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.

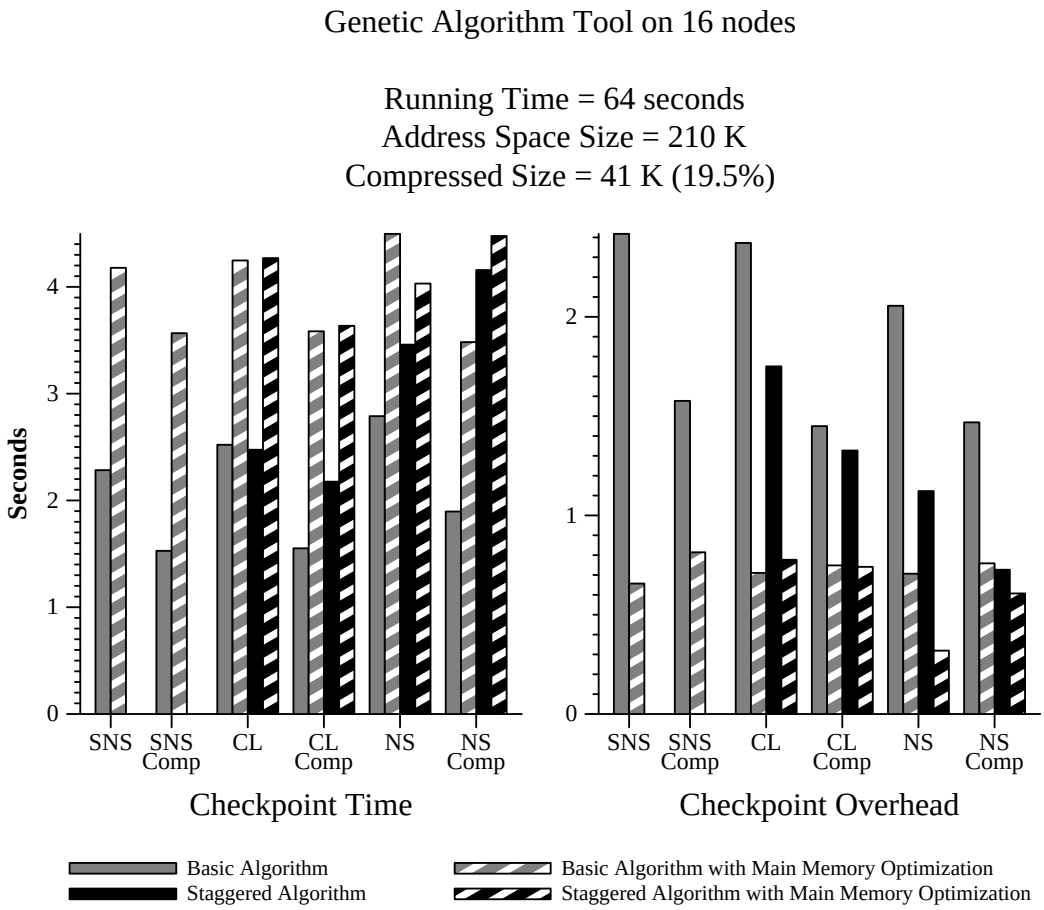

Figure 4.17: Results of ICKP on the Genetic Algorithm Tool

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

