Hiding Checkpoint Overhead in HPC Applications with a Semi-Blocking Algorithm

Xiang Ni, Esteban Meneses, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
E-mail: {xiangni2, emenese2, kale}@illinois.edu

***Abstract*—The HPC community has seen a steady increase in the number of components in every generation of supercomputers. Assembling a large number of components into a single cluster makes a machine more powerful, but also much more prone to failures. Therefore, fault tolerance has become a major concern in HPC. To deal with node crashes in large systems, checkpoint/restart is by far the preferred method. A typical way to implement checkpoints is by using a blocking algorithm, which suspends the execution of the application while the checkpoint is safely stored. One limitation of the blocking algorithm is that it saturates the network bandwidth at the time of checkpoint. This problem will become even more critical because the projected network bandwidth increase will not match the increase in memory per node. To alleviate this problem, we have developed a semi-blocking checkpoint algorithm that overlaps execution of the application with transmission of checkpoints. Our implementation decomposes a checkpoint into small messages that are interleaved with application messages. The experimental results show a dramatic reduction in the checkpoint overhead for various applications. We present a model for our approach and use this model to compute the benefit of the semi-blocking algorithm for different failure rates predicted at Exascale. We estimate our method can reduce up to 22% the total execution time of an iterative scientific application.**

***Keywords*-fault tolerance; checkpoint/restart; semi-blocking algorithm; adaptive runtime system; SSD**

I. Motivation

Current supercomputers assemble thousands of parts, from processor chips to routers and disks. Even when each individual component may be highly resilient, the net result of clustering too many parts into a single machine is an alarmingly low mean-time-between-failures (MTBF) of the system itself. Recent studies show supercomputers have a MTBF between 6 hours and several days [1], [2]. However, predictions for Exascale forecast MTBF values from 2 to 60 minutes [1], [3]. It will be hard for applications to make any progress in such circumstances without incorporating a fault tolerance mechanism.

In the HPC community, checkpoint/restart is the most popular fault tolerance technique. Its success comes from its simplicity: an application periodically saves its state and if one failure brings part of the system down, the application rolls back to the most recent checkpoint and restarts from there. Storing a global checkpoint in the file system (such as Lustre) is expensive and can be a bottleneck. One promising alternative is to use local storage (memory, SSD, local disks) [1], [2], [4]. During checkpoint, the application usually stops the execution until the checkpoint is safely stored, using what is called the *blocking* algorithm [5].

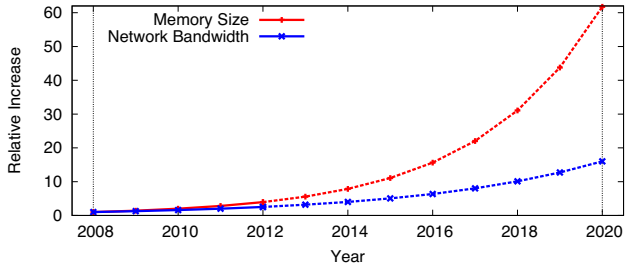


Fig. 1. Disparity between network bandwidth and memory size.

One drawback of the blocking algorithm is that it stalls the application while the checkpoint phase finishes. This delay accounts for most of the overhead of checkpointing. It is estimated that total checkpoint overhead goes beyond 10% of the total execution time [2]. In popular implementations of the blocking algorithm, each node an application is running on must send its checkpoint to another node across the network [1], [4]. At this point, the network usually gets saturated with the large amount of data it must transport before the application can make any progress. Further, a congested network transports data at a rate lower than the peak bandwidth may allow.

Recent studies [6], [7] estimate that the annual increase in memory size and network bandwidth is 41% and 26%, respectively. Figure 1 shows the trends in both memory size and network bandwidth for the period between 2008 and 2020. The disparity between the two curves aggravates the problem of a saturated network during checkpoint.

In this paper, we propose a solution for this problem. The key contributions of this paper are as follows:

- We introduce, in Section III, a semi-blocking checkpoint protocol that hides most of the checkpoint overhead. We also present a model to compute the optimal checkpoint interval for this protocol, which allows us to predict the benefit of our protocol for scenarios with different failure rates.
- We present an implementation of the semi-blocking checkpoint protocol in Section IV. It is based on the Charm++

978-0-7695-4807-4/12 $26.00 © 2012 IEEE
DOI 10.1109/CLUSTER.2012.82

runtime system and contains techniques to adaptively overlap checkpoint with application execution based on the communication and computation characteristic of the applications.
• We demonstrate that semi-blocking checkpoint protocol can hide the checkpoint overhead with real-world applications and improve the performance up to 22% compared to using blocking checkpoint protocol. Section V summarizes these results.

II. Background

Checkpoint/restart is widely used to provide fault tolerance in large-scale systems. Each node will save its state, forming a global checkpoint. If there is a failure on one node, all other nodes will rollback and restart from the previous checkpoint. This section presents two different algorithms to obtain a global coordinated checkpoint. These two methods are usually referred as *non-blocking* and *blocking*. We list the advantages and disadvantages of each. The traditional method to compute the frequency of checkpoints is also covered at the end of the section.

