© 2012 Xiang Ni

A SEMI-BLOCKING CHECKPOINT PROTOCOL TO MINIMIZE CHECKPOINT OVERHEAD

BY

XIANG NI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Professor Laxmikant V. Kalé

Abstract

The increasing number of cores on current supercomputers will quickly decrease the mean time to failures (MTTF) of the system. With such high failure rates, long time running applications will have little chance to complete successfully if they don't use any fault tolerance strategy.

Double in memory/disk checkpointing is a production fault tolerance strategy in Charm++ runtime system. Each node will store one copy of its checkpoint in its own memory or disk as a local checkpoint and another copy in other node's memory or disk as a global checkpoint. This method takes advantage of the relatively high network bandwidth compared to I/O bandwidth. It is able to store a checkpoint faster than the traditional NFS- based checkpoint/restart.

However, as the core counts on each node keep increasing, the large checkpoint size of a node will quickly saturate the limited network bandwidth. In this thesis, we introduce the semi-blocking checkpoint/restart protocol to hide the checkpoint overhead by overlapping global checkpoint with applications. To further analyze the benefits of using semi-blocking checkpoint protocol in case of failures, we extend Daly's model and show the usefulness of the semi-blocking protocol for different kinds of applications.

Solid state disk (SSD) is used in the semi-blocking checkpoint protocol when there is no space to store checkpoint in memory. We present two strategies to choose what data to store in SSD based on the memory usage of applications.

In this thesis, we show the scalability and benefits of the semi-blocking checkpoint protocol. Semi-blocking checkpoint protocol has a performance improvement of 20%

compared to blocking checkpoint. And the overhead of semi-blocking checkpoint protocol could be as low as 1.6% with the consideration of checkpoints dumping time and the extra time to recover applications from failures.

To my parents.

Acknowledgements

I would like to thank my advisor Prof. Laxmikant V. Kalé for his consistent support on this project. His insightful thoughts and ideas, energy and enthusiasm always kept me going. I could always learn a lot from him when I met difficulties in this project.

Esteban Meneses also played important roles in this project. His experience and knowledge helped in finding the correct direction for my thesis work. And he is always ready to patiently hear my concerns and doubts.

I would like to express my many thanks to my colleagues in Parallel Programming Laboratory(PPL). Their help with CHARM++ related questions and other technical issues paved a smooth road for my project. Most of the results presented in the thesis were done on Trestles cluster maintained by SDSC. Help provided by the cluster administrators was critical to do these runs.

Finally, I would like to thank my family and friends who continue to impart principles in me and have been a very strong source of inspiration for me.

Table of Contents

List of Tables

List of Figures

1 Introduction and Motivation

One concern for the high performance computing community for Exascale is the ability to tolerate failures. Even when the mean time to failure(MTTF) of an individual node is still acceptable, as the number of nodes increases in Exascale, the reliability of the whole system will keep decreasing. Jaguar, the 3rd supercomputer on top500 today, experienced 2.33 failures/day from August 2008 to February 2010. The "ASIC Q" supercomputer at Los Alamos National Laboratories has a MTTF of less than 6.5 hours [1]. Such high failure rates will force today's scientific simulation to checkpoint frequently. However, periodically writing several GBs of data or even TBs to a NFS fault free space may consume up to 20% of the application time [2].

Two main methods are used to initiate checkpointing: system or application based. System based checkpoint could take up to 40 minutes to checkpoint for the best machines on top500 list(2008) [3, 4]. Application initiated checkpointing could help reduce checkpoint size by only checkpointing the data that couldn't be recovered or recomputed, however it would require programmer's effort in identifying which data needs to be stored in checkpoints. Some compiler technology is also used to pack the live variables at the time of checkpointing [5].

The diskless checkpoint protocol in Charm++ uses local memory or disk to store checkpoint data, and can take advantage of the high speed interconnect to speed up the checkpoint process. Basically, each Charm++ object will periodically create two checkpoints. One checkpoint is stored in its local memory or disk while the other one is stored in other node (called the buddy node) as a global checkpoint. When failures happen, the crashed node could recover from the check-

point in buddy node's memory or disk. Whereas the surviving nodes restore from their local checkpoint. This scheme is able to tolerate one single failure or multiple failures if the failed nodes do not store the checkpoint of each other [6].

In this thesis, we discuss the limitation of this blocking diskless checkpointing in Charm++ to large scale processors and propose a semi-blocking protocol which could further hide the high checkpoint overhead for applications of moderate memory footprint.

1.1 Network and Memory Constraints for Checkpointing

Double in memory/disk checkpointing requires each node to transmit the whole application checkpoint to its buddy every checkpoint interval. As seen in Table 1.1, a prediction for Exascale machine, the increase of network bandwidth couldn't catch up with the increase of memory and it falls far behind the speed up of FLOPS. The dumping time of global checkpoint will keep increasing as the memory consumption of applications grows up in Exascale. The emergence of communication avoiding algorithms [7] will aggravate this problem due to the redundant memory it uses to eliminate communication. During the diskless checkpointing, after local checkpoint is achieved on every node, there is no need for the application to stall for the accomplishment of global checkpoints since a consistent checkpoint copy has already been obtained. We explore to overlap global checkpoint with application execution to hide the checkpoint overhead as a semi-blocking protocol. Solid state disk(SSD) is also used as a back storage when memory becomes scarce.

Table 1.1: Projection of Exascale System

	1 petaFLOPS	1 exaFLOPS
FLOPS	10^{15}	10^{18}
# of sockets	20000	100000
Memory per socket	4GB	210GB
Memory BW	10GB/s	32GB/s
Network BW	2GB/s	20GB/s

1.2 Daly's Model for Optimal Checkpoint Interval

Checkpoint overhead of the whole application is also related to how frequently we do checkpoint. Daly [8] investigates the optimum checkpoint interval to minimize the application execution time. The total execution time is divided into:

$$T = solve\ time + dump\ time + rework\ time + restart\ time \tag{1.1}$$

The more frequently an application checkpoints, the more time an application would spend dumping checkpoint. However, the applications would experience less rework time on failures. So there is always a balance between the checkpoint dumping time, mean time to failure and rework time. In Daly's model, the optimum checkpoint interval is

$$\tau = \sqrt{2\delta(M+R)} \tag{1.2}$$

where δ is the checkpoint dumping time, M is the mean time to failure and R is the restart time.

The rest of the thesis is organized as follows: §2 will give the background of the Charm++ runtime system and the blocking checkpoint protocol in Charm++, §3 introduces the semi-blocking protocol for Exascale and a model of it to minimize the execution time, §4 details the implementation of the semi-blocking on Charm++,

experiment results and analysis will show in §5, we review the related work in this area in §6, and we conclude the work in §7.

2 Blocking Checkpoint in Charm++

In this section, we summarize the design of blocking in-memory checkpoint-based fault tolerance scheme, which is a production fault tolerance strategy in CHARM++ and has been used for years. The supporting parallel runtime systems are CHARM++, a message driven runtime system, and Adaptive MPI [9], an implementation of MPI on top of Charm++. These fault tolerant runtimes take advantage of the migratable objects and threads.

2.1 Runtime Support for Checkpoint/Restart

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

Adaptive MPI [10] runs MPI "processes" in light-weight threads, which are easier to checkpoint and restart compared to processes. Thread migration during restart would raise the problem of pointer reference. *Isomalloc* [10, 11] is used to solve this problem for fully automated checkpointing, similar to the technique in the PM^2 system [12]. 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. This method reduces the amount of data to checkpoint, and so checkpointing becomes faster.

