A Fault Tolerance Protocol with Fast Fault Recovery

Sayantan Chakravorty and Laxmikant V. Kalé

Department of Computer Science,
University of Illinois at Urbana-Champaign
{schkrvrt,kale}@uiuc.edu

Abstract

Fault tolerance is an important issue for large machines with tens or hundreds of thousands of processors. 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 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 object-based processor virtualization. We evaluate our implementation of the protocol in the Charm++/Adaptive MPI runtime system. We show that our protocol provides fast restarts and, for many applications, has low fault-free overhead.

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 even larger numbers of processors are being planned and designed. Machines with large numbers of components are likely to suffer from partial failures frequently. ASCI-Q was reported to suffer a failure every few hours [4]. 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 ([4, 5]) 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 new 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. On a large machine, processors that are not dependent on data from the failed processor can continue to execute further during crash recovery. More importantly, object based virtualization [18] allows us to distribute the work of the 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. Object-based virtualization also helps us reduce the performance penalty imposed by message logging on applications. Our scheme distinguishes itself from other sender-side-message-logging protocols, by using object based virtualization to speedup recovery and moderate the cost of message logging.

Our scheme has been implemented for a version of MPI, called Adaptive MPI[16], and so can be used by all MPI programs. Applications written in Charm++[19], which is the underlying layer of Adaptive MPI, can use our scheme as well.

2 Related Work

We discuss existing fault tolerance protocols and the key ideas of object based virtualization in this section.

Fault Tolerance: The solution space for fault tolerance protocols can be divided into two main categories: check-point based and log-based recovery protocols [12]. 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 save its checkpoint independent of the other processors, are fast and memory efficient [25]. 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 [23], Starfish [1], Clip [10] and AMPI [15, 26] to provide fault tolerant versions of MPI. A coordinated checkpointing algorithm that uses application level checkpointing is presented in [8]. 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[7]. However it was found that communication induced methods did not scale well to large number of processors [2].

The second category of fault tolerance protocols depend on stored message logs and uncoordinated checkpoints for recovery. After a processor crashes, all the messages are resent to the recovering 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 [24]. 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 allowing it to be processed. 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 [3] or by storing the message log in the sender's memory [17]. MPICH-V1 and V2 [4, 5] 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 [24]. 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. Tracking causality and recovery from faults are complex operations in causal protocols. Manetho [13], MPICH-Vcausal [6] and the protocol in [20] are examples of causal logging systems.

Virtualization: Object based virtualization [18] (also referred to as processor virtualization) encourages a user to view his computation as a large number of interacting objects. These objects are also referred to as virtual processors(VPs). The user decomposes his computation into VPs without caring about the number of physical processors present. The runtime system is responsible for mapping VPs to physical processors and can change the mapping during the execution of a program. VPs 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.

Processor virtualization is the primary idea behind Charm++ [19] and Adaptive-MPI(AMPI) [16]. It renders applications latency tolerant by letting them overlap communication and computation. While one VP on a processor is waiting for a message, another VP can continue with its computation. Processor virtualization also enables runtime measurement based dynamic load balancing. The idea has been useful in numerous applications from molecular dynamics [21] to cosmology [14]. AMPI is an implementation of MPI on top of the Charm++ runtime system that allows traditional MPI codes to reap the benefits of virtualization such as latency tolerance and runtime load balancing. The Charm++ runtime system supports multiple checkpoint-based fault tolerance protocols [26, 15] and a basic message logging-based protocol [9].

3 Protocols

Our fault tolerance protocol is entirely software based and does not depend on any specialized hardware. It however makes the following assumptions about the hardware. i) The processors in the system are fail-stop [22]. This means that when a processor crashes it remains halted. 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.

Our protocol uses sender-side pessimistic message logging and object-based virtualization for fast fault recovery. Virtualization in conjunction with our message logging protocol yields a number of benefits. First, it makes applications more latency tolerant. This helps us hide the increased latency due to the message logging protocol. It is also the primary idea behind faster restarts since it allows us to spread the work of the failed processor among other processors. We treat the VPs, and not the physical processors, as the communicating entities that send and receive messages. Since a VP's state is modified only by the messages it receives, we can apply the PWD assumption to VPs instead of physical processors. After a crash, if a VP reexecutes messages in the same sequence as before, it can recover its exact pre-crash state.

