Improved Failure Detection and Propagation Mechanisms for MPI

Pedro Henrique Di Francia Rosso[1], **Emilio Francesquini**[1]

[1]Universidade Federal do ABC (UFABC)

pedro.rosso@ufabc.edu.br, e.francesquini@ufabc.edu.br

Abstract. *The MPI standard is largely used in HPC systems. Such systems employ a large number of computing nodes. Thus, Fault Tolerance (FT) is a concern since a large number of nodes leads to more frequent failures. Two essential components of FT are Failure Detection (FD) and Failure Propagation (FP). This paper proposes improvements to existing FD and FP mechanisms, like initial position shuffling, false-positive detection, and a chord-like propagation, to provide more portability, scalability, and low overhead. Results show that the methods proposed can achieve better or at least similar results to existing methods while providing portability to any MPI standard-compliant distribution.*

1. Introduction

MPI (Message Passing Interface) is a standard largely used in HPC (High-Performance Computing) environments nowadays. In HPC environments, Fault Tolerance (FT) is a common concern since a large number of nodes ultimately leads to an increased number of failures (the Mean Time Between Failures (MTBF) can be as low as a few hours[4])). Two significant components present in fault-tolerant systems are the Failure Detection (FD) and the Failure Propagation (FP), used to report, at any time, the state (`DEAD` or `ALIVE`) of all processes in an application [1]. In many cases, these solutions can incur a high-performance overhead and high latency for detecting and propagating failures. In terms of scalability, for example, they can impact the overall performance of the applications [1].

This work[1] proposes changes to existing FD solutions and a new algorithm for FP. The main objectives are to address portability (FD and FP must work out-of-the-box with different MPI distributions), scalability, and low overhead.

2. Background and Related Work

Fault Tolerance (FT) is an active research subject in HPC systems. This paper focuses on two of the fault tolerance components, the FD and FP mechanisms.

Usually, for FD, some monitoring system is employed. Several different approaches have been employed as FD mechanisms. Some use daemon processes (one per node) to monitor other processes in the same node (via OS signals). These daemon processes are monitored (addressing node failures), either by the root process [5] or by heartbeat messages exchanged between the daemons [7]. Others rely on the `slurmd` daemon to detect process failures via exit codes [2]. The gossip-style approach is based on message exchanges, where the group to exchange alive messages is randomly selected [3]. However, this random approach was shown not to be scalable [7]. Finally, the heartbeat ring approach works by distributing processes throughout a ring and exchanging messages between neighbors [1]. This paper focuses on the heartbeat ring FD mechanism. The approaches that rely on OS-related mechanisms depend on specific tools, such as PMIx or the Slurmd tool. These approaches do not provide portability and are incompatible with our proposal that aims to be usable with any standard-compliant MPI implementation.

[1]The authors are grateful to the Center of Petroleum Studies (CEPETRO-Unicamp/Brazil) and PETROBRAS S/A for the support to this work as part of OmpCluster Project (https://ompcluster.gitlab.io/).

Figure 1. Heartbeat ring topology

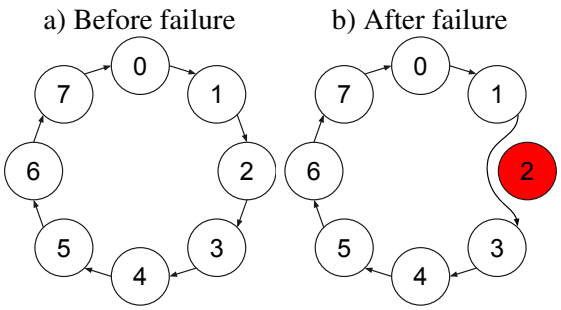
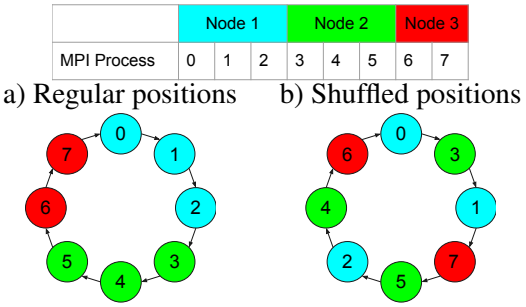


Figure 2. Shuffled positions



For FP, it is common to employ a reliable broadcast mechanism. The hyper-cube broadcast algorithm (HBA) is an option in which a propagator sends the broadcast to the 2^k processes forward and backward in the ring [1] (with $k = \lfloor log_2(N) \rfloor$, and N the number of processes). Circulant graphs, like binominal graphs (BMG), are a good option to achieve a balance between propagation delay (time for the propagation to achieve all processes) and scalability (number of exchanged messages). In BMG, the broadcast is sent to processes following powers of two (*i.e.*, a process with rank k sends messages to processes with ranks $k + 2^0, k + 2^1, k + 2^2, ...$), forward and backward in the ring. This approach creates redundant messages since a process can be selected more than once (one time forward and backward in the ring) [7]. As it is also the case with FD, gossip protocols are also an option for FP. Propagation is fast but, in the worst case, some processes may never be reached [7]. In this work, we propose an adaptation of the Chord protocol [6] to make FP, which can be a better alternative to current FP schemes. Our approach is capable of improving scalability while still providing low propagation delays.

