FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI

Gengbin Zheng, Lixia Shi, Laxmikant V. Kalé

Department of Computer Science
University of Illinois at Urbana-Champaign
E-mail: {gzheng, lixiashi, kale}@cs.uiuc.edu

Abstract

As high performance clusters continue to grow in size, the mean time between failure shrinks. Thus, the issues of fault tolerance and reliability are becoming one of the challenging factors for application scalability. The traditional disk-based method of dealing with faults is to checkpoint the state of the entire application periodically to reliable storage and restart from the recent checkpoint. The recovery of the application from faults involves (often manually) restarting applications on all processors and having it read the data from disks on all processors. The restart can therefore take minutes after it has been initiated. Such a strategy requires that the failed processor can be replaced so that the number of processors at checkpoint-time and recovery-time are the same. We present FTC-Charm++, a fault-tolerant runtime based on a scheme for fast and scalable in-memory checkpoint and restart. At restart, when there is no extra processor, the program can continue to run on the remaining processors while minimizing the performance penalty due to losing processors. The method is useful for applications whose memory footprint is small at the checkpoint state, while a variation of this scheme — in-disk checkpoint/restart can be applied to applications with large memory footprint. The scheme does not require any individual component to be fault-free. We have implemented this scheme for Charm++ and AMPI (an adaptive version of MPI). This paper describes the scheme and shows performance data on a cluster using 128 processors.

1 Introduction

Parallel simulations in science and engineering often run for several hours or even days at a time. If a processor crashes during the run, all the computation until then would be wasted. To guard against this possibility, application programmers often write code for periodically checkpointing the state of the application to the disk. In addition to tolerating failure of individual processors, such checkpointing to stable storage also serves the purpose of saving the state of the application for planned shutdowns.

In recent years, parallel machines with very large numbers of processors have been designed and the trend toward massively parallel systems continues to increase. Examples of such machines are Earth Simulator, System X, ASCI-Q and BlueGene/L [?]. The size of the machines in the largest systems is now on the order of 10,000 processors. As the number of processors on present day supercomputers increases, the probability of one of the processors crashing during simulation increases substantially. This makes it more difficult to complete long-running jobs without facing faults from both hardware and software. For such failures, it is desirable to have the system automatically recover from them and continue the execution of the program without user intervention.

The requirements of fault-tolerant parallel applications have changed [10]. In the past, most applications that needed fault tolerance were mission-critical applications. These applications' primary concerns are continuous availability as well as the ability to tolerate arbitrary failures. The associated costs and the overhead induced by the fault-tolerance techniques are often not the primary concern. However, most of the emerging parallel applications are not necessarily mission-critical and thus don't require foolproof fault tolerance. They desire the fault-tolerance techniques which impose minimal overhead on failure-free execution and provide fast recovery from common case failure scenarios.

In a disk-based checkpoint/restart scheme for fault tolerance, the state of an application is saved to reliable storage periodically. Some traditional approaches let the entire application be killed once a failure occurs. The user has to experience a "down time" until he/she finds another allocation of both time and machine (in a job scheduling environment) to continue the execution from the latest checkpoint. This prolonged running cycle is impractical when the probability of failure is high.

Most of the traditional checkpoint-based fault tolerant protocols assume the availability of a pool of extra processors that can be used to replace the crashed ones at recovery. This is not practical especially in a job scheduling environment. However, it is challenging to let the program continue to run on the remaining processors with sustained performance. Load imbalance due to losing one processor may show great impact on the overall performance.

The performance metrics for a desirable fault-tolerant scheme include:

1. The run-time system does not rely on any fault-free component such as stable storage.
2. Impact on fault-free run time: how much the application slows down when there are no faults.
3. Recovery time: how long it takes for the system to restart the application after a processor crashes.
4. Execution efficiency after restart: the speed at which the application runs after it has lost one processor.

In this paper, we propose a fault tolerant protocol based on double in-memory checkpoint/restart and the idea of processor virtualization and migratable objects. The protocol does not assume any reliable storage for checkpoints. The restart protocol supports both cases with and without extra processors. When there is no extra processors to replace the crashed ones, the program can continue to run on the surviving processors. The impact of losing processors (load imbalance due to crash) is kept minimal by the capability of automatic load balancing at run-time. The scheme can be applied to a wide class of applications written in both message passing paradigms (MPI) and message driven languages such as Charm++ [18].

We will demonstrate that the scheme we present does very well on these criteria with some performance results on a cluster using 128 processors. The rest of the paper is organized as follows. Section 2 discusses some background and related work. The design of our system is presented in Section 3 and details of its implementation in Section 4. Performance results are provided in Section 5. Finally, Section 6 summarizes the contribution of our approach and thoughts for future work.

2 Background

A range of possible solutions for fault-tolerance have been extensively studied in the literature [12]. The two major classes of solutions are checkpoint-based and log-based rollback-recovery schemes.

2.1 Checkpoint-based methods

In checkpoint-based methods, the state of the computation as a checkpoint is periodically saved to *stable storage*, which is not subject to failures. When a failure occurs, the computation is restarted from one of these previously saved states. According to the type of coordination between different processes while taking checkpoints, checkpoint-based methods can be broadly classified into three categories: uncoordinated checkpointing, coordinated checkpointing and communication-induced checkpointing.