One way to obtain a global checkpoint of an application is to get a global snapshot of the different tasks that compose the application. A popular alternative is Chandy-Lamport algorithm [8], a non-blocking algorithm. This protocol does not require a global synchronization point in the application for the tasks to trigger a checkpoint. Instead, it works using a *checkpoint scheduler* to initiate a checkpoint by sending *marker messages* to every task. After receiving the marker message, a task stores its local state and sends a marker to every other task. Messages received through a channel after the local checkpoint has been taken but before having received the marker from the sender must be recorded. This non-blocking algorithm is totally asynchronous and runs in conjunction with the application. However, since it needs to store the in-flight messages as part of the checkpoint, it has a higher memory footprint and a non-trivial implementation [5].

The other way to implement global checkpointing is by using a blocking algorithm. In this case, once the checkpoint starts, the application must stop making progress until the checkpoint has finished. This method is totally synchronous. Usually, applications use global synchronization points to trigger the checkpoint in each task. This method does not require to store messages as part of the checkpoint, because the checkpoint is triggered at a synchronization point where there are no in-flight messages. Finding these synchronization points is not difficult in most scientific computing applications. Besides, these points may be carefully chosen by the programmer as those places where the state size of the application is minimal. Some runtime systems, like CHARM++ [9] provide a flexible scheme, where programmers decide what to store as part of the checkpoint of a task.

There are multiple options for where to store the checkpoints of the tasks. Traditionally, NFS disks have been the choice. However, more recently, local storage have been used to keep the checkpoints [1], [2], [4]. One of the earliest such schemes in HPC is the double in-memory checkpoint/restart approach [4], where each node X has a *buddy* node Y that will hold the checkpoint of X in main memory. Each node will store its checkpoint in its own memory too. If node X crashes, its buddy Y will provide it with its checkpoint. All other nodes except the crashed node X will get their checkpoint from their own memory.

Since the nodes will checkpoint with a certain frequency, a natural question to ask is how often they need to checkpoint. The overhead of checkpointing the whole application is also related to how frequently the application checkpoints. Daly [10] investigates the optimum checkpoint interval to minimize the application execution time. The total execution time (T) is divided into:

$$T = T_{solve} + T_{dump} + T_{rework} + T_{restart} \quad (1)$$

where T_{solve} is the uninterrupted time to solve the problem, T_{dump} is the time to perform the checkpoint, T_{rework} is the time to recover the lost work due to a failure and $T_{restart}$ stands for the time required to resume execution after a failure.

The more frequently an application checkpoints, the more time an application will spend dumping checkpoints. However, the application will experience less rework time when a failure happens. So, there is always a balance between the checkpoint dumping time, mean time to failure and rework time. In Daly's first-order model, the optimum checkpoint interval τ is given by the following formula:

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

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

III. Semi-Blocking Checkpoint

In this section we describe a semi-blocking checkpoint algorithm that strikes a balance between the blocking and the non-blocking protocols from the previous section. Like the non-blocking approach, it allows interleaving the checkpoint process with application execution. However, similar to the blocking protocol, it does not need to store in-flight messages as part of the checkpoint. To facilitate the presentation of the algorithm, we summarize the most important parameters in table I. With the exception of $Benefit$, the unit of all other parameters is seconds.

A. Algorithm

The semi-blocking algorithm is based on the double in-memory application initiated checkpoint/restart algorithm [4] mentioned in the previous section. Figure 2(a) presents a sketch of the double in-memory blocking algorithm. The diagram presents nodes 1 and 2, where α is the application's data living on node 1 and β is the application's data living on node 2. Nodes 1 and 2 are buddies of each other. Upon reaching a global synchronization point (labeled *barrier* in the figure), the checkpoint mechanism kicks in. Each node saves its own checkpoint in memory and sends its checkpoint to its buddy. After receiving the checkpoint from the buddy, each node will store it in main memory. The checkpoint phase

TABLE I
PARAMETERS OF CHECKPOINT MODEL

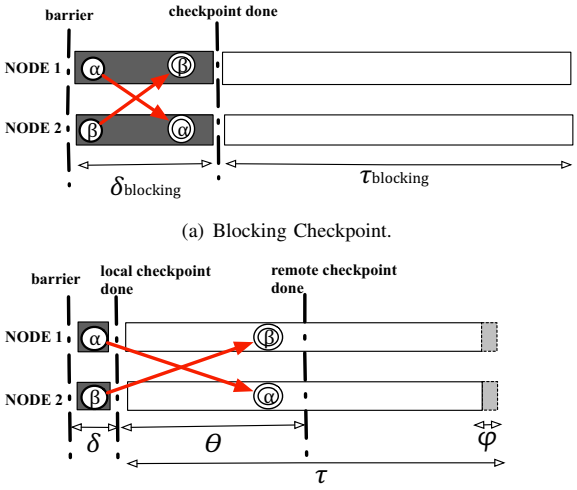
θ	Time to finish remote checkpoint
φ	Average interference of remote checkpoint to application
τ	Checkpoint interval (semi-blocking protocol)
$\tau_{blocking}$	Checkpoint interval (blocking protocol)
δ	Dump time of local checkpoint
$\delta_{blocking}$	Dump time of checkpoint (blocking protocol)
R	Restart time
M	Mean time between failures of the system
T_s	Workload of application
T	Total execution time (semi-blocking protocol)
$T_{blocking}$	Total execution time (blocking protocol)
$Benefit$	Performance improvement of the semi-blocking algorithm over the blocking algorithm

has a duration of $\delta_{blocking}$ and consists of two parts: saving their own checkpoint in local memory and saving their own checkpoint in the remote memory of the buddy. Clearly, it is the second part that consumes most of the time. That part involves almost no computation and, for applications with a non-negligible memory footprint, it may cause network congestion.