3. Improved mechanisms

In the ring heartbeat topology, each process is an `emitter` (sends heartbeat messages to the successor process in the ring) and `observer` (receives heartbeat messages from the predecessor process in the ring), as shown in Figure 1-a), where node 0 is emitting to node 1 and observing node 7, and so on. Two properties control the heartbeats: the `period` δ (time between the heartbeat messages); and `timeout` η (time before the observer considers the emitter failed). After the timeout, the ring is recomposed: the process that was emitting to the failed process waits for a new observer, that will be the process that observed the failure. Figure 1-b) shows this mechanism after node 2 fails.

In this model (the same used by Bosilca et al. [1]), there is no differentiation between individual processes and nodes. Thus, when a node with several processes fails, if those processes are observing each other (like shown in Figure 2-a), there will be a $p \times \eta$ time delay to detect all failures, where p is the number of failed processes. Unfortunately, this problem is not rare since this distribution of processes, depending on the configuration, might be the default. To tackle this, we propose randomizing the initial position of processes in the ring. This is shown in Figure 2-b), which might not be optimal, but it is a simple and scalable solution with good results in practice as shown in Section 4.

Another concern FT mechanisms have to take into consideration is false-positive failures. In HPC environments, disputes for resources or even intermittent failures may cause delays in the heartbeat. The algorithm by Bosilca et al. does not deal with false-positive failures. We propose to make the observer watch messages from any source other than the emitter. If the heartbeat is received from a non-emitter process (that was a previous emitter), it means that it was falsely detected as a failed process. Thus we add the process back to the ring between the observer and the emitter.

Figure 3. Accuracy and overhead evaluation

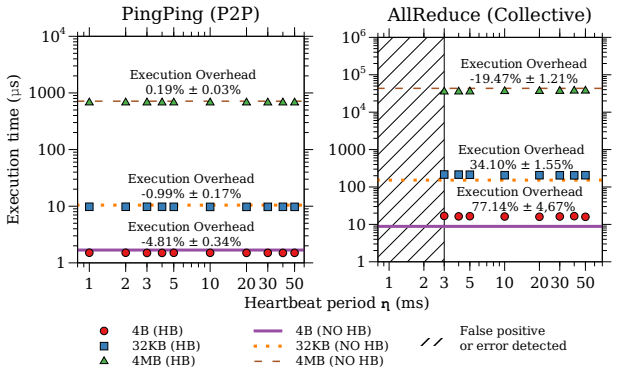
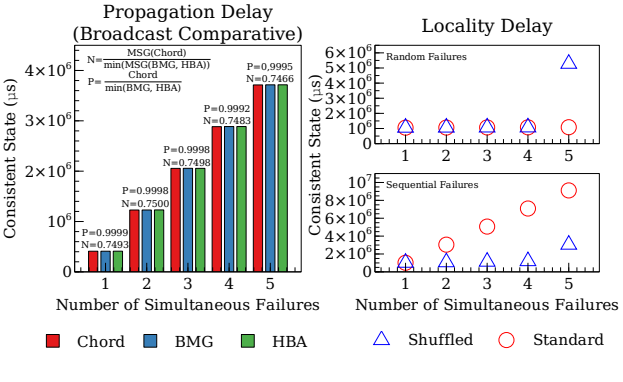


Figure 4. Detection and propagation delay evaluation



Concerning FP, HBA and BMG can achieve full propagation in $\lfloor log_2(N) \rfloor$ rounds of communication. However, the propagation is made both forward and backward in the ring. We propose using a Chord-style algorithm [6], in which we send the broadcast messages to the nodes with ranks following steps in powers of two, but only forward, stopping at $\lfloor log_2(N) \rfloor$ processes. Thus, full propagation can be achieved in $\lfloor log_2(N) \rfloor$ rounds (like BMG and HBA), while still taking into account FT, since each process will receive $\lfloor log_2(N) + 1 \rfloor$ broadcast messages (in this algorithm, each process, upon receiving the broadcast message for first time, replicates the message), which is approximatelly half the number of messages exchanged by HBA ($2 \times \lfloor log_2(N) \rfloor$) and BMG ($2 \times (\lfloor log_2(N) \rfloor + 1)$).

