A Log-Scaling Fault Tolerant Agreement Algorithm for a Fault Tolerant MPI

Joshua Hursey[1], Thomas Naughton[1],
Geoffroy Vallee[1], and Richard L. Graham[1]

Oak Ridge National Laboratory, Oak Ridge, TN USA 37831
{hurseyjj,naughtont,valleegr,rlgraham}@ornl.gov

Abstract. The lack of fault tolerance is becoming a limiting factor for application scalability in HPC systems. The MPI does not provide standardized fault tolerance interfaces and semantics. The MPI Forum's Fault Tolerance Working Group is proposing a collective fault tolerant agreement algorithm for the next MPI standard. Such algorithms play a central role in many fault tolerant applications. This paper combines a log-scaling two-phase commit agreement algorithm with a reduction operation to provide the necessary functionality for the new collective without any additional messages. Error handling mechanisms are described that preserve the fault tolerance properties while maintaining overall scalability.

Keywords: MPI, Fault Tolerance, Agreement Protocol, Run-through Stabilization, Algorithm Based Fault Tolerance

1 Introduction

The lack of fault tolerance will soon become a limiting factor for application scalability in High Performance Computing (HPC) systems, in particular exascale systems. It is projected that the mean time to failure (MTTF), a measure of system reliability, will drop from days to hours or minutes in such HPC systems [2]. This indicates that process failure will be a normal event that the application must be prepared to handle to fully utilize next generation HPC systems. As a result, applications are looking to augment (or replace) their existing checkpoint/restart fault tolerance techniques with Algorithm Based Fault Tolerance (ABFT) techniques to improve the efficiency of application recovery.

Unfortunately, application developers are hindered by the lack of any, let alone scalable, resilience models necessary for ABFT in fundamental support libraries like the Message Passing Interface (MPI) [13]. The current MPI standard does not provide standard semantics in the presence of process failure except in the default, abort case (i.e., MPI_ERRORS_ARE_FATAL). Such semantics are left to be optionally defined by individual implementations.

The MPI Forum created the Fault Tolerance Working Group (FTWG) in response to the growing need for portable, scalable fault tolerant semantics and

interfaces in the MPI standard. Fault tolerant agreement algorithms serve as a fundamental building block for most fault tolerant applications and libraries [1]. These algorithms provide uniform agreement of a value (or set of values) even in the presence of process failure during the execution of the algorithm. The FTWG's run-through stabilization (RTS) proposal provides an interface to such an algorithm in the MPI_Comm_validate_all collective operation over communicators. A similar collective interface is also available for windows and file handles.

The MPI_Comm_validate_all collective operation must be able to be implemented in a scalable manner if it is to be relied upon in highly scalable, fault tolerant HPC applications. Most fault tolerant agreement algorithms struggle to scale well to large numbers of processes, while others propose overly complex algorithms that are difficult to implement in practice. These algorithms focus on the agreement of a single state (namely `COMMIT` or `ABORT`) after the execution of the transaction body. The MPI_Comm_validate_all collective operation must agree upon a set of failed processes constructed by the group. Using the existing agreement protocols would require a separate fault-aware reduction operation followed by a separate agreement protocol.

The presented algorithms combine the reduction operation with a two-phase agreement operation to construct the list of known failures during the *voting* phase and uniformly agree upon a single list during the *commit* phase. This paper describes algorithmic adjustments made to the *two-phase commit* agreement algorithm, for both linear-scaling and log-scaling variations, to provide this functionality. The log-scaling algorithm variation sustains a point-to-point message complexity of $O(2log(n))$. This paper describes the error handling mechanisms that preserve the fault tolerance guarantee of uniform agreement even in the presence of process failure, in addition to an optimization to the termination protocol.

2 Related Work

Applications are experimenting with the integration of fault tolerance techniques into their code to improve the efficiency of application recovery. ABFT techniques require specialized algorithms that are able to adapt to and recover from process loss [10]. ABFT techniques typically rely upon data encoding, algorithm redesign, and diskless checkpointing in addition to a fault tolerant message passing environment (e.g., MPI). Related to ABFT are natural fault tolerance techniques [5]. Natural fault tolerance techniques focus on algorithms that can withstand the loss of a process and still get an approximately correct answer, usually without the use of data encoding or checkpointing.