Figure 2(b) depicts the basic operations of the semi-blocking checkpoint protocol. The intuition behind the semi-blocking algorithm is to hide the second part of the checkpoint process, by interleaving the transmission of the checkpoints to remote memory with the execution of the application. The diagram shows that once the local checkpoint is saved in memory, the application resumes execution while the checkpoint traffic dribbles through slowly, preferably using the network when the application is not using it. Since the remote checkpoint runs in the background while the application continues executing, this method can substantially reduce the checkpoint overhead. With the semi-blocking algorithm, the checkpoint overhead can be reduced to the cost of just saving a local checkpoint if the checkpointing can perfectly overlap with application. This cost stands for a tiny percentage of the total cost of checkpointing in the blocking algorithm. We observe a 22% reduction in the total execution time of iterative scientific applications because of the low cost to do checkpoints as seen in Section V. Figure 3 gives the pseudo-code of the semi-blocking algorithm. In addition to storing the latest checkpoint, each node will also keep the checkpoint from the previous round to ensure that the application can recover from failures during remote checkpointing. We use the names *current* and *previous* to refer to the latest and older checkpoints, respectively.

The benefits of the semi-blocking algorithm are clear. It can potentially hide the checkpoint overhead by stalling the application only while the local checkpoint completes instead of waiting for the remote checkpoint to finish. Additionally, since checkpointing is faster, we can afford to checkpoint more frequently to reduce the amount of work lost in a failure. However, there are drawbacks to this algorithm that need to be addressed.

First, interleaving the remote checkpoint transmission with



(a) Blocking Checkpoint.

(b) Semi-Blocking Checkpoint.

Fig. 2. Checkpoint operation of blocking and semi-blocking algorithms.

Local Variables
localCkpt[2]← [NULL,NULL]
remoteCkpt[2]← [NULL,NULL]
Upon Global Synchronization
localCkpt[previous]← localCkpt[current]
localCkpt[current]← $Data_{checkpoint}$
reduction to mark the finish of local checkpoint
Upon Local Checkpoint Done
resume computation
send $Data_{checkpoint}$*to buddy node*
Upon Receiving $Data_{checkpoint}$ **from Buddy Node**
remoteCkpt[previous]← remoteCkpt[current]
remoteCkpt[current]← $Data_{checkpoint}$
reduction to mark the finish of remote checkpoint
Upon Remote Checkpoint Done
delete localCkpt[previous]
delete remoteCkpt[previous]

Fig. 3. Pseudocode of semi-blocking algorithm.

the application's messages may create interference and decrease the progress rate of the application. We explore an effective technique in Section IV to decompose the checkpoint messages into chunks and inject them into the network at the appropriate time. This mechanism reduces the impact of remote checkpointing on the execution of the application.

Second, if a failure brings down a node during the remote checkpoint phase, it will require the system to rollback to an older checkpoint instead of the most current one. Figure 4 shows the two types of failures the semi-blocking protocol has to handle. In *Failure 1*, between the completion of the remote checkpoint and the start of the new checkpoint phase, the system rolls back to the first checkpoint. The case of *Failure 2*, between the start of the checkpoint phase and the end of the remote checkpoint, still requires the system to rollback to the first checkpoint. The reason is that a global state is not available until remote checkpoint is completed. The rest of this

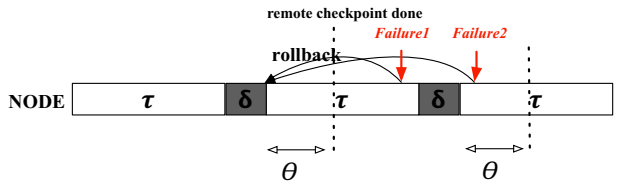


Fig. 4. Rollback operation of the semi-blocking algorithm.

section presents a model that determines how these two types of failures affect the reliability of the system under different failure rates.

B. Model

In order to estimate the benefit of the semi-blocking checkpoint algorithm, we present a model that includes all the fundamental parameters. Two of them deserve special consideration. We use θ to denote the time in seconds to finish a remote checkpoint. Since the remote checkpoint runs concurrently with the application, it must hold that $\delta + \theta \geq \delta_{blocking}$. As we saw above, the larger θ is, the higher the chance a failure will require the system to rollback to an older checkpoint. To model the interference remote checkpoint may have on the execution of the application we use φ. It denotes the extra time in seconds per each checkpoint interval an application requires due to interference. The lower φ is, the better the semi-blocking algorithm is able to hide the checkpoint overhead.

