A Fault Tolerant Protocol for Massively Parallel Systems

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

Abstract

As parallel machines grow larger, the mean time between failure shrinks. With the planned machines of near future, therefore, fault tolerance will become an important issue. The traditional method of dealing with faults is to checkpoint the entire application periodically and to start from the last checkpoint. However, such a strategy wastes resources by requiring all the processors to revert to an earlier state, whereas only one processor has lost its current state. We present a scheme for fault tolerance that aims at low overhead on the forward path (i.e. when there are no failures) and a fast recovery from faults, without wasting computation done by processors that have not faulted. The scheme does not require any individual component to be fault-free. We present the basic scheme and performance data on small clusters. Since it is based on Charm++ and Adaptive MPI, where each processor houses several virtual processors, the scheme has potential to reduce fault recovery time significantly, by migrating the recovering virtual processors.

1 Introduction

Scientific and technical computing exhibits a definite trend toward massively parallel systems with tens of thousands of processors. Examples of such machines are ASCI-Q, System X, Earth Simulator and Bluegene/L. These machines, with a large numbers of components, are likely to suffer from frequent partial failures. The Mean Time Between Failures (MTBF) of such systems, unlike present systems, could plausibly be on the order of minutes . Machines like the ASCI-Q already experience a failure every few hours [6].

Thus for any application to run for a useful length of time, it must tolerate node failures. However, fault tolerance on such huge machines has its own set of requirements. The cost of the fault tolerant protocol should be a small percentage of the execution time of the application. The impact of a crash, on a fault free processor's performance, should depend on the degree of coupling between the crashed processor and the fault free processor. The protocol should not be dependent on any "completely reliable component".

In this paper, we present the design and current implementation of a protocol for fault tolerant computation on massively parallel machines. The aim is to build a system that has a low cost during normal execution, while allowing fast restarts at the same time. The implementation of the protocol in Charm++ [18] will provide AMPI [16] users with a transparent fault tolerant MPI. This work is a part of ongoing research on fault tolerance in our laboratory.

2 Problem Specification

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 [19]. It 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 network is assumed to be reliable. However, it makes no guarantees about the delivery order of messages. iv) The piecewise deterministic assumption (PWD) [21] should hold. It is assumed that the only non-deterministic events affecting a processor are message receives. v) The machine is assumed to have a distributed system for detecting faults. vi) There are no fully reliable nodes in the system.

We came up with the following set of requirements for our fault tolerance protocol. i) The first and foremost criterion for any such protocol is correctness. ii) The protocol for taking checkpoints should be fast and scale to a large number of processors. iii) The restart should be as fast as possible. The recovery mechanism should restart only the crashed processor. The effect of the crash on other processors should be minimized. iv) The fault free run-time overhead should be as low as possible. v) It should be as transparent to the user as possible. This is important for any usable fault tolerance protocol because most parallel programs are written by application scientists who do not want to be bothered with writing complicated routines for handling faults.

3 Related Work

The literature describes a range of possible solutions for fault tolerance [14]. The two major classes of solutions are checkpoint-based and log-based rollback-recovery schemes.

3.1 Checkpoint-based methods

In checkpoint-based methods, the state of the computation is periodically saved to stable storage. If there is a failure, then the computation is restarted from one of these previously saved states. Checkpoint-based methods can be classified into three categories, by the type of coordination between different processes while taking checkpoints: uncoordinated checkpointing, coordinated checkpointing and communication induced checkpointing.

In *uncoordinated* checkpointing, each process saves its state independently of the other processes. The checkpoint not only is fast but also can be taken when most convenient, like when the state of the process is small [23]. However uncoordinated checkpointing is susceptible to *rollback propagation*. Rollback propagations also makes it necessary for each processor to store multiple checkpoints. A garbage collection algorithm must be run periodically to keep a bound on the amount of memory consumed by these multiple checkpoints. The potentially unbounded cost of rollback along with the other drawbacks makes uncoordinated checkpointing unsuitable for our requirements.

Coordinated checkpointing requires processes to coordinate their checkpoints in order to form a consistent global state. It can be blocking as in [22] and the hardware blocking used to take system level checkpoints in IBM-SP2. or non-blocking like Chandy-Lamport's distributed snapshot algorithm [11]. This method is appropriate when faults are rare. However, on large machines, coordinated checkpointing of all hues suffer from the disadvantage that after a crash all the processors have to be rolled back to the previous checkpoint. CoCheck [20], Starfish [1], Clip [12], AMPI [18] use coordinated checkpointing to provide fault tolerant versions of MPI. A non-blocking coordinated checkpointing algorithm that uses application level checkpointing is presented in [10].

