A Scalable Double In-memory Checkpoint and Restart Scheme towards Exascale

Gengbin Zheng, Xiang Ni, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
E-mail: {gzheng, xiangni2, kale}@illinois.edu

***Abstract*—**
As the size of supercomputers increases, the probability of system failure grows substantially, posing an increasingly significant challenge for scalability. It is important to provide resilience for long running applications. Checkpoint-based fault tolerance methods are effective approaches at dealing with faults. With these methods, the state of the entire parallel application is checkpointed to reliable storage. When a failure occurs, the application is restarted from a recent checkpoint.

In previous work, we have demonstrated an efficient double in-memory checkpoint and restart fault tolerance scheme, which leverages Charm++'s parallel objects for checkpointing. In this paper, we further optimize the scheme by eliminating several bottlenecks caused by serialized communication. We extend the in-memory checkpointing scheme to work on MPI communication layer, and demonstrate the performance on very large scale supercomputers. For example, when running a million atom molecular dynamics simulation on up to 64K cores of a BlueGene/P machine, the checkpoint time was in milliseconds. The restart times were measured to be less than 0.15 seconds on 64K cores.

I. Introduction

One concern of high performance computing for exascale is the ability to tolerate faults. Even though today's supercomputers are composed of reliable parts, the *mean time between failure* (MTBF) still drops down as the number of processors increases. Jaguar, the 3rd ranked supercomputer on top 500 today, had 2.33 average failures per day during the period from August 2008 to February 2010. For today's scientific simulation running for many days, such failures are catastrophic. One common solution to support fault tolerance are the disk-based checkpoint/restart schemes. In these methods, applications or HPC system checkpoint the state of the entire parallel application needed for restart to a reliable medium, typically a NFS fault free space. However, periodical checkpointing to such slow storage can be expensive, for example, it could consume up to 20% of the application time [1].

There are two main methods to start checkpointing: system or application initiated checkpoint. System based checkpoint may take 20 to 40 minutes to checkpointing for the best machines on the top 500 list (2008) [2], [3]. Therefore, system level checkpointing to the file system is clearly impractical at exascale. Application-level checkpointing can help reducing checkpoint size, but is still challenging in terms of the scalability at exascale, and places additional burden on application programmer. A runtime-level checkpointing scheme reduces the burden on the programmer, especially by automating the protocol for triggering checkpoints, and carrying out a recovery. In our previous work [4], [5], we explored an in-memory checkpoint/restart scheme in a fault tolerant CHARM++ [6] and Adaptive MPI [7] runtime systems. The protocol does not rely on any reliable storage for checkpoints. The restart protocol allows application to continue to run on the surviving processors after a crash without a full stop. The protocol uses local memory or disk for checkpoints, and can leverage the high speed communication network to speed up the checkpointing process.

The basic idea is to periodically create two checkpoints for each pieces of application data encapsulated in CHARM++ objects. One checkpoint is stored in local storage (memory or local disk), and the other is stored on a different node, called a buddy node. On failure, the surviving nodes restores their object data from the local checkpoints, whereas the objects on a chosen spare node (to replace the failed one) are restored using checkpoints stored at a buddy node. This method is capable of tolerating all single failures, and most multiple failures, if failed nodes are not buddy to each other. Although this scheme does not provide an infallible method of fault tolerance, it should be sufficient to apply to very large machines as single failures are the most common failure in today's HPC system. For example, 95% of the failures on TSUBAME are single failure.

The two checkpoints constitute a memory overhead for this scheme, but (somewhat surprisingly) this is tolerable for a large class of applications: those that have a smaller memory footprint at checkpoint. These include molecular dynamics, N-body codes, certain quantum chemistry (nanomaterials codes), etc. For others, the scheme relies on future local storage (or of course, a global file system as a fall back).

In this paper, we study the techniques required in the double in-memory checkpoint and restart scheme for very large scale. We examine the overheads of the implementation that could potentially limit scalability, and further improve the implementation by optimizing communication (especially the collective communication).

