Proactive Fault Tolerance in Large Systems

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

Abstract

High-performance systems with thousands of processors have been introduced in the recent past, and systems with hundreds of thousands of processors should become available in the near future. Since failures are likely to be frequent in such systems, schemes for dealing with faults are important.

In this paper, we introduce a new fault tolerance solution for parallel applications that proactively migrates execution from a processor where a failure is imminent. Our approach assumes that some failures are predictable, and leverages the fact that current hardware devices contain various features supporting early indication of faults. By using the concepts of processor virtualization in Charm++ and Adaptive MPI (AMPI), we describe a mechanism that migrates objects when a failure is expected to arise in a given processor, without requiring spare processors. After migrating objects, and applying a load balancing scheme, the execution of an MPI application can proceed and achieve optimized efficiency. We modify the implementation of collective operations, such as reductions, so that they continue to operate efficiently even after a processor is evacuated and crashes. To demonstrate the feasibility of our approach, we present preliminary performance data.

1 Introduction

Various high-performance systems with thousands of processors have been introduced in the recent past, providing support for execution of large-scale scientific applications. Meanwhile, the present trends in circuit fabrication and packaging technologies indicate that systems with hundreds of thousands of processors should become available in the next few years. In such environments, reliability becomes a major concern, because the overall system reliability is a product of the individual components' reliabilities. Thus, it is very likely that a large system will incur a failure within the execution of an application.

Many production-level scientific applications are currently written using the MPI paradigm [11]. However, the original MPI standards specify very limited features related to reliability or fault tolerance [10]. In most MPI implementations, the entire application has to be shutdown when one of the executing processors experiences a failure. Some alternative MPI implementations have been recently proposed (we discuss representative examples in §5). Most of these solutions implement some form of redundancy, forcing the application to periodically save part of the execution state. When a failure is detected, the most recent state prior to the failure is restored, and execution proceeds from that point. In our previous work, we have demonstrated solutions following this general scheme. In some of them, we used checkpointing/restart mechanisms [13, 21], with the option of saving the checkpointed data to memory or to disks; in another, we used sender-based message-logging [7], where messages are recorded by the sender and resent in case of failures.

In this paper, we introduce a new solution that goes one step ahead: instead of waiting for failures to occur and reacting to those failures, we proactively migrate the execution from a processor where a failure is imminent, without requiring the use of spare processors. To be effective, this approach requires that failures be predictable. We leverage the fact that current hardware devices contain various features supporting early fault indication. As an example, most modern disk drives follow the SMART protocol [3], and provide indications of suspicious behavior like transient access errors, retries, etc. Similarly, motherboards contain temperature sensors, which can be accessed via interfaces like lm_sensor [1] and ACPI [12]. Meanwhile, many network drivers, like those for Myrinet interface cards [5], maintain statistics including packet loss and retransmission counts. Processor manufacturers are building similar infrastructure to detect transient errors inside processor chips [**?**]. By periodically collecting this kind of information, one can create a very powerful mechanism to predict that a severe hardware failure is developing, and act appropriately before that failure becomes catastrophic to the application.

The remainder of this paper is organized as follows. In §2 we describe the major concepts of processor virtualization and object migration. Next, in §3 we present our approach for relocating the execution upon detection of imminent failures, which is based on object migration. We report preliminary results of applying our proactive technique to Charm++ and MPI applications in §4, review related work in §5, and conclude the paper with an overview of our next steps for this research in §6.

2 Processor Virtualization

Our strategy for proactive fault handling is based on the concepts of processor virtualization and object migration, as provided by Charm++ [16] and Adaptive-MPI (AMPI) [14]. Processor virtualization [15] involves the user dividing the problem into a large number of objects without considering the actual number of physical processors. Each such object is called a virtual processor. The user's view of a program is of these virtual processors and their interactions with each other. The programmer leaves the mapping of virtual processors to physical processors to the runtime system. Charm++ organizes the virtual processors as collections of C++ objects that interact via asynchronous method invocations [18]. It refers to each object by a globally unique, user assigned logical address, called an array index. All communication occurs via the array index, which allows the object to be migrated in a way that is completely user transparent. Charm++ supports message delivery to and creation, deletion, migration of the virtual processors in a scalable and efficient manner. It also allows reductions and broadcasts in the presence of migrations.