Communication induced checkpointing allows processes to take some of their checkpoints independently, while preventing the domino effect by forcing the processors to take additional checkpoints [9]. However it doesn't scale well with increasing number of processors [3] and the large number of forced checkpoints nullify the benefit accrued from the autonomous local checkpoints.

3.2 Log-based methods

Log-based methods are based on the piecewise deterministic (PWD) assumption [21]. PWD claims that all the non-deterministic events affecting a process can be identified and that the data required to re-execute that event can be stored in the log. After a crash, the non-deterministic events, in this case messages, can be re-executed in the same order as before to bring the processor to the exact pre-crash state. Message logging based rollback recovery protocols can be classified into three types by the frequency with which the message log is saved to stable storage.

The first type is *pessimistic* logging, in which each received message must be logged to stable storage before it can be processed. Since a crashed processor always recovers to the last pre-failure state visible to other processors, pessimistic logging doesn't suffer from rollback propagation. As a result, each processor needs to store only one checkpoint and garbage collection of old logs is trivial. However, pessimistic logging does incur a fault-free runtime cost due to the synchronous logging of messages before their execution. The MPICH-V [6, 7] protocols are examples of systems using pessimistic logging protocols.

Optimistic logging saves the logs in a volatile storage and periodically flushes it to stable storage [21]. Though this reduces the failure free execution overhead significantly, it suffers from complicated recovery and garbage collection. It also suffers from an unbounded rollback problem, caused by messages whose logs have been lost. These disadvantages make optimistic logging unacceptable according to our specifications.

Causal logging has a low failure free overhead like optimistic logging, while like pessimistic logging it limits the rollback of any failed process to the most recent checkpoint [2, 13]. It however requires complicated causality tracking and recovery. Manetho[13] is an implementation of a causal logging scheme.

In a pessimistic protocol, the cost of synchronous logging on the receiver side results in high performance overhead. This overhead can be reduced by using specialized hardware [4, 5]. The overhead of synchronous logging can also be avoided by logging the messages in the volatile memory of the sender. If the receiver crashes, the sender just resends the messages maintained in its log. This reduces the overhead of logging and also removes the need for a special stable storage. The SBML protocol specified in [17] utilizes this technique for logging based fault tolerance. FT-MPI [15], which lets the application handle the rollback and recovery, does not meet the requirement of transparency.

4 Design

Message logging was preferred over checkpoint based protocols because the latter fail to meet our third requirement set out in Section 2. As a result, as [8] shows, for large data sets and high rates of failures message logging outperforms checkpointing based protocols. Sender based message logging was chosen as it has a lower logging cost compared to receiver side logging and doesn't need a stable storage. However, it does have a cost in terms of message transmission latency. Though optimistic logging schemes might have lower logging overheads, the rollback is slower and more complicated.

It was further decided that the asynchronous checkpoints would be in-memory checkpoints. Each processor would have a buddy and would store its checkpoint in the buddy's memory. The buddy relationship may be symmetric but should never be reflexive. Most protocols that use a separate stable storage server for storing the checkpoint, like MPICH-V2 [7], also ship their process images across the network to the servers. However, the time cost of storing the checkpoint in memory will be much lower than a stable storage system (e.g. disks). The low checkpointing overhead and faster restart should allow us to achieve better performance than traditional checkpoint based schemes, in spite of the higher forward path cost of our protocol as shown in Fig 1. Our solution of course has a higher memory overhead than the other possible solutions. This overhead might be reduced by paging out the checkpoint and memory logs to local disk.

