Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for AMPI and Charm++

Gengbin Zheng, Chao Huang and Laxmikant V. Kalé [*]
Department of Computer Science
University of Illinois at Urbana-Champaign

ABSTRACT
As the size of high performance clusters multiplies, the probability of system failure grows substantially, posing an increasingly significant challenge for scalability. Checkpoint-based fault tolerance methods are effective approaches at dealing with faults. With these methods, the state of the entire parallel application is checkpointed to reliable storage. When a fault occurs, the application is restarted from a recent checkpoint. However, the application developer is required to write significant additional code for checkpointing and restarting. This paper describes disk-based and memory-based checkpointing fault tolerance schemes that automate the task of checkpointing and restarting. The schemes also allow the program to be restarted on a different number of processors. These schemes are based on self-checkpointable, migratable objects supported by the Adaptive MPI (AMPI) and Charm++ run-time and can be applied to a wide class of applications written using MPI or message-driven languages. We demonstrate the effectiveness of the strategies and evaluate their performance.

1. INTRODUCTION
Parallel computing has been playing an increasingly important role in scientific and engineering research. Some problems, for instance rocket simulation, are too large to be solved by any single machine. Challenged by the need for higher computing capability, people are building larger parallel machines to handle the ever growing needs of parallel applications. Examples of such machines are ASC Purple and BlueGene/L [1]. Even with existing computational power, most parallel simulations still run for several hours or even days. As the size of new parallel machines multiplies, the probability of system failure increases substantially. As a result, it is almost certain for applications that run for a long time to face faults from both hardware and system software. Thus, finding fault tolerance methods to make sure that programs survive infrastructure failures has become an active research area.

Developing checkpoint and restart mechanism is an important effort toward fault tolerance. It provides the programmer with the capability to take snapshots of the state of the application (and sometimes of the whole system), periodically or on command. On occurrence of system failure, this checkpointed data would be used to restore the application to the previous checkpoint, and the forward progress of the application can resume from that point. Most of the traditional checkpoint-based fault tolerance protocols require the application developer to write significant additional code for both checkpointing and restarting, for chores like reading/writing disk files and re-initializing data structures. Moreover, it is typically assumed that the restart occurs on the same number of processors as when checkpointed. With this assumption, the user has to either wait a turnaround time for the restoration of the failed nodes, or rely on the availability of a pool of extra standby nodes that can be used to replace the crashed ones. In other words, traditional checkpoint-based fault tolerance entails substantial costs in programming complexity, time to solution and/or hardware resources.

In this paper, we describe and evaluate two checkpoint-based fault tolerance schemes: a simple on-disk scheme that writes checkpoint data onto reliable NFS hard drives (henceforth noted as the "on-disk" scheme), and another type of scheme that keeps two copies of checkpoint data, either in memory (called the "double in-memory" variation) or in faster local disk (the "double in-disk" variation). Both schemes require minimal programmer involvement for checkpointing or restarting, because the underlying run-time system is built on self-checkpointable and migratable parallel objects. Both schemes can restart parallel jobs on different number of processors, and hence do not rely on standby processors. It is even possible to restart the job on a larger number of processors to speed up the execution as more processors become available. As an option, both schemes can still use the standby processors as the replacement for the failed ones similar to other traditional fault tolerance schemes. We also describe our efforts in maintaining high overall utilization of the platform by supporting automatic load balancing of the parallel jobs, especially after they are restarted on a different set of processors with the old load balance destroyed. The fault tolerance schemes can be applied to a wide class of applications written in both message passing paradigms and message driven languages such as Charm++.

The rest of the paper is organized as follows. Section 2 discusses checkpoint-based fault tolerance efforts in general and some related work. Section 3 briefly introduces the Charm++ and AMPI run-time system — the infrastructure that the work of this paper builds on. The design of the two checkpoint-based fault tolerance schemes is presented in Section 4. Performance evaluation of the two schemes is provided in Section 5. Finally, Section 6 summarizes the

[*] {gzheng,chuang10,kale}@cs.uiuc.edu

contribution of our approach.

2. CHECKPOINT-BASED FAULT TOLERANCE

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 [23]. However, uncoordinated checkpointing is susceptible to *rollback propagation*, the domino effect [19] 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 [21] 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 [20] sits on top of message passing library and implements its functionality in its own MPI library tuMPI. A special process is used to coordinate the checkpointing, triggering the processors to save their states as well as incoming messages until all processors have finished doing so. At restart phase, receive operations need to first look at the saved messages for any match. This restricts when the checkpoint can be taken and sometimes may change MPIs semantics of synchronous communication. CLIP [10] is another project implemented on top of message passing paradigm and it is specifically built for Intel Paragon. They claim to be a semi-transparent mechanism because the user is expected to make minor changes to invoke the checkpoint procedure. Also it is the programmers responsibility to make sure that it is invoked at an appropriate time. Because a totally transparent implementation usually involves the operating system and can be very difficult to implement, this trade-off does make sense in many cases. A notable point is that CLIP is built on top of a compiler-based checkpointer libckpt [17]. A non-blocking coordinated checkpoint algorithm that uses application level checkpointing is presented in [7].