The first three subsections in this section describe the message logging, checkpointing and restart protocols for single faults. Next, we extend them to deal with multiple faults. The last subsection describes the fast restart scheme.

3.1 Message Logging Protocol

We designed the message logging protocol such that, after a crash, a Charm++ object (or AMPI process) processes the same messages in the same order as before the crash. The protocol requires that each message exchanged between objects have four data fields: i) The *sender id* identifies the object that sent the message. ii) The *receiver id* identifies the object that is to receive this message. iii) The *sequence number* (*SN*) of a message is a count of the number of messages sent from the message's sender to its receiver before this message was generated. iv) The *ticket number*(*TN*) of the message. The receiver of a message assigns the TN and processes all the messages it receives in increasing order of their TNs. TNs of all the messages received by an object form a single contiguous sequence.

Each Charm++ object has a unique *id*. Every object maintains a table called the *SNTable* that tracks the number of messages sent to different objects. The SNTable is used to assign *SN* to messages. Each message sent by an object is stored in the object's *message log*. An object stores the sender's id, SN and TN for each message received since the last checkpoint in a table called the *TNTable*. An object stores the highest TN processed by it as *TNProcessed*. An object stores the highest TN assigned by it as *TNCount*.

3.1.1 Remote Mode

When the sender(α) and receiver(β) objects are on different processors, the message logging protocol is said to operate in the *remote* mode. Before sending a message, α looks up β in its SNTable and finds the number of messages sent to β previously. Object α increments that count and assigns it as the SN of the message. Object α then stores the message in its message log.

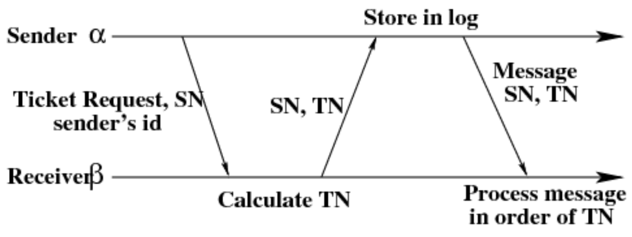


Figure 1. Remote mode of the message logging protocol

As seen in Figure 1, the sender α then sends a request for a ticket, consisting of α's id and the message's SN, to β. On receiving the request, β looks up the tuple $\{\alpha, SN\}$ in β's TNTable. If there is no matching entry in the TNTable, there are two possibilities: i) the common case that α is sending a new message to β, ii) the rare case where the sender α is recovering from a fault and is re-sending a message that was processed by β before its last checkpoint. We can distinguish between the two cases by comparing the SN in the ticket request to the lowest SN from α in β's TNTable (say *l*). If the SN in the ticket request is higher than *l*, then we are dealing with the common case that α is sending β a new message. Object β increments TNCount and decides on this value as the TN. β also adds an entry for the tuple $\{\alpha, SN, TN\}$ to its TNTable. The TN is returned to the sender α along with α's id and the SN. If the SN in the ticket request is lower than *l* then the sender α has sent a ticket request for a message that was processed by β before its last checkpoint. Since pessimistic message logging does not suffer from cascading rollbacks, β can safely tell α to discard this message. If β finds an entry corresponding to the ticket request in its TNTable, it means that in the past β has assigned a TN to this message from α and that α is recovering from a fault. Object β will reply back with this TN. If the value of this TN is lower than the TNProcessed for β, β marks the TN as old. An old TN corresponds to a message β has already processed since the last checkpoint.

When α 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, α sends the message to β. When β receives the message, it checks if the message's TN is less than or equal to its TNProcessed. If it is, β discards the message as it has already processed this message and should not do so again. If the message's TN is higher than $TNProcessed + 1$, β defers processing this message. If the message's TN is exactly equal to $TNProcessed + 1$, then β processes the message and then increments β's TNProcessed by 1.

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 is the same as in the sender side message logging protocols of [17, 5]. However, as we shall in Section 4, virtualization allows us to mitigate the penalty imposed by this increased latency.