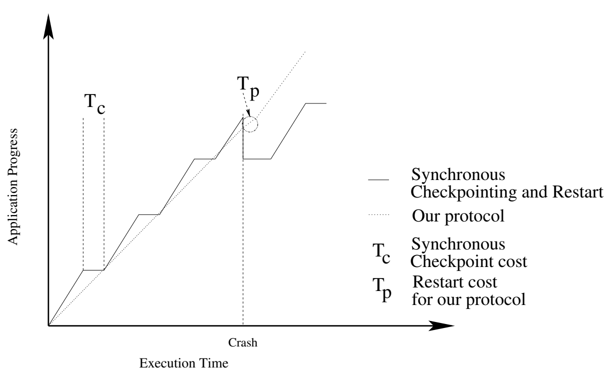

Figure 1. Desired performance of our protocol

Charm++[18] was decided on as the system for implementing this fault tolerant protocol. The virtualization inherent in Charm++ automatically makes applications written in it latency tolerant. The fact that there are multiple virtual processors on one physical processor means that while one virtual processor waits for some message, other virtual processors can keep on computing. Moreover Charm++ already supports the migration of virtual processors. These *virtual processors* are referred to in the charm literature as *chares*. The terms are used interchangeably through out the rest of the paper. Unlike other systems, we don't need to checkpoint the entire process image on a physical processor. The state of a Charm++ process can be represented by the state of the chares residing in that process.

AMPI [16] is an implementation of MPI written on Charm++ using Charm++ array elements. Once Charm++ is fault tolerant, AMPI would automatically become so since it would run as just another program on top of Charm++. This would provide a transparently fault tolerant version of AMPI.

The definition for a correct pessimistic log protocol is presented in [6]. It also presents a sketch of the proof that any pessimistic log protocol is fault tolerant to a crash with recovery and rollback to unsynchronized checkpoints. We apply the same definition to a Charm++ based system to come up with a set of requirements that need to be met in order to guarantee correctness.

The logging criterion, mentioned in [6], means that all the events on the crashed processor, between the last checkpoint and the state before the crash, should be re-executed in the same sequence as before. For Charm++, it means that all the chares residing on the crashed processor must re-execute their messages in the same order as before.

During the recovery phase, the chares on the crashed processor may send out messages that have already been received and processed by the chares on the other processors. These duplicate messages must be ignored to satisfy the silent re-execution property defined in [6].

5 Protocol

Next, we describe the details of the sender based message logging protocol we designed.

5.1 Data Structures

The implementation of our protocol required the addition of some data structures to Charm++. Every virtual processor is assigned a unique *id*. Each message has a *sequence number (SN)*, which is the number of messages sent by the sender to the receiver until then. A *ticket number (TN)* is also attached to each message. The receiver assigns a TN to a message and processes messages in the increasing order of their TNs.

Each physical processor maintains a *checkpoint* of all the resident chares in the memory of a buddy processor. Each message sent by a chare, is stored by it in a message log along with the TN of the message. If messages are sent between two chares on the same processor, they get logged in a *local message log* on the buddy processor.

Each virtual processor maintains a table of the highest SN sent to different virtual processors,called SNTable. It also maintains a sliding window of the SNs received from different virtual processors. This is used for duplicate message detection. A single variable *TCount* storing the highest TN returned by a chare is also maintained. A chare maintains a table of TNs that it has handed out since the last checkpoint. It is indexed by the sender and SN of the request. It also marks whether the message corresponding to the TN has been received.

5.2 Message Logging Protocol

A virtual processor A sending a message to a virtual processor B must perform the steps specified in Fig. 2.

```
X Sends message m to Y
    m.SN = SNTable[Y]
    if(Y is on same physical processor as X){
        get TN(m) from local chare Y
        m.TN = TN(m)
        send sender,m.SN,m.TN to buddy for logging
        wait for ack from buddy processor
        send m to Y }
    else{
        store m in message log
        send Ticket Request along with m.SN to Y
        wait for TN(m)
        if (TN(m) is marked old){
            remove m from message log
            return }
        if (TN(m) is marked received){
            set m.TN = TN(m) in message log
            return }
        set m.TN = TN(m) in message log
        retrieve m from message log
        send m to Y }
```

