A Fault Tolerance Protocol with Fast Fault Recovery

Sayantan Chakravorty Laxmikant V. Kalé
Department of Computer Science, University of Illinois at Urbana-Champaign
{schkrvrt,kale}@uiuc.edu

Abstract

Large machines with tens or even hundreds of thousands of processors are currently in use. Fault tolerance is an important issue for these and the even larger machines of the future. Checkpoint based methods, currently used on most machines, rollback all processors to previous checkpoints after a crash. This wastes a significant amount of computation as all processors have to redo all the computation from that checkpoint onwards. In addition, recovery-time in checkpoint based fault tolerance protocols is bound by the time between the last checkpoint and the crash. Protocols based on message logging avoid the problem of rolling back all processors to their earlier state. However, the recovery time of existing message logging protocols is no smaller than the time between the last checkpoint and crash. We present a fault tolerance protocol, in this paper, that provides fast restarts by using the ideas of message logging and processor virtualization. We evaluate our implementation of the protocol in the Charm++/Adaptive MPI runtime system. We show that our protocol not only provides fast restarts but also has low fault-free overhead for many applications.

1 Introduction

Massively parallel systems with tens of thousands and even hundreds of thousands of processors, such as ASCI-Purple, Red Storm and Bluegene/L, are being used for scientific computation. More powerful machines with more processors are being planned and designed. Machines with large numbers of components are likely to suffer from partial failures frequently. ASCI-Q is reported to suffer a failure every few hours [1]. Therefore any application running on machines with thousands of processors for an appreciable length of time will have to be able to tolerate faults. Traditional checkpoint and restart systems roll back all processors in an application, when a single processor crashes. This not only wastes computing time but also slows down the progress of the application in the presence of frequent faults. Even current fault tolerant protocols that do not roll back all processors ([1, 2]) redo all the computation of the crashed processor on a single processor. As a result the recovery time of all protocols are bound by the time interval between the crash and the previous checkpoint.

We present the design and implementation of a protocol for fault tolerant computation in this paper. We combine sender side message logging and object based virtualization to build a system that has low overhead during normal execution and allows fast restarts when recovering from a crash. We do not assume the existence of any "fully reliable or stable" component that never fails (assumed by other researchers), since we think that it is difficult to realize such an assumption in real life. Our scheme has many advantages compared with the traditional checkpoint/restart scheme. First, only the work of the failed processor is re-executed. Second, object based virtualization [3] allows us to distribute the work of this failed processor among the other processors (especially those that are waiting for data from the failed processor). This speeds up the restart procedure, making the recovery time considerably lower than the time interval between the last checkpoint and the crash. This would not have been possible if all processors had rolled back to their previous checkpoint as in traditional checkpointing based protocols. With our scheme, an application can (potentially) make progress even when the mean time between failure (MTBF) is lower than the checkpoint period. On a large machine, processors that are not dependent on data from the failed processor can continue to execute further during crash recovery. Processors that are dependent but do not receive messages from the failed processor can be idle during recovery (thus reducing power consumption and network contention). In principle, the system can also use these processors to execute low-priority background jobs. Compared with other sender-side-message-logging protocols, our scheme distinguishes itself by its ability to speedup recovery by parallelizing it using object-based virtualization.

Our scheme has been implemented for a version of MPI, called Adaptive MPI[4], and so can be used by all MPI programs. Applications written in Charm++[5], which is the underlying layer of Adaptive MPI, can use our scheme as well. In this paper we describe the scheme, and demon-

strate its performance on several benchmarks. The overheads are analyzed and shown to be reasonable for many realistic applications. Clearly, the advantage of our scheme over regular checkpointing is strongest in a large-system regime where faults are more frequent than today. However, a solution to this problem is both an interesting and challenging academic research issue in its own right as well as a practically essential strategy, if the predictions about future system sizes and reliabilities are realized.

2 Related Work

The solution space for fault tolerance protocols can be divided into two main categories: checkpoint based and log based recovery protocols as described in [6]. Checkpoint based protocols periodically save the state of a computation to stable storage. After a crash, the computation is restarted from a previously saved state. Checkpoint based protocols can be divided into three types: uncoordinated, coordinated and communication induced. Uncoordinated checkpointing methods, which allow each processor to take its checkpoint independent of the other processors, are fast and memory efficient[7]. However, they suffer from the fatal flaw of cascading rollbacks. In coordinated checkpointing schemes, all the processors in a computation coordinate to save a globally consistent state. Such schemes are used by CoCheck [8], Starfish [9], Clip [10] and AMPI [11, 12] to provide fault tolerant versions of MPI. A non-blocking coordinated checkpointing algorithm that uses application level checkpointing is presented in [13]. Communication induced checkpoint protocols try to combine the advantages of coordinated and uncoordinated by allowing processors to take a mix of independent and coordinated checkpoints[14]. However it was found that communication induced checkpoint methods did not scale well to large number of processors [15].