3.1.2 Local Mode

If we were to use the above protocol for messages between two objects on the same processor, the log of a message would exist on the same processor as its receiver. If this processor crashes, it would become impossible to re-execute the messages in the correct sequence at the receiver. Therefore, we define a *local* mode of the message logging protocol to deal with this case. Figure 2 shows the different methods called and messages exchanged during the local mode of the message logging protocol.

Each processor is assigned a *buddy* processor. A processor has only one buddy and is the buddy of only one processor. Let us say that object α on processor C wants to send a message to object β on the same processor. As the first step, α assigns the message a SN in the same way as in the remote mode described in Section 3.1.1. Object α then asks β for a ticket by invoking the ticket generation routine with α's id and the message's SN as arguments. The ticket generation routine uses the same algorithm described in Section 3.1.1. We are able to use a method invocation instead of a message because in this case, α and β are on the same processor.

After β has returned a ticket number, α stores α's id, β's id, SN and TN (referred to as the *message meta data*) in the *message meta data table* (*MDTable*) maintained on C's buddy processor (D). Object α sends the message to β only after receiving an acknowledgment from D that the meta-data for the message has been stored in D's MDTable. As a result, the latency for a message to a local object becomes the same as that of a message to a remote object. After β has processed the message, it tells α to remove the message from its message log. We can do this because, as described later in Section 3.2, we checkpoint all the objects on a processor at the same time. So any checkpoint in the future will save the state of α as having sent the message and β as having processed it. If processor C is ever restarted from that checkpoint, α would not need to resend the message. This allows α to remove the message from α's log after β has processed it.

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 affect the rest of the protocol. In this paper, we chose to implement an in-memory checkpoint. Storing a checkpoint in the memory

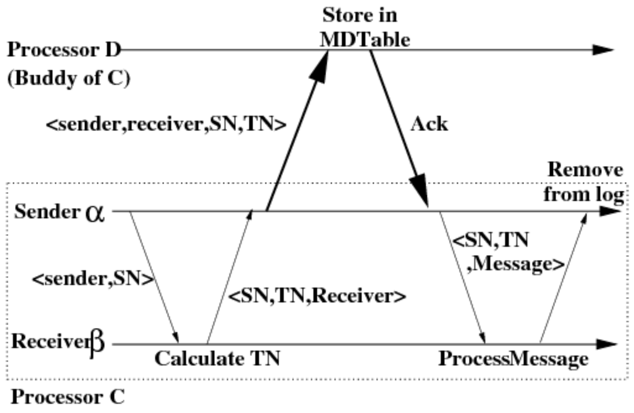


Figure 2. Messages in the local mode of the message logging protocol

of a remote processor is much faster than storing it in a remote storage server [26], 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 buddy processor simplifies 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). It should be noted that the only messages in the message log that were sent to objects on the same processor are those that have been sent but not processed at the time of checkpoint. All the objects on a processor checkpoint at the same time.

The checkpoint protocol also provides a mechanism to perform garbage collection on the message logs. A processor, say C, packs up the state of all the objects on it and sends it to its buddy processor, say D. Each object on C also stores its TNProcessed at the time of checkpoint as *TNCheckpointed*. D stores the new copy of C's checkpoint, deletes the old copy and sends an acknowledgment to C. On receiving the acknowledgment, the TNTable of each object on C can garbage collect entries with TN less than TNCheckpointed. Each object on C sends out garbage collection messages containing TNCheckpointed to all objects that had sent it messages since its previous checkpoint. When an object γ receives a garbage collection message from object α on processor C, it removes all messages to α in its message log that have a TN lower than the TNCheckpointed. A similar garbage collection message is sent to processor D, so D can remove old entries from the MDTable. Garbage collection is done lazily so that it interferes as little as possible with the application.

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 on 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. Figure 3 shows the messages exchanged after the new processor C has started up. C recreates all the objects that used to exist on it from the checkpoint and MDTable fetched from D. The entries in the MDTable are separated by receiver and added to each receiver's TNTable. C then broadcasts a request to resend 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. If an object α on the restarted processor C has a message to object β, also on C, in its log, α needs to resend this message. Object α had sent this message before C had checkpointed but β processed the message after the checkpoint. Since α will not regenerate this message during restart, the message needs to be resent to allow β to make progress.

