CS717 : Reducing Message Log Sizes Through Iteration State Checkpointing

Rohit C. Fernandes

12/12/01

1 Introduction

Rollback recovery through checkpointing and message logging has been the primary paradigm used for obtaining fault tolerance in long running scientific applications. While checkpoints are usually taken on disk, the message logging protocol usually logs the messages on main memory to reduce the overhead associated with logging. However, there exists a number of scientific programs which communicate large amounts of data between processors. For these programs, we face the problem of the main memory getting saturated with the logs too fast.

Each time a consistent checkpoint is taken, the messages delivered before the checkpoint can be flushed. However, for the above mentioned programs the checkpoints would have to be taken very frequently. This would cause a tremendous overhead for the case of system level checkpoints. In this project, we try to measure the performance overhead of application level checkpoints taken on a per iteration basis. Optimizations include mirroring the checkpoints on the memory of other processors and exploiting redundancy in the application data to reduce the amount of checkpoint data transferred.

The rest of the report is organized as follows. Section 2 provides necessary background in terms of the fault model desired by scientific applications and the various application level checkpointing schemes. Section 3 describes the implementation of the checkpoints. Section 4 describes the experiments performed and the results obtained. Finally, section 5 concludes with the insights obtained from the experiments.

2 Background

2.1 Fault Model

The fault model assumed in this report is a simple fail-stop model. When a node fails or crashes, it simply comes to a halt. It does not behave maliciously sending arbitrary messages across the network as in the Byzantine case.

Node failures are few but they do occur and it is important that the protocol used is able to recover correctly from the failure. Multiple failures are possible though the probability of multiple nodes failing at the same time decreases rapidly with the number of concurrent failures. When a node fails, it eventually recovers. As a consequence, the number of nodes before and after failure remains the same.

Not only must the recovery mechanism be correct, but it must also be efficient. As failures occur rarely, the single most important property of the recovery protocol is low failure-free overhead. The next important characteristic is low recovery time from a single node crash. So the failure of a single node should preferably not cause a majority of the other processes to be rolled back.

2.2 Checkpointing Scheme

The checkpointing scheme used is application level i.e. at each checkpoint the live data of the application is checkpointed. The applications considered for this project follow the model of an iterative computation where the same computation is performed over a large number of iterations. The iterative computation is generally preceded by an initialization phase. Because of the application model of iterative computation and the use of application level checkpointing, recovery of the pre-failure state is done in two phases.

The first phase involves recreation of the state created by the initialization phase. There is one application level checkpoint associated with this phase.When the application executes in normal mode, some of the data that it creates is added to this checkpoint file. Some of the data is not checkpointed but it is recomputed on recovery. If a failure occurs, the application runs in recovery mode and it regenerates the data either from the checkpoint file or recomputes it. At present, this choice is made by a manually editted recovery script which is executed when the program is in recovery mode as indicated by a status variable which can be set through a commandline parameter.

The second phase involves restoring the state to the prefailure iteration. There is another application level checkpoint associated with this phase. This checkpoint is different from the one mentioned in the previous paragraph. The checkpoint is taken at the end of an iteration and it contains the iteration number of the next iteration. On recovery, the recovery script simply restores the data from the checkpoint and restores the iteration index from the application level checkpoint.

Three versions of the above scheme have been tried. The first involves storing the application checkpoint on disk. The second involves mirroring the application checkpoint on the memory of some other processor. In this version, every processor is associated with another processor which is termed its mirror. When the application level checkpoint is to be taken, a processor simply sends its data to its mirror where it is stored in a local data structure. The third version is an optimized version of the second where redundancy in the application data is exploited. If some of the checkpoint data is available locally on the mirror, then it is not sent over the network but is instead copied locally by the mirror processor.

3 Implementation

This project implemented the checkpointing schemes described above for MPI programs.

For the disk based checkpointing, a set of functions were implemented in C for saving a set of program data structures to disk as a checkpoint. Functions were also written to restore these data structures from the disk. These functions are callable from Fortran programs as well.

For mirror checkpointing, the data was sent to the mirror using MPI calls. For the optimized versions, a C function was written which made a local copy of the program data structure.

4 Experimental Evaluation

The above schemes were tried out on 2 NAS benchmark kernels - CG and SP. CG is a conjugate gradient solver. SP is a solver for three sets of scalar pentadiagonal systems of equations. Both the programs were compiled for 4 processors. Classes B and A were chosen for CG and SP respectively.

The results obtained for CG are shown in table 1. The first column titled Interval denotes the number of iterations after which successive checkpoints are taken. The next three columns stand for checkpointing to the disk, checkpoint-mirroring and optimized checkpointing. For the case of disk based checkpointing, the checkpoints are stored on the remote disk. In the case of mirroring, the MPI process identifiers are used to map processors to mirrors. In this case, processor i's mirror is taken to be the processor with identifier (i+2) mod 4. The first row represents the case when no checkpoints are taken at all.