The second category of fault tolerance protocols depend on stored message logs for recovery. After a processor crashes, all the messages are resent to the crashed processor and reprocessed in the same order as before the crash. This brings the restarted processor to its exact state before the crash, according to the piecewise deterministic (PWD) assumption [16]. Message logging can be divided into three classes based on the frequency with which the message log is saved to stable storage: pessimistic, optimistic and causal. Pessimistic log based protocols save each message to stable storage before processing it. Restart and garbage collection of old logs are very simple. On the other hand they increase the message latency by saving each message to stable storage before processing it. The overhead can be reduced by using specialized hardware [17] or by storing the message log in the sender's memory [18]. MPICH-V1 and V2 [1, 2] are systems that provide fault tolerant versions of MPI by using pessimistic log based methods. Optimistic protocols save the message logs temporarily in volatile storage before sending them all to stable storage [16]. Though optimistic schemes have a lower message latency overhead than pessimistic ones, they are susceptible to cascading rollbacks. Moreover, garbage collection and recovery are more complicated. Causal logging stores message logs temporarily in volatile storage but prevents cascading rollbacks by tracking the causality relationships between messages [19]. Tracking causality and recovering from faults are complicated operations in causal message logging protocols. Manetho[19], MPICH-Vcausal [20] and the protocol in [21] are examples of causal logging systems. A discussion about MPI and its relation to fault tolerance can be found in [22].

Object based virtualization[3], used in Charm++ [5] and Adaptive-MPI(AMPI)[4], encourages the user to view his computation as a large number of interacting objects. These objects are also referred to as virtual processors. The user decomposes his computation into virtual processors without caring about the number of physical processors available. The Charm++ runtime system is responsible for mapping the virtual processors to physical processors, and can change this mapping during the execution of a program. Virtual processors interact with each other by sending messages that are delivered by the runtime system without the user needing to know about the receiver's physical location [23]. The Charm++ runtime system can perform measurement based dynamic load balancing and communication optimizations. The Charm++ runtime system supports multiple checkpoint based fault tolerance protocols[12, 11] and a basic message logging based protocol [24].

3 Protocols

Our fault tolerance protocol is entirely software based and doesn't depend on any specialized hardware. It however makes the following assumptions about the hardware. i) The processors in the system are fail-stop [25]. This means that when a processor crashes it remains halted and other processors may detect its crash. ii) All communication between processes is through messages over the network. iii) The PWD assumption should hold. It is assumed that the only non-deterministic events affecting a processor are message receives.

We bring together the ideas of sender side message logging and object based virtualization to develop a fault tolerance protocol that provides fast recovery from faults. Virtualization affords us a number of potential benefits with respect to our message logging protocol. First, it makes applications more latency tolerant. This helps us hide the increased latency due to the sender side message logging protocol. It is also the primary idea behind faster restarts since it allows us to spread the work of the restarting processor among other processors. We treat the virtual processors, and not the physical processors, as the communicating

entities that send and receive messages. Since an object's state is modified only by the messages it receives, we can apply the PWD assumption to virtual processors instead of physical processors. After a crash, if a virtual processor re-executes messages in the same sequence as before, it can recover its exact pre-crash state.

Our protocol has three major components: message logging, checkpointing, and restart. Although all three components are very closely related, we describe them as separate protocols for the sake of clarity. As we shall see, processor virtualization has a significant impact on all components. We discuss the protocol for single faults in the first three subsections. Next, we extend the protocol to deal with multiple faults. Finally, we describe the fast restart protocol in the last subsection.

3.1 Message Logging Protocol

We design the message logging protocol such that, after a crash, a Charm++ object processes the same messages in the same order as before the crash. We also make sure that a Charm++ object does not reprocess a message that it has already processed. As described below, we achieve this by associating a sequence number and a ticket number with each message. Each Charm++ object is given a unique *id*. Every object maintains a table called the *SNTable* that keeps track of the number of messages sent to different objects. The SNTable is used to assign *sequence numbers*(*SN*) to messages. Each message sent by an object is stored in the object's *message log*. The receiver of a message assigns it a *ticket number*(*TN*) and processes messages in increasing order of TNs. An object stores the highest TN processed by it as *TNProcessed*. An object stores the highest TN assigned by it as *TNCount*. An object stores the sender's id, SN and TN for each message received since the last checkpoint in a table called the *TNTable*.

