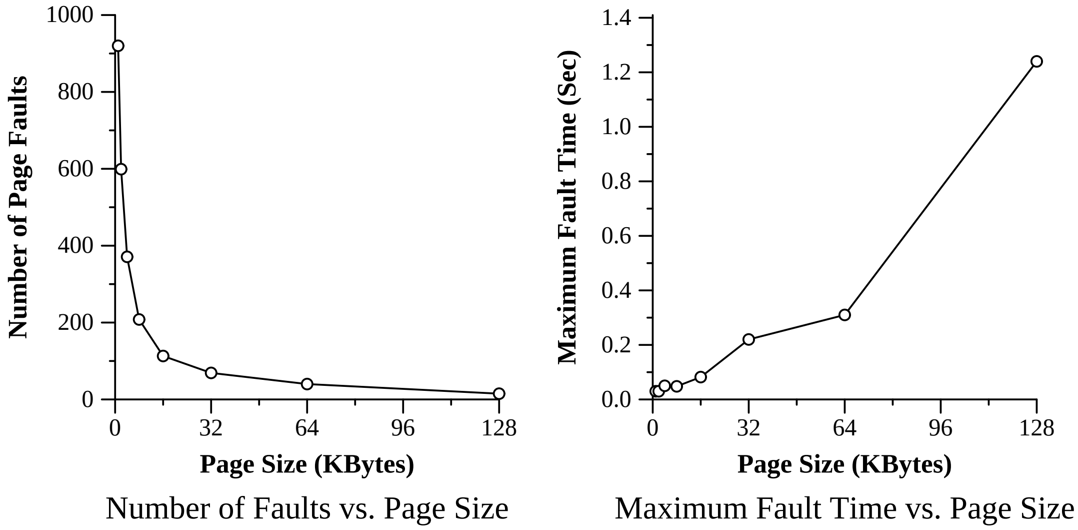

Figure 4.4: Results of Altering the Page Size.

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

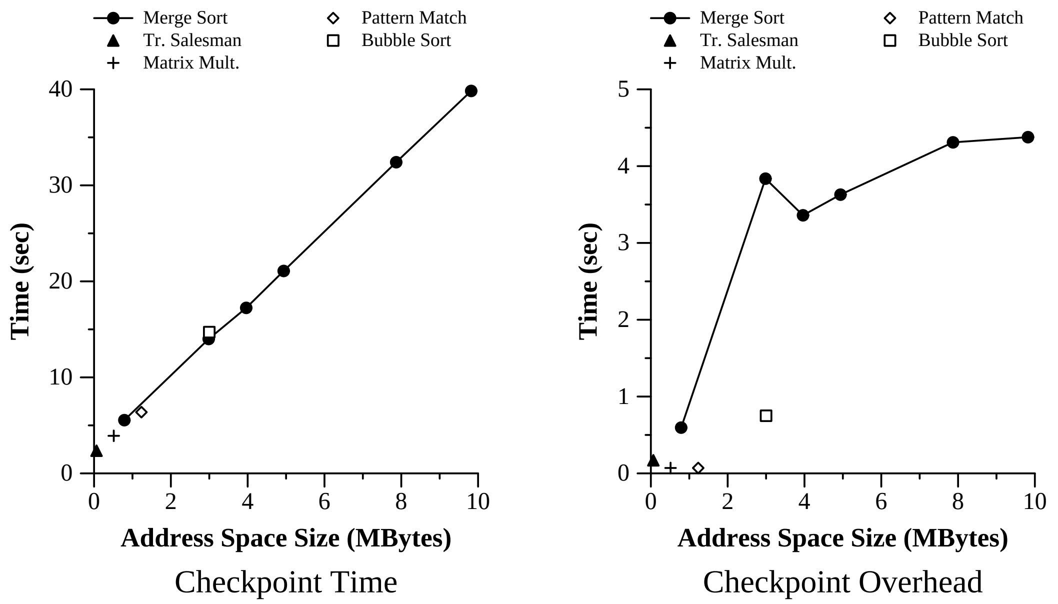

Figure 4.5: Checkpoint Time and Overhead of Other Test Programs.

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-

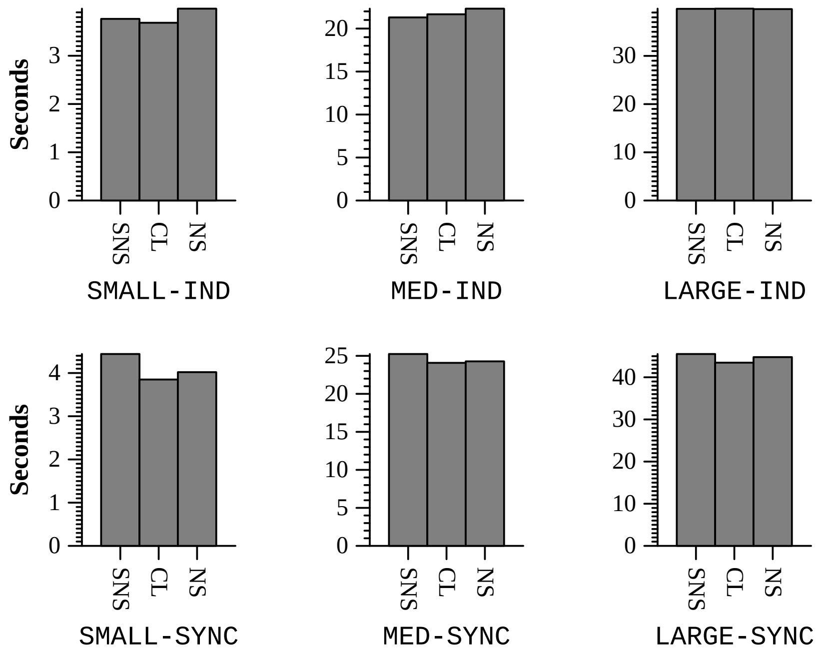

Figure 4.6: Checkpoint Times of Basic Algorithms.

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,