Coordinated checkpoint schemes suffer from 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. In most cases, a global synchronization is needed to determine such a consistent global state. Fortunately, many scientific applications, such as molecular dynamics simulation and finite element method simulation, are iterative in nature. A consistent global state can be easily identified between the iterations. In addition, such a checkpoint time often involves minimal checkpoint data because the intermediate computation data is freed or not essential to checkpoint.

Communication-induced checkpointing allows the 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 [6]. 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 the technique for taking a snapshot of 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. Authors of [18] 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. All N processors cooperatively maintain their local checkpoints of a consistent global state. A "checkpoint processor" and a "backup processor" are reserved for storing "parity checkpoint" which is calculated by applying XOR operation on all local checkpoints of each processor. When a processor fails, all surviving processors can be recovered to their previous local checkpoints in the memory, and the failed processor calculates its checkpoint from all the other checkpoints, and from the parity checkpoint. However, in this protocol, each processor communicates with the parity processor when calculating the parity, which might become a communication bottleneck. The recovery of the failed processor needs checkpoints from all other application processors as well as parity/backup processors, which is also communication intensive. The protocol also requires two extra processors for storing parity as well as standby processors to replace failed application processors.

3. CHARM++ AND ADAPTIVE MPI

Most traditional checkpoint-based fault tolerance schemes perform checkpointing at the process level which involves saving the entire process images [4]. These schemes are limited in their usefulness due to the complex nature of the subject coupled with many architecture dependent issues. They are also too inflexible to handle the scenarios where crashed processors are either irreplaceable or prohibitively expensive to replace. Allowing an application to adaptively shrink/expand the number of processors during execution, and dynamic load balancing are two desirable features of a run-time in order to better support a fault tolerance scheme in that scenario.

3.1 Charm++

Charm++ employs a novel approach called *processor virtualization* [16]. 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 although $N >> P$ is ideal. The user's view of the program consists of these N 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).

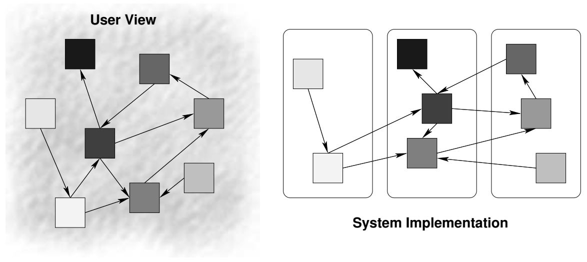

Figure 1: Virtualization in Charm++

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 and data are location independent. Hence chares can migrate from processor to processor freely. Object migration does not have to deal with system kernel issues like inter-process communication (IPC), therefore it is not architecture dependent. One application of migratable objects is load balancing. Objects can migrate from overloaded processors to underloaded processors to achieve better load balance.

The Charm++ load balancing framework [24] implements a dynamic automatic load balancing based on runtime instrumentation. During the execution of a Charm++ program, the load balancing framework collects workload information and object-communication pattern on each physical processor in the background, and at load balancing time, load balancer uses this information to redistribute the workload, migrating objects from overloaded processors to underloaded ones. Charm++ supports a range of sophisticated dynamic load balancing algorithms including greedy-based algorithms and refinement algorithms that improve the load balance by incrementally adjusting the existing object distribution.

In order to migrate an object, one needs to pack the data from the object into a serialized buffer and unpack on the destination. Charm++ provides a PUP (Pack-and-UnPack) framework to describe the in-memory layout of the object [15]. In the checkpointing context, the object is simply packed and migrated to the checkpoint storage (memory or disk).

3.2 Adaptive MPI

Adaptive MPI(AMPI) [14, 13] is an adaptive implementation and extension of MPI built on top of the Charm++ runtime system. AMPI implements virtualized MPI processes (VPs) using light-weight migratable user-level threads, several of which may be mapped to one physical processor.

Inherited from Charm++, the virtualization in AMPI brings various benefits to the traditional message passing programming paradigm. Beside automatic system-level fault tolerance, other benefits include dynamic overlapping of computation and communication, automatic load balancing, flexibility to run on arbitrary number of physical processors, and optimized communication library support. AMPI is now a mature implementation of MPI; it can be and has been used in real world applications, such as rocket simulations at the Center for Simulation of Advanced Rockets at UIUC and the FEM framework[5] in a dynamic 3D crack propagation simulation program. Thanks to the good portability of Charm++, AMPI is able to provide parallel programmers with productivity and performance on a wide range of high performance platforms.

4. DESIGNS OF TWO FAULT TOLERANCE SCHEMES

In this section, we summarize two designs of checkpoint-based fault tolerance schemes for Charm++ and AMPI. One scheme is based on on-disk checkpointing [12], and the other is based on double in-memory checkpointing [25]. Both schemes take advantage of the support for fault tolerance in the Charm++ and AMPI run-time system.

4.1 Run-Time Support

The rum-time supports checkpointing application data encapsulated in parallel objects and AMPI threads in two levels: fully automated checkpointing or flexible user-controlled checkpointing by additional helper functions.