Object α on processor C also collects a list of the TNs of all the messages resent to it. α then adds to this list the TNs of messages in the MDTable obtained from C's buddy D. After sorting this list α might find that some TNs in the middle are missing. These missing TNs correspond to messages that were given TNs but were not processed by α before the crash. We are sure they were not processed since α would have processed a message only after its TN had been saved in either the sender's log or the MDTable. So when α gives out new TNs it hands out these missing TNs before continuing with TNs higher than TNCount. α should not skip handing out any TN since α will not be able to process any message with a TN higher than the skipped one.

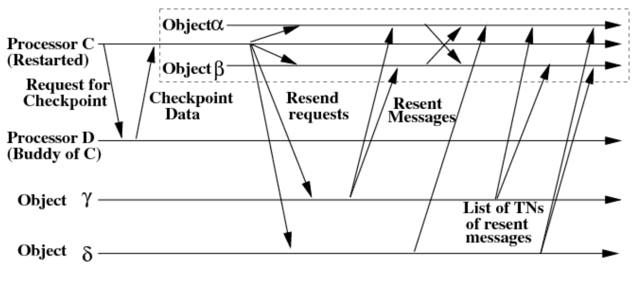


Figure 3. Messages sent during the basic restart protocol

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 since 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 to objects on I that had been processed before I's crash, but their logs were lost when H crashed and rolled back to its previous checkpoint. Without the logs from objects on H, the objects on I cannot give these messages the same TNs after the crash as before. 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 $\langle$sender id, receiver id, SN and TN$\rangle$ tuple to its buddy processor to be logged in the MDTable. After the buddy acknowledges the receipt of the data, the message is processed in increasing order of TN at the receiver. We implement these improvements, 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 still fail is when processor C crashes just after its buddy processor D has crashed and restarted. As D no longer has C's MDTable, C cannot restart. The probability of such a pair of crashes happening can be reduced by having a processor (C) checkpoint as soon as it detects that its buddy (D) has restarted. This shortens the length of the time window during which a crash might cause an irrecoverable error. This situation arises because unlike [4, 5] we do not use an idealized stable storage. We show in [9] that despite this, the protocol reduces the probability of unrecoverable error by several orders of magnitude.

3.5 Fast Restart

The initial part of the fast restart protocol is similar to the basic restart protocol in Section 3.3. After processor C crashes and is restarted, C fetches its checkpoint and MDTable from its buddy processor D. As in the case of basic restart, processor C recreates its objects using the data from D. However, at this point the fast restart protocol diverges from the basic one. Processor C distribute its objects among different processors. Then C broadcasts the request to resend logged messages. In the case of fast restart, the resend request also contains the new location (physical processor) of each object that used to exist on processor C. The resent messages as well as the list of TNs is sent to the new location of each object instead of processor C. This allows us to distribute the work on the restarted processor among other processors and speed up the restart.

However, it is possible that while an object α is being moved from processor C to E either C or E crashes. Figure 4 shows a protocol that makes sure that even if C or E were to crash while α is being moved, α would get recreated and there would be only one copy of it in the whole system. At the end of the protocol, α has migrated from processor C to E. If E crashes after that, α will be recreated on E from its checkpoint on F (E's buddy). If E crashes before the protocol completes, α will be recreated on C. If C crashes again before D has received the acknowledgments from E and F, D asks if α and its checkpoint exist on E and F respectively. E stops processing messages for α after being asked this question. If both answer in the positive, D does not recreate α on C and asks E to continue with the execution of messages for α. If not, D recreates α on C and asks E and F to throw away α and its checkpoint. If E's buddy processor F, crashes before sending the acks the migration of object α from C to E is aborted. Though the fast restart protocol involves more messaging 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.

