of incremental checkpoints. The reason is that we want to test the worst-case behavior of our checkpointing algorithms. The work involved in taking an incremental checkpoint is nearly a subset of the work involved in taking a full checkpoint (some extra work must be performed to figure out which pages to checkpoint). Therefore if the algorithms perform well according to the speed metrics with non-incremental checkpointing, then they should perform even better with incremental checkpointing. The incremental checkpointer by Elnozahy, Johnson and Zwaenepoel confirms this claim [EJZ92].

1.4 Related Work

1.4.1 Database and Uniprocessor Checkpointing

The bulk of work on and implementations of checkpointing for fault tolerance has been in database and transaction-processing systems [DKO+84, Pu85, Hag86, SGM87, LN88]. Although these are a major class of applications, the fact that database computations can be viewed as atomic transactions greatly simplifies the act of checkpointing. We concentrate on general purpose programs, so no such computational model can be assumed.

General-purpose checkpointing has been studied and implemented on uniprocessors. Proposals and overviews have been described [Ran75, AL81, Lam81], and implementations for playback debugging have been provided [FB89, PL88, Wit89]. As mentioned above, the debugger Igor [FB89] performs incremental checkpointing. It uses virtual memory management to de-

tect changed pages and reconstructs an execution state from a number of snapshots.

Of special note is a system called *Condor* [LS92], which is a checkpointing facility that runs on numerous commercial uniprocessors. Condor's focus is on process-migration and job-swapping for batch computing, rather than on fault-tolerance. Its mechanism for checkpointing is to take a core-dump at each recovery point. This way their code is easily portable to a number of different machines (although the checkpoints are not). Condor runs completely in the user's address space, and imposes the same restrictions on checkpointable programs as outlined in Section 1.2.2 above.

There has been research in making file and operating systems recoverable. Taylor and Wright describe a backward recovery implementation that focuses on a file system for Unix that is fully recoverable [TW86]. This removes the restrictions on randomly written read-write files described in Section 1.2.2. Some operating systems, such as Sprite [OCD+88], Accent [FR86], and the V System [TLC85] include the ability to migrate processes as one of their major design goals. In such a system, checkpointing should be fully supportable with no restrictions, as checkpointing is a process migration to stable storage.

Finally, there have been some hardware designs that would facilitate checkpointing. Staknis proposed a new memory design called sheaved memory [Sta89] for supporting checkpointing in paged systems. In a sheaved memory, physical page frames can be bundled together so that data written to one frame in the bundle is simultaneously written to all frames in the bundle. If the entire address space consists of bundles of more than two pages each, then one may checkpoint the address space by removing one frame from each bundle.

The price of non-volatile memory is decreasing to the point that within the next ten years, it might be affordably incorporated into workstation architectures [BAD+92]. Such memories can greatly improve checkpointing speeds as they do not have the large write latency of disks. However, like sheaved memory, non-volatile memories are currently expensive to build, and not yet widespread enough to warrant further consideration.

1.4.2 Multiprocessor Checkpointing

There has been little explicit work published on checkpointing multiprocessors. As we will explain in Chapters 2 and 3, multiprocessor checkpointing is very similar to uniprocessor checkpointing, except that there is more parallelism that can be used to decrease the overhead of checkpointing. Using a multiprocessor in this way is suggested by Feldman and Brown [FB89] in terms of checkpointing for playback debugging. Moreover, the hardware designs for checkpointing mentioned above would also greatly facilitate multiprocessor checkpointing.

1.4.3 Distributed Checkpointing Algorithms

Multicomputers bear a great deal of similarity to distributed systems, and thus the vast amount of research on distributed checkpointing algorithms is relevant to checkpointing multicomputers. Checkpointing algorithms for distributed computing fall into four general categories. First are algorithms for distributed database checkpointing. As stated above, these rely on special properties of database computations, and are of little use for general-purpose computations. Second are *pessimistic* algorithms. These are algorithms that assume frequent failures of the system, and thus take strict precautions to ensure the reliability of computations. Pessimistic algorithms tend to impose heavy penalties on failure-free performance for the sake of reliability. *Optimistic* checkpointing algorithms are the opposite. They assume that failures occur relatively infrequently, and therefore attempt to push most of the overhead of checkpointing into the recovery process. The last type of checkpointing algorithms are called *consistent* checkpoints. Consistent checkpointers try to strike a balance between optimistic and pessimistic checkpointers. In other words, they attempt to provide good failure-free performance, yet simple recovery. Consistent checkpointers tend to be best suited for multicomputer checkpointing.

Pessimistic Checkpointing Algorithms

In a pessimistic checkpointing scheme, each process is backed up by a *mirror* process. All messages in the system are sent to three locations: the specified destination process, the destination's mirror, and the sender's mirror. The system must guarantee that this operation be atomic. In other words, if a process receives a message m_1 before receiving m_2, then its mirror should receive those two messages in that order as well. If a process in the distributed system fails, then its mirror takes over (starting another mirror process to back itself up). It first must catch up to the state at which the failing processor failed, which it does using messages it has saved. Then it continues as normal. Implementations of such schemes have been written [BBG83, PP83, BBG+89].

One of the problems with this simple form of checkpointing is that the mirror process has to restart its computation from the beginning until it catches up with the others. While this can be tolerated in a distributed environment, where the processes that are waiting for the mirror to catch up can be swapped off their machines, in a multicomputer environment it is much more wasteful. For example, on the Intel iPSC/860, users must allocate a portion of the processors for exclusive use. If $n-1$ of them wait while one catches up, then $n-1$ processors end up doing nothing for that period of time. The recovery takes just as long as doing the entire computation from the beginning, minus the message traffic. In the TARGON/32 system, the nodes periodically checkpoint themselves to prevent this problem of having one node restart from the beginning [BBG+89].

Another problem with this kind of checkpointing on some multicomputers is that with no job-swapping or multiprogramming, entire processors must be assigned as mirrors. This effectively cuts the size of the machine in half. Finally, the requirement that messages be transmitted to three places atomically is unreasonable for some machines without rewriting the operating system. In TARGON/32, extra hardware was added to facilitate this atomic transmission [BBG+89].

Optimistic Checkpointing Algorithms

In optimistic checkpointing algorithms [SY85, JZ89, Joh89], processors independently checkpoint themselves and maintain a log of the messages that they have received. They also maintain some information about the state of the processors that send them messages. This information is usually "piggybacked" on top of the messages themselves. When a processor fails, then all processors cooperate to decide to which state the failed processor should restore itself, and to which state they should restore themselves. This is decided by a complex distributed algorithm, after which processors may have to restore themselves to saved checkpoints, and then replay their computations using messages from their message logs. This is all to get the system to a plausible state that either did occur previously, or that could have occurred previously according to the information saved in the message logs.

One of the strengths of the optimistic algorithms, especially in widely dispersed distributed systems, is that during failure-free operation, they allow a large degree of autonomy. That is, processors can checkpoint and log messages whenever they want: They are not affected by the checkpointing and logging of other processors. This autonomy does not come for free—it is a source of much of the complexity in the recovery algorithm. On a multicomputer, autonomy is not quite as important as in a distributed system. For example, in a distributed system, processors may be separated by thousands of miles. Single-processor failures and partitions are likely to occur and do not affect the processing ability of the other processors. In a multicomputer, all the nodes are physically close, and more often than not, a single-processor or power failure brings down the entire machine. It makes sense for the processors to give up autonomy and cooperate more tightly than in optimistic checkpointing.

Finally, the recovery mechanisms for optimistic checkpointing are not useful for job-swapping. Rolling back and replaying can be time-consuming activities. Existing optimistic recovery methods are not appropriate for fast job swapping on multicomputers.

Consistent Checkpointing Algorithms

All consistent checkpointing algorithms stem from a paper by Chandy and Lamport [CL85]. In these algorithms, all processors cooperate at checkpoint time to define a *consistent state*, and write it to disk. Recovery is simple—all processors read their states from disk, and the computation may proceed. Consistent checkpointing seems to be the best for multicomputer checkpointing, as it does not overly penalize failure-free performance, and recovery is simple and fast enough to be used for general-purpose job-swapping. The main drawback of consistent checkpointing is that it requires all processors to cooperate at checkpoint time. This is not a big problem for multicomputers, where the processors are more tightly coupled than in general distributed systems.

There has been a great deal of research on consistent checkpointing algorithms [SK86, LY87, KT87, Ahu89, CT90, CJ91, KMBT91, LNP92]. Much of this work is discussed in Chapter 2, where we explain consistent checkpointing in detail, outline algorithms from the literature that are relevant to multicomputer checkpointing, and then describe the "Network Sweeping" algorithm, which is a new consistent checkpointing algorithm specifically designed for multicomputers.

1.4.4 Other Related Work

Many of the algorithms and optimizations described in the next two chapters stem from work in areas other than fault-tolerance and checkpointing. The Sync-and-Stop algorithm described below is an application of the two phase commit protocol [KS86] to distributed checkpointing. The latency-reducing optimizations of Chapter 3 were motivated by copy-on-write [TLC85, FR86], as well as shared virtual memory [LH89] and real-time, concurrent garbage collection [AEL88]. The compression algorithms mentioned in Chapter 3 are modified versions of those employed in a study of compressed log-structured file systems [BJLM92].

1.5 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 describes basic algorithms for checkpointing. One algorithm is given for multiprocessors, and three are described for multicomputers. Two of these three are new algorithms. Chapter 3 presents optimizations for the algorithms to improve the speed metrics of checkpoint time, overhead and latency. Chapter 4 presents the results of two checkpointing implementations, one on a multiprocessor, and the other on a multicomputer. All algorithms from Chapter 2 and all optimizations from Chapter 3 have been implemented and the results are discussed. Conclusions and future work are presented in Chapter 5.

Parts of this dissertation have been published previously. The initial portions of Chapters 2 through 4 were presented in [LNP90]. The rest of Chapter 2 has been published in [LNP91] and [LNP92].

Chapter 2

Checkpointing Algorithms

This chapter presents basic algorithms for checkpointing and restarting parallel programs on MIMD architectures. Combined with the optimizations in Chapter 3 they give a spectrum of efficient algorithms for checkpointing these machines. First, we describe an algorithm for checkpointing multiprocessors called "sequential checkpointing." Next, we describe consistent checkpointing in detail and present three consistent checkpointing algorithms for multicomputers. Two of these three algorithms are new. The last algorithm, called Network Sweeping, uses properties of wormhole routing networks present in all of today's multicomputers, to achieve superior performance. This algorithm is discussed in detail.

2.1 Checkpointing Multiprocessors

The only difference between checkpointing a uniprocessor and checkpointing a multiprocessor is that there are several CPU states to save to disk instead of just one. To establish a recovery point for a multiprocessor, the CPU states saved in the checkpoint should all be valid with respect to the state of memory saved in the checkpoint. That is, the processors all need to synchronize with each other at the memory boundary. Otherwise the checkpointer runs the risk of saving an invalid recovery point to disk.

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.

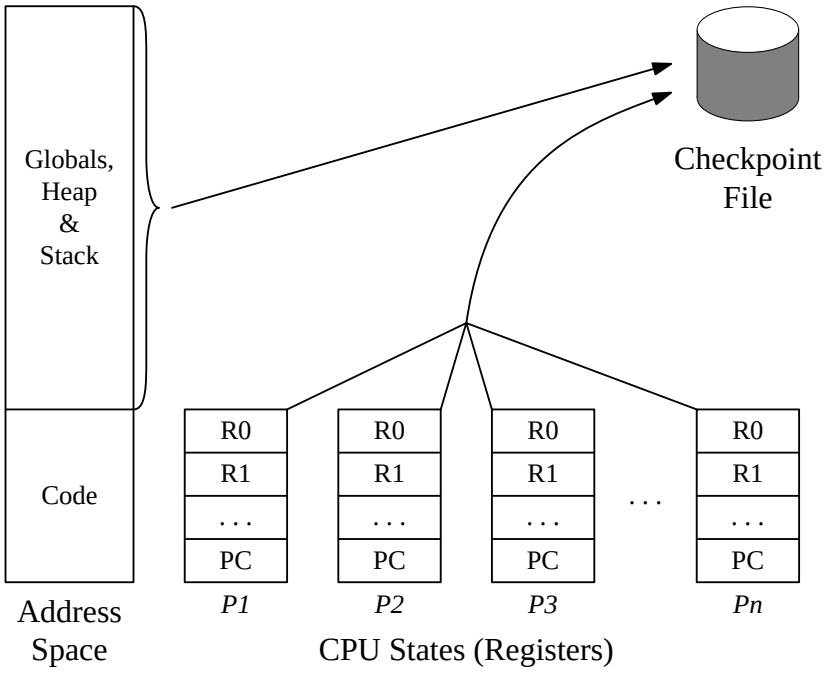

Figure 2.1: Sequential Checkpointing Algorithm for Multiprocessors.

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*.

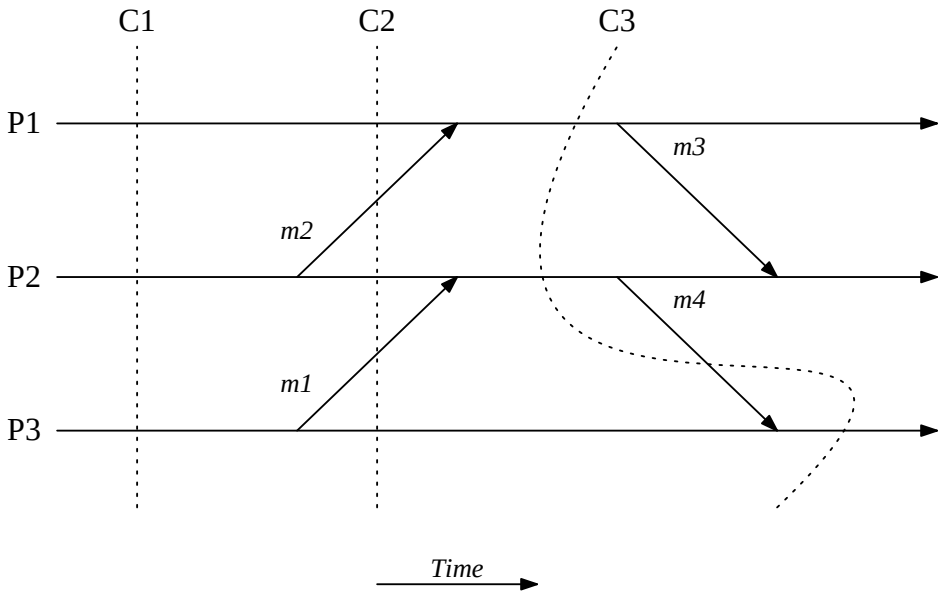

Figure 2.2: A Sample Distributed System.

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

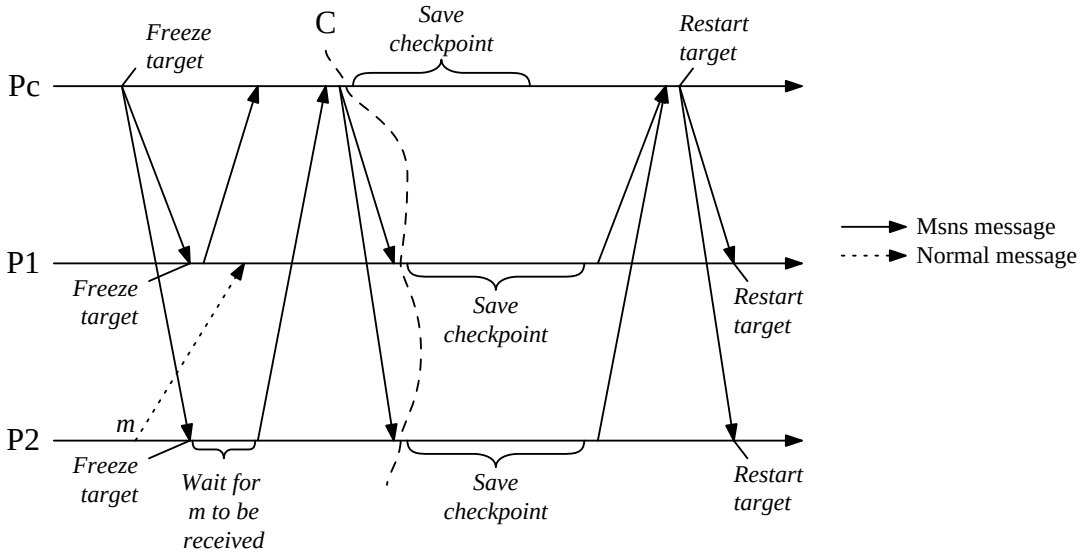

Figure 2.3: The Sync-and-Stop algorithm at work.

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. □

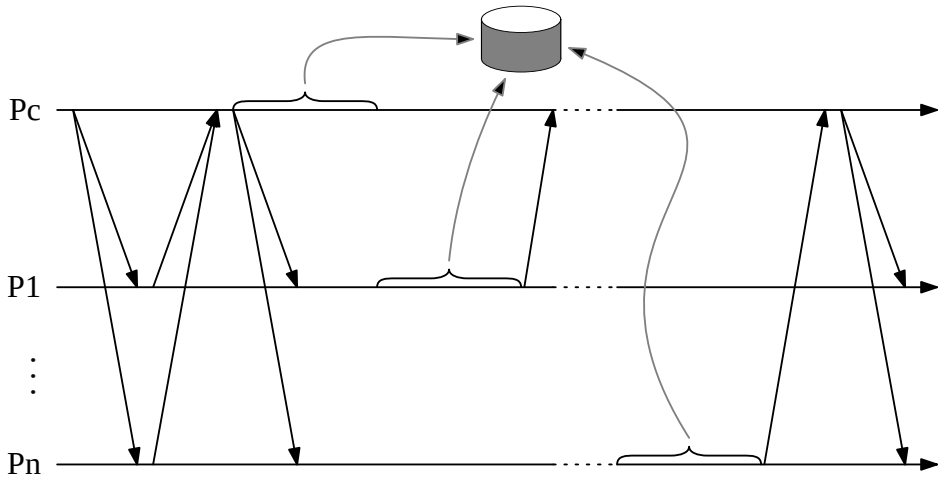

Figure 2.4: The Sync-and-Stop algorithm serializing at the disk.

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.

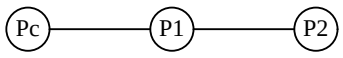

Figure 2.5: A simple distributed system.

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.

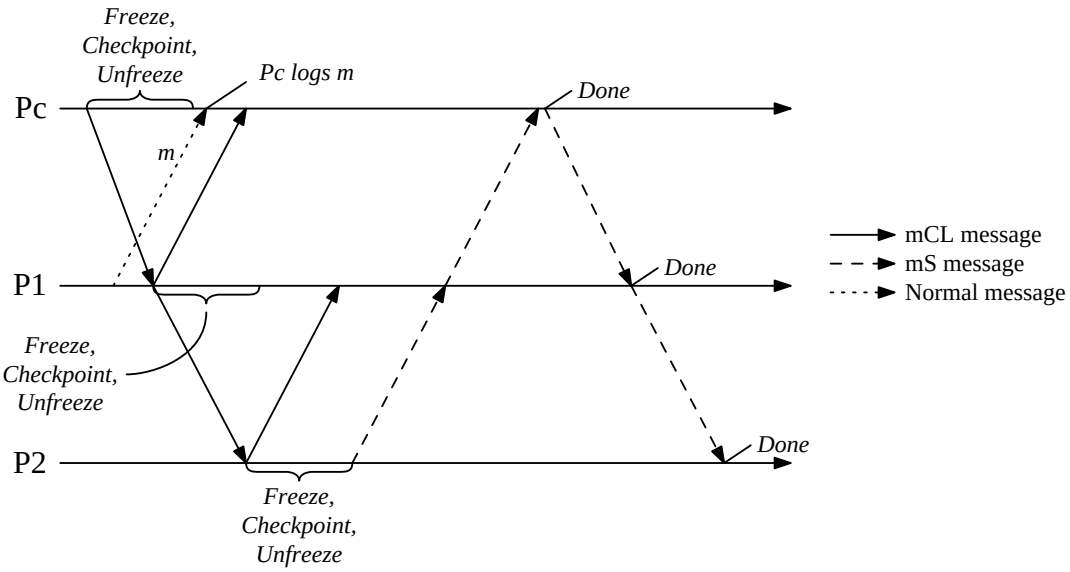

Figure 2.6: The CL Algorithm at work.

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.

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 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 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 (($i+1$) mod 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 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:

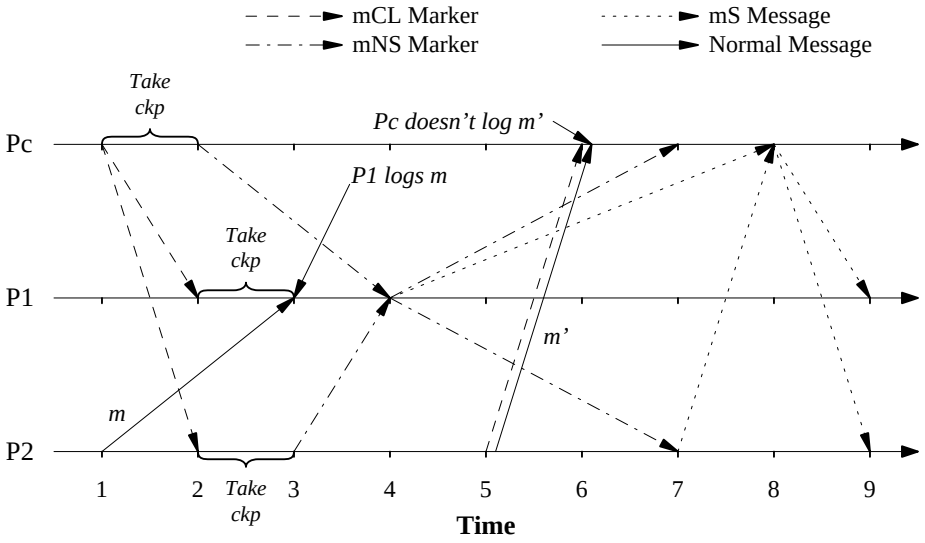

Figure 2.7: Time line of Example 1.

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. □

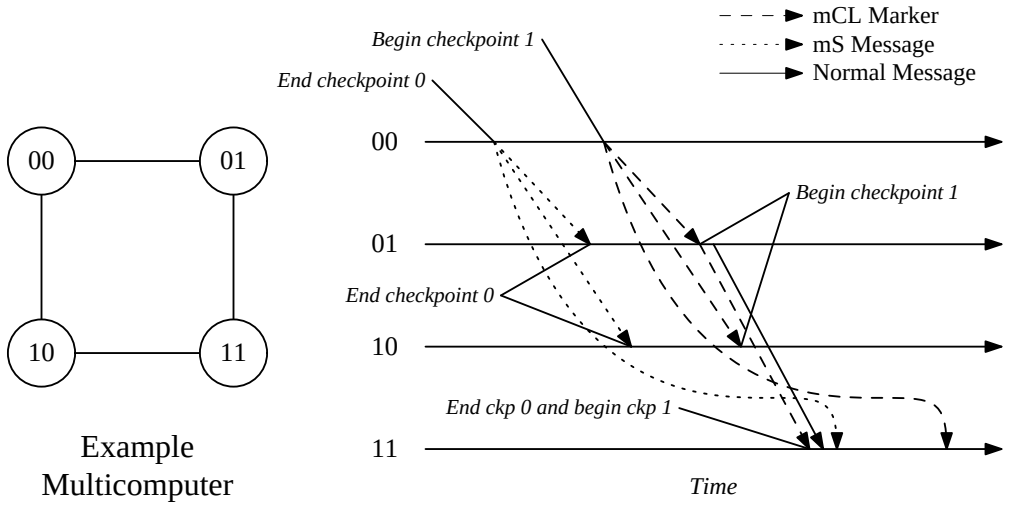

Figure 2.8: Time line of Example 2.