One obstacle for demonstrating fault tolerance on MPI applications is that the queueing system on supercomputers kills the entire job when a process fails. To work around this limitation, we developed a fault injection scheme that mimics a failure of a process without actually killing it. This allows us to demonstrate the fault tolerance scheme with MPI on very large scale supercomputers.

The rest of this paper is organized as follows. Section II describes the double in-memory checkpoint/restart protocol as background. The technique we use to demonstrate it for MPI applications and further optimization of the checkpoint/restart scheme to exascale is presented in Section III. Performance results of the optimized fault tolerance scheme on up to 64K cores are provided in Section IV. We discuss some related work in Section V, and finally, Section VI concludes the paper.

II. Background

In this section, we summarize a design of a scalable in-memory checkpoint-based fault tolerance scheme targeting very large scale parallel applications. The supporting parallel runtime systems are Charm++, a message driven runtime system, and Adaptive MPI [7], an implementation of MPI on top of Charm++. These fault tolerant runtimes take advantage of the migratability objects and threads.

A. Runtime Support for Checkpoint/Restart

The fault tolerant runtime system supports checkpointing of application data in two levels: fully automated checkpointing or flexible user-controlled checkpointing by additional helper functions.

Adaptive MPI [8] runs MPI "processes" in light-weight threads, which are easier for checkpoint and restart to handle compared to processes. Thread migration during restart would raise the problem of pointer reference. *Isomalloc* [8], [9] is used to solve this problem for fully automated checkpointing, similar to the technique in the PM^2 system [10]. Isomalloc reserves a range of virtual address space for all the processors. During checkpointing, virtual addresses of the MPI threads or objects and the data associated with them are saved automatically. A object or thread can then be restored on any processor since the allocated data can be restored without changing its address.

Another option is that users can write their own helper functions to pack and unpack heap data for checkpointing and restoring an object. This is sometimes useful in reducing the size of data involved in checkpointing and restoring. Application developers could use application specific knowledge to pack only the live variables at the time of checkpointing, or use a compiler to automate this [11]. This method reduces the data amount to checkpoint, and so checkpointing becomes faster.

B. Basic Double In-Memory Checkpoint/Restart Scheme

Checkpoint/restart scheme requires nodes to frequently save their complete state to stable storage or the memory of another node. The optimum time between checkpoints has been analyzed elsewhere [12], [13].

The in-memory checkpointing scheme [4] introduced the idea of diskless checkpointing that checkpoints data in memory. It uses a coordinated checkpoint strategy, which requires

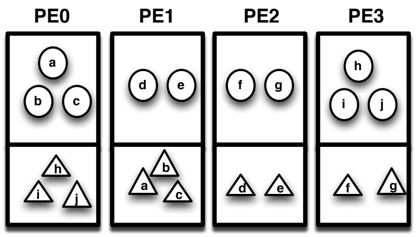

Fig. 1: In-Memory Single Checkpoint

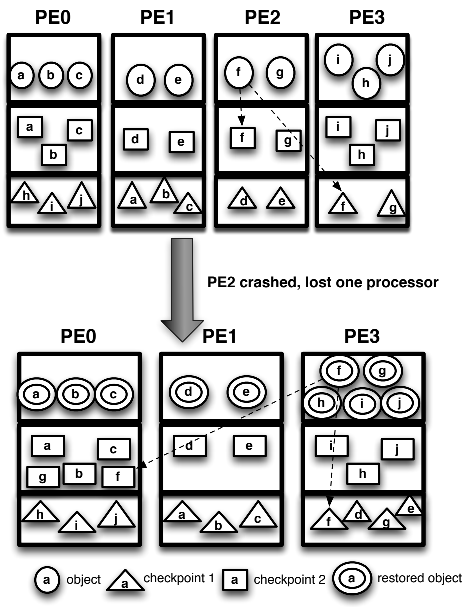

Fig. 2: In-Memory Single Checkpoint

applications to have a synchronization point where they could start a global collective operation to checkpoint. In order to handle one failure at a time, a common case scenario, one checkpoint of the application state in the memory of a different processor is not sufficient as illustrated in Figure 1. In this scenario with 4 processors, each Charm++ object (represented as a circle) checkpoints only one copy of its checkpoint (represented as a triangle). When processor 2 crashes, the checkpoints for object d and e in memory of that processor are permanently lost, so we couldn't recover from the checkpoint. This suggests that at least two copies of the checkpoint at different locations are needed. In particular, we adopted an in-memory *double-checkpointing* scheme which can tolerate at least one failure at a time.