2.2 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 in-memory checkpointing scheme[6] introduced the idea of diskless checkpointing that checkpoints data in memory. It uses a coordinated checkpoint strategy, which requires 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 node is not sufficient as illustrated in Figure 2.1. In this scenario with 4 nodes, each CHARM++ object (represented as a circle) checkpoints only one copy of its checkpoint (represented as a triangle). When node 2 crashes, the checkpoints for object d and e in memory of that node 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 2.2 illustrates an example of this scheme. The top half of the figure shows the scenario before one node crashes. Each circle represents an object being checkpointed, while each triangle and square represents its first and second checkpoints. We call these two nodes *buddy nodes* for the checkpointing object. Note that one of the two *buddy nodes* can be the same node where the object resides. This can

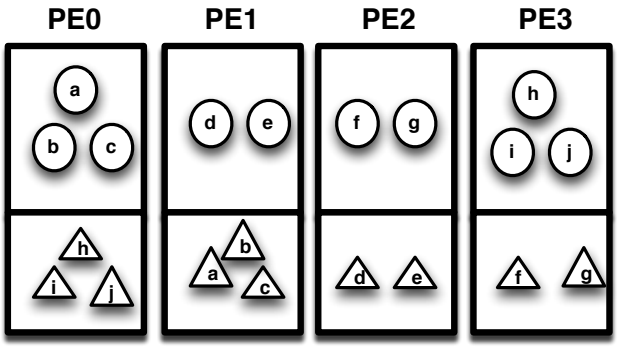

Figure 2.1: In-Memory Single Checkpoint

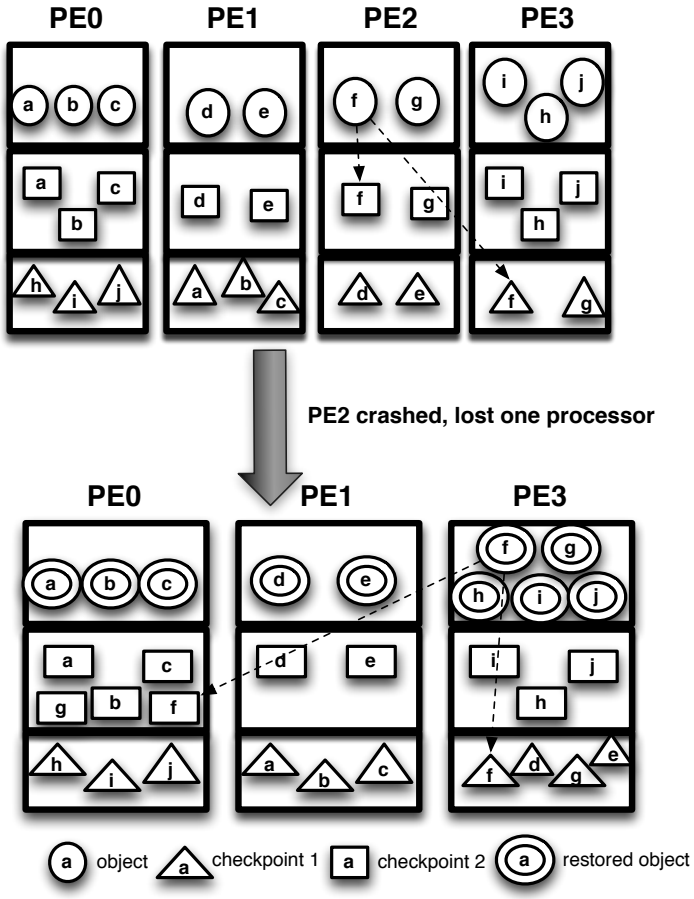

Figure 2.2: In-Memory Single Checkpoint

help reduce the checkpointing overhead, since the checkpointing is basically a local memory copy, which is much faster than accessing memory of remote nodes. 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 node. 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 nodes examine the checkpoints in their memory and check for missing buddy nodes. A new node is chosen (which can be either a spare node, or a running node) to replace the crashed node and the latest checkpoint data is copied to that node to maintain the double checkpoints. One of the two buddy nodes is then responsible for restoring the corresponding objects from its checkpoints in memory. At restart, if the replacement node is from a running node, then a load imbalance may occur since that node restores more checkpoints. This can be fixed by a load balancing phase after restart [6].

The bottom half of the Figure 2.2 illustrates a snapshot of the objects and their checkpoints distributed on nodes after a recovery is complete. The lost checkpoints on the crashed node 2 are recovered to node 3 and node 0 respectively. Node 3 is chosen to restore node 2's objects(f,g) locally to avoid communication overhead, since node 3 is node 2's original *buddy* node.

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

2.3 Dumping Time Increases with Checkpoint Size

The blocking double in-memory checkpoint protocol scales well for application of small or moderate memory footprint. In one of our previous papers [13], we show the good scalability of the checkpoint and restart time using this protocol for LeanMD

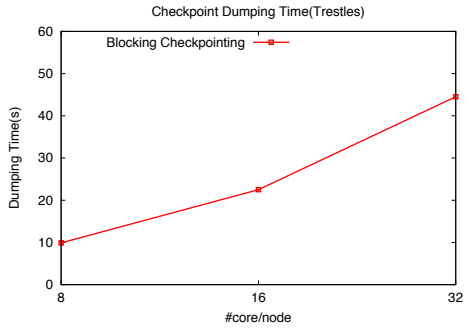

Figure 2.3: Increase of the dumping time with number of cores on node

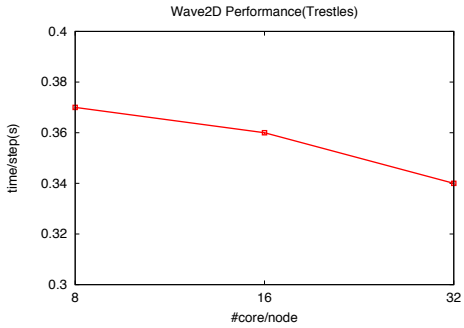

Figure 2.4: Performance improvement using more cores on each node

on up to 64K cores of BG/P machines.

However our experiment of wave2D benchmark with problem size of 58368 × 15872 shows checkpoint dumping time increases as the number of cores used per node increases on Trestles where each physical node has 32 cores. In Figure 2.3 and Figure 2.4, wave2D is run on 128 cores, however, we use different number of cores per node. So with 8 cores per node, we end up using a total of 16 physical nodes to run the application. While with 32 cores per node configuration, we use 4 physical nodes on Trestles cluster. Using more cores on one node could help us improve the performance of wave2D as seen in Figure 2.4 since some inter-node communication is avoided. However, checkpoint dumping time increases from 10 seconds with 8 cores per node configuration to 45 seconds while using all of the 32 cores per node in Figure 2.3. Our blocking checkpoint protocol will have performance limitation as the increasing number of cores per node.

3 Semi-Blocking Checkpoint Protocol

Coordinated checkpointing will quickly saturate the network bandwidth with the increase of checkpoint size, making checkpoint overhead unacceptable for applications at exascale. In this section we show the effort of semi-blocking checkpoint protocol in hiding the checkpoint overhead. A model will also be introduced to demonstrate the benefits of semi-blocking protocol.

3.1 Solution: Semi-Blocking Checkpointing

Semi-blocking protocol is based on the double in-memory checkpoint protocol in Charm++ [6].

Figure 3.1 illustrates the operation of semi-blocking checkpoint protocol. In Figure 3.1, three nodes will synchronize and then dump their local checkpoints to local memory or disk coordinately. After all the nodes are done with their local checkpoints, they are safe to continue with the computation. In the mean time, Charm++ runtime system will take over to make global checkpoints. The red lines show that checkpoint of each node is sent to its buddy node as global checkpoints. Previous checkpoints are still kept in memory or local disk of each node until the global checkpoints are accomplished on all the nodes in case of failures happening during global checkpoints. Figure 3.2 and Figure 3.3 show how two types of failures are dealt with in the semi-blocking checkpoint protocol.