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.

2.6.2 Sending m_{NS} Messages on Wormhole Routing Interconnection 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 do not state specifically how p makes this determination. The example in Figure 2.7 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 [DS87].

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 [LLG+90], and MIT's J-Machine [Dal91b]. 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 [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 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 is non-adaptive because it always yields the same path from one node to another, and it is memoryless because the function works from channel to channel, retaining no knowledge of the source node once it is 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 theorems prove the method's correctness:

Theorem 2.6.5 *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. □

Theorem 2.6.6 *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' (as p_2 does to p_1 in Figure 2.7). 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} will not be received on c_k until after m has been put on c_{k+1}. Thus, m_{NS} will not 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. □

Hypercube Interconnections

The Intel iPSC/2 and iPSC/860 [Nug88] 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 2.9 shows a 3-dimensional hypercube with its links.

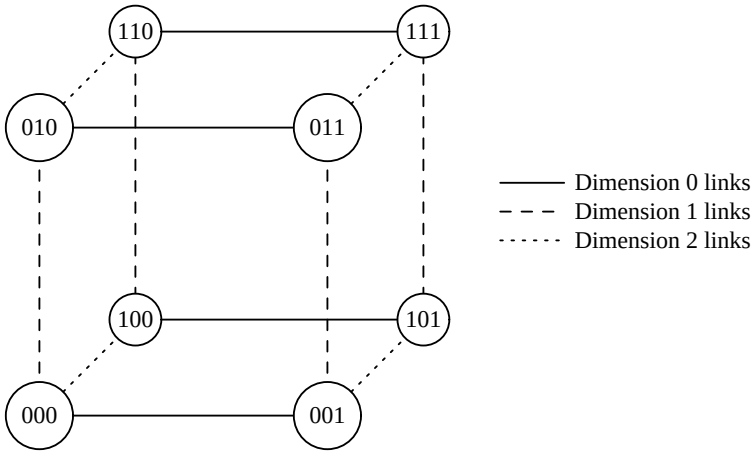

Figure 2.9: A 3-dimensional hypercube

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 2.9, 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 2.9, $f((000, 001), 111) = (001, 011)$.

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 Delta and 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:

$$N = \{n_{ij} \mid i < R \text{ and } j < C\}$$

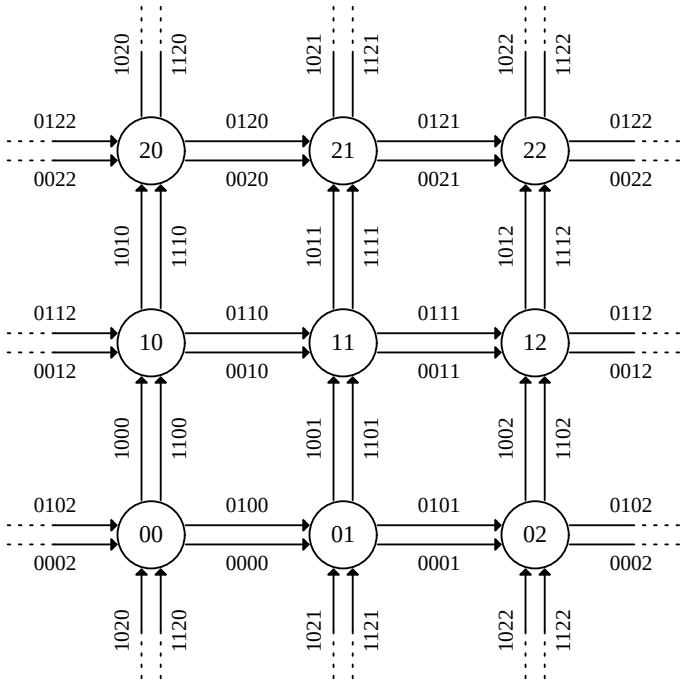

Figure 2.10: A 3×3 toroidal mesh

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

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 is a high channel, 0 if it is a low channel.
- r = row number of its source node.
- c = column number of its source node.

Figure 2.10 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 as Dally has shown that multicomputers built using low-dimensional k-ary d-cubes for their interconnection networks give optimally low latency and high throughput [Dal90]. Accordingly, many new multicomputers, such as the Paragon, Stanford's DASH [LLG+90], the J-Machine [Dal91b], the Ametek 2010 [Sei88a] and the Mosaic [Sei88b], 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 do not include the details here, but they follow as extensions of the mesh example above.

Finally, Dally has proposed *Express Cubes* as an optimization to standard k-ary d-cubes [Dal91a]. 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.

2.6.3 Optimizations

The Network Sweeping algorithm imposes an overhead of $O(l(n) + pp(n))$ extra messages. As stated above, $pp(n)$ 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 Venkatesan's algorithm [Ven89], 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 one in Lai and Yang [LY87]), 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 the 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.

2.7 Recovery

Recovery is the same in all consistent checkpointing algorithms: Each processor reads its processor 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 that 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 communi-

cation between processors, and no reading of other processors' message logs. This latter form is used in our multicomputer implementation.

Chapter 3

Optimizations

This chapter outlines several optimizations that can be made to the basic algorithms presented in Chapter 2. These optimizations are meant to improve the algorithms' performance on the three speed metrics described in the introduction: checkpoint time, overhead, and latency.

While the optimizations themselves are quite different, they have similar approaches with respect to improving each metric:

- **Checkpoint time:** The best way to reduce checkpoint time is to decrease the amount of data written to disk. As mentioned in the introduction, incremental checkpointing has the potential to improve the checkpoint time in this way, as does compression, mentioned in Section 3.5 below. On the other hand, checkpoint time can be adversely affected by the checkpointer performing excess work for other optimizations. As we shall see in Chapter 4, there can be situations where the main memory and copy-on-write algorithms described below fall into this trap.

- **Checkpoint overhead:** To reduce the overhead of checkpointing, one must increase the concurrency. Thus, optimizations for this metric attempt to overlap as much checkpointing as possible with the execution of the target program. Main memory checkpointing, outlined in Section 3.1 below, is an example of this technique. Another source of overhead is

serialization of the processors in the checkpointing algorithm. Finding bottlenecks and eliminating them is a second technique of reducing overhead.

- **Latency:** The main way to reduce latency is to break up the checkpoint into small pieces, and then to perform work on a piece at a time only when it is absolutely necessary. We employ copy-on-write to perform this breaking up at the page level. Sections 3.2 and 3.3 detail how copy-on-write is incorporated into checkpointing.

The remaining sections of this chapter outline individual optimizations for checkpointing. Each optimization has been implemented either for multiprocessors or multicomputers. The results are given in Chapter 4.

3.1 Main Memory Checkpointing

Main memory checkpointing is a simple way of increasing the concurrency of checkpointing by overlapping the execution of the target program with the writing of the checkpoint to disk. We first explain it in terms of multiprocessor checkpointing. In the sequential multiprocessor checkpointing algorithm outlined in Section 2.1, the processors are all frozen while the checkpoint is written to disk. This is wasteful as writing the checkpoint to disk is time consuming, and on most multiprocessors it is a sequential activity (for example, the Firefly has one I/O processor through which all disk traffic must pass). Thus, a solution is to copy the checkpoint to another location in main memory while the processors are frozen, and then write the copy to disk once the processors are unfrozen. The algorithm is illustrated in Figure 3.1, and outlined below:

- Freeze all the processors.
- Allocate a second address space in main memory.
- Copy the state of the processors to this second address space.

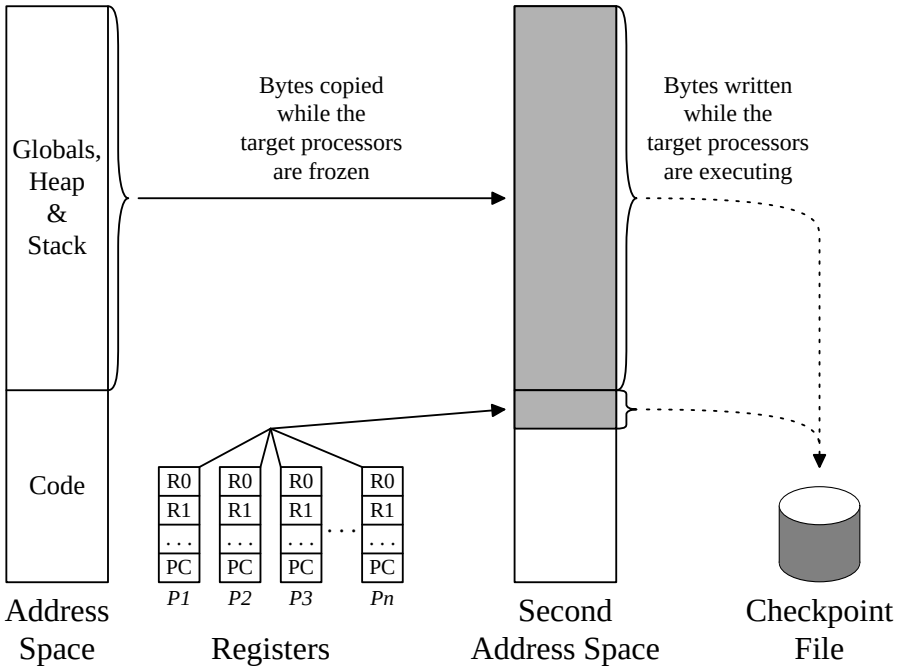

Figure 3.1: Main Memory Checkpointing for Multiprocessors.

- Copy the state of the memory to the second address space.
- Unfreeze the processors.
- Start up a new program thread, which writes the contents of the second address space to disk.

The checkpoint produced by this algorithm is the same as in the sequential algorithm described in Section 2.1. However, the values of the speed metrics should be different. The checkpoint time should increase: It should be equal to the sequential checkpoint time plus the time to make the memory-to-memory copy of the address space. The checkpoint overhead, however, should decrease from being equal to the checkpoint time to being equal to the memory-to-memory copy time. Finally, the latency should decrease to equal the checkpoint overhead.

This optimization also applies to multicomputer checkpointing. It is performed by each processor when it takes a local checkpoint: Instead of writing

the checkpoint directly to disk, a copy is made in main memory, and then after the target is restarted, the copy is written to disk. This should reduce the overhead of checkpointing significantly for all three algorithms described in Chapter 2 for the same reason as in the multiprocessor case: The processors are not frozen during the time-consuming activity of writing to disk.

One of the underlying assumptions of this optimization is that there actually be enough physical memory to hold the main memory checkpoint. If not, and the machine has a backing store, then the optimization will force pages of memory to be swapped to disk. This will degrade both the checkpoint time and the overhead. If the machine does not support paged virtual memory, then the optimization must be modified so that the part of the main memory checkpoint that does not fit in physical memory gets written directly to disk while the processors are frozen. This should not have such an adverse effect on the checkpoint time, but it will not reduce the checkpoint overhead as much as if the entire checkpoint does fit into main memory. Section 3.3 discusses this issue further.

3.2 Copy-on-write Checkpointing

The main memory optimization should decrease both the overhead and latency of checkpointing. However, it still might take on the order of seconds to make an in-memory copy of the address space. This next optimization attempts to improve the latency of the main memory optimization by using *copy-on-write* [TLC85, FR86] to make the main memory checkpoint. Copy-on-write is a technique that uses a processor's virtual memory hardware to make a memory-to-memory copy with low latency. For the multiprocessor case, a copy-on-write checkpointing algorithm works as follows:

First, it freezes the processors and copies their registers to the second address space. Also at this time, it sets the page protection bits of all pages in main memory to be "read-only." Next, it unfreezes the processors, and starts a separate thread of execution, which we call the *"copier thread"*, that

copies pages to the new address space and resets the pages' protection to "read-write." If a thread of the target program generates a page access violation, then it must copy that page to the new address space before setting its protection to "read-write" and restarting. When the copier thread is done copying, the main memory checkpoint is complete, and the copier thread writes the checkpoint to disk. This algorithm is illustrated in Figure 3.2.

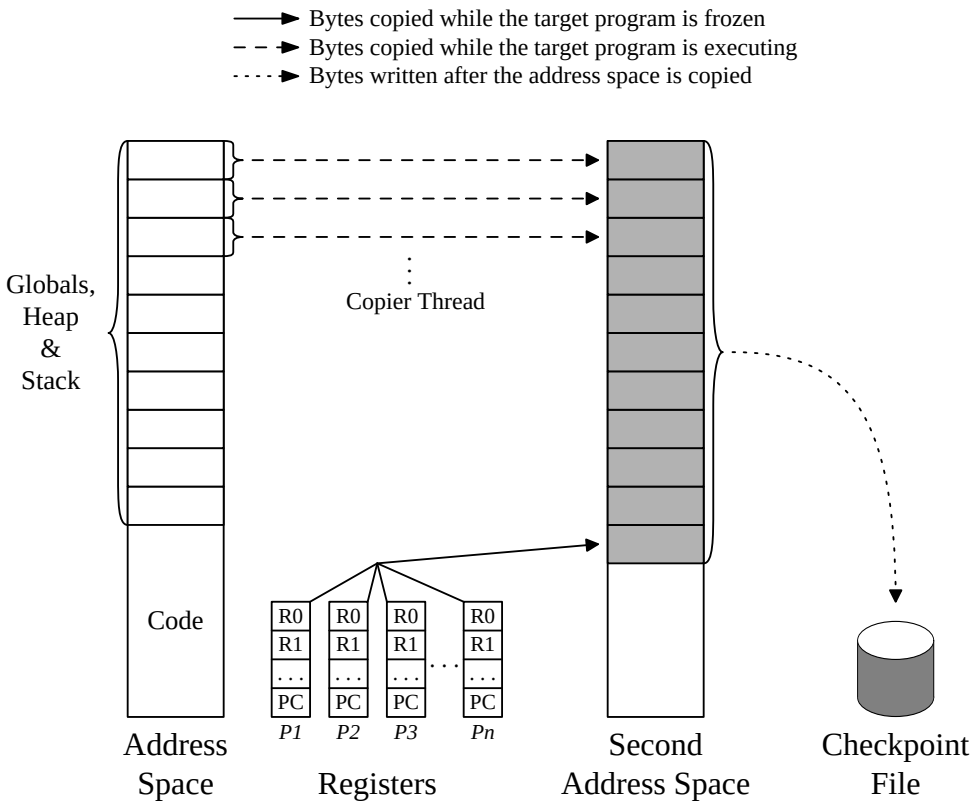

Figure 3.2: Copy-on-write Checkpointing for Multiprocessors.

The multicomputer version of this algorithm is similar: Each processor performs copy-on-write to make the main memory copy of its local checkpoint.

The copy-on-write algorithms produce the same checkpoint files as the main memory algorithms. Moreover, the checkpoint time should be further increased, as there is an extra action in protecting the address space, and the

memory-to-memory copy is done in chunks, rather than all at once. In the multiprocessor case, the checkpoint overhead should decrease once more, because some of the target program's processors are running while the copier thread copies pages, as opposed to the main memory algorithm in which all processors are halted while the address space is copied. The latency of these algorithms is equal to the maximum of the time it takes to copy the registers and protect the address space, and the largest time to process a page fault. Both of these should be significantly smaller than the main memory optimization's latency, which is the time to copy the entire address space.

One variable of this optimization that can have a major effect on the latency is the page size of virtual memory. Although this value is fixed by a machine, one can emulate larger page sizes by increasing the number of bytes that are copied to the second address space by the copier thread and by the page fault handler. Larger pages should increase the time to handle a fault, thereby increasing latency, but they will also decrease the number of faults, and because of locality of reference, they may also decrease the rapidity at which faults occur.

3.3 The CLL Optimization

The copy-on-write optimization can be improved further by adding buffering, a standard operating systems technique usually used to hide disk latency during file system writes [SP88]. What the buffering does in this case is allow the checkpoint to be written to disk at the same time as it is being copied from memory. We call this the *CLL* optimization, which stands for "Concurrent, Low-Latency." For multiprocessors, the CLL optimization works as follows: It first allocates a fixed-size pool of pages (the buffer) in the second address space that the copier thread and page fault handlers fill, and that a new thread called the *"writer thread"* empties by writing the pages to disk. The writer and copier threads are both started immediately after the processors are unfrozen. The algorithm is illustrated in Figure 3.3.

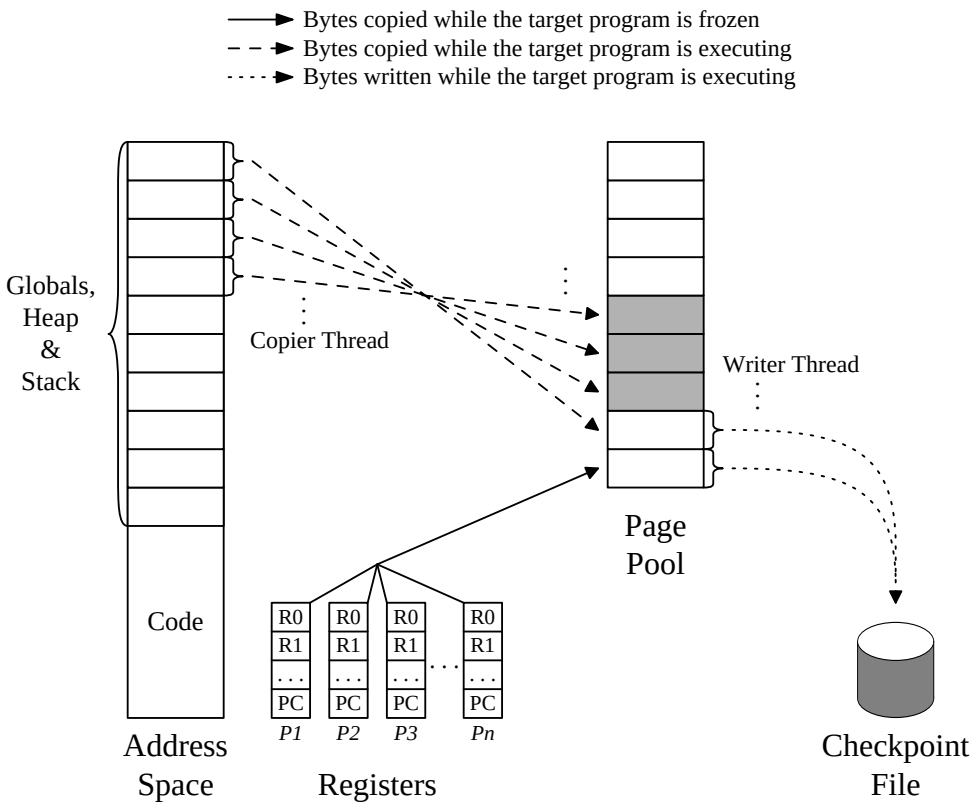

Figure 3.3: The CLL Optimization for Multiprocessors.

As in the copy-on-write optimization, one can apply the CLL optimization to multicomputers by having each processor allocate, fill, and empty its own buffer for its own local checkpoint.

The improvements of the CLL optimization over the copy-on-write optimization should be twofold. First, the extra memory requirements of this scheme are fixed—they are the size of the page pool. The copy-on-write scheme needs extra space that is the same size as the target program's address space, which is more likely to overflow physical memory and cause thrashing. Second, the overall checkpoint time of a CLL checkpointer should be less than that of a copy-on-write checkpointer. This is because the CLL optimization starts

writing to disk as soon as the processors are restarted, instead of waiting until a complete main memory checkpoint has been made.

A possible concern of the CLL optimization is what happens when the page pool fills up. Then pages in the pool are freed only as fast as they can be written to disk. If the size of the page pool is chosen to be the amount of available physical memory, then the CLL optimization should outperform the copy-on-write optimization for the following reason: In the copy-on-write case, pages might be swapped to the swap area on disk so that the checkpoint can fit into main memory. If these pages are part of the checkpoint, then they must be swapped back into memory to be written to the checkpoint file. If they are part of the target program, then they will eventually have to be swapped back into memory when the program needs them. In either case, the copy-on-write scheme performs an extra disk write and read for each swapped page, while the CLL scheme needs no swapping, and therefore performs no extra disk writes. This extreme case for both optimizations is tested in our implementation.

3.4 Staggering Writes

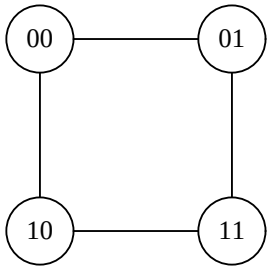

Figure 3.4: Simple multicomputer with Hypercube Routing.

One can change the behavior of the Chandy-Lamport and Network Sweeping algorithms significantly by altering the time at which the m_{CL} messages are sent. For example, in the Chandy-Lamport algorithm, processors can broadcast their m_{CL} messages immediately after taking their local checkpoint instead of immediately before. Similarly, in the Network Sweeping algorithm, the co-

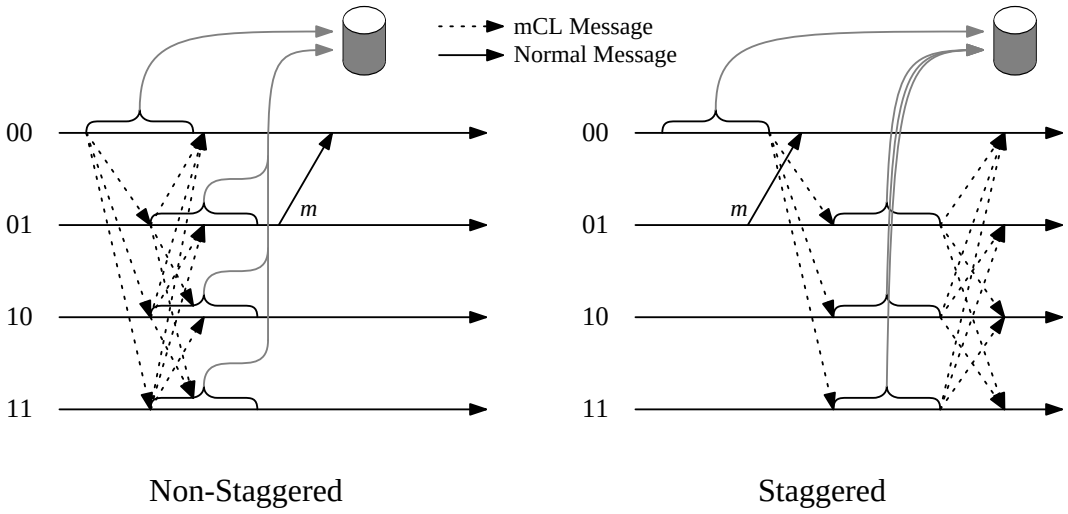

Figure 3.5: Two versions of Chandy-Lamport Checkpointing.

ordinating processor p_c need not broadcast m_{CL} messages immediately before taking its local checkpoint. Instead, the other processors can be notified to start checkpointing by the m_{NS} messages and the other m_{CL} messages.

The result of these changes is that checkpointing is spread out over a larger period of time. In the original versions of the algorithms, p_c broadcasts to all processors to start checkpointing as soon as it starts checkpointing. The result is that all the processors start checkpointing at roughly the same time. With these changes, the processors start checkpointing at different times: The checkpoints are more staggered over time. Thus, we call these algorithms *staggered*, and the original versions *non-staggered*.