Checkpointing objects (including AMPI threads) can be fully automated using *isomalloc stacks and heaps* [13]. It is portable on most platforms except for those where the *mmap* system call is unavailable. Similar to the idea in PM^2 system [3], the Charm++ memory allocator allocates data with a globally unique virtual address, that is, isomalloc reserves the same virtual space on all processors. When the memory allocator allocates memory in the context of an object, it records the pointer to the allocated data associated with the object. When the object is checkpointed, both the stack (for the user-level thread) and the heap data associated with the object are checkpointed automatically. Isomalloc also enables automatic restart. An object or a thread checkpoint can be restored on any processor because the isomalloc'ed data can be restored without changing its address.

Alternatively, users can write their own helper functions to pack and unpack heap data for checkpointing and restoring an object. This is useful when application developers wish to reduce the data volume by using application specific knowledge and/or by packing only variables that are live at the time of checkpointing. If the amount of data in a checkpoint is reduced, the checkpoint overhead can be cut down, and consequently more frequent checkpoints can be taken at the same total cost.

4.2 On-Disk Checkpointing

As described in Section 3, the Charm++ run-time system supports migratable objects (user-level threads in AMPI). During migration, the member data of the object (and the stack data of the user-level thread) is first packed on the source processor, then shipped to the destination processor, and finally unpacked before the execution on that object (thread) can be resumed. The simple on-disk checkpoint scheme is analogous to this object migration process. Imagine the source or the target of migration is instead reliable storage media like NFS hard disk drives. In the checkpoint phase, the object data is packed and saved onto hard disks as files, and the application continues its normal execution. At restart, objects are resurrected from the disk files and the parallel execution resumes from that point, as if the objects had just arrived at the processors.

To use this scheme, the programmer typically makes calls, periodically or on command, to checkpoint the application onto disks. When the application crashes, the user has to manually restart the program from the latest (or desired) snapshot. With isomalloc stack and isomalloc heap support[13], checkpoint/restart mechanism in AMPI can be fully automated, with no effort from the programmer or user needed at all. On the other hand, AMPI provides the potential for higher efficiency with some user involvement; the programmer can choose what data in the MPI task is useful or alive and worth saving. This makes sense because after all the programmer has the best knowledge about which part of the job needs to be saved.

Our on-disk checkpoint scheme is distinguished from other traditional disk-based checkpoint schemes in that our scheme is based on self-checkpointable objects implemented in Charm++ run-time. These location-independent objects give our fault tolerance mechanisms the freedom to recover checkpoint data encapsulated in migratable objects to any processors in the system. Therefore, the restart can occur on an arbitrary set of physical processors.

4.3 In-Memory Double Checkpointing

In-memory checkpoint scheme [25] adopted the idea of diskless checkpointing that checkpoints data in memory. It also uses a *coordinated* checkpoint strategy. In order to handle one fault at a time — a common case scenario, one checkpoint of the application state in the memory of a different processor is not sufficient as illustrated in Figure 2. In this 4 processor scenario, each Charm++ object (represented as a circle) checkpoints only one copy of its checkpoint (represented as a triangle). When processor 1 crashes, the checkpoints for objects a,b and c in the memory of that processor are permanently lost, so the application will fail. This suggests that each checkpoint needs be stored in the memory of two different processors. This *double-checkpointing* thus ensures the availability of one checkpoint in case the other is lost.

Figure 3 illustrates an example of double in-memory checkpoint and restart. The top half of the figure shows the scenario before one processor crashes. Each circle represents an object in an application, while each triangle and square represent its first and second checkpoints. In our scheme, we call the two processors that have the identical checkpoints *buddy processors*. It should be noted that one of the two checkpoints can reside on the same processor as the object which helps 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 send its checkpoint across network to processor 2, while the other checkpointing on processor 1 is done locally.

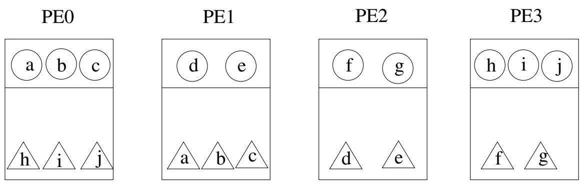

Figure 2: In-memory Single Checkpoint

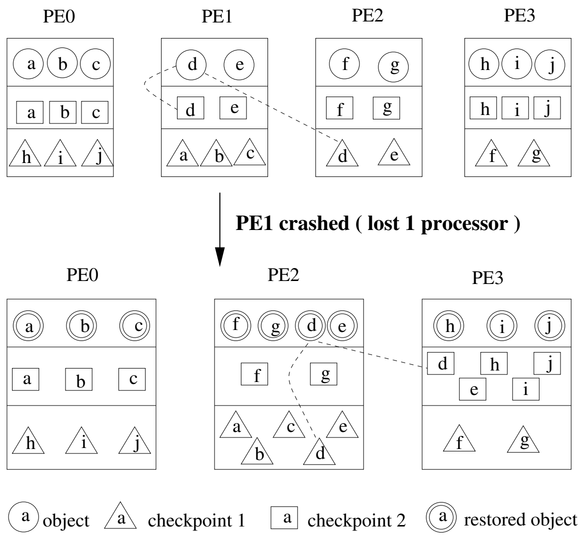

Figure 3: In-memory Double Checkpoint

Since accessing memory is much faster than accessing disk, the potentially low checkpoint overhead and faster restart allows us to achieve better performance than traditional disk-based checkpoint schemes. Sending checkpoint data to the memory of other processors takes advantage of the high speed interconnect, resulting in much lower overhead compared with on-disk checkpointing. With the distributed nature of the checkpoint protocol, our checkpoint protocol scales when number of processors increases.