Two questions need to be answered before demonstrating the benefit of semi-blocking checkpoint protocol.

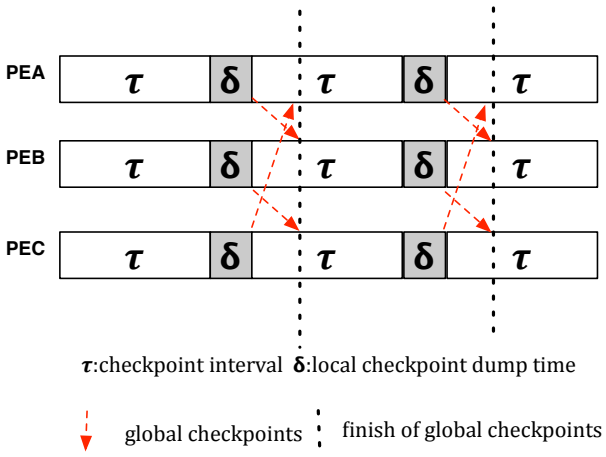

Figure 3.1: Forward Path of SemiBlocking Protocol

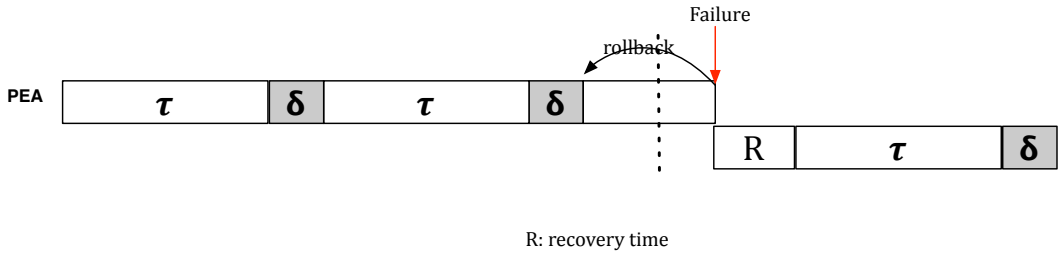

Figure 3.2: Failure happens after global checkpoint is done, application could recover from the latest checkpoint

• How many failures will force the application to rollback to a previous checkpoint? The number of failures inducing a further rollback depends on how soon the global checkpoint is finished. We use *overlap* to illustrate the time to finish a global checkpoint in seconds as global checkpoint always overlapping with application. The smaller the *overlap* is, the less amount of failures will require a further rollback.

• Will semi-blocking protocol slow down the application? Global checkpoints and application would share the same NIC for communication, and thus the message sending for global checkpoints may interfere with the execution of application. We use *overhead* as the slow down of application in seconds in each checkpoint interval. The less the *overhead* is, the more we could gain from semi-blocking checkpointing in a forward case with no failures.

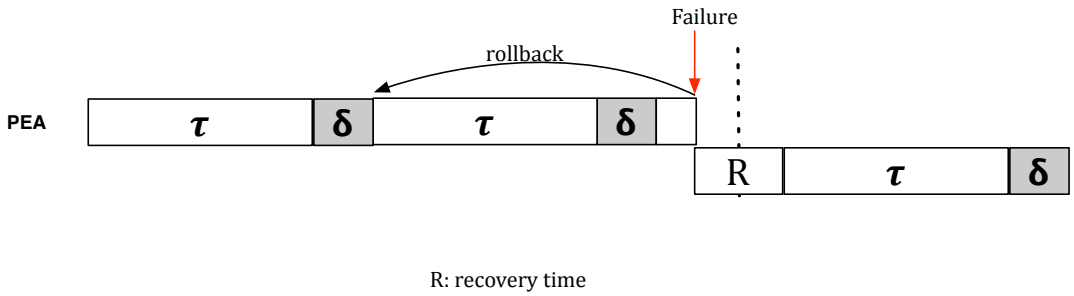

Figure 3.3: Failure happens during global checkpoint, application must discard the current checkpoint and recover from the previous checkpoint

3.2 A New Model for Semi-Blocking Protocol

Table 3.1: Semi-Blocking Checkpoint Parameters

$overlap$	The time to finish global checkpoint
$overhead$	Interference of global checkpoint to application
f_a	The number of failures not in global checkpoint
τ	Checkpoint interval using semi-blocking protocl
$\tau_{blocking}$	Checkpoint interval using blocking protocl
δ	The dumping time of local checkpoint
c	The dumping time of checkpoint using blocking protocol
R	The recover time
$MTTF$	The system mean time to failure
T_s	The workload of application
T_{total}	The total execution time using semi-blocking protocol
$T_{blocking-total}$	The total execution time using blocking protocol
$benefit$	The performance improvement of semi-blocking over blocking protocol

Daly's model for the optimum checkpoint interval seeks a balance between the time to dump checkpoints with MTTF. However their model only supports blocking checkpoint/restart. We extend Daly's mode for the semi-blocking protocol.

Table 3.1 lists the parameter used in the model. The total time of a checkpointed

workload with failures would be divided into five parts:

$$\begin{aligned} T_{total} &= T_s + T_{local} + T_{overhead} \\ &+ T_{rollback,recover} + T_{further-rollback} \end{aligned} \tag{3.1}$$

where T_s is the pure computation time of the application without doing any checkpoints, T_{local} is the time to dump checkpoints in local memory or disk, $T_{overhead}$ is the total interference of global checkpoint to application, $T_{rollback,recover}$ is corresponding to recover from failures described in Figure 3.2, when global checkpoint is finished, $T_{further-rollback}$ is the cost to recover from failures during the global checkpoint as in Figure 3.3.

The local checkpoints dumping time T_{local} is the product of the number of checkpoints and the dumping time of each checkpoint. The pure computation time in each checkpoint interval would be (τ-$overhead$), thus,

$$T_{local} = \frac{T_s}{\tau - overhead}\delta \tag{3.2}$$

$T_{overhead}$ is calculated in the similar way,

$$T_{overhead} = \frac{T_s}{\tau - overhead} overhead \tag{3.3}$$

Failures are distributed exponentially according to Daly's mode. The number of failures not happening during global checkpoint would be,

$$f_a = \left(1 - \frac{overlap}{\tau}\right) \frac{T_{total}}{MTTF} \tag{3.4}$$

Overhead is avoided when recovering from such failures since there is no need to redo any checkpoints, so

$$T_{rollback,recover} = f_a \left(R + \frac{\tau + overlap}{2} - overhead \right). \tag{3.5}$$

And R would be at the same order of magnitude with the τ in blocking checkpoint protocol since restart is the reverse process of checkpointing as seen in §2.

For failure during global checkpoints, application needs to recover from the previous checkpoint and redo the current checkpoint, thus,

$$\begin{aligned} &T_{further-recover} = \\ &\frac{overlap}{\tau} \left(R + \tau + \frac{overlap}{2} + c - overhead \right) \frac{T_{total}}{MTTF} \end{aligned} \tag{3.6}$$

After all, the total execution time with checkpoints and rework for a workload of T_s would be

$$\begin{aligned} T_{total} &= T_s + \frac{T_s}{\tau - overhead}\delta + \frac{T_s}{\tau - overhead} overhead \\ &+ \left(1 - \frac{overlap}{\tau} \right) \left(R + \frac{\tau + overlap}{2} - overhead \right) \frac{T_{total}}{MTTF} \\ &+ \frac{overlap}{\tau} \left(R + \tau + \frac{overlap}{2} + c - overhead \right) \frac{T_{total}}{MTTF}. \end{aligned} \tag{3.7}$$

Similarly, for the blocking checkpoint, the total execution time $T_{blocking-total}$of an application with T_s workload is

$$T_{blocking-total} = T_s + \frac{T_s}{\tau_{blocking}} c + \frac{T_{blocking-total}}{M} \left(R + \frac{\tau_{blocking} + c}{2} \right) \tag{3.8}$$

Next, we use MATLAB to minimize 3.7 and 3.8 by finding the optimum checkpoint interval for the two protocols. Checkpoint interval τ has strict bounds in the

semi-blocking model: $overlap < \tau < MTTF$. So We optimize the T_{total} in the given bounds with *overlap* and *overhead* as input for the semi-blocking model. In the blocking checkpoint model $\tau_{blocking}$ has the bound of $0 < \tau_{blocking} < MTTF$.