Figure 3.5 shows an example of non-staggered and staggered versions of the Chandy-Lamport algorithm on the example multicomputer of Figure 3.4 (which we assume to use the e-cube routing function described in Section 2.6.2). Similarly, Figure 3.6 shows an example of the staggered and non-staggered versions of the Network Sweeping algorithm on the same multicomputer (m_S messages are omitted in both figures for clarity). In the Chandy-Lamport example, there is one regular message m in the system, which gets logged in the staggered version of the algorithm, but not in the non-staggered version. In

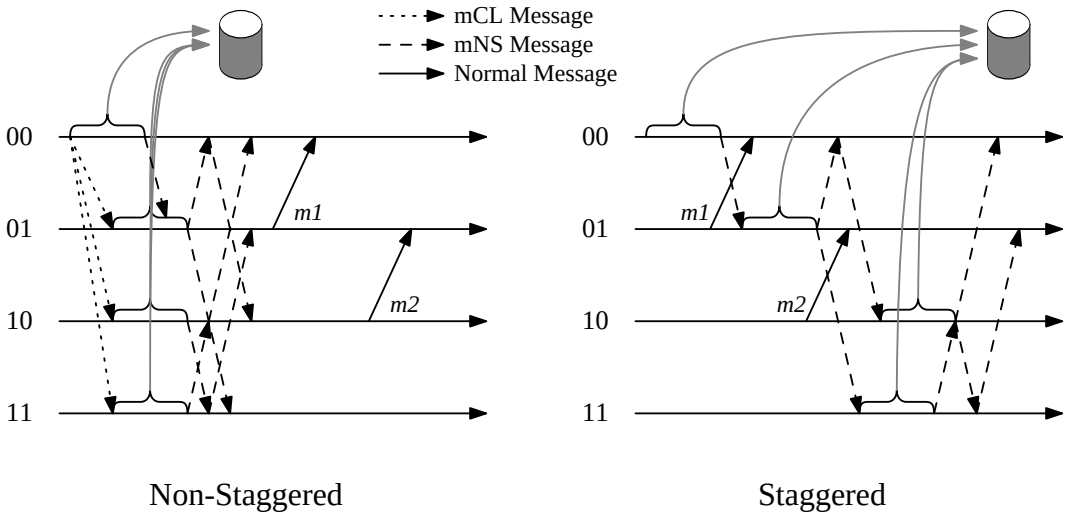

Figure 3.6: Two versions of Network Sweep Checkpointing.

the Network Sweeping example, there are two regular messages $m1$ and $m2$, both of which get logged in the staggered version of the algorithm, but not in the non-staggered version.

These figures demonstrate one of the differences between the staggered and non-staggered algorithms: Since the staggered ones spread the local checkpoints out over a period of time, they are more likely to cause cross-cut messages, which must be logged. If these messages are large, then the checkpointer must perform extra work writing the contents of the messages to disk, which means extra time and overhead for the checkpointer. Thus, in cases where the target program sends large messages, we anticipate that non-staggered checkpointing will perform better than staggered checkpointing.

Another time when non-staggered checkpointing might perform better than staggered checkpointing is when the target program does a lot of synchronizing. Consider the example in Figure 3.7. Here there are two processors that communicate in pairs: Whenever one sends a message to the other, the other acknowledges it with a second message. The first processor stalls until it receives the acknowledgement. In normal operation, neither processor spends much time waiting for acknowledgements. However, when the system is check-

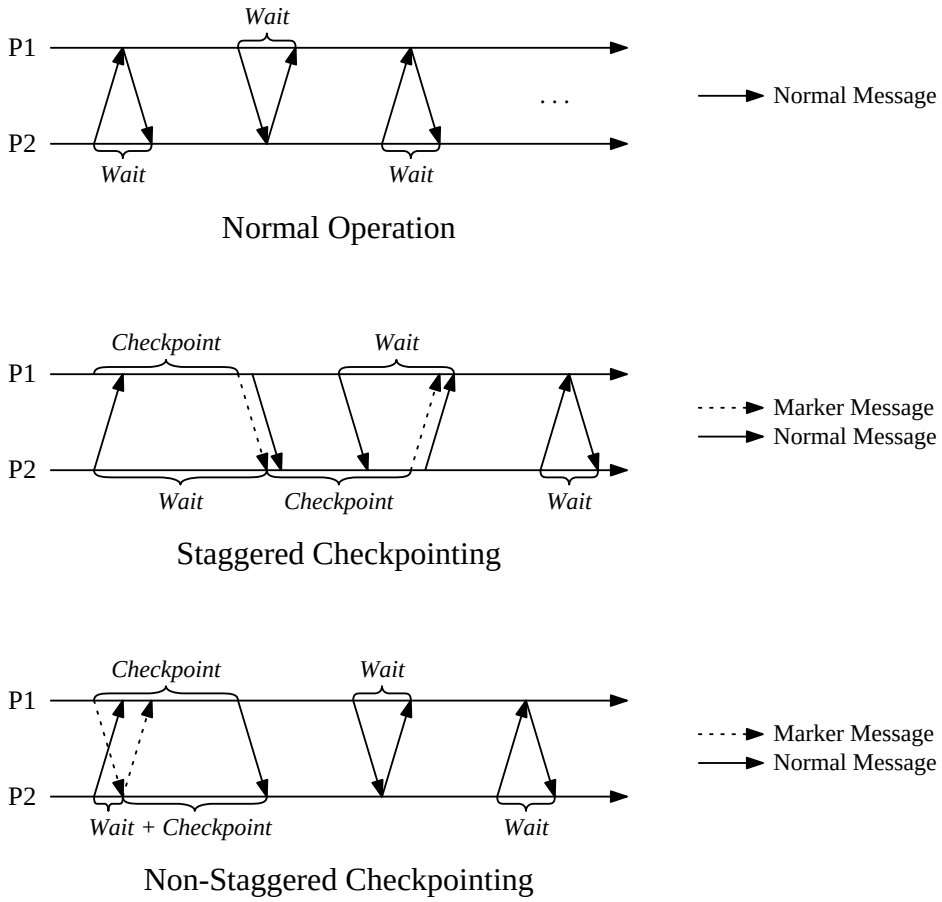

Figure 3.7: Staggering Checkpoints with a Synchronizing Pair of Processors.

pointed with a staggered algorithm, there can be cases when a processor has to wait for the other to finish taking its local checkpoint before it receives its acknowledgement. Thus, even though the checkpoints to disk are local, their duration can possibly cause overhead on another processor that is waiting for it to send a message. With non-staggered checkpoints, the possibility of this kind of overhead is diminished, as the local checkpoints occur roughly at the same point in time.

There are some situations where staggered checkpoints should cause less overhead than non-staggered ones. Consider the case when the number of processors greatly outnumbers the number of disks. For example, on the

iPSC/860, there may be up to 128 processors, but only four disks. If all the processors try to write their local checkpoints at the same point in time, then there is going to be heavy contention for the shared disks, and some processors are going to be delayed while waiting for other processors to write to disk. In this case, the staggered algorithms should cause less overhead than the non-staggered ones, since the writing of checkpoints is staggered. Moreover, the Network Sweeping algorithm should show the best performance because its spreads the checkpoints over a greater period of time than the Chandy-Lamport algorithm.

This last situation is not as important for taking main memory checkpoints. The reason is because once the checkpoint is copied to main memory, the target program is restarted and the checkpoint may be written to disk at time, not necessarily immediately. Thus, the processors may coordinate so that only one is writing to disk at a time, thereby eliminating any contention for the disks.

3.5 Compression

A rather obvious optimization to checkpointing is to use compression to decrease the number of bytes that get written to disk, thereby reducing the checkpoint time. File compression has seen widespread use in computer systems, as most user and computer generated files have enough structure for compression algorithms to shrink the file size by at least a factor of two. Thus, compression should also benefit checkpointing if the address spaces have the kind of structure of which compression algorithms can take advantage.

A concern of compression optimizations is the speed of the compression algorithm versus the amount of the checkpoint that gets compressed. If the compression algorithm is several times slower than a memory-to-memory copy, and it does not shrink the size of the checkpoint file considerably, then it is likely to increase both the checkpoint time and overhead rather than decrease them. To evaluate this tradeoff, we define the following quantities:

$$
\begin{aligned}
S &= \text{The size of the checkpoint file} \\
m(S) &= \text{The time to make a memory-to-memory copy of } S \\
c(S) &= \text{The time to compress } S \\
cf(S) &= \text{The size of } S \text{ compressed} \\
w(S) &= \text{The time to write } S \text{ to disk}
\end{aligned}
$$

The checkpoint times of sequential, main memory, and compressed checkpointing should then be as follows (This is assuming the main memory checkpoint fits in physical memory):

$$
\begin{aligned}
\text{Sequential:} &\quad w(S) \\
\text{Main memory:} &\quad m(S) + w(S) \\
\text{Compressed:} &\quad c(S) + w(cf(S))
\end{aligned}
$$

And the checkpoint overhead as follows:

$$
\begin{aligned}
\text{Sequential:} &\quad w(S) \\
\text{Main memory:} &\quad m(S) + \alpha w(S) \\
\text{Compressed:} &\quad c(S) + \alpha w(cf(S))
\end{aligned}
$$

In the above, α is a constant that denotes the overhead of concurrently writing a chunk of memory to disk. That is, even when a disk write is done at the same time as the execution of the target program, it affects the running of the target via cycle stealing and memory bus utilization. α is a measure of this overhead.

Thus, we expect compression to outperform sequential and main memory checkpointing whenever the gains of writing a smaller checkpoint file to disk outweighs the penalty of the compression time. That is, if the difference between $w(S)$ and $w(cf(S))$ is enough to offset the difference between $c(s)$ and $m(S)$, or $c(s)$ and 0.

If the main memory checkpoint does not fit into physical memory but the compressed checkpoint does, then it is obvious that compressed checkpointing should be better at decreasing overhead: No disk writes need be performed while the target threads are frozen and there need be no thrashing. If the compressed checkpoint does not fit into physical memory, then compression

can still be used to improve the checkpoint time of sequential and main memory checkpointing, by compressing and writing the checkpoint in chunks that will fit in main memory. All these cases are tested in our implementation.

Chapter 4

Implementations and Experiments

We have implemented checkpointing and recovery on a multiprocessor (the DEC Firefly) and a multicomputer (the Intel iPSC/860). These implementations encompass all algorithms and optimizations described in Chapters 2 and 3. The first three optimizations of Chapter 3 are implemented on the multiprocessor, and the rest are implemented on the multicomputer. The results of the experiments presented in this chapter demonstrate that we are able to implement checkpointing efficiently on both of these machines.

Before going into the details of the implementations, we reiterate our goals concerning the speed of checkpointing. First, we desire that the checkpoint time be near optimal. That is, it should be a small constant factor from optimal, like twice or three times. Second, the checkpoint overhead should be as low as possible. This means that the checkpoint has less effect on the target's running time. Finally, we would like to get latency down to a level where a user watching the program can always perceive that the target is getting work done. As stated in Chapter 1, we set this value at a tenth of a second.

4.1 Multiprocessor Implementation

We have implemented four checkpointing algorithms—sequential, main memory, copy-on-write, and CLL—as well as recovery on a DEC Firefly, an experimental shared-memory multiprocessor developed at the DEC System Research Center [TS87]. Fireflies consist of up to seven CVAX processors, each with a floating point unit and a direct-mapped 64 Kbyte cache attached to a central bus. The caches are snoopy caches, and thus listen to the shared bus to keep their caches coherent with the shared memory. The operating system for the Firefly is Taos [MS87], a version of Ultrix with threads and cheap thread synchronizations.

The Firefly upon which we tested our algorithms has four processors that share 16 Mbytes of physical memory. It also has a separate I/O processor that shares memory with the other four processors and has a separate bus connecting the Firefly to I/O devices and the outside world. In our system, the I/O processor is connected to a RD54 disk drive, which writes at a speed of 320 Kbytes per second.

The implementation is written in Modula-2+ [RLW85], an extension of Modula-2. The user interface to the checkpointing routines is a single call to *setup_checkpoint()* at the beginning of the user's program. This call may be used either to restore the program's execution to a saved recovery point, or to specify how long the checkpointer should wait before interrupting the program to take a checkpoint. Freezing the processors is provided by Taos through a system call. As stated in Section 2.1, in the absence of such a system call, one could freeze the processors either through interprocessor interrupts, or by protecting all the pages in memory to be "no access," and gaining control of the processors in the page fault handlers.

As mentioned in Section 3.2, one of the variables for the copy-on-write and CLL optimizations is the machine's page size. Although the actual page size on the Firefly's memory management unit is 512 bytes, we emulate pages sizes from 1K to 16K in our experiments.

4.1.1 Experiments with Merge Sort

For our initial experiments, we tested all four checkpointing algorithms on a parallel implementation of merge sort. We also checkpointed four other parallel programs: traveling salesman, matrix multiplication, pattern matching, and bubble sort. Since the results from the experiments with these programs are so similar to the results with merge sort, we only present merge sort in detail. The program sorts 250,000 indexed records, where the record size can be changed to modify the size of the program's address space.

In all experiments, the four processors of the Firefly are partitioned so that the target program uses three and the checkpointer uses one. This is to measure the maximal concurrency of our checkpointing methods. The checkpointer waits for the target to run for ten seconds, and then it takes one complete snapshot. For the initial results, the page size is eight Kbytes, and the page pool size of the CLL algorithm is 1 Mbyte. All times represent wall-clock time when the target program and checkpointer have exclusive use of the system. All times are averages of five or more runs.

Checkpoint Time and Overhead

Figures 4.1 and 4.2 display the checkpoint time and overhead imposed by the four checkpointing algorithms as a function of the size of merge sort's address space.[1] The total checkpoint time displayed in Figure 4.1 measures the elapsed time from the start of the checkpoint to its conclusion. The second graph in Figure 4.1 extends the first graph to include the checkpoint time when the address space approaches the size of physical memory. The checkpoint overhead in the first graph of Figure 4.2 is the amount of time by which the checkpoint increases the target's running time. The second graph displays the overhead as a percentage of the checkpoint's running time. This is equal to checkpoint overhead divided by the checkpoint time. As stated in Chapter 1,

[1]The raw data for all the graphs in this chapter are in Appendices B and C.

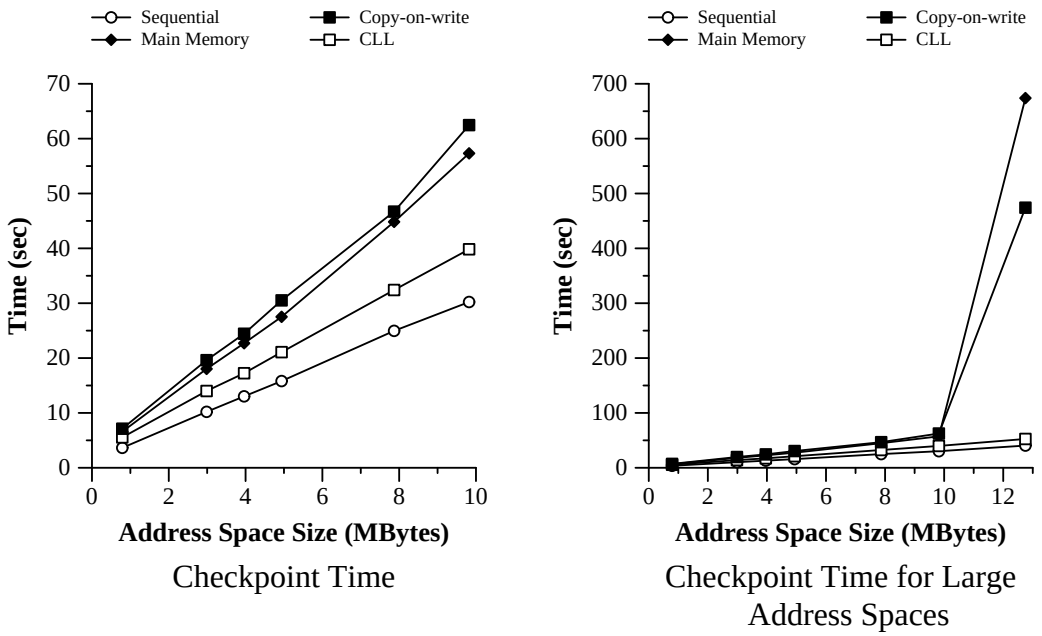

Figure 4.1: Checkpoint Time for the Multiprocessor Algorithms

concurrency can be calculated as:

$$concurrency = 100\% - overhead\ percentage.$$

It comes as no surprise that in all four algorithms, checkpoint time is proportional to the size of the address space. Also as expected, the sequential algorithm records the fastest checkpoint time, followed by the CLL algorithm. The other two algorithms take longer to record a checkpoint because they wait for a complete copy of the checkpoint to exist in main memory before writing the checkpoint to disk.

As shown in the second graph of Figure 4.1, when the address space approaches the size of physical memory, the main memory and copy-on-write algorithms exhibit severe thrashing, as their memory needs far exceed the size of physical memory. The other two algorithms keep their memory needs below the size of physical memory, and therefore do not suffer such rash penalties. It is worth noting that for all but the smallest address space tested, the pool

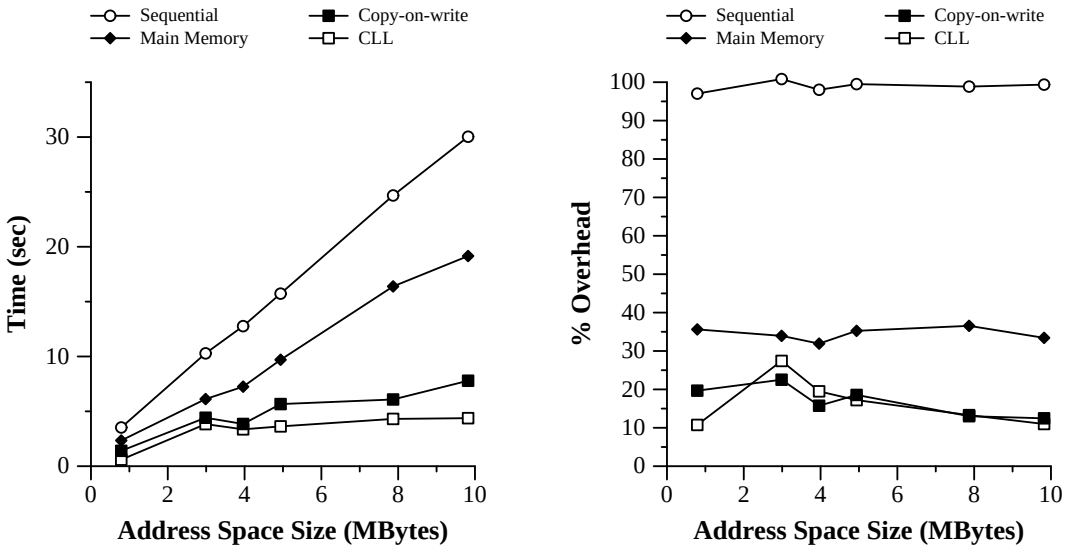

Figure 4.2: Overhead Data for the Multiprocessor Algorithms.

of pages in the CLL algorithm became completely filled. Therefore, some worst-case data is included in the graphs.

Figure 4.2 shows that the two algorithms based on copy-on-write display the smallest overhead, and therefore the greatest concurrency. This is because these algorithms freeze the processors for the smallest amount of time. Taken as a whole, Figures 4.1 and 4.2 show that the CLL algorithm is clearly the best of the four in regard to the combination of checkpoint time and concurrency. The results that follow pertain only to this algorithm.

Latency Data

Figure 4.3 shows latency data for the CLL algorithm. The overhead of the algorithm is divided into two parts—the time that all the threads are stopped initially to protect the address space and save the threads' states, and the time that the target threads are trapped, waiting to process page faults. The first curve of the first graph in Figure 4.3 represents the initial stop time as a function of address space size, and the second curve represents the maximum time that any thread waits as a result of a page fault.

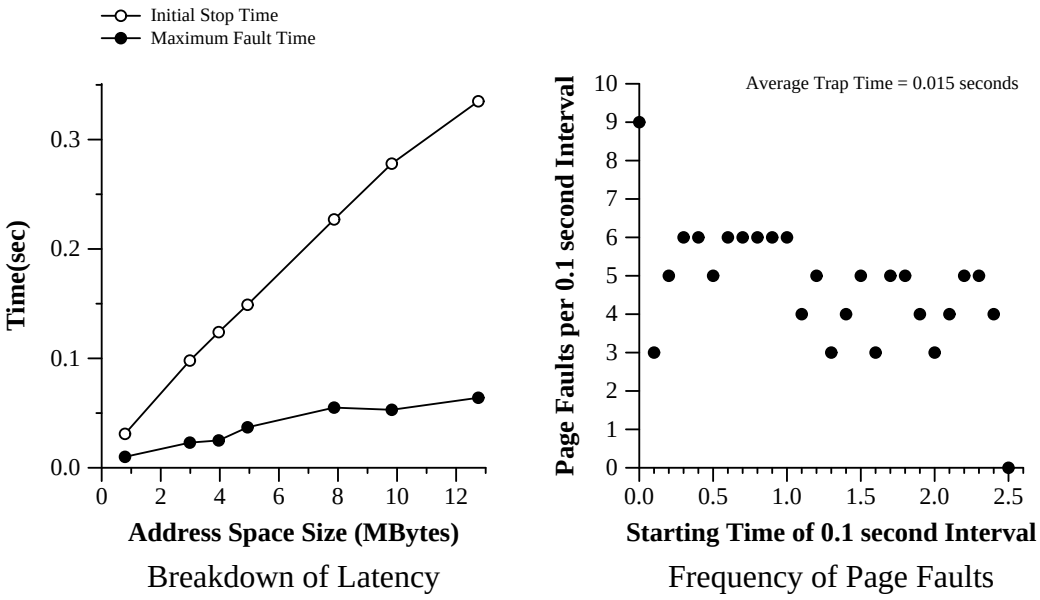

Figure 4.3: Latency Data for the CLL Algorithm.

For address spaces up to three Mbytes, the initial stop time is kept below a tenth of a second. Moreover, for all address space sizes, the maximum page fault time is well below our low-latency goal of 0.1 seconds. The second graph in Figure 4.3 displays the frequency of page faults over time for a run with a four Mbyte address space. In this graph, the checkpoint is broken into 0.1-second intervals, and the number of page faults in each interval is plotted. The purpose of the graph is to show that work is indeed being accomplished by the target threads during the initial phases of the checkpoint.

Note that after an initial burst of nine page faults, the trapping frequency steadies at six faults per tenth of a second for the first second. Then it slows to about four traps per tenth of a second until there are no more page faults. The average time to process a page fault is 0.015 seconds. Thus, during the first second of the checkpoint, the threads spend around 0.09 seconds per 0.1-second interval processing page faults. Since there are three target threads, this means that the threads spend only a third of their time processing page faults in the first tenth of a second; the rest of the time is devoted to completing the merge.

