Proactive Fault Tolerance in MPI Applications via Task Migration

Sayantan Chakravorty Celso L. Mendes Laxmikant V. Kalé

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

Abstract. Failures are likely to be more frequent in systems with thousands of processors. Therefore, schemes for dealing with faults become increasingly important. In this paper, we present a fault tolerance solution for parallel applications that proactively migrates execution from processors where failure is imminent. Our approach assumes that some failures are predictable, and leverages the features in current hardware devices supporting early indication of faults. We use the concepts of processor virtualization and dynamic task migration, provided by Charm++ and Adaptive MPI (AMPI), to implement a mechanism that migrates tasks away from processors which are expected to fail. To demonstrate the feasibility of our approach, we present performance data from experiments with existing MPI applications. Our results show that proactive task migration is an effective technique to tolerate faults in MPI applications.

1 Introduction

High-performance systems with thousands of processors have been introduced in the recent past, and the current trends indicate that systems with hundreds of thousands of processors should become available in the next few years. In systems of this scale, reliability becomes a major concern, because the overall system reliability decreases with a growing number of system components. Hence, large systems are more likely to incur a failure during execution of a long-running application.

Many production-level scientific applications are currently written using the MPI paradigm [1]. However, the original MPI standards specify very limited features related to reliability or fault tolerance [2]. In traditional MPI implementations, the entire application has to be shutdown when one of the executing processors experiences a failure. Given the practical problem of ensuring application progress despite the occurrence of failures in the underlying environment, some alternative MPI implementations have been recently proposed (we discuss representative examples in Section 5). Most of these solutions implement some form of redundancy, forcing the application to periodically save part of its execution state. In our previous work, we have demonstrated solutions following this general scheme, using either checkpointing/restart mechanisms [3, 4] or message-logging approaches [5].

In this paper, we present a new solution that goes one significant step further: instead of waiting for failures to occur and reacting to them, we proactively migrate the execution from a processor where a failure is imminent, without requiring the use of spare processors. We build on our recent work on fault-driven task migration [6], and develop a scheme that supports MPI applications transparently to the user. Based on processor virtualization and on the migration capabilities of AMPI, our runtime system handles imminent faults by migrating the MPI tasks to other processors.

To be effective, this approach requires that failures be predictable. We leverage the features in current hardware devices supporting early fault indication. As an example, most motherboards contain temperature sensors, which can be accessed via interfaces like ACPI [7]. Meanwhile, recent studies have demonstrated the feasibility of predicting the occurrence of faults in large-scale systems [8] and of using these predictions in system management strategies [9]. Hence, it is possible, under current technology, to act appropriately before a system fault becomes catastrophic to an application. In this paper we focus on handling warnings for imminent faults and not on the prediction of faults. For unpredictable faults, we can revert back to traditional recovery schemes, like checkpointing [3] or message logging [5].

Our strategy is entirely software based and does not require any special hardware. However, it makes some reasonable assumptions about the system. The application is warned of an impending fault through a signal to the application process on the processor that is about to crash. 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 node. The application continues to run on the remaining processors, even if one processor crashes.

We decided on a set of requirements before setting out to design a solution. The time taken by the runtime system to change (response time), so that it can survive the processor's crash, should be minimized. Our strategy should not require the start up of a new "spare" [4, 5] process on either a new processor or any of the existing ones. When an application loses a processor due to a warning, we expect the application to slow down in proportion to the fraction of computing power lost. Our strategy should not require any change to the user code. We verify in Section 4 how well our protocol meets these specifications.

2 Processor Virtualization

Processor virtualization is the key idea behind our strategy for proactive fault tolerance. The user breaks up his computation into a large number of objects without caring about the number of physical processors available. These objects are referred to as virtual processors. The user views the application in terms of these virtual processors and their interactions. Charm++[10] and Adaptive-MPI (AMPI) [11] are based on this concept of processor virtualization. The Charm++ runtime system is responsible for mapping the virtual processors to physical processors. It can also migrate a virtual processor from one physical processor to another at runtime. 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.