To quantify the performance improvement using semi-blocking checkpoint protocol compared to the blocking one, we calculate the $benefit$ of semi-blocking protocol like

$$benefit = \frac{T_{blocking-total} - T_{total}}{T_{blocking-total}}. \tag{3.9}$$

Here, T_{total} and $T_{blocking-total}$ are the total execution time of the application with the optimum checkpoint interval using semi-blocking and blocking protocol respectively.

4 Implementation

In this section, we describe the implementation of the semi-blocking checkpoint protocol in Charm++. We quantify the interference of global checkpoint to applications with different communication to computation ratios. Two schemes are developed to ensure global checkpoint is finished in an appropriate time period. We also discuss how to use solid state disk in the semi-blocking protocol to reduce memory pressure.

4.1 Charm++ for Multicore Clusters

Charm++ runtime system provides an *SMP* extension for each platform specific implementation to explore the shared memory of multicore machines. The SMP version of Charm++ allows the process space to consist of multiple flows of control as *worker threads* instead of one. Worker threads are typically implemented via pthreads and share the process's address space but have their own event schedulers. Using SMP mode of Charm++, we can achieve faster startup, reduction in memory consumption and an optimized node-level collective communication. Charm++ works for both SMP and non-SMP versions without any SMP specific changes to applications.

In SMP mode of Charm++ runtime system, one node has a dedicated *communication thread* to handle all the inter-node communication while cores on the same node communicate using shared memory. The worker threads don't need to pay for the communication overhead themselves. Communication thread is bound

to a certain core per node. Therefore, cores on the same node are mapped either to worker threads or communication thread. When a worker thread sends a network message, it enqueues the message to the communication thread's outgoing message queue. More benefits of the communication thread are showed in [14].

4.2 Overlapping Global Checkpoint and Computation

During global checkpoint, each node sends checkpoint message to its buddy; those messages are enqueued in the outgoing message queue right after local checkpoint is finished. Thus, the communication of application is stalled by the sending of checkpoint messages. To solve this problem, we design a separate checkpoint message queue on each communication thread. Worker threads enqueue the checkpoint message to this separate queue. The communication thread only sends checkpoint messages when there is no application message ready to be sent to ensure the minimum interference of the checkpoint message to application. We call it opportunistic sending of checkpoint messages. The checkpoint message is split into multiple small chunks for better overlap with the computation of applications.

Interference of the global checkpoint to application is heavily affected by the communication needs of applications. Figure 4.1 shows different communication to computation ratios we achieve using a synthetic benchmark: FT_Test with a dumping time of 6.5 seconds using blocking checkpoint protocol. With FT_Test, users could control the computation time, number of messages and message size of each computation step to see the sensitivity of the semi-blocking protocol. As seen in Table 4.1, with the increase of communication to computation ratio, *overhead* will also increase. And *overlap* has an upward trend until the communication to computation ratio reaches 3.4. Figure 4.2 corresponds to the benefit of semi-blocking

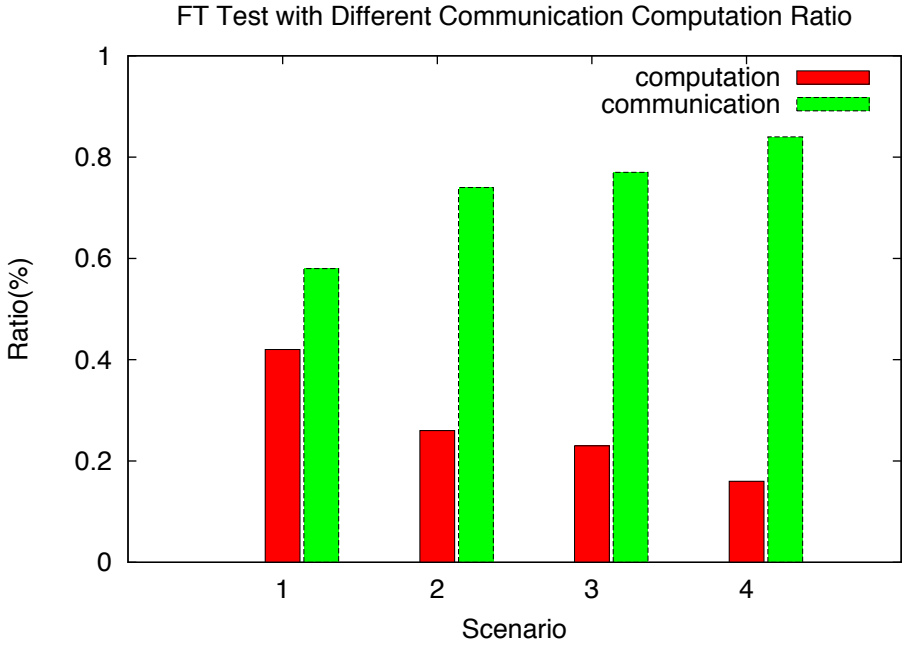

Figure 4.1: Different Communication/Computation Ratio for FT Test,Checkpoint Size is 512MB/node

protocol for different communication to computation ratios. Semi-blocking protocol gives us more than 5% benefit compared to blocking checkpoint protocol with a communication to computation ration up to 3.4. With the emergence of communication avoiding algorithm [7], the checkpoint data size will increase on each node because of the redundant computation which means checkpointing using blocking protocol will take a lot of time and the communication to computation ratio will decrease. We believe the semi-blocking protocol will show more and more benefits as programmers try to cut down the communication to computation ratio of applications.

Table 4.1: Overlap and Overhead associated with Different Communication to Computation Ratio

Scenario	Comm/Comp Ratio	$Overlap(s)$	$Overhead(s)$
1	1.4	9.6	0.2
2	2.9	13.5	0.5
3	3.4	17.3	1.0
4	5.1	14.3	4.0

4.3 Opportunistic and Random Scheduling of Checkpoint Message

Ideally, semi-blocking protocol could achieve the best benefit with the smallest *overlap* and the smallest *overhead*. Reducing overlap period decreases the number of failures that require a further rollback. On the other hand increasing the overlap period may incur less overhead to applications. Finding a good overlap period is critical to the success of semi-blocking protocol.

Opportunistic sending of checkpoint messages will only give us a fixed overlap period. Will this fixed overlap and overhead relationship bring the most benefit of semi-blocking protocol? An option is to use the idea of *lottery scheduling* [15] to control the overlap period. Lottery scheduling is a randomized resource allocation mechanism used to control the relative execution rates of computations. It also supports resource management such as I/O bandwidth or memory. The clients' allocations to access the shared resource are represented by the number of lottery tickets they hold. Each time the resource is granted to the client with the winning ticket. In the semi-blocking protocol, transmission of global checkpoint messages and application messages share the same network interface controller (NIC). By controlling their allocations to use the NIC, we could have different overlap periods.

