Fault Tolerant Runtime for Open MPI

Wesley Bland[1], George Bosilca[1], Thomas Herault[1], Jack J. Dongarra[1,2,3]

[1] Innovative Computing Laboratory, University of Tennessee, Knoxville
{wbland, bosilca, herault, dongarra}@eecs.utk.edu
[2] Oak Ridge National Laboratory, USA
[3] University of Manchester, UK

Abstract. If the MPI library, or any other parallel environment, expects non-failed processes to continue their execution despite failures, it is necessary that the runtime supporting the parallel execution environment becomes resilient to such failures. Such a fault tolerant runtime is required to remain capable of providing its services, in addition to offering additional services related to faults. Thus, runtime environments have a key role to play in resilient capabilities of an MPI implementation. In this paper, we introduce a minimalistic set of basic services related to fault handling, and we present and evaluate a resilient runtime environment, adapted from ORTE, the runtime environment of Open MPI.

1 Introduction

Fault tolerance is an increasingly necessary consideration in high performance computing (HPC). As machine sizes increase past hundreds of thousands of computing cores[4] into the millions computing resources, the likelihood of failures also increases. Schroeder and Gibson [9] announced a mean time between failure (MTBF) on some of the machines at Los Alamos National Laboratory (LANL) of 8 hours. This work was published in 2007 and the MTBF of large-scale machines is only expected to shrink as the machine size grows.

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 [5]. 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.

Other approaches, not based on coordinated checkpointing, have been proposed to tackle this issue, each one addressing the fault at a different level in the software stack. In Algorithmic Based Fault Tolerance approaches [8], the

[4] http://www.top500.org
[5] http://www.open-mpi.org

surviving processes will regenerate the data, and continue the computation with substitute processes; other times (e.g. using uncoordinated checkpointing with message logging [1]), the MPI system itself will start substitute processes and provide them with the information needed to ensure the consistency of the recovery. In other situations (e.g. a Monte-Carlo simulation), failed processes can simply be ignored, and the computation continue without replacing them or recovering their data. The common point between these more scalable approaches is that non-failing processes need to continue participating in the application, and therefore need support from their managing runtime to achieve this goal.

To ensure this continued progression, the MPI environment relies on a runtime system. In this paper, we focus on the first building block necessary to produce a resilient MPI implementation: a resilient runtime for MPI. The responsibilities of the runtime system with respect to resiliency are twofold:

Fault Detection Failure detection requires, in many systems, to actively probe the reactivity of processes. Since the application processes may enter computation-intensive phases for an arbitrary duration, another level must be responsible to answer these probes, serving as witness of activity for remote monitoring processes. In most environments, the runtime processes are best suited to accomplish this role.

Fault Notification Once a failure is detected, all processes that interact with the failed processes (this is often all processes of the system, by closure of the connected relationship as defined by the MPI standard) must be eventually notified in order for them to take correcting actions. This notification could have happened inside the MPI library, but because the runtime system must already provide a resilient out-of-band communication mechanism, we consider that this is a role that it can better fulfill.

Once the application is notified of failures, it will take corrective actions. These actions (like starting new processes to replace the failed ones) may require services from the runtime level to work. Even if no special corrective actions are necessary, the runtime system needs to continue providing its basic services (I/O forwarding, termination and cleaning of the application, out-of-band messaging system) to the parallel application until its normal termination point is reached.

In this paper, we present how an MPI runtime system can be made resilient, to continue providing its services in presence of fail-stop failures. We evaluate the performance of this runtime system using ad-hoc applications by comparing it with the performance of a non-resilient runtime system and through micro-benchmarks in the presence of different scenarios of faults.

Section 2 of this paper will outline previous work in this field as well as describe the Open MPI architecture. Section 3 will present the changes we have introduced to implement the resilient layer of the runtime system. Section 4 will showcase the performance of the work, and Section 5 will summarize the work and point to future research.

2 Related Work

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

Charm++ is an object-oriented parallel programming language developed at the University of Illinois at Urbana-Champaine [7]. 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 [10], including an MPI implementation which uses Charm++ as its runtime [6].

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 [3, 4]. 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.

The MPI forum in currently in the process of writing the new MPI specification, MPI 3. Expected in this version of the specification is language to describe how MPI applications should handle process faults including both stabilization and recovery of failed processes. This work is being done publicly and progress can be found on the MPI 3 Fault Tolerance Working Group's website[6].

3 A Resilient Runtime

We present in this section how the Open MPI Runtime Environment (ORTE) [2] has been modified to become resilient as an example of the expected changes that need to be undergone on 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 its user (the MPI library or any other parallel environments): Failure Detection and Notification. The modified runtime is capable of running any MPI application. However, since the MPI level does not provide a stabilization interface yet, the MPI application cannot use the additional services to continue its execution.

3.1 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. The epoch tracks the number of times a process has

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

failed by incrementing the epoch every time the *Head Node Process* (HNP) is notified of a failure. The runtime uses this epoch to prevent transient process failure 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 forces a fail-stop fault model [5] upon the processes, simplifying error detection and recovery.