Figure 2 illustrates an example of this scheme. The top half of the figure shows the scenario before one processor crashes. Each circle represents an object being checkpointed, while each triangle and square represents its first and second checkpoints. We call these two processors *buddy processors* for the checkpointing object. Note that one of the two *buddy processors* can be the same processor where the object resides. This can help reduce the checkpointing overhead, since the checkpointing is basically a local memory copy, which is much faster than accessing memory of remote processors. Overall, compared to the traditional on-disk checkpointing, in-memory checkpointing scheme uses memory as storage in a distributed way, taking advantage of the high speed interconnect, which tends to be more efficient.

The restart procedure is initiated by a crash of a physical processor. On clusters, the crash detector in the runtime system detects the crash through broken pipe socket errors. When the restart procedure is initiated, all surviving processors examine the checkpoints in their memory and check for missing buddy processors. A new processor is chosen (which can be either a spare processor, or a running processor) to replace the crashed processor and the latest checkpoint data is copied to that processor to maintain the double checkpoints. One of the two buddy processors is then responsible for restoring the corresponding objects from its checkpoints in memory. At restart, if the replacement processor is from a running processor, then load imbalance may occur since that processor restores more checkpoints. This can be fixed by a load balancing phase after restart [4].

The bottom half of the Figure 2 illustrates a snapshot of the objects and their checkpoints distributed on processors after a recovery is complete. The lost checkpoints on the crashed processor 2 are recovered to processor 3 and processor 0 respectively. Processor 3 is chosen to restore processor 2's objects(f,g) locally to avoid communication overhead, since Processor 3 is processor 2's original *buddy* processor.

Our protocol ensures the recovery from a single node failure and we can recover from multiple concurrent failures if the crashed processors are not buddies to each other.

III. Optimizations for Scalability

In previous work [4], we have shown that the basic scheme achieved good performance on clusters of more than a hundred cores [4]. However, when we test this scheme on a Blue Gene/P machine with tens of thousands of cores, we found both the checkpointing and restarting time increases almost linearly as the number of processors increases, as shown in Figure 3 and 6, which is not scalable. Although the absolute performance is still surprisingly good (for example, a restart on a 64K processors only took less than 4 seconds), for exascale systems, it might still be a problem when there are, for example, millions of cores. This motivates the work in this paper to further optimize the double in-memory checkpoint/restart scheme for exascale machines.

A. Optimization Techniques

Stale message handling: Since the double in-memory scheme restarts application on the fly without actually restarting the parallel job, at the beginning of the restart phase, when the application is restored to recent checkpoints, old messages that are sent before checkpoints are restored are still in transmission or buffered, and possibly mixed with system messages due to the ongoing parallel restart. Even more complicated, it is possible for the stale messages to be eventually delivered on a node after checkpoints are restored. Therefore, it is critical to discard all stale messages from the crash. To differentiate the stale messages with the legal messages sent after crash, an *epoch* number is used in our double in-memory checkpoint scheme. However, fault detection and fault announcement to all nodes occurs in parallel with the continuous execution of the surviving nodes. When computation only depends on neighboring cores, the computation on those cores may not be affected and keep going even after a node is crashed until the fault notification message finally arrives. During this period of time, more "stale" messages may be generated from those cores, which increases the burden of discarding these stale messages. On very large scale machines, dealing with stale messages system wide may be expensive, therefore, it is critical to throttle the execution of the program as soon as possible after a crash is detected to prevent propagation of the stale messages. In the optimized scheme, we introduce a new phase in restart which is dedicated to throttling the stale messages. This phase starts immediately after a fault is detected. All nodes enter a state in which they discard all received messages until a *quiescence* is detected, which means there are no stale messages. The quiescence detection is an efficient tree-based algorithm that has the complexity of $O(logP)$, where P is the number of nodes.