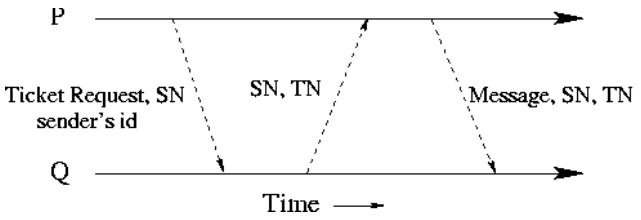


Figure 1. Messages in the remote mode of the message logging protocol

When the sender(P) and receiver(Q) objects are on different processors, the message logging protocol is said to operate in the *remote* mode. As seen in Figure 1, the sender P sends a request for a ticket, consisting of P's id and the message's SN, to Q. On receiving the request, Q looks up the sender id and the SN in the TNTable. If Q finds that it has already assigned a TN to this SN from P, it replies with that TN. If the TN has already been processed by Q, the returned TN is marked as old. (Both of these may happen after a restart.) If there is no matching entry in the TNTable, the receiver increments TNCount and returns this value as the TN. It also adds an entry for the tuple P, SN and TN to the TNTable. When P receives a TN in reply, it assigns the TN to the message stored in its log. If the received TN is not marked as old it sends the message containing the TN to Q. The time between the sender starting to send a message and the receiver sending a message of its own as a result of processing the sender's message is increased by the the round trip time of a short message. This overhead is the same as in the sender side message logging protocols of [18, 2].

If we were to use the above protocol for messages between two objects on the same processor, the log of a message and its receiver would exist on the same processor. If this processor crashes, it will become impossible to re-execute the messages in the correct sequence. Therefore, we define a *local* mode of the message logging protocol to deal with this case. This mode logs the sender id, receiver id, SN and TN for a message between objects on the same processor in the *remote message log* maintained on the *buddy* processor. The sender fetches the TN from the receiver through a method invocation and then sends the log to the buddy processor. The sender sends the message to the receiver only after receiving an acknowledgment from the buddy that it has logged the message in the remote message log. As a result, the latency for a message to a local object becomes the same as that of a message to a remote object.

3.2 Checkpoint Protocol

The checkpoint of a processor can be stored on the global file system, in the memory, or on local disk of a remote processor. The storage location does not really affect the rest of the protocol. In this paper, we chose to implement an in-memory checkpoint. Storing a checkpoint in the memory of a remote processor is much faster than storing it in a remote storage server [12], as long as adequate memory is available. As the message logging protocol already requires that each processor have a buddy, storing the checkpoint on this same processor simplified the implementation.

The state of a Charm++ object consists of user data, a small amount of runtime system data, as well as TNCount, TNProcessed, SNTable and the messages in the message log that were sent to objects on the same processor (the reason for this is explained in Section 3.3). In our design it is possible for objects on a processor to take their checkpoints independent of each other. However, by checkpointing all the objects on a processor at the same time, we can aggregate checkpoint data to reduce overhead. The checkpoint protocol also provides a mechanism to perform garbage collection on the message logs.

A processor, say A, packs up the state of all the objects on it and sends it to its buddy processor, say B. Each object

on A also stores its TNProcessed at the time of checkpoint as *TNCheckpointed*. B stores the new copy of A's checkpoint, deletes the old copy and sends an acknowledgment to A. On receiving the acknowledgment, the TNTable of each object on A can garbage collect entries with TN less than TNCheckpointed. Each object on A sends out garbage collection messages containing TNCheckpointed to all objects that had sent it messages since its previous checkpoint. When an object Y receives a garbage collection message from object X on processor A, it removes all messages to X in its message log that have a TN lower than the TNCheckpointed. A similar garbage collection message is sent to processor B, so B can remove old entries from the remote message log. Garbage collection is done lazily so that it interferes as little as possible with the application.

We have to deal with an interesting trade-off between memory and speed while deciding when to checkpoint. If the checkpoint period is too low, the message logs on senders are garbage collected frequently. This saves memory but increases the time cost because of frequent checkpoints. If the period is too high, the message logs on senders become large though the checkpointing cost is lower. The rate of expected failure is also an important factor in deciding the checkpoint period. Checkpoints might also be performed when the message logs become larger than a particular size.

