Enabling Application Resilience With and Without the MPI Standard

Wesley Bland
Innovative Computing Laboratory, University of Tennessee, Knoxville
wbland@eecs.utk.edu

***Abstract*—As recent research has demonstrated, it is becoming a necessity for large scale applications to have the ability to tolerate process failure during an execution. As the number of processes increases, checkpoint/restart fault tolerance approaches requiring large concurrent state checkpointing become untenable and radically new methods to address fault tolerance are needed. This work addresses these challenges by proposing a novel approach to a minimalistic fault discovery and management model. Such a model allows application to run to completion despite fail-stop failures. As a proof of concept, in addition to the proposed fault tolerance model, an implementation in the context of the Open MPI library is provided, evaluated and analyzed.**

***Keywords*-Fault Tolerance, Message Passing Interface, Distributed Runtime**

I. Introduction

Fault tolerance is an increasingly necessary consideration in High Performance Computing (HPC). As machine sizes increase past hundreds of thousands of computing cores[1] into the millions of computing resources, the likelihood of failures also increases. In 2007, Schroeder and Gibson [1] announced a mean time between failure (MTBF) on some of the machines at Los Alamos National Laboratory of 8 hours. More recently, Heien et. al. [2] observed failures at a rate of between 1.8 and 3.6 failures per day on a system of only 635 nodes. This research confirms what has become an accepted reality of HPC going forward. Failures will occur at an increasing rate and for large scale applications to be useful, the failures will need to be handled in software while allowing the applications to continue running relatively uninterrupted.

The Open MPI Project is an open source MPI-2 implementation that is developed and maintained by a consortium of academic, research, and industry partners [2]. It has already made strides to include some fault tolerance support by developing checkpoint/restart and message logging frameworks, allowing users to recover processes that fail by reverting to a previously stored snapshot of the application. While this method of fault tolerance is useful for many applications, as the number of processes increases, the overhead of saving the state of as many as millions of concurrent processes becomes untenable and other methods of fault tolerance must be envisioned.

Our contribution in this area is to provide a resilient MPI implementation which gives the users the familiarity of the MPI standard which their applications already use, but also to expand the capabilities of MPI to handle process failures with as little intervention as possible from the application. In order to meet this goal, our work is two-fold:

- **Runtime Layer:** The runtime layer of Open MPI is the low level environment that supports the application by providing a reliable communication library. It is responsible for deploying applications and setting up an efficient application's communication topology. The runtime layer needs to be able to detect a process failure, notify other processes of the detected failure, and perform local recovery techniques necessary for the runtime layer to continue operation.
- **MPI Layer:** The MPI layer is the library which the application uses directly. It provides all the MPI standard calls while efficiently implementing communication patterns, such as collective operations. In the MPI layer, changes are necessary to notify the application of a process failure so it may decide whether to abort, continue without the failed process, replace the failed process or any other failure model that may be envisioned.

We also created a model to describe the expected behavior of our work. This model shows the efficiency of our work and explains why the performance of our implementation will never decrease because of a process failure.

In this paper, we will discuss of the work being done to expand the capability of Open MPI to handle process failures while allowing application developers to decide how to proceed. In Section II, we will describe some of the other resilient environments provided by the scientific community. Section III will describe the work being done to improve the Open MPI runtime to tolerate process failures. Section IV will describe the current work and ongoing efforts to give users a well-known MPI environment while causing minimal disruptions following a failure. Section V will demonstrate some of the preliminary results observed from this work and lay out the model we have constructed to analyze our result. Section VI will summarize the work being done and describe some of the future work yet to be performed.

[1] http://www.top500.org
[2] http://www.open-mpi.org

II. Related Work

Many groups have worked on fault tolerant runtimes in the past, each proposing different methods. Most of these runtimes focus on employing checkpoint/restart, but a few also use new ideas to recover from process failures.

Charm++ is an object-oriented parallel programming language developed at the University of Illinois at Urbana-Champaign [3]. In addition to message passing, it also performs load balancing, process migration and other interesting properties due to the fact that it treats each process as an individual object, called "chares", that can be saved and moved at any time. This has lead to work which leverages these chares to provide fault tolerance guarantees [4], including an MPI implementation which uses Charm++ as its runtime [5]. Our work diverges from Charm++ because they focus primarily not on MPI, but a separate programming environment of their own invention.

FT-MPI extended the MPI semantics provided by the MPI standard to include fault tolerance. This enabled application developers to adapt their applications without the need to rewrite them using an entirely different message passing system [6]. FT-MPI could withstand $n-1$ process failures in a job of size n. This is similar to the work being done in this paper as both are designed to recover from arbitrary fail-stop failures. However, FT-MPI only provides this semantic to the functions supported by the version 1.2 of the MPI standard. Our work is built in conjunction with Open MPI, which has ongoing development and includes support for the most recent MPI standard (2.2).