3.2 Fault Handling

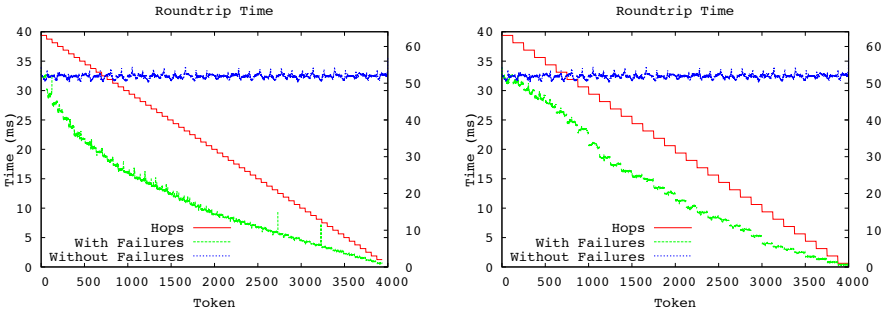
Concurrently with the notification of failure throughout the runtime, the HNP and the ORTE daemons (ORTEDs) also perform fault handling tasks to stabilize the runtime and allow processes to continue. The most noticeable portion of the runtime that must be updated is the routing layer. Most routing layers have some sort of topology that passes messages from one node to the next rather than all messages being routed through the HNP. This routing layer must be mended after any of the nodes fails. 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.

3.3 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 3.2. In the future, when an MPI layer will be placed on top of the runtime, it will need to send errors to the runtime if it wants the errors to be handled in a consistent way. The current method of handling failures in 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.

3.4 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



(a) One Process failing at a time (b) Two concurrently failing processes

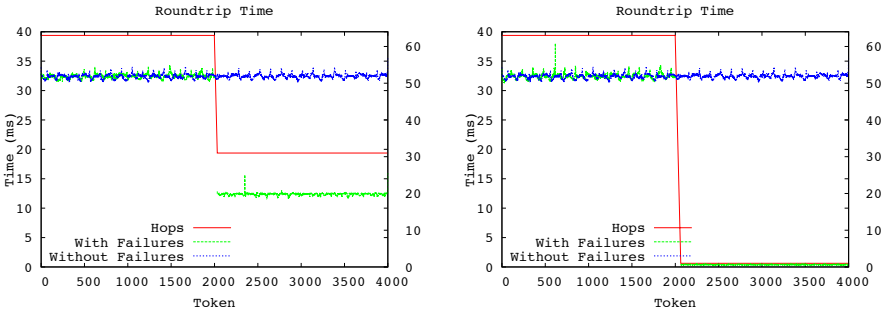
Fig. 1: Evenly Failing Processes

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.

4 Results

To test our implementation of the fault tolerant runtime, we use a simple ring and token test. Rank 0 generates a group of tokens and periodically sends one around the ring. As each rank receives the token, it immediately passes it to the next rank in the ring. After a parameterized number of tokens passes a rank it simulates a process failure. We use this test with a variety of failure patterns to demonstrate the performance impact of multiple failures on the runtime. Our runtime can withstand a failure from any process at any time other than the HNP. This is an acceptable constraint however because while the probability of losing *any* node is high, the probability of losing a *specific* node is relatively low. This is increasingly true as individual machine reliability improves. Because there are currently no MPI semantics for fault tolerance, we can only use runtime level communication and process management. Our tests were run using 64 nodes on Grid5000.[7] Rank 0 generated 4000 tokens to be sent around the ring. After

[7] 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) N/2 Processes Failing at Midpoint (b) N-2 Processes Failing at Midpoint

Fig. 2: Many Processes failing together

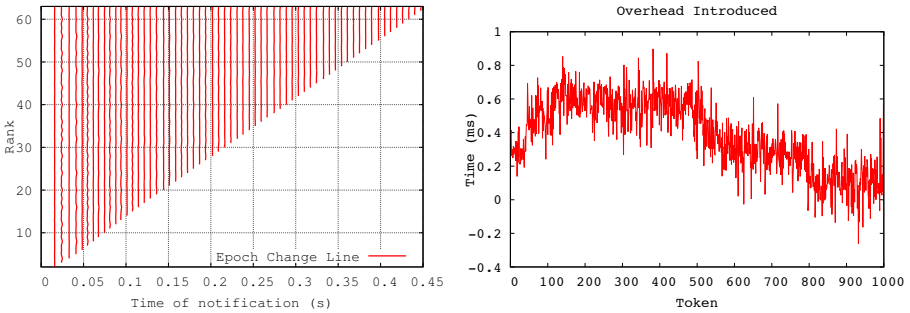
completing a circuit of the ring, rank 0 recaptures the tokens and gather data about the round trip time and number of hops.

The ring token test has been written as an ORTE application. After it initializes the ORTE subsystem, each rank reads a scenario file, that defines how failures will happen (if any), and registers a fault handler that will take statistics on the time of failures notifications. Then, the root process (process of `vpid` 0) starts sending the tokens, and other processes forward them according to the ring algorithm. After the scenario has been completed, each process still alive finalizes the ORTE subsystem, and prints statistics of the run.