Storing the checkpoint in memory is not a problem for applications with a small checkpoint state such as molecular dynamics. However, if the application is memory intensive the checkpoint can be stored in the local disk of the buddy processor. If there are no local disks in the system, the checkpoint can be stored on the cluster's file system. Even message logs can be lazily moved to local disk or the file system to keep the memory overhead low. Of course, moving checkpoints and message logs to disks from memory will slow down restart.

3.3 Restart Protocol

We assume that a pool of spare processors is available to the parallel job. When the crash detector finds out that a processor, say C, has crashed, it restarts a Charm++ process on a spare processor. Then the new Charm++ process corresponding to C requests its buddy processor D for its previous checkpoint and the remote message log. C recreates all the objects that used to exist on it from the checkpoint data. C then broadcasts to all processors, including itself, a request to resend the necessary logged messages.

When a processor receives a request to resend logged messages, each object resident on it looks in its message log for messages sent to the objects recreated on C. If such a message has a TN it is resent; otherwise a new ticket request is issued for that message. It is important that each restarted object on C also resend the messages in its message log that were sent to other objects on C. This is necessary to ensure correctness in the case when an object U on C sends a message to another object V on C, such that the message is sent before C checkpoints but is processed by V after the checkpoint. We can correctly restart V's execution after the crash only if U resends the message after restart. When assigning TNs during restart, both the TNTable and the remote message log obtained from processor D are checked for a message with the same sender id, receiver id and SN. This makes sure that a restarted object processes all messages in the same sequence as earlier.

When an object W restarts on processor C it collects a list of the TNs of all the messages resent to it. W then adds to this list the TNs of messages in the remote message log obtained from C's buddy D. After sorting this list it might find that some TNs in the middle are missing. These missing TNs correspond to TNs that were handed out by W before processor C crashed, but they failed to reach the processors of the requesting objects. Some TNs handed out to later requests, however, got to their destinations before C crashed. Though this might seem like a rare condition, it occurs frequently in applications with large numbers of messages. When W has to hand out new TNs it hands out these missing TNs first before continuing with TNs higher than TNCount. W should not skip handing out any TN since W would not able to process any message with a TN higher than the skipped one.

3.4 Multiple Simultaneous Failures

The protocol discussed in the previous subsections works for consecutive crashes only if a second processor crashes after the system has recovered from the previous crash. We now extend the protocol to allow it to deal with most multiple failures. Let us say, a processor H crashes and starts recovering. Now, another processor, say I, crashes and rolls back to a state such that, it needs messages from objects on H that were sent before H's last checkpoint. Rolling H back further than I in order to recover I's state is out of the question because, we want to avoid cascading rollbacks of any sort. Therefore, we need the logs of messages that were sent by objects on H to objects on processor I. However, these logs are not available as the logs of messages sent to objects on other processors are not part of an object's checkpoint. This problem can be solved by making the logs of messages to objects on other processors and the TNTable part of the checkpoint state of Charm++ objects.

Another problem is that there might be messages, from objects on H, that objects on I had processed before I's crash, but their logs were lost when H crashed and rolled back to its previous checkpoint. We modify the remote mode such that, instead of sending a ticket request to the receiver, the sender sends the message itself with sender id and SN attached. The receiver assigns the message a TN and sends the sender id, receiver id, SN and TN tuple to its buddy processor to be logged in the remote message log.

After the buddy acknowledges the receipt of the data, the message is processed in increasing order of TN at the receiver. During restart this logged data about all messages since the last checkpoint is brought back from the buddy and resent messages are assigned TNs by looking up their sender, received ids and SNs in this log. We implement this improvement, but let the users turn it off to avoid the overhead of checkpointing message logs, if they think that the chances of simultaneous failures are low.

The only case in which our solution might fail occurs when processor G crashes just after its buddy processor F has crashed and restarted. As F no longer has G's checkpoint, G cannot restart. The probability of such a pair of crashes happening can be reduced by having G checkpoint as soon as F restarts. This shortens the length of the time window during which a crash might cause an irrecoverable error. This situation arises because unlike [1, 2] we do not use an idealized stable storage. It can be proven that despite this, the protocol reduces the probability of unrecoverable error by several orders of magnitude [24].