The MPI forum is currently examining options for the future direction of MPI for MPI-3. One of the workgroups is dedicated to propose a standard form of MPI-supported fault tolerance [3]. The proposal outlines a method of run-through stabilization which allows the application to acknowledge and repair communications, both collectively and between specific ranks in a point-to-point way [7]. The emphasis of the proposal is a set of "validation" functions which the application is required to call to repair and re-enable communication within an MPI communicator containing a failed process. These functions give the MPI implementation an opportunity to acknowledge failures and discover or ensure that other MPI processes also acknowledge the same failures. It also gives the MPI library a chance to repair communication channels between remaining processes, optimizing communication topologies if possible and necessary.

While this method of fault tolerance is sufficient for Algorithmic Based Fault Tolerance, it is not without its drawbacks. The calls necessary to recover from collectives incur a non-trivial overhead even during the fault free case. MPI_COMM_VALIDATE requires a distributed consensus algorithm which is currently best implemented at log scale [8]. While this level of overhead might exhibit better performance than the current state of the art of periodic checkpointing, it still presents a significant cost that not all applications want or need to pay Also, this proposal does not yet include process recovery, which is left to a future proposal to the MPI forum.

The work in this paper could be extended to implement the proposal from the MPI forum if accepted. However, it also includes more flexible options for the user when selecting a behavior to handle failures.

[3]https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/FaultToleranceWikiPage

III. Runtime

We present in this section how the Open MPI Runtime Environment (ORTE) [9] has been modified to become resilient as an example of the expected changes that need to be undergone on other runtime environments to provide resilient capabilities. We focus first on how the runtime itself has been made fault tolerant; then on the additional services that the runtime should provide to the MPI library or any other fault-aware parallel environment: Failure Detection and Notification.

A. Out-Of-Band Message Consistency Using Epochs

Before implementing process recovery, a way of tracking the status of processes is necessary. A commonly used method in literature is to add an epoch to the process naming scheme. By incrementing the epoch every time the *Head Node Process* (HNP) is notified of a failure, we can use it to track the number of times an individual process has failed. The runtime uses this epoch to prevent transient process failures from introducing unexpected behavior by cutting off a process with an epoch less than the most recent value from interfering with the other processes. As each message is processed by the communication library, it is checked against the most recent known epoch for the originating process. If the message's epoch is less than the most recently known epoch, the message is dropped and will need to be retransmitted to the new version of the previously failed process. This essentially imposes a fail-stop fault model [10] upon the processes, simplifying error detection and recovery.

B. Fault Handling

Concurrently with the notification of failure throughout the runtime, the HNP and the ORTE daemons (ORTEDs) also perform fault handling tasks to stabilize the runtime and allow processes to continue. The most noticeable portion of the runtime that must be updated is the routing layer. Most routing layers have some sort of underlying topology that passes messages from one node to the next rather than all messages being routed through a central entity, or allowing direct communication between nodes for the out-of-band messaging system. The latter would require n^2 opened connections, imposing a huge load on the system. This routing layer must be mended after any of the nodes

fail. One of the most common routing topologies is a tree. When the fault is detected, the tree must remove the faulty process and create connections from the failed process's parent to its children. This must also take into account any subsequent failures so that if necessary, the routing layer will continue to look upward or downward in the tree to find the closest living neighbor and prevent any child from becoming orphan due to a lack of connection to a living parent.

C. Failure Detection

Failure detection is accomplished using the existing detectors in ORTE. The primary detection method is to monitor the status of communication channels. If a connection fails, the ORTE error handler begins the process of managing the fault as detailed in Section III-B. In the future, when an MPI layer will be placed on top of the runtime, it will need to send errors to the runtime to allow errors to be handled in a consistent way. The current method of handling failures in Open MPI is to abort as soon as possible after detecting an error. By passing the error information to the runtime rather than acting within the MPI layer, that singular method of handling faults can be improved and the application may survive the fault.

D. Failure Notification

When a failure occurs, ORTE quickly attempts to stabilize the runtime system to allow the surviving processes to continue. The first step in this process is to notify the HNP. The HNP is responsible for maintaining the state of the application and notifying all runtime processes of any changes to which they need to respond. Once the HNP has received the message from the ORTEDs who detected the failure, it broadcasts this information to all other daemons. While some of them might already know about the failure, because they detected it via the direct connections between daemons, by including the epoch of the failed process they can prevent duplicate or out-of-order handling of the faults. At this point, the runtime would notify the MPI layer of the error, therefore allowing the MPI processes themselves to decide how the parallel application will react to the fault.