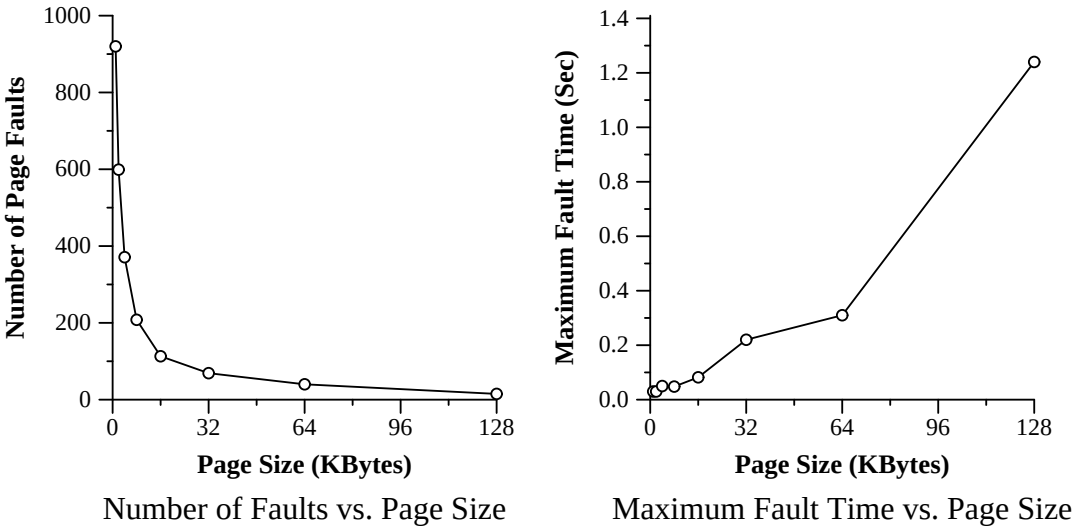

Figure 4.4: Results of Altering the Page Size.

Thus, even at the beginning of the checkpoint, when one expects the highest frequency of page faults, the target still performs an adequate amount of work.

Figure 4.4 displays the results of altering the page size, again for a merge sort example with a four Mbyte address space. As would be expected, the total number of page faults is proportional to the inverse of the page size, while the maximum time to process a trap increases almost linearly with the page size. Therefore, the ideal page size is one that significantly decreases the number of page faults, while not significantly increasing the maximum page fault time.

This data shows that a page size of 16 Kbytes is ideal. The number of page faults is kept relatively small (around 120 faults), and the maximum page fault time is still below a tenth of a second. Note that this large page size has one other advantage: If the hardware were built with an actual page size of 16 Kbytes, then upon protecting the address space, it would only have to change $\frac{1}{16}$ the number of page table entries than it currently has to change. This should reduce the initial stop time (as displayed in Figure 4.3) by the same

factor, which would bring the checkpointer's latency to well under a tenth of a second for all address space sizes.

4.1.2 Other Experiments

We tested the four checkpointing algorithms on four other test programs. As in the merge sort example, three processors of the Firefly are allocated for the target program, and one is used for the checkpointer. One complete checkpoint is taken ten seconds after the target has started running. The target and checkpointer have exclusive use of the system.

1. Traveling Salesman. This is a parallel program that solves the NP-complete Traveling Salesman problem for an instance with ten cities. Parallelism is achieved by each processor working on a subset of the possible paths. The program allocates an address space of 64 Kbytes, and takes 65 seconds to run in the absence of checkpointing.

2. Matrix Multiplication. This program multiplies two 200 X 200 matrices. The straightforward $O(n^3)$ algorithm is used, and parallelism is achieved by each processor calculating a subset of the product's elements. The address space is 531 Kbytes, and the running time is 43 seconds.

3. Pattern Matching. This parallel program takes a pattern array of 10 characters and finds the location of the 10 contiguous characters in a 1,200,000 element array that are the smallest edit distance from the pattern. Its address space is 1.2 Mbytes and it runs in 73 seconds.

4. Bubble Sort. This is a program that uses the $O(n^2)$ bubble-sort algorithm to sort 3000 indexed records. Upon running, it allocates a heap of 3.0 Mbytes, and takes 44 seconds.

Figure 4.5 shows the results of the CLL algorithm on these benchmark programs, as compared with instances of merge sort with comparable heap sizes. The first graph of Figure 4.5 displays that checkpoint time is essentially

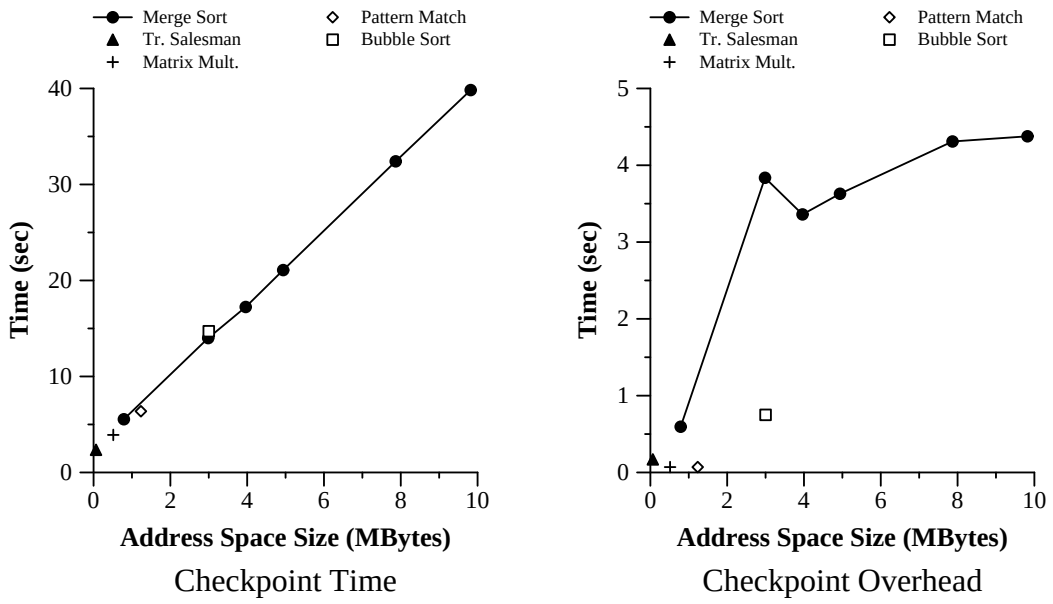

Figure 4.5: Checkpoint Time and Overhead of Other Test Programs.

a function of heap size, no matter what kind of target it is checkpointing. The second graph shows that at least for the benchmarks tested, merge sort acts as a worst case target, because of its almost random patterns of memory access. The pattern-matching target, which has an extremely high locality of reference, produces almost no overhead at all, as does the matrix multiplication, which likewise tends to have more local memory access patterns.

4.1.3 Conclusions

The implementation of checkpointing and recovery on the Firefly shows that the four algorithms tested indeed work as checkpointing algorithms. Moreover, for all the programs tested, it shows that the CLL optimization is successful in reducing checkpoint overhead and latency without increasing checkpoint time by an unreasonable amount: For all five test programs, checkpointing time is always within 50% of optimal, where optimal is defined to be the checkpoint time of the sequential algorithm, 80 to 90% of the checkpointing executes

concurrently with the target programs, and the latency is kept below a tenth of a second.

4.2 Multicomputer Implementation

We have implemented the three basic consistent checkpointing algorithms on the Intel iPSC/860 multicomputer. These algorithms are the Sync-and-Stop, Chandy-Lamport, and Network Sweeping algorithms. We also implemented all the optimizations described in Chapter 3, except for the copy-on-write and CLL optimizations. Thus, our results only concern the minimizing of checkpoint time and overhead, and not the lowering of latency. In the conclusion of this section, we discuss how we expect the CLL optimization to interact with the algorithms and the other optimizations.

4.2.1 The iPSC/860

The Intel iPSC/860 is a multicomputer consisting of Intel 80860 processors connected with a hypercube interconnection and wormhole routing. The largest configuration of the iPSC/860 is 128 processors, each of which may have up to 16 megabytes of physical memory. Unlike the Firefly, which is an experimental machine with limited power, the iPSC/860 is a powerful workhorse that has been used extensively for scientific computing and numerical applications.

The iPSC/860 must be connected to a host machine, which is called the *Shared Resource Manager (SRM)*. To execute a program on the iPSC/860, users employ the SRM to allocate a portion of the hypercube's processors, and then load their program onto the allocated nodes. Once a user has allocated a number of processors, he or she owns those processors exclusively until they are deallocated. Thus, users cannot share individual nodes of the iPSC/860. Moreover, the iPSC/860 allows only one user process to execute on a processor at a time. Thus at any point in time, exactly one program may have complete control over each processor in the allocated cube.

The iPSC/860 runs an operating system called NX/2 [Pie88], which implements a subset of Unix System-V and primitives for message-passing. NX/2 allows the use of three types of file systems: The SRM's file system, Network File System (NFS), and Concurrent File System (CFS). Any file activity that uses NFS must go through the SRM. Thus, the first two file systems are sequential—all file activity is serialized at the SRM. The Concurrent File System is implemented on four Intel 80386 processors that are connected directly to the iPSC/860's nodes [Pie89]. All CFS files are striped over multiple disks that are connected to the 80386 processors. In this way, CFS is more effective as a file system for a multicomputer, as it is not as likely to become a bottleneck.

Most applications for the iPSC/860 are written in C or FORTRAN. In these programs, any processor can send messages to any other processor and to the SRM. However, because of the speed discrepancy between the iPSC/860 nodes and the SRM (which is a 80386-based workstation), using the SRM to do processing with node programs is discouraged.

The main message-sending primitives are the procedures `csend()` and `crecv()`, which are defined as follows (in C):

```
void csend(long type, char *buf, long len, long node, long pid);
void crecv(long type, char *buf, long len);
```

`Csend()` sends `len` bytes of memory starting at `buf` to `node`. On the iPSC/860, `pid` must always be 0. The `type` argument is for the user's convenience. Although it forces the user to match sends and receives by type, it also lets them receive messages out of order: Even though messages are transmitted from processor to processor in FIFO order, they are buffered in the receiving processor's kernel until the user calls `crecv()` with a matching `type`. Thus, the following sequence of calls is perfectly legal on the iPSC/860, and does not cause deadlock:

Node 1	Node 2
`csend(10, 0, 0, 2, 0);`	`crecv(11, 0, 0);`
`csend(11, 0, 0, 2, 0);`	`crecv(10, 0, 0);`

Messages of the same type are always received in FIFO order. `Csend()` and `crecv()` are synchronous message passing calls. That is, `csend()` blocks until the message is stored in the receiving node's kernel, and `crecv()` blocks until its processor receives the desired message and copies it into the specified location in memory.

There are two other flavors of send and receive primitives. `Isend()` and `irecv()` are asynchronous primitives. They return to the user immediately, regardless of whether or not the send or receive has completed. The user must test for completion later. Finally, `hrecv()` is a primitive that allows the user to set up an interrupt handler for a given message type. When a message of that type arrives, it invokes the specified interrupt handler. This allows the user to process messages in a demand-driven fashion.

There are many other primitives on the iPSC/860. Most of them, however, are implemented with the send and receive primitives described above.

4.2.2 Checkpointing on the iPSC/860

This section gives an overview of the program ICKP, the checkpointer that we wrote for the iPSC/860. ICKP consists of two parts: A library of object files, and a main object file. To compile a C program for checkpointing, one must first change the name of one's *main()* subroutine to *ickp_target()*. To compile a FORTRAN program for checkpointing, one must first change the *PROGRAM* module to be a subroutine called *ickp_target*. Then, one must recompile the new *ickp_target()* file and relink all the program's modules with ICKP's main object file and object library. The resulting executable may be checkpointed and recovered.

An ICKP program first reads information from a control file. This file specifies information like the checkpointing algorithm and optimizations to use, in which directory to write the checkpoint, and how frequently to checkpoint. Next, node 0 sets up an `hrecv` handler for a specific message type, which is assumed to be off limits to general users (the iPSC/860 has many such types reserved for its library calls). After the appropriate time interval has passed, a message of that type is sent to node 0, and the checkpoint is started. When the checkpoint is finished, node 0 sets up a new `hrecv` handler for the next checkpoint, and any older checkpoint files are deleted.

If an ICKP program is called with a special set of command line arguments, the program will recover from the most recent checkpoint instead of running from the beginning. After recovering, the program may continue to take checkpoints.

Checkpointing on the iPSC/860 is much more subtle than checkpointing on the Firely. This is because there is a message state on the iPSC/860 that is not present on a multiprocessor like the Firefly. Moreover, this message state is complex. In Chapter 2, we made the assumption that messages are sent directly from user process to user process and that messages are received by user processes in FIFO order. On the iPSC/860, these assumptions do not hold. Instead, messages are sent from user processes to kernel processes, and although the messages are received by the kernel in FIFO order, the `type` argument of the send and receive primitives allows them to be received by the user processes in arbitrary order. Thus, to implement the checkpointing algorithms on the iPSC/860, we had to convert the model of a multicomputer from Chapter 2 to the model implemented on the iPSC/860. As this conversion is complex and messy and not crucial for understanding the results of the implementation, we include it in Appendix A.

4.2.3 A Synthetic Program: `A_m_s`

For the preliminary evaluation of the checkpointing algorithms, we have them checkpoint a synthetic test program called `a_m_s`, which stands for "allocate,

multiply, and synchronize." This program takes four arguments: `iter`, `n`, `size`, and `rand`. Once loaded onto the cube, each node performs the following pseudo-C code:

```
ams(int iter, int n, int size, int rand)
{
  int i;
  unsigned char *a;

  a = malloc(size);

  for (i = 0; i < size; i++) {
    a[i] = (rand) ? random()%256 : 0;
  }

  for (i = 0; i < iter; i++) {
    <perform n multiplications>
    <synchronize>
  }
}
```

The reason we show the results of this program is because one can use it to exemplify many types of program behavior. The address space size can be set with the `size` parameter, and the degree of compressibility can be altered with `rand`. If `n` is large, then the nodes execute independently, with little synchronization. If `n` is small, then the nodes execute with tight synchronization. `Iter` is used to alter the running time. Table 4.1 displays the instances of `a_m_s` that are used in the following discussion. In the abbreviations, `IND` stands for "independent," since the processors only synchronize at the beginning and end of the program. `SYNC` stands for "synchronous," as the processors synchronize often—once a second. The `SMALL`, `MED` and `LARGE` refer to the size of the address space.

After presenting and discussing results of the `a_m_s` program, we will show ICKP's performance on numerous other test programs. These are programs written by various other people to help them conduct their research on the iPSC/860. They represent a spectrum of real application programs for the iPSC/860.

Abbreviation	iter	n	size	rand	Address Space Size (Kbytes)	Synchronization Interval (Sec)	Running Time (Sec)
SMALL-IND	1	30,000,000	10	0	196	90	90
MED-IND	1	30,000,000	1,500,000	0	1,696	90	90
LARGE-IND	1	30,000,000	3,000,000	0	3,196	90	90
SMALL-SYNC	80	336,400	10	0	196	1	80
MED-SYNC	80	336,400	1,500,000	0	1,696	1	80
LARGE-SYNC	80	336,400	3,000,000	0	3,196	1	80
*-RAND	-	-	-	1	-	-	-

Table 4.1: Instances of a_m_s used to evaluate ICKP

4.2.4 Basic Checkpointing Algorithms

Figures 4.6 and 4.7 show the checkpoint times and overheads for the three basic consistent checkpointing algorithms when they checkpoint different instances of a_m_s, running on a 32-node iPSC/860. Each node of the iPSC/860 has eight megabytes of physical memory. In this and all subsequent tests, the nodes checkpoint to CFS files, as the other two file systems are slower by at least an order of magnitude. All results represent averages of three or more runs where each run contains one, two, or three complete checkpoints, depending on how many checkpoints could be taken during the target's execution.

These graphs show that for basic checkpointing with no optimization, there's very little difference between the three algorithms. Their checkpoint times are all about the same: As in the multiprocessor case, they are directly proportional to size of the address space. The overhead of the SNS algorithm is slightly higher than the other two because of the extra time that the processors are frozen, waiting to synchronize with the others.

The overhead of the marker messages for checkpointing is small in these experiments. The Chandy-Lamport and Network Sweeping algorithms both give similar times and overheads, even though the Chandy-Lamport algorithm sends $32 * 31 = 992$ m_{CL} messages, while the Network Sweep algorithm sends only 31 m_{CL} messages plus $32 * 5 = 120$ m_{NS} messages. The average time to send a marker message on the iPSC/860 is 0.001 second. Thus, in the Chandy-

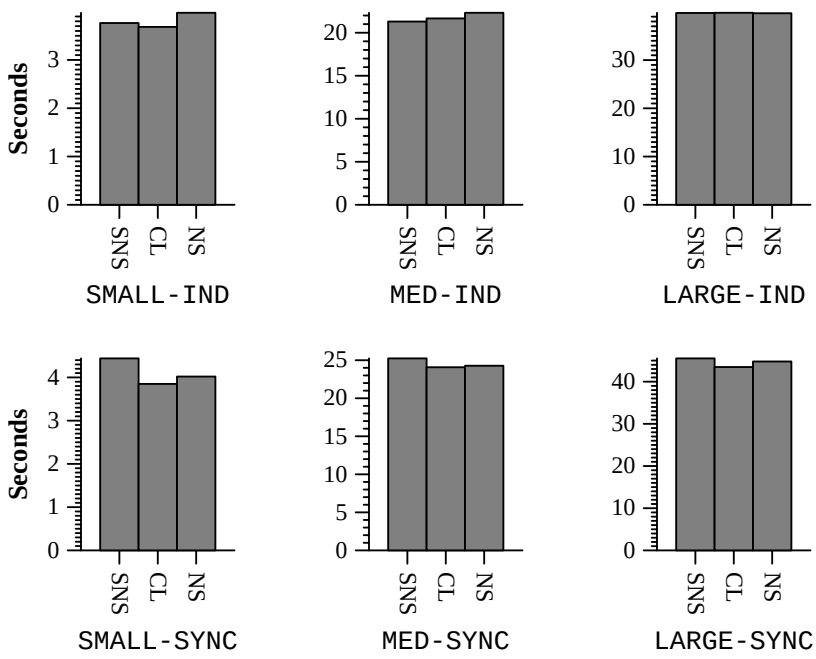

Figure 4.6: Checkpoint Times of Basic Algorithms.

Lamport algorithm, each processor spends about 0.031 seconds sending and receiving marker messages, while in the Network Sweep message, each spends 0.006 seconds. These values are small enough to be in the noise of the data. For machines with a larger number of processors, this portion of checkpointing would be a more significant factor.

4.2.5 Results of the Staggering Optimization

Figures 4.8 and 4.9 show the results of staggering the CL and NS checkpoints for the six instances of the `a_m_s` program. The graphs corroborate our expectations about staggering checkpoints. In all cases, staggering increases the checkpoint time of the algorithm. In the `SYNC` instances of `a_m_s`, staggering increases the NS algorithm's checkpoint time less than in the `IND` instances, because the processors are sending synchronization messages to one another every second, and thus many checkpoints are started with an m_{CL} message that was sent before the sender's synchronization message. In the `IND` instances,

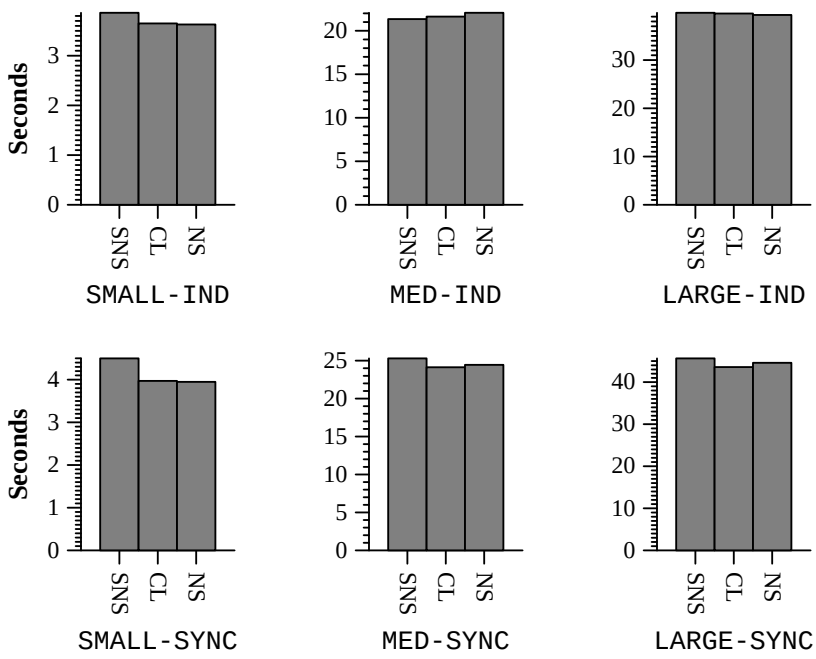

Figure 4.7: Checkpoint Overhead of Basic Algorithms.

there are no m_{CL} messages sent in the staggering version of the NS algorithm. Thus, the degree of staggering is much greater than in the `SYNC` instances.

The staggering causes predictable results in Figure 4.9 as well. In the `SYNC` instances of `a_m_s`, staggering increases the overhead of the algorithm. This is because the processors spend extra time waiting to receive synchronization messages from processors that are busy checkpointing, as illustrated in Figure 3.7 of Chapter 3. In the `IND` algorithms, however, staggering is beneficial to checkpoint overhead, since there are no synchronization messages upon which processors may stall, and the staggering helps reduce contention on the disks. It may be recalled that there are four disks that CFS uses to perform file writes. Moreover, CFS stripes files on those four disks. Thus, even though CFS reduces the overhead of file I/O from file systems that go through the SRM, the disks are still a source of contention: Staggering the checkpoints is indeed a good way to reduce the overhead of checkpointing. As further proof, when the `LARGE-IND` instance of `a_m_s` was checkpointed with the non-staggered version of the NS algorithm, each processor took an average

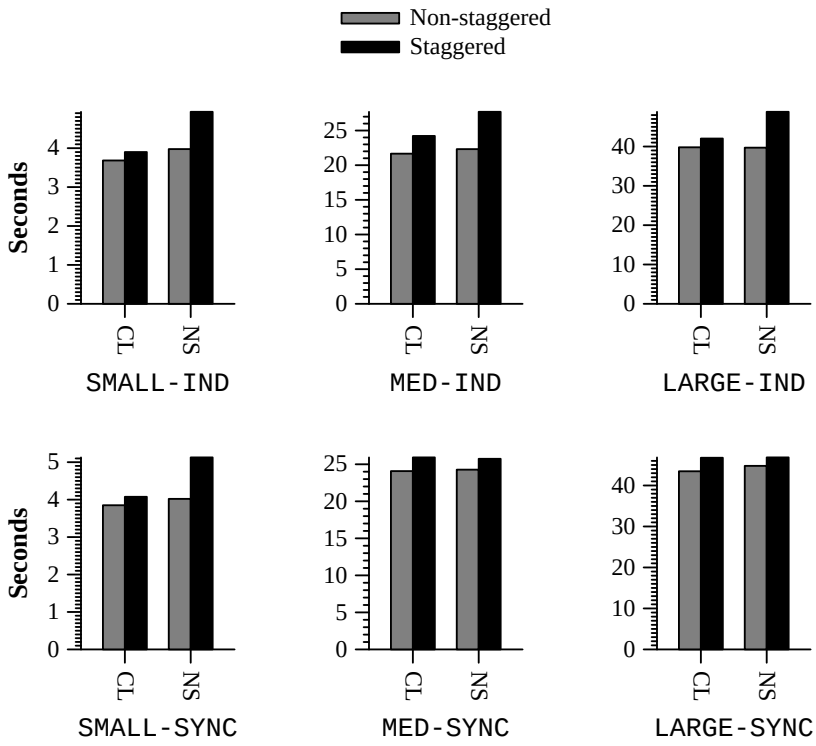

Figure 4.8: Effect of Staggering on Checkpoint Time.

of 34 seconds to write its checkpoint to disk. In the staggered version, the average was 12 seconds.