Finally, the last thing to point out is the portability. The algorithm proposed by Bosilca et al. relies on Open MPI's internal tools (byte transportation layer), making its use restricted to Open MPI [1]. Although the authors argue that it is a better option to keep the heartbeat mechanism low in the application stack [1], we propose to execute the heartbeat as an MPI program (in a thread of the application), using only MPI standard functions, not relying on specific tools of the operating systems or on a specific MPI distribution. To do that, we implemented the original algorithm with our modifications inside a C++ library, compatible with various MPI implementations, including Open MPI and MPICH.

All the events described by the Bosilca et al. algorithm [1], with the proposed additions, are iterated in an extra thread. Every θ seconds, the algorithm looks if it is time to execute any of the events, for example, if it is time to send an alive message, or if the process received an alive (or broadcast) message and takes the necessary actions.

4. Experimental Evaluation

To evaluate the proposed modifications, we tested them on a small cluster with 3 nodes with 16 cores each (running a 3.10.0 Linux-Server). For these tests, we used MPICH 3.3.2 configured with UCX. Experiments were executed 30 times, and we present the average results. For all the tests, we used $\theta = 1ms$, and like Bosilca et al., we define $\delta = 10 \times \eta$.

The first set of tests use IMB-MPI1 benchmarks[2]. Accuracy (values of δ and η, which reports no false-positive failures or errors) and overhead (comparing executions with and without the heartbeat) were tested, evaluating two communication patterns, point-to-point and collective communication for different message sizes. Left plot in Figure 3 shows the first case, using 2 processes in 2 nodes (low resource competition). Tests managed to achieve values of η from $1ms$ to $50ms$ without any error or false positive detection and very low execution overhead. The right plot shows results for collective patterns, using 32 processes in 2 nodes (high resource competition). The tests with η less than $3ms$ presented timeout errors in the benchmark, meaning that there is too much overhead. In comparison,

[2]Intel MPI benchmarks: https://github.com/intel/mpi-benchmarks

tests with period over or equal to $3ms$ executed without errors or false-positive detection. Concerning execution overhead, as the message size increases, the overhead reduces (the negative overhead could be explained by external (machine performance) or internal (MPI optimizations) factors, this will be investigated in the future). These results are as good as or better than the proposal made by Bosilca et al. ($2.5ms$ for point-to-point and $10ms$ for collectives), and at the same time makes the algorithm's portability possible by running it at the application level.

Custom benchmarks compose the second set of tests (using 48 processes in 3 nodes). Each propagation algorithm was tested with the number of failures ranging from 1 to 5 simultaneous failures. Then, the total elapsed time for every process to be notified about all failures, and the number of messages received by each process (with a tolerance of 2 seconds to account for late messages) were measured. The left plot of Figure 4 shows that the propagation delay (P) is similar for all algorithms. In particular, our approach reduces the number of messages by 25%, in comparison to the best of the other two (HBA), which decreases network congestion. Evaluating the locality problem (right plot in Figure 4), when sequential simultaneous failures occur, the shuffled option performs better since the probability of the failed nodes being sequential on the ring is reduced. For random failures, the behavior is similar to both approaches, since normally the the process are not sequential, but shuffling might distribute processes, with non-sequential ranks, sequentially on the ring.

5. Conclusions

FT is a common concern in HPC applications. Two essential components of FT are FD and FP. This paper proposes false-positive detection and initial shuffling for the ring topology to improve FD and proposes a different FP method. Results show our approach is at least as good as the original algorithms and, in some ways, better. This proposal is also portable to any MPI standard-compliant distribution while having low overhead in performance. Future works could explore tests with larger clusters as well as statistical evaluation for position shuffling.

References

[1] George Bosilca, Aurelien Bouteiller, Amina Guermouche, Thomas Herault, Yves Robert, Pierre Sens, and Jack Dongarra. A failure detector for hpc platforms. *The International Journal of High Performance Computing Applications*, 32(1):139–158, 2018.

[2] Sourav Chakraborty, Ignacio Laguna, Murali Emani, Kathryn Mohror, Dhabaleswar K Panda, Martin Schulz, and Hari Subramoni. Ereinit: Scalable and efficient fault-tolerance for bulk-synchronous mpi applications. *Concurrency and Computation: Practice and Experience*, 32(3):e4863, 2020.

[3] Abhinandan Das, Indranil Gupta, and Ashish Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks*, pages 303–312. IEEE, 2002.

[4] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[5] Giorgis Georgakoudis, Luanzheng Guo, and Ignacio Laguna. Reinit^{++}: Evaluating the performance of global-restart recovery methods for mpi fault tolerance. In *International Conference on High Performance Computing*, pages 536–554. Springer, 2020.

[6] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[7] Dong Zhong, Aurelien Bouteiller, Xi Luo, and George Bosilca. Runtime level failure detection and propagation in hpc systems. In *Proceedings of the 26th European MPI Users' Group Meeting*, pages 1–11, 2019.