Adaptive MPI (AMPI)[14] implements its MPI processes as user-level threads bound to Charm++ objects. Message passing between AMPI processes is implemented as communication among these Charm++ objects and the underlying messages are handled by the Charm++ runtime system. Thus AMPI allows traditional MPI codes to exploit the advantages of processor virtualization as well.

The ability to migrate virtual processors means that the runtime system can change the mapping of virtual to physical processors in the middle of an execution to account for the changing load characteristics of a program. Coupling this ability with the fact that for most applications the computation loads and communication patterns exhibited by the virtual processors tend to persist over time, one can now build measurement based runtime load balancing techniques. Distributed or centralized load balancing strategies can be devised to provide efficient remappings. Dynamic load balancing has been used to scale molecular dynamics to thousands of processors [17].

Processor virtualization and its attendant benefits, automatic migration and load balancing, make Charm++ suitable for responding to processor failure predictions. We build upon these available features to implement a system that can respond to a predicted failure in a timely fashion and still allow the application to run efficiently on the remaining processors.

3 Fault Tolerance Strategy

We now describe our technique to migrate tasks from processors where failures are imminent.

3.1 Problem Specification

Our strategy for reacting to fault predictions is entirely software based. However, it requires some support from the hardware and makes the following assumptions about it:

1. The application program is warned of an impending fault through a signal to the application process on the processor that is about to crash.

2. The processor, memory and interconnect subsystems on a warned node continue to work correctly for some period of time after the warning. This gives us an opportunity to react to a warning and adapt the runtime system to survive a crash of that processor.

3. The application continues to run on the remaining processors, even if one processor crashes.

4. We currently assume that warnings for two processors do not occur simultaneously.

For a machine that satisfies the above assumptions, we define the following set of requirements for our strategy to respond to warnings:

1. The response time of our strategy should be as low as possible. The time taken by the runtime system to react, so that it can survive the processor's crash, should be minimized.

2. Our strategy should not require the start up of a new application process on either a new processor or any of the existing ones. The runtime system should be able to deal with a warning and possible crash without having to resort to ”spare” processes [21][7].

3. After responding to a fault warning on a processor, the efficiency loss in the application should be proportional to the fraction of computing power lost.

3.2 Solution Design

Our solution has three major parts. The first part migrates the Charm++ objects off the warned processor and ensures that point-to-point message delivery continues to function even after a crash. The second part deals with allowing collective operations to cope with the possibility of the loss of a processor. The third part makes sure that the runtime system can balance the application load among the remaining processors after a crash. The three parts are interdependent, but for the sake of clarity we describe them separately.

Each migratable object in Charm++ is identified by a globally unique index which is used by other objects to communicate with it. A scalable algorithm is used for point-to-point message delivery in the face of asynchronous object migration, as described in [18]. The system maps each object to a *home* processor, which always knows where that object can be reached. An object need not reside on its home processor. As an example, in Figure 1 an object on processor A wants to send a message to an object (say X) that has its home on processor B but currently resides on processor C. If processor A has no idea where X resides, it sends the message to B, which then forwards it to C. Since forwarding is inefficient, C sends a routing update to A, advising it to send future messages for object X directly to C.

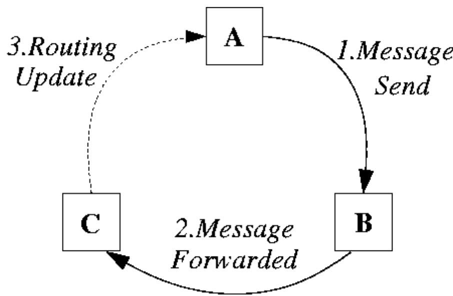


Figure 1. Message forwarding among processors: A is the source, B the home, and C the destination.

The situation is complicated slightly by migration. If a processor receives a message for an object that has migrated away from it, the message is forwarded to the object's last known location. Assume object X migrates from C to another processor D at the same time that A starts sending it a message. As a result, X's home processor (B) does not yet have the correct address for X when it decides to forward the message to C. However, C forwards it to D and then D sends a routing update to A. B also receives the migration update from C and forwards any future messages to D. The protocol is described in much greater detail in [18].