In *uncoordinated* checkpointing, each process independently saves its state. During restart, these processes search the set of saved checkpoints for a consistent state from which the execution can resume. The main advantage of this scheme is that a checkpoint can take place when it is most convenient. For efficiency, a process may perform checkpoints when the state of the process is small [25]. However uncoordinated checkpointing is susceptible to *rollback propagation*, the domino effect [21] which could possibly cause the system to rollback to the beginning of the computation resulting in the waste of a large amount of useful work. Rollback propagations also make it necessary for each processor to store multiple checkpoints, potentially leading to a large storage overhead. Due to the potentially unbounded cost of rollback, we consider 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 [24] and the hardware blocking used to take system level checkpoints in IBM-SP2, or non-blocking like Chandy-Lamport's distributed snapshot algorithm [8]. Coordinated checkpointing simplifies recovery from failure because it does not suffer from rollback propagations. It also minimizes storage overhead since only one checkpoint is needed. CoCheck [22], Starfish [1], Clip [9] and 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 [6].

The main disadvantage of coordinated checkpointing is the large latency involved in saving the checkpoints since a consistent checkpoint needs to be determined before the checkpoints can be written to stable storage. However, many scientific applications are iterative in nature, such as molecular dynamics simulation and finite element method simulation, and allow a consistent checkpoint to be taken between the iterations. In addition, at the end of an iteration the size of the global checkpoint state is often minimal.

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 based on protocol-related information piggybacked on the application messages it receives from other processors [5]. However, the forced checkpoint must be taken before the application may process the contents of the message, possibly leading to high latency and overhead. It does not scale well with increasing number of processors [2] and a large number of forced checkpoints nullify the benefit accrued from the autonomous local checkpoints.

Diskless checkpointing is a technique for checkpointing the state of a program on a distributed system without relying on stable storage. It replaces stable storage with memory and processor redundancy. Diskless checkpointing is a desirable alternative to disk-based checkpointing that can improve the performance of distributed applications in the face of failures. Diskless checkpointing often requires high memory overhead for storing checkpoints. The paper [20] presented a way to perform fast, incremental checkpointing by using $N + 1$ parity to alleviate this problem. The algorithm eliminates stable storage and disk writing by using a combination of extra physical memory and $N + 1$ parity. However, the protocol is difficult to scale to very large number of processors. When checkpointing, each processor needs to checkpoint its data (local address space) to the parity processor which could become a communication bottleneck. The recovery of one processor needs checkpoints from all other application processors as well as parity/backup processors, which prevents the protocol from applying to very large number of processors. The protocol also requires a number of extra processors for storing parity as well as processors to replace failed application processors.

The ability to checkpoint and restart applications has a number of other uses in a parallel environment besides fault tolerance. *Process migration* is one feature that is made possible by the ability to save a process image. A process can be moved from one node to another by writing the process image directly to a remote node. The process can then be resumed on the new node without having to kill the entire application and start it over again.

However, there are a number of disadvantages for process migration based fault-tolerance. First, saving the entire process image often incurs significant memory or disk space overhead and may be unnecessary since not all the data in a process space, e.g. temporary variables, needs to be saved. Second, in order to restore the process image after a failure, a new processor has to be available to replace the crashed processor. This requires a pool of standby processors for multiple unexpected failures which may not be practical. As an alternative, one may choose to restore the process image of the crashed processor on a running processor. This, however, may result in deteriorated parallel performance due to load imbalance created by overloading the processor.

2.2 Log-based methods

Message logging protocols are built on the assumption that the state of a process is determined by its initial state and by the sequence of messages it delivers. In principle, a crashed process can be recovered by restoring the process to its initial state and rolling it forward by re-playing to it messages in the same order they were delivered before the crash. In practice, message logging protocols limit the extent of roll-forward by having each process periodically save its local state in a checkpoint. Examples of log-based methods include MPICH-V [3], Manetho [11], [23], FT-MPI [13] and FTL-Charm++ [7].

Log-based methods may incur a fault-free runtime cost due to the logging of messages. It may also suffer from complicated recovery.

3 Design

One of performance metrics described in Section 1 is the impact of fault tolerance on normal run time. Since message logging exhibits constant cost in terms of message transmission latency, the checkpoint based protocol was preferred. The checkpoint scheme we designed imposes almost no overhead on normal execution when there is no fault. The coordinated checkpointing happens periodically. The time cost of checkpointing is very small as demonstrated in Section 5, and almost negligible for applications with low memory usage. Therefore it potentially allows more frequent checkpointings and hence wastes a less amount of useful computation when rollback occurs.

3.1 Double Checkpointing

Another design goal as described in Section 1 is that the run-time system should not rely on any fault-free component such as reliable storage. We adopted the idea of diskless checkpointing. Since the design mainly targets on the scientific applications with relatively small memory footprint on very large number of processors, we decided to use a simple double checkpointing scheme which is shown to be scalable (Section 5.1). In order to handle a fault at a time, each checkpoint data would be stored to *two different* locations. This *double-checkpointing* is to ensure the availability of one checkpoint in case the other is lost. In our scheme, we call the two processors that have identical checkpoints *buddy processors*. It should be noted that the double checkpointing scheme does not provide foolproof fault tolerance. However, the reliability can be substantially improved by our system as presented in Section 4.3.