In FT_Test, the amount of messages sent is evenly distributed over computation. Thus we use the number of application messages sent to represent the amount of

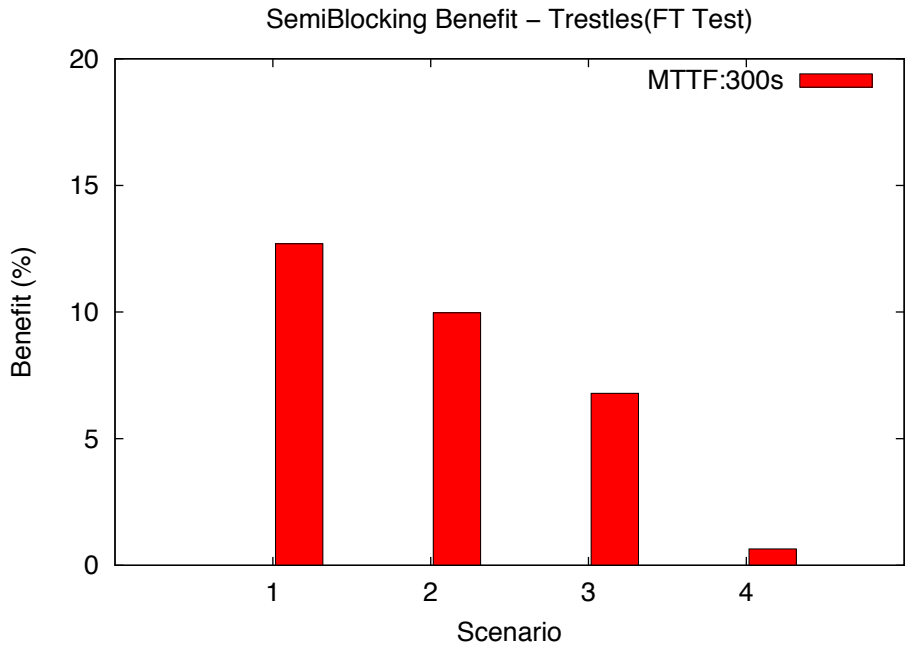

Figure 4.2: Benefit of Semi-Blocking is Affected by the Communication/Computation Ratio, Checkpoint Size is 512MB/node, MTTF: 300s

work that has been finished. The longer the *overlap* is, the more application messages will be sent through global checkpoint. The number of application messages s and the number of checkpoint messages c in each checkpoint interval could be statistically obtained from previous checkpoints. Given an expected *overlap*, the number of application messages sent in global checkpoint would be approximately

$$\frac{overlap}{\tau}s,$$

while the number of checkpoint messages are c. The number of application and checkpoint messages sent in global checkpoint can be used as their lottery tickets.

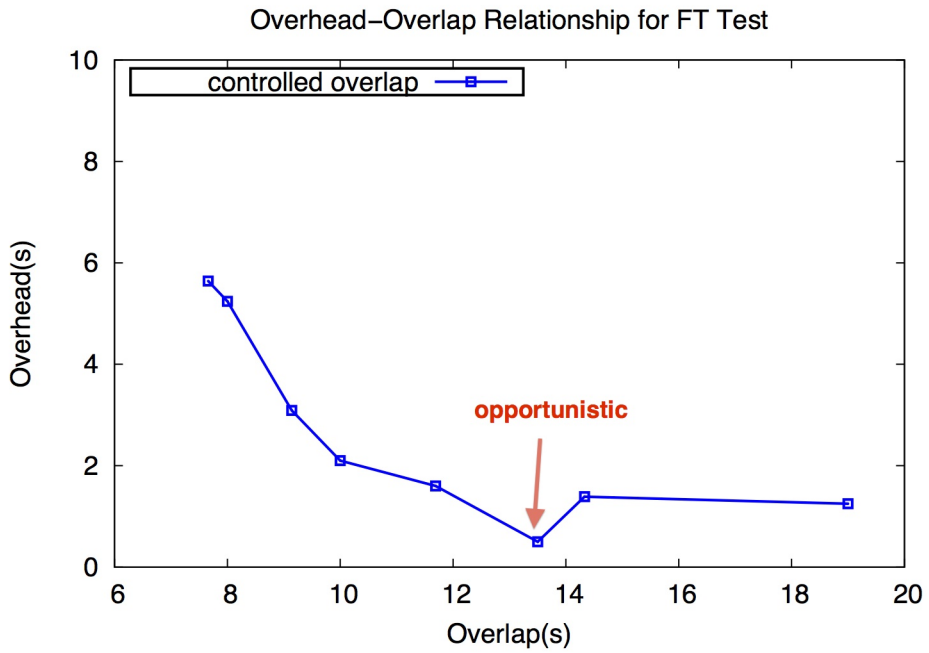

Figure 4.3: Overhead Associated with Different Overlap for FT Test Scenario 2

So the probability to send an application message during global checkpoint is

$$\frac{\frac{overlap}{\tau}s}{\frac{overlap}{\tau}s + c},$$

and the probability to send a checkpoint message is

$$\frac{c}{\frac{overlap}{\tau}s + c}.$$

As seen in Figure 4.3 for a FT_Test of communication/computation ratio set to 2.9, the overhead of opportunistic sending of checkpoint message is the least. Increasing the overlap period will increase the overhead slightly while decreasing the overlap period will make the overhead dramatically increasing, both fail to buy us more benefit using the model in §3 as seen in Figure 4.4.

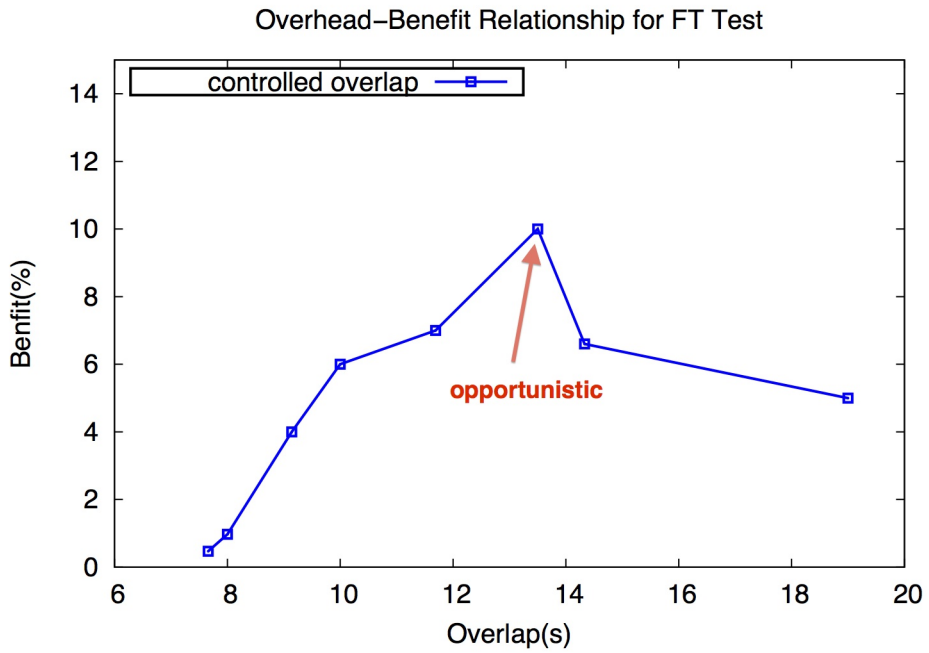

Figure 4.4: Benefit of Semi-Blocking Protocol Associated with Different Overlap for FT Test Scenario 2, MTTF:300s

4.4 Relieving Memory Pressure of Checkpoint with Solid State Disk

Solid state disk (SSD) is becoming more and more promising for its good random access performance and low power consumptions, however there is no clear answer of how to use it for checkpointing. As exascale machines are targeted for large and complicated scientific application, SSD would be a good choice to store the huge checkpoint data. Considering the SSD bandwidth is not comparable to memory bandwidth nowadays, we need to carefully select what checkpoint data to store in SSD.