When a processor (say B) starts failing, it is easy enough in Charm++ to migrate away the objects residing there. However, if B were to crash, it would disrupt message delivery to objects such as X which have their homes on B. There would be no processor that would always know where that object can be reached. We solve that problem by changing the index-to-home mapping such that all objects mapped to B now map to some other processor D. This mapping needs to change on all processors in such a way that they stop sending messages to B as soon as possible. The protocol has two parts, one running on the processor that has received a warning and the second part running on the other valid processors.

Once processor B receives a warning that it might crash, it must:

1. Change the index-to-home mapping so that all objects that previously had their homes on B, now map to D.
2. Send a high priority *evacuation* message to all other valid processors (processors that have not sent *evacuation* messages to this one in the past). The message contains this processor's number.
3. Send all objects on B to their home processors, including objects that previously had their homes on B.

Meanwhile, when a processor receives an *evacuation* message, it must:

1. Mark the sending processor as invalid.
2. Change the mapping so that all objects previously mapped to B now map to D. The mapping should be changed in such a way that all processors independently agree on the same replacement for B.
3. Change routing records for any object that point to B so that those records now point to that object's new home processor.
4. If this processor contains any object that previously had its home on B, inform its new home D about the object's current position.

We now discuss the protocol's behavior in different cases and whether B needs to process a message after being warned. Of course, any messages to B sent before a processor receives the *evacuation* message from B will have to be processed or forwarded by B. There is no way around it, although the high priority of the *evacuation* message tries to reduce the chances of such a case.

We first analyze the effect of this algorithm on objects that had their homes on B. This protocol assigns a new home D for all such objects (let X be one of them). If some of these objects were on B, they are migrated to D; if they existed on other processors, D is informed of their current

position. If D receives a message for object X after having received the *evacuation* message from B but before X has migrated into it or it has been informed of its new position, the message is buffered. Thus, no messages are sent to B in this case. If a processor sends to D a message for object X before D has received the *evacuation* message from B, D has no option but to send it to either B or some other processor which has previously told D that X exists on it. In this case, it is possible that B would have to process a message after receiving a fault warning.

Any object (say Y) existing on B but having its home on some other processor (say C) is sent to its home processor (C). The *evacuate* message changes the routing tables of all processors such that they will send to C all messages for object Y, instead of sending to B. If any message for Y gets to C before Y itself, but after the *evacuate* message, it is buffered. Again the only case in which B might receive a message is if C has not received the *evacuate* message when it receives a message for Y. All objects on B are sent to their home processors and not some other ones because, in this case, B does not need to send a migration update to the home processors. Thus, according to this protocol, B might have to forward some messages sent or forwarded by other processors before they had received the *evacuate* message. Once all processors have received the *evacuate* message, no messages destined for Charm++ objects will be sent to B.

3.3 Support for Collective Operations

Collective operations are important primitives for parallel programs. It is essential that they continue to operate correctly even after a crash. Asynchronous reductions are implemented in Charm++ by reducing the values from all objects on one processor and then reducing these partial results across all processors [18]. The processors are arranged in a k-ary reduction tree. Each processor reduces the values from its local objects and the values from the processors that are its children, and passes the result along to its parent. Reductions occur in the same sequence on all objects and are identified by a sequence number. If a processor were to crash, the tree could become disconnected. Therefore, we try to rearrange the tree around the warned node. If the warned node is a leaf, then the rearranging involves just deleting it from its parent's list of children. In the case of an internal node, the transformation is shown in Figure 2. Though this rearrangement increases the number of children for some nodes, the number of nodes whose parent or children change is limited to the warned node, its parent and its children.