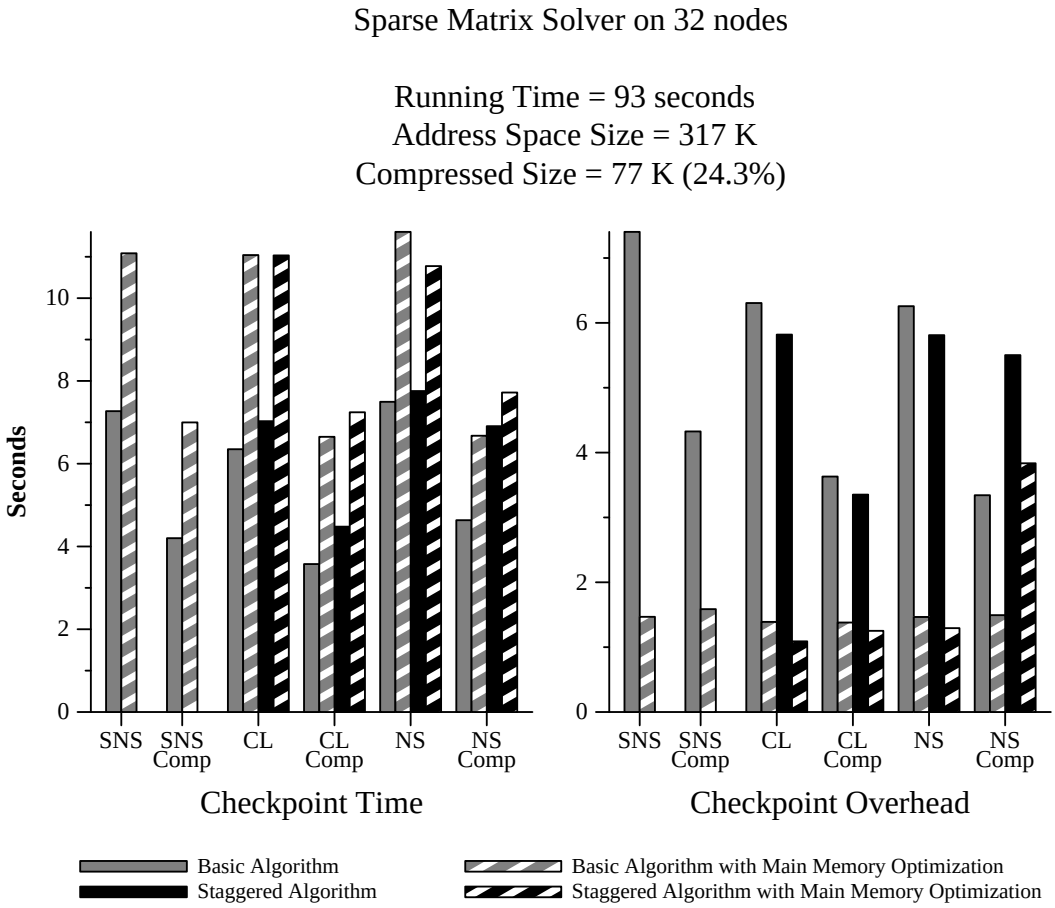

Figure 4.18: Results of ICKP on the Sparse Matrix Solver

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

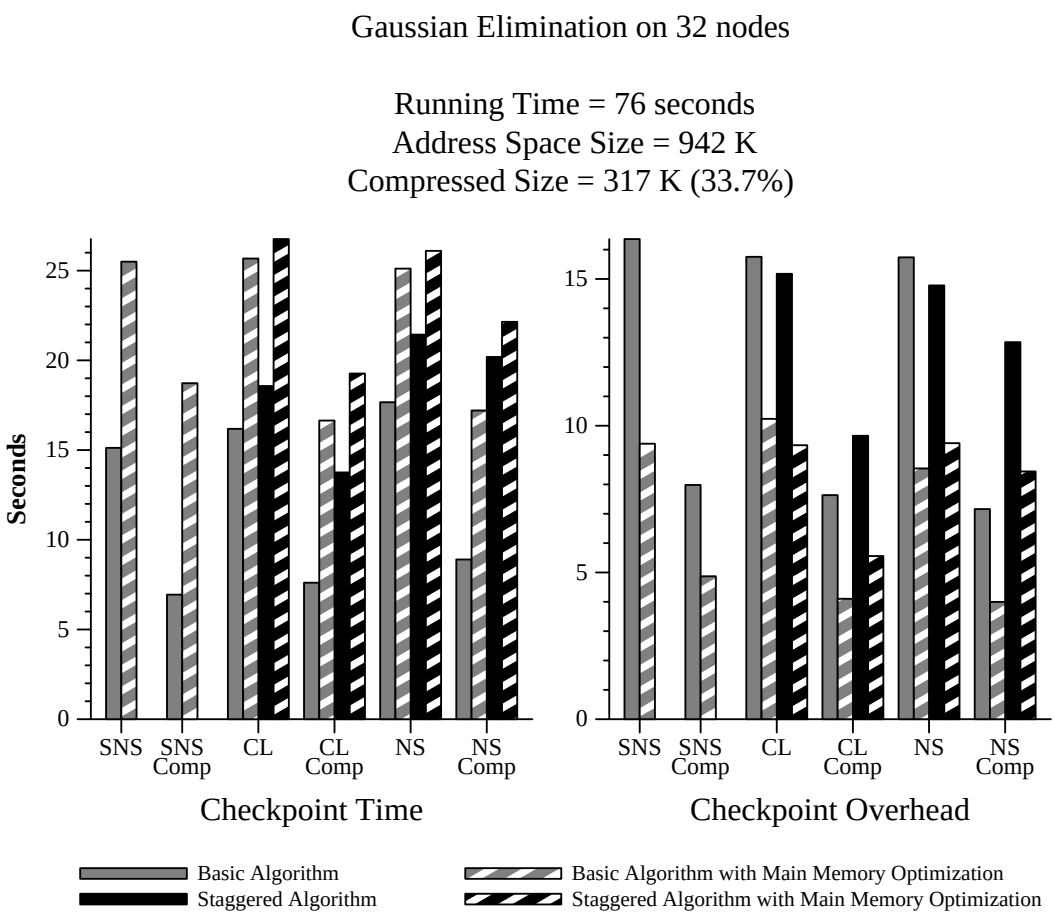