Figure 2. Algorithm for sending a message

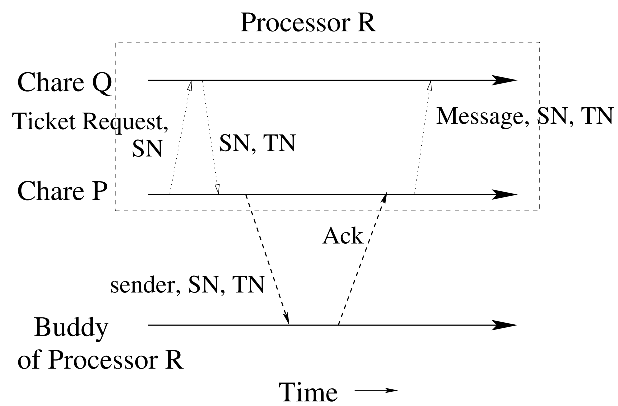

Figure 3. Message between local chares

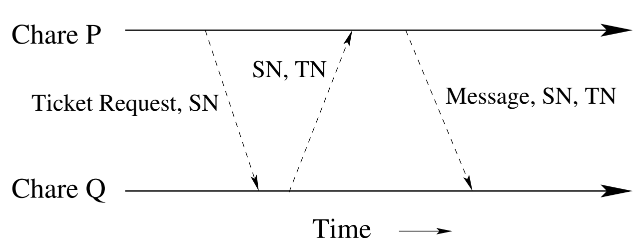

Figure 4. Message between remote chares

Messages to Local Chares: Virtualization means that multiple chares might be mapped to the same physical processor. Two chares on the same physical processor are referred to as being local to each other. A message sent to a local chare has its log and receiver on the same physical processor. If this processor crashes, all trace of the message is lost from the system. Although the message will be regenerated when the sender recovers, by the logging criterion mentioned in Section 4, it needs to be processed by the receiver in exactly the same sequence as before. Since sender and sequence number can identify a message uniquely as long as the correctness criteria are met, only sender id, SN and TN need to be logged.

TN(m) from local chare Y can be fetched by a method invocation. The local message with the ticket number is then sent to the buddy processor for logging. Only after receiving the buddy's acknowledgment can the processing of the message begin. The series of messages exchanged for sending a local message is illustrated in Fig. 3. As a result of logging on a remote processor the latency of a message to a local chare becomes the same as that of a message to a remote chare.

The term *wait* in the algorithm doesn't mean that execution actually stops there. Rather the chare remembers that it is waiting for a certain message and continues execution. Whenever the awaited message comes, the message-sending algorithm is resumed after the wait statement. This is true for all the following algorithms.

Messages sent to Remote Chares: Two chares on different physical processors are said to be remote to each other. In this case the protocol behaves in its default mode and exchanges messages as shown in Fig. 4. It logs the message, makes the ticket request and waits for it to come back. When a chare receives a ticket request with a certain SN, it looks up the <sender, SN> tuple in the table containing TNs. If it finds the SN there, it returns the value stored in the table. If it finds that the message for this SN has already been received, but after its last checkpoint it marks the TN as received. If it finds that the message was received before its last checkpoint it marks the TN as old. If none of the conditions are satisfied it means that this is a request for a new message. It increments the TCount and returns it as the TN for this SN to Y. It also adds the entry for this sender, SN and TN tuple to the table. A TN marked received means that the message need not be sent to the receiver unless it restarts. So the sender simply adds the ticket number to the message in its log. The message corresponding to a TN marked old need not even be logged. If the TN is new, it is assigned to the corresponding message in the log and the message is sent to the receiver. This table also helps deal with the case when a sender X resends a ticket request. If Y were to hand out a new TN to X for this request, Y would never receive a message with the old TN. Since messages are processed in the increasing order of their TN this would stall execution of Y at the old TN. However if the old TN were handed out for the same SN then this could be avoided.

The time difference between when X starts sending m and Y starts sending a message of its own, while processing m, is increased by a short message round trip time in not only our protocol but also those in [17] and [7].

5.3 Checkpointing Protocol

Each physical processor periodically decides to checkpoint its state. A processor, say P packs up the state of all the virtual processors on it and sends them to the buddy processor say Q. Each local chare on P also notes the message with the highest TN that it has processed. The table, storing the TNs sent to different processors, can be garbage collected by removing entries corresponding to messages that have already been processed.

On receiving the checkpoint, Q replaces its old copy of P's checkpoint with the new one. It then sends an acknowledgment to P. After receiving the acknowledgment, each virtual processor on P sends out a garbage collection message, containing the highest processed TN, to all the virtual processors that sent it a message since P's last checkpoint. After chare Y on physical processor R receives a garbage collection message from a chare X, Y removes all those messages to X in its message log that have a TN lower than that specified in the message. P sends another similar garbage collection message to its buddy Q, to remove old entries in the local message log.

There is an interesting trade off between memory and speed involved in deciding the checkpoint period. If the checkpoint period is too low, the message logs on senders are small saving memory but the time cost of checkpointing is higher. If the period is too high, the message logs on senders become large though the checkpointing cost is lower. Moreover infrequent checkpoints would make the restart slower because of larger number of old messages being resent. Thus the number of expected failures is also an important factor in deciding on the checkpoint period. Moreover some applications might be memory constrained whereas speed might be more of an issue for others. Thus the checkpoint period might be taken as a user input or might be decided on adaptively. A demand driven scheme for checkpointing might also be used.