IV. MPI

The current MPI standard does not provide much guidance for applications in presence of faults. According to the standard, when implementing a high-quality MPI library, the application should regain control following a process failure. This control gives the application the opportunity to save its state and exit gracefully, rather than the usual behavior of being aborted by the MPI implementation itself. This makes continuing meaningful execution very difficult and usually requires the application to restart itself from a previously saved checkpoint.

Our work deviates from the current standard to provide a more flexible suite of tools to implement fault tolerance. If a process failed during an MPI call, the call will return an error code to reflect the failure. This allows the application to know that a process has failed and to perform any internal recovery operations necessary. Once the application has been alerted via the MPI return code, the application will not receive another notification until a new process fails. By not requiring an acknowledgment from the remaining processes, our method of fault tolerance imposes as little burden on the applications as possible and allows failure free executions to incur the minimum amount of overhead.

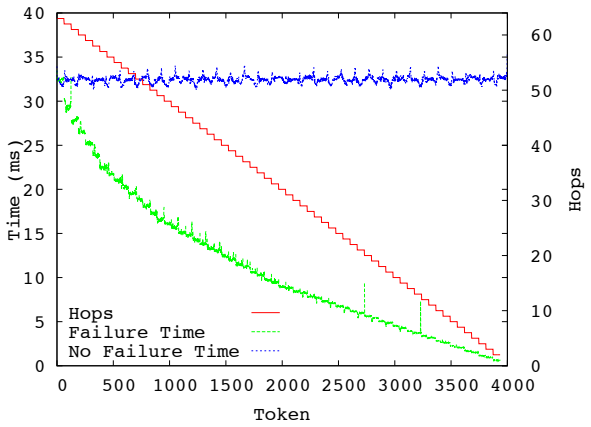
As most of the MPI applications take advantage, in addition from point-to-point message, of collective communications, we took a particular interest in providing a clear semantic of how fault can integrate with collective communications. We are implementing fault tolerant collective operations which allow the application to run collectives over MPI_Communicators including failed processes. For collectives which do not include any data, such as MPI_Barrier, this is simple enough operation. For collectives which require data combination, such as MPI_Gather or MPI_Reduce, this can be a slightly more complicated task. However, all of the MPI collective operations can eventually be simplified to a communicator pattern with some amount of data, which may or may not be combined with or without an operation in the process. When determining how many processes to include in the collective operation, our MPI library does not include the failed processes. Also, when a process fails during a collective, the operation is updated to remove the failed process. Once the participating group is determined, we can continue the MPI collective as if no failures occurred.

This work will also eventually include process recovery which will allow the application to decide to recreate the failed process by launching a new MPI rank on a fault free node. The user will be responsible for bringing the new rank back to the point of the failure, but this will support another set of applications which require a specific number of processes to continue in the presence of a process failure. The specifics of the process recovery techniques are not yet ready for publication.

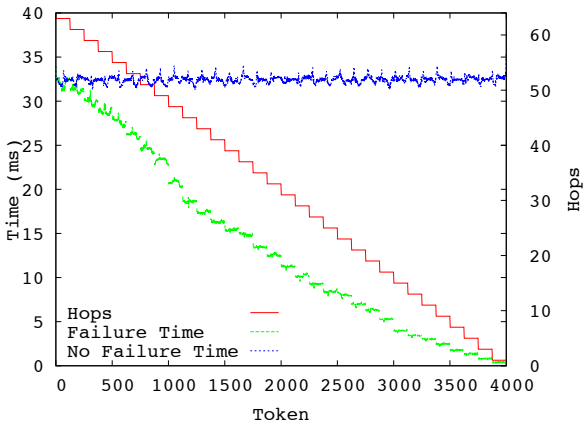
V. Performance Discussion

In this section we describe the results of our work to this point. First we present a model to describe the expected results of our resilient runtime's failure notification method. Second, we will demonstrate the observed values derived from our tests performed on a 64 node cluster within Grid5000. [4]

[4]Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr)