The total time of a checkpointed workload with failures is divided into five parts:

$$T = T_s + T_{local} + T_{overhead} + T_{rework} + T_{restart} \quad (3)$$

where T_s is the pure computation time of the application without any checkpoints or rollbacks, T_{local} is the total time to dump local checkpoints in memory or local disk, $T_{overhead}$ is the total interference of remote checkpoint to applications, T_{rework} stands for the wall clock time required to bring the application back to the point it was right before the crash and $T_{restart}$ is the time to restart applications from checkpoint.

The dumping time of local checkpoints 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 $\tau - \varphi$, thus,

$$T_{local} = \frac{T_s}{\tau - \varphi}\delta \quad (4)$$

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

$$T_{overhead} = \frac{T_s}{\tau - \varphi}\varphi \quad (5)$$

The dotted lines in Figure 4 mark the finish of remote checkpoint. Between them is one checkpoint interval τ plus the local checkpoint dumping time δ. Failures that happen during that period will rollback to the first checkpoint. The overlap period θ consists of the useful application work and the interference of the remote checkpoint φ. The useful application work of $\theta - \varphi$ is not checkpointed in the first checkpoint. So, rework time would be at least $\theta - \varphi$ if it happens right after the first remote checkpoint is done but at most $\theta - \varphi + \tau + \delta$ if it happens right before the second remote checkpoint is done. On the average, failures will occur half through the checkpoint interval. Thus,

$$T_{rework} = \frac{T}{M}\left(\frac{\tau + \delta}{2} + \theta - \varphi\right) \quad (6)$$

$T_{restart}$ depends on the number of failures and the restart time R, so

$$T_{restart} = \frac{T}{M}R \quad (7)$$

Hence, the total execution time including checkpoint and restart time for a workload of T_s is

$$T = T_s + \frac{T_s}{\tau - \varphi}\delta + \frac{T_s}{\tau - \varphi}\varphi + \frac{T}{M}\left(R + \frac{\tau + \delta}{2} + \theta - \varphi\right) \quad (8)$$

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

$$T_{blocking} = T_s + \frac{T_s}{\tau_{blocking}}\delta_{blocking} + \frac{T_{blocking}}{M}\left(R + \frac{\tau_{blocking} + \delta_{blocking}}{2}\right) \quad (9)$$

In order to minimize the equations 8 and 9 to find the optimum checkpoint interval for these two protocols we may use a standard numerical optimization technique. The checkpoint interval τ has strict bounds in both semi-blocking and blocking model, respectively, $\theta < \tau < M$ and $0 < \tau_{blocking} < M$.

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

$$Benefit = \frac{T_{blocking} - T}{T_{blocking}} \quad (10)$$

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

IV. Implementation

In this section, we describe the implementation of the semi-blocking checkpoint protocol. We show a technique to overlap the transmission of the remote checkpoint with the execution of the application. Two schemes to ensure that remote checkpointing is finished in a timely manner are also discussed. Then we discuss how to use solid state disk to reduce memory pressure.

A. A runtime system for multicore clusters

We implemented the semi-blocking algorithm in the Charm++ [9] runtime system, which is based on a message-driven programming model. Applications are divided into fine-grain tasks using this model and tasks perform computation and communication through asynchronous method invocation associated with each message. Charm++ provides an

SMP extension for machines with multicore nodes. The SMP version of CHARM++ creates multiple *worker threads* per node. Additionally, each node has a dedicated *communication thread* to handle all the inter-node communication while cores on the same node communicate via shared memory. The worker threads do not need to pay for the communication overhead themselves. The communication thread is bound to a certain core in the node. When a worker thread sends a network message, it enqueues the message into the outgoing message queue of the communication thread. Using the SMP mode of CHARM++, we can achieve faster startup, reduction in memory consumption and optimized node-level collective communication operations.

B. Overlapping remote checkpointing and computation

The success of semi-blocking checkpoint depends on the ability to overlap remote checkpoint transmission with the execution of the application. During remote checkpointing, each node sends checkpoint messages to its buddy; those messages are enqueued in the outgoing message queue right after local checkpoint is finished. Thus, the transmission of application messages may be delayed by the transmission of checkpoint messages. To solve this problem, we designed a separate checkpoint message queue on each communication thread. Worker threads enqueue the checkpoint messages to this separate queue. The communication thread sends checkpoint messages only when there is no application message ready to be sent, to reduce the interference of the checkpoint message to applications. We call this strategy the *opportunistic* sending of checkpoint messages. Each checkpoint message is split into multiple small chunks for transmission to better overlap with the application execution.

TABLE II
OVERLAP AND INTERFERENCE ASSOCIATED WITH DIFFERENT COMMUNICATION TO COMPUTATION RATIOS

Case	Comm-Comp Ratio	θ	φ
1	0.2	6.6	0.6
2	1.6	13.5	0.8
3	3.1	17.3	1.5

Interference of the remote checkpoint to application is heavily affected by the communication profile of applications. We use a synthetic benchmark called FT_Test to analyze the sensitivity of the semi-blocking protocol to different conditions. FT_Test is a program that simulates a stencil computation. With FT_Test, it is possible to change the computation time per step, the message size, the communication topology of the objects and the checkpoint size. Table II presents 3 different levels of communication-computation ratios for FT_Test by changing the communication topology of the objects. We consider the time not spent in computation as communication time. We use the same checkpoint size of 512MB per node for the different communication-to-computation ratios. This amount of data requires 6.5s to checkpoint using the blocking protocol. As seen in the table high communication