3.5 Fast Restart

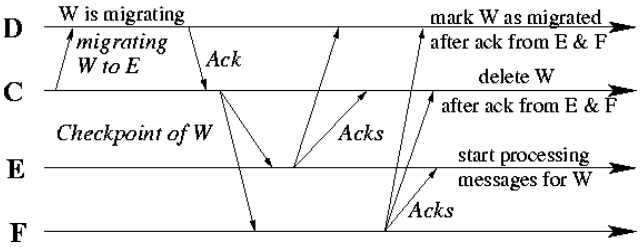


Figure 2. Messaging when processor C sends object W to restart on processor E

After processor C crashes and is restored from its checkpoint, we can redistribute the objects on it among other processors to speed up recovery. We designed a fast restart protocol that would make sure that if there were crashes while distributing objects among other processors, all migrating objects would get recreated and there would be only one copy of each. Let processor C decide to send object W to processor E for restarting. Processor D is the buddy of processor C while processor F is the buddy of processor E. Figure 2 shows the messages exchanged by different processors and the actions taken on receiving those messages. Processor C informs its buddy D of the decision. Processor D marks W as potentially migrated. After processor D has acknowledged the receipt of this short message, processor C sends a copy of W's checkpoint to both processors E and F. After receiving the object W, processor F stores it in the checkpoint of E and sends acknowledgments to C, D and E. Processor E stores the checkpoint and sends acknowledgments to C and D. Processor E starts re-executing messages for W only after it has received the acknowledgment from F. C deletes its copy of W after hearing from both E and F. D also marks W as migrated after it has heard from E and F. At this point W has migrated from processor C to E and if E crashes it will be recreated on E from its checkpoint on F.

During a fast restart if C crashes again before D has received the acknowledgments from E and F, D asks if W and its checkpoint exist on E and F respectively. E stops processing messages for W after being asked this question. If both answer in the positive, D does not recreate W on C and asks E to continue with the execution of messages for W. If not, it recreates W on C and asks E and F to throw away W and its checkpoint. The case in which E but not F has received W and E crashes can be resolved by continuing with W's execution on C after confirming that F does not indeed have W's checkpoint.

Though the fast restart protocol is more complicated than the basic one, the speed up in recovery gained by dividing the work among multiple processors more than makes up for the additional overhead. So, fast restart can significantly shorten the recovery time for an application.

We do a rough analysis of our fast restart protocol. We compare the completion time of an application running the fast restart protocol with the same application running a traditional checkpoint/restart protocol. Let the mean time between failure for the system be m time units. Let the system checkpoint every c time units (not including the checkpoint duration itself). Let duration of a checkpoint be d. Let the runtime of the application without any fault tolerance support be t_0. So time to complete the application with checkpoints $t_c = t_0 + \frac{t_0}{c}d$. If there are n faults, the worst case runtime under the checkpoint scheme will be $t'_c = t_c + n(c + k_c)$ where k_c is the constant overhead for restarting in the checkpoint scheme. On an average, we expect $n = \frac{t'_c}{m}$ faults during a run, so $t'_c = \frac{t_0(1+\frac{d}{c})}{1-\frac{c+k_c}{m}}$. t'_c goes rapidly to infinity as m approaches $c + k_c$. For the message logging protocol, runtime without faults is $t_{ml} = \alpha(t_0 + \frac{t_0}{c}d)$ where α is the ratio of increase in runtime due to the message logging protocol. If the number of objects per processor is v and k_{ml} the overhead of fast restart, then the runtime with faults can be calculated to be $t'_{ml} = \frac{\alpha t_0(1+\frac{d}{c})}{1-\frac{\frac{c}{v}+k_{ml}}{m}}$. The runtime for the message logging protocol goes to infinity rapidly as m approaches $\frac{c}{v} + k_{ml}$. As long as k_{ml} is not much larger than k_c, this is smaller than $c + k_c$ This shows that our fast recovery protocol can deal with higher rates of failure than the checkpointing protocol. Moreover the performance of the fast protocol is better than the checkpoint protocol as long as $\alpha < \frac{m-(\frac{c}{v}+k_{ml})}{m-(c+k_c)}$

4 Experiments

We evaluate the performance of the basic and fast recovery protocols and characterize the applications most suitable to our scheme. We test our protocol on a cluster of 16

Virtual processors per processor	Basic Restart Time(s)	Fast Restart Time(s)
2	28.45	18.31
4	28.21	13.45
8	28.17	9.57
16	29.37	7.58