In Figure 1a, 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 ranks until it reaches rank 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 ranks become full at the beginning of the run when the tokens are still being generated. As the buffers clear the first few ranks, the round trip time stabilizes into a linear speedup.

In Figure 1b, processes fail in groups of two. This shows that the runtime can withstand larger groups of processes failing at roughly the same time. This would simulate a two processor node failing at once. 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 rank 0.

Figure 2a introduces much larger failures at a time. In this test, half of the processes fail all at once at the midpoint of the run. While it takes the runtime some time to discover all of the failures, as illustrated by the larger number of lost tokens, it does eventually recover from the losses and any tokens that were



(a) Detection Time for Failures (b) Overhead Introduced by Changes

Fig. 3: Overheads of the Resilient Runtime

not on one of the failed processes at the time of the failure continue around the ring until rank 0 receives them again.

Figure 2b introduces a critical failure. In this test, all but two processes are killed at the same time at the midpoint of the run. Again, the runtime requires a discovery period before stabilizing, but it is able to recover and continue delivering the remaining tokens back to rank 0. Figures 2a and 2b are encouraging as they show that the runtime can withstand any number of failures at any time. This implementation can withstand concurrent or consecutive failures without requiring any minimal "cooling off" period between failures.

Figure 3a shows the detection and notification time for each failure. It uses the same case as figure 1a where failures occur evenly throughout the lifetime of the application. Each line represents the detection time of each epoch among all the ranks, collected using the ad-hoc fault handler. To reduce the effect of clock drift, the time is measured from the beginning of the local application rather than attempting to synchronize with a single value from rank 0. When the line is very straight, it demonstrates that the latency of detection and notification for the failure 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.

Figure 3b 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.

5 Conclusion and Future Work

This paper has demonstrated a runtime system that is able to quickly detect and recover from multiple process failures. The runtime stabilizes its internal accounting and passes failures on to an application layer through callback functions to be invoked following a process failure notification. Experiments, based on an ad-hoc application, have demonstrated that the runtime is able to tolerate an arbitrary number of failures while still providing its basic services and adding fault-related services.

This demonstrates the feasibility of such a runtime, and the fact that resilience can be achieved based on series of basic building blocks that will allow developers who wish to implement a fault tolerant MPI to easily add their own MPI level fault tolerant capabilities on top of this runtime. In the near future, Open MPI will be modified to implement the fault handler "`MPI_ERRORS_RETURN`", or other user-level handler, using these features.

As a further goal, we plan to use this runtime layer to implement anticipated changes to MPI 3 including run-through stabilization. More changes could be implemented at the runtime level including a method for process recovery to give applications more options in their reactions to faults. With recovery, applications could choose either to ignore failed processes or create new processes to replace the failed ones. This introduces a new set of interesting problems at the runtime layer, such as the restoration of processes to routing layers, and allows for new methods of recovery such as uncoordinated rollback. At the MPI layer, restoring processes becomes even more complex and requires that the MPI processes perform appropriate steps to recover it failed neighbors. This work continues in the MPI Forum and will hopefully be adopted soon.

References

1. Bouteiller, A., Cappello, F., Herault, T., Krawezik, G., Lemarinier, P., Magniette, F.: MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In: Supercomputing Conference (2003)
2. Castain, R.H., Woodall, T.S., Daniel, D.J., Squyres, J.M., Barrett, B., Fagg, G.E.: The Open Run-Time Environment (OpenRTE): A transparent multicluster environment for high-performance computing. Future Generation Computer Systems 24, 153–157 (2008)
3. Fagg, G.E., Dongarra, J.: FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In: Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. pp. 346–353. Springer-Verlag, London, UK (2000)
4. Fagg, G.E., Gabriel, E., Bosilca, G., Angskun, T., Chen, Z., Pjesivac-Grbovic, J., London, K., J Dongarra, J.: Extending the MPI Specification for Process Fault Tolerance on High Performance Computing Systems. International Conference on Supercomputing pp. 1–33 (2004)
5. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32, 374–382 (April 1985), http://doi.acm.org/10.1145/3149.214121

6. Huang, C., Zheng, G., Kale, L.V.: Supporting Adaptivity in MPI for Dynamic Parallel Applications. Tech. Rep. 07-08 (2007)
7. Kale, L.V., Krishnan, S.: CHARM++: A Portable Concurrent Object Oriented System Based On C++. In: In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applicaions. pp. 91–108 (1993)
8. Nair, V., Abraham, J., Banerjee, P.: Efficient techniques for the analysis of algorithm-based fault tolerance (ABFT) schemes. Computers, IEEE Transactions on 45(4), 499 –503 (apr 1996)
9. Schroeder, B., Gibson, G.: A Large-Scale Study of Failures in High-Performance Computing Systems. IEEE Transactions on Dependable and Secure Computing 7(4), 337–351 (oct 2010)
10. Zheng, G., Shi, L., Kale, L.V.: FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. Cluster Computing, IEEE International Conference on 0, 93–103 (2004)