However, double in-memory checkpointing undoubtedly increases the memory overhead. It is most beneficial to application paradigms with small memory footprints or machines with very large memory capacity. Another interesting variant of this scheme — *double in-disk checkpointing* — that checkpoints two copies of data to the local scratch disks of two different processors is a potential solution to the memory constraint. Double in-disk checkpointing fully utilizes the local disks as a distributed storage and uses redundancy to improve the reliability. Writing to the local disks of remote processors via high speed interconnect potentially yields much lower checkpointing overhead compared with writing checkpoints to centralized reliable disks.

Like the simple on-disk checkpoint scheme, this double checkpoint scheme is distinguished from other traditional checkpoint schemes in that it does not necessarily assume the availability of standby processors. Furthermore, the application can be restarted and continue to run on the surviving processors without halting the entire job as described below.

The restart procedure is initiated by the crash of a physical processor. On clusters, the crash detector in the run-time system detects the crash through broken pipe errors of sockets used in the command channel among processors. When the restart procedure is initiated, all surviving processors examine the surviving checkpoints in their memory and check for missing buddy processors. If the buddy processor is the crashed processor, a new buddy processor is chosen and the latest checkpoint is copied to that processor to maintain the double checkpoints. One of the two buddy processors then is responsible for restoring the corresponding object to the state in the checkpoint data.

The bottom half of the Figure 3 illustrates a snapshot of objects on processors after a recovery is complete. The lost checkpoints (d,e,a,b,and c) on the crashed processor 1 are recovered to processor 3 and processor 2 respectively. After double checkpoints are recovered, each object is locally restored by one of the two buddy processors having the checkpoints. For example, object *d* in Figure 3 originally on crashed processor 1 now has its new buddy processors 2 and 3 after restart, and processor 2 is chosen to restore the object locally. This improves the restart performance by avoiding the communication overhead incurred by the case when the object is restored on a processor other than buddy processors.

4.3.1 Reliability Analysis

The double in-memory checkpoint protocol not only tolerates one fault at a time, but also tolerates multiple faults as long as one copy of all the checkpoints survive. 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 summarize an analysis of the reliability based on a simple model [25] to show that our protocol greatly increases the reliability of a system.

Consider a parallel system with n processors. Let each 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 also 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 which gives 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 both 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 run time of the application by a factor of 3, i.e. $R^{'} = 1200\ hours$. It is difficult to estimate $R^{'}$ accurately because it includes not only the checkpoint/restart overhead, but also the extra time spent due to rollback and the slowdown due to fewer processors. 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 decreases the probability of failure for an application from near certainty to a very unlikely chance.

4.4 Load Balancing

For all these fault tolerance schemes, after recovery, load imbalance is very likely to occur especially when the application has to run on fewer number of processors. In this case, due to the fact that the restoration of objects to processors is determined in simple schemes such as round-robin without considering the load, the load may not be evenly distributed anymore. In a heterogeneous environment, even when a failed processor can be replaced by a standby processor but with a different CPU speed, load imbalance may also occur.

Our fault tolerance schemes let the load balancing step fix the load imbalance possibly introduced during restart. The Charm++ run-time lets the application run for a short period of time, monitoring and measuring the computation and communication load, and then automatically performs a global load balancing step to fix the load imbalance. This coupling of fault tolerance protocol with automatic dynamic load balancing framework in Charm++ is shown to be very effective in sustaining the parallel performance even after a crash. Section 5.2 demonstrates the benefits of such schemes.

4.5 Comparison and Discussion

Both our on-disk and in-memory checkpoint/restart schemes in AMPI and Charm++ can restart an application on a different number of processors. The application can shrink or expand accordingly to adapt to the changing environment with the capability of the dynamic load balancing by the run-time system. However, both schemes have their own

FT Protocols	Shrink/Expand	Portability	Fault-proof	Diskless	Halts Job	NFS Bottleneck
On-disk	Yes	High	Yes	No	Yes	Yes
Double In-memory	Yes	Low	No	Yes	No	No
Double In-disk	Yes	Low	No	No	No	No

Table 1: Comparison of Disk-based and Memory-based Checkpoint Schemes

strengths and target different application paradigms. Table 1 shows the comparisons of the on-disk and in-memory fault tolerance protocols in Charm++/AMPI.

The on-disk checkpoint scheme does not involve sophisticated online error detection and recovery support from infrastructure that in-memory double checkpointing is based on, and thereby it works on a wider variety of platforms. The in-memory checkpoint scheme and its implementation of error detection and recovery scheme however rely on the socket-based network version of Charm++, where an error is detected by broken socket pipe error and a new process is spawned if necessary on new processors.