In double checkpointing, checkpoints can be stored either in the memory or local disk of two processors. We call these *double in-memory checkpointing* and *double in-disk checkpointing* schemes. Unlike most protocols that store checkpoints to a separate central storage server across a network like MPICH-V2 [4], our schemes store checkpoints in a distributed fashion to avoid both the network bottleneck to the central server and the volatility of the central server. Our experiments in Section 5.2 manifest a very high cost of simultaneous checkpointing from all the processors to a shared stable disk (via NFS).

3.1.1 Double In-memory Checkpointing

In the double in-memory checkpointing scheme, each process stores its data to memory of two different processors. Since memory accessing is much faster than disk accessing, the potentially low checkpoint overhead and faster restart should allow us to achieve better performance than traditional disk-based checkpoint schemes. Therefore, we mainly focus on the double in-memory checkpoint scheme in this paper, although the implementation of the two schemes only differs in where to store checkpoints. In Section 5.2 we will present performance comparisons of both schemes.

Double in-memory checkpointing undoubtedly will increase the memory overhead. Our scheme provides several solutions for reducing the memory overhead: (a) instead of storing everything in a process including unused and/or temporary memory allocation, we allow a programmer to encapsulate the application data so that only the useful data is checkpointed [1]; (b) an application can choose to initiate checkpointing at a time when the memory footprint is small in the application. This can be applied to many scientific and engineering applications such as molecular dynamics simulations that are iterative; (c) for applications with very large memory footprint, the double in-disk checkpointing can be used.

3.1.2 Double In-disk Checkpointing

Double in-disk checkpointing scheme is another variation of the above in-memory checkpointing scheme in which checkpoints are stored on local scratch disk instead of in processor memory. Like the in-memory checkpointing scheme, it does not rely on any reliable storage due to the duplicate copies of checkpoints. Although it incurs much higher disk I/O overhead in checkpointing, it does not suffer from the dramatic increase in memory usage as in the double in-memory checkpointing scheme. It is useful for applications with very big memory footprint.

[1] small memory footprint applications may still choose to checkpoint the whole process image

Double in-disk scheme makes local disks useful for fault tolerance, whereas most traditional checkpointing schemes have to depend on a central reliable storage. Taking advantage of distributed local disks, the double in-disk checkpointing avoids the I/O bottleneck to the central file server. In fact, the experiments in Section 5.2 indicate that the double in-disk checkpointing scheme outperforms the traditional scheme that checkpoints to the central NFS disk.

3.2 Load Balancing

Unlike most other fault-tolerance schemes, our scheme does not assume the availability of processors to replace the crashed ones. Having a pool of extra processors is a convenient assumption for most fault-tolerance schemes. This is impractical in current job scheduling environments. Further, the number of extra processors reserved in the pool limits the maximum faults allowed during the execution and may be a waste of resources if faults occur rarely.

In our scheme, a program will continue to run on the remaining physical processors after a crash without coming to a full stop. In this scenario, one of the crucial performance issues is to minimize the impact of crashed processors on the execution so that the program continues to run at a speed not much slower than the pre-crash speed. A naive implementation could move all the work from the crashed processor to a running processor. However, this may result in unfavorable *load imbalance* and lead to poor parallel performance. This is a challenging issue for traditional methods of checkpointing that use the process image as checkpoint. Instead, we use finer-grained objects to encapsulate application data. These make it easier to perform load balancing by moving the objects around.

We decided to implement this scheme on Charm++ and Adaptive MPI, an MPI implementation based on Charm++, since Charm++ already supports parallel migratable objects at user level. In the next section, we will briefly describe the advantages of Charm++ run-time system for implementing fault tolerance.

3.3 Charm++ and Adaptive MPI

The basic mechanism in process migration that performs transparent migration of the internal and external process state is to provide applications with a *location independent* view of the world. However, due to the complex nature of the subject coupled with architecture dependent issues, process migration is limited in its usefulness.

Charm++ takes a different approach called *processor virtualization* [16]. Instead of taking the entire process as migration subject, it implements *parallel migratable objects*. An application divides a problem into a large number of components (N) (implemented as migratable objects) that will execute on P processors. N is independent of P though ideally $N >> P$. The user's view of the program consists of these components and their interactions; the user need not be concerned with how the components map to processors. The underlying run-time system takes care of this and any subsequent remapping (see Figure 1).

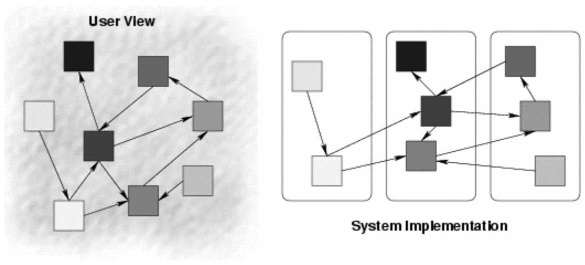


Figure 1. Virtualization in CHARM++

In Charm++, these components are known as *chares*. Chares are C++ objects with methods that may be invoked asynchronously from other chares. Since many chares can be mapped to a single processor, Charm++ uses *message-driven execution* to determine which chare executes at a given time.

Objects or chares that carry application code are location independent. Hence chares can migrate from processor to processor freely [2]. One application of migratable objects is load balancing. Objects can migrate from overloaded processors to underloaded processors to achieve better load balance. In order to migrate an object, one needs to pack the data of the object into a message. The message is sent to another processor where the data is unpacked and the object is restored. The Charm++ PUP framework was designed to describe the in-memory layout of an object [15]. The object migration based on the PUP framework can be extended into broader usage, such as migrating an object to disk at runtime for out-of-core execution. In the checkpointing scenario, when an object is checkpointed, it is simply packed and migrated to another location (memory or disk).

