Preserving Collective Performance Across Process Failure for a Fault Tolerant MPI

Joshua Hursey, Richard L. Graham
Oak Ridge National Laboratory Oak Ridge, TN USA 37831
Email: {hurseyjj,rlgraham}@ornl.gov

***Abstract*—Application developers are investigating Algorithm Based Fault Tolerance (ABFT) techniques to improve the efficiency of application recovery beyond what traditional techniques alone can provide. Applications will depend on libraries to sustain failure-free performance across process failure to continue to efficiently use High Performance Computing (HPC) systems even in the presence of process failure. Optimized Message Passing Interface (MPI) collective operations are a critical component of many scalable HPC applications. However, most of the collective algorithms are not able to handle process failure. Next generation MPI implementations must provide fault aware versions of such algorithms that can sustain performance across process failure. This paper discusses the design and implementation of fault aware collective algorithms for tree structured communication patterns. The three design approaches of rerouting, lookup avoiding and rebalancing are described, and analyzed for their performance impact relative to a similar fault unaware collective algorithm. The analysis shows that the rerouting approach causes up to a four times performance degradation while the rebalancing approach can bring the performance within 1% of the fault unaware performance. Additionally, this paper introduces the reader to a set of run-through stabilization semantics being developed by the MPI Forum's Fault Tolerance Working Group to support ABFT. This paper underscores the need for care to be taken when designing new fault aware collective algorithms for fault tolerant MPI implementations.**

***Keywords*-MPI; Fault Tolerance; Collective Communication; Algorithm Based Fault Tolerance; Run-through Stabilization**

I. Introduction

As scientists run their High Performance Computing (HPC) applications longer and scale them further to address complex scientific questions, they often exceed the reliability of a given HPC system. Today, administrators of large HPC systems often measure system reliability, in terms of mean time to failure (MTTF), in days or weeks [1]. It is anticipated that future exascale HPC systems will reduce the MTTF to minutes or hours, further exposing the application to the risk of failure during normal computation. Process failures, in particular, will no longer be rare events, but normal events that the application must be prepared to handle [2].

In preparation for these next generation HPC systems, applications are looking to augment (or replace) their existing checkpoint/restart fault tolerance techniques with more application focused, Algorithm Based Fault Tolerance (ABFT) techniques to improve the efficiency of application recovery after process failure. Unfortunately, application developers are hindered by the lack of resilience necessary for ABFT in fundamental supporting libraries like the Message Passing Interface (MPI) [3]. The current MPI standard does not address how an implementation should behave after a failure, except in the default, abort case (i.e., MPI_ERRORS_ARE_FATAL). The MPI Forum created the Fault Tolerance Working Group in response to this growing need for portable, fault tolerant semantics and interfaces in the MPI standard.

While implementing a fault tolerant MPI implementation care must be taken to ensure that the failure-free performance is preserved, to the greatest possible extent, after the recovery from the process failure. MPI collective algorithms play a central role in many scalable HPC applications. Maintaining the performance of optimized MPI collective operations through process failure will continue to enable such scalable HPC applications to efficiently use available computational resources. By making small adjustments to existing collective algorithms it is easy for a implementer to degrade the post-failure performance of or hang processes participating in a collective algorithm previously optimized for failure-free performance. For example, one common approach is to recursively route around failed processes during the collective operation. As we will show in Section IV, this can lead to significant slowdown even at relatively small scales, up to four times in one case.

This paper will discuss the design of fault aware collective algorithms for a binomial tree structured communication optimization pattern. Though the designs variations are applied to this specific communication topology they are relevant to other optimized collective communication patterns. We will analyze the impact of these three design decisions on MPI collective algorithm performance. Additionally, this paper will introduce the reader to a set of run-through stabilization semantics being developed by the MPI Forum's Fault Tolerance Working Group. The developing prototype implementation in Open MPI [4] will be used for the experimentation in Section IV.