The on-disk checkpoint scheme provides a fault-proof fault tolerance where the reliability of the system is essentially determined by the MTBF of the "stable" storage. The double in-memory checkpoint scheme greatly improves the reliability, and is able to handle all single faults and most double faults, except those involving failure of both buddy processors between two checkpoints. To tolerate more faults at a time, more duplications of checkpoints are needed, which requires even larger memory footprint and may not be feasible.

The in-memory checkpoint scheme is well suited for applications with a small memory footprint so that the checkpoints can be stored in memory. Such applications are prevalent, as typified by molecular dynamics simulations. It provides an attractive solution to fast diskless fault tolerance. It takes advantage of both the faster memory access and high speed network. The checkpointing of application data to another processor's memory can be easily sped up with high speed interconnect. With the distributed nature of the checkpoint algorithm, the scheme does not suffer from the NFS bottleneck, and is shown to be scalable with increasing number of processors. In comparison, the on-disk checkpoint scheme does not depend on the memory for checkpointing, so it works for any applications without imposing more memory overhead.

For applications with a large memory footprint, a variation of the double in-memory checkpoint scheme — double in-disk checkpoint scheme is useful provided that each node has local disk. Just like the double in-memory checkpoint scheme, it writes duplicate copies of checkpoints, however to local scratch disk instead of to the memory of each processor. Although this scheme involves higher overhead due to disk I/O, it does not suffer from the memory constraint as in the in-memory checkpoint scheme, and shares other benefits of the double in-memory checkpoint scheme. Compared with the on-disk checkpointing, double in-disk checkpoint scheme does not depend on central reliable storage. Instead, it takes advantage of the distributed local disks and thus avoids the I/O bottleneck that is common in today's central file systems.

5. EXPERIMENTS AND ANALYSIS

We evaluated and compared the checkpoint overhead and restart performance on several different platforms with a variety of applications.

5.1 Checkpoint Overhead

To illustrate the checkpoint overhead of our schemes, we perform our experiments with CG and FT in the NAS benchmarks, with class A and class B. The total amount of data for each of the 4 combination is different: CG class A has 50MB total checkpoint data, and data from CG class B is in the order of 100MB. FT has much larger amount of data to save, with FT class A having around 500MB and FT class B having nearly 2GB.

The platforms we used include CSAR's Turing cluster [22], a 1280-processor Apple G5 Xserve cluster connected with Myrinet fast network and 100 Mbit Ethernet, and EPCC's BlueGene/L machine[11]. This BG/L machine consists of 1024 compute chips (nodes) in a single cabinet. With each chip having two processors, it has a total of 2048 processors. The machine is configured with 512 MB of DDR memory per chip, shared between the two processing cores.

First we present results on the Turing Cluster in Figures 4, 5, 6 and 7, for CG class A, CG class B, FT class A, and FT class B. The x-axis is increasing number of processors, and the y-axis is checkpoint time in second. Both axes are in logarithmic scale.

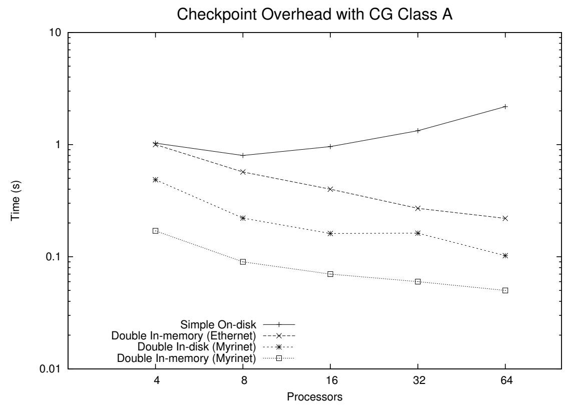

Figure 4: Checkpoint Overheads with CG Class A on Turing Apple Cluster

In each figure, we show 4 curves representing 4 series of runs with the benchmark. The on-disk scheme is usually