The FTWG's RTS proposal defines semantics and interfaces for the handling of *fail-stop* process failure [7]. *Fail-stop* (a.k.a. *crash fault*) process failures are failures in which a process is permanently stopped often due to a component crash event in the system [1]. For the detection of such failures, the proposal provides the application with a *perfect* failure detector. A *perfect* failure detector is both *strongly accurate* and *strongly complete* [4]. *Strong accuracy* means that

no process is reported as failed before it actually fails. *Strong completeness* means that eventually every failed process will be known by all other processes.

Fault tolerant agreement algorithms play a central role in many fault tolerant applications, libraries, and distributed transaction processing services for database systems [1]. These collective algorithms provide uniform agreement of a state even in the presence of process failure during the execution of the algorithm. There are three often cited fault tolerant agreement algorithms: *two-phase commit*, *three-phase commit*, and *Paxos*. Agreement algorithms can be either *blocking* or *non-blocking*. A *blocking* algorithm may block, in some failure scenarios, in an undecided state until a peer process is restored and makes a decision from a write-ahead log file. A *non-blocking* algorithm does not require the restart of failed processes for the collective group to decide.

The *two-phase commit* algorithm is a *blocking* algorithm built from two linear reliable broadcast operations and one linear reliable gather operation [9]. An optional, termination detection algorithm can be used to reduce the opportunity for blocking when the coordinator fails. The linear nature of the reliable broadcast and gather operations allow for relative simplicity in the handling of process failures, but at the cost of poor scalability to large numbers of processes.

Multi-level, tree structured two-phase agreement algorithms have been explored in, and proven correct for transaction processing systems [14, 15]. The algorithm presented in this paper combines the multi-level communication topology with the construction of the global list of failed processes. Combining operations reduces the message complexity required to do both consecutively.

The *three-phase commit* algorithm extends the two-phase commit algorithm by adding another round of messages to eliminate the need for blocking [17]. Since this algorithm adds another round of operations (one additional broadcast and gather) it further adds to the message complexity.

The *Paxos* algorithm is a non-blocking algorithm that uses *replicas* instead of a single coordinator to reach agreement [12]. This algorithm scales as well as the two-phase commit algorithm while still being a non-blocking algorithm. However, this algorithm has proven challenging to implement correctly in practice [3].

The FT-MPI project provided a first attempt at extending the semantics and interface of the MPI-1 standard to support ABFT [6]. FT-MPI extended the MPI communicator states and modified the MPI communicator construction functions. The FT-MPI project provided inspiration for the FTWG's RTS and Process Recovery proposals. The RTS proposal provides semantics similar to FT-MPI's `blank` communicator mode, where failed processes are replaced by MPI_PROC_NULL. Both projects have complementary semantics regarding point-to-point and collective operations. The main difference between these projects is in the handling of communicator and group objects. Upon process failure, FT-MPI destroys all MPI objects with non-local information (e.g., communicators and groups), except MPI_COMM_WORLD, requiring the application to manually recreate these objects after every failure. In contrast, the RTS proposal preserves all communicators and groups. Instead of providing a fault tolerant agreement protocol to the application (i.e., MPI_Comm_validate_all), the FT-MPI project

provides it as a transparent component of the runtime environment which is used to determine the group membership for MPI_COMM_WORLD after each process failure. As such, FT-MPI requires that every process failure be recognized globally by all alive processes. In the RTS proposal, process failures can be recognized locally, and on a per-communicator basis. These differences allow the RTS proposal to more flexibly support libraries, and, by allowing for localized failure recognition, open the door to more scalable fault tolerance solutions.

3 Two-Phase Commit Algorithm

This section briefly describes the structure and fault management properties of the linear scaling two-phase commit algorithm [9]. The two-phase commit algorithm is discussed in terms of how it was implemented in Open MPI to support the MPI_Comm_validate_all collective operation.

The two-phase commit algorithm relies upon linear-scaling reliable broadcast and gather operations, in addition to an optional linear termination detection algorithm upon coordinator failure. The *coordinator* initiates the algorithm by broadcasting a *vote request* to all of the *participants*, skipping failed processes. Since the MPI_Comm_validate_all is a collective operation, all alive processes will eventually enter the operation, so the vote request round can be eliminated from the MPI implementation reducing the message complexity, called the *unsolicited vote* optimization [18]. The algorithm will decide either `DECIDED` or `UNDECIDED` (a.k.a, `COMMIT`, `ABORT`) along with a globally constructed list of process failures.

Upon entering the MPI_Comm_validate_all operation, the participants send their *vote* to the coordinator. In the Open MPI implementation, the vote is a bit-field of locally known failed processes at that rank. If the coordinator fails before a participant sends its vote, then the participant can safely decide `UNDECIDED` and exit the algorithm since the coordinator could not have made a decision without their contribution.