Coupling the capability of migration 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. Dynamic load balancing in Charm++ has been used to scale diverse applications such as cosmology[12] and molecular dynamics [13] to thousands of processors .

Adaptive MPI (AMPI) [11] is an implementation of the Message Passing Interface (MPI) on top of Charm++. Each MPI process is a user-level thread bound to a Charm++ virtual processor. The MPI communication primitives are implemented as communication between the Charm++ objects associated with each MPI process. Traditional MPI codes can be used with AMPI after no or slight modifications. These codes can also take advantage of automatic migration, automatic load balancing and adaptive overlap of communication and computation.

3 Fault Tolerance Strategy

We now describe our technique to migrate tasks from processors where failures are imminent. 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. It also helps to ensure that the runtime system can balance the application load among the remaining processors after a crash. The third part migrates AMPI processes away from the warned processor. The three parts are interdependent, but for the sake of clarity we describe them separately.

3.1 Evacuation of Charm++ Objects

Each migratable object in Charm++ is identified by a globally unique index which is used by other objects to communicate with it. We use a scalable algorithm for point-to-point message delivery in the face of asynchronous object migration, as described in [14]. 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, 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 X-bound messages directly to C.

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. Figure 1 illustrates this case as object X migrates from C to another processor D. X's home processor (B) may not yet have the correct address for X when it forwards 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. [14] describes the protocol in much greater detail.

When a processor (say E) detects that a fault is imminent, it is possible in Charm++ to migrate away the objects residing there. However, this crash would disrupt message delivery to objects which have their homes on E, due to the lack of updated routing information. We solve that problem by changing the index-to-home mapping such that all objects mapped to E now map to some other processor F. This mapping needs to

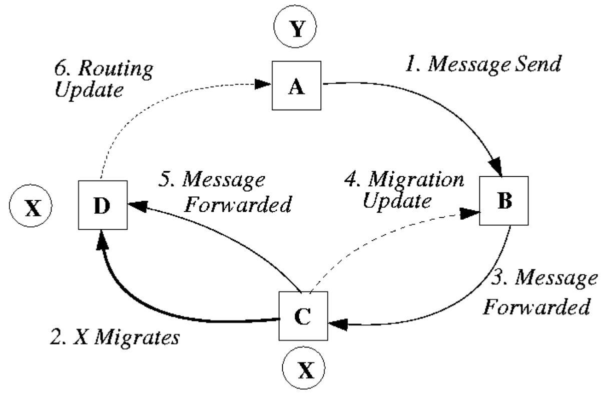


Fig. 1. Message from object Y to X while X migrates from processor C to D.

change on all processors in such a way that they stop sending messages to E as soon as possible. The messages exchanged for this distributed protocol are shown in Figure 2. As previously described, objects which had their home on E now have their home on F.

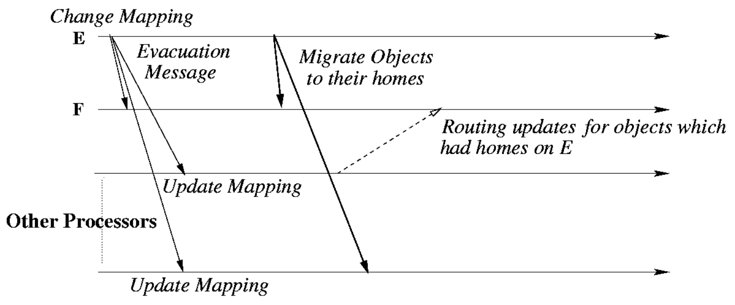

Change Mapping
E
Evacuation Message
Migrate Objects to their homes
F
Routing updates for objects which had homes on E
Update Mapping
Other Processors
Update Mapping

Fig. 2. Messages exchanged when processor E is being evacuated.