In the following analysis we set the total memory of system to be M and application memory usage to be M_a. Checkpoint memory is less than or equal to

M_a. With the semi-blocking protocol, the peak memory consumption could be as high as $5M_a$ during global checkpoint and falling down to $3M_a$ when global checkpoint is finished. Two strategies are designed for the use of SSD depending on the application memory usage for semi-blocking protocol.

• When application memory consumption M_a is more than $M/3$, all the checkpoints will be saved to SSD. This is called the full SSD strategy.

• When application's memory consumption M_a is less than $M/3$, only the buddy's checkpoints are stored in SSD. In this way half the writes to SSD are avoided during checkpoints. At restart, only checkpoints of the crashed node need to read from SSD while other nodes could recover from the checkpoints in their local memory. We call it half SSD strategy.

In §5 we will show the performance differences of the two strategies at restart and how half SSD strategy could help us achieve a faster recover.

4.5 Virtualization Analysis

Over decomposition and asynchronous communication in CHARM++ can greatly help overlap communication and computation of applications. In CHARM++, programs are broken up into objects called chares. Usually, there are more chares than the number of processors. The number of chares divided by the number of processors is called *virtualization ratio*. A chare will begin computation by the invocation of entry methods associated with each CHARM++ message.

We illustrate the benefit of high virtualization ratio for the semi-blocking protocol in Figure 4.5 and Figure 4.6. Different virtualization ratios wouldn't change the total amount of data to be communicated with, but the number of messages and each message size. In Figure 4.5 and 4.6, we set the application virtualization ratio to be 1 and 4 seperately. As can be seen in Figure 4.6, high virtualization

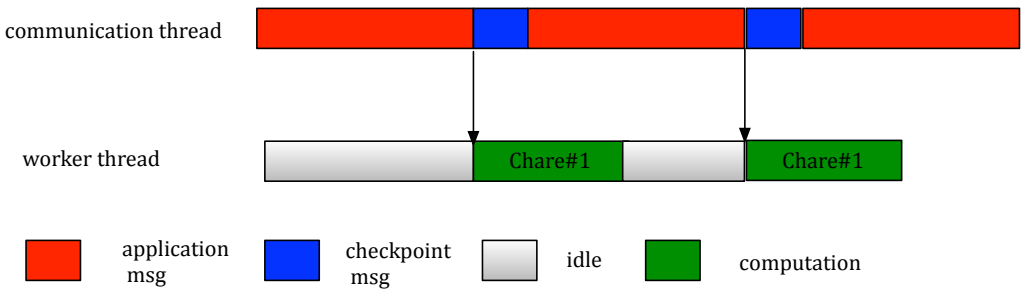

Figure 4.5: Virtualization Ratio of 1

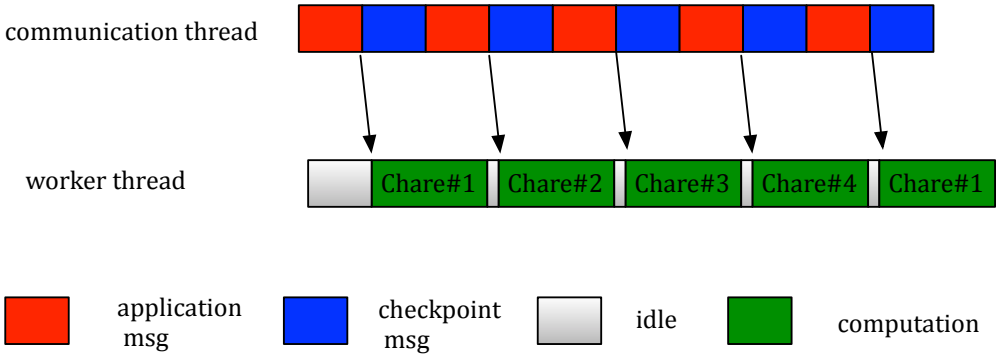

Figure 4.6: Virtualization Ratio of 4

will keep the processor busy and hide the transmission of the checkpoint message. In Figure 4.5 with virtualization ratio of 1, the idle time of the processor will be longer. Even if we send one checkpoint message after the sending of each application message, there is potential to enlarge the overlap period for global checkpoint with a low virtualization ratio.

5 Experiments and Analysis

We evaluate the benefit of the semi-blocking checkpoint protocol as well as the performance of restarting application after a failure.

Three applications are used in the experiments because of their different communication and computation patterns. The first one is wave2D, which uses a finite differencing scheme to calculate pressure information over a discretized 2D grid. The second one is a canonical benchmark, jacobi2D, that uses 5 point stencil to average values in a 2D grid using 2D decomposition. The third application, ChaNGa, is for N-Body based parallel simulations and especially used in Cosmology and Astronomy [16].

The experiments are done on Trestles at the San Diego Supercomputer Center. Trestles consists of 324 nodes with 32 cores per node. The theoretical peak performance of the system is 100 teraflops. Each compute node contains four sockets, each with a 8-core 2.4 GHZ AMD Magny-Cores processor. Each node has 64 GB of DDR3 RAM and 120GB of flash memory(SSD).

5.1 Benefits of Semi-Blocking Protocol

We use wave2D to demonstrate the benefits of semi-blocking Protocol with different problem size on 512 cores of Treatles. The checkpoint size goes up with problem size from 0.45GB/node to 4GB/node. Since there is no solid answer about the MTTF in exascale, projections goes from 1 minutes [17] to half an hour [18], we end up using different values of MTTF for our model as seen in Figure 5.1.

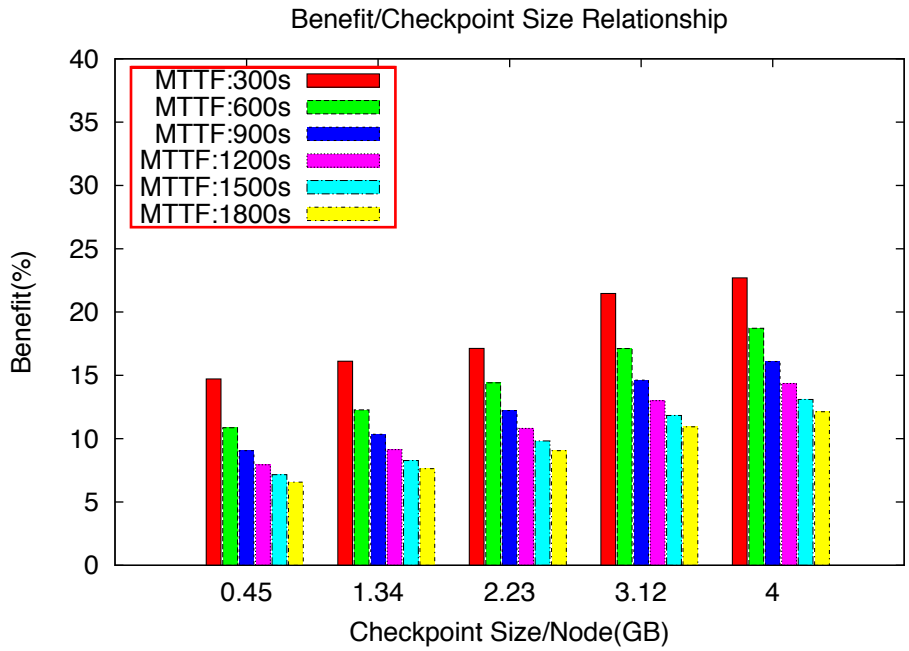

Figure 5.1: Benefits of SemiBlocking Protocol for Wave2D on 512cores

Suppose the problem size for wave2D to be N. The computation complexity of wave2D would be $O(N)$, while the communication complexity is $O(\sqrt{N})$, and the checkpoint size is $2N$. The communication to computation ratio of wave2D will decrease as N increase. So semi-blocking is expected to have more benefits with the increase of problem size according to §4.