Since rearranging a reduction tree while reductions are in progress is very complicated, we adopt a simpler solution. The warned node polls its parent, children and itself for the highest reduction that any of them has started. Because the rearranging affects only these nodes, each of them shifts to using the new tree when it has finished the highest reduction started on the old tree by one of these nodes. The exact sequence of messages is the following:

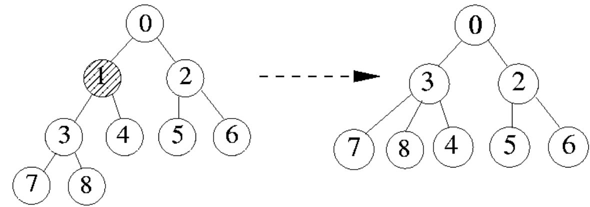


Figure 2. Rearranging of the reduction tree, when processor 1 receives a fault warning.

1. Warned node sends the tree modifications to parent and children.

2. Parent and children store the changes but do not make them to the current tree. They reply with the highest reduction number that they have seen. They also buffer any further reduction messages.

3. The warned node finds the maximum reduction number and informs the parent and children.

4. The parent and children unblock and continue until they reach the maximum reduction number; at that point, they change to the new tree.

It is evident from the protocol that evacuating a processor might lead to severe load imbalance (since all of B's objects move to D in our example). Therefore, it is necessary that the runtime system be able to balance the load after a migration. Minor changes to the already existing Charm++ load balancing framework allow us to map the objects to only the remaining valid processors. However, as we show in §4, this capability has a major effect on performance of an application.

The protocols described above need to be slightly modified to accommodate AMPI objects (corresponding to MPI processes) as they are implemented currently. According to its current implementation, AMPI objects can not be migrated at an arbitrary place in the code [14]. Thus, we add a *MPI_Evacuate* call to mark a place in the code where it is safe to migrate the object away from a warned processor. In the fault-free case the overhead of this method is just one *if* statement. However, we are working on modifying AMPI's implementation so that it can support *anytime* migration. This would allow AMPI objects to use the same protocols as any other Charm++ object.

4 Experimental Results

We tested our strategy on a 16 processor cluster of 1Ghz Pentium III nodes with 1 GB of RAM and 2 GB of swap space, connected by Gigabit as well as 100 Mbit Ethernet. We compiled all programs with GNU GCC version 3.2 with the -O2 flag. We used a simple 5-point stencil calculation written in Charm++ to perform most of the initial evaluations of our protocol. It is easier to control the memory usage and granularity in this program than in a real application. We evaluated the current AMPI version of our protocol as well. We simulated a fault warning by sending the USR1 signal to an application process on a computation node.

We wanted to evaluate how fast our protocol is able to morph the runtime system such that if the warned processor crashes, the runtime system remains unaffected. We call this the *processor evacuation* time. However, it is not evident how this should be exactly measured. One option is to measure the time taken to process all messages that need to be processed by the warned processor before the runtime system can survive a fault. However, it does not include the time taken for the objects on the warned processor to be actually sent to the destination processor. Since the outgoing objects are actually buffered on the sending side the user code does not know the exact time when the objects are sent. We decided to measure this time by having all the other processors reply back to the warned one after receiving all their objects. Thus, for a certain run, we measure processor evacuation time as the maximum of the time taken for all the processors to reply back and the last message processed by the warned processor. It should be noted that these reply messages are not necessary for the protocol, but are needed solely for evaluation. The result is, of course, a pessimistic estimate of the actual processor evacuation time, since it includes the overhead of extra messages.

The processor evacuation time for the Charm++ stencil program on 16 processors, for different problem sizes and for both interconnects, is shown in Figure 3. The evacuation time increases linearly with the total problem size until at least 1.4 GB. This shows that it is dominated by the time to transmit the data out from the warned processor. For the same reason, the processor evacuation time for Gigabit is a fraction of the time for 100Mbit. However, our method of measurement is biased against faster interconnects, since the measurement overheads form a more significant part of the evacuation time than in the case of slower interconnects. Hence the actual performance of Gigabit, in comparison to 100Mbit, is even better.