The first observation is that for the case of CG, the overheads are not very high. This is because the size of the checkpoints that need to be saved is quite small(300KB). Also, checkpoint mirroring seems to perform better than checkpointing to disk. This is probably caused by the fact that the bandwidth available over the network is higher than that to the disk especially when all the 4 processors pound data to the same disk while in the case of mirroring, each processor only receives the data from one other processor.

The second observation that can be made is that the overhead for mirror checkpointing is negligible. So, in this case the optimized checkpointing does not seem to have much of a gain over mirroring. In the case of CG, the application data is fully replicated allowing for significant optimization to be made, but still the results are not impressive because of the small size of the checkpoints.

On the whole, however application level checkpointing seems to be a win. The total number of iterations in the program is 75. So, for instance if we checkpoint every 6 iterations, we can obtain a log-size reduction by a factor of 12 for a negligible overhead.

Interval	Disk	Mirror	Optimized
-	421.74	422.94	422.99
1	440.04	423.94	421.89
2	428.98	422.18	421.84
4	424.98	422.36	421.65
6	423.19	421.63	421.45
9	423.56	421.38	422.86
12	423.06	423.21	421.59
15	422.52.	422.07	423.30
20	422.51	422.00	421.19

Table 1: Execution time in seconds of class B of CG running on 4 processors with the different checkpointing schemes

Interval	Disk	Mirror	Optimized
-	488.92	485.87	485.11
1	1275.81	742.51	758.37
2	897.76	614.43	623.64
4	717.62	571.73	576.00
8	696.32	528.21	523.06
16	531.35	509.84	504.35
32	521.89	504.92	508.00
48	518.69	481.16	493.91
64	512.33	502.33	482.70

Table 2: Execution time in seconds of class A of SP running on 4 processors with the different checkpointing schemes

In the case of SP, the runtimes obtained are shown in table 2. The overheads obtained are much higher in this case.This is because the checkpoint sizes are much larger(about 4MB). Again as expected the mirrored checkpointing scheme seems to work better than disk based checkpointing.Also in this case, the optimized version does not do much better than the unoptimized version. This is because there does not seem to be much scope for the optimization. Only the elements at the faces of the 3 dimensional array can be locally copied so the bulk of the data needs to be communicated. The overheads are more because of the overhead involved with extracting the interior elements and transporting them. This was done by collecting all the elements in a buffer using mpi_pack and transporting them. Some of the readings in the runtime for the optimized case seem to be better than the values obtained for unoptimized checkpoint mirroring making it possible for the existence of bugs at the present time in the optimization. Also because the recovery has not been worked out it is not clear if the above optimization is correct. A difficulty associated with checkpointing parts of data structures was the fact that it was not always laid out in memory in a convenient manner requiring explicit loops to traverse the area to be extracted.

In this case it is not clear if the overheads are acceptable or not as they seem to be considerable. The total number of iterations in the program is 400. So for example if we checkpoint every 32 iterations, we get a log size reduction by a factor of 12 but at an overhead of about 4 percent. However, system level checkpoints are likely to be atleast a factor of 5 or 6 larger than the application level checkpoint because of several similar large arrays also present in the program which would also have to be checkpointed if system level checkpointing was used. This would be true even if incremental checkpointing was used as the other arrays are modified during each iteration.

5 Conclusion

In this project, 3 different schemes were tried out for application level iteration based checkpointing in order to reduce the message log sizes. The results obtained show that application level checkpoints are likely to be much smaller than system level checkpoints and one can afford to take them more frequently. The experiments also showed that checkpoint mirroring has a lower overhead than checkpointing to the remote disk.

One of the optimizations that was tried to checkpoint mirroring was trying to recognize replication and exploiting it to reduce the amount of data sent over the network. This optimization did not seem to be very helpful. In the case of CG, there was tremendous scope because of the replicated nature of the implementation but because of the small size of the checkpoint, not much improvement could be observed. In the case of SP, there was hardly any replication in the data. It is not very clear if applications in general have a lot of replication to make this optimization useful in practice.

References

[1] T. Chiueh and P. Deng. Efficient checkpoint mechanisms for massively parallel machines. In *26th Int. Symp. on FaultTolerant Comp.,* Sendai, June 1996.

[2] James S. Plank, Kai Li, Michael Puening. Diskless Checkpointing. In *IEEE Transactions on Parallel and Distributed Systems,* 1997

[3] A. Beguelin, E. Seligman and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. In *Journal Parallel and Distributed Computing,* 43(2):147-155, Jun. 1997.

[4] E. Seligman and A. Beguelin. High-level fault tolerance in distributed programs. Technical Report CMU-CS-94-223, Department of Computer Science, Carnegie Mellon University, Dec. 1994.