Optimizing small messages using streaming techniques: During restart, many small bookkeeping messages are sent to update the runtime system about the migratable objects. For example, for each object restored, a small message containing its new location is sent to its *home* processor. On very large systems, these extremely large number of small messages become a problem - it causes traffic jam and greatly slows down the restart process. In the optimized version, we applied a streaming optimization technique that combines small messages sent to the same destination processors into one bigger message.

Optimizing collective communication: In the original scheme, several barriers are needed during checkpoint and restart to ensure the order of checkpoint/restart phases, for example, the recovery of the array elements from checkpoints happens only after removing all the stale object data in memory of all processors. These barriers create a scalability challenge. Although, efficient spanning tree-based reduction implementation exist in CHARM++, it can not be easily used in this case. This is because the implementation can not handle the case when there is a crashed processor in the tree and the inconsistent reduction sequence number caused by the failure. Therefore, the original scheme implemented a simple fault tolerant barrier based on point-to-point communication synchronized by a central processor. This simple implementation is sufficient for machines with hundreds of nodes, however, clearly, it does not scale. To optimize the fault tolerant barrier, we re-implemented it based on the reconstructed spanning tree during restart. The new barrier also ignores the reduction sequence number, which is only needed when there are multiple concurrent reductions. During checkpoint/restart, we only use barrier to separate different

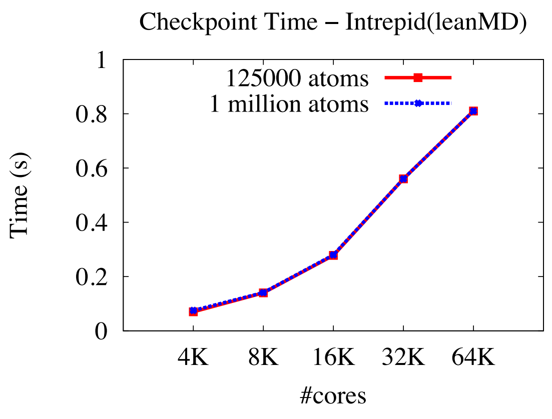

Fig. 3: LeanMD Checkpoint time before optimization (time in seconds)

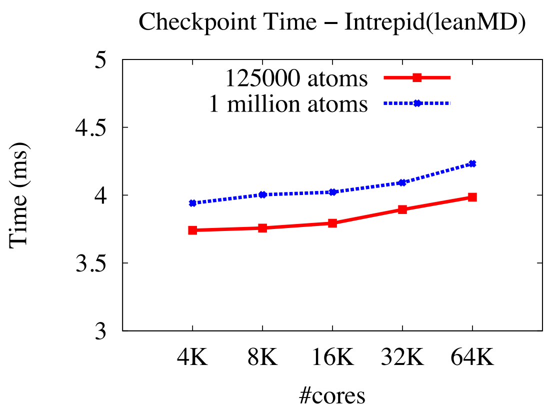

Fig. 4: LeanMD Checkpoint time after Optimization (Note: time in milliseconds)

phases, there is no multiple concurrent reductions occurring. We found the new optimized barrier significantly improves the checkpoint/restart performance. For example, on 64K cores, the restart time is dramatically reduced from 3.6s to 0.15s for an 1-million atom system. The experiment result is detailed in Section IV.

B. Fault Tolerance for MPI Application on Supercomputers

One obstacle for demonstrating fault tolerance on MPI applications is that the job schedulers on supercomputers kill the entire job when a process fails. To allow any of the fault tolerance schemes to work with the job queueing systems, it would require modification to the job scheduler to let a fault tolerant job recover itself. However, such change to the job scheduler is not feasible on today's supercomputers. Instead, we developed a scheme that mimics a failure of a process without actually killing it.