Figure 4 presents the processor evacuation time for a particular problem size (268 MB) on different numbers of processors. For both interconnects, the evacuation time decreases linearly with the data volume per processor. Gigabit responds significantly faster than 100 Mbit. This shows that

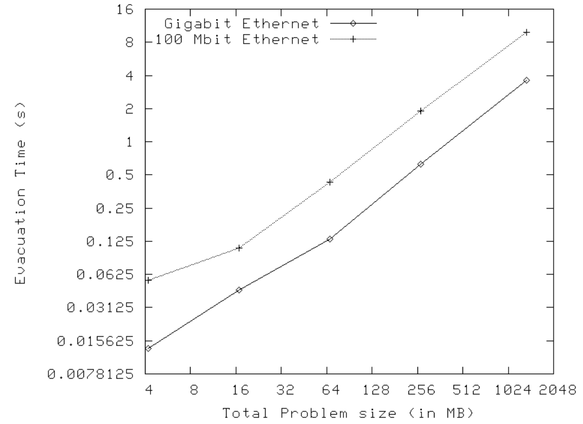


Figure 3. Processor evacuation time on 16 processors in a Charm++ 5-point stencil computation.

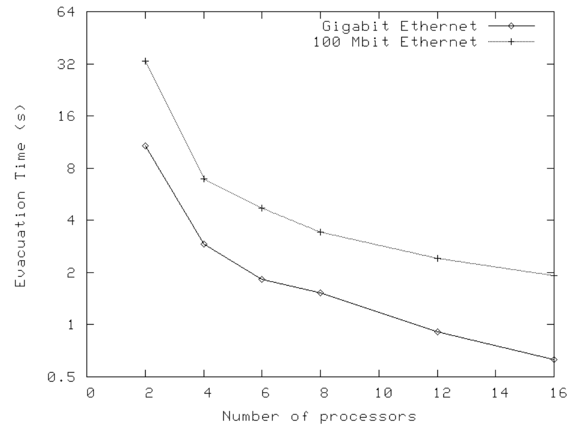
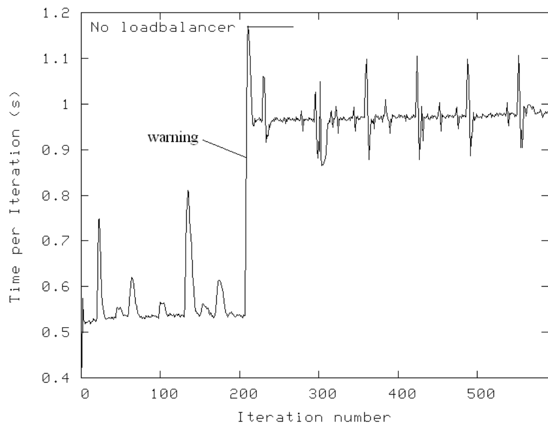


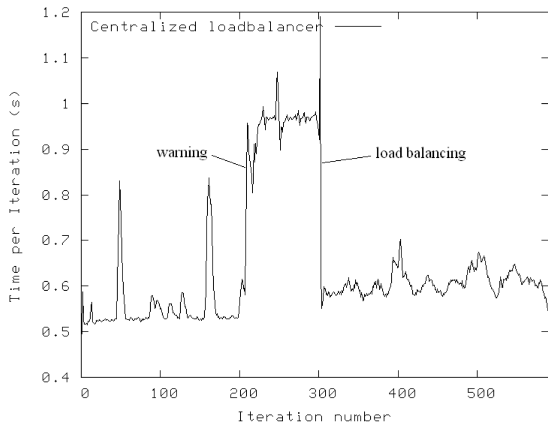
Figure 4. Processor evacuation time for 268 MB of total data in a Charm++ 5-point stencil computation.

our protocol scales to at least the number of processors used in the experiment. In fact, the only part in our protocol that is dependent on the number of processors is the initial *evacuate* message sent out to all processors. The other parts of the protocol scale linearly with either the size of objects or the number of objects on each processor.

In Figure 5, we compare the performance of the stencil computation after a warning, with and without a subsequent load balancing phase. It plots the average time taken for each iteration over all Charm++ objects in the calculation. In Figure 5(a), the execution time per iteration nearly doubles after a processor receives a warning and is evacu-



(a) Performance after a warning without load balancing



(b) Performance after a warning with Centralized load balancing

Figure 5. Time per Iteration for a Charm++ 5-point stencil calculation with 576 MB of data on 16 processors in the presence of faults.