II. Fault Tolerant MPI Semantics and Interfaces

The MPI Forum's Fault Tolerance Working Group is charged with defining a set of semantics and interfaces to enable fault tolerant applications and libraries to be portably constructed on top of the MPI interface. This paper analyses the design of fault aware collectives that conform to the run-through stabilization component of the developing proposal which is being extended to include flexible recovery strategies [5]. Run-through stabilization is sufficient for many applications and is a necessary step for applications that

```
1  MPI_Rank_info {
2    rank, /* Rank in the communicator */
3    generation, /* Generation of this rank */
4    state {
5      MPI_RANK_OK, /* Normal running state */
6      MPI_RANK_FAILED, /* Failed, not recognized */
7      MPI_RANK_NULL /* Recognized as failed */
8    }
9  }
10 /* Local operations */
11 MPI_Comm_validate_rank(comm, rank, rank_info);
12 MPI_Comm_validate(comm, incount, outcount,
13                                rank_infos[]);
14 MPI_Comm_validate_clear(comm, count,
15                                      rank_infos[]);
16 /* Collective operation */
17 MPI_Comm_validate_all(comm, outcount);
18 MPI_Icomm_validate_all(comm, outcount, request);
```

Fig. 1: MPI Communicator Management Extensions

may require process recovery. The run-through stabilization component of the proposal provides an application with the ability to continue running and using MPI even when one or more processes in the MPI universe fail. The proposal assumes fail-stop process failure meaning that a process is permanently stopped, often due to a crash [6]. For a discussion on how transient failures should be handled by the MPI implementation see the proposal [5]. Other types of faults not currently addressed by the MPI standard (i.e, reliable message delivery), like Byzantine failures [7], are left to the application to address, as necessary.

The proposal currently assumes that the MPI implementation provides the application with a view of the failure detector that is both *strongly accurate* and *strongly complete*, thus a *perfect* failure detector [8]. This means that eventually every failed process will be known to all processes in the MPI universe (strong completeness), and that no process is reported as failed before it actually fails (strong accuracy). The application is notified of a process failure once it attempts to communicate directly (e.g., point-to-point operations) or indirectly (e.g., collective operations) with the failed process through the return code of the function, and error handler set on the associated communicator. This proposal does not change the default error handler of MPI_ERRORS_ARE_FATAL, so the application must explicitly change the error handler to, at least, MPI_ERRORS_RETURN on all communicators involved in fault handling in the application to use these semantics.

The subset of the new interfaces that relate to our discussion are presented in Figure 1. The proposal and corresponding prototype implementation in Open MPI used in this paper currently support all of MPI-1 functionality including collective and group management operations. We are currently extending both the proposal and prototype to support the remainder of the MPI standard including parallel I/O and one-sided operations.

The proposal is based in the principle that the application should explicitly *recognize* process failures that affect them in each communicator they intend to continue using. Unrecognized process failures continue to generate errors when the failed process is referenced, while recognized failures have MPI_PROC_NULL semantics and do not generate errors when referenced.

A process can locally query for the state of an individual process using the MPI_Comm_validate_rank function, or access an array of all failed ranks using the MPI_Comm_validate function. A process can recognize a set of process failures locally on a specific communicator using the MPI_Comm_validate_clear function. Local recognition of the process failure allows for the continued use of point-to-point operations with the specified processes, but not collective operations. Additionally, a process can collectively recognize all failures in a communicator by using the MPI_Comm_validate_all function. The collective validate function returns the total number of failures in that communicator as agreed upon by all of the alive processes in the communicator, and re-enables collective operations on that communicator. This operation will return either success everywhere or some error at each alive rank. This means that the MPI_Comm_validate_all function provides the application with an implementation of a fault tolerant consensus algorithm [9]. Failures are recognized on a per-communicator basis to guarantee that libraries are able to receive notification of the failure, even if the main application has previously recognized the failure on a duplicate communicator. In the prototype, we maintain a bitfield with each communicator handle to account for the recognition of process failures on that communicator. Though this bitfield does contribute slightly to the memory footprint of a communicator handle, other implementation techniques could be used to help mitigate the memory impact of tracking recognized failures.