Unlike some protocols like [7] our scheme doesn't rely on multiple reliable servers for checkpointing. Instead it depends on a processor and its buddy not failing within the same checkpointing period.

5.4 Recovery Protocol

The recovery protocol is more involved in our case than [17] because of the presence of multiple chares on the crashed processor. The steps involved in the recovery are discussed in chronological order.

The recovery protocol is initiated by the crash of a physical processor, say R. The crash detector detects the crash and restarts a Charm++ process on one of a pool of extra processors. After the Charm++ process corresponding to R has restarted on a spare processor, it asks its buddy Q for the checkpoint. Then Q sends R its checkpoint and the local message log. R reads in the checkpoint data for all the chares that were resident on it and stores the local message log. R broadcasts that it is ready to receive the logged messages.

As a response to the broadcast, all processors retransmit logged messages, that have a TN attached, to chares on R. For any logged messages without TNs, a ticket request is sent out. Each processor also sends a message containing the highest TN that it has seen for each chare on R. Chares reject any duplicate messages that they receive. Once a chare knows the maximum ticket number it handed out before the crash, it can start handing out tickets again. If a local message is generated during recovery, the local message log from Q is used to find the ticket number.

5.5 Correctness

We now sketch a proof that our protocol satisfies the requirements of a pessimistic logging protocol from Section 4.

Logging Property: All messages processed by a chare are always logged on the sender. Since the message logs are a part of the sender's state, even if it crashes, the logs can be restored from the checkpoint or recreated during the recovery phase. Moreover, all messages are processed in the increasing order of TNs. After a restart, all old messages are processed in the same sequence as before. The local message log was necessary to meet the logging criterion for local messages.

Silent Re-execution property: Each virtual processor maintains a window-like data structure, to keep track of the different SNs it has received from other virtual processors. By the logging property and the PWD assumption, after a restart, messages have the exact same SN as before. So, chares on uncrashed processors can detect duplicate messages and thus satisfy the silent re-execution property.

The only case, in which our protocol might not meet the correctness criterion, occurs when the checkpoint of a crashed processor becomes unavailable. When processor X, the buddy of Y, crashes and restarts, the checkpoint of Y is lost. Now if Y crashes before checkpointing, there won't be a valid checkpoint in the system from which to restart processor Y. It is simple to reduce the probability of such a pair of crashes happening. As soon as X restarts, Y checkpoints. This would reduce the time during which a crash might cause an irrecoverable error.

This scenario arises because unlike [6] or [7] we don't assumed any reliable nodes for logging or checkpointing. This removes an assumption that could be very costly to fulfill in a massively parallel system.

5.6 Reliability

We present a calculation based on a simple model to prove that our protocol increases the reliability of a system, in spite of not being foolproof.

Let the number of processors be n. Let failure rate of a single processor be λ. Let λ be the same for all n processors. Let the run time of the application without a fault tolerance protocol be R units. The probability of a particular processor crashing during the runtime of the application can be approximated by λR. The probability that the application will fail = $1 - (1 - \lambda R)^n$ (1).

Now, we consider the case when the application uses our fault tolerant protocol. Let the run time of the application in this case be R' units, where $R' > R$. The probability of any particular processor crashing during the runtime of the application = $\lambda R'$.

In our protocol, there is an upper bound on the time difference between two consecutive checkpoints taken by a processor. In order to simplify the analysis, we assume that the upper bound is t units of time for all processors.

The probability of an unrecoverable error, given that a processor has crashed, is λt. So, the probability of a particular processor causing an unrecoverable fault is $(\lambda t)(\lambda R')$ = $\lambda^2 t R'$. Therefore the probability of an unrecoverable fault during the application run is $1 - (1 - \lambda^2 t R' n)^n$ (2).

In order to bring out the huge difference between the expressions in (1) and (2), we evaluate them for some plausible system parameters.

Let the mean time between failures (MTBF) for any node be 20 $years$, n be 5000, and R be 400 $hours$. The MTBF of 20 $years$ yields a λ of 5.71×10^{-6} $per\ hour$. Plugging these values into (1) we get a probability of failure of 0.999989169. This means that for such a case, it is almost certain that the application will fail.