Adaptive MPI(AMPI) [14] is an MPI implementation and extension based on Charm++. AMPI implements virtual MPI processes, or VPs, using migratable user-level threads several of which may be mapped to a single physical processor. AMPI supports adaptive load balancing by migrating MPI threads. A fault tolerant AMPI has been implemented in traditional file-based checkpoint/restart scheme. MPI is a special case of AMPI when exactly one VP is mapped to a physical processor.

[2]Object migration does not have to deal with system kernel issues like interprocess communication (IPC).

4 Protocol and Implementation Details

In this section, we describe the double in-memory checkpoint/restart protocol we designed and briefly describe the implementation details. What we present here also applies to the double in-disk checkpoint/restart scheme. The same implementation works for both Charm++ and AMPI. In fact, in AMPI each migratable user-level MPI thread is simply treated as a Charm++ object (chare).

4.1 Checkpoint Protocol

We adopt a *coordinated* checkpointing strategy. All processors coordinate their checkpoints to form a consistent global state. Global state includes run-time system state (as virtual processor object) and user data which is encapsulated in objects (as *Chares* in Charm++). On each physical processor, there is one copy of the run-time system state with an arbitrary number of objects and their states. Each object has two buddy processors for checkpoints. The checkpointing process involves two concurrent steps: (a) each processor packs up its system state and sends it to two buddy processors. (b) each object packs up its own user data and sends it to two buddy processors.

Figure 2 illustrates an example of checkpointing as shown in the first row of processors before one processor crashes. Each circle represents an object in an application, while each square and triangle represent its first and second checkpoints on two buddy processors. It should be noted that one of the two checkpoints and the object can reside on the same processor to reduce communication overhead at checkpointing. For example, object *d* on processor 1 has two buddy processors 1 and 2. During checkpointing, object *d* only needs to sent its checkpoint across network to buddy processor 2, while the other checkpointing is done locally.

The checkpointing protocol has almost no overhead in the normal fault-free execution unlike that found in the message log-based scheme. The overhead only occurs during the coordinated checkpointing. Our scheme breaks down the whole process image into finer grained objects which provides flexibility for fast recovery. However, the checkpoint process in our scheme is communication intensive. Its checkpoint overhead is dominated by the network bandwidth. Since our scheme mainly targets scientific applications with small memory usage, this should not be a problem. In fact, our experiments in Section 5 show that even with an application that uses a total of about 1GB of memory, it takes less than one second on a Myrinet network and two seconds on 100Mbit network to finish the double in-memory checkpointing on 32 processors. Our scheme is also shown to be scalable [3] — the

[3]when application is decomposed in a load balanced fashion

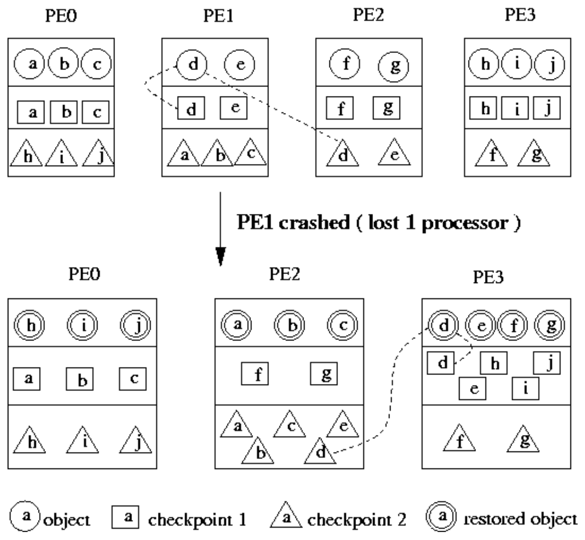


Figure 2. In-Memory Double Checkpoint/Restart Schemes

checkpoint overhead drops for an application when the number of processors increases. This is because checkpoint data size on each processor reduces when the number of processors increases.

4.2 Recovery Protocol

The recovery protocol is initiated by the crash of a physical processor. Every processor starts to rollback to the state preserved in the checkpoint. The recovery protocol is more complicated in our case because of the presence of multiple *Chare* objects on a crashed processor.

In this section, we only describe the recovery protocol when there is no replacement processor and skip the simpler case when there is replacement processor. The steps involved in the recovery are discussed in chronological order.

(1) A crash is detected when broken pipe errors occur on sockets used in the communication across processors. (2) The crash detector starts a "dummy" process on one of the remaining processors. This new dummy process does not carry any application data and checkpoint. It does not participate in any computation in the program. The only purpose of its existence is to replace the crashed processor in the processor-level spanning tree [4] used by Charm++ run-time. This dummy process can be eliminated in future, which will be discussed in Section 6. After the dummy process is started, it restores its runtime system data by sending a request to the processor having its checkpoint. The dummy process then broadcasts to invoke the parallel rollback protocol on every processor.

[4]This spanning tree is used for broadcast and reduction operations

As a response to the broadcast, all processors except the dummy processor start to perform the following substeps to rollback states to the recent checkpoints: (3) All *Chare* objects on the processors are removed which will be reconstructed from the checkpoints; (4) Restore double checkpoints. The lost checkpoints on the crashed processor are restored on other processors using the survived copies; and (5) *Chare* objects are restored from checkpoints. This is done by one of the buddy processors for each *Chare* object.