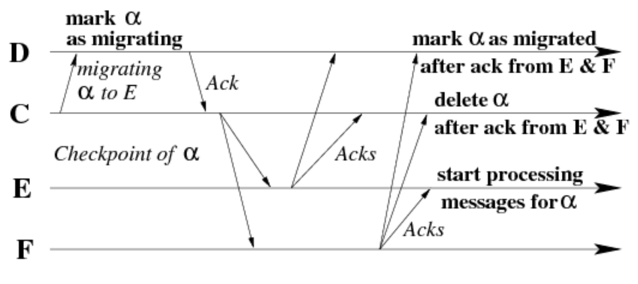


Figure 4. Messaging when processor C sends object α to restart on processor E

We now present 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 using a traditional checkpoint /restart protocol. Let the MTBF 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 time units. Let the runtime of the application without any fault tolerance support be t_0 time units. 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 dual Opteron (Processor 244) machines with 1 GB of memory and 1 GB of swap, connected by switched Gigabit.

4.1 Restart Performance

We use a 7-point stencil with 3D domain decomposition written in MPI to evaluate the perform-ance of the restart protocols. In each iteration an MPI process 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 (VPs) 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.

To understand the factors limiting the speedup of our restart protocol, Table 2 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 MDTable from the buddy, and as the number of objects per processor increases, the number of entries in the MDTable 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 increases as the fast restart protocol sends out more messages to distribute more objects. However, the re-execution 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 a prohibitively high performance cost.

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	09.57
16	29.37	07.58

Table 1. Restart time on 16 processors

Phase of Restart	Basic 1 VP	Fast 2VP	Fast 4VP	Fast 8VP	Fast 16VP
New Process	1.29	1.24	1.34	1.27	1.28
Get Checkpoint	0.44	0.76	1.05	1.31	1.55
Recreate VP	0.05	0.09	0.12	0.17	0.21
Distribute VP	0.00	0.65	0.71	0.81	0.91
Re-execute	25.18	15.57	10.23	6.01	3.64
Total	26.96	18.31	13.45	9.57	7.59

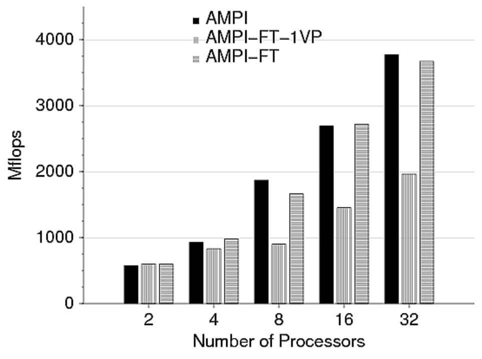
Table 2. Time spent in different phases of the Basic and Fast restart protocols

4.2 Application studies

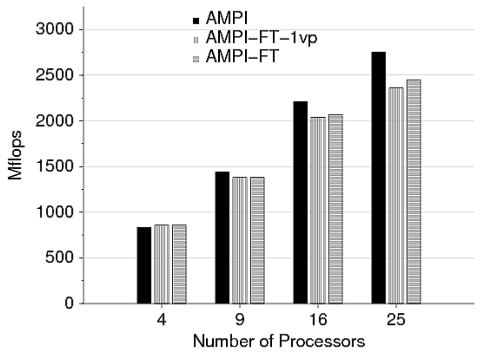
We want to characterize the applications that are most suitable to our message logging protocol and evaluate the effect of processor virtualization in reducing the performance penalty of the message logging protocol for different applications. We do so by measuring the performance penalty suffered by the NAS parallel benchmarks due to the increased message latency of message logging. We run NPB3.1 with versions of AMPI with and without the fault tolerance protocol. Since we want to evaluate the performance penalty of just the message logging protocol, we do not take any checkpoints or simulate any faults. We show performance data for only four benchmarks in Figure 5 due to lack of space: CG, MG, SP and LU. during the execution of the benchmarks. We run each benchmark with varying numbers of VPs for both AMPI and AMPI-FT. We compare the best performance of AMPI-FT with not only the best performance of AMPI on a particular number of physical processors but also the performance of AMPI-FT with 1 VP per processor. This allows us to clearly see the benefit of processor virtualization for different benchmarks and compare our message logging protocol with existing ones.