The coordinator gathers the contributions of each participant (skipping failed processes), and creates a *decision* – represented as list of globally known failed processes constructed from the local list at each process. The coordinator then broadcasts the decision to all alive participants.

If the coordinator fails after a participant sent its vote, but before the participant receives the decision message then the participant is in an *uncertain* state since it does not know if a decision was made before the coordinator failed. Without the optional termination detection algorithm, the uncertain participant would *block* and wait for the coordinator to be recovered. With the termination detection algorithm, the uncertain participant linearly asks all other participants if they have decided or not. If a peer participant has decided (either `DECIDED` or `UNDECIDED`), then the uncertain participant decides with them. Otherwise, if no other alive participant has decided, then this process blocks waiting for recovery.

The algorithm must keep at least two log entries in the volatile memory of each local process. A log entry contains the list being decided upon, the state of the agreement, and a monotonically increasing sequence number to distinguish

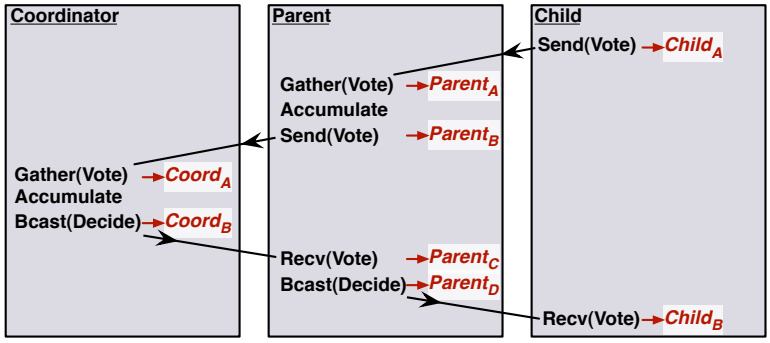


Fig. 1. Illustration of the three participants in the log-scaling two-phase commit algorithm. Error handling annotations highlighted in red italics (e.g., $\{Parent_C, Child_B\}$).

rounds. One entry stores the last decision made by the group used to catchup a process in the termination protocol. The other entry maintains that state for the current round of the operation.

The RTS proposal does not provide the capability to restart processes, so if a process blocks it must call MPI_Abort. Progressing forward from an uncertain state may cause consistency issues with other communication contexts. In the Open MPI implementation, a runtime option is provided to allow the user to promote the uncertain state to an `UNDECIDED` decision, for the case where this semantic protection is not desired.

Ideally the MPI implementation would provide a non-blocking, fault tolerant agreement algorithm (e.g., three-phase commit), to avoid the uncertainty problem described above. This paper focuses on augmenting the two-phase algorithm for the sake of simplicity in explanation, and leaves the non-blocking algorithm extension to future work.

4 Log Scaling Two-Phase Commit Algorithm

The log-scaling two-phase commit algorithm uses a tree structure for both the broadcast and gather operations similar in communication structure to multi-level variations [14, 15]. In practice, the *gather* is replaced by a *reduce*, but referred to in this section as a *gather* for consistency with Section 3.

The log-scaling two-phase commit algorithm differentiates between two types of participants: those that are not leaf elements, called *parent* participants; and those that are leaf elements, called *child* participants. The *coordinator* (at the root of the tree) is still ultimately responsible for making the final decision - construction of the final list of failed processes. The dependencies in the tree structure require additional error handling mechanisms to maintain the consistency of the algorithm and data in the presence of fail-stop process failures.

Figure 1 illustrates the three kinds of participants in the algorithm along with the basic communication pattern and error handling annotations (highlighted in

red italics). This algorithm can be used with a tree of any depth, even though this figure shows only one level of parent participants.

In Figure 1, the primary error handlers of the coordinator and parents for the reliable gather operation are $\{Coord_A, Parent_A\}$ and for the broadcast operation are $\{Coord_B, Parent_D\}$. During the reliable gather and broadcast operations, parent participants of a failed child must recursively *adopt* the grandchildren. This ensures that all alive processes in the collective group have the opportunity to *vote* and *decide* uniformly with the remaining portion of the group.

