

Scalable Distributed Consensus to Support MPI Fault Tolerance^{*}

Darius Buntinas
Argonne National Laboratory

Abstract. As system sizes increase, the amount of time in which an application can run without experiencing a failure decreases. Exascale applications will need to address fault tolerance. In order to support algorithm-based fault tolerance, communication libraries will need to provide fault-tolerance features to the application. One important fault-tolerance operation is distributed consensus. This is used, for example, to collectively decide on a set of failed processes. This paper describes a scalable, distributed consensus algorithm that is used to support new MPI fault-tolerance features proposed by the MPI 3 Forum’s fault-tolerance working group. The algorithm was implemented and evaluated on a 4,096-core Blue Gene/P. The implementation was able to perform a full-scale distributed consensus in 305 μ s and scaled logarithmically.

1 Introduction

As process counts in applications grow toward exascale, the length of time an application can run without experiencing a failure, known as the mean time between failures (MTBF), decreases. Applications will need to be fault tolerant in order to be useful on future exascale machines. Checkpointing can provide fault tolerance to an application without the need to modify it. As the failure frequency increases, however, checkpoints will need to be taken more often, decreasing the amount of useful work the application can perform between failures.

Whereas checkpointing provides fault tolerance to an application in a transparent manner, when using algorithm-based fault tolerance (ABFT) [1][3][4], the application is aware of faults and handles them explicitly. The fault-tolerance working group of the MPI 3 Forum has been working on a proposal [5], that adds fault-tolerance features to MPI in order to support ABFT applications. The proposal defines the behavior of an MPI library if processes fail. For example, existing operations such as `MPI_Comm_split` are now required to either succeed at every process or return an error at every process, even if processes fail before or during the operation. The proposal also introduces new functions, such as `MPI_Comm_validate_all`, that require all processes to return the same list of failed processes. A distributed consensus algorithm is needed to implement these operations.

This paper presents a scalable, fault tolerant, distributed consensus algorithm used to implement the `MPI_Comm_validate_all` function. The `MPI_Comm_validate_all` implementation is evaluated on a 4,096-core IBM Blue Gene/P machine and shows $O(\log n)$ scaling.

^{*} This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

2 Algorithm

We present the distributed consensus algorithm as it would be used in the `MPI_Comm_validate_all` operation. However, the algorithm could also be used in other operations requiring distributed consensus, such as `MPI_Comm_split`. In this section, we give a brief overview of the algorithm at a high level. A detailed description of the algorithm can be found in [2].

The `MPI_Comm_validate_all` function uses distributed consensus to decide on a set of failed processes, which must contain every failed process known by any participating process at the time the function is called. The same set of failed processes must be returned by the function at every process. If a process fails during the `MPI_Comm_validate_all` operation (i.e., after the first process calls the function and before the first process returns), the set of failed processes returned may or may not contain the failed processes.

The algorithm is similar to the three-phase commit algorithm except that, rather than sending and receiving individual messages, a reliable broadcast algorithm is used to send and collect messages. In the `BALLOTING` phase, after the root is chosen, the root creates a ballot containing the set of failed processes and broadcasts it to the rest of the processes. Once the processes receive the ballot, the responses to the ballot are collected back up the tree. If all the processes have accepted the ballot, the algorithm enters the `COMMIT` phase; otherwise a new ballot is generated, and the `BALLOTING` phase is repeated. In the `COMMIT` phase the root broadcasts a commit message. Once all processes receive the commit message, acknowledgments are collected back up to the root. The last phase is the `ALL_COMMIT` phase. In this phase the root broadcasts the all-commit message. Once a process receives the all-commit message, it can return from the `MPI_Comm_validate_all` function.

3 Performance Evaluation

To evaluate the validate-all operation, we implemented it as an MPI program. This allowed us to evaluate the operation at a large scale on a Blue Gene/P without modifying the MPI implementation. We expect the performance of the operation implemented this way to be an upper bound on the performance of the operation if it were integrated into an MPI implementation. The evaluation was performed at Argonne National Laboratory on Surveyor, a Blue Gene/P with 1,024 quad-core nodes.

Figure 1 shows the results of the evaluation. As expected, the operation scales logarithmically. For comparison, we evaluated the time taken to perform a communication pattern similar to that of the validate-all operation using broadcast and reduction operations. The figure shows the results with optimized collectives using the Blue Gene/P collective tree network and with unoptimized collectives using the same torus network that the validate-all operation uses. At full scale, the validate-all implementation took 305 μ s to perform the operation, which is 1.66 times slower than performing a similar communication pattern with unoptimized collectives. We expect the performance of the validate-all algorithm to

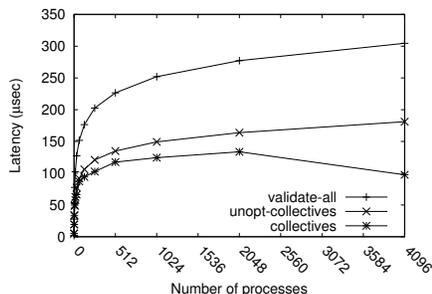


Fig. 1. Comparison of the validate-all operation with collectives operations.

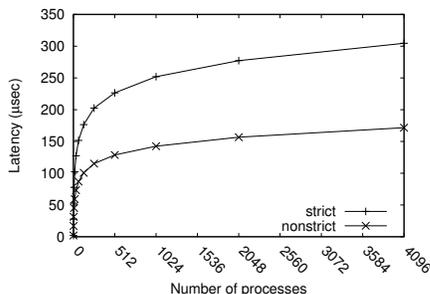


Fig. 2. Comparison of validate-all using strict and loose semantics.

improve when the operation is integrated into the MPI implementation, making the algorithm more responsive to incoming messages.

We also evaluated the performance of the operation with loose semantics, as described in the proposal [5]. Figure 2 shows the comparison. The loose implementation performs the operation 133 μ s faster at full scale than does the strict implementation (which is 1.78 times as fast). Depending on the requirements of the application and the frequency at which the application calls validate-all, using the loose implementation can provide some performance improvement.

4 Conclusion

This paper presented a scalable distributed consensus algorithm used to implement the MPI_Comm_validate_all operation proposed by the MPI 3 fault-tolerance working group. The algorithm was evaluated on a 4,096-core Blue Gene/P machine and was shown to be extremely scalable. The implementation was able to perform a full-scale validate-all operation in 305 μ s and scaled logarithmically.

Using the loose implementation saved only 133 μ s over the strict implementation. Therefore, unless the application performs many validate-all operations, relaxing the semantics is unlikely to improve the overall performance of the application significantly.

References

1. Anfinson, J., Luk, F.T.: A linear algebraic model of algorithm-based fault tolerance. IEEE Transactions on Computing 37, 1599–1604 (1988)
2. Buntinas, D.: Scalable distributed consensus to support MPI fault tolerance. Tech. Rep. ANL/MCS-TM-314, Argonne National Laboratory (Jun 2011)
3. Chen, Z., Dongarra, J.: Algorithm-based fault tolerance for fail-stop failures. IEEE Transactions on Parallel and Distributed Systems 19(12) (Dec 2008)
4. Chen, Z., Dongarra, J.: Highly scalable self-healing algorithms for high performance scientific computing. IEEE Transactions on Computers (Jul 2009)
5. Fault Tolerance Working Group: Run-through stabilization proposal, http://svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_stabilization