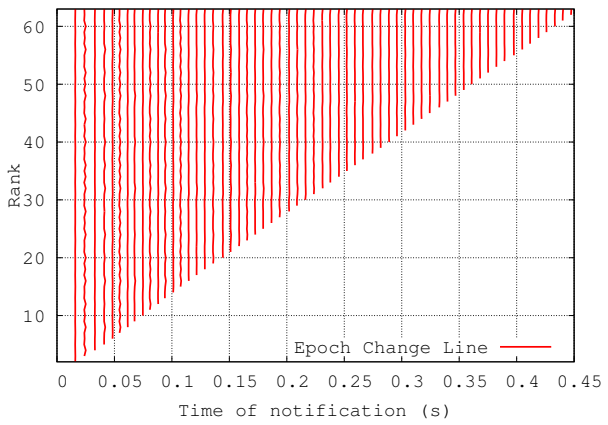


(a) One Process failing at a time

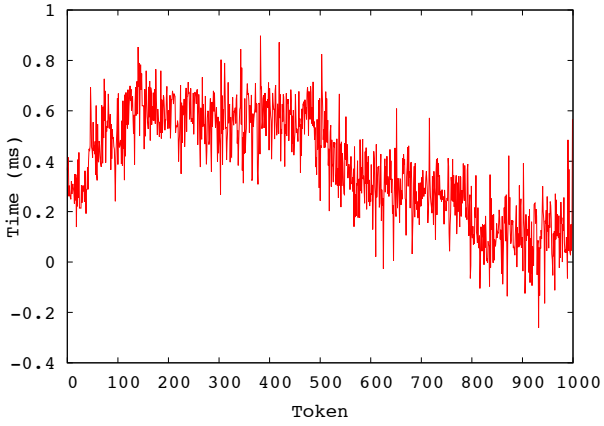


(b) Two concurrently failing processes

Figure 1. Token Roundtrip Time with Evenly Failing Processes



(a) Detection Time for Failures



(b) Overhead Introduced by Changes

Figure 2. Overheads of the Resilient Runtime

A. Performance Model

We devised a model of the expected failure detection behavior of our runtime system. Let T be the initial routing topology of the system. Let $L_T(d, f, H)$ be the time it takes process d to notice a failure of process f, knowing that $\forall q \in H$, q is also failed (potentially before f). We call $L_T(d, f, H)$ the latency of detection of d for f knowing H.

$$L_T(d, f, H) \leq \alpha_0 + \gamma D_{T \setminus H \cup \{f\}}(f, d)$$

Where α_0 is the time taken by an immediate neighbor of f to detect its failure, γ is the point-to-point message latency, and $D_{T \setminus H \cup \{f\}}(f, d)$ is the distance (in number of hops) between f and d in the routing topology, $T \setminus H \bigcup \{f\}$. Note that this distance could shrink as processes between f and d in the routing topology fail.

Whatever the history of failures H, if $|T| = n$, then $D_{T \setminus H}(s, d) \leq \log_2(n)$. This is true because the routing layer we use is a binomial tree, which has a maximum depth of $log_2(n)$. Also, when repairing the routing tree after a failure, the routing layer does not add new hops, it only mends the routes by removing failed processes from the topology and creating links between the next surviving neighbors both above and below f. Thus, for system $Bin(n)$ with n processes initially, with a fault tolerant binomial tree routing layer,

$$\forall d, f \in Bin(n), \forall H \subseteq Bin(n),$$
$$L_{Bin(n)}(d, f, H) \leq \alpha_0 + \gamma \log_2(n)$$

This shows that our improved routing topology should perform no worse than a fault free one following a process failure.

B. Performance Data

When validating the runtime work described in Section III, our goal was to devise a test that would confirm its

correctness while being simple to describe and understand. It should be emphasized that this is not necessarily representative of a specific application, but it does demonstrate that the improved runtime would be able to support a full application in the presence of a process failure. It is designed to show that the runtime can heal its routing layers following multiple and catastrophic process failures. Once a resilient MPI layer is built on top of the runtime, a full application could be tested with MPI semantics.

Our test is a ring test which dispatches multiple tokens from process 0. These tokens are passed around the ring in increasing sequential order until they reach process 0 again where various measurements are made. This is a simplistic test and therefore not designed to demonstrate the recovery of a lost token, however the test could easily be modified to demonstrate that behavior as described by Hursey [11]. Process 0 generates 4000 tokens and passes them along. As the tokens are passed around the ring, the test generates failures at predetermined times to give an idea of the behavior of the runtime with different failure patterns. The loss of tokens gives an idea of how long the routing layer took to patch itself and continue sending tokens to the next living process.

Figure 1 shows the round trip time of each of the tokens as the failures are generated within the system. Each graph shows both the time from a failure free run and a run with a specific failure pattern. The graphs also show how many hops each token traversed while traveling around.