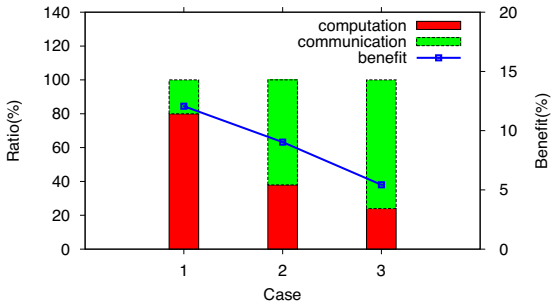


Fig. 5. Effect of communication to computation ratio.

to computation ratio will increase both the overlap period θ and the interference of checkpointing to applications φ using the semi-blocking algorithm. Figure 5 shows the benefit of the semi-blocking protocol for these 3 communication-to-computation ratios. As expected, the benefit is reduced when the communication-to-computation ratio of applications increases because of the increment in θ and φ . Even with a communication-to-computation ratio as high as 3.1, the semi-blocking protocol gives us more than 5% benefit compared to the blocking checkpoint protocol with an M value of 300s (M is the MTBF of the system).

C. Opportunistic vs. random scheduling

The semi-blocking protocol can achieve the most benefit with the smallest overlap period θ and the smallest interference φ. For a given application of certain communication to computation ratio, reducing θ can help reduce the rework time when a failure happens. On the other hand, increasing θ may reduce interference with the application. Finding a good overlap period is critical to the success of the semi-blocking protocol.

Opportunistic sending of checkpoint messages will only give us a fixed overlap period. However, we decided to test if this approach brings the most benefit of the semi-blocking algorithm. We implemented *lottery scheduling* [11] 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. Each time the resource is granted to the client with the winning ticket. The allocations of the clients to access the shared resource are represented by the number of lottery tickets they hold. In the semi-blocking algorithm, transmission of remote checkpoint messages and application messages share the same NIC. By controlling their allocations to use the NIC, we can achieve 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 work that has been finished. The longer the overlap period is, the more application messages will be sent during the remote 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 period θ, the number of application

messages sent during remote checkpoint is approximately $\frac{\theta}{\tau}s$. The number of application and checkpoint messages sent during remote checkpoint can be used as their lottery tickets. So the probability to send an application message during remote checkpoint is $\frac{\theta s}{\theta s+\tau c}$, while the probability to send a checkpoint message is $\frac{\tau c}{\theta s+\tau c}$.

Remote Checkpoint Done:
  $appRatio \leftarrow \frac{\theta s}{\theta s+\tau c}$
  $ckptRatio \leftarrow 1 - appRatio$
Send Network Message:
  $choose \leftarrow rand()\%100$
  if $choose \leq ckptRatio * 100$ **then**
    Send Checkpoint Message
  else
    Send Application Message
  end if

Fig. 6. Using lottery scheduling to control overlap period.

The pseudo-code of using lottery scheduling to control the overlap period is shown in Figure 6. Each time, before sending a network message, a random number generator is used to randomly select a winning ticket. Then we locate whether the application or checkpoint message is holding that ticket: the one who gets the chance to be sent this time. As seen in Figure 7 for a FT_Test with communication/computation ratio set to 1.6, the interference of opportunistic sending of checkpoint message is the minimum. Increasing the overlap period will increase the overhead slightly while decreasing the overlap period will make the interference to increase dramatically. Increasing or decreasing the overlap period both fail to bring us a higher benefit using the model in section III. So, opportunistic sending of the checkpoint message is better than lottery scheduling for the semi-blocking algorithm, empirically. Even though a reduced overlap period θ can reduce the rework time, it cannot offset the interference gained with the reduced θ.

D. Relieving memory pressure of checkpoint with SSD

The local and remote checkpoints constitute a memory overhead for this semi-blocking protocol, but (somewhat surprisingly) this is tolerable for a large class of applications that have a smaller memory footprint at checkpoint. These include molecular dynamic, N-body codes, certain quantum chemistry (nanomaterials codes), etc. Even for applications requesting large memory footprint, we leverage the use of SSD to relieve the memory pressure.

Solid state disk (SSD) is becoming more and more promising to store the large amount of checkpoint data because of its good random access performance and low power consumptions. Considering the SSD bandwidth is not comparable to memory bandwidth nowadays, we need to carefully select what checkpoint data is to be stored in SSD. We propose two strategies to checkpoint the data on SSD.

1) Full SSD Strategy. All the checkpoints will be saved to SSD which can fully relieve the memory pressure.

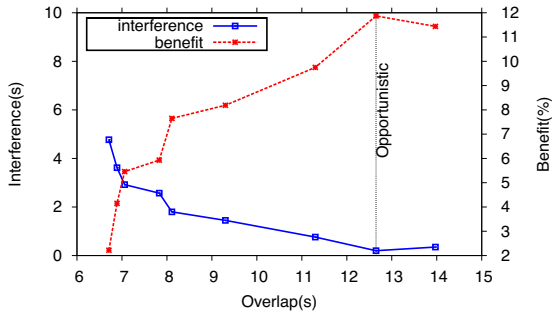


Fig. 7. Interference and benefit of different overlap periods.