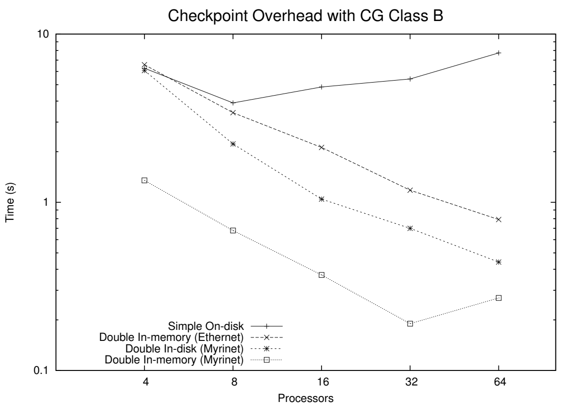

Figure 5: Checkpoint Overheads with CG Class B on Turing Apple Cluster

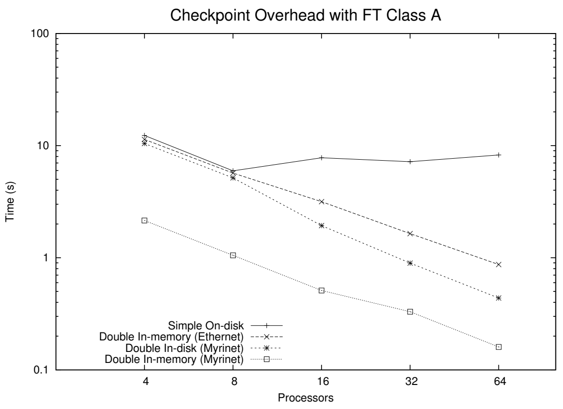

Figure 6: Checkpoint Overheads with FT Class A on Turing Apple Cluster

the most expensive in terms of checkpoint overhead. Thanks to RAID disks, it scales on 4 and 8 processors, but beyond that, the NFS bottleneck becomes the limiting factor, and the performance deteriorates as number of processors increases. The double in-memory runs utilizing the Ethernet and the Myrinet have similar scaling behavior, their curves almost parallel, both showing good scalability. The fourth curve shows the in-disk variation of double checkpoint protocol. As other double checkpoint variations, its performance is scalable, since it utilizes the interconnect to transfer data to save to peers and writes data onto local disks. Because writing to hard disks is not as fast as storing to peers' memory, we observe that the overhead of this protocol is always higher than its in-memory counterpart with the same interconnect. With lower overhead from in-memory schemes, we can checkpoint the program more often, and hence reduce the work lost since last checkpoint when a fault occurs.

There is one interesting point to note here. In Figure 7 with FT class B, the double checkpoint scheme with the Myrinet is unable to run on 4 processors, because the memory footprint for that specific benchmark (2GB) is relatively too large for the machine. We face similar problems on later runs on BG/L too. In this scenario, the user has two potential solutions. First, the user can use the in-disk variation

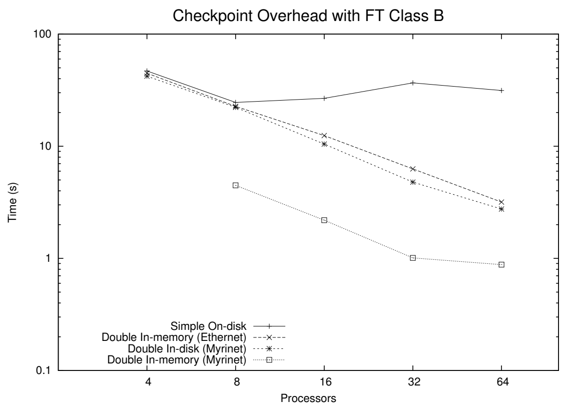

Figure 7: Checkpoint Overheads with FT Class B on Turing Apple Cluster

of double checkpoint scheme. However, when a local disk is unavailable, as on BG/L, the on-disk checkpoint can still serve the purpose. Moreover, when the socket error detection, based on which the double checkpoint scheme is built, is missing, the on-disk scheme becomes the only choice.

We repeated the tests with the NAS benchmarks (CG and FT Class B) on Blue Gene/L. Figure 8 illustrates the checkpoint overhead for the FT class B benchmark with both on-disk and double in-memory checkpoint protocol. Figure 9 illustrate the same checkpoint overhead for the CG class B benchmark. The x-axis is increasing number of processors, and the y-axis is checkpoint time in seconds which is in logarithmic scale.

Due to the fact that Blue Gene/L has only 512MB memory on each node, FT class B benchmark does not even run on 4 nodes. On-disk checkpoint scheme runs on the rest of tests. In comparison, several FT benchmark runs with double in-memory checkpoint protocol ran out of memory during checkpointing.

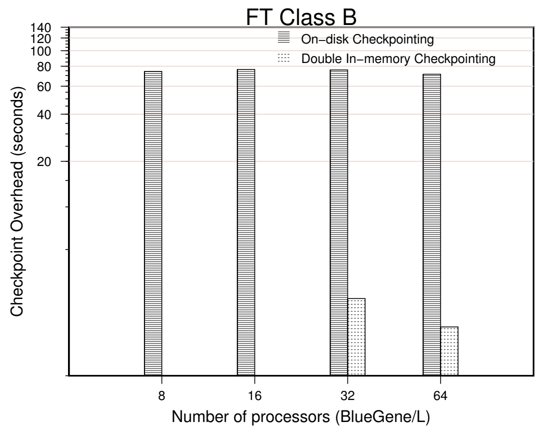

Figure 8: Checkpoint Overhead with FT Class B on Blue Gene/L

To examine how well our fault tolerance protocols scale with

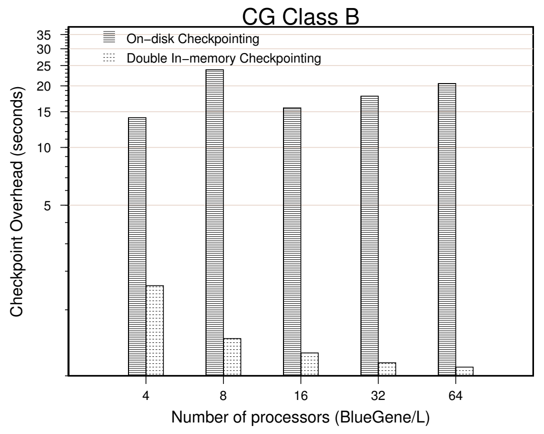

Figure 9: Checkpoint Overhead with CG Class B on Blue Gene/L

the data size, we run a test program with a simple 7-point stencil computation with a 3-D decomposition (Jacobi3D) written in MPI on the 32 processors of the Turing Apple cluster. This simple program is flexible in controlling the checkpoint data size, so we can measure the overhead of checkpointing varying the problem size.