Table 1. Restart performances on 16 processors

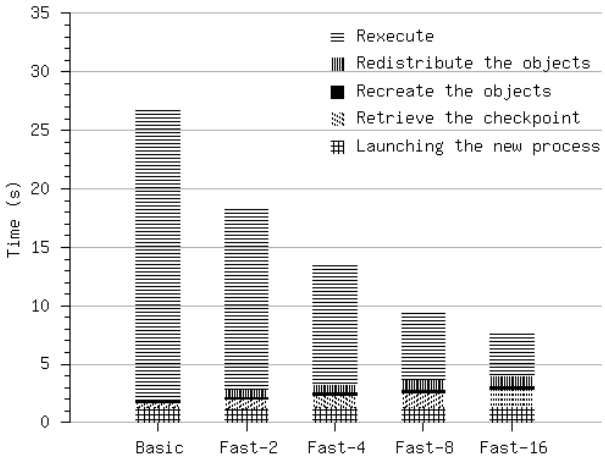


Figure 3. Different phases of the Basic and Fast restart protocols

dual Opteron (Processor 244) machines with 1 GB of memory and 1 GB of swap, connected by Gigabit switched ethernet. We use gcc 4.0.1 and gfortran as the C++ and Fortran compilers respectively.

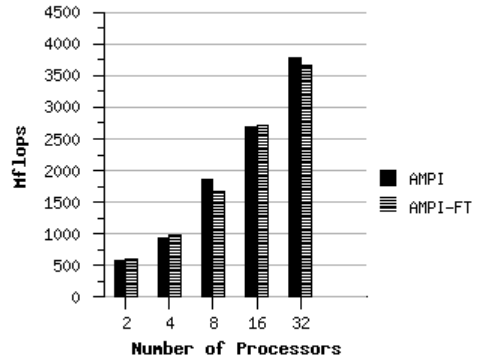
4.1 Restart Performance

We use a 7-point stencil with 3D domain decomposition written in MPI to evaluate the performance of the restart protocols. In each iteration a Charm++ object gets data from its neighbors on all 6 sides and performs some computation. We ran the stencil code with two versions of AMPI, one with the fault tolerance protocol (*AMPI-FT*) and the other without (*AMPI*). In the case of AMPI-FT we checkpointed every 30 seconds. We simulate a fault on a processor by sending SIGKILL to a process running on it. After a processor crashes, the iteration time for objects on the surviving processors increases as those objects wait for the objects on the restarted processor to catch up. We use the maximum increase in iteration runtime over all the surviving objects as a measure of the restart time for both the basic and fast restart protocols. Table 1 shows the time taken for basic and fast restart for different numbers of virtual processors per processor. We ran the stencil code on 16 processors and triggered a fault 27 seconds after a checkpoint. We checkpointed every 30 seconds. Higher numbers of objects per processor allowed the fast restart to distribute work among more processors and led to significantly shorter restart times. Table 1 demonstrates that even having just two objects per processor reduces the restart time significantly. Thus, the recovery time for fast restart is much lower than the time between the crash and the previous checkpoint.

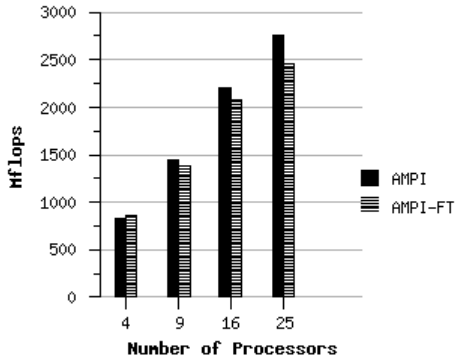
Figure 3 can be used to understand the factors limiting the performance of our restart protocol. Figure 3 compares the time spent in different phases of the basic and fast restart protocols. The basic restart case was run with 16 objects on 16 processors and the fast restart protocol was run with numbers of objects per processor varying from 2 to 16. The time to launch a new process is constant across the different runs. The overhead for retrieving the checkpoint increases with increasing number of objects, because retrieving the checkpoint also includes retrieving the remote message log from the buddy, and as the number of objects per processor increases, the number of entries in the remote message log also increases. The cost of recreating the objects is low and more or less constant across the different runs. The overhead of redistributing the objects across different processors increases slowly with the number of objects per processor. Larger numbers of objects per processor means that objects are distributed among more processors and the fast restart protocol sends out more messages. However, the reexecution time decreases sharply with increasing number of objects per processor as the work of the restarted processor gets distributed among more processors. This decrease is far more than the rise in restart overheads due to higher numbers of objects. As a result, with larger numbers of objects per processor the fast restart protocol can recover much faster than the basic restart. We also found that the forward path overhead for the stencil application was around 10% for the 16 processor run (a more detailed analysis of the forward path cost is presented in Section 4.2). Thus, our protocol provides the stencil application with fast recovery without imposing an unacceptably high performance cost.