In Figure 1(a), processes fail one at a time in evenly spaced intervals until only 2 processes remain. As each process fails, the round trip time decreases linearly. This is the expected behavior as each token must traverse fewer processes until it reaches process 0 again. The number of hops decreases slowly as each process fails, but the round trip time with failures actually decreases at a greater rate temporarily. This is because the buffers at the first few processes become full at the beginning of the run when the tokens are still being generated. As the buffers clear the first few processes, the round trip time stabilizes.

In Figure 1(b), processes fail in groups of two. This shows that the runtime can withstand larger groups of processes failing at roughly the same time. The gaps in the graph show the points at which some tokens are lost. This is expected as the application makes no attempt to recover or regenerate tokens temporarily hosted by failed processes, and simply continues to run with whatever tokens return back to the originating process.

Our runtime is also able to handle failure rates much higher than one or two processes at a time. In other tests, we were able to handle rates of $n/2$ or $n-2$ simultaneous failures. This is encouraging as it shows that the runtime can withstand any number of failures at any time. The failures can occur concurrently or consecutively without requiring a "cooling off" period between failures.

Figure 2(a) shows the detection and notification time for each failure. It uses the same case as figure 1(a) where failures occur evenly throughout the lifetime of the application. Each line represents the detection time of each epoch among all the processes, collected using the ad-hoc fault handler. When the line is straight, it demonstrates that the latency of detection and notification is relatively low among all the remaining process. This figure shows that the detection time for the all processes is very tight, demonstrating that all processes can maintain a consistent view of the current epoch. The earlier epochs have a slightly varying detection time as the notification messages have further to travel through the routing layer, but as the tree becomes smaller as it is repaired, the oscillations decrease, demonstrating a much smaller window of time for each epoch.

Figure 2(b) shows the overhead that was introduced by modifying the Open MPI source code. It compares the runtime of a fault-free case using both our implementation of the Open MPI runtime, and the revision 24614 of the Open MPI trunk. The overhead was measured on a local 8 node development cluster with minimal system noise to eliminate outside effects on the data. We subtract the round trip time for each token in the resilient version of Open MPI from the same test using the trunk version of Open MPI. The results show that the changes made to the code actually had little impact on performance in the fault-free case. Variations can be explained by the network jitter and the small increase in the header size of the messages due to the epoch algorithm.

VI. Concluding Remarks & Future Work

This work has outlined some of the improvements being performed within Open MPI to create a fault tolerant environment in order to allow applications to continue their execution in the face of process failures. Our work requires modifications to the runtime environment to stabilize message routing and provide reliable fault detection and notification.

These improvements provide the tools necessary for application developers to implement algorithms that can run reliably at a larger scale than is currently possible due to the inability to handle the inherent failures which occur at such extreme scales. Developers have new and more diverse options to allow them to choose the appropriate resilience method for their application.

In the near future a system for process recovery will be included to allow applications not only to stabilize themselves after a process failure, but to replace the failed process with a new one. Such a process can use other forms of fault tolerance (checkpointing, message logging, etc.) to recover lost data and continue with minimal interruption to the living processes.

REFERENCES

[1] B. Schroeder and G. Gibson, "Understanding failures in petascale computers," in *Journal of Physics: Conference Series*, vol. 78. IOP Publishing, 2007, p. 012022.

[2] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello, "Modeling and tolerating heterogeneous failures in large parallel systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 45:1–45:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063444

[3] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based On C++," in *In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applicaions*, 1993, pp. 91–108.

[4] G. Zheng, L. Shi, and L. V. Kale, "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *Cluster Computing, 2004 IEEE International Conference on*, 2004, pp. 93–103.

[5] C. Huang, G. Zheng, and L. V. Kale, "Supporting Adaptivity in MPI for Dynamic Parallel Applications," Tech. Rep. 07-08, 2007.

[6] G. Fagg and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," *EuroPVM/MPI*, 2000.

[7] J. Hursey, R. L. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, and D. G. Solt, "Run-through stabilization: An MPI proposal for process fault tolerance," in *EuroMPI 2011: Proceedings of the 18th EuroMPI Conference*, Santorini, Greece, September 2011.

[8] J. Hursey, T. Naughton, G. Vallee, and R. L. Graham, "A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI," in *EuroMPI 2011: Proceedings of the 18th EuroMPI Conference*, Santorini, Greece, September 2011.

[9] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G. E. Fagg, "The Open Run-Time Environment (OpenRTE): A transparent multicluster environment for high-performance computing," *Future Generation Computer Systems*, vol. 24, pp. 153–157, 2008.

[10] M. Fischer and N. Lynch, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, 1985.

[11] J. Hursey and R. Graham, "Building a Fault Tolerant MPI Application: A Ring Communication Example," *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1549–1556, 2011.