The MPI_Rank_info object is used by the validate functions to express the rank, generation, and state of a specific process. The `rank` field indicates the rank in the associated communicator. The `generation` field is a monotonically increasing number that is used to distinguish between multiple recovered versions of a process. The `state` field indicates which one of the three states that the rank is in. The MPI_RANK_OK state indicates that the rank is running normally. The MPI_RANK_FAILED state indicates that the rank has failed, and not yet been recognized by this process on this communicator. The MPI_RANK_NULL state indicates that the rank has failed, and has been recognized by this process on this communicator.

Once any process fails in a communicator, all collective operations on that communicator will return an error in the class MPI_ERR_RANK_FAIL_STOP until the communicator is repaired using the collective MPI_Comm_validate_all function. This requirement allows the MPI implementation an opportunity to re-optimize collective operations for improved performance after the failure, the mechanism for doing so is the focus of this paper. Once the communicator

has been collectively validated, then recognized failed ranks participate as if they were MPI_PROC_NULL (see [5] for more details). In order to preserve failure-free performance of collective operations, the working group decided to not require consistent return codes from collective operations (with the exception of MPI_Comm_validate_all). This means that collective algorithms must be fault aware, but not necessarily fault tolerant. For example, if the MPI implementation uses a tree implementation for MPI_Bcast then it is possible for a process to successfully leave the collective early once it has received the message and propagated the message to its children. However, a failure may occur while traversing the remainder of the tree that would cause some processes to return error. The MPI_Comm_validate_all function is useful in creating recovery blocks for sets of MPI operations [10].

III. Fault Aware Collectives

Collective operations allow an application developer using MPI to leverage years of scalable algorithmic optimization research without needing to know exactly how the collective is implemented. Little attention has been given to preserving these optimizations across process failure. After a process failure, communication structures must be adapted to operate not just on dense communicators without process failures, but also on sparse communicators containing recognized failed processes. Collective algorithms can be classified into three major categories: *fault unaware*, *fault aware*, and *fault tolerant*. Existing collective algorithms are often fault unaware, whereas the run-through stabilization proposal requires at least fault aware collectives algorithms, and, optionally, fault tolerant algorithms.

Fault unaware collective algorithms operate only over dense communicators without failed processes. If a process failure is recognized or emerges during the collective then the collective may hang on some ranks while returning from others. Given the tendency of an MPI implementation to abort the MPI job after a process failure, the hang is often remedied by the preemption of the entire job. Take for example a tree based MPI_Barrier implementation in which a leaf child fails just before entering the collective operation. If the root is not directly connected to the child in the tree then a fault unaware collective algorithm may hang at the root. The hang can be avoided if the intermediary ranks forward the error information throughout the tree unblocking the remainder of the ranks. If the collective algorithm contains such logic to propagate failure information, we no longer classify it as fault unaware, but as fault aware.

Fault aware collective algorithms recognize that failures can occur during the collective operation and that the communicator may contain failed processes. At minimum, a fault aware collective should not hang when a failed process is encountered. The run-through stabilization proposal requires that, to the greatest possible extent, the fault aware collectives should work around recognized failures in the communicator to complete the collective successfully. If a new failure is encountered during the collective operation, fault aware collectives are allowed to return success in some ranks and some error in other ranks depending upon when the error was detected in the course of the algorithm. As an example, Section II discussed how such behavior may emerge in a tree implementation of MPI_Bcast.

Fault tolerant collective algorithms are fault aware collectives that guarantee that the collective operation completes successfully everywhere or returns some error at every participating rank. A fault tolerant collective can be built from a fault aware collective followed by a collective MPI_Comm_validate_all since it requires a fault tolerant consensus protocol at the bottom of the operation. The strict consistency provided by fault tolerant collective algorithms provides consistency guarantees at the cost of performance for each collective operation. By decoupling the fault aware collective from the consensus protocol, an MPI application developer can group many collective operations into a recovery block, and preserve the performance of each individual collective operation. MPI implementors are provided the opportunity to add fault tolerant semantics to the proposed fault aware collective semantics as an implementation specific option without breaking the existing proposal.