We can see in Figure 5(a) that the MG benchmark benefits tremendously from processor virtualization. Without virtual-ization, MG would have experienced a much higher performance penalty similar to that of the message logging protocol in [5]. Even in the case of SP in Figure 5(b), virtualization helps performance for higher number of processors. Therefore, with multiple VPs per processor MG and SP have low performance penalties. The performance penalty for CG in Figure 5(c) is moderate, whereas that for LU in Figure 5(d) is significant. Even in the case of CG and LU , processor virtualization helps to greatly moderate the performance penalty of message logging by adaptively overlapping communication and computation. This allows our message logging protocol to perform better than other sender based message logging protocols.

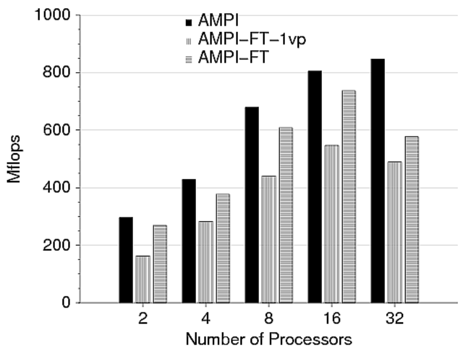
The different performance penalties imposed by AMPI-FT on each benchmark can be explained by considering the number of instructions (of user computation) executed per



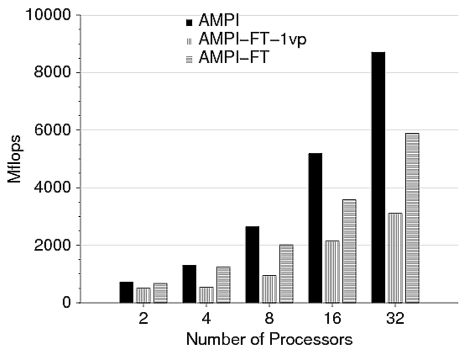
(a) MG class B



(b) SP class B



(c) CG class B



(d) LU class B

Figure 5. Fault-free performance of the MG, SP, CG and LU class B benchmarks

message by each benchmark. As shown in [11], both LU and CG have less than $\frac{1}{5}^{th}$ the number of instructions per message than MG and SP. This means that the increase in message latency is 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.

In Table 3 the time spent by the cpu in different phases of the protocol is expressed as a percentage of the runtime while using AMPI on 8 processors. AMPI shows the best performance when there is 1 VP per processor. For AMPI-FT we look at the cases when each processor has 1 VP and 4 VPs (4 VPs gave the best performance for both benchmarks). Both MG and LU show a drastic increase in idle time (time spent waiting for communication) when AMPI-FT is used with 1 VP per processor. This is because message logging increases the message latency and a lot of time is wasted waiting for messages. Although having multiple VPs per processor increases the protocol overheads slightly, it decreases the idle time sharply for both benchmarks by allowing them to adaptively overlap computation and communication of different VPs on a processor. Since MG has a coarser computational granularity than LU it is better able to overlap computation and the time spent waiting for communication. As a result, with multiple VPs the performance penalty for MG is much smaller than that of LU. For MG, protocol overhead is the source of most of the performance penalty. On the other hand for LU the increased message latency is the primary source of performance penalty. Thus, we see that processor virtualization helps improve the performance of the message logging protocol and hide the performance penalty in applications with coarse as well as fine computational granularity.

4.3 Performance vs Granularity

We use a synthetic benchmark to take a closer look at the relationship between performance and the granularity of an application. The synthetic benchmark is a very simple it-

	MG on 8 processors			LU on 8 processors		
	AMPI	AMPI-FT	AMPI-FT	AMPI	AMPI-FT	AMPI-FT
VP per processor	1	1	4	1	1	4
Computation Time	68.18%	68.22%	68.29%	86.56 %	86.83%	87.81%
Idle Time	25.56%	130.90%	22.75%	12.41 %	189.73%	48.28%
Message Send	4.34%	4.65%	5.01%	0.62 %	1.31%	2.30 %
Ticket Request Send		3.11%	4.54%		0.34%	0.63%
Ticket Send		1.12%	1.37%		0.86%	1.01%
Local Message Protocol		0.00%	2.10%		0.00%	0.00%
Total	98.08 %	208.00%	104.06%	99.59 %	279.07%	140.03 %

Table 3. Overheads expressed as a % of the runtime of AMPI for MG and LU on 8 processors

erative MPI program. The MPI processors are logically arranged in a ring. In every iteration, each MPI process sends a short message to its neighbors on the left and right. Each MPI process also receives a message from its two neighbors. After that, every MPI process performs some calculations for a specified amount of time. The time spent in the calculation in each iteration is the *granularity* of the application.

