Mitigation of Failures in High Performance Computing via Runtime Techniques

Xiang Ni (xiangni2@illinois.edu)
Department of Computer Science
University of Illinois at Urbana-Champaign
Advisor - Laxmikant V. Kalé

High performance computing has become a powerful tool to solve complex problems in a timely manner. The number of components assembled to create a supercomputer keeps increasing in pursuit of more computational power required to enable breakthroughs in science and engineering. However, the reliability and the capacity of each individual component has not increased as fast as the increase in the total number of components. As a result, the machines fail frequently and hamper smooth execution of high performance applications. Checkpointing the application state to local storage such as memory is an effective way of dealing with fail-stop failures. However, the increase in memory capacity is much behind the increase in computing power; the severe memory pressure makes the checkpointing and execution of high performance applications even harder. With regards to these challenges, my thesis strives to answer the following questions: how can a runtime system provide fault tolerance support more efficiently with minimal application intervention? What are the effective ways to detect and correct silent data corruption? Given the limited memory resource, how do we enable the execution and checkpointing of data intensive applications?

I. Reducing Checkpoint Overhead

Traditional checkpoint/restart works like this: an application periodically saves its state and when failure happens, the application rolls back to the most recent checkpoint and restarts from there. Storing a global checkpoint to the file system is expensive. Other promising alternatives include using local storage (memory, SSD, local disk) to store the checkpoints [3, 1, 7]. However, one common fact among all these checkpointing methods is that applications stop execution until the checkpoint has been safely stored. In popular implementation of the blocking checkpoint algorithm, each node the application is running on must send its checkpoint to another node across the network [3, 7]. The network quickly gets saturated with large amount of data as the checkpoint size increases. Since a congested network transports data at a much lower rate than the peak bandwidth, the time spent in checkpointing increases tremendously in such cases. My research has helped reduce the checkpoint overhead while maintaining the application execution rate. I have also created a method to let runtime system guide applications to checkpoint at the best times.

A. Semi-blocking checkpoint algorithm

In [5], we have described a semi-blocking checkpoint algorithm to reduce the checkpoint overhead caused by the network congestion. Semi-blocking checkpointing algorithm tries to hide the checkpoint overhead by interleaving the transmission of the checkpoint to the buddy node with application execution. Once the checkpoint is safely stored in local memory, the application resumes execution while transmitting the checkpoint to the buddy node in the background. The transmission of the checkpoint may interleave with application communication, which could delay the application progress. To solve this problem, in our implementation, we first decompose the checkpoint transmission into several small messages. Next we leverage the knowledge of the runtime system to only send out the checkpoint messages when there are no application messages waiting to be sent. We have shown that

semi-blocking checkpointing algorithm can help reduce checkpoint overhead by 5 times using two applications, Wave2D and ChaNGa.

B. Automatic checkpoint decision

Using optimal checkpoint interval is important to reduce the overall fault tolerance overhead [2]. A fixed checkpointing interval, which is often used in existing checkpointing libraries, fails to adapt to the dynamically changing failure behavior and results in extra overhead due to fault tolerance support. Moreover, as online failure prediction becomes more accurate, checkpointing right before the failure occurs can help increase the mean time between the failures visible to applications.

Our runtime level checkpointing mechanism [4] can adapt to the changing of failure rate and schedule optimal checkpoints based on observations. To evaluate it, we performed a 30 minutes run of Jacobi3D benchmark on 512 cores of Blue Gene/P with 19 failures injected during the run. The failures are injected according to Weibull process with a decreasing failure rate. We found that the runtime system schedules more checkpoints at the beginning when the failure rate is high. Towards the end since the failure rate is lower, the checkpoint interval decided by the runtime system is longer.

C. Ongoing work

We plan to empirically study the network congestion caused by the coordinated checkpointing with a network simulation tool. We are interested in measuring the severity of network congestion with different network topologies and placements of buddy nodes. Next we will work on assessing how semi-blocking checkpointing algorithm can help reduce the network congestion and the optimal strategy to interleave checkpointing with application communication in order to reduce the interference with applications.

II. Detection and Correction of Silent Data Corruptions

Silent data corruption (SDC) refers to the unintended changes in the computer data. Unlike hard error, such as fail-stop type of failures, silent data corruptions may not cause the application to crash but changes the application output silently.

A. Replication enhanced checkpointing

To effectively detect and correct both SDCs and hard errors, we proposed and implemented ACR [4]: a replication enhanced automatic checkpoint/restart framework. We explored different resilience schemes in ACR. With various optimizations, such as checksums and topology aware mapping, ACR scales very well to 131,072 cores with minor overhead.

ACR uses checkpointing and replication to detect SDC and enable fast recovery of applications from SDCs and hard errors. When a user submits a job using ACR, a few nodes are marked as spare nodes and are not used by the application, but only replace failed nodes when hard errors occur. The rest of the nodes are equally divided into two partitions that execute the same program and checkpoint at the same time. We refer to these two partitions as replica 1 and replica 2. Each node in replica 1 is paired with exactly one unique node in replica 2; we refer to these pairs as *buddies*. Logically, checkpointing is performed at two levels: local and remote. When a checkpoint is invoked by the runtime, each node generates a local checkpoint and saves it in local memory. Local checkpoint of a node in one replica serves as remote checkpoint of the buddy node in another replica.

In order to detect SDC, every node in the replica 1 sends a copy of the local checkpoint to its buddy node in replica 2. Upon receiving the remote checkpoints from their buddy nodes in replica 1, every node in replica 2 compares the remote checkpoint with its local checkpoint using the same serialization framework used to pack the checkpoint. If a mismatch is found between the two checkpoints, ACR rolls back both the replicas to the previous safely stored local checkpoint, and then resumes the application.