This fault injection scheme is implemented as a *DieNow()* function, which is inserted by the user at any place in their program to trigger a failure, typically controlled by a random number generator. When the DieNow() function is called by a process, the process will be forced to hang and stop responding to any communication as if it has died. The fault detection scheme is implemented as a keep-alive protocol. Each MPI process pings its *buddy* process periodically to inform its buddy that it is still alive. If there is no response from a process for a certain period of time, the *buddy* process will diagnose that the process is dead, and announce the dead process's rank to all other processes. All processes collectively respond to the "failure" and execute the restart protocol. A pool of some spare MPI processes are created at the job startup time to be used for restart. These processes do not initially run user application code. During restart, one spare MPI process will be chosen to replace the "failed" MPI rank to execute the user application.

Compared to the real scenario, the only difference in our emulated fault injection scheme is that the dead process does not really go away. However, the rest of the fault tolerance protocol is the same as it would happen in real scenario. When a job scheduler is extended to allow a failed process, our scheme can immediately take advantage of it to provide true fault tolerance support to MPI applications.

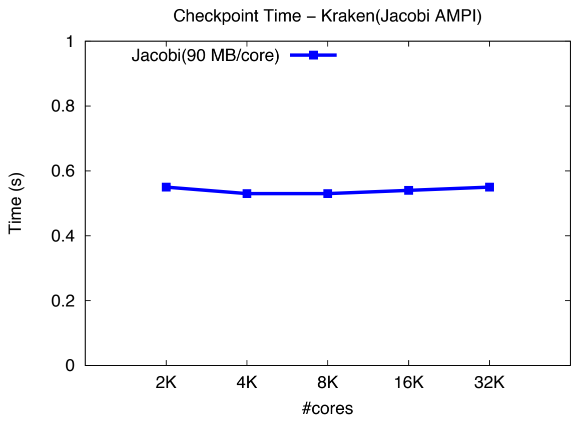

Fig. 5: Checkpoint time for Jacobi (on Kraken)

IV. Experiments

We evaluate the overhead of periodic checkpointing as well as the performance of restarting applications after a failure.

Two applications are used in our experiments. One is LeanMD, a molecular dynamics simulation program written in CHARM++. As typical molecular dynamic simulation programs, this application has a medium memory footprint. The other benchmark is Jacobi, a 7 point stencil MPI program which uses 3D decomposition. It has a larger memory footprint.

The experiments are done on Intrepid at Argonne National

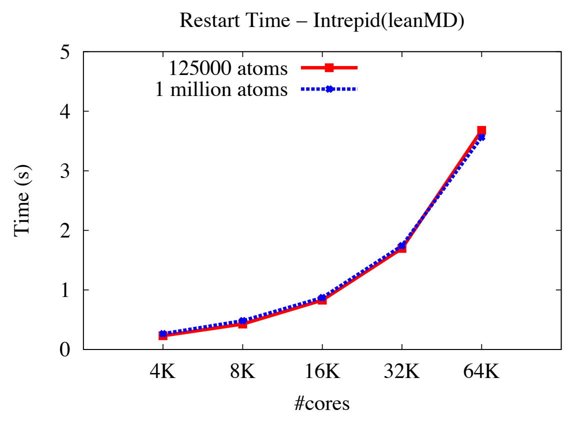

Fig. 6: Restart time-LeanMD (Before Optimization)

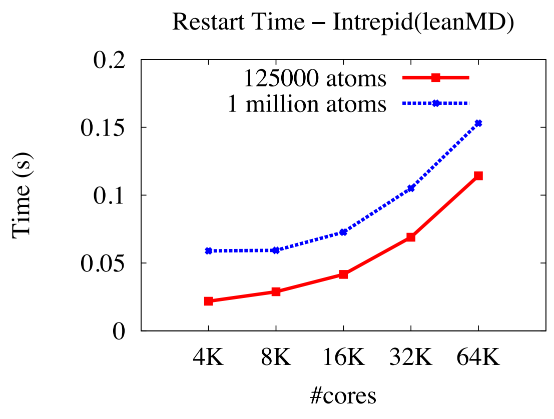

Fig. 7: Restart time-LeanMD (After Optimization)

Laboratory, and Kraken at the National Institute for Computational Science. Intrepid consists of $163,840$ cores, 80 terabytes of RAM, with a peak performance of 557 teraflops. Kraken has $9,408$ compute nodes and each node contains 12 cores, 16GB of memory, with a peak performance of 1.17 petaflops.