4.2.6 Main Memory Checkpointing

Figures 4.10 and 4.11 show the results of using the main memory optimization to improve checkpoint overhead. The non-staggering versions of the CL and NS algorithms were used. Moreover, each node of the iPSC/860 was limited to four megabytes of physical memory. The reason for this is to make the `LARGE-IND` and `LARGE-SYNC` instances of `a_m_s` have address spaces that are too large to copy into main memory. When such a situation occurs, the checkpointer writes enough of the checkpoint to disk while the processors are frozen that the rest of the checkpoint fits into main memory. This last fragment of the checkpoint

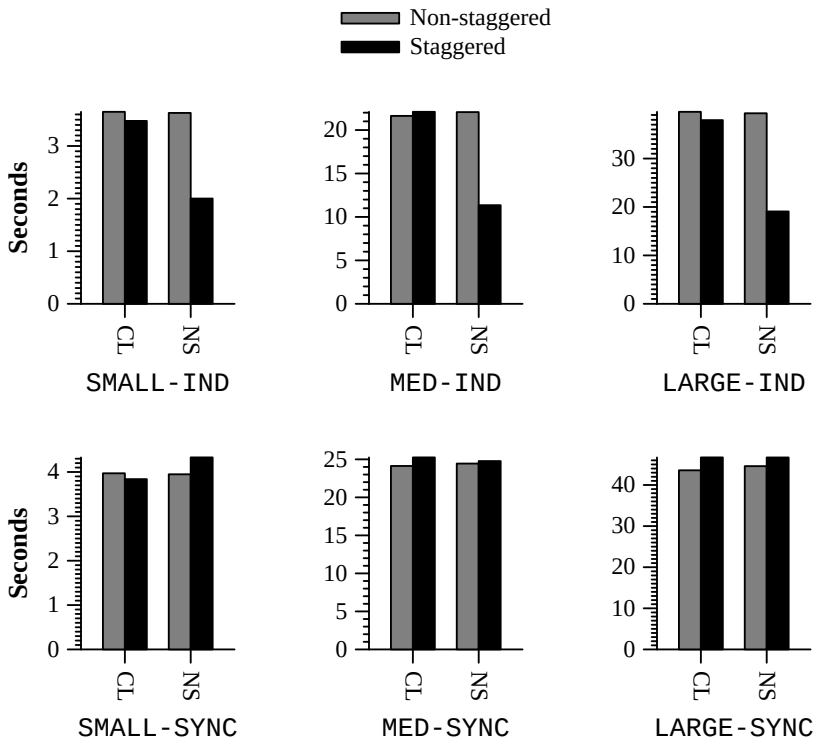

Figure 4.9: Effect of Staggering on Checkpoint Overhead.

is then copied to main memory, and the processors are restarted before writing the fragment to disk.

As mentioned in Section 3.4, main-memory checkpointing is extra beneficial in multicomputer checkpointing, for when the main memory checkpoint fits into physical memory, the checkpointer may write the checkpoint to disk at its leisure. Thus, ICKP attempts to reduce disk contention by enforcing serialization among the processors' disk writes. That is, processor 0 writes its checkpoint to disk immediately after restarting its target program. Thereafter, processor n must wait for processor $n-1$ to finish writing its checkpoint to disk before it may write its checkpoint.

Figure 4.10 shows how these optimizations affect the overall checkpoint time. For the `SMALL` and `MED` instances of `a_m_s`, the main memory algorithms increase the checkpoint time significantly. This is because of the extra time copying bytes, and because of the extra time that processors spend waiting

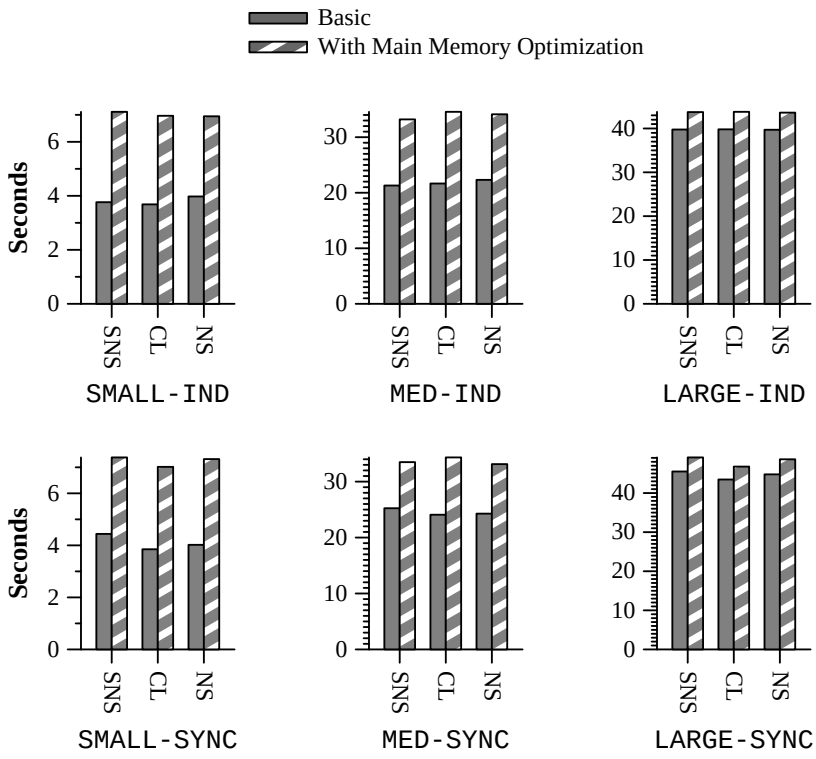

Figure 4.10: Effect of Main Memory Checkpointing on Checkpoint Time.

for their turn to write their checkpoint to disk. However, Figure 4.11 shows that the optimizations have a dramatically beneficial effect on the checkpoint overhead. Essentially, the only overhead is the memory-to-memory copy time.

In the `LARGE` instances, the increase of the checkpoint time and the decrease of the checkpoint overhead are not nearly so dramatic. This is because the main memory checkpoint does not fit into physical memory, and thus, most of it has to be written to disk while the processors are frozen. Moreover, since the checkpoint does not fit into main memory, the disk writes cannot be staggered and thus contention cannot be reduced in that manner. Still, this way of dealing with the limitations of physical memory is far superior than in the Firefly checkpointer, where the main memory optimization causes thrashing in the virtual memory system if the main memory checkpoint does not fit. Here, the main memory optimization still improves checkpoint overhead, albeit to a much smaller degree than when the checkpoint fits into physical memory.

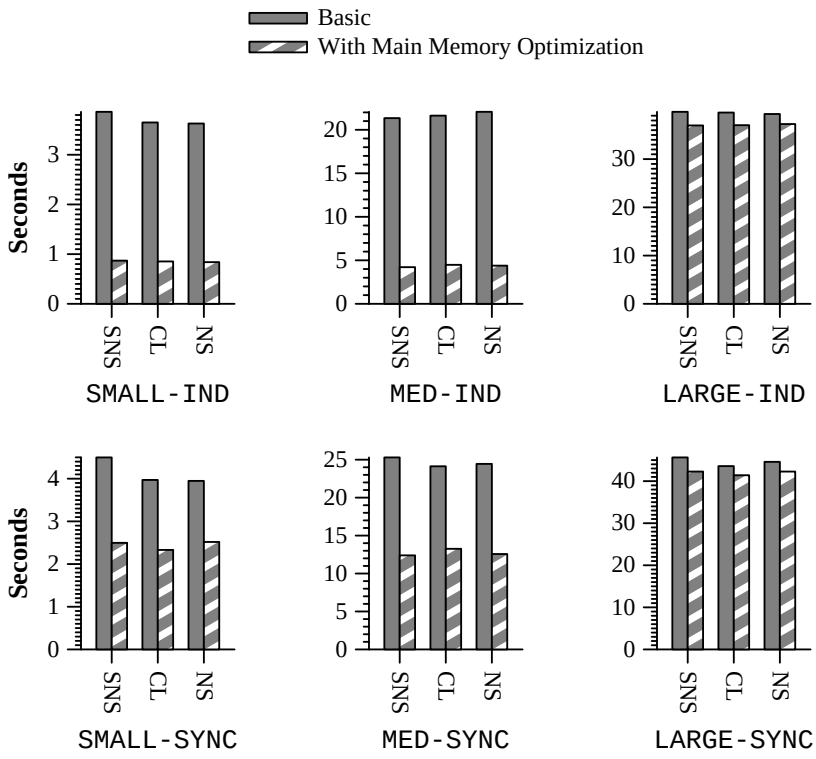

Figure 4.11: Effect of Main Memory Checkpointing on Checkpoint Overhead.

Figures 4.12 and 4.13 show the effects of staggering the main memory algorithms. As expected, in the `SMALL` and `MED` instance of `a_m_s`, there is little change between staggering and non-staggering. This is because both main memory algorithms can stagger the writes to disk once they restart the processors. In the `LARGE` instances, there is little change from the results of the basic algorithms. This is because most of the checkpoint is written to disk while the processors are frozen, and therefore staggering helps in the `LARGE-IND` case, and hurts in the `LARGE-SYNC` case.

The checkpoint overheads measured with the `MED-SYNC` instance of `a_m_s` increases with staggering for the same reason the overhead increases in Figure 4.9: Because of the staggering, the processors have to wait an extra period of time to receive their synchronization messages.

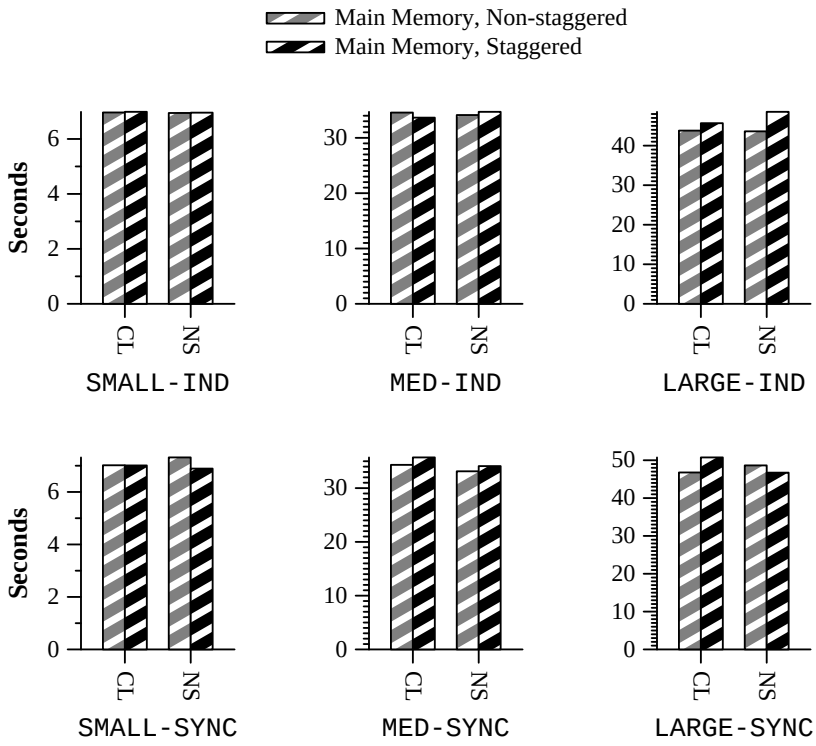

Figure 4.12: Effect of Staggering on Main Memory Checkpoint Times.

4.2.7 Results of Compression

The compression algorithm implemented in ICKP is a slight variation of an algorithm defined by Wheeler and called "Algorithm 1" in [BJLM92]. It was chosen as the algorithm of choice for on-line data compression in their log-structured file system, because it achieves decent compression ratios for most data files, and is much faster than more traditional compression algorithms, such as the "LZC" algorithm used in the Unix `compress` utility [BWC89]. The algorithm works as follows: While compressing, maintain a hash table that hashes on the last three bytes compressed. If the next byte matches the value stored in the hash table, then emit a '0'. Otherwise, emit a '1' followed by the byte, and update the value in the hash table.

ICKP implements a variation of the algorithm that adds run-length encoding: Instead of emitting a '0' on a hash table hit, simply increment a

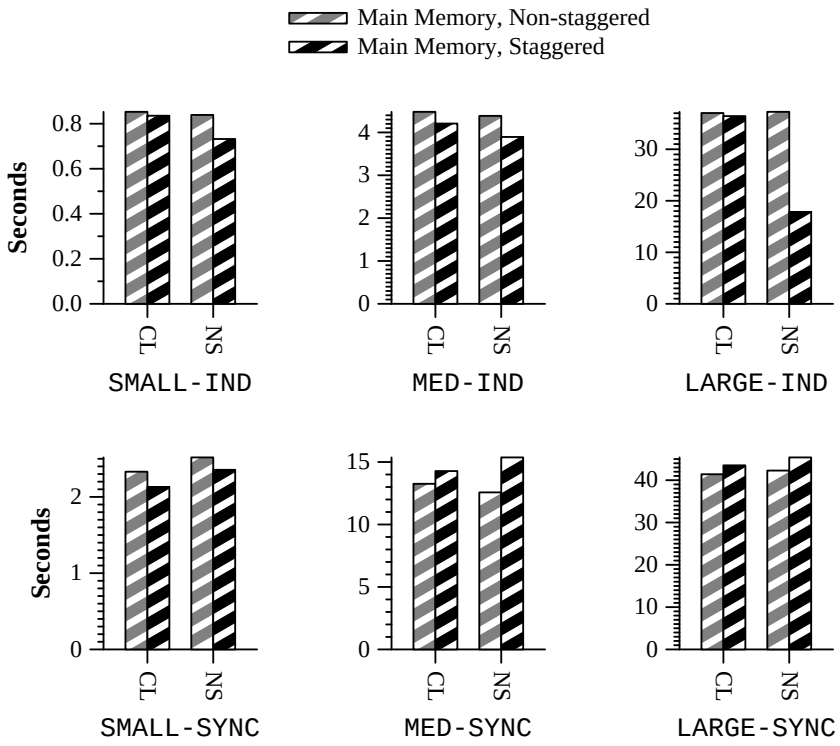

Figure 4.13: Effect of Staggering on Main Memory Checkpoint Overhead.

counter called *nhits*. Upon a miss, if *nhits* is greater than zero, then emit $\lfloor \log_2 nhits \rfloor + 1$ zeros, followed by *nhits* starting with its most significant bit. Then emit a '1' and reset *nhits* to zero. This variation improves the compression ratio of the algorithm when the hash table hits come in groups. For example, a sequence of 128 hash table hits will yield 16 bytes of zeros in the original algorithm. The same sequence will yield just two bytes with run-length encoding: `'00000000 1000000'`. When the hash hits are sparse, then the variation can degrade compression by up to one bit per hit.

We chose the variation for ICKP because address spaces often contain large chunks of uninitialized data. As will be shown later, this is especially true in many FORTRAN programs, as FORTRAN requires static allocation of memory, and thus many programmers preallocate huge chunks of memory in their programs so that they can scale up to larger data sizes without recompilation.

Moreover, in the case where the variation degrades compression, hash table hits are sparse, and thus the compression factor will not be very large.

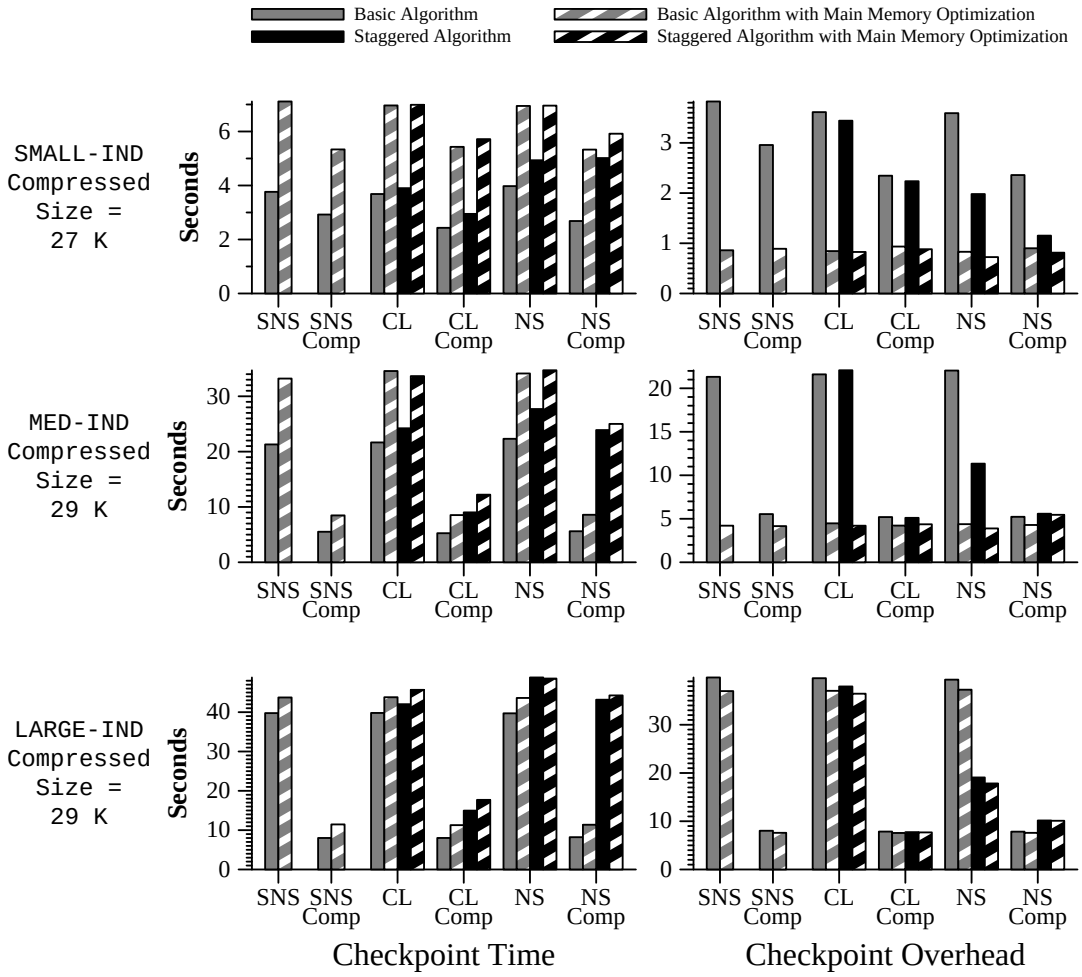

Figure 4.14: Effect of Compression on the `IND` Instances of `a_m_s`

Figures 4.14 and 4.15 show the results of compression on every consistent checkpointing algorithm and optimization. Note that since the `MED` and `LARGE` instance of `a_m_s` allocate large chunks of zeros, their checkpoints are compressed down to very small sizes. Thus in these instances, compression improves almost every algorithm in both checkpoint time and overhead. Note that the compressed and staggered optimizations of the Network Sweep algorithm have a drastic increase in checkpoint time over the non-staggered

versions. This is because the time to compress the checkpoint is larger than the time to make a memory-to-memory or disk copy, and thus the degree of staggering is larger than when the checkpoints are not compressed. This degree of staggering is responsible for the drastic increase in checkpoint overhead for the SYNC instances of a_m_s: Staggering is heavily penalized because the processors have to wait extra long for synchronization messages.

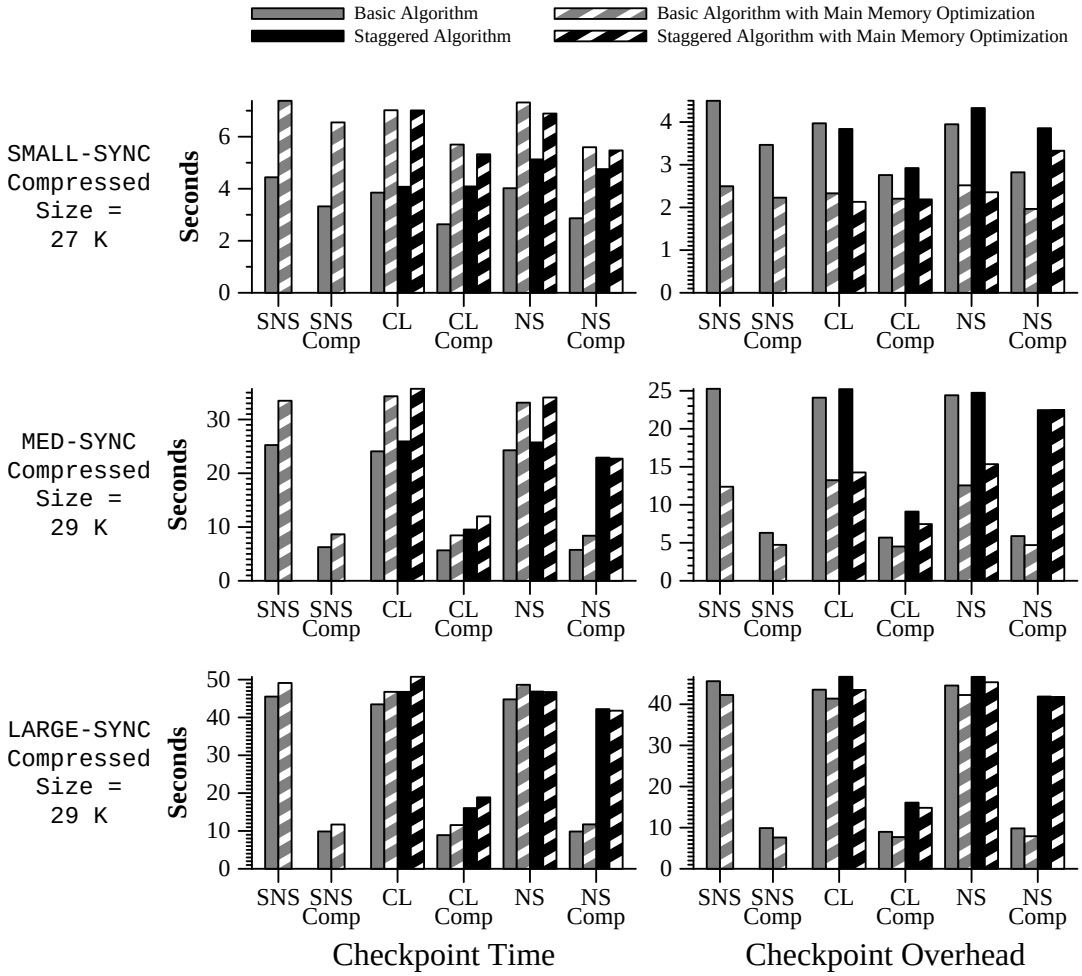

Figure 4.15: Effect of Compression on the SYNC Instances of a_m_s

The compression of the SMALL instances of a_m_s are interesting. One would expect that since the program allocates almost no extra memory, the address space would not compress very well. However, it is compressed by

more than 84%. Thus, its checkpointing times are reduced with compression. Yet, since the checkpoints files are so small, the checkpoint overheads of the main memory optimizations are not improved. This is because the increase in the time that it takes to do the compression is not offset by the reduction in time that it takes to do the write.