The index-to-home mapping is a function that maps an object index and the set of valid processors to a valid processor. If the set of valid processors is given by the bitmap *isValidProcessor*, the initial number of processors is *numberProcessors* and *sizeOfNode* is the number of processors in a node, then an index-to-home mapping is given by:

```
$start \leftarrow possible \leftarrow (index$ **mod** $numberProcessors)$
**while** $!isValidProcessor[possible]$ **do**
  $possible \leftarrow (possible +$ **sizeOfNode**$)$ **mod** $numberProcessors$
  **if inSameNode**$(start, possible)$ **then**
    **abort**$(``No\ valid\ node\ left'')$
  **end**
**end**
**return** $possible;$
```

For efficiency, we derive the mapping once and store it in a hashtable for subsequent accesses. When an *evacuation* message is received, we repopulate the hashtable. An analysis of the protocol (omitted here due to space constraints) shows that the only messages that E needs to process after being warned were sent to it by processors which had not yet received the *evacuation* message from E. Once all processors receive the *evacuation* message, no messages destined for Charm++ objects will be sent to E.

This protocol is robust enough to deal with multiple simultaneous fault warnings. The distributed nature of the algorithm, without any centralized arbitrator or even a collective operation, makes it robust. The only way two warned processors can interfere with each other's evacuation is if one of them (say H) is the home for an object existing on the other (say J). This might cause J to evacuate some objects to H. Even in this case once J receives the *evacuation* message from H, it changes its index-to-home mapping and does not evacuate objects to H. Only objects that J evacuates before receiving an *evacuation* message from H are received by H. Though H can of course deal with these by forwarding them to their new home, this increases the evacuation time. This case might occur if H receives J's *evacuation* message before it receives its own warning and so does not send an evacuation message to J. We reduce the chances of this by forcing a processor to send an *evacuation* message to not only all valid processors but also processors that started their evacuation recently.

3.2 Support for Collective Operations in the Presence of Fault Warnings

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 residing in a processor and then reducing these partial results across all processors [14]. 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 tree node corresponding to the warned processor. If such node is a leaf, then the rearranging involves just deleting it from its parent's list of children. In the case of an internal tree node, the transformation is shown in Figure 3. Though this rearrangement increases the number of children for some nodes in the tree, the number of nodes whose parent or children change is limited to the node associated to the warned processor, its parent and its children.

Since rearranging a reduction tree while reductions are in progress is very complicated, we adopt a simpler solution. The node representing the warned processor 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. If there are warnings on a node and on one of its children at the same time, we let the parent modify the tree first and then let the child change the modified tree. Other changes to the tree can go on simultaneously in no specific order.

The Charm++ runtime provides support for asynchronous broadcasts to its objects [14]. It simplifies the semantics of using broadcasts by guaranteeing that all objects

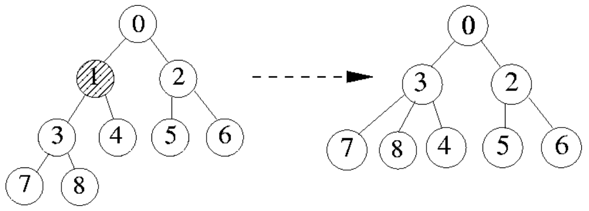


Fig. 3. Rearranging of the reduction tree, when processor 1 receives a fault warning.

receive broadcasts in the same sequence. All broadcasts are forwarded to an appointed serializer. This processor allots a number to a broadcast and sends it down the broadcast tree to all other processors. Each processor delivers the broadcast messages to the resident objects in order of the broadcast number. Contrary to intuition, this does not create a *hotspot* since the number of messages received and sent by each processor during a broadcast is unchanged.

We can change the broadcast tree in a way similar to the reduction tree. However, if the serializer receives a warning we piggyback the current broadcast number along with the *evacuation* message. Each processor changes the serializer according to a predetermined function depending on the set of valid processors. The processor that becomes the new serializer stores the piggybacked broadcast count. Any broadcast messages received by the old serializer are forwarded to the new one.

It is evident from the protocol that evacuating a processor might lead to severe load imbalance. Therefore, it is necessary that the runtime system be able to balance the load after a migration caused by fault warning. Minor changes to the already existing Charm++ load balancing framework allow us to map the objects to the remaining subset of valid processors. As we show in Section 4, this capability has a major effect on performance of an application.