Figure 4.19: Results of ICKP on the Gaussian Elimination Program

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

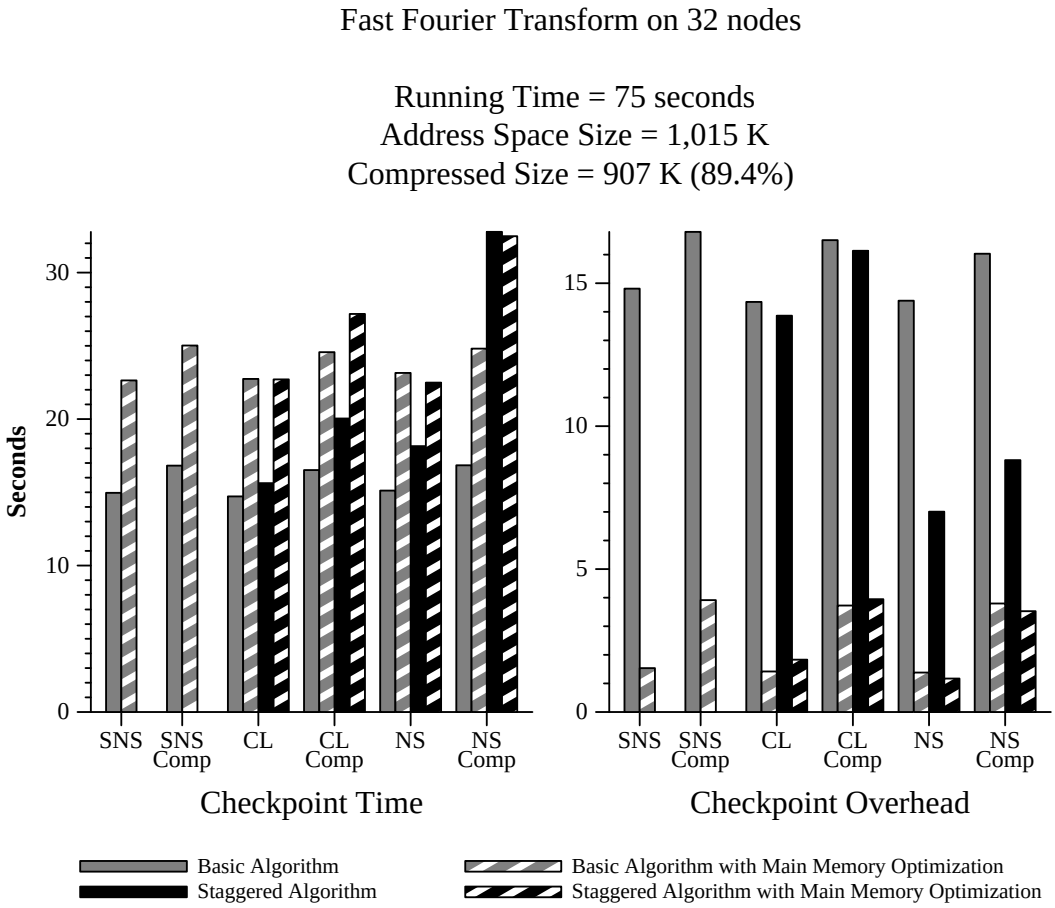

Figure 4.20: Results of ICKP on the Fast Fourier Transform