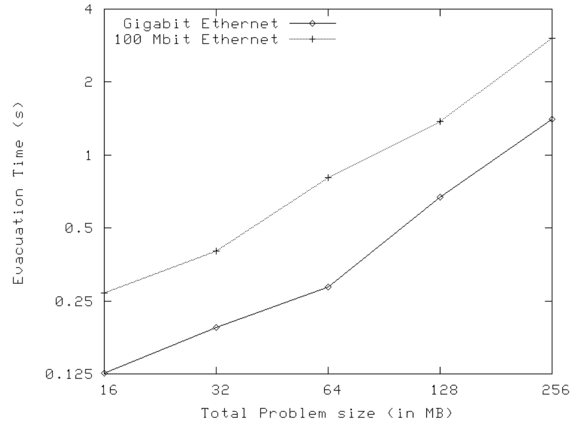


Figure 6. Processor evacuation time on 16 processors in a AMPI 7-point stencil computation.

ated. This happens because all the objects on the warned processor are sent to their homes, including the objects that previously had homes on the warned processor. This creates a load imbalance across the system. As a result, the performance of the application degrades significantly. However, in 5(b), load balancing occurs after the migration and the performance of the application shows a marked improvement over the previous case. The performance penalty after the crash is proportional to the computation power lost.

Figure 6 evaluates the processor evacuation time for an AMPI application consisting of a 7-point stencil computation, for various problem sizes on 16 processors. It shows a similar behavior as in the Charm++ program. The evacuation time increases linearly with data size; Gigabit performs significantly better than 100Mbit. However, the evacuation time with the same data size for AMPI is significantly higher than that of Charm++. Current restrictions on when an AMPI object is allowed to migrate, due to our present AMPI implementation, form the primary reason for this.

Thus, these experiments show that our protocol's response to a fault warning is restricted by the amount of data on that processor and the speed of the interconnect. The protocol itself scales well with both data size and number of processors. Load balancing is seen to have a significant impact on the performance of an application after a fault warning.

5 Related Work

The techniques for fault tolerance in message-passing environments can be broadly divided in two classes: checkpointing schemes and message-logging schemes. In checkpoint-based techniques, the application status is periodically saved to stable storage, and recovered when a failure occurs. The checkpointing can be coordinated or independent among the processors. However, due to the possible rollback effects in independent schemes, most implementations use coordinated checkpointing. Representatives of this class are CoCheck [20], Starfish [2] and Clip [8].

In message-logging techniques, the central idea is to retransmit one or more messages when a system failure is detected. Message-logging can be optimistic, pessimistic or causal. Because of the complex rollback protocol, opti-

mistic logging [?] is rarely used; instead, pessimistic logging schemes are more frequently adopted, like in FT-MPI [9], MPI/FT [4], MPI-FT [19] and MPICH-V [6]. Causal logging (such as in [?]) attempts to strike a balance between optimistic and pessimistic logging; however, its restart is also non-trivial.

In all of these proposed fault-tolerant solutions, some corrective action is taken in reaction to a detected failure. In contrast, with the proactive approach that we present in this paper, fault handling consists in migrating a task from a processor where failures are imminent. Thus, no recovery is needed. In addition, migration is already an existing feature in AMPI, as it supports dynamic load balancing via task migration. Hence, adding fault tolerance support is not a drastic change to the original AMPI implementation.

6 Conclusion and Future Work

We have presented a new technique for fault tolerance in MPI applications. Our approach is supported by the object migration and load balancing capabilities of Charm++ and AMPI. Upon receiving warnings that a failure is imminent on a given processor, our runtime system proactively attempts to migrate execution off that processor before a crash actually happens. Our preliminary results show that task migration time is constrained mainly by the interconnect speed. The migration performance scales well with the dataset size.

We are currently working to complete and validate our protocol in several important aspects. First, we are modifying our AMPI implementation so that migration can occur at any moment; this should provide a faster evacuation time for MPI applications. Secondly, we will extend our protocol to make it capable of handling simultaneous faults on different system nodes. Finally, we plan to apply our technique to a wider set of MPI applications, and will conduct those tests on large system configurations. These tests will enable verification of the good scalability that we observed in our preliminary experiments.