3.3 Processor Evacuation in AMPI

We modified the implementation of AMPI to allow the runtime system to migrate AMPI threads even when messages are in flight, i.e. when there are outstanding MPI requests or receives. This is done by treating outstanding requests and receives as part of the state of an AMPI thread. When a thread migrates from processor A to B, the queue of requests is also packed on A and sent to processor B. At the destination processor B, the queue is unpacked and the AMPI thread restarts waiting on the queued requests. However, just packing the requests along with the thread is not sufficient. Almost all outstanding requests and receives are associated with a user-allocated buffer where the received data should be placed. Packing and moving the buffer from A to B might cause the buffer to have a different address on B's memory. Hence the outstanding request that was copied over to the destination would point to a wrong memory address on B.

We solve this problem by using the concept of *isomalloc* proposed in PM^2 [15]. AMPI already uses this to implement thread migration. We divide the virtual address space equally among all the processors. Each processor allocates memory for the user only in the portion of the virtual address space alloted to it. This means that no two buffers allocated by the user code on different processors will overlap. This allows all user buffers in a thread to be recreated at the same address on B as on A. Thus, the buffer addresses in the requests of the migrated thread point to a valid address on B as well. This method has the disadvantage of restricting the amount of virtual address space available to the user on each processor. However, this is a drawback only for 32-bit machines. In the case of 64-bit machines, even dividing up the virtual address space leaves more than sufficient virtual address space for each processor.

4 Experimental Results

We conducted a series of experiments to assess the effectiveness of our task migration technique under imminent faults. We measured both the response time after a fault is predicted and the overall impact of the migrations on application performance. In our tests, we used a 5-point stencil code, written in C and MPI, and the *Sweep3d* code, which is written in Fortran and MPI. The 5-point stencil code allows a better control of memory usage and computation granularity than a more complex application. Sweep3d is the kernel of a real ASCI application; it solves a 3D Cartesian geometry neutron transport problem using a two-dimensional processor configuration.

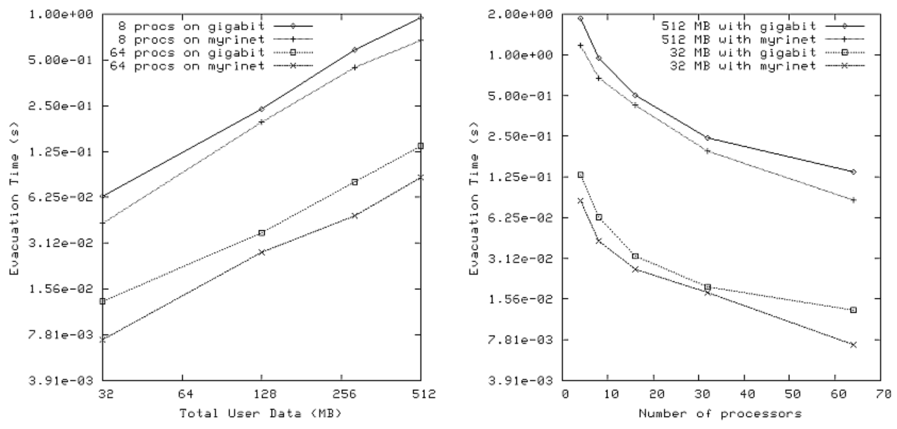
We executed our tests on NCSA's Tungsten system, a cluster of 3.2 GHz dual-Xeon nodes, with 3 GBytes of RAM per node, and two kinds of interconnects, Myrinet and Gigabit-Ethernet. Each node runs Linux kernel 2.4.20-31.9. We compiled the stencil program with GNU GCC version 3.2.2, and the Sweep3d program with Intel's Fortran compiler version 8.0.066. For both programs, we used AMPI and Charm++ over the Myrinet and Gigabit interconnects. We simulated a fault warning by sending the USR1 signal to an application process on a computation node.

4.1 Response Time Assessment

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. We estimate the processor evacuation time as the maximum of the time taken to receive acknowledgments that all evacuated objects have been received at the destination processor and the last message processed by the warned processor. It should be noted that these acknowledgment messages are not necessary for the protocol; they are needed solely for evaluation. The measured value is, of course, a pessimistic estimate of the actual processor evacuation time, because it includes the overhead of those extra messages.