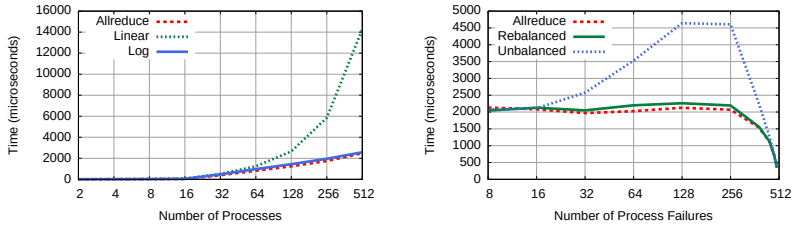
Upon detection of parent participant failure, the dependent children are either in the $\{Parent_B, Child_A\}$ or $\{Parent_C, Child_B\}$ error handlers. The dependent children search for the nearest grandparent participant (which could be the coordinator). If an alive grandparent is found, the dependent child posts a query message to the new parent asking how it should continue participating in the algorithm. If the dependent child has not yet voted (in $\{Parent_B, Child_A\}$) then the new parent tells it to participate in the gather phase. If the child has voted (in $\{Parent_C, Child_B\}$), the new parent tells it to either participate in the gather phase (if the old parent failed before propagating the message up the tree), or the decision phase (if the old parent failed after propagating the message up the tree). If the new parent had successfully completed the collective operation, then it replies with the decision value and the child decides with the new parent. If the new parent also fails, the child queries the next alive parent.

If the coordinator fails, the termination detection algorithm is activated in each of the uncertain dependent parent participants of the coordinator, which may involve a parent at a lower level in the tree for cases where a first level parent failed. If a dependent parent participant has not voted (in $\{Parent_B, Child_A\}$) then it can decide `UNDECIDED`. If a dependent parent participant has voted (in $\{Parent_C, Child_B\}$) it becomes *uncertain* and enters the termination protocol. When a parent participant makes a decision, it must propagate that decision down the tree using the reliable broadcast operation.

As an optimization to the termination algorithm, instead of asking *all* participants in the collective group for a decision (as with the original two-phase algorithm), the uncertain participant only asks the parent participants dependent upon the coordinator. Notice that if the coordinator made a decision then the directly dependent parent participants of the coordinator would be the first to know, since they are in the broadcast group of the coordinator. Further, since children in the sub-tree below the parent only decide with their direct ancestors in the tree, those dependent children do not need to be queried since they cannot make a different decision than that of their parent.

The tree structure remains static between successful completions of the algorithm. Upon successful completion, the agreed upon list of failed processes is used to rebalance the tree structure. Rebalancing has been shown to improve the performance of collective operations after fail-stop process failure [11].

Even with the tree structured algorithm, there is still a possibility for blocking if the coordinator fails after the gather and before the broadcast operation. As



(a) Failure free scaling performance.

(b) Performance of the log-scaling algorithm at 512 processes varying the number of failures.

Fig. 2. Performance of fault tolerant log-scaling two-phase commit algorithm.

mentioned in Section 3, in this one scenario the uncertain processes are aborted, by default, to protect the user.

5 Results

The following analysis used a prototype of the FTWG's RTS proposal based on the development trunk of the Open MPI implementation of the MPI standard [8]. Fault aware collectives are provided in the ftbasic component of the coll framework [11]. Three variations of the MPI_Comm_validate_all collective operation were implemented: a linear two-phase, a log-scaling two-phase, and a non-fault tolerant log-scaling allreduce. The allreduce algorithm is similar in form to the two-phase commit algorithm so it serves as an appropriate baseline for performance comparison. Though any tree topology can be used with the presented algorithm, a binomial tree is used in this implementation.

These tests used 32 nodes with each node containing four quad-core 2.0 GHz AMD Opteron processors. The Ethernet (tcp) and shared memory (sm) network drivers in Open MPI were used for these tests since they are the only fully supported interconnects provided by the prototype at this time. In the failure-full tests specific ranks are forcibly terminated before the performance testing by sending them the `SIGKILL` signal. This paper assumes fail-stop process failures, and once a process fails it is never restored. After a warmup phase, each data point is the average of 20 sets of an inner timing loop of 200 operations.

Figure 2(a) shows the failure-free performance of the three implementations. This figure illustrates the expected log-scaling performance of the algorithm presented in this paper. At 512 processes, the log-scaling algorithm shows a significant improvement over the linear-scaling algorithm, while staying within 3% of the baseline performance.

Figure 2(b) explores the performance impact on the log-scaling two-phase commit algorithm as the number of failures increases for a fixed sized job of 512 processes. The baseline performance in this figure is a failure-free run of the allreduce algorithm on a reduced sized communicator. As the number of failures

increases the need to rebalance the validation tree upon successful agreement becomes readily apparent (2.6 times slower at 256 failures). The rebalanced performance is at worst 6% slower (at 64 failures) than the baseline allreduce algorithm. This difference indicates that there may be further room for improvement in the implementation.