One issue in the recovery protocol is how to handle obsolete messages. These messages may be sent before the crash, in flight on the network, or buffered in a queue in the system. If they eventually show up, they will confuse the newly restored objects in the old states. Therefore, these messages need to be identified and ignored. In our scheme, we use an *epoch number* to denote the period of execution between consecutive faults. When a fault occurs, a new epoch begins and the epoch number is advanced by one. The sender processor timestamps each message with its current epoch number. On the receiver, only those messages with an epoch number that is not less than the current epoch number on the processor will be delivered.

The second row in Figure 2 illustrates a snapshot of objects on processors after a recovery is complete. After recovery, an object does not have to live on the same processor as the one before the crash. In fact, it does not matter where a *Chare* object lives in Charm++ because the messages directed to it can be automatically forwarded by the object manager efficiently [19]. Thus, an object is always locally restored by one of two buddy processors to avoid communication overhead. For example, object *d* in Figure 2 originally on crashed processor 1 has its new buddy processors 2 and 3 at restart, and object *d* is chosen to be restored by processor 3 locally.

Since both the buddy processors can be used to restore an object, some protocol needs to be established to avoid double restorations and at the same time create a balanced assignment of objects to processors. In our protocol, we use the rule that only the higher number buddy processor is responsible for performing the task. To avoid overloading the highest number processor, we use a wrap-around scheme so that processor number 0 is considered higher than processor $P - 1$ if there are P processors.

After recovery, *load imbalance* is very likely to occur since the restoration of objects to processors is determined without considering the load of each object. Therefore, some processors may have too many heavy objects and become overloaded which could dramatically slow down the entire execution. Load balancing is then desirable to migrate objects away from overloaded processors. Charm++ implements an automatic *measurement-based load balancing framework* [17] which dynamically monitors the load of the objects and performs load balancing based on load statistics. In our fault-tolerant protocol, the load balancer can be configured to automatically start shortly after a crash. The integration of fault-tolerance and load balancing in our system helps sustain the parallel performance even after a crash. The benefits of this post-crash load balancing is demonstrated in Section 5.3.

Our recovery scheme is very efficient and is not communication intensive. Steps (3) and (5) are performed locally on every processor. The only steps that involve communication across network are steps (2) and (4). In step (2), the communication overhead is low because the system data checkpoint is small (typically in the order of 10KB). In step (4), the recovery process of the lost checkpoints (belonging to the crashed processor) is also efficient because every processor holding the survived checkpoint can work individually to find a new buddy processor for the second checkpoint.

4.3 Reliability Analysis

In our fault-tolerance protocol, the only case in which our protocol might fail occurs when both an object's buddy processors crash during the time period between two consecutive checkpoints. In this section, we provide a calculation based on a simple model similar to [7] to show that our protocol increases the reliability of a system, in spite of being fallible.

Consider a parallel system with n processors. Let each single processor have a failure rate of λ and let λ be the same on all processors. Let the mean time between failure (MTBF) be M and let M be the same on all processors. The mean time between failure (MTBF) $M = \frac{1}{\lambda}$. Let the total execution time of an application without faults be R units. Thus, the probability that the application will fail is $1 - (1 - \lambda R)^n$ (1).

Now, consider the case when the application is running with our fault-tolerance protocol. Let the total run time of the application in this case be $R^{'}$ units, where $R^{'} > R$. Let C be the time difference between two consecutive checkpoints. For simplicity, ignore the probabilities of the cases when unrecoverable failures occur due to crashes of more than two processors. Let two buddy processors form a group giving a total of $n/2$ groups of buddies.

The probability of an unrecoverable error during C, given that a processor in a buddy group has already failed, is λC. So the probability that two processors in a buddy group crash during C is $(\lambda R^{'})(\lambda C) = \lambda^2 R^{'} C$. Therefore, the probability of an unrecoverable error during the execution is $1 - (1 - \lambda^2 R^{'} C)^{n/2}$ (2).

To get a better idea of the huge different between (1) and (2), we evaluate these two equations with some plausible system parameters.

To be optimistic, let the MTBF(M) for any node be 20 years. Let n be 5000, and R be 400 hours. So $\lambda = 1/M = 5.71 \times 10^{-6}$ per hour. Plugging these values into (1) yields a probability of failure of 99.9989%, which means almost certain failure for the application.

We assume that our protocol increases the running time of an application by a factor of 3, i.e. $R^{'} = 1200\ hours$. Let each processor checkpoint every 6 minutes, $C = 0.1\ hour$. Therefore, the probability of the unrecoverable failure with our fault tolerant protocol using (2) is only 0.000977%. Thus, our protocol, although not foolproof, decreases the probability of failure for an application from near certainty to a very unlikely chance.

5 Performance

We examine the overhead introduced by the protocol as well as its performance in restarting after failures.

Two major applications are used to perform the evaluations. One is a simple 7-point stencil computation with a 3-D decomposition (Jacobi3D) written in MPI; the other is a real world application — *LeanMD*, a molecular dynamics simulation program written in Charm++.