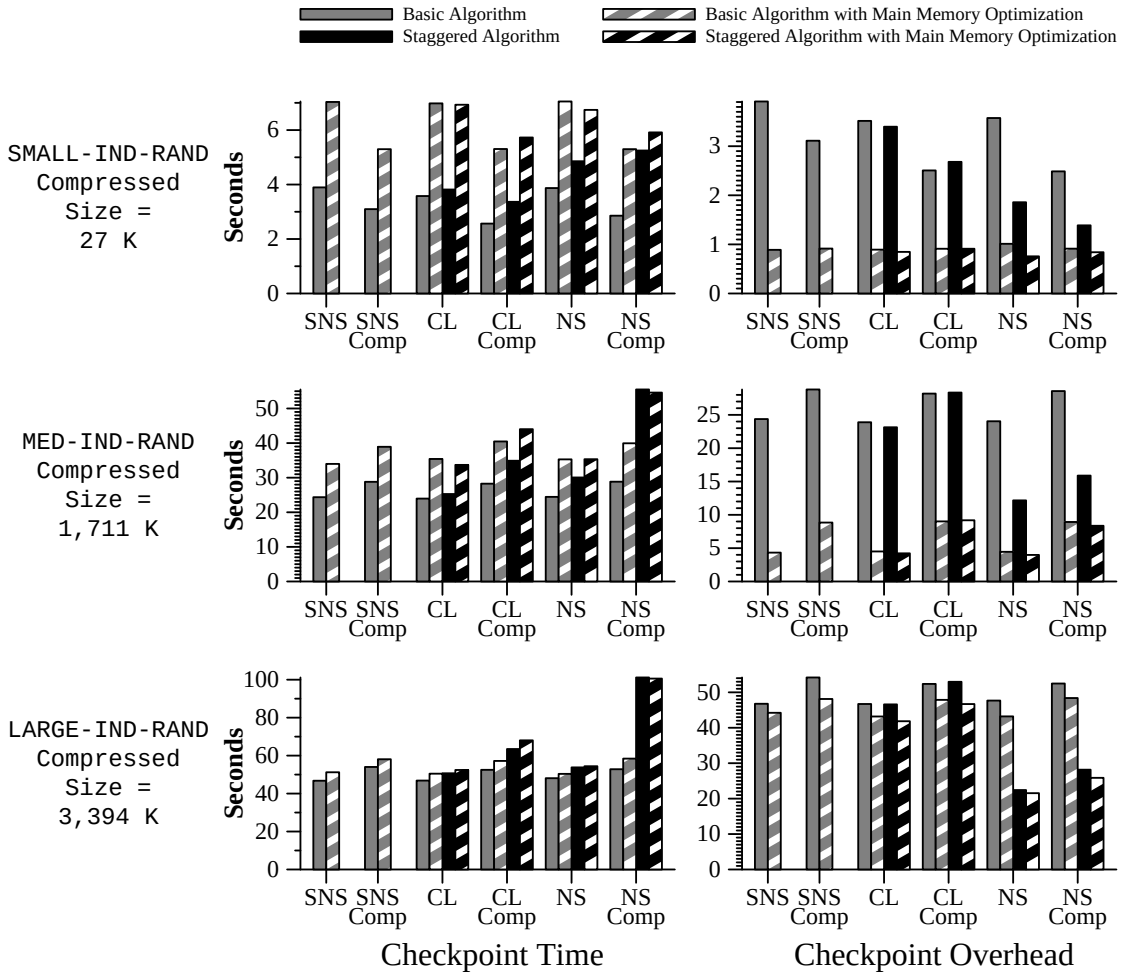

Figure 4.16: Effect of Compression on Random `a_m_s` Programs

The graphs in Figure 4.16 show the opposite extreme of compression: The bytes that `a_m_s` allocates are set randomly, and the compression algorithm fails to compress the checkpoint. Thus, compression is doubly bad—it increases the size of the checkpoint, and the compression takes longer than a

Instance of a_m_s	Non-compressed		Compressed	
	Checkpoint Time (SNS)	Recovery Time	Checkpoint Time (SNS)	Recovery Time
SMALL-IND	3.76 (sec)	4.72 (sec)	2.92 (sec)	4.50 (sec)
SMALL-SYNC	4.44	4.78	3.32	4.62
MED-IND	21.30	22.08	5.52	6.39
MED-SYNC	25.42	23.67	6.26	5.44
LARGE-IND	39.76	43.09	8.01	9.21
LARGE-SYNC	45.5	42.56	9.87	9.38

Table 4.2: Recovery Times of a_m_s Programs

simple memory-to-memory copy. Obviously, most programs fall between these two extremes. The data from the real programs in this chapter show that compression can indeed be beneficial to checkpointing, especially with FORTRAN programs that tend to overestimate their program's needs.

4.2.8 Recovery

The recovery procedure for all checkpointing algorithms and optimizations is the same: Read the contents of the address space from memory, replay messages from the message log, and restart the processor. The data in Table 4.2 shows that like checkpoint time, recovery time is proportional to the size of the checkpoint files, and is entirely dependent on the disk read times. In the situations where compression is successful and reduces the size of the checkpoint file, compression is also beneficial to recovery time.

4.2.9 Results with Real Programs

The rest of this chapter is devoted to showing results of ICKP on real programs written for the iPSC/860. While the programs are all quite different from `a_m_s`, the performance of ICKP on most of them is analogous to one of the above instances of `a_m_s`. A combination of the size and compressibility of the address space, and the synchronizing behavior of the messages usually determines which algorithm performs checkpointing with the least amount of overhead. The details of each program along with ICKP's performance is presented below:

A Genetic Algorithm Tool

This is a program that creates color bitmaps for visualizing the rate of convergence and basins of convergence for genetic algorithms. It is a C program written by Michael Vose at the University of Tennessee, and is much like the `SMALL-IND` instance of `a_m_s`: The nodes cooperate in the beginning of the program to distribute data, and then they run without sending any messages to one another until the program is finished. Each node periodically writes output to a CFS file, and at the end of the program, node 0 concatenates all the output files into one. All input files are read-only, and all output files are write-only. Thus, the program adheres to ICKP's restrictions concerning file I/O.

Figure 4.17 shows the results of all the ICKP algorithms checkpointing test runs of the program. The parameters of these test runs create a 16x16 output bitmap and produce a run which takes about a minute to complete in the absence of checkpointing. More typical runs of this program create a 256x256 output bitmap, and take between four and five hours to complete. However, the program's synchronization behavior and the size of its address space are the same for both short and long runs. Thus, the data presented here represents valid data for a parallel program which benefits greatly from checkpointing.

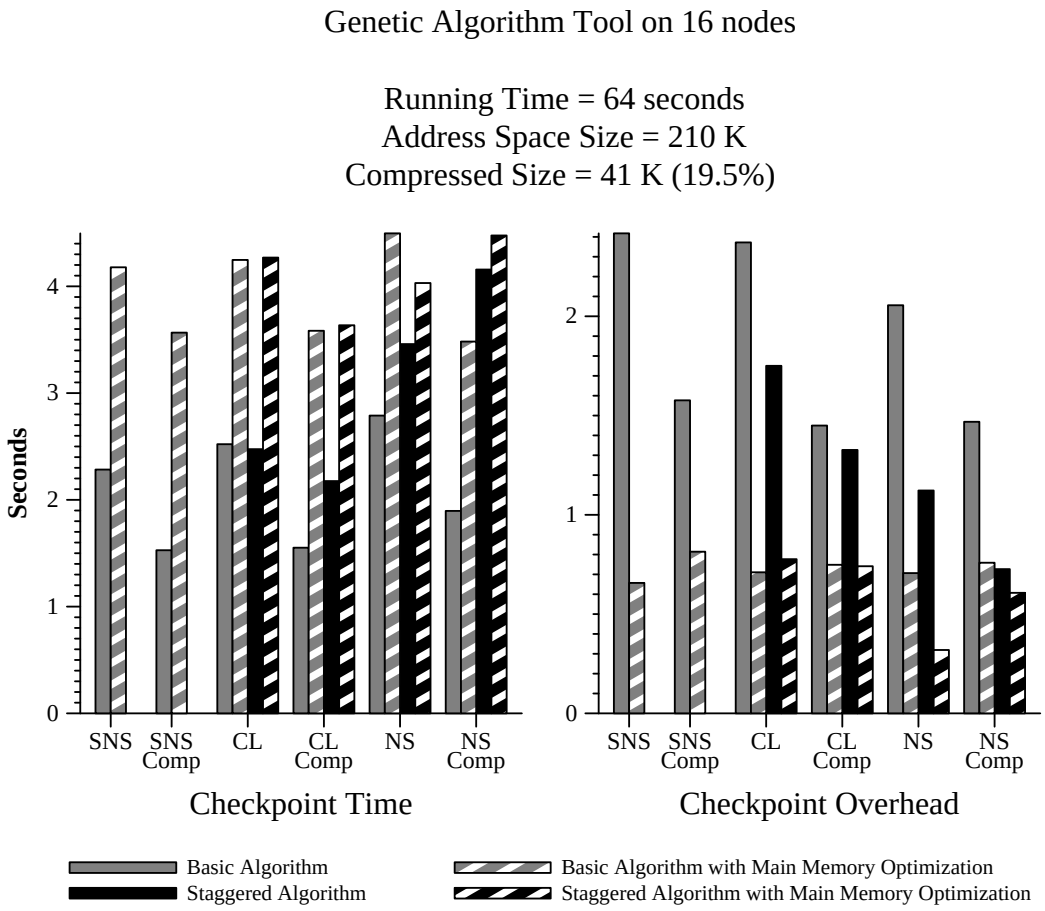

Figure 4.17: Results of ICKP on the Genetic Algorithm Tool

The graphs in Figure 4.17 are very similar to `SMALL-IND`'s graphs in Figure 4.14. Since the address space is very small, main memory checkpointing is quite beneficial to the checkpoint overhead, but compression, even though effective in reducing the checkpoint size, makes little difference to the checkpoint overhead. As the nodes execute independently, staggering is also beneficial to the checkpointing overhead. From the data in this figure, the non-compressed, staggered, and main memory optimizations of the Network Sweeping algorithm give the best results for checkpointing.

Sparse Matrix Solver

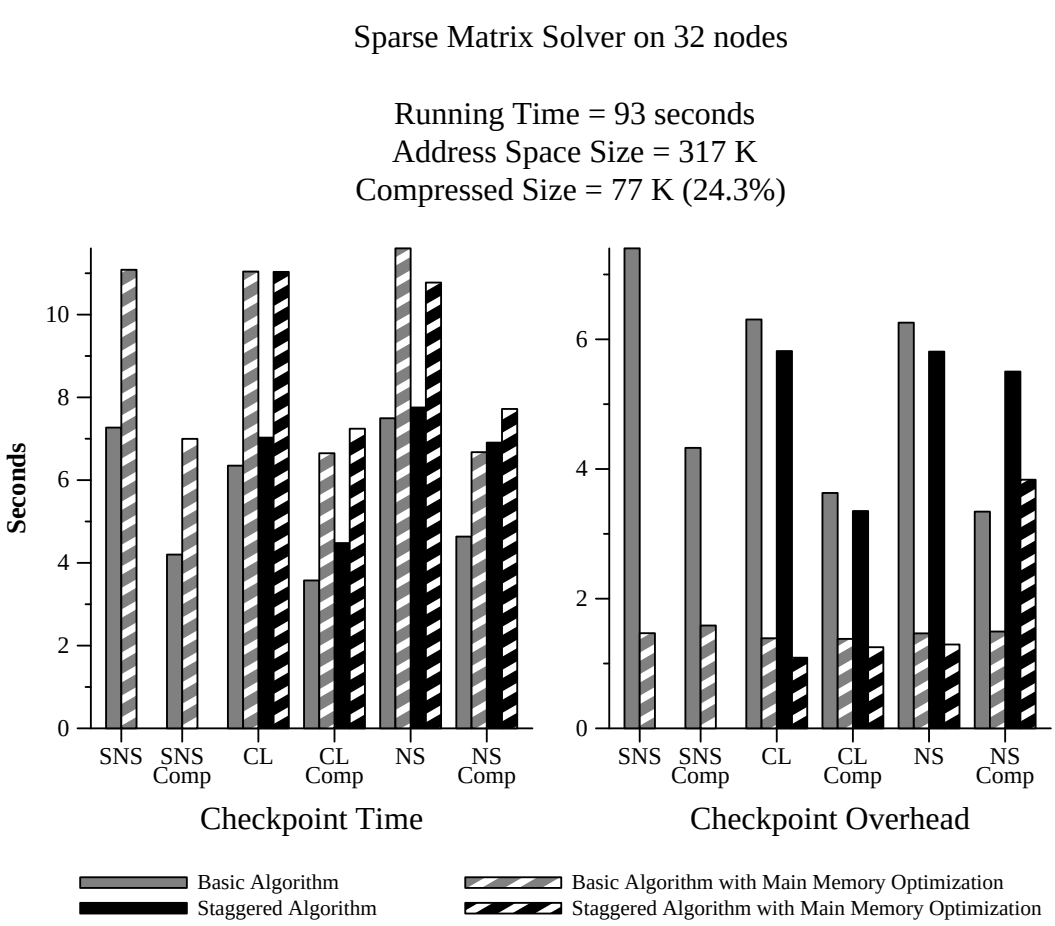

Figure 4.18: Results of ICKP on the Sparse Matrix Solver

This program solves a linear system of equations represented by a sparse matrix. It was written at Rice University to test checkpointing on distributed systems [EJZ92]. Very little work was required to convert it to run on the iPSC/860. The graph in Figure 4.18 shows results of ICKP checkpointing an instance of the program that solves a sparse matrix of 2,000 equations, where the matrix is less than 10% full. The test is on 32 nodes.

The pattern of communication in this program is like that in `SMALL-SYNC`: Each processor is given a subset of the unknown variables, and then iterates solving for that subset, and then sending the new values to the other processors. Accordingly, the data in Figure 4.18 is also like `SMALL-SYNC`. Compression improves the checkpoint time, but not the overhead. Main memory

checkpointing is greatly beneficial to the checkpoint overhead. Staggering is of little help in all cases.

Gaussian Elimination

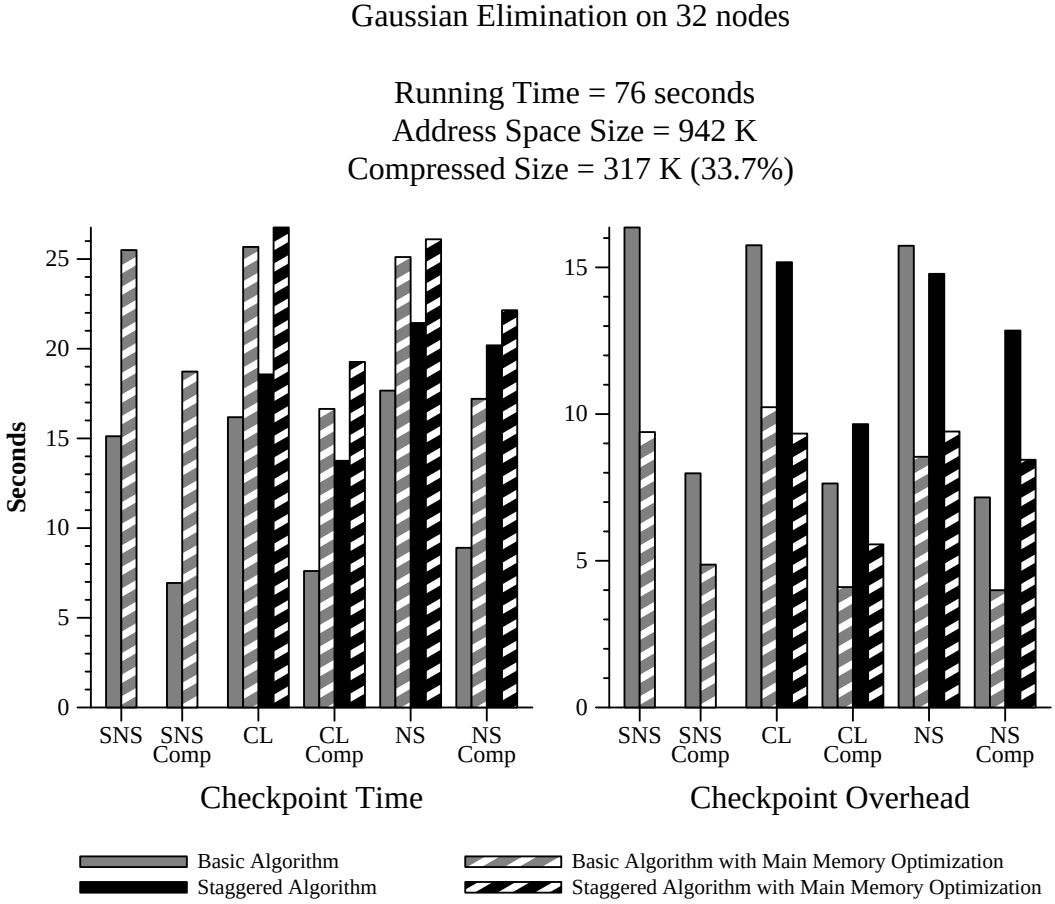

Figure 4.19: Results of ICKP on the Gaussian Elimination Program

This is another program written at Rice University and ported to the iPSC/860. It performs Gaussian elimination with partial pivoting on a given input matrix. The data in Figure 4.19 shows the performance of ICKP on an instance of the program running on a 1500-row square matrix. The program's pattern of communication is as follows: The nodes read in the matrix and distribute the columns evenly among themselves. Then, at each step of the reduction,

the processor that holds the pivot element broadcasts that column to the other processors.

The graphs in Figure 4.19 are similar to the `MED-SYNC` graphs in Figure 4.15. The checkpoint fits into main memory, and can be significantly compressed, thereby allowing the main memory and compression optimizations to improve the checkpoint overhead. The pattern of communication is neither synchronizing nor independent; however, it seems bad for staggering checkpoints. The reason is because of message logging, as described in Section 3.4. With staggering in the CL and NS algorithms, there is often a node that broadcasts the pivot column before checkpointing, to other nodes that receive the message after checkpointing. Thus, the message has to be logged. In the staggered version of the NS algorithm, over 300 Kbytes of messages sometimes had to be logged, thereby defeating the purpose of the staggering optimizations.

Fast Fourier Transform

This is yet another part of the Rice program suite. It performs a Fast Fourier Transform on 25,000 points. Each processor works independently on a subset of the points, and there is no communication during the nodes' computation. Thus, as one would expect, the data in Figure 4.20 looks much like the data in the `MED-IND` instance of Figure 4.14, except that due to the poor compression ratio, the compression optimization does not improve the checkpoint time or overhead. The poor compression comes from the fact that most of the address space is the data points of the FFT, which are essentially random floating point numbers.

Molecular Dynamics

This is a FORTRAN program given to us by researchers at Oak Ridge National Laboratory. The code calculates dynamics of molecules on a lattice over a given number of timesteps. The data in Figure 4.21 shows the results of checkpointing an instance of the code calculating 200 timesteps on a $11 \times 11 \times 11$ lattice. As in the `a_m_s` examples, we fixed the size of physical

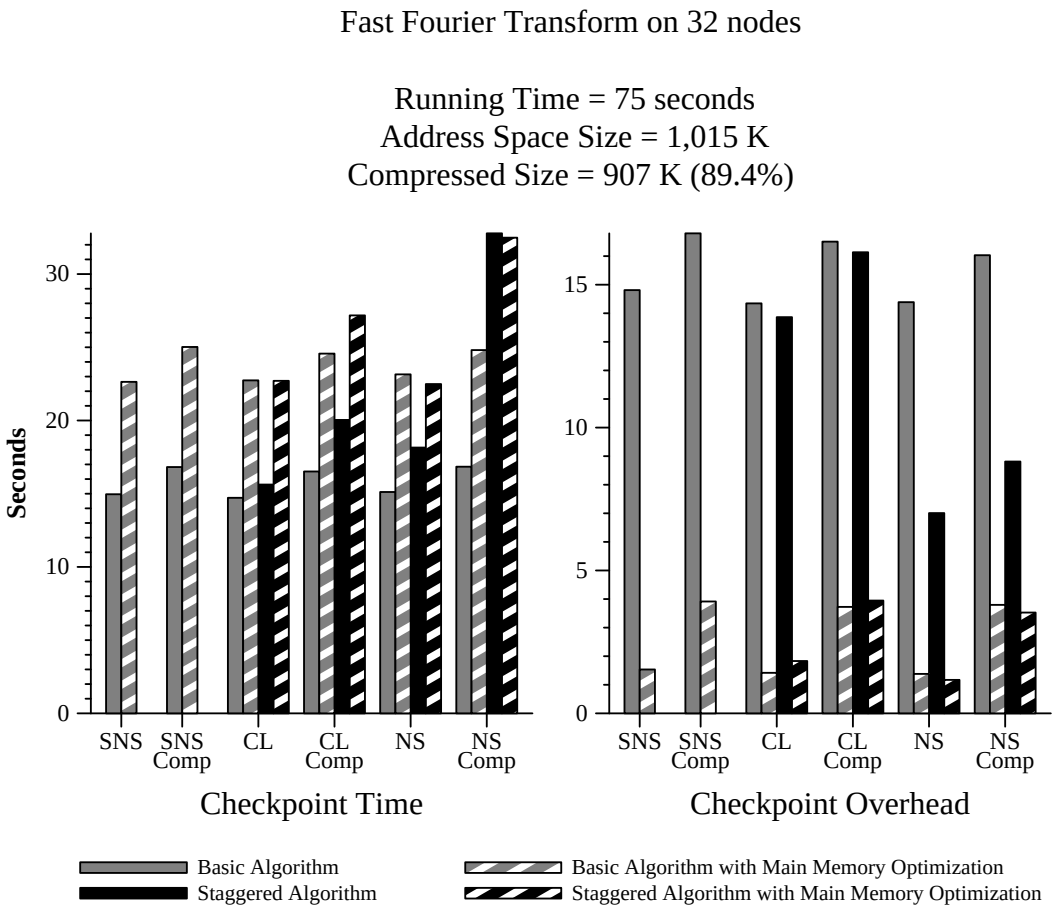

Figure 4.20: Results of ICKP on the Fast Fourier Transform

memory at 4 Mbytes, so that the main memory checkpoint would not fit into physical memory. The results are analogous to the `LARGE-SYNC` results in Figure 4.15. Besides being tightly synchronized, the message-sending behavior of this program is like the Gaussian Elimination program: In the staggered versions of the CL and NS algorithms, node 0 often had to log up to 62 Kbytes of messages. The non-staggered versions cause no messages to be logged, nor any extra waiting due to synchronization message, and therefore display less overhead.