6 Conclusion

Fault tolerant agreement algorithms play a foundational role in many fault tolerant applications, but existing algorithms often struggle with scalability to large numbers of processes. The MPI Forum's FTWG proposed, as part of the RTS proposal, a MPI_Comm_validate_all collective operation to encapsulate such algorithms. This operation provides uniform agreement of the set of known failed processes in the specified communicator (variations are also available for windows and file handles) even in the presence of process failures during the algorithm.

This paper explores an enhancement to the well established two-phase commit algorithm. By replacing the linear broadcast and gather operations with tree-based, log-scaling operations the point-to-point message complexity was reduced from $O(2N)$ to $O(2log(N))$. Further by combining the list construction operation with the two-phase commit protocol, no additional messages were added to fully support the MPI_Comm_validate_all collective operation. This paper describes the additional error handling mechanisms required to maintain the fault tolerance guarantees of uniform agreement even in the presence of failures. Using a prototype implementation in Open MPI, performance results showed scaling performance comparable to an MPI_Allreduce operation.

Since the two-phase commit algorithm is blocking in some situations, we are extending this design to non-blocking algorithms. Concurrently, we are exploring other methods for implementing this collective algorithm to further improve scalability and performance. Finally, we will consider dynamically constructed trees, instead of statically rebalanced trees, which may help manage the effects of process skew [16].

Acknowledgments

Research sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

References

1. Barborak, M., Dahbura, A., Malek, M.: The consensus problem in fault-tolerant computing. ACM Computing Surveys 25, 171–220 (June 1993)

2. Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., Snir, M.: Toward exascale resilience. International Journal of High Performance Computing Applications 23(4), 374–388 (2009)
3. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: An engineering perspective. In: Proceedings of the twenty-sixth annual ACM symposium on Principles of Distributed Computing. pp. 398–407. PODC '07, ACM, New York, NY, USA (2007)
4. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43, 225–267 (March 1996)
5. Engelmann, C., Geist, A.: Super-scalable algorithms for computing on 100,000 processors. In: Proceedings of International Conference on Computational Science (ICCS). vol. 3514, pp. 313–320 (May 2005)
6. Fagg, G.E., Gabriel, E., Chen, Z., Angskun, T., Bosilca, G., Pjesivac-Grbovic, J., Dongarra, J.J.: Process fault-tolerance: Semantics, design and applications for high performance computing. International Journal for High Performance Applications and Supercomputing 19(4), 465–478 (2005)
7. Fault Tolerance Working Group: Run-though stabilization proposal, `svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_stabilization`
8. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings of the 11th European PVM/MPI Users' Group Meeting. pp. 97–104. Budapest, Hungary (September 2004)
9. Gray, J.: Notes on data base operating systems. In: Operating Systems, An Advanced Course. pp. 393–481. Springer-Verlag, London, UK (1978)
10. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers 33(6), 518–528 (1984)
11. Hursey, J., Graham, R.: Preserving collective performance across process failure for a fault tolerant MPI. In: 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS) held in conjunction with the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS). Anchorage, Alaska (May 2011)
12. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems (TOCS) 16, 133–169 (May 1998)
13. Message Passing Interface Forum: MPI: A Message Passing Interface. In: Proceedings of Supercomputing '93. pp. 878–883. IEEE Computer Society Press (1993)
14. Mohan, C., Lindsay, B.: Efficient commit protocols for the tree of processes model of distributed transactions. ACM SIGOPS Operating Systems Review 19, 40–52 (April 1985)
15. Mohan, C., Lindsay, B., Obermarck, R.: Transaction management in the R* distributed database management system. ACM Transactions on Database Systems (TODS) 11, 378–396 (December 1986)
16. Raz, Y.: The dynamic two phase commitment (d2pc) protocol. In: Proceedings of the 5th International Conference on Database Theory. pp. 162–176. ICDT '95, Springer-Verlag, London, UK (1995)
17. Skeen, D.: Nonblocking commit protocols. In: Proceedings of the 1981 ACM SIGMOD international conference on Management of Data. pp. 133–142. SIGMOD '81, ACM, New York, NY, USA (1981)
18. Stonebraker, M.: Concurrency control and consistency of multiple copies of data in distributed Ingres. IEEE Transactions on Software Engineering SE-5(3), 188 – 194 (May 1979)