We conservatively assume that our protocol increases the application's runtime by a factor of 3. So R' = 1200 $hours$. Let each processor checkpoint every 6 minutes, i.e. t = 0.1 $hours$. So the probability of the application failing with our fault tolerant protocol running = 1.956×10^{-5}. Thus we see that our protocol decreases the probability of failure of an application from near certainty to a small chance.

6 Experimental Evaluation

The overhead introduced by the protocol as well as its performance in the face of failures was examined. The cluster consisted of 8 quad 500 Mhz Intel Xeon machines with 500 MB of RAM and 1 GB of swap space, connected by 100 Mbit switched ethernet, running Linux 2.4.20. All programs were compiled with GNU GCC, version 3.2.2, under the -O flag. Since this is an ongoing research project, the implementation of our protocol for Charm++ has not yet been extended to AMPI. As a result, we were unable to make use of the standard NAS parallel benchmarks for evaluating our performance. A simple 5-point stencil computation with a 1-D decomposition, was used to perform the initial evaluations of our protocol.

6.1 Fault tolerance Overhead

There are two major sources of potential overhead in our protocol. The first source is the extra message latency introduced by the message logging and ticketing protocols. The other source is checkpointing. We measure the overhead of message logging alone as well as that of the entire protocol to find out the amount of overhead caused by the two sources. The performance under these two conditions are compared with the same program running on a non fault tolerant version of Charm++.

Figure 5 shows the performance of an application in which the ratio of communication to computation is low. Such an application can tolerate an increase in latency well, especially in Charm++. It is evident that the overhead in such an application is minimal. The fault tolerant protocol actually outperforms the baseline in some cases. The application runs for approximately 300 seconds and checkpoints every 100 seconds in this example.

Figure 6 compares the performance of our protocol with the baseline, when the communication to computation ratio is high. This is simulated by executing the application with a smaller data set than in the earlier case. This results in the communication time becoming an important factor in the execution time. This figure shows that there is

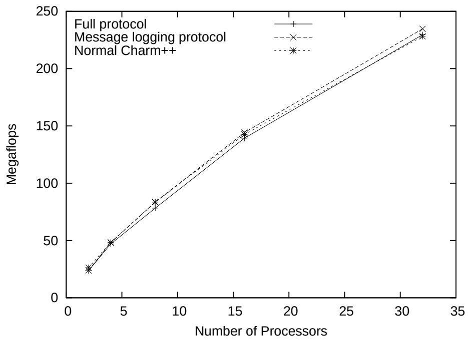

Figure 5. Low communication application

very little difference between our protocol with and without checkpoints. So checkpoints don't appear to be an important source of overhead. This is of course dependent on the checkpoint period used.This figure shows a substantial forward path overhead for our protocol, in the case of an application with a high communication to computation ratio, similar to those seen in [6] and [7]. The virtualization in Charm++ will surely help us better hide the latency as shown in [16]. All figures show executions with just 1 virtual processor per physical processor since the support for the fault tolerant migration of virtual processors is not yet complete. We expect to reduce the overhead significantly, even for applications with high communication to computation ratio, once the implementation is complete.

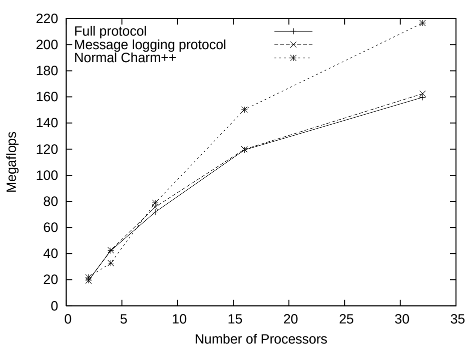

Figure 6. High communication application

6.2 Recovery performance

This section shows the performance of our protocol in the face of multiple failures. The application is run on 8 processors in a configuration with low communication to computation ratio. There is about 16 MB of data on each processor, which is checkpointed every 30 seconds. Failures are simulated by killing one of the Charm++ processes randomly. The crashed process restarts immediately and starts the recovery protocol specified in Section 5.4. Fig. 7 clearly demonstrates that the recovery cost of our protocol is low. Even with 7 faults the execution time doesn't increase the execution time more than 50%. So, our protocol behaves well even in the presence of multiple failures.