The processor evacuation time for the 5-point stencil program on 8 and 64 processors, for different problem sizes and for both interconnects, is shown in Figure 4(a). The evacuation time increases linearly with the total problem size until at least 512 MB. This shows that it is dominated by the time to transmit the data out from the warned processor. Thus, the evacuation time in Myrinet is significantly smaller than in Gigabit.

Figure 4(b) presents the processor evacuation time for two problem sizes, 32 MB and 512 MB, of the 5-point stencil calculation on different numbers of processors. For both interconnects, the evacuation time decreases more or less linearly with the data



(a) Scaling Evacuation time with data size (b) Scaling Evacuation time with processors

Fig. 4. Processor evacuation time for MPI 5-point stencil calculation

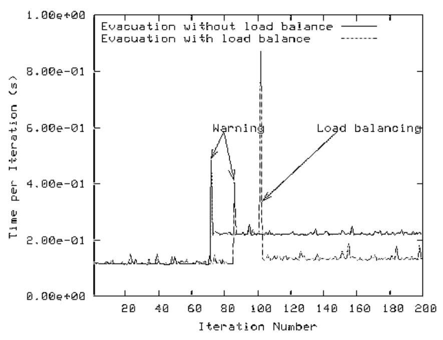
volume per processor. Myrinet has a significantly faster response time than Gigabit. Table 1 shows similar data corresponding to the evacuation time for Sweep3d, for a problem size of 150×150×150. These experiments reveal that the response to a fault warning is constrained only by the amount of data on the warned processor and the speed of the interconnect. In all cases, the evacuation time is under 2 seconds, which is much less than the time interval demanded by fault prediction as reported by other studies [8]. The observed results show that our protocol scales to at least 256 processors. 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.

4.2 Overall Application Performance

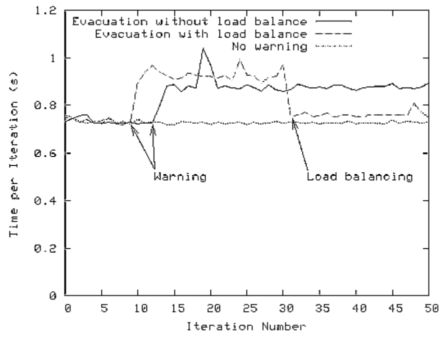
We evaluated the overall performance of the 5-point stencil and Sweep3d under our task migration scheme in our second set of experiments. We were particularly interested in observing how the presence of warnings and subsequent task migrations affect application behavior. For two executions of the 5-point stencil on 8 processors and a dataset size of 288 MB, we observed the execution profiles shown in Figure 5(a). We generated one warning in both executions. In the first execution, the evacuation prompted by the warning at iteration 85 forces the tasks in the warned processor to be sent to other processors. The destination processors become more loaded than the others, resulting in much larger iteration times for the application. In the second execution, we

Table 1. Evacuation time for a 150^3 Sweep3d problem on different numbers of processors

Number of Processors	4	8	16	32	64	128	256
Evacuation Time (s)	1.125	0.471	0.253	0.141	0.098	0.035	0.025



(a) 5-point stencil with 288MB of data on 8 processors



(b) 150^3 Sweep3d problem on 32 processors

Fig. 5. Time per iteration for different applications in the presence of warnings

used AMPI's dynamic load balancing capabilities to re-balance the load among the remaining processors after an evacuation at iteration 70. After the re-balancing occurs (at iteration 100), the remaining processors have nearly the same load, and performance improves significantly. In this phase, the performance drop relative to the beginning of the execution is exactly proportional to the computational capability that was lost (namely, one processor in the set of eight original processors).

We evaluated the effects of evacuation and load balancing on Sweep3d executions on 32 processors that solved a 150^3 sized problem. Figure 5(b) shows the application behavior, in terms of iteration durations, for the cases when no processor is evacuated, one processor is evacuated and when an evacuation is followed by a load balancing phase. The evacuations are triggered by externally inserted fault warnings. The performance of the application deteriorates after a warning in both cases. The performance hit of around 10% increase in time per iteration is far more than the loss in computation power of 3%. This is probably caused by computation and communication load imbalance among processors. After load balancing the performance improves significantly and the increase in time per iteration relative to the warning-free case is around 4%. Thus the loss in performance is very similar to the loss in computation power once AMPI has performed load balancing.