One of the amazing results of this program is the 4.6% compression ratio of the checkpoint. This can be attributed to the quirk of FORTRAN programming described in Section 4.2.7: To allow for larger problem sizes, the

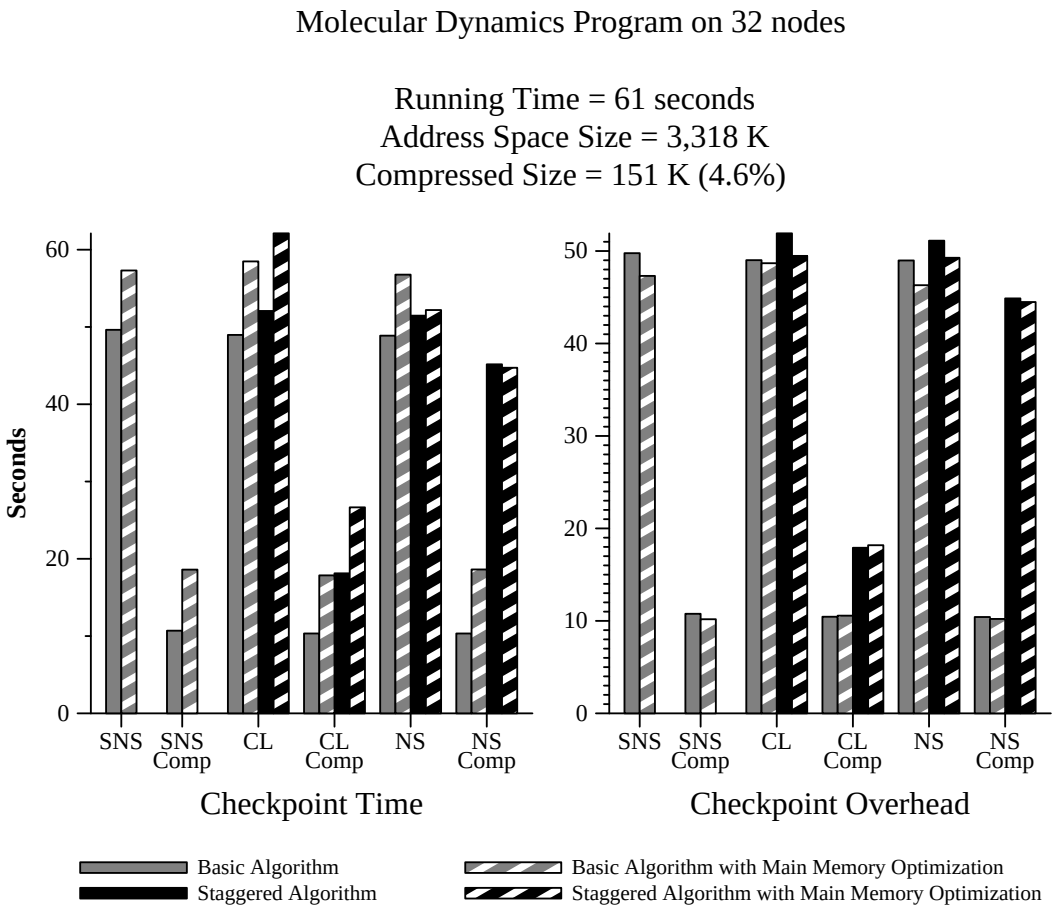

Figure 4.21: Results of ICKP on the Molecular Dynamics Program

programmer statically allocates huge arrays that end up being uninitialized, and therefore compress away to almost nothing.

An Iterative Equation Solver

This last example is another FORTRAN program. It was written by Viktor Eijkhout at the University of Tennessee, and it uses iterative methods to solve a system of equations. The data in Figure 4.22 shows the results of checkpointing an instance of the code with a 700×500 matrix of equations. The results are very similar to the Molecular Dynamics results: The address space is huge, but can be compressed away to less than a megabyte. This one is also

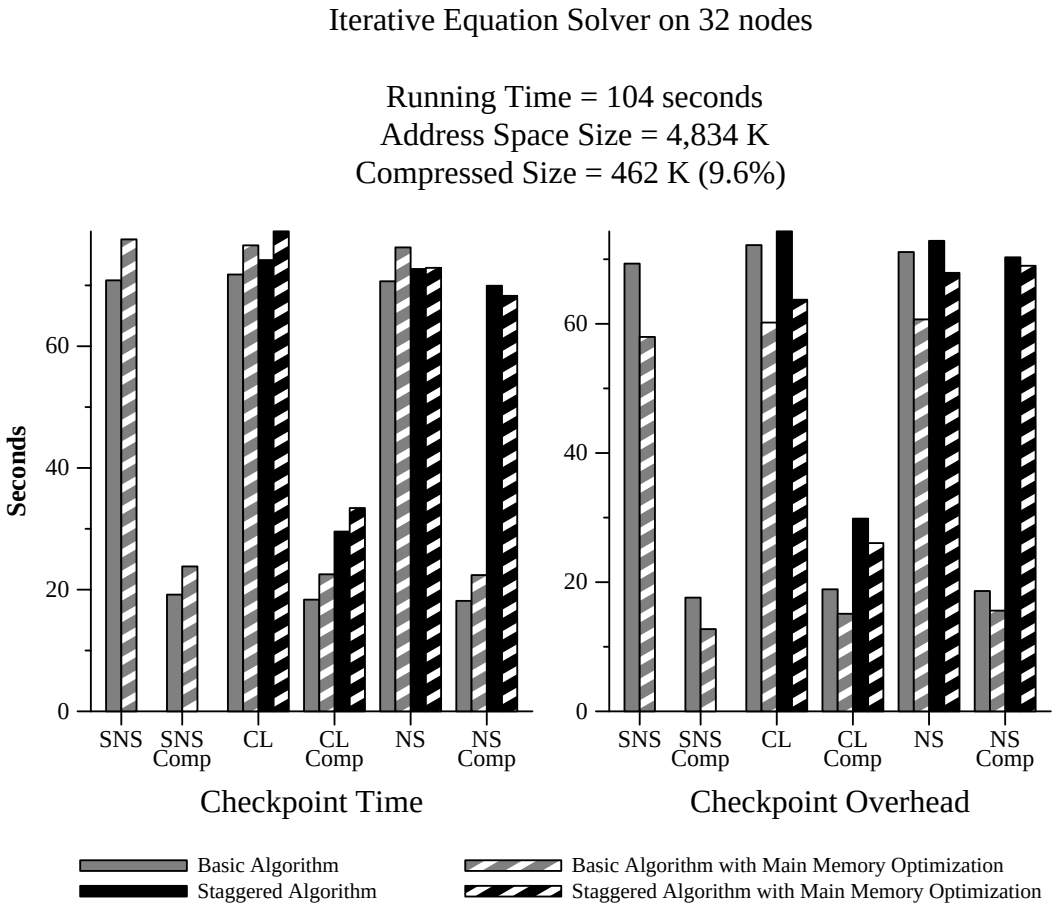

Figure 4.22: Results of ICKP on the Iterative Equation Solver

like `LARGE-SYNC` in that the processors synchronize often, thereby degrading the performance of the staggering optimizations. Message logging was not a problem, as a maximum of 208 bytes were logged during any one checkpoint.

4.2.10 Conclusions

The implementation of checkpointing has allowed us to evaluate the three consistent checkpointing algorithms and their optimizations on a large variety of parallel programs. What we can conclude from this is that the characteristics of the program largely determine which algorithm and optimization works the best. Since all of them checkpoint in a reasonable period of time (under a

Size	Com-press-ible	Synchro-nization	Message Size	Best Algorithm(s) / Optimizations				Example Program
				Algorithm	Main Memory	Stag-gered	Com-pressed	
Small	-	Loose	-	NS	Yes	Yes	No	SMALL-IND SMALL-IND-RAND Genetic Alg. Tool
Small	-	Tight	-	NS or CL	Yes	No	No	SMALL-SYNC SMALL-SYNC-RAND Sparse Mat. Solver
Medium	Yes	Loose	Small	NS	Yes	Yes	Yes	MED-IND
Medium	No	Loose	Small	CL or NS	Yes	No	No	MED-IND-RAND FFT
Medium	Yes	Loose	Large	CL or NS	Yes	No	Yes	Gauss. Elimination
Medium	No	Loose	Large	CL or NS	Yes	No	No	-
Medium	Yes	Tight	-	CL or NS	Yes	No	Yes	MED-SYNC
Medium	No	Tight	-	CL or NS	Yes	No	No	MED-SYNC-RAND
Large	Yes	Loose	Small	NS	Yes	Yes	Yes	LARGE-IND
Large	Yes	Loose	Large	CL or NS	Yes	No	Yes	Mol. Dynamics
Large	Yes	Tight	-	CL or NS	Yes	No	Yes	LARGE-SYNC Iterative Eq. Solver
Large	No	Loose	Small	NS	Yes	Yes	No	LARGE-IND-RAND
Large	No	Loose	Large	CL or NS	Yes	No	No	-
Large	No	Tight	-	CL or NS	Yes	No	No	LARGE-SYNC-RAND

Table 4.3: Program Characteristics and their Ideal Checkpointing Algorithms

minute or two), we are most concerned with the checkpoint overhead. Table 4.3 gives a compilation of the program characteristics that seem to determine the behavior of checkpointing, and which algorithm or algorithms seem to give the lowest overhead. We also include examples of programs that exhibit these characteristics.

The conclusions can be summarized as follows:

- The Network Sweeping is clearly the best algorithm for all applications. It does not halt the entire multicomputer to take a checkpoint, as does the Sync-and-Stop algorithm, and it allows for the greatest amount of staggering when staggering is beneficial. Moreover, as the number of processors increases for massively parallel machines, its checkpoint time and overhead will not be adversely affected by the number of marker messages to send, as will the Chandy-Lamport algorithm.

- The main memory algorithm, especially as implemented here, without support from the backing store of virtual memory, is quite successful at reducing overhead.

- Staggering the disk writes is clearly a good idea, as it drastically reduces the processors' serialization at the disks. When the program's behavior is not characterized by synchronization or large messages, this staggering can be effected by the Network Sweeping algorithm much more so than the other algorithms. When the main memory checkpoint fits into physical memory, any algorithm can stagger the writes, regardless of the behavior of the target program.

- Although we have not yet implemented the CLL algorithm, the results of the multiprocessor implementation suggest that it would be doubly beneficial to any of the checkpointing algorithms. First, since it imposes a small initial stop time, it would not be penalized by programs with tight synchronization or large messages. Second, since it is an algorithm that naturally spreads out the disk writes over time, it should help in reducing contention for the disk. Thus, a combination of the Network Sweeping algorithm and the CLL algorithm should prove ideal for multicomputer checkpointing.

- Compression is good for certain programs, and bad for certain programs. Users should be able to predict when compression will benefit their programs, and accordingly should only use it in those cases.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The preceding chapters have presented a thorough discussion of checkpointing algorithms and speed optimizations for multiprocessors and multicomputers. With the implementations in Chapter 4, we have shown that we can provide efficient fault-tolerance for MIMD architectures, thereby enhancing their general usability. Specifically, we draw the following conclusions about checkpointing algorithms and their speed optimizations:

5.1.1 Algorithms

For multiprocessor checkpointing, there is only one basic algorithm for checkpointing, which proves successful in providing fault-tolerance on the Firefly. For multicomputer checkpointing, we studied three basic algorithms: Sync-and-Stop, Chandy-Lamport, and Network Sweeping. The latter two algorithms outperform the Sync-and-Stop algorithm in all cases as they do not freeze up the entire system like the Sync-and-Stop algorithm does. The Network Sweeping algorithm is more efficient than the Chandy-Lamport algorithm when the target program benefits from staggering the checkpoints, as Network Sweeping staggers them to a greater degree. As the number of processors

grows, the Network Sweeping will further outperform the Chandy-Lamport algorithm, as it sends far fewer marker messages.

5.1.2 Reducing Checkpoint Time

As stated in the Introduction, the main way to decrease checkpoint time is to reduce the amount of data written to disk. Previous research [FB89, EJZ92] shows that incremental checkpointing is one way to reduce the checkpoint size. Our implementation shows that on-line compression is another alternative. In our multiprocessor implementation, the CLL algorithm reduces the checkpoint time from the other optimizations by starting the disk writes as soon as possible, instead of waiting for a complete copy of the checkpoint to exist in main memory.

The other optimizations all increase the checkpoint time in order to decrease the overhead or latency. The experimental results reported in Chapter 4 show that this is a reasonable tradeoff. Except for when the optimizations thrash virtual memory, they never increase the checkpoint time by a large factor. Even when writing 128 megabytes of checkpoint data to disk as in the Iterative Equation Solver of Section 4.2.9, the checkpoint time never increases to over two minutes. As we envision that users will checkpoint in sparse intervals, such as once every ten minutes or once an hour, these checkpoint times are indeed adequate.

5.1.3 Reducing Checkpoint Overhead

Checkpoint overhead is the most important component of a checkpointer's speed, as it measures how much the checkpointer affects the target's running time. The main-memory optimization, especially as implemented in the CLL algorithm, is the most effective at reducing the checkpoint overhead, as it substitutes memory-to-memory copy time for disk write time as the major component of the overhead. In multicomputer checkpointing, staggering the disk writes to reduce contention on shared resources is also a good way of

reducing overhead. The Network Sweeping algorithm allows for effective staggering with certain types of program behavior (loose synchronization, small messages). Main memory checkpointing allows for complete staggering when the checkpoint fits into physical memory. The CLL algorithm should be effective in staggering disk writes in all cases.

5.1.4 Reducing Latency

Latency is a measure of the real-time invasiveness of a checkpointer. The copy-on-write and CLL optimizations both use a machine's virtual memory hardware to intersperse the taking of the checkpoint with the running of the target program. Both optimizations are successful in interrupting the target only for small, fixed periods of time. In multiprocessor implementation, the latency was kept under a tenth of a second. The copy-on-write and CLL optimizations should both improve multicomputer latency as well.

5.1.5 Uses

Both the multicomputer and multiprocessor implementations show that our checkpointing algorithms can be useful for fault-tolerance. For example, suppose a user's iPSC/860 program runs on 32 nodes for a week, and uses four megabytes per node. Regardless of the algorithm used and the behavior of the program, the checkpoint time and overhead will be less than two minutes. Therefore, if the user checkpoints once every hour, then the program's running time will be increased by no more than five and a half hours. If the checkpoint can be compressed into main memory, then the data from Figure 4.15 suggests that the checkpoint overhead will be around twenty seconds per checkpoint. This means that the running time of the program will be increased by under an hour, which is less than 0.6% of the total running time.

For job-swapping, the data presented in Chapter 4 suggest that our checkpointers should be useful for coarse-grained job-swapping. On the iPSC/860, this is very beneficial, as there are no other facilities provided by NX/2 for

job-swapping. Without ICKP, users are very reluctant to try running long programs, as most machines are widely shared, and hogging a significant portion of processors for several hours is usually discouraged. ICKP makes programs like the genetic algorithm tool easier to use in a considerate manner as it enables them to be swapped off the iPSC/860 during periods of high use.

Fine-grained job-swapping requires low latency. On multicomputers, a combination of the network sweeping algorithm and the CLL algorithm can be used to implement such a swapping mechanism. Once the target address spaces have been protected, the processor can start serving a process of the second program. Protected memory pages can be moved to the CLL buffer and then to backing stores on demand. Thus, the latency of job-swapping is similar to the checkpoint latency. The overhead depends on memory requirements.

Program migration is much like job-swapping, except that the checkpoint need not be written to disk. Instead it must be sent as a message to the second machine. This is a much faster operation than disk writing. For example on the Paragon, the bandwidth of messages is 200 Mbytes per second, whereas the speed of CFS disk writes is around four Mbytes per second. These fast speeds combined with main memory or CLL checkpointing should reduce checkpoint time and overhead significantly enough for checkpointing to be an effective means of program migration.

5.2 Future Work

We anticipate doing two kinds of research in the future: modifications and extensions of work done in this dissertation, and working on new types of checkpointing. In the former case, we plan to implement incremental checkpointing on all of our platforms, and to install the copy-on-write and CLL optimizations into the multicomputer implementation. This will give us a more complete picture of how the checkpointing optimizations work and interact with each other.

More importantly, we plan to port the iPSC/860 implementation to larger multicomputers, such as the Delta and Paragon, which can have up to 570 and 1024 processors respectively. Here, both file compression and incremental checkpointing will be imperative, as the amount of data comprising each global checkpoint will be staggeringly large—up to 16 gigabytes. Moreover, the difference between the Chandy-Lamport and Network Sweeping algorithms will become more evident.

Another platform on which to test checkpointing is *Shared Virtual Memory (SVM)* [LH89]. Here, the underlying machine is a multicomputer, which supports only message passing, but the programming environment implements shared memory using the page tables and fault handlers of each machine to manage the shared address space. Paging is performed from processor to processor instead of from processor to disk. Thus, a checkpointer can take advantage of the shared memory model to checkpoint as on a multiprocessor, even though the underlying machine is a multicomputer. Moreover, replicated pages on different processors may be identified by the checkpointer, and only one of them need be written as part of the checkpoint file. Thus, global checkpoints should be both smaller and simpler when taken as part of an SVM system than as a generic application on the multicomputer.

5.2.1 $N+1$ Parity

$N+1$ parity is an idea that has been used to add reliability to disk arrays [Gib90]. It can also be used to checkpoint a multicomputer system given the stipulation that only one processor fails at any point in time. The definition of $N+1$ parity is as follows: Suppose there are N pages of memory, $P_1, P_2, \ldots P_N$, all of which are m bytes long. We designate the i-th byte of page P_j as P_j^i. The *parity page*, P_{N+1}, of these N pages is defined to be the bytewise exclusive or of P_1 through P_N. Specifically:

$$P_{N+1}^i = P_1^i \oplus P_2^i \oplus \ldots \oplus P_N^i \text{ for } 1 \leq i \leq m.$$

Now, if we define the set P to be $P_1 \cup \ldots \cup P_N \cup P_{N+1}$, then P has the following interesting property: Any subset of P with exactly N pages can be used to construct the remaining page. Specifically:

For any j between 1 and $N+1$, $P_j = P_1 \oplus \ldots \oplus P_{j-1} \oplus P_{j+1} \oplus \ldots \oplus P_N \oplus P_{N+1}$.

$N+1$ parity can be used for checkpointing in the following way: Suppose that for every processor p in a multicomputer, p's local checkpoint fits into main memory. Then, when taking a global checkpoint, each processor simply creates its main-memory checkpoint, and then cooperates with the rest of the processors to make the *parity checkpoint*. This is done by having each processor pad its main memory checkpoint so that all checkpoints are the same size, and then making all processors cooperate to take the bytewise exclusive or of their checkpoints. The result is the parity checkpoint, which is analogous to the parity page above. The parity checkpoint is written to disk, and the processors retain their main-memory checkpoints in physical memory. If a processor fails, then its checkpoint can be recreated from the other processors' main memory checkpoints and the parity checkpoint.

If each checkpoint is m bytes long and there are N processors in the system, then $N+1$ parity writes only m bytes to disk, whereas normal checkpointing writes Nm bytes. Thus, $N+1$ parity should improve checkpoint time by a factor of N. This is at the expense of needing enough physical memory for each processor to keep a copy of its checkpoint resident in memory.

We plan to implement checkpointing using $N+1$ parity on the Intel Delta and Paragon.

5.2.2 User-Defined Checkpointing

One of the ways to simplify the task of checkpointing, remove the restrictions imposed by the operating system, and reduce the amount of information written to disk is to allow users to give the checkpointer directives concerning their program's behavior. For example, users could mark certain places in their code as "safe" places to checkpoint. These would be places where, for

example, there are no open files or sockets. Moreover, users could mark certain pieces of memory as temporary—that is, at the safe places to checkpoint, these pieces of memory need not be saved as part of the checkpoint.

The assumption here is that programmers often make use of large chunks of temporary storage, which at certain points in a program do not contain meaningful data. For example, they may get periodically re-initialized from disk, or recalculated from other program data. Moreover, there is an assumption that while there might be points in a user's program where the restrictions imposed by the checkpointer are broken, these points might be small in duration, and thus with a few hints from the user, the checkpointer may avoid them. Finally, with multicomputer checkpointing, the user may prohibit checkpointing to occur while sending large amounts of data, thereby decreasing the potential size of the message log.

We plan to define programming constructs and libraries for user-defined checkpointing, first for uniprocessor checkpointing and then for multiprocessor and multicomputer checkpointing. To implement these, we anticipate having to rewrite the memory allocator, enabling it to mark certain chunks of memory as temporary, and to reallocate specific chunks of memory on demand (for recovery). Some work has already been completed to write this allocator.

5.2.3 Shared Single-Level Store

A more long-range result of work in checkpointing is to develop a programming environment based on the concept of a *shared single-level store*. That is, for main memory to be both persistent and sharable by different processes and users. In such a system, file caching is taken to an extreme—files are simply mapped into main memory, and are persistent and sharable because main memory is persistent and sharable.

We envision being able to implement a shared single-level store on a multicomputer system using Shared Virtual Memory and checkpointing ideas to make memory both sharable and persistent.

Bibliography

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, Nov 1986.

[AEL88] A.W. Appel, J.R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. In *The 17th Annual International Symposium on Computer Architecture*, pages 148–159, Seattle, Washington, May 1990.

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

[AL81] T. Anderson and P. A. Lee. *Fault Tolerance Principles and Practice*. Prentice/Hall International, Inc., Englewood Cliffs, New Jersey, 1981.

[BAD+92] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file system. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, October 1992.

[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.

[BBG+89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrman, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions of Computer Systems*, 7(1):1–24, Feb 1989.

[BJLM92] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. I believe this is a DEC SRC Tech Report – Kai says it will also be an ASPLOS 92 paper., 1992.

[BWC89] T. Bell, I.H. Witten, and J.G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–589, December 1989.

[CJ91] Flaviu Cristian and Farnam Jahanain. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 12–20, October 1991.

[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.

[Dal90] William J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[Dal91a] 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.

[Dal91b] William J. Dally. Fine-grain concurrent computing. *Research Directions in Computer Science: An MIT Perspective*, pages 127–154, 1991.

[DKO+84] David J. DeWitt, Randy H. Katz, Frank Olken, Lenard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–8, June 1984.

[DS87] 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.

[EJZ92] Elmootazbellah Nabil Elnozahy, David B. Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, 1992.

[FB89] 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.

[FR86] R. Fitzgerald and R.F. Rashid. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[Gib90] Garth Alan Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, University of California, Berkeley, December 1990.

[GLL+90] Kourosh Gharchorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *The 17th Annual International Symposium on Computer Architecture*, pages 148–159, Seattle, Washington, May 1990.

[Hag86] Robert B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computers*, C-35(9):839–843, September 1986.

[Joh89] David B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Rice University, December 1989.

[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.

[KMBT91] M. Frans Kaashoek, Raymond Michiels, Henfy E. Bal, and Andrew S. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. Technical Report IR-258, Vrije Universiteit, Amsterdam, October 1991.

[KS86] Henry F. Korth and Abraham Silberschatz. *Database System Concepts.* McGraw-Hill, New York, 1986.

[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.

[Lam81] B.M. Lampson. Atomic transactions. In B.M. Lampson, M. Paul, and H.J. Siegert, editors, *Distributed Systems – Architecture and Implementation*, pages 246–264. Springer-Verlag, 1981.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *TOCS*, 7(4):321–359, Nov 1989.

[LLG+90] 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.

[LN88] Kai Li and Jeffrey F. Naughton. Multiprocessor main memory transaction processing. In *Proceedings of the International Symposium on Database in Parallel and Distributed Systems*, pages 177–187, Dec 1988.

[LNP90] Kai Li, Jeffrey Naughton, and James 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.

[LNP91] Kai Li, Jeffrey F. Naughton, and James S. Plank. Checkpointing multicomputer applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 1–11, October 1991.

[LNP92] Kai Li, Jeffrey F. Naughton, and James S. Plank. An efficient checkpointing method for multicomputers with wormhole routing. *International Journal of Parallel Processing*, 20(3), June 1992.

[LS92] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the Unix kernel. In *Conference Proceedings, Usenix Winter 1992 Technical Conference*, pages 283–290, January 1992.

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

[MS87] P.R. McJones and G.F. Swart. Evolving the unix system interface to support multithreaded programs. Tech Report 21, DEC Systems Research Center, Sep 1987.

[Nug88] Steven F. Nugent. The iPSC/2 direct-connect communications technology. In *Proceedings of the Third Hypercube Conference*. ACM, November 1988.

[OCD+88] J.K. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[Pie88] Paul Pierce. *The NX/2 Operating System*, pages 51–57. Intel Corporation, 1988.