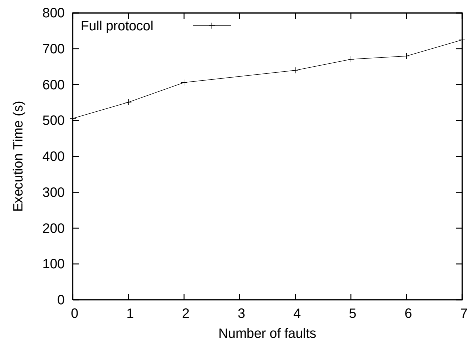

Figure 7. Performance with faults

7 Summary and Future Work

We described a scalable protocol for fault tolerance for parallel applications. The protocol builds upon previous work in this area, but unlike some other approaches, does not assume any completely reliable component. It was devised with the aim of minimizing the impact of a failure on the application progress, especially on a very large number of processors. It is implemented in the Charm++ system, which is a powerful parallel programming system that supports dynamic load balancing via processor virtualization, and automatic adaptive overlap of communication and computation, and has been the basis of several applications including the Gordon-Bell award winning NAMD program used routinely by biophysicists. Via Adaptive MPI (AMPI), implemented on top of Charm++, fault tolerance will be available for a wide collection of programs written in MPI.

The implementation described in this paper is an early implementation. We plan several optimizations and extensions. Firstly, we plan to extend the work to AMPI, and study performance on standard MPI benchmarks. A broader performance study of a wider class of applications, including focused studies of categories such as applications with low communication-to-computation ratio is also planned.

An interesting avenue of research is the checkpoint frequency. As discussed earlier, a method to adaptively decide on the checkpoint period could potentially reduce the overhead of our system. A method for staggering the checkpoints of different processors might avoid a potential network choke, caused by multiple processors checkpointing at the same time. Lazy checkpointing and incremental checkpointing optimizations can also be investigated.

So far, we have not taken advantage of processor virtualization supported in Charm++. We believe that this can be used to facilitate extremely fast recovery. As processors start blocking for messages from chares on the restarted processor, some of these chares can be migrated to the blocked processors. This would allow the restarted virtual processors to catch up with the rest of the computation much faster. Further, virtualization can also be used to hide the latency created by the ticketing protocol. The increase in message latency could also be alleviated by using high priority short messages for acquiring the tickets. Interrupting messages supported in Charm can be used for this purpose.

Finally, We also aim to use our system on some truly large machine and use our fault tolerant Charm++ or AMPI to run applications for long periods. We will do this in two parts: extending and testing our current system on larger clusters, and testing the system using a simulator [24] for large machines that we are developing .

Acknowledgments

We would like to thank Gengbin Zheng, Orion Lawlor, Chao Huang and Sameer Kumar for all their help and insightful comments, without which this work would not have been possible. This work was supported in part by the National Science Foundation (NGS 0103645, ITR 0081307).

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, pages 167–176. IEEE, 1999.

[2] L. Alvisi. Understanding the message logging paradigm for masking process crashes. Technical Report TR96-1577, 1, 1996.

[3] 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.

[4] J. P. Banatre, M. Banatre, and G. Muller. Ensuring data security and integrity with a fast stable storage. In *Proceedings of the fourth Conference on Data Engineering*, pages 285–293, February 1988.

[5] 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.

[6] 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.

[7] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *SC 2003*.

[8] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. In *Cluster 2003*, 2003.

[9] 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.

[10] 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.

[11] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Transactions on Computer Systems*, pages 3(1):63–75, February 1985.

[12] Y. Chen, K. Li, and J. S. Plank. CLIP: A checkpointing tool for message-passing parallel programs. 1997.

[13] E. N. Elnozahy. *Manetho: Fault-Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication*. PhD thesis, Rice University, October 1993.

[14] M. 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.

[15] G. Fagg and J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in Dynamic World. In S. Verlag, editor, *Euro PVM/MPI User's Group Meeting*, pages 346–353, Berlin, Germany, 2000.

[16] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *LCPC*, College Station, Texas, 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. Kale 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.

[19] 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.

[20] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th IPPS* , Honolulu, Hawaii, 1996.

[21] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[22] Y. Tamir and C. Equin. Error recovery in multicomputers using global checkpoints. In *13th International Conference on Parallel Processing*, pages 32–41, August 1984.

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

[24] G. Zheng, G. Kakulapati, and L. V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *2004 IPDPS Conference*, Santa Fe, New Mexico, April 2004.