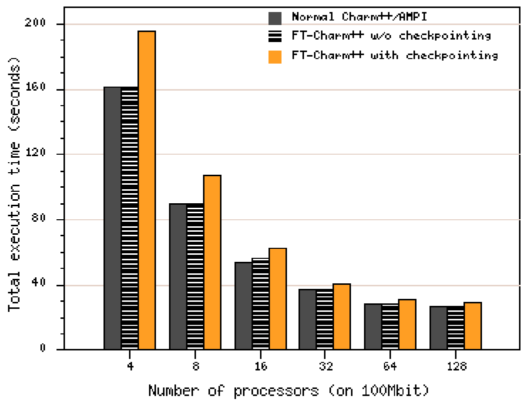
The cluster we used is NCSA Platinum IA-32 Linux Cluster. The cluster is comprised of 512 dual 1Ghz Intel Pentium III processors with 1.5GB of RAM connected by both Myrinet 2000 interconnect network and 100 Mbit Ethernet.

5.1 Checkpoint Overhead of Fault-tolerance

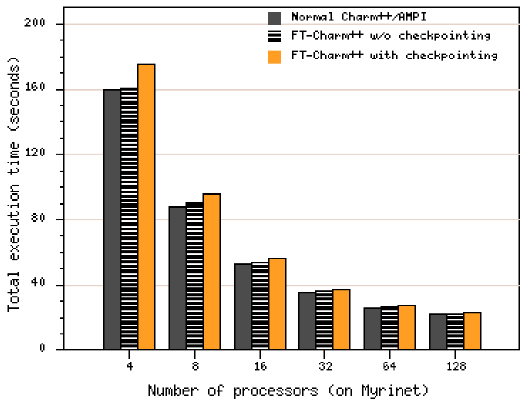
Some experiments were conducted to measure the overhead of adding the checkpoint/restart capability to Charm++ and AMPI. We expected the overhead in our protocol due to the fault-tolerance extension would be very small. In fact, for each message delivered in our protocol only an *IF* statement is introduced to check the *epoch number* (Section 4.2) of the message in order to filter those sent from pre-crash time.

We have run the Jacobi3D AMPI program under three scenarios: (i) with *normal* Charm++/AMPI without fault-tolerance extension; (ii) with fault-tolerant extension to Charm++/AMPI, but *without* actual doing checkpointing and (iii) with fault tolerant Charm++/AMPI and *with* 8 checkpointing steps. These runs were carried out on both Myrinet and 100Mbit networks from 4 to 128 processors. For all these runs, the problem size was fixed at 200MB, therefore the total checkpointed application data size in the entire program is also fixed at about 400MB (double checkpointing). The program ran with 100 steps.

Figure 3(a) shows the comparisons of total execution time for the runs on 100Mbit Ethernet. Figure 3(b) repeated the same execution on the faster Myrinet network. It is evident that the overhead of fault-tolerance alone



(a) Jacobi3D MPI on 100Mbit Ethernet



(b) Jacobi3D MPI on Myrinet

Figure 3. Checkpoint overhead on up to 128 processors

without doing checkpointing is minimal because of vertical bars of almost equal height in (i) and (ii). The time cost for 8 checkpointings in 100 steps was also reasonably small even on the slow 100Mbit network. To manifest the cost of the checkpointing, Figure 4 shows the time cost of a single checkpointing for the same run in (iii). Specifically, it took only 0.32 second for checkpointing with 100Mbit network and 0.089 second with Myrinet on 128 processors. Further, it can be seen that the checkpoint overhead decreases linearly when number of processors increases. This is because the application data to be checkpointed on each processor are reduced linearly [5]. Thus our checkpoint protocol is scalable.

5.2 Performance Comparisons with Traditional Disk-based Checkpointing

We have compared our protocol with the traditional disk-based checkpointing protocols under the following five scenarios: (a) checkpointing to a shared NFS drive, (b) checkpointing to each processor's locally mounted drive, (c) double in-memory checkpointing via fast Myrinet, (d) double in-memory checkpointing via 100Mbit Ethernet, and (e) double in-disk checkpointing via fast Myrinet.

Cases (a) and (b) are the traditional checkpointing schemes which store checkpoints to reliable disk storage and only one copy of checkpoint is saved. Note that (b) assumes every local disk in the system is reliable which itself may be difficult to fulfill in practice. Cases (c), (d) and (e) use our double checkpointing protocol. Cases (c) and (d) stores checkpoints in memory while (e) stores them in local disk.

[5]due to the fact that the global application data size remains the same

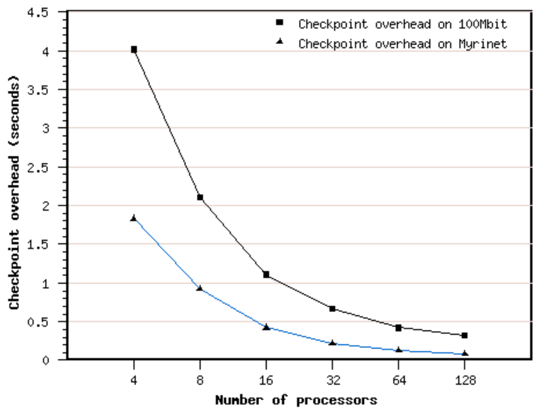


Figure 4. Overhead of one checkpointing (Jacobi3D MPI)

Figure 5 shows the time cost in checkpointing when problem size increases. We used the same jacobi program because it is easy to control the checkpoint size. The tests were carried out on 32 processors of NCSA Platinum cluster with 1.5GB memory each.