A. Design Options

This paper analyzes three implementation options for fault aware collective algorithms that conform to the existing proposal. In our discussion, we use a binomial tree structure for communication, though the concepts relate to any tree-based communication structure and likely others. The three design options are *rerouting*, *lookup avoiding*, and *rebalancing*. The goal of the design is to regain as much performance as possible after process failure, ideally matching the performance of the fault unaware collective algorithm over the reduced number of processes.

In the *rerouting* design, the collective checks for a failed process before interacting with it, and recursively routes around failed processes. The check for recognized failure is needed to distinguish between a recognized failed process and an alive process in point-to-point operations. Since recognized failures have semantics equal to MPI_PROC_NULL, sending to an alive process and a recognized failed process will both return success, though each case must be handled differently. If a recognized failure is detected, then the parent will recursively adopt the children of the failed child, and the children will search for the nearest alive grandparent. Root failure is handled by choosing the lowest numbered rank from the next level of the tree. So in the rerouting design, recognized failures are routed around while maintaining the original communication structure. This approach is often considered sufficient when first approaching fault aware collectives since it is the often the most direct way to create a functional fault aware collective.

Rerouting in a binomial tree containing recognized failures can hurt performance if the tree becomes imbalanced. Figure 2 shows a binomial tree with zero, one, four, and eight failures. This figure highlights that, depending on which ranks fail, the tree could become significantly imbalanced causing some

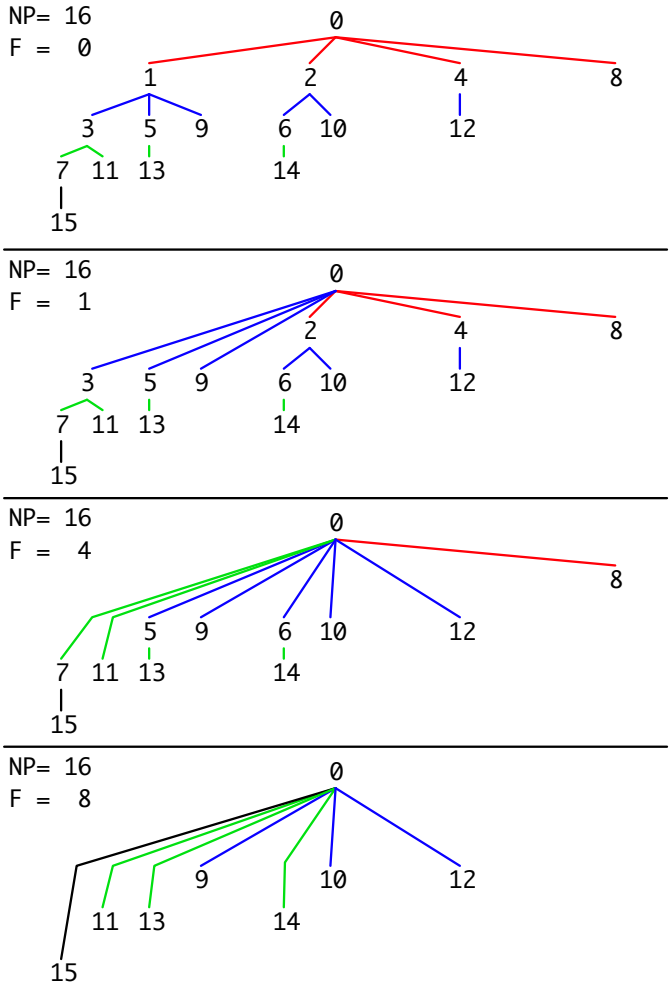

Fig. 2: Binomial tree representation with 0, 1, 4, and 8 failures in a 16 process job, using the rerouting method.

ranks to become overloaded. In the case of four failures, the root (rank 0) goes from 4 outgoing edges in the failure-free case to 8 outgoing edges. Also notice that at 8 failures the outgoing edges from the root is reduced to 7 as the tree flattens into a linear operation. This aspect will be revisited in Section IV as the cause for some performance improvement for large numbers of failures.

