Toward Message Passing Failure Management

Wesley Bland, Jack J. Dongarra (Advisor)
Innovative Computing Laboratory, University of Tennessee, Knoxville
wbland@eecs.utk.edu, dongarra@eecs.utk.edu

I. Motivation

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 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.

This realization has lead to much research to attempt to solve the problems presented by the necessity for fault tolerance as indicated in [3]. The first and most well understood form of fault tolerance is rollback-recovery using periodic checkpointing. This form of fault tolerance has been widely adopted and works well for small scale machines where failure rates are expected to be relatively low. However, at larger scales, even with more reliable hardware, the time spent performing checkpointing operations is expected to exceed the amount of time spent performing useful computation. To resolve this problem, we turn to Application Based Fault Tolerance (ABFT).

ABFT changes the way applications recover from failure. Rather than loading a previous checkpoint from disk and restarting an entire application, the algorithm itself recovers from the loss of a process and continues without the need to perform costly, large-scale checkpointing operations. The exact method used to recover from failures changes from application to application, but all applications have some requirements in common. The programming model and environment, together with the supporting runtime, need to provide basic functionality in order to allow applications to build a comprehensive fault management solution.

This is a challenging requirement. Most applications continue to use message passing to perform communication between processes, and in the message passing paradigm, collective communication is a popular and necessary way for groups of processes to communicate efficiently. However, collective communication also creates problems when trying to maintain the functionality of the communication library following a process failure due to the complex communication patterns and topologies that must be repaired.

In addition to maintaining a functional communication library with scalable fault tolerance mechanisms, a fault tolerance solution must also provide extensibility. Because ABFT takes different forms for different applications, the fault tolerance provided by the communication library should also be able to adapt. For example, while one type of application may work best with a transactional model of fault tolerance where sections of the application are re-executed when recovery is necessary, a master-slave type of application may choose to simply spawn a replacement for any process which fails and continue on without needing further recovery.

To this end, we have modified a runtime system to support two new forms of fault tolerance within the message passing paradigm which can provide a suite of tools necessary for developers to include resilience in their applications.

II. Runtime

This section summarizes the modifications that were necessary for us to create our resilient runtime layer. Details of our implementation of the work outlined in this section can be found in [4]. The major requirements fall into three categories: failure detection, failure notification, and failure handling.

A. Failure Detection

While failure detection can take place throughout an applications layers, the runtime layer is the most well positioned to discover failure quickly. Because it manages process launching and communication channels, it can quickly detect failures. Once it detects a failure, it can begin the process of failure notification.

B. Failure Notification

When a failure is detected, the runtime environment is responsible for propagating knowledge of the failure to all appropriate parts of the application. This includes the runtime environments on other nodes, as well as to the MPI layer of the application.

[1]http://www.top500.org

C. *Failure Handling*

Concurrently with failure notification, the runtime must also perform failure handling tasks to stabilize the runtime environment and allow the application to continue. This includes route healing, where communication channels within the runtime environment must adapt to losing a node as a potential routing hop. It also includes ensuring message consistency to prevent repeated failure notification and to prevent loss of messages as much as possible.

III. MPI

Once the runtime work described in Section II is put in place, more options for fault tolerance in message passing unfold. This section describes two new fault tolerance methods we developed: a method contained in the Message Passing Interface (MPI) standard version 2.2 and one which goes beyond the current standard.

A. *Checkpoint-on-Failure*

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 but the state of the library is undefined. This makes continuing meaningful execution very difficult and usually requires the application to restart itself. However, for applications which can take advantage of it, this provides the opportunity to perform last chance checkpointing. Using a technique called Checkpoint-on-Failure (CoF) [5], applications can register an MPI error handler to receive notifications about process failures. When the error handler is triggered, the application saves a minimal amount of state to stable storage, restarts itself with the correct number of processes, and performs the necessary recovery operations to continue execution. The advantage of this approach is that while it does require some code changes to support ABFT, it has similar semantics to existing checkpointing methods while removing the necessity of discovering the optimal checkpoint interval. By only checkpointing after a known failure, the checkpoint is always taken optimally.

B. *User Level Failure Mitigation*

For some applications, CoF is not the best method of failure recovery. These applications may require recovery models that allow the application to continue execution. To support these applications, another method of MPI level fault tolerance has been created name User Level Failure Mitigation (ULFM) [6]. ULFM is more extensive than CoF and requires modifications to the current MPI standard. While ULFM contains many pieces, it revolves around three new MPI functions:

- **MPI_COMM_REVOKE:** The revoke operation prevents deadlock either at the application level or the MPI library level by stopping further communication on a communicator containing a failed process.
- **MPI_COMM_SHRINK:** The shrink operation allows continued collective communication by creating a new communicator from a communicator containing known failed processes while leaving out any known failures.
- **MPI_COMM_AGREE:** An agreement operation provides a mechanism to create a consistent view between processes so a decision can be reached when such a strong consistency is required (e.g. during algorithm completion).

IV. Applications and Libraries

These two methods of handling failures target two different communities. CoF is designed to be used directly by applications to recover from failures. In [5], we discuss our use of CoF in a linear algebra QR algorithm to allow recovery during large matrix factorizations while introducing low levels of overhead. This technique can be applied to other applications which can support Application Based Fault Tolerance (ABFT).

ULFM provides a more extensible interface which allows library developers to create new levels of fault tolerance. It provides the minimal complete set of tools necessary for recovery. For example, a library may choose to provide collective operations which provide consistent return codes to all processes. While this would be an expensive burden to place on applications which do not require it, by providing library developers the tools necessary to implement such fault tolerance, applications that choose to can have the form of consistency they need.

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] F. Cappello, "Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, July 2009.

[4] W. Bland, "Enabling application resilience with and without the mpi standard," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, May 2012.

[5] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "A checkpoint-on-failure protocol for algorithm-based recovery in standard mpi," August 2012.

[6] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. Dongarra, "An evaluation of user-level failure mitigation support in mpi," September 2012.