It can easily be seen that the checkpoint overhead increased almost linearly with the problem size (or checkpoint size) in all five cases except (a) up to at least about 6GB of total checkpoint data.

Among these runs, checkpointing to NFS drive (case (a)) incurred dramatically higher overhead due to the communication bottleneck to the file server. Checkpointing to local disk with traditional method (case (b)) performed much better than (a) however with the assumption that all

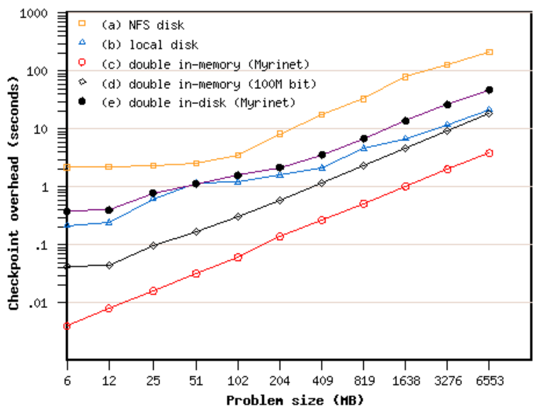


Figure 5. Performance comparison of in-memory vs. disk checkpointing

local scratch disks are reliable. Otherwise, the reliability of the system is determined by the disk failure rate since any single disk failure will result in an unrecoverable crash. This apparently cannot scale to massive parallel machines when the number of local disks in the parallel system is large.

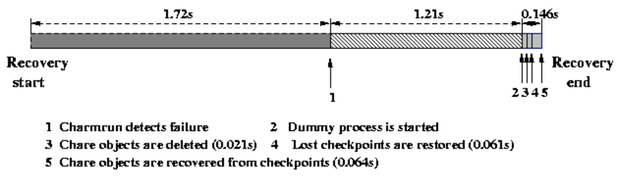
As to the three test cases of our fault tolerant protocol ((c), (d) and (e)), the two in-memory checkpointing schemes (c) and (d) performed the best. When taking advantage of a fast network like Myrinet, the checkpoint overhead is almost negligible in these tests. Specifically, in case (c) it only took about 4 seconds to perform the double in-memory checkpointing of total 6 gigabytes of application data on 32 processors. The protocol running with much slower 100Mbit network in case (d) also performs reasonably well. The checkpoint overhead was even less than the traditional disk checkpointing in (b). As expected, double in-disk checkpointing in (e) took about twice as much time as traditional local disk checkpointing in (b) since twice as much checkpoint data was written to disks.

This comparison of our schemes with traditional disk checkpointing schemes demonstrates that our double in-memory checkpoint protocol performs very well and is able to take advantage of fast network hardware. Even the double in-disk checkpointing scheme outperforms the traditional checkpointing scheme (a) that uses a central reliable file server. While our protocol is not infallible, it increases the reliability of a system dramatically (Section 4.3) and does not rely on any foolproof hardware which is either impractical or expensive.

5.3 Recovery Performance

This section shows the performance of our in-memory fault-tolerant protocol in the face of failures. Failures were simulated by killing one of the processes randomly.

The application used in the following tests was LeanMD, a molecular dynamics simulation program. Simulations were conducted using *Apoa1*, a 92,224 atom system benchmark. LeanMD generates 8498 parallel objects including 700 *Cells* (atoms cubes) and 7798 *CellPairs* (for force calculations). In each timestep of simulation, a Cell sends up to 14 messages to CellPairs. CellPairs perform force calculations and send the forces back. After each Cell receives up to 14 messages back from CellPairs, the Cell integrates the forces received and advances to the next step of the simulation. The checkpointing step is inserted after a timestep has finished. Due to the nature of molecular dynamics simulations, the memory footprint is very small [6] at this point. For this simulation, the checkpoint size for each processor is only about 400KB on each processor. We ran the simulation on 128 processors of NCSA Platinum cluster and the simulation consists of 600 timesteps.



1.72s
1.21s
0.146s
Recovery start
1
2 3 4 5
Recovery end
1 Charmrun detects failure
2 Dummy process is started
3 Chare objects are deleted (0.021s)
4 Lost checkpoints are restored (0.061s)
5 Chare objects are recovered from checkpoints (0.064s)

Figure 6. Time taken for each substep in a recovery

Figure 6 shows the time spent for each substep (described in Section 4.2) in a recovery. It is clear that a big portion of recovery time was spent in (1) and (2) in which the failure was detected and a new dummy process was started. The time cost of these steps depends on the operating system and tends to be a constant overhead for all applications. The time cost in (3), (4) and (5) are shown to be very efficient. This experiment shows that our recovery protocol is able to restart an application from a crash within a few seconds.

Figure 7 demonstrates the impact of crashes on the total execution time of LeanMD on 128 processors. In these runs, checkpointing happened for every 10 steps, and an automatic load balancing step was performed 5 timesteps after each crash. "Crashes" occurred randomly. Each point in the figure shows the total run time during which a number of processors "crashed". As illustrated in the figure, the total execution time was almost unaffected when

[6]CellPairs contain transient data used only when doing force calculations which does not need to be checkpointed.

one or two processors failed. Even in the test case when losing 10 physical processors (118 processors in the end), which was about one crash in every 40 seconds, the total execution time was not increased by more than 50%.