A. Checkpoint Time

Figure 3 shows the checkpoint time of LeanMD before optimization, on from 4K to 64K cores of Intrepid. Two different size molecular systems are uses in the experiments, one with $125,000$ atoms, and the other with 1 million atoms. As can be seen in the figure, although the checkpoint overhead is relatively small because of our in-memory scheme, it increases linearly as the the number of cores increases. This was largely due to the necessity of using an inefficient collectives protocol during recovery.

In comparison, Figure 4 illustrates the time to checkpoint LeanMD after applying the optimizations described in Section II. Instead of using a point-to-point implementation of a fault-aware barrier, using the spanning tree-based barrier reduces the time to checkpoint. The affect of the optimization can be observed in Figure 4. We can see that the checkpoint time remains almost flat when number of core increases from 4K to 64K for both molecular systems. In particular, the checkpoint time on 64K cores for the 1 million molecular system is only about 4.2 ms. In comparison, the simulation in LeanMD would take tens of milliseconds per time-step.

The second benchmark is a Jacobi program written in MPI, which has a larger memory print for checkpoint. In this experiment, we keep the checkpoint size fixed as 90 megabytes per core, and test the checkpoint time on Kraken supercomputer, with varying number of cores from 2048 to $32,768$. Note that as the number of cores doubles, the total amount of checkpoint across the entire system also doubles. This Jacobi MPI program runs on our fault tolerant MPI runtime, AMPI, and uses isomalloc for fully automated checkpointing. The result is illustrated in Figure 5. As we can see that although the total checkpoint size doubles every time when the number of cores doubles, the checkpoint time remains constant in these tests, and is less than 0.6 second. This demonstrates that our in-memory checkpoint scheme is highly efficient, and can potentially scale to even larger systems.

B. Recovery Performance

We use LeanMD to evaluate the restart performance, with the same two different size molecular systems. Restart is performed on Kraken, using the fault injection techniques described in Section III-B. The recovery time is measured from the time a failure is detected to the point where the application is recovered and ready to continue its execution from the last checkpoint. The crashed processor is replaced by a spared processor, and its state is restored from the checkpoint on the crashed processor's buddy during this period of time. The results before the optimization and after are shown in Figure 6 and 7 for comparison. Since several barriers are involved in this process to ensure consistency until the crashed processor is recovered. By using barrier based on a transient spanning tree constructed during recovery, the complexity of the recovery overhead is decreased from $O(P)$ to $O(logP)$. Before optimization (Figure 6), the restart time increases almost linearly from 0.2 second on 4K cores to 3.6 second on 64K cores for both molecular systems, while after optimization, the restart time is dramatically reduced, which takes only 0.06 second on 4K cores up to 0.15 second on 64K cores for the 1 million atom system. Overall, the restart process of our in-memory fault tolerance scheme is very efficient, partly due to the fault tolerance protocol which allows the application to restart from the last checkpoint in the local memory. It also benefits from the fact that the application can restart without a full stop, so the job turn-around time and a new job submission are avoided.