2) Half SSD Strategy. we reduce the writes to SSD by only storing the buddy's 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 letting each core stall to writing to SSD, a dedicated IO thread is used on each node for asynchronous access to SSD. Each worker thread enqueues its IO requests to the queue on the IO thread and gets a notification when the IO request is completed by the IO thread. Access to SSD can thus be adaptively overlapped with useful computation.

In section V we will show the performance differences of using these two strategies for checkpoint and restart and the performance gained with asynchronous access to SSD.

V. Experiments and Analysis

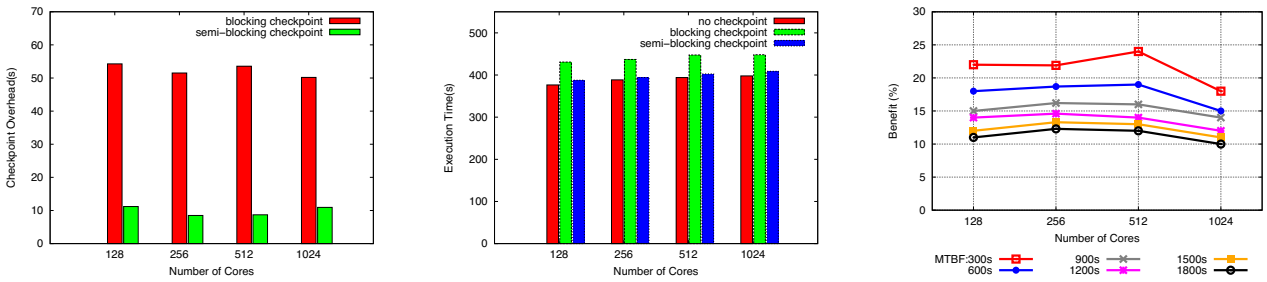
This section presents an evaluation of the semi-blocking checkpoint algorithm with different applications. We also include some experiments of the performance of restarting applications after a failure.

Two applications are used in the experiments. The first one is wave2D, which uses a finite differencing scheme to calculate pressure information over a discretized 2D grid. The second application, ChaNGa, is for N-Body based parallel simulations and is used in cosmology and astronomy [12].

The experiments were performed on Trestles supercomputer 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 64GB of DDR3 RAM and 120GB of flash memory(SSD).

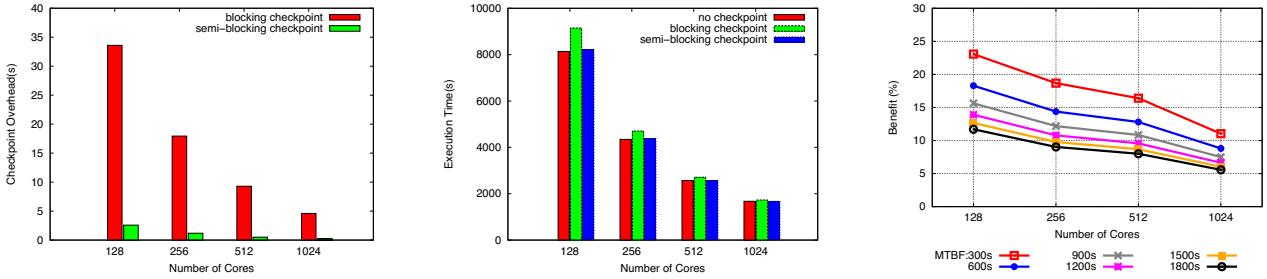
A. Scalability

Figure 8(a) shows the overhead of one checkpoint based on a weak scaling test with wave2D using blocking and semi-blocking checkpoint protocol from 128 cores to 1K cores. The checkpoint size is 4GB per node. Semi-blocking algorithm reduces the checkpoint overhead from 52s to 10s. The optimal checkpoint interval of the blocking algorithm is 372s given an M value of 1800s (M is the MTBF of the system) and $\delta_{blocking}$ of 52s. This requires the wave2D application to checkpoint every 960 iterations. In Figure 8(b), we show the checkpoint overhead to the execution of the applications using the two algorithms. The checkpoint overhead is reduced from



(a) Single Checkpoint (b) Checkpoint Overhead (c) Semi-Blocking Benefit

Fig. 8. Weak scaling results - wave2D.



(a) Single Checkpoint (b) Checkpoint Overhead (c) Semi-Blocking Benefit

Fig. 9. Strong scaling results - ChaNGa.

14% to 3% by using the semi-blocking algorithm. With the decreasing checkpoint overhead, the semi-blocking algorithm can afford to checkpoint more frequently to reduce the amount of rework time. So, the optimum checkpoint interval of the semi-blocking algorithm is different from that of the blocking algorithm.

Then, we compare the actual benefit of the semi-blocking algorithm compared to the blocking algorithm using our model and considering both the checkpoint and rollback-recover overhead. As seen in Figure 8(c), for different M values, the benefit of the semi-blocking checkpoint protocol is mostly constant from 128 cores to 1K cores. With the decrease of M, the checkpoint and restart overhead for the blocking checkpoint protocol will keep increasing, while on the other hand the semi-blocking protocol shows more benefit. So benefit of the semi-blocking protocol goes from 10% for M of 1800s to 22% for M of 300s for checkpoint size of 4GB/node.