The benefit of semi-blocking protocol increases with the checkpoint size from 15% for 0.45GB to 22% for 4GB. And also with decrease of MTTF, semi-blocking protocol shows off more benefit. This is because that the checkpoint and restart overhead for blocking checkpoint protocol will keep increasing as MTTF goes down while semi-blocking protocol does a good job in hiding the checkpoint overhead and provides a tolerable rework time on failures. As seen in the Figure 5.1, for checkpoint size of 4GB/node, benefit of semi-blocking protocol goes from 10% for MTTF of

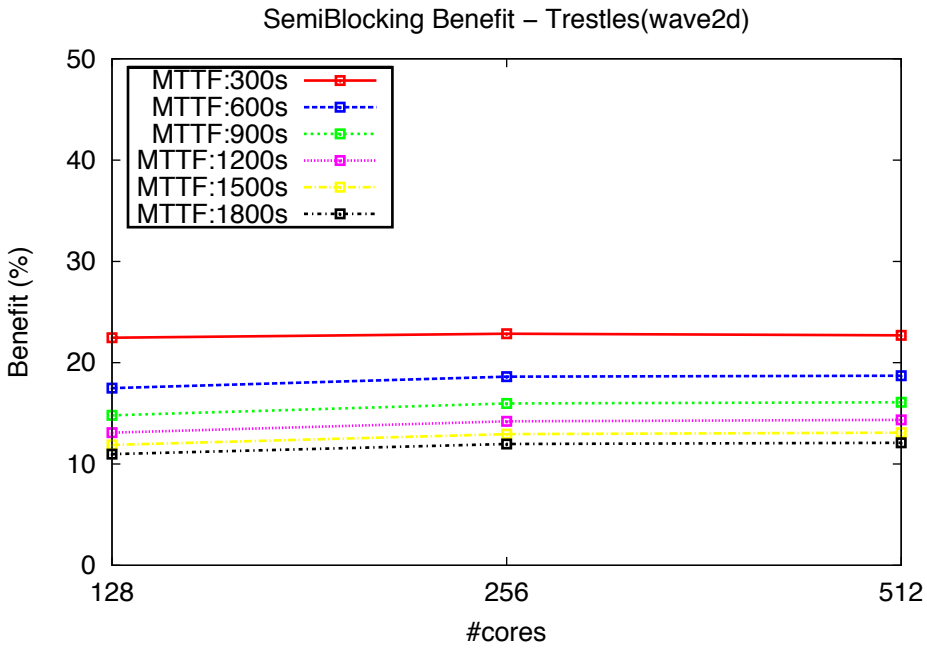

Figure 5.2: Weak Scalability

1800s to 22% for MTTF of 300s.

5.2 Scalability

In Figure 5.2 we first shows the weak scalability of the semi-blocking protocol with wave2D of a constant checkpoint size of 4GB per node. Semi-blocking checkpoint protocol has good weak scalability from 128 cores to 512 cores. For certain MTTF, the benefit of semi-blocking checkpoint protocol keeps a straight line from 128 cores to 512 cores. The benefit goes from 22% with MTTF of 300s to 10% with MTTF of 1800s. We could foresee the semi-blocking protocol would have more benefit with the decrease of MTTF in exascale.

The ChaNGa application is used to demonstrate the strong scalability of the

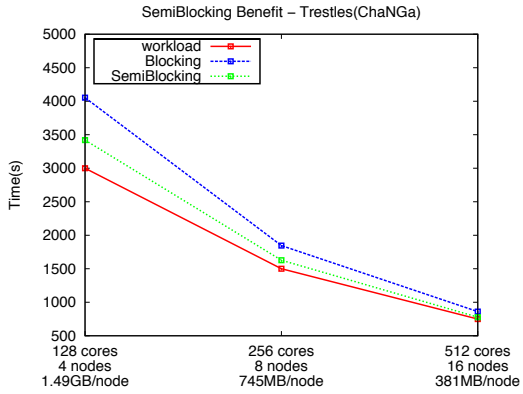

Figure 5.3: Strong Scalability, MTFF:600s

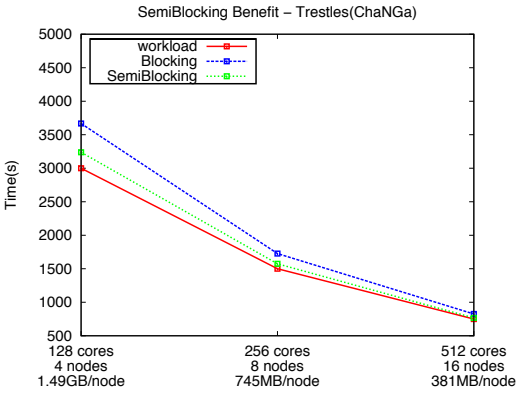

Figure 5.4: Strong Scalability, MTFF:1200s

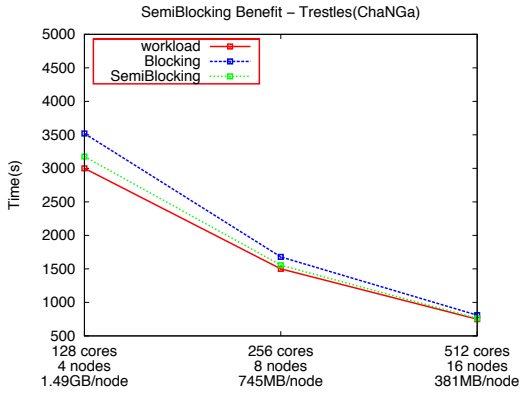

Figure 5.5: Strong Scalability, MTFF:1800s

semi-blocking protocol. We use a 50 million particles system for ChaNGa. The checkpoint size decreases from 1.49GB per node on 128 cores to 381MB per node on 512 cores. Figure 5.3, 5.4 and 5.5 show the overhead of blocking and semi-blocking checkpoint protocol compared to the normal workload with different value of MTTF. With a MTTF of 1800s, the overhead of semi-blocking checkpoint protocol is only 1.6% compared to pure computation time even with the consideration of the time to recover and restart when failures happen. However the overhead of blocking checkpoint protocol is as high as 8%. Even with a MTTF of 600s, semi-blocking protocol will only impose an overhead of 5% to application while the overhead of

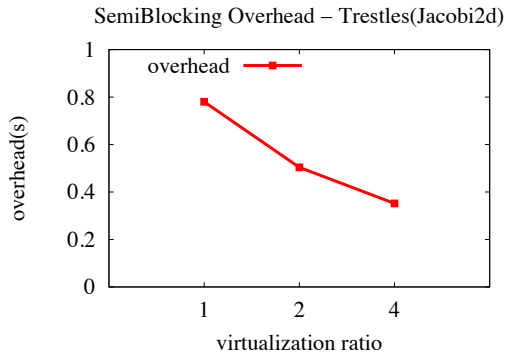

Figure 5.6: Virtualization Effect on Overhead for Jacobi

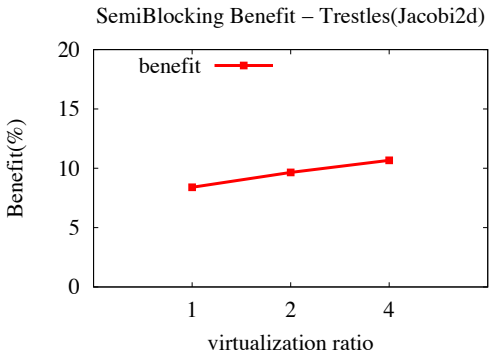

Figure 5.7: Virtualization Effect on Benefit for Jacobi

blocking protocol will go up to 14.8%. The overhead of semi-blocking decreases as the number of cores increase, which is due to the fact that checkpoint dumping time becomes less with the increase of core number in a strong scale test. The optimum checkpoint interval would also decrease since we can afford to do more checkpoints and thus there is less overhead to recover from failures. So the overhead of semi-blocking checkpoint is falling down as the number of cores increase.