References

[1] Hardware monitoring by lm_sensors. Available at http://secure.netroedge.com/-lm78/info.html.

[2] AGBARIA, A., AND FRIEDMAN, R. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing 6*, 3 (July 2003), 227–236.

[3] ALLEN, B. Monitoring hard disks with SMART. *Linux Journal* (January 2004).

[4] APPARAO, P., AND AVERILL, G. Firmware-based platform reliability. Intel white paper, October 2004.

[5] BATCHU, R., SKJELLUM, A., CUI, Z., BEDDHU, M., NEELAMEGAM, J. P., DANDASS, Y., AND APTE, M. Mpi/fttm: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid* (2001), IEEE Computer Society, p. 26.

[6] BODEN, N. J., COHEN, D., FELDERMAN, R. E., KULAWIK, A. E., SEITZ, C. L., AND SEIZOVIC, J. N. Myrinet: A gigabit per second local area network. *IEEE Micro 15* (1995), 29–36.

[7] BOUTEILLER, A., CAPPELLO, F., HÉRAULT, T., KRAWEZIK, G., LEMARINIER, P., AND MAGNIETTE, F. MPICH-V2: A fault tolerant MPI for volatile nodes based on the pessimistic sender based message logging programming via processor virtualization. In *Proceedings of Supercomputing'03* (Phoenix, AZ, November 2003).

[8] CHAKRAVORTY, S., AND KALÉ, L. V. A fault tolerant protocol for massively parallel machines. In *FTPDS Workshop at IPDPS'2004* (Santa Fe, NM, April 2004), IEEE Press.

[9] CHEN, Y., PLANK, J. S., AND LI, K. Clip: A checkpointing tool for message-passing parallel programs. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)* (1997), pp. 1–11.

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

[11] FAGG, G. E., AND DONGARRA, J. J. Building and using a fault-tolerant MPI implementation. *International Journal of High Performance Computing Applications 18*, 3 (2004), 353–361.

[12] GROPP, W., AND LUSK, E. Fault tolerance in message passing interface programs. *International Journal of High Performance Computing Applications 18*, 3 (2004), 363–372.

[13] GROPP, W., LUSK, E., AND SKJELLUM, A. *Using MPI*, second ed. MIT Press, November 1999.

[14] HEWLETT-PACKARD, INTEL, MICROSOFT, PHOENIX, AND TOSHIBA. Advanced configuration and power interface specification. ACPI Specification Document, Revision 3.0, September 2004. Available from http://www.acpi.info.

[15] HUANG, C. System support for checkpoint and restart of Charm++ and AMPI applications. Master's thesis, Dep. of Computer Science, University of Illinois, Urbana, IL, 2004. Available at http://charm.cs.uiuc.edu/papers/CheckpointThesis.html.

[16] HUANG, C., LAWLOR, O., AND KALÉ, L. V. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)* (College Station, TX, October 2003).

[17] KALÉ, L. V. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)* (Madrid, Spain, February 2004).

[18] KALÉ, L. V., AND KRISHNAN, S. Charm++: Parallel programming with message-driven objects. In *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds. MIT Press, 1996, pp. 175–213.

[19] KALÉ, L. V., KUMAR, S., ZHENG, G., AND LEE, C. W. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)* (Melbourne, Australia, June 2003).

[20] LAWLOR, O. S., AND KALÉ, L. V. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience 15* (2003), 371–393.

[21] LOUCA, S., NEOPHYTOU, N., LACHANAS, A., AND EVRIPIDOU, P. MPI-FT: Portable fault tolerance scheme for MPI. *Parallel Processing Letters 10*, 4 (2000), 371–382.

[22] STELLNER, G. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium* (1996), pp. 526–531.

[23] STROM, R., AND YEMINI, S. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems 3*, 3 (1985), 204–226.

[24] ZHENG, G., SHI, L., AND KALÉ, L. V. FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *2004 IEEE International Conference on Cluster Computing* (San Diego, CA, September 2004).