4.2 Application studies

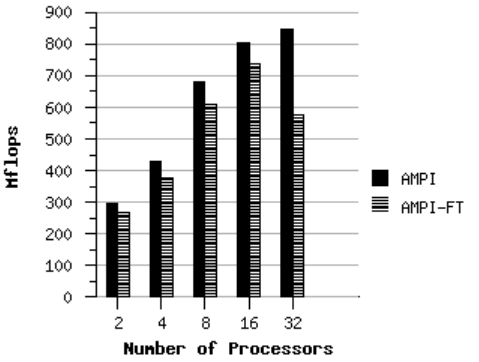
We want to characterize the applications that are most suitable to our message logging protocol. We use the NAS parallel benchmarks to identify the types of applications that would suffer the least performance penalty in the face of this increased message latency. We run NPB3.1 with versions of AMPI with and without the fault tolerance protocol. We show data for only four benchmarks due to lack of space: CG, MG, SP and LU. We run each benchmark with varying numbers of virtual processors and report the best performance for a particular number of physical processors. We do this for both the AMPI and AMPI-FT cases. As we are trying to measure the overhead of the message



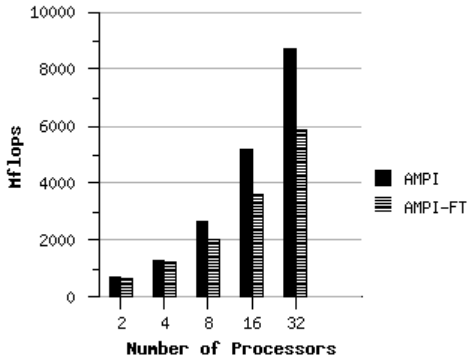
(a) MG class B



(b) SP class B



(c) CG class B



(d) LU class B

Figure 4. Performance of the MG, SP, CG and LU class B benchmarks

	MG with AMPI	MG with AMPI-FT	LU with AMPI	LU with AMPI-FT
Computation Time	68.18 %	68.29 %	86.56 %	87.81%
Idle Time	25.56 %	22.75 %	12.41 %	48.28%
Message Send	4.34 %	5.01 %	0.62 %	2.30 %
Ticket Request Send		4.54 %		0.63%
Ticket Send		1.37 %		1.01%
Local Message Protocol		2.10 %		0.00%

Table 2. Overhead of the protocol expressed as a percentage of the runtime of AMPI for MG and CG on 8 processors

logging protocol, we do not take any checkpoints during the execution of the benchmarks.

Figures 4(a) and 4(b) show that the performance penalty is low for the MG and SP benchmarks respectively. The performance penalty for CG in Figure 4(c) is moderate, whereas that for LU in Figure 4(d) is significant. The different performance penalties imposed by AMPI-FT on each benchmark can be explained if we consider the number of instructions executed per send by each benchmark. As shown in [26] both LU and CG have low instructions per send whereas MG and SP have much higher instructions per send. This means that the increase in message latency forms a smaller fraction of the computation time per message for MG and SP than for LU and CG. So the overall performance penalty is lower for MG and SP. SP has a higher performance penalty compared to MG since SP sends more

and larger messages than MG [26].

In Table 2 we look at the cpu overheads associated with different parts of the message logging protocol. For the high granularity benchmark MG, the cpu overhead of the message logging protocol is the primary source of increased execution time. However, in case of the low granularity benchmark LU, a sharp increase in idle time is responsible for increasing the overall execution time (idle time is the time spent waiting for messages). The increased message latency due to message logging means that objects have to wait longer for messages. MG manages to overlap computation with the wait for messages by using multiple virtual processors per processor. LU fails to do so since it has too little computation per message.

5 Conclusions and Future Work

We presented a protocol for fault tolerant computation that combines sender side message logging with virtualization to provide fast restarts. We evaluated it and found that the fast restarts took much less time than the time interval between the crash and the previous checkpoint. We think that this is a very important feature for large systems that may suffer from frequent partial failures. This allows an application to make much faster progress in the face of failures than traditional fault tolerance protocols. We believe we can speed up the fast restart even more by having the buddy of a restarting processor distribute the objects among different processors rather than shipping the checkpoints to the restarting processor and then distributing them. We would also like to study different strategies for distributing the objects among the remaining processors to get the fastest possible restart. The NAS benchmarks did not have to undergo any modification to use our fault tolerance protocol. We also found that our protocol is very well suited to applications with large computational granularity per message. The latency tolerance provided by virtualization lets us scale in cases where other pessimistic message logging protocols have difficulty doing so. We also found that the cpu overhead of the protocol imposes a performance penalty in high granularity applications, whereas the increased message latency causes performance degradation in low granularity applications.