5.3 Benefits of High Virtualization Ratio

The experiment result with different virtualization ratio on 128 cores is consistent with the analysis in §4. We use a jacobi2D benchmark of a checkpoint size 512MB/node. The overhead of global checkpoint per checkpoint interval decreases from 0.78 seconds with 1 chare per core to 0.352 seconds with 4 chares per core in Figure 5.6. Correspondingly, the benefit of semi-blocking protocol to blocking version increases from 8.4% to 10.7% with a MTTF of 300 seconds as seen in Figure 5.7.

Appropriate virtualization ratio will not only help overlap normal communication of application with computation, but also hide the global checkpoint communication,

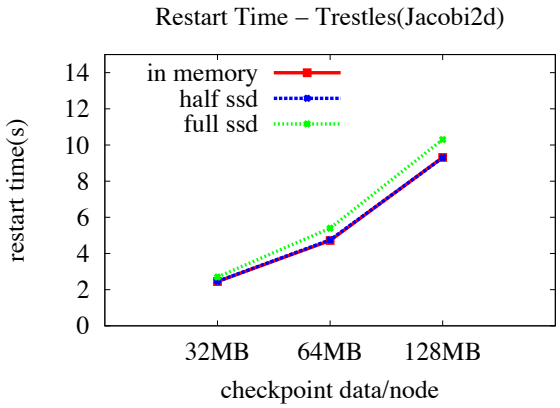

Figure 5.8: Restart time for Jacobi (Trestles)

trying to minimize the interference of global checkpoint to application running.

5.4 Restart

As discussed in section §4, half SSD and full SSD scheme could be adaptively used depending on the memory consumption level of applications. During restart, half SSD scheme only requires to fetch the crashed node's checkpoints from SSD, while all the nodes need to access SSD for checkpoint data in the full SSD scheme.

Figure 5.8 shows the restart time of the default(in-memory), half SSD and full SSD scheme for a jacobi2D benchmark with 512MB of checkpoint data per node . As we could see, half SSD scheme has negligible overhead compared to the default scheme, while full SSD scheme has around 1s overhead. We should always use half SSD scheme when memory consumption is not that high for quick restart.

6 Related Work

There are three main methods to checkpoint HPC applications: 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 [19] 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.[20] proposed an uncoordinated checkpointing without domino effect with the help of logging useful application messages, which is applicable to Send-Deterministic MPI applications. The logging of application message may also consume much memory on computation nodes.

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. BLCR [21] implements kernel level checkpointing, but will incur in a lot of overhead for application at production level. Some multi-level approaches has been proposed recently to deal with failures at different frequency of occurrence. FTI [2] is a multi-level coordinated checkpoint scheme using topology-aware RS encoding with about 8% checkpoint overhead. In [22] 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. Darius et al. [23] provides the comparison of the blocking and non-blocking approach for coordinated checkpointing. Their non-blocking protocol is based on Chandy-Lamport algorithm[24] and writes the checkpoints to a checkpoint server.

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 [25]. However it has scalability issues on large numbers of processors.

Solid state disk is used to store local and global checkpoints in [1] and show significant benefit to hard drive disk with the consideration of failures. However in their approach, application could only resume computation after a global checkpoint is achieved on all the processors. In [26], the authors replace hard disk drive with SSD and using staging IO to buffer the write to SSD and show a benefit of 55%.

7 Conclusion and Future Work

As the size of supercomputers increases, the probability of system failure grows substantially, posing an increasingly significant challenge for scalability. Complicated applications on large scale machines don't allow us to count on the low network bandwidth for fast checkpoints.

This thesis proposes a semi-blocking checkpoint protocol to reduce the checkpoint overhead and lessen the effect of failures. With this protocol, applications can quickly resume to computation after local checkpoint is done. And the overhead of global checkpoint could almost be hidden for applications of certain communication to computation ratios. A full implementation of the semi-blocking protocol is accomplished in Charm++ runtime system. Then we extend Daly's model to compute the optimum checkpoint interval and minimize the execution time of the semi-blocking checkpoint protocol.

We show the benefits of our protocol with three different kinds of applications wave2D, jacobi2D and ChaNGa. The semi-blocking protocol has good strong and weak scalability with different MTTF projected for exascle. For a wave2D benchmark of 4GB checkpoint data per node, semi-blocking has over 20% benefit to blocking protocol with MTTF of 300s and 10% benefit with MTTF of 1800s. For a 50 million particles simulation with ChaNGa, semi-blocking protocol would only incur a 1.6% overhead to pure computation with the consideration of checkpoint time and recover time on failures while blocking protocol has an overhead of 8%.

In future, we plan to use asynchronous IO access to SSD for better overlap of checkpoint and computation and fast recover of application using full SSD scheme.

Remote direct memory access (RDMA) could also be used in our protocol to reduce the interference of NIC to applications.

References

[1] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654117 pp. 57:1–57:12.

[2] 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.

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

[5] 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. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345253 pp. 275–276.

[6] 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.

[7] E. Solomonik, A. Bhatele, and J. Demmel, "Improving communication performance in dense linear algebra via topology aware collectives," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063487 pp. 77:1–77:11.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] G. Zheng, X. Ni, E. Meneses, and L. Kale, "A scalable double in-memory checkpoint and restart scheme towards exascale," Parallel Programming Laboratory, Tech. Rep. 12-04, Feburary 2012.

[14] C. Mei, Y. Sun, G. Zheng, E. J. Bohm, L. V. Kalé, J. C.Phillips, and C. Harrison, "Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime," in *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[15] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: flexible proportional-share resource management," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, ser. OSDI '94. Berkeley, CA, USA: USENIX Association, 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267638.1267639

[16] F. Gioachin, A. Sharma, S. Chakravorty, C. Mendes, L. V. Kale, and T. R. Quinn, "Scalable cosmology simulations on parallel machines," in *VECPAR 2006, LNCS 4395, pp. 476-489*, 2007.

[17] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, D. Barkai, T. Boku, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, R. Fiore, A. Geist, R. Harrison, M. Hereld, M. Heroux, K. Hotta, Y. Ishikawa, Z. Jin, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, B. Lucas, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. Mueller, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, A. V. D. Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The international exascale software project roadmap 1."

[18] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems peter kogge," 2008.

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

[20] 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. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.95 pp. 989–1000.

[21] 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.

[22] 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.

[23] D. Buntinas, C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi protocols," *Future Generation Computer Systems*, vol. 24, no. 1, pp. 73 – 84, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X07000258

[24] K. M. Chandy and L. Lamport, "Distributed snapshots : Determining global states of distributed systems," *ACM Transactions on Computer Systems*, Feb. 1985.

[25] 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.

[26] X. Ouyang, S. Marcarelli, and D. Panda, "Enhancing checkpoint performance with staging io and ssd," in *Storage Network Architecture and Parallel I/Os (SNAPI), 2010 International Workshop on*, may 2010, pp. 13 –20.