ChaNGa is used to demonstrate the strong scalability of the semi-blocking checkpoint algorithm. We use a 100 million particle system. In one big step of ChaNGa, it first does domain decomposition of the particle space, then builds the Barnes-Hut trees, computes the gravitational forces, and finally updates the particles. Checkpoint is taken periodically after a certain number of steps. Figure 10 displays the view of communication bytes sent over time from our PROJECTIONS performance analysis tool. There is less amount of communication data in the first two phases of one big step: domain decomposition and tree building as seen in the figure. Sending more checkpoint messages in these phases can help us incur less interference to the application. With the opportunistic sending of the checkpoint message, our scheme can identify

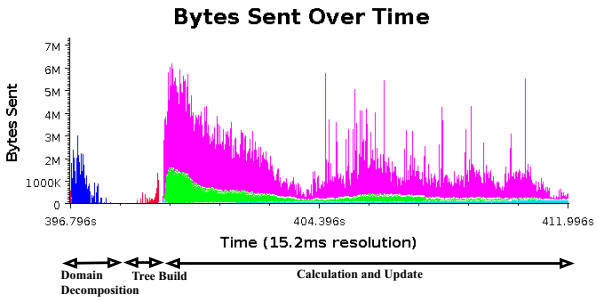


Fig. 10. Bytes sent in one step of ChaNGa.

such phases without application knowledge.

Figure 9(a) shows the checkpoint overhead of one checkpoint based on a strong scaling test of ChaNGa application using the blocking and semi-blocking algorithms separately. Checkpoint size per node is decreasing for a strong scaling test, so the blocking checkpoint time is reduced from 33s on 128 cores to 5s on 1K cores. The semi-blocking checkpoint time decreases from 2.6s to 0.27s, almost hiding the checkpoint overhead.

In Figure 9(b), we display the checkpoint overhead of the execution of ChaNGa. The optimum checkpoint interval is adjusted to the blocking checkpoint time. The blocking checkpoint overhead decreases from 12% on 128 cores to 5% on 1K cores because of the decreased checkpoint size per node. With the semi-blocking algorithm, the checkpoint overhead is below 1%. Of course with such low checkpoint overhead of the semi-blocking algorithm, applications can benefit more from frequent checkpoints so as not to lose lots

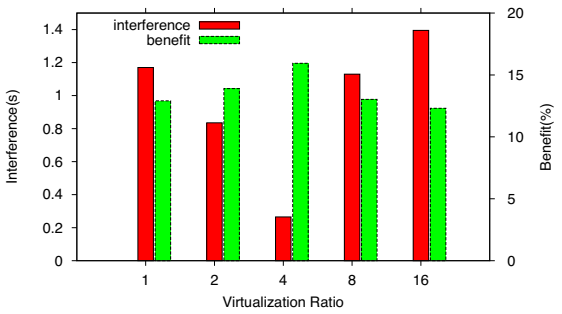


Fig. 11. Effect of virtualization.

of wall clock cycles when failure happens.

Figure 9(c) depicts the percentage benefit of the semi-blocking algorithm to the blocking algorithm at their own optimal checkpoint intervals. Semi-blocking algorithm achieves the largest benefit on 128 cores where the blocking checkpoint overhead is at its maximum in a strong scaling experiment. Even running on 1K cores with M of 1800s, the semi-blocking algorithm still has over 6% benefit compared to the blocking algorithm. Given that the memory consumption is only 763MB per node when running on 1K cores, which is 1% of the total memory, we can expect more benefit of the semi-blocking algorithm for applications with larger memory consumption.

B. 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*. With high virtualization ratio, the communication of the checkpoint or application message of one chare can be overlapped with the computation of other chares on the same core which can further hide the checkpoint overhead, so we can expect more benefits.

We use the wave2D benchmark to illustrate the benefit of high virtualization ratio for the semi-blocking algorithm. The checkpoint size is 0.9GB per node. The interference of remote checkpoint per checkpoint interval decreases from 1.4s with 1 chare per core to 0.3s with 4 chares per core in Figure 11. Correspondingly, the benefit of the semi-blocking protocol to the blocking version calculated from the model is increased from 12.3% to 15.9% when M is 300s as expected. However, when the virtualization ratio increases to 8 and beyond, there is extra overhead to schedule the work of multiple chares, so there is more interference.

C. Checkpoint and restart with SSD

As discussed in section IV, half SSD and full SSD schemes can be used depending on the memory consumption of applications.

Figure 12 shows the checkpoint timing penalty using SSD with checkpoint data size ranging from 0.45GB to 2.23GB per node for the wave2D benchmarks. We compare the performance of half and full SSD scheme with asynchronous (half-aio, full-aio) and synchronous IO access (half-sio, full-sio) respectively. Using full SSD scheme with asynchronous IO access saves us more than half the time of writing checkpoint data to SSD with synchronous IO. In Figure 13, we show the restart time of in-memory checkpointing, half and full SSD checkpointing with asynchronous IO access. Half SSD scheme has almost negligible overhead compared to the in-memory checkpointing, while full SSD scheme has around 1s overhead. During restart, objects first need to get their checkpoints either from their own or their buddy node's local disk or memory, and then restore the application and process data from the checkpoints. With asynchronous IO access and high virtualization ratio, the restoring of one object can be overlapped with the process of getting checkpoints for another object. Thus we see the restart time is not much affected by checkpointing to SSD.