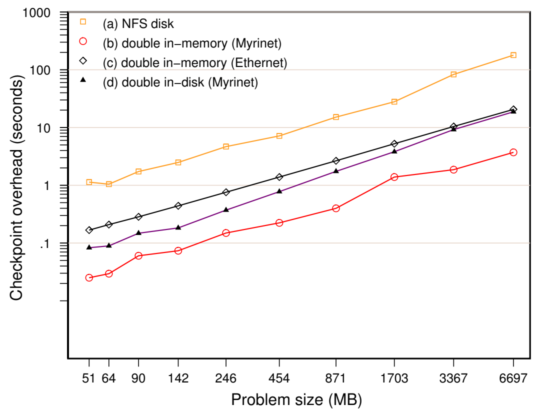

Figure 10: Performance Comparison of In-memory vs. On-disk Checkpointing Varying Problem Size on Turing Apple Cluster

Figure 10 shows the time cost (in logarithmic scale) with our 4 checkpoint schemes: (a) checkpointing to a shared NFS drive, (b) double in-memory checkpointing via the Myrinet, (c) double in-memory checkpointing via the Ethernet, and (d) double in-disk checkpointing via the Myrinet.

It can be seen that the double in-memory checkpointing on the Myrinet performs the best. The checkpoint overhead is almost negligible in the tests. It only took about 4 seconds to checkpoint two copies of 6.7 gigbytes of application data on 32 processors. Double in-disk checkpointing on the Myrinet performs quite well and is almost comparable to the double in-memory checkpointing on the Ethernet interconnect. Checkpointing to NFS drive (case(a)) incurs much higher overhead due to the network contention to the file server and the slow disk I/O. This comparison of double checkpoint schemes ((b),(c) and (d)) with the simple disk checkpoint scheme (a) demonstrates that both the memory and disk-based double checkpoint schemes perform very well and are able to take advantage of the fast network hardware.

5.2 Restart Performance

We evaluated the performance of restarting an application using the in-memory fault-tolerance scheme without stopping an application when fault occurs. We have experimented our in-memory fault tolerance scheme with LeanMD - a molecular dynamics simulation program. Failures were simulated by killing one of the processes randomly. 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). The simulation consists of 600 timesteps. The experiments were carried out on 128 processors of NCSA

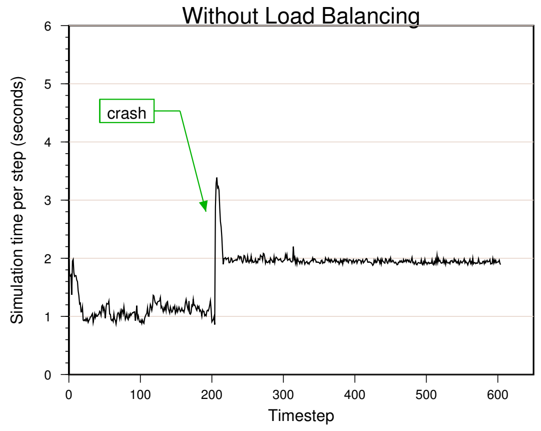

Figure 11: Simulation Time per Step of LeanMD on 128 Processors without Load Balancing

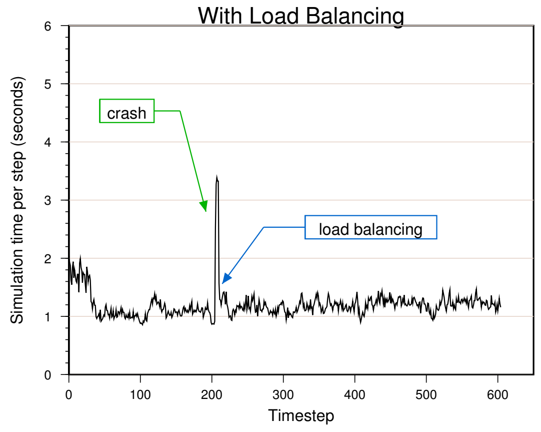

Figure 12: Simulation Time per Step of LeanMD on 128 Processors with Load Balancing

Platinum (IA-32) cluster with 1.5G of RAM on each processor, connected by both Myrinet 2000 interconnect network and 100 Mbit Ethernet.

We measured the time cost in restart from the time when the process is killed to the time just before the parallel job resumes from restart. It took only about 3 seconds for this LeanMD test on 128 processors, which shows that recovery protocol is able to restart an application from a crash within just a few seconds.