V. Related Work

There are three main methods to checkpoint HPC application: uncoordinated checkpointing, coordinated checkpointing and communication-based checkpointing. In uncoordinated checkpointing, each process independently saves its state. The benefit is that a checkpoint can take place when it is most convenient and thus doesn't require synchronization to initiate checkpointing. However, uncoordinated checkpointing is susceptible to rollback propagation, the domino effect [14] which could cause systems to rollback to the beginning of the computation, resulting in the waste of a large amount of useful work. Guermouche et al. [15] proposed an uncoordinated checkpointing without domino effect with the help of logging useful application messages, which is applicable to Send-Deterministic MPI applications. Coordinated checkpointing requires processes to coordinate their checkpoints in order to form a consistent global state. Coordinated checkpointing simplifies recovering from failures because it does not suffer from rollback propagations. FTI [1] is a multi-level coordinated checkpoint scheme using topology-aware RS encoding with about 8% overhead. BLCR [16] implements kernel level checkpointing and is widely used in applications with production quality. In [17] Moody et al. proposed a multi-level checkpoint and used a Markov probability model to describe its performance. One drawback for those methods is that the application couldn't recover in the current run just after the failure happens but would require the user to rerun the application, reading the checkpoint form the disk. Communication-induced checkpointing allows the processes to take some of their checkpoints independently while preventing the domino effect by forcing the processors to take additional checkpoints based on protocol-related information piggybacked on the application messages it receives from other processors [18]. However it has scalability issues on large numbers of processors.

VI. Conclusion and Future Work

As the size of supercomputers increases, the probability of system failure grows substantially, posing an increasingly significant challenge for scalability. This paper presented several optimization techniques to a scalable double in-memory checkpoint/restart scheme to improve its scalability towards exascale. We demonstrate its performance with a million atom molecular dynamics simulation on up to 64K cores of a BlueGene/P machine, and show a checkpoint time in milliseconds. The restart times were measured to be less than 0.15 seconds on 64K cores.

In future, we plan to implement nonblocking checkpoint in our scheme. For applications with large memory footprint running on multicore machines, nonblocking checkpoint could further overlap checkpoints with computation and thus reduce the checkpoint overhead.

Acknowledgments

This work was partially supported by the US Department of Energy under grant DOE DE-SC0001845, by NSF grant OCI-0725070 for Blue Waters, and by the Institute for Advanced Computing Applications and Technologies (IACAT) at the University of Illinois at Urbana-Champaign. This work used machine resources from Teragrid under award ASC050039N.

References

[1] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, nov. 2011, pp. 1 –12.

[2] "Top500 supercomputing sites," http://top500.org.

[3] F. Cappello, "Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities," *Int. J. High Perform. Comput. Appl.*, vol. 23, pp. 212–226, August 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1572226.1572229

[4] 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 International Conference on Cluster Computing*, San Diego, CA, September 2004, pp. 93–103.

[5] G. Zheng, C. Huang, and L. V. Kalé, "Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for AMPI and Charm++," *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, vol. 40, no. 2, April 2006.

[6] L. V. Kale and G. Zheng, "Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects," in *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.

[7] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé, "Performance Evaluation of Adaptive MPI," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[8] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, College Station, Texas, October 2003, pp. 306–322.

[9] G. Zheng, O. S. Lawlor, and L. V. Kalé, "Multiple flows of control in migratable parallel programs," in *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*. Columbus, Ohio: IEEE Computer Society, August 2006, pp. 435–444.

[10] G. Antoniu, L. Bouge, and R. Namyst, "An efficient and transparent thread migration scheme in the PM^2 runtime system," in *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico. Lecture Notes in Computer Science 1586*. Springer-Verlag, April 1999, pp. 496–510.

[11] G. Bronevetsky, D. J. Marques, K. K. Pingali, R. Rugina, and S. A. McKee, "Compiler-enhanced incremental checkpointing for openmp applications," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 275–276. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345253

[12] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Comp. Syst.*, vol. 22, no. 3, pp. 303–312, 2006.

[13] J. W. Young, "A first order approximation to the optimal checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[14] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975, pp. 437–449. [Online]. Available: http://doi.acm.org/10.1145/800027.808467

[15] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic MPI applications," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 989–1000. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.95

[16] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for Linux clusters," *Journal of Physics Conference Series*, vol. 46, pp. 494–499, Sep. 2006.

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

[18] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A distributed domino-effect free recovery algorithm." in *Symposium on Reliability in Distributed Software and Database Systems'84*, 1984, pp. 207–215.