VI. RELATED WORK

There are two main checkpointing methods in HPC: uncoordinated checkpointing and coordinated 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 no synchronization is required to initiate checkpointing. However, uncoordinated checkpointing is susceptible to rollback propagation, the domino effect 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. [13] proposed an uncoordinated checkpointing without domino effect by 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. BLCR [14] implements kernel-level checkpointing, but incurs excessive overhead for application at production level. Some multi-level approaches have been proposed recently to deal with failures at different frequencies of occurrence. FTI [2] is a multi-level coordinated checkpoint scheme using topology-aware RS encoding with about 8% checkpoint overhead. Moody et al. [1] propose a multi-level checkpoint scheme that is able to store the checkpoint in different places. Each place represents a different level and uses a Markov probability model to decide the checkpoint frequency of each level. The semi-blocking algorithm can be used instead of the blocking algorithm in any of the levels of the multi-level approach.

Kai Li et al. implemented the concurrent checkpointing [15] for shared-memory multiprocessors using a forked process and buffers to overlap the writing of checkpoints to disk with the copying of checkpoints from memory. Our approach differentiates from theirs in that we provide techniques to reduce the interference of checkpoint for distributed memory clusters.

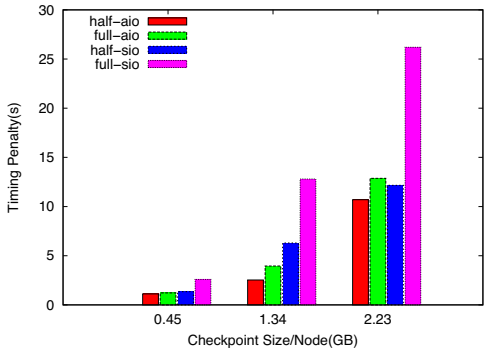


Fig. 12. Penalty of checkpointing to SSD.

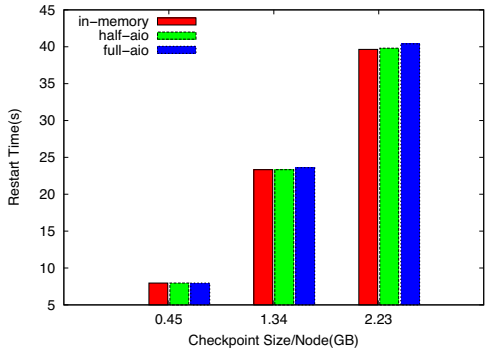


Fig. 13. Low overhead of restarting from SSD.

Dong et al. leverage PCRAM [7] for checkpointing and propose the hybrid local/global checkpointing mechanism. Their approach can be incorporated with the semi-blocking algorithm by relaxing the stall of computation when taking global checkpoint. Ouyang et al. [16] enhance the performance of checkpointing to SSD with aggregation of checkpoint buffer and staging IO. Our approach is different from theirs in that checkpoints are distributed among the SSD disks rather than stored in a central checkpoint server.

VII. Conclusion and Future Work

This paper presents a semi-blocking checkpoint algorithm to provide low-overhead resilience for HPC applications. Our algorithm succeeds in hiding checkpoint overhead by overlapping checkpoint transmission with execution of the application. We provide a model for the algorithm to find out the optimum checkpoint interval and to compute the expected benefit of the algorithm under various scenarios.

The experimental results are very encouraging. Both the strong and weak scalability of the semi-blocking algorithm are demonstrated with different applications. On 1024 cores, the semi-blocking algorithm reduces the checkpoint overhead from $50s$ to $10s$ for a stencil computation. A strong scaling test using a cosmology application shows the checkpoint overhead is almost negligible. Using our model and a range of different failure rates predicted at Exascale, we show the semi-blocking checkpoint algorithm may reduce 22% of the total execution time compared to the traditional blocking checkpoint algorithm.

For future work, we plan to use direct memory access (RDMA) to transmit checkpoint messages to further reduce the interference of our algorithm.

Acknowledgments

This work was partially supported by the US Department of Energy under grant DOE DE-SC0001845, by the National Science Foundation under grant NSF OCI-0725070 and by a machine allocation on XSEDE under award ASC050039N.

References

[1] 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.

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

[3] P. Kogge, 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, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[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] D. Buntinas, C. Coti, T. Hérault, 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 Comp. Syst.*, vol. 24, no. 1, pp. 73–84, 2008.

[6] S. L. Graham, M. Snir, and C. A. Patterson, Eds., *Getting Up to Speed, The Future of Supercomputing*. The National Academies Press, 2006.

[7] X. Dong, N. Muralimanohar, N. P. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems," in *SC*, 2009.

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

[9] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[10] 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.

[11] 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

[12] 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.

[13] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *IPDPS*, 2011, pp. 989–1000.

[14] 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.

[15] K. Li, J. Naughton, and J. Plank, "Low-latency, concurrent checkpointing for parallel programs," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 8, pp. 874 –879, aug 1994.

[16] 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.