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

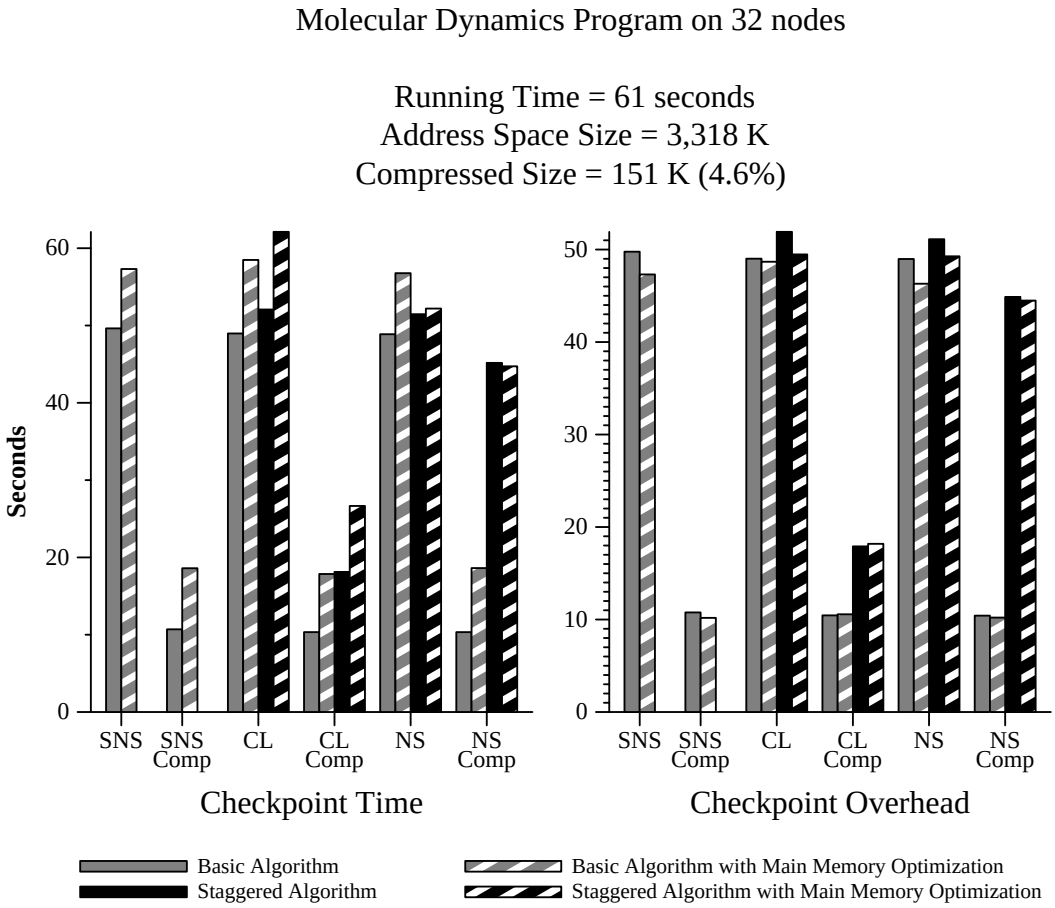

Figure 4.21: Results of ICKP on the Molecular Dynamics Program