The Projections analysis tool processes and displays trace data collected during application execution. We use it to assess how parallel processor utilization changes across a Sweep3d execution of a 150^3 problem on 32 processors. We trigger warnings on Node 3 which contains two processors: 4 and 5. This tests the case of multiple simultaneous warnings by evacuating processors 4 and 5 at the same time. Before the warnings occur, processors have nearly uniform load and similar utilization (Figure 6(a)). After evacuation takes place, processors 4 and 5 quit the execution and their objects get distributed among the other processors. However, this creates a load imbalance among the processors, with some taking longer than others to finish iterations. The redistribution of objects can also increase communication load by placing objects that communicate frequently on different processors. This reduces the utilization significantly on all processors (Figure 6(b)). Finally, after load balancing, the remaining processors divide the

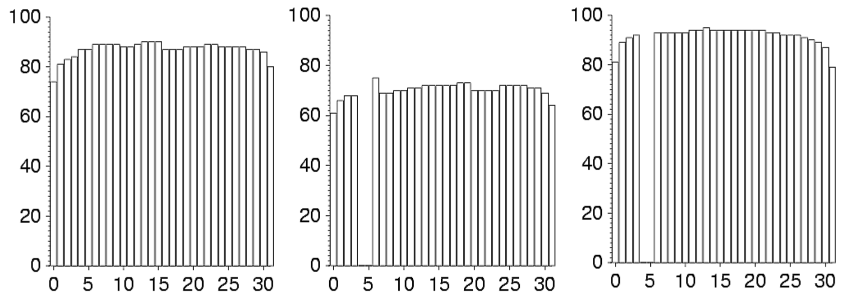

(a) Before warnings (b) After warnings (c) Post load balancing

Fig. 6. Utilization per processor for the 150^3 Sweep3d on 32 processors.

load more fairly among themselves and objects that communicate frequently are placed on the same processor, resulting in a much higher utilization (Figure 6(c)). These experiments verify that our protocol matches the goals laid out in Section 1.

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 [16], Starfish [17] and Clip [18].

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, optimistic logging [19] is rarely used; instead, pessimistic logging schemes are more frequently adopted, like in FT-MPI [20], MPI/FT [21], MPI-FT [22] and MPICH-V [23]. Causal logging (such as in [24]) 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, both checkpointing and message-logging impose some execution overhead even in the case of no faults, whereas our technique incurs no overhead if faults are not present. Other studies have proposed proactive fault-tolerant schemes for distributed systems [25], but no previous study has considered MPI applications.

6 Conclusion and Future Work

We have presented a new technique for proactive fault tolerance in MPI applications, based on the task migration and load balancing capabilities of Charm++ and AMPI. When a fault is imminent, our runtime system proactively attempts to migrate execution off that processor before a crash actually happens. This processor evacuation is implemented transparently to the application programmer. Our experimental results with existing MPI applications show that the processor evacuation time is close to the limits allowed by the amount of data in a processor and the kind of interconnect. The migration performance scales well with the dataset size. Hence, the fault response time is minimized, as required in our specifications in Section 1. Our experiments also demonstrated that MPI applications can continue execution despite the presence of successive failures in the underlying system. Load balancing is an important step to improve parallel efficiency after an evacuation. By using processor virtualization combined with load balancing, our runtime system was able to divide the load among the remaining fault-free processors, and application execution proceeded with optimized system utilization.

We are currently working to enhance and further extend our technique. We plan to bolster our protocol so that in the case of false positives it can expand the execution back to wrongly evacuated processors. We will also extend our protocol to allow recreating the reduction tree from scratch. We plan to investigate the associated costs and benefits and the correct moment to recreate the reduction tree. In addition, we will generate our fault warnings with information derived from temperature sensors in current systems.