Based on the reliability and performance requirements, ACR is able to recover from hard failures in different ways. The most reliable strategy is called strong resilience scheme which guarantees 100%

protection from SDC. In this scheme, the crashed replica will roll back to the most recent checkpoint when failure happens. To achieve that, the crashed node needs to get the previous checkpoint from its buddy node. The amount of rework in this scheme is large and thus it may slow down the application progress. In comparison the weak resilience scheme can help reduce the application execution time at the cost of incomplete SDC protection. Using weak resilience scheme, the crashed replica will wait until the other replica reaches the next checkpoint and recover using the latest checkpoint. Thus the crashed replica does not need to perform any rework. However, the system is left unprotected from silent data corruptions for the entire checkpoint interval.

B. Ongoing work

Replication-based scheme provides a baseline scheme to detects all the silent data corruption at the cost of doubling the resource usage. Currently we are exploring cheaper methods to detect SDC automatically based on the attributes of scientific data.

Many scientific simulations calculate values on points in a discretized space based on iterative numerical integration methods. If a computer program simulates a phenomena in nature, the rules of nature should also be preserved in the simulated date unless SDC occurs or there are bugs in the computer program. The simulated data embodies the attribute of spatial smoothness, which means if two points are close in space, the difference between the values at the two points should also be small. We examine this attribute with a real application *OpenAtom* [6] and a benchmark *Jacobi2D* that performs a 5-point stencil based computation on a two dimensional structured mesh. We find that the data value changes gradually from one point to another and obeys the attribute spatial smoothness in both applications.

We plan to utilize such attributes of scientific data to detect any anomaly caused by SDCs. Afterwards we will look for methods to correct such errors approximately utilizing the spatial smoothness of data without re-execution.

Although the proposed approach may help protect scientific data from silent data corruptions with low cost, it has limitations to protect all other data except scientific data. We refer to the other data as *control data*. Even though the amount of control data is relatively small in a typical simulation based program, one bit flip on the control data may cause more severe consequences. For example, if the iteration count is bit flipped, the program may directly jump to the last iteration and skip all the simulation in the middle. We plan to enhance the proposed approach with selective replication to protect control data from silent data corruption as well.

III. Relieving Memory Constraints

The increase in memory capacity is much behind the increase in computing power; the severe memory pressure makes the checkpointing and execution of high performance applications even harder. The advent of non-volatile memory could help relieve the memory pressure. However, compared to DRAM, the bandwidth of non-volatile memory is still 10 times less. If non-volatile memory is used as storage for application checkpoints, how do we reduce the overhead to write/read checkpoints? If data intensive applications utilize non-volatile memory as a secondary memory partition, what a runtime system can do to make the data access easier and faster? These are some of the questions that my thesis attempts to answer.

A. Checkpointing to SSD

Double in-memory checkpointing [7] introduced the idea of diskless checkpointing to reduce the checkpoint overhead. This approach has achieved good performance since it does not cause any I/O congestion. However, it is not feasible for data intensive applications. For such applications, it is difficult to have enough memory for application checkpointing. To relieve the memory pressure of checkpointing while maintaining high checkpointing speed, we proposed two strategies to store checkpoints in SSD [5].

Full SSD Strategy In this strategy all the local and remote checkpoints will be saved to SSD which can fully relieve the memory pressure.

Half SSD Strategy We reduce the writes to SSD by only storing the buddy's remote checkpoints in SSD. At restart, only checkpoints of the crashed node need to read from SSD while other nodes can recover from the checkpoints in memory concurrently.

Instead of blocking the application execution to write checkpoints to SSD, in our implementation a dedicated IO thread is used on each node for asynchronous access to SSD. Our experiments have shown that asynchronous IO access could help hide checkpoint overhead. Half SSD strategy reduces the checkpoint overhead by half in many cases.

B. Ongoing work

For data intensive applications that are hard to fit in memory, we plan to explore the use of non-volatile memory as a secondary memory partition with runtime system guidance. The goal is to build a framework that users can transparently allocate data on non-volatile memory. The runtime system transports the data from non-volatile memory to DRAM before it is being used. By learning application access patterns, the runtime system only offloads the less used data to the secondary memory partition to reduce the amount of communication between the two levels memory system. We also plan to explore the use of asynchronous I/O access to help us hide the data transportation time with application execution.

IV. Conclusion

To summarize, my thesis research focuses on developing efficient runtime level fault tolerance schemes to protect applications from both soft and hard errors transparently. Presenting my current work to the community will help connect me with the application community to improve the usability of the fault tolerance schemes. At the same time, it presents a bigger picture based on some of my useful work to the rest of the community.

References

[1] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka. FTI: High performance fault tolerance interface for hybrid systems. In *Supercomputing*, pages 1 –12, Nov. 2011.

[2] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Comp. Syst.*, 22(3):303–312, 2006.

[3] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC*, pages 1–11, 2010.

[4] X. Ni, E. Meneses, N. Jain, and L. V. Kale. Acr: Automatic checkpoint/restart for soft and hard error protection. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13. IEEE Computer Society, Nov. 2013.

[5] X. Ni, E. Meneses, and L. V. Kalé. Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm. In *IEEE Cluster 12*, Beijing, China, September 2012.

[6] R. V. Vadali, Y. Shi, S. Kumar, L. V. Kale, M. E. Tuckerman, and G. J. Martyna. Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers. *Journal of Comptational Chemistry*, 25(16):2006–2022, Oct. 2004.

[7] G. Zheng, L. Shi, and L. V. Kalé. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *2004 IEEE Cluster*, pages 93–103, San Diego, CA, September 2004.