We test the synthetic benchmark on the Tungsten cluster using the myrinet interconnect. We vary the granularity from 100 μs to 100 ms and measure the average iteration time 10000 iterations. We evaluate the performance of AMPI and AMPI-FT on 8 processors with both 1 and 4 VPs per processor. Table 4 measures the relative performance penalty by showing the ratio of the average iteration time to granularity for AMPI and AMPI-FT (the closer the ratio is to 1, the lower the performance penalty). We can see that the relative performance penalty for AMPI-FT is low for granularities equal to and coarser than $10ms$. For finer granularities, particularly the 100 μs case, there is significant performance penalty. However, in all cases the performance penalty is lower when each processor has 4 VPs rather than 1. This is broadly in line with the conclusions drawn in the previous section that performance penalty of AMPI-FT is lower for coarser granularity and that virtualization helps reduce the penalty in almost all cases.

In spite of virtualization, the performance of AMPI-FT in the $100\mu s$ granularity case is unsatisfactory. One major difference between AMPI and AMPI-FT was that with 4 VPs per processor, AMPI sent out only 2 messages on the network per processor per iteration compared to 18 for AMPI-FT. This happens because messages between VPs on the same physical processor result in messages to the buddy processor in the case of AMPI-FT but not in the case of AMPI. This meant that in the $100\mu s$ case, AMPI-FT was trying to send out a massive 180000 messages per second. We found that the round trip time for the myrinet network degraded sharply when there were more than 60000 messages/second between two processors.

This led us to try and reduce the number of messages

		Granularities			
Protocol	VP	100 μs	1 ms	10 ms	100 ms
AMPI	1	1.58	1.042	1.0045	1.0008
AMPI	4	1.42	1.079	1.0135	1.0019
AMPI-FT	1	4.36	1.289	1.0309	1.0041
AMPI-FT	4	3.71	1.191	1.0167	1.0025
AMPI-FT message combining	4	2.50	1.103	1.0151	1.0020

Table 4. The ratio of average iteration time to granularity of the synthetic benchmark on 8 processors

exchanged by a processor and its buddy. We merged multiple message-meta-data sent to a buddy into one message. We also merged multiple acknowledgements sent from the buddy into one message. The last column in Table 4 shows that message combining reduces the performance penalty sharply for the $100\mu s$ case. Message combining also reduces the performance penatly of coarser granularities.

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 showed that the fast restarts took much less time than the time interval between the crash and the previous checkpoint. This allows an application to make much faster progress in the face of failures than traditional fault tolerance protocols. We plan to study different strategies for distributing the objects among the remaining processors to get the fastest possible restart. 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 improved the performance of fine granularity applications by combining some protocol messages. In the future, we plan to investigate methods to further reduce the number of protocol messages. We also 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. This will allow us to fully leverage all benefits of virtualization such as dynamic measurement based load balancing.

Acknowledgements: We would like to thank Gengbin Zheng, whose timely help and insight made this work possible. The work in this paper was supported in part by the Department of Energy under grant W-7405-ENG-48.

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing*, 6(3):227–236, July 2003.

[2] 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*, pages 242–249, 1999.

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

[4] 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.

[5] 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.

[6] 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*, page 97, 2005.

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

[8] 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.

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

[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*, pages 1–11, 1997.

[11] 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.

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

[13] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, 1992.

[14] F. Gioachin, A. Sharma, S. Chackravorty, C. Mendes, L. V. Kale, and T. R. Quinn. Scalable cosmology simulations on parallel machines. In *7th International Meeting on High Performance Computing for Computational Science (VECPAR)*, July 2006.

[15] 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.

[16] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of LCPC 03*, College Station, TX, October 2003.

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

[18] 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.

[19] L. V. Kalé and S. Krishnan. Charm++: Parallel programming with message-driven objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

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

[21] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[22] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

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

[24] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

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

[26] 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.