Acknowledgments This work was supported in part by the US Dep. of Energy under grant W-7405-ENG-48 (HPC-Colony) and utilized parallel machines of NCSA

References

1. Gropp, W., Lusk, E., Skjellum, A.: Using MPI. Second edn. MIT Press (1999)
2. Gropp, W., Lusk, E.: Fault tolerance in message passing interface programs. International Journal of High Performance Computing Applications **18**(3) (2004) 363–372
3. 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.
4. Zheng, G., Shi, L., 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 (2004)
5. Chakravorty, S., Kalé, L.V.: A fault tolerant protocol for massively parallel machines. In: FTPDS Workshop at IPDPS'2004, Santa Fe, NM, IEEE Press (2004)
6. S. Chakravorty, C. L. Mendes and L. V. Kale: Proactive fault tolerance in large systems. In: HPCRI Workshop in conjunction with HPCA 2005. (2005)
7. Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba: Advanced configuration and power interface specification. ACPI Specification Document, Revision 3.0 (2004) Available from http://www.acpi.info.
8. Sahoo, R.K., Oliner, A.J., Rish, I., Gupta, M., Moreira, J.E., Ma, S., Vilalta, R., Sivasubramaniam, A.: Critical event prediction for proactive management in large-scale computer clusters. In: Proceedings og the ACM SIGKDD, Intl. Conf. on Knowledge Discovery Data Mining. (2003) 426–435

9. Oliner, A.J., Sahoo, R.K., Moreira, J.E., Gupta, M., Sivasubramaniam, A.: Fault-aware job scheduling for BlueGene/L systems. Technical Report RC23077, IBM Research ((2004))
10. Kalé, L.V., Krishnan, S.: Charm++: Parallel programming with message-driven objects. In Wilson, G.V., Lu, P., eds.: Parallel Programming using C++. MIT Press (1996) 175–213
11. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03), College Station, TX (2003)
12. Gioachin, F., Sharma, A., Chackravorty, S., Mendes, C., Kale, L.V., Quinn, T.R.: Scalable cosmology simulations on parallel machines. In: 7th International Meeting on High Performance Computing for Computational Science (VECPAR). (2006)
13. Kalé, L.V., Kumar, S., Zheng, G., 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 (2003)
14. Lawlor, O.S., Kalé, L.V.: Supporting dynamic parallel object arrays. Concurrency and Computation: Practice and Experience **15** (2003) 371–393
15. Antoniu, G., Bouge, L., Namyst, R.: 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, Springer-Verlag (1999) 496–510
16. Stellner, G.: CoCheck: Checkpointing and process migration for MPI. In: Proceedings of the 10th International Parallel Processing Symposium. (1996) 526–531
17. Agbaria, A., Friedman, R.: Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. Cluster Computing **6**(3) (2003) 227–236
18. Chen, Y., Plank, J.S., Li, K.: Clip: A checkpointing tool for message-passing parallel programs. In: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM). (1997) 1–11
19. Strom, R., Yemini, S.: Optimistic recovery in distributed systems. ACM Transactions on Computer Systems **3**(3) (1985) 204–226
20. Fagg, G.E., Dongarra, J.J.: Building and using a fault-tolerant MPI implementation. International Journal of High Performance Computing Applications **18**(3) (2004) 353–361
21. Batchu, R., Skjellum, A., Cui, Z., Beddhu, M., Neelamegam, J.P., Dandass, Y., 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, IEEE Computer Society (2001) 26
22. Louca, S., Neophytou, N., Lachanas, A., Evripidou, P.: MPI-FT: Portable fault tolerance scheme for MPI. Parallel Processing Letters **10**(4) (2000) 371–382
23. Bouteiller, A., Cappello, F., Hérault, T., Krawezik, G., Lemarinier, P., 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 (2003)
24. Elnozahy, E.N., Zwaenepoel, W.: Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. IEEE Transactions on Computers **41**(5) (1992) 526–531
25. Pertet, S., Narasimhan, P.: Proactive recovery in distributed CORBA applications. In: Proceedings of the International Conference on Dependable Systems and Networks. (2004) 357–366