[Pie89] Paul Pierce. A concurrent file system for a highly parallel mass storage subsystem. In *Proceedings of the 4th. Conference on Hypercubes, Concurrent Computers and Applications*, volume I, pages 155–160, March 1989.

[PL88] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution of parallel programs. In *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, May 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.

[Pu85] Calton Pu. On-the-fly, incremental, consistent reading of entire databases. In *Proceedings of the Eleventh International Conference on Very Large Databases*, pages 369–375, Stockholm, Sweden, August 1985.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.

[RLW85] Paul Rovner, Roy Levin, and John Wick. On extending modula-2 for building large, integrated systems. Research report 3, DEC Systems Research Center, 1985.

[Sei88a] 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.

[Sei88b] C. L. Seitz *et al.* Submicron systems architecture project semiannual technical report. Technical Report TR-88-18, Caltech Computer Science Tech. Rep., November 1988.

[SGM87] Kenneth Salem and Hector Garcia-Molina. Checkpointing memory-resident databases. Technical Report CS-TR-126-87, Department of Computer Science, Princeton University, 1987.

[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.

[SP88] Abraham Silberschatz and James L. Peterson. *Operating Systems Concepts*. Addison-Wesley, Reading, Mass., 1988.

[Sta89] Mark E. Staknis. Sheaved memory: Architectural support for state saving and restoration in paged systems. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–103, April 1989.

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

[TLC85] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 2–11, Orchas Island and Washington, December 1985.

[TS87] C. P. Thacker and L. C. Stewart. Firefly: a multiprocessor workstation. In *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, Oct 1987.

[TW86] David J. Taylor and Michael L. Wright. Backward error recovery in a UNIX environment. In *16th Annual International Symposium on Fault-Tolerant Computing Systems*, pages 118–123. IEEE Computer Society, July 1986.

[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.

[Wit89] Larry D. Wittie. Debugging distributed C programs by real-time replay. *SIGPLAN Notices*, 24(1):57–67, January 1989.

Appendix A

Implementational Details of ICKP

As stated in Chapter 4, the iPSC/860 presents a somewhat different model of computation than the one used Chapter 2. The reason is that iPSC/860 messages are buffered in the kernel until the user process extracts them, thereby hiding the FIFO ordering of messages with different types. This section presents the mechanism implemented in ICKP that allows us to deduce a FIFO ordering for all messages, regardless of their types.

A.1 The M_{EXTRA} Message

Our solution to the kernel buffering problem is to send two messages for every user message. The first has a special reserved type, and is called M_{EXTRA}. Its contents consist of the type of the user message. The second message is the user message itself. With the M_{EXTRA} message, `crecv`, `irecv` and `hrecv` can all deduce in which order their messages have been received in the kernel by observing the order of the M_{EXTRA} messages. In the Chandy-Lamport and Network Sweeping algorithms, the m_{CL} and m_{NS} messages are implemented as M_{EXTRA} messages whose contents are reserved types. Therefore, ICKP can

use the ordering of the M_{EXTRA} messages to determine which messages to log, and which messages not to log.

The overhead imposed by these M_{EXTRA} messages is that the total number of messages sent by an application program is doubled. However, the M_{EXTRA} messages are all short (4 bytes), and therefore take a small period of time to send: around 0.001 seconds. In practice, the overhead imposed by M_{EXTRA} messages on the test programs of Chapter 4 never exceeds 1% of the total running time.

A.2 Extracting Message State from the Kernel

When ICKP has a node take a local checkpoint, there are three types of information in the kernel that must be extracted as part of the checkpoint:

- The buffered messages that have not been `recv`'d by the user process.
- `Irecv` requests. In these, the user's program has called `irecv`, and either the message has not been received by the kernel, or the user has not tested whether the `irecv` has completed.
- `Hrecv` requests. Here, the user has called `hrecv` for a certain message type, and either the message has not been sent, or it has, but the `hrecv` handler cannot be invoked until ICKP relinquishes control (this is because ICKP gains control via a `hrecv` handler).

ICKP extracts this information in a straightforward way. At checkpoint time, ICKP copies all messages that are in the system buffers to a queue in user space (this can be done by calling `crecv` for each message type). This queue is saved in the checkpoint as part of the processor's address space. Thus, whenever a node executes `crecv`, `irecv` or `hrecv`, it first checks the user queue for the message. If present in the user queue, the message is taken from that queue. Otherwise, the correct `recv` system call is made.

To deal with `irecv` and `hrecv` calls at checkpoint time, the checkpointer maintains a queue of pending `irecv` and `hrecv` calls in user space. At checkpoint time, ICKP tests all the `irecv`'s for completion, and marks the result in the queue. Upon recovery ICKP uses this information to decide whether to redo the `irecv` call: It makes a new `irecv` call for the `irecv`'s that were not completed at checkpoint time. No new call is made for `irecv`'s that were completed.

`Hrecv` calls are more subtle. ICKP uses the M_{EXTRA} messages to determine whether `hrecv` messages have been received at checkpoint time. If not, ICKP simply remakes the `hrecv` call upon recover. However, if an `hrecv` message has been received at checkpoint time, and is waiting for ICKP to finish before invoking the handler, then ICKP must remember that in the checkpoint, so that it may invoke the handler immediately following recovery.

Thus, the message state present in the kernel may be extracted and saved as part of the checkpoint, and rebuilt at recovery time.

Appendix B

Tables for the Multiprocessor Implementation

Heap Size (Mbytes)	Checkpoint Time (sec)			
	Sequential	Main Memory	CLL	Copy-on-Write
0.789	3.64	6.58	5.54	7.13
2.985	10.20	18.03	14.00	19.62
3.962	13.02	22.70	17.24	24.43
4.938	15.81	27.52	21.08	30.52
7.868	24.96	44.83	32.41	46.70
9.821	30.22	57.32	39.83	62.47
12.751	40.48	673.89	52.53	474.13

Table B.1: Checkpoint Time of Multiprocessor Algorithms

Heap Size	Checkpoint Overhead (sec)			
(Mbytes)	Sequential	Main Memory	CLL	Copy-on-Write
0.789	3.53	2.34	0.60	1.40
2.985	10.28	6.12	3.84	4.42
3.962	12.76	7.25	3.36	3.85
4.938	15.73	9.70	3.63	5.66
7.868	24.67	16.39	4.31	6.08
9.821	30.03	19.16	4.38	7.79

Table B.2: Checkpoint Overhead of Multiprocessor Algorithms

Heap Size	Latency Data (sec)	
(Mbytes)	Initial Stop Time	Maximum Page Fault Time
0.789	0.03	0.010
2.985	0.10	0.023
3.962	0.12	0.025
4.938	0.15	0.037
7.868	0.23	0.055
9.821	0.28	0.053
12.751	0.34	0.064

Table B.3: Latency Data of Multiprocessor Algorithms

Page Size (Kbytes)	Number of Faults	Maximum Page Fault Time (sec)
1	920	0.03
2	599	0.03
4	371	0.05
8	208	0.05
16	113	0.08
32	69	0.22
64	40	0.31
128	15	1.24

Table B.4: Data Concerning Page Size of the CLL Algorithm

Test Program	Heap Size (Mbytes)	Checkpoint Time (sec)	Checkpoint Overhead (sec)
Travelling Salesman	0.064	2.36	0.17
Matrix Multiplication	0.513	3.91	0.07
Pattern Match	1.233	6.37	0.07
Bubble Sort	3.000	14.71	0.75

Table B.5: Data of Other Test Programs

Appendix C

Tables for the Multicomputer Implementation

Algorithm	Main Memory	Com-pressed	Stag-gered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	3.76	3.82
SNS	Yes	No	-	7.11	0.86
SNS	No	Yes	-	2.92	2.96
SNS	Yes	Yes	-	5.33	0.89
CL	No	No	No	3.68	3.61
CL	No	No	Yes	3.90	3.44
CL	Yes	No	No	6.96	0.84
CL	Yes	No	Yes	6.99	0.83
CL	No	Yes	No	2.43	2.35
CL	No	Yes	Yes	2.95	2.24
CL	Yes	Yes	No	5.43	0.93
CL	Yes	Yes	Yes	5.72	0.88
NS	No	No	No	3.98	3.59
NS	No	No	Yes	4.93	1.98
NS	Yes	No	No	6.95	0.83
NS	Yes	No	Yes	6.96	0.72
NS	No	Yes	No	2.68	2.36
NS	No	Yes	Yes	5.02	1.15
NS	Yes	Yes	No	5.33	0.90
NS	Yes	Yes	Yes	5.92	0.81

Table C.1: Checkpointing Data for `SMALL-IND`

Algorithm	Main Memory	Com-pressed	Stag-gered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	4.44	4.50
SNS	Yes	No	-	7.38	2.50
SNS	No	Yes	-	3.32	3.47
SNS	Yes	Yes	-	6.55	2.23
CL	No	No	No	3.85	3.97
CL	No	No	Yes	4.08	3.84
CL	Yes	No	No	7.02	2.33
CL	Yes	No	Yes	7.01	2.13
CL	No	Yes	No	2.63	2.76
CL	No	Yes	Yes	4.09	2.92
CL	Yes	Yes	No	5.70	2.20
CL	Yes	Yes	Yes	5.32	2.19
NS	No	No	No	4.02	3.95
NS	No	No	Yes	5.13	4.33
NS	Yes	No	No	7.32	2.52
NS	Yes	No	Yes	6.89	2.36
NS	No	Yes	No	2.87	2.82
NS	No	Yes	Yes	4.75	3.86
NS	Yes	Yes	No	5.59	1.96
NS	Yes	Yes	Yes	5.47	3.33

Table C.2: Checkpointing Data for `SMALL-SYNC`

Algorithm	Main Memory	Compressed	Staggered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	3.90	3.91
SNS	Yes	No	-	7.03	0.89
SNS	No	Yes	-	3.10	3.11
SNS	Yes	Yes	-	5.30	0.91
CL	No	No	No	3.58	3.52
CL	No	No	Yes	3.82	3.40
CL	Yes	No	No	6.98	0.89
CL	Yes	No	Yes	6.93	0.85
CL	No	Yes	No	2.57	2.51
CL	No	Yes	Yes	3.37	2.68
CL	Yes	Yes	No	5.30	0.91
CL	Yes	Yes	Yes	5.73	0.91
NS	No	No	No	3.87	3.57
NS	No	No	Yes	4.86	1.86
NS	Yes	No	No	7.05	1.01
NS	Yes	No	Yes	6.74	0.76
NS	No	Yes	No	2.86	2.49
NS	No	Yes	Yes	5.25	1.39
NS	Yes	Yes	No	5.30	0.91
NS	Yes	Yes	Yes	5.91	0.84

Table C.3: Checkpointing Data for `SMALL-IND-RAND`

Algorithm	Main Memory	Compressed	Staggered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	21.30	21.32
SNS	Yes	No	-	33.21	4.21
SNS	No	Yes	-	5.52	5.54
SNS	Yes	Yes	-	8.47	4.16
CL	No	No	No	21.66	21.60
CL	No	No	Yes	24.22	22.07
CL	Yes	No	No	34.57	4.48
CL	Yes	No	Yes	33.65	4.20
CL	No	Yes	No	5.25	5.20
CL	No	Yes	Yes	9.02	5.11
CL	Yes	Yes	No	8.53	4.22
CL	Yes	Yes	Yes	12.22	4.37
NS	No	No	No	22.32	22.04
NS	No	No	Yes	27.70	11.34
NS	Yes	No	No	34.12	4.38
NS	Yes	No	Yes	34.70	3.89
NS	No	Yes	No	5.61	5.23
NS	No	Yes	Yes	23.90	5.58
NS	Yes	Yes	No	8.58	4.29
NS	Yes	Yes	Yes	25.01	5.46

Table C.4: Checkpointing Data for `MED-IND`

Algorithm	Main Memory	Com-pressed	Stag-gered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	25.24	25.27
SNS	Yes	No	-	33.48	12.38
SNS	No	Yes	-	6.26	6.31
SNS	Yes	Yes	-	8.65	4.73
CL	No	No	No	24.08	24.10
CL	No	No	Yes	25.93	25.22
CL	Yes	No	No	34.33	13.24
CL	Yes	No	Yes	35.71	14.26
CL	No	Yes	No	5.67	5.69
CL	No	Yes	Yes	9.53	9.11
CL	Yes	Yes	No	8.45	4.51
CL	Yes	Yes	Yes	11.98	7.46
NS	No	No	No	24.27	24.42
NS	No	No	Yes	25.74	24.74
NS	Yes	No	No	33.13	12.55
NS	Yes	No	Yes	34.09	15.35
NS	No	Yes	No	5.75	5.88
NS	No	Yes	Yes	22.91	22.46
NS	Yes	Yes	No	8.38	4.70
NS	Yes	Yes	Yes	22.71	22.48

Table C.5: Checkpointing Data for `MED-SYNC`

Algorithm	Main Memory	Compressed	Staggered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	24.36	24.37
SNS	Yes	No	-	33.98	4.33
SNS	No	Yes	-	28.80	28.81
SNS	Yes	Yes	-	38.92	8.84
CL	No	No	No	23.96	23.89
CL	No	No	Yes	25.27	23.14
CL	Yes	No	No	35.41	4.50
CL	Yes	No	Yes	33.69	4.22
CL	No	Yes	No	28.27	28.19
CL	No	Yes	Yes	34.89	28.35
CL	Yes	Yes	No	40.47	9.01
CL	Yes	Yes	Yes	44.02	9.18
NS	No	No	No	24.45	24.04
NS	No	No	Yes	30.06	12.17
NS	Yes	No	No	35.29	4.45
NS	Yes	No	Yes	35.34	3.98
NS	No	Yes	No	28.84	28.57
NS	No	Yes	Yes	55.49	15.89
NS	Yes	Yes	No	39.94	8.93
NS	Yes	Yes	Yes	54.59	8.36

Table C.6: Checkpointing Data for `MED-IND-RAND`

Algorithm	Main Memory	Com-pressed	Stag-gered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	39.76	39.80
SNS	Yes	No	-	43.74	36.98
SNS	No	Yes	-	8.01	8.03
SNS	Yes	Yes	-	11.46	7.60
CL	No	No	No	39.80	39.65
CL	No	No	Yes	42.02	37.93
CL	Yes	No	No	43.80	37.03
CL	Yes	No	Yes	45.68	36.44
CL	No	Yes	No	8.02	7.86
CL	No	Yes	Yes	14.97	7.74
CL	Yes	Yes	No	11.28	7.57
CL	Yes	Yes	Yes	17.70	7.68
NS	No	No	No	39.71	39.35
NS	No	No	Yes	48.82	19.07
NS	Yes	No	No	43.62	37.27
NS	Yes	No	Yes	48.55	17.83
NS	No	Yes	No	8.21	7.84
NS	No	Yes	Yes	43.17	10.16
NS	Yes	Yes	No	11.38	7.59
NS	Yes	Yes	Yes	44.26	10.10

Table C.7: Checkpointing Data for `LARGE-IND`

Algorithm	Main Memory	Compressed	Staggered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	45.52	45.61
SNS	Yes	No	-	49.12	42.25
SNS	No	Yes	-	9.87	9.90
SNS	Yes	Yes	-	11.70	7.61
CL	No	No	No	43.47	43.55
CL	No	No	Yes	46.76	46.68
CL	Yes	No	No	46.77	41.38
CL	Yes	No	Yes	50.75	43.48
CL	No	Yes	No	8.90	8.98
CL	No	Yes	Yes	16.05	16.08
CL	Yes	Yes	No	11.56	7.72
CL	Yes	Yes	Yes	18.89	14.83
NS	No	No	No	44.78	44.56
NS	No	No	Yes	46.85	46.66
NS	Yes	No	No	48.64	42.25
NS	Yes	No	Yes	46.72	45.35
NS	No	Yes	No	9.86	9.83
NS	No	Yes	Yes	42.21	41.88
NS	Yes	Yes	No	11.74	7.92
NS	Yes	Yes	Yes	41.79	41.77

Table C.8: Checkpointing Data for `LARGE-SYNC`

Algorithm	Main Memory	Com-pressed	Stag-gered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	46.76	46.77
SNS	Yes	No	-	51.22	44.22
SNS	No	Yes	-	54.02	54.19
SNS	Yes	Yes	-	58.08	48.12
CL	No	No	No	46.85	46.71
CL	No	No	Yes	50.68	46.59
CL	Yes	No	No	50.48	43.18
CL	Yes	No	Yes	52.43	41.85
CL	No	Yes	No	52.50	52.38
CL	No	Yes	Yes	63.46	52.97
CL	Yes	Yes	No	57.20	47.87
CL	Yes	Yes	Yes	68.03	46.69
NS	No	No	No	48.10	47.67
NS	No	No	Yes	53.79	22.42
NS	Yes	No	No	50.37	43.21
NS	Yes	No	Yes	54.40	21.54
NS	No	Yes	No	52.82	52.49
NS	No	Yes	Yes	101.17	28.16
NS	Yes	Yes	No	58.42	48.39
NS	Yes	Yes	Yes	100.58	25.87

Table C.9: Checkpointing Data for `LARGE-IND-RAND`

Algorithm	Main Memory	Com-pressed	Stag-gered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	2.28	2.42
SNS	Yes	No	-	4.18	0.66
SNS	No	Yes	-	1.53	1.58
SNS	Yes	Yes	-	3.57	0.81
CL	No	No	No	2.52	2.37
CL	No	No	Yes	2.47	1.75
CL	Yes	No	No	4.25	0.71
CL	Yes	No	Yes	4.27	0.78
CL	No	Yes	No	1.55	1.45
CL	No	Yes	Yes	2.18	1.33
CL	Yes	Yes	No	3.58	0.75
CL	Yes	Yes	Yes	3.64	0.74
NS	No	No	No	2.79	2.06
NS	No	No	Yes	3.46	1.12
NS	Yes	No	No	4.50	0.71
NS	Yes	No	Yes	4.03	0.32
NS	No	Yes	No	1.90	1.47
NS	No	Yes	Yes	4.16	0.73
NS	Yes	Yes	No	3.48	0.76
NS	Yes	Yes	Yes	4.48	0.61

Table C.10: Checkpointing Data for the Genetic Algorithm Tool

Algorithm	Main Memory	Compressed	Staggered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	7.27	7.40
SNS	Yes	No	-	11.08	1.47
SNS	No	Yes	-	4.20	4.33
SNS	Yes	Yes	-	7.00	1.59
CL	No	No	No	6.35	6.31
CL	No	No	Yes	7.03	5.82
CL	Yes	No	No	11.04	1.39
CL	Yes	No	Yes	11.03	1.09
CL	No	Yes	No	3.58	3.63
CL	No	Yes	Yes	4.48	3.35
CL	Yes	Yes	No	6.65	1.38
CL	Yes	Yes	Yes	7.24	1.25
NS	No	No	No	7.50	6.26
NS	No	No	Yes	7.76	5.81
NS	Yes	No	No	11.60	1.47
NS	Yes	No	Yes	10.77	1.29
NS	No	Yes	No	4.63	3.34
NS	No	Yes	Yes	6.91	5.50
NS	Yes	Yes	No	6.68	1.49
NS	Yes	Yes	Yes	7.72	3.83

Table C.11: Checkpointing Data for the Sparse Matrix Solver

Algorithm	Main Memory	Com-pressed	Stag-gered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	15.12	16.36
SNS	Yes	No	-	25.50	9.39
SNS	No	Yes	-	6.94	7.98
SNS	Yes	Yes	-	18.73	4.87
CL	No	No	No	16.18	15.75
CL	No	No	Yes	18.57	15.18
CL	Yes	No	No	25.67	10.23
CL	Yes	No	Yes	26.76	9.34
CL	No	Yes	No	7.61	7.64
CL	No	Yes	Yes	13.75	9.66
CL	Yes	Yes	No	16.64	4.10
CL	Yes	Yes	Yes	19.26	5.56
NS	No	No	No	17.66	15.74
NS	No	No	Yes	21.43	14.78
NS	Yes	No	No	25.11	8.54
NS	Yes	No	Yes	26.10	9.41
NS	No	Yes	No	8.90	7.16
NS	No	Yes	Yes	20.19	12.85
NS	Yes	Yes	No	17.21	4.00
NS	Yes	Yes	Yes	22.14	8.44

Table C.12: Checkpointing Data for Gaussian Elimintaion

Algorithm	Main Memory	Com-pressed	Stag-gered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	14.96	14.81
SNS	Yes	No	-	22.64	1.54
SNS	No	Yes	-	16.82	16.80
SNS	Yes	Yes	-	25.02	3.91
CL	No	No	No	14.72	14.35
CL	No	No	Yes	15.62	13.86
CL	Yes	No	No	22.74	1.42
CL	Yes	No	Yes	22.71	1.83
CL	No	Yes	No	16.52	16.51
CL	No	Yes	Yes	20.04	16.13
CL	Yes	Yes	No	24.57	3.73
CL	Yes	Yes	Yes	27.18	3.95
NS	No	No	No	15.12	14.39
NS	No	No	Yes	18.15	7.01
NS	Yes	No	No	23.15	1.38
NS	Yes	No	Yes	22.49	1.17
NS	No	Yes	No	16.84	16.03
NS	No	Yes	Yes	32.78	8.81
NS	Yes	Yes	No	24.81	3.80
NS	Yes	Yes	Yes	32.48	3.53

Table C.13: Checkpointing Data for Fast Fourier Transform

Algorithm	Main Memory	Com-pressed	Stag-gered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	49.63	49.77
SNS	Yes	No	-	57.31	47.31
SNS	No	Yes	-	10.70	10.77
SNS	Yes	Yes	-	18.60	10.18
CL	No	No	No	48.97	49.01
CL	No	No	Yes	52.07	51.91
CL	Yes	No	No	58.48	48.68
CL	Yes	No	Yes	62.11	49.47
CL	No	Yes	No	10.34	10.45
CL	No	Yes	Yes	18.12	17.91
CL	Yes	Yes	No	17.86	10.57
CL	Yes	Yes	Yes	26.66	18.18
NS	No	No	No	48.87	48.98
NS	No	No	Yes	51.46	51.12
NS	Yes	No	No	56.77	46.30
NS	Yes	No	Yes	52.19	49.27
NS	No	Yes	No	10.33	10.42
NS	No	Yes	Yes	45.16	44.87
NS	Yes	Yes	No	18.62	10.21
NS	Yes	Yes	Yes	44.72	44.48

Table C.14: Checkpointing Data for the Molecular Dynamics Program

Algorithm	Main Memory	Com-pressed	Stag-gered	Checkpoint Time (sec)	Checkpoint Overhead (sec)
SNS	No	No	-	70.83	69.32
SNS	Yes	No	-	77.57	57.97
SNS	No	Yes	-	19.18	17.61
SNS	Yes	Yes	-	23.82	12.74
CL	No	No	No	71.79	72.19
CL	No	No	Yes	74.16	74.32
CL	Yes	No	No	76.58	60.20
CL	Yes	No	Yes	78.88	63.73
CL	No	Yes	No	18.36	18.90
CL	No	Yes	Yes	29.54	29.84
CL	Yes	Yes	No	22.52	15.11
CL	Yes	Yes	Yes	33.41	26.05
NS	No	No	No	70.68	71.12
NS	No	No	Yes	72.70	72.85
NS	Yes	No	No	76.24	60.69
NS	Yes	No	Yes	72.88	67.89
NS	No	Yes	No	18.16	18.63
NS	No	Yes	Yes	69.94	70.30
NS	Yes	Yes	No	22.40	15.59
NS	Yes	Yes	Yes	68.28	68.99

Table C.15: Checkpointing Data for the Iterative Equation Solver