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

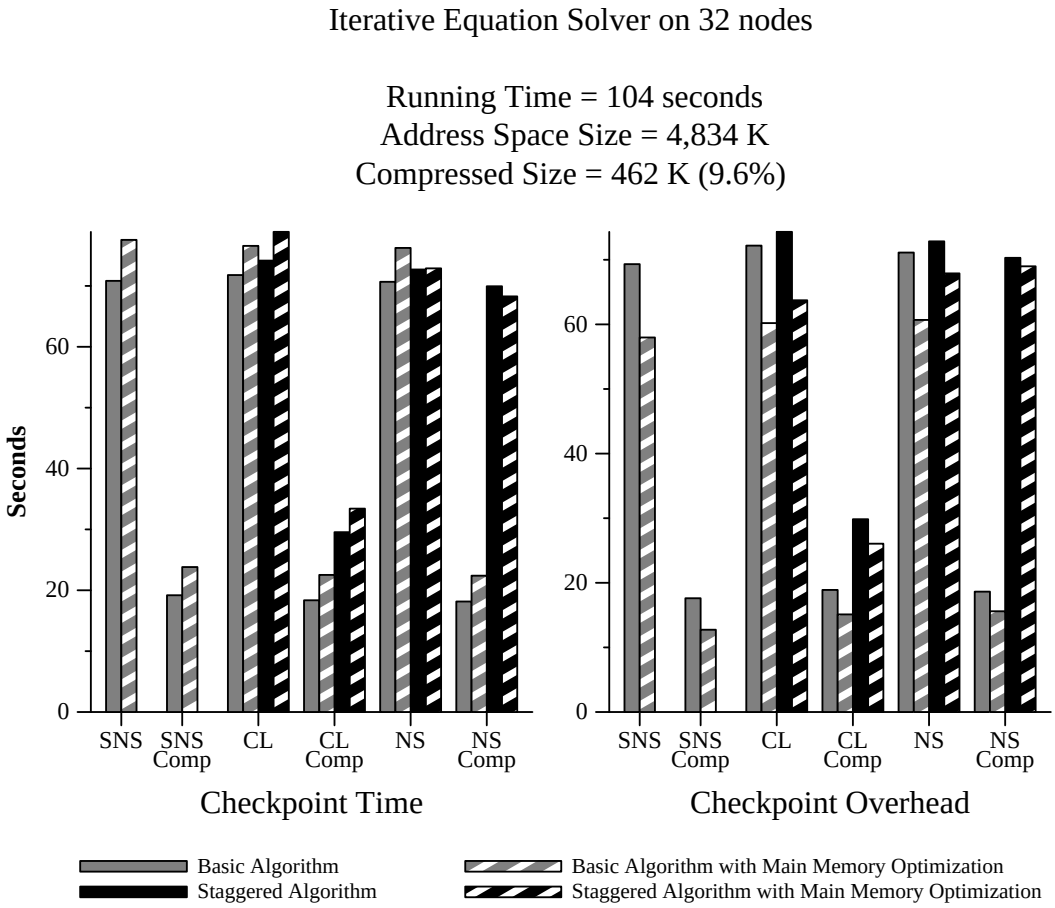

Figure 4.22: Results of ICKP on the Iterative Equation Solver

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.