In the future, we want to analyze the performance of low granularity applications. We believe that reducing the number of protocol messages might help the performance of such applications. In the future, we intend to evaluate the performance penalty of our protocol for real applications. We expect that many large real applications will have a lower penalty than the small NAS benchmarks. We also plan to extend our protocol so that it can deal with the migration of virtual processors in the middle of a computation for load balancing. We expect to be able to borrow ideas from the fast restart protocol to come up with a scheme that would survive crashes in the middle of a migration. It will have to be robust so that it always creates exactly one copy of every object after a crash. This will allow us to fully leverage all benefits of virtualization such as dynamic measurement based load balancing.

References

[1] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "Toward a scalable fault tolerant mpi for volatile nodes," in *Proceedings of SC 2002*, IEEE, 2002.

[2] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: A fault tolerant MPI for volatile nodes based on the pessimistic sender based message logging programming via processor virtualization," in *Proceedings of SC'03*, November 2003.

[3] L. V. Kalé, "Performance and productivity in parallel programming via processor virtualization," in *First Intl. Workshop on Productivity and Performance in High-End Computing (HPCA 10)*, (Madrid, Spain), February 2004.

[4] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of LCPC 03*, (College Station, TX), October 2003.

[5] L. V. Kalé and S. Krishnan, "Charm++: Parallel programming with message-driven objects," in *Parallel Programming using C++* (G. V. Wilson and P. Lu, eds.), pp. 175–213, MIT Press, 1996.

[6] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message passing systems," Tech. Rep. CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Oct. 1996.

[7] Y. M. Wang, *Space reclamation for uncordinated checkpointing in message-passing systems*. PhD thesis, University of Illinois U-C, Aug 1993.

[8] G. Stellner, "CoCheck: Checkpointing and process migration for MPI," in *Proceedings of the 10th International Parallel Processing Symposium*, pp. 526–531, 1996.

[9] A. Agbaria and R. Friedman, "Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations," *Cluster Computing*, vol. 6, pp. 227–236, July 2003.

[10] Y. Chen, J. S. Plank, and K. Li, "Clip: A checkpointing tool for message-passing parallel programs," in *Proc. of the 1997 ACM/IEEE conference on Supercomputing*, pp. 1–11, 1997.

[11] C. Huang, "System support for checkpoint and restart of Charm++ and AMPI applications," Master's thesis, Dep. of Computer Science, University of Illinois, Urbana, IL, 2004.

[12] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *IEEE International Conference on Cluster Computing*, September 2004.

[13] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of mpi programs," in *Principles and Practice of Parallel Programming*, June 2003.

[14] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A distributed domino-effect free recovery algorithm," in *IEEE International Symposium on Reliability, Distributed Software, and Databases*, pp. 207–215, December 1984.

[15] L. Alvisi, E. N. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel, "An analysis of communication induced checkpointing," in *Symposium on Fault-Tolerant Computing*, pp. 242–249, 1999.

[16] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 3, pp. 204–226, 1985.

[17] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under unix," in *ACM Transactions on Computer Systems*, pp. 1–24, February 1989.

[18] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging," in *The 7th annual international symposium on fault-tolerant computing*, IEEE Computer Society, 1987.

[19] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 526–531, 1992.

[20] A. Bouteiller, B. Collin, T. Herault, P. Lemarinier, and F. Cappello, "Impact of event logger on causal message logging protocols for fault tolerant mpi," in *IPDPS'05*, p. 97, 2005.

[21] I. Lee, H. Y. Yeom, T. Park, and H.-W. Park, "A lightweight message logging scheme for fault tolerant mpi.," in *PPAM*, pp. 397–404, 2003.

[22] W. Gropp and E. Lusk, "Fault tolerance in message passing interface programs," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 363–372, 2004.

[23] O. S. Lawlor and L. V. Kalé, "Supporting dynamic parallel object arrays," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 371–393, 2003.

[24] S. Chakravorty and L. V. Kalé, "A fault tolerant protocol for massively parallel machines," in *FTPDS Workshop at IPDPS'2004*, (Santa Fe, NM), IEEE Press, April 2004.

[25] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Transactions on Computer Systems*, vol. 1, no. 3, pp. 222–238, 1983.

[26] W. E. Cohen, R. K. Gaede, and W. D. Garrett, "Interconnection network independent characterization of communication traffic in the nas benchmarks via processor performance monitoring hardware."