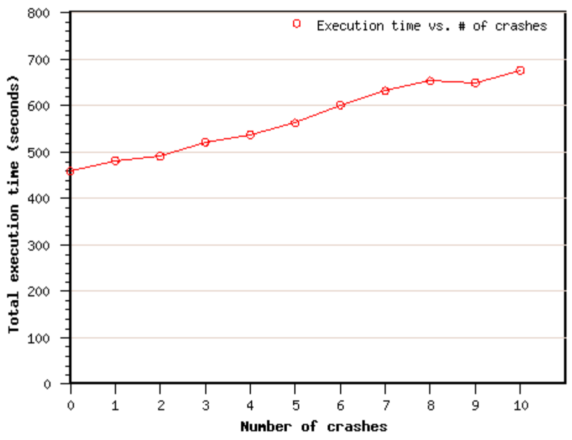


Figure 7. Run time with multiple crashes on 128 processors (in 10-crash case, one crash in about every 40 seconds)

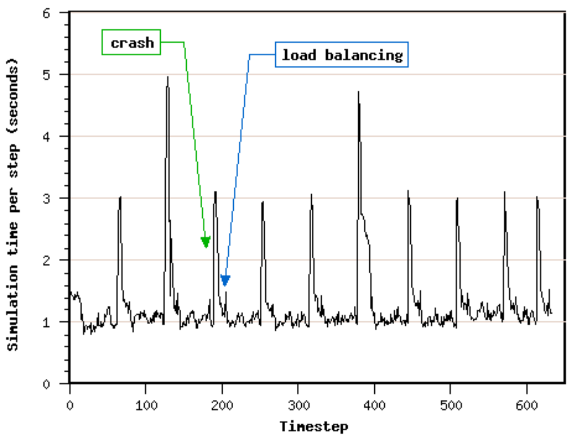


Figure 8. Simulation speed over timestep with 10 crashes and load balancing on 128 processors (crashes occur at spikes)

To better understand how the performance was dramatically improved by load balancing in the multiple failure scenario, we plotted Figure 8 with simulation time per step over each timestep for the same simulation that had 10 crashes on 128 processors. Due to rollback, the actual number of timesteps performed by this simulation is about 640 steps. Load balancing is triggered 5 timesteps after each crash. It is clear that after each crash and recovery, due to load imbalance, the simulation time per step increased dramatically (seen as a spike in the figure). After load balancing, however, the simulation time per step was brought down to a reasonable one. It also shows that with smaller number of processors left available, the simulation speed was affected very little and the simulation time per step increased very slowly. It demonstrates that our protocol integrated with load balancing capability provides a good solution for maintaining the execution efficiency even after losing physical processors in crashes.

6 Summary and Future Work

We presented a scalable protocol for fault-tolerance based on double in-memory checkpoint and restart for parallel applications. The protocol builds upon well-studied checkpoint/restart techniques in this area, but unlike some other approaches does not assume any completely reliable component. It implements a novel approach of automatically restarting an application from checkpoints without "down time". In addition, the scheme is designed for both cases with and without replacement processors. It allows a program to continue its execution after crashes on a smaller number of physical processors, without the unrealistic assumption of the availability of extra replacement processors. Most importantly, our scheme provides a solution for retaining the execution efficiency on the remaining processors after a crash. Our scheme is implemented in Charm++ and Adaptive MPI which allows fault-tolerance features to be available for a wide collection of applications, especially for scientific applications with relatively small memory footprint. It takes advantage of the idea of processor virtualization in migratable objects and automatic adaptive dynamic load balancing.

One extension of our scheme is double *in-disk* checkpoint/restart. It is useful for applications with very big memory footprint when the memory is not enough to hold both the application memory and checkpoint memory. Our performance data in Section 5.2 suggests that the time cost in writing checkpoints to local disk is reasonably low and is affordable.

Future work includes completely eliminating the dummy process created at recovery time. Since the only purpose of this process is to replace the crashed processor in forming the processor-level spanning tree used by Charm++ run-time, we should be able to reconstruct the spanning tree by skipping the crashed processor.

We aim to use our system on some extremely large parallel machines such as IBM Blue Gene/L. We will first test our system using a simulator [26] for large machines that we are developing, even before such a machine is built.

Acknowledgements

This work was supported in part by the National Science Foundation (NGS 0103645) and National Institutes of Health (PHS 5 P41 RR05969-04).

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, 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] 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.

[4] 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*, 2003.

[5] 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.

[6] 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.

[7] S. Chakravorty and L. V. Kale. A fault tolerant protocol for massively parallel machines. In *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.

[8] 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.

[9] Y. Chen, K. Li, and J. S. Plank. CLIP: A Checkpointing Tool for Message-passing Parallel Programs. 1997.

[10] Egida - lightweight fault-tolerance for distributed systems, 2003. http://www.cs.utexas.edu/users/vin/research/egida.shtml.

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

[12] 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.

[13] 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.

[14] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas, October 2003.

[15] R. Jyothi, O. S. Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.

[16] L. V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[17] L. V. Kale, M. Bhandarkar, and R. Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

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

[20] J. S. Plank and K. Li. Faster Checkpointing with N+1 Parity. In *24th Annual International Symposium on Fault-Tolerant Computing*, June 1994.

[21] B. Randell. System structure for software fault-tolerance. In *IEEE Trans. on Software on Software Engineering*, volume SE-1 (2), pages 226–232, June 1975.

[22] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

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

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

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

[26] G. Zheng, G. Kakulapati, and L. V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 78, Santa Fe, New Mexico, April 2004.