We compared the change of simulation speed due to one crash and the effect of load balancing after recovery, as illustrated in Figure 11 and Figure 12. In Figure 11, after crash at about step 200, the simulation was slowed down by a factor of 2, although only 1 out of the 128 processors was lost. This is because some processors receive much more work load than others after restart, and these slower processors slow down the overall execution. In contrast, in Figure 12, load balancing was called automatically when recovery from the crash was complete, after which the simulation time per step was brought down to very close to the pre-crash speed. Our result demonstrates that the load balancing techniques in Charm++ provide a strong support to maintain execution efficiency after a crash, keeping the impact on the overall performance of losing processor low.

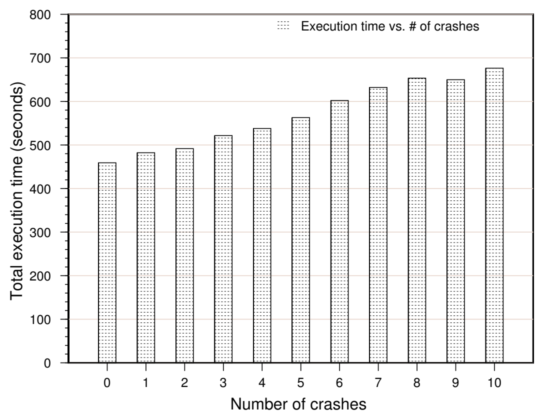

Figure 13: Total Execution Time with Varying Number of Crashes on 128 Processors

Figure 13 further illustrates 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. The x-axis is the increasing total number of crashes during an execution, while the y-axis is the total execution time. As illustrated in the figure, the total execution time was almost unaffected when one or two processors failed. Even in the case when 10 processors failed (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%.

6. CONCLUSION

We presented two fault tolerance protocols based on on-disk and in-memory schemes implemented in the Charm++ and AMPI run-time system. These schemes automate the checkpoint and restart process with minimal effort from the application developer. Moreover, the protocols extend traditional checkpoint-based fault tolerance by allowing the application to restart on a different number of processors. Our analysis shows that these schemes offer a wide selection of features and options to meet the needs of various fault tolerance scenarios. The simple on-disk checkpoint scheme does not entail any memory overhead and is applicable to Charm++ run-time on top of any communication subsystem. The double in-memory checkpoint scheme is well suited for applications with a small memory footprint, without assuming any reliable storage like hard disk. It automatically detects faults and restarts the application on the remaining processors. It takes advantage of fast interconnects and shows good scalability. A variation of this scheme, double in-disk checkpointing, is suitable for applications with large memory footprint running on machines where each node has access to a local disk. All these fault-tolerance schemes and variations are implemented in Charm++ and AMPI run-time, which is portable to a variety of platforms and used by a variety of parallel applications. The fault tolerance schemes described in the paper are available with the standard distribution of AMPI and Charm++[9].

Acknowledgements

This work was supported in part by DOE (Grant B341494 and B505214), the National Science Foundation (NGS 0103645 and ITR 0205611).

7. REFERENCES

[1] NR Adiga, G Almasi, GS Almasi, Y Aridor, R Barik, D Beece, R Bellofatto, G Bhanot, R Bickford, M Blumrich, AA Bright, and J. An overview of the bluegene/l supercomputer, 2002.

[2] Lorenzo Alvisi, E. N. Elnozahy, Sriram Rao, Syed Amir Husain, and Asanka De Mel. An analysis of communication induced checkpointing. In *Symposium on Fault-Tolerant Computing*, pages 242–249, 1999.

[3] Gabriel Antoniu, Luc Bouge, and Raymond Namyst. An efficient and transparent thread migration scheme in the PM^2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico. Lecture Notes in Computer Science 1586*, pages 496–510. Springer-Verlag, April 1999.

[4] Amnon Barak, Shai Guday, and Richard G. Wheeler. The mosix distributed operating system. In *LNCS 672*. Springer, 1993.

[5] Milind Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science*, volume 1970, pages 385–395. Springer Verlag, December 2000.

[6] 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.

[7] 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.

[8] K.M. 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] Charm++ website. http://charm.cs.uiuc.edu/.

[10] Yuqun Chen, Kai Li, and James S. Plank. CLIP: A checkpointing tool for message-passing parallel programs. 1997.

[11] Epcc blue gene/l. http://www.epcc.ed.ac.uk/.

[12] Chao Huang. System support for checkpoint and restart of charm++ and ampi applications. Master's thesis, Dept. of Computer Science, University of Illinois, 2004.

[13] Chao Huang, Orion 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.

[14] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Performance evaluation of adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

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

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

[17] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. Technical Report UT-CS-94-242, 1994.

[18] James S. Plank and Kai Li. Faster checkpointing with n+1 parity. In *24th Annual International Symposium on Fault-Tolerant Computing*, June 1994.

[19] 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.

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

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

[22] Turing cluster. http://www.cse.uiuc.edu/turing.

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

[24] Gengbin Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[25] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *2004 IEEE International Conference on Cluster Computing*, San Dieago, CA, September 2004.