In the *lookup avoiding* design, the check for recognized failures is removed from the collective algorithm by calculating the parent/child relationship at the end of the MPI_Comm_validate_all function. Each rank stores its parent/child relationship in a data structure associated with the communicator. This design also removes the recursive descent of the rerouting algorithm since a full list of children is predetermined. By removing the check from the collective algorithm, this optimization is useful in determining the performance impact of frequent state lookup operations, and functional recursion in high process failure scenarios. This optimization does not address the potential for imbalance in the communication structure.

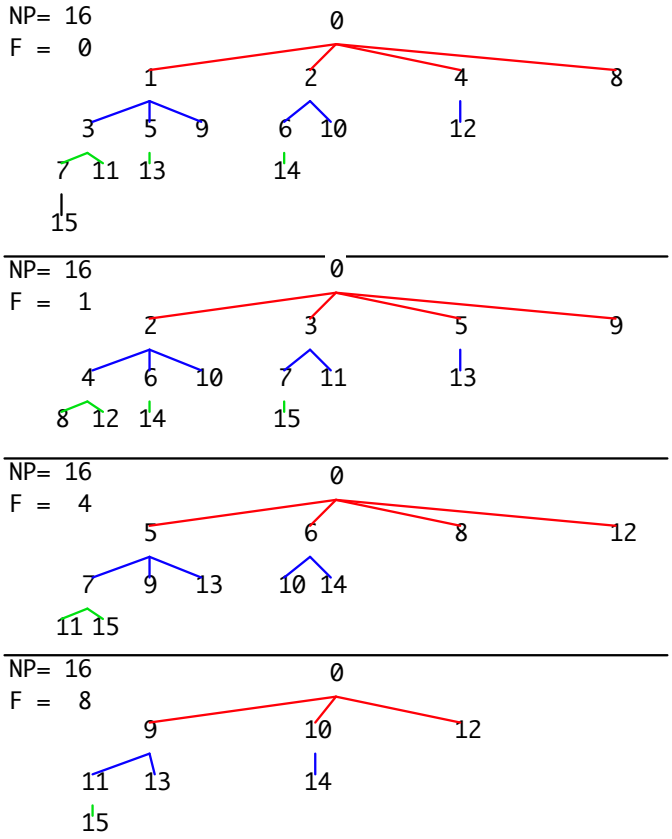

Fig. 3: Binomial tree representation with 0, 1, 4, and 8 failures in a 16 process job, using the rebalancing method.

In the *rebalancing* design, the check for recognized failures is removed, and the tree is rebalanced at the end of the MPI_Comm_validate_all function. The tree is rebalanced by projecting the alive ranks into a dense rank ordering, determining the new tree structure, and then projecting the ranks back into the sparse rank ordering. This design both avoids the recognized failure check in the collective algorithm, and the performance penalty of an unbalanced communication structure. Figure 3 illustrates rebalancing the binomial tree structure after zero, one, four and eight failures. This illustrates how the tree structure is maintained by repositioning ranks in the tree, and pruning at the leaves.

B. Other Implementation Considerations

As mentioned earlier, fault aware collective algorithms must avoid hanging processes during the collective operation when new process failures emerge. The Open MPI prototype marks each collective point-to-point operation with a flag indicating that if any new failure is detected on this communicator then this operation should complete immediately with a specific error even if the specified target is not failed. Once a failure is detected the collective algorithm returns an appropriate error code to the application. Since all processes eventually know of all process failures, this prevents any one process from hanging in the collective operation. All of the fault aware

collective operations are built using these fault aware point-to-point operations.

Recall from Section II that collectives are disabled whenever there is an unrecognized failure and are reenabled when the application calls the collective MPI_Comm_validate_all function. With this in mind, we chose to rebalance the communication structure at the end of this function when all processes have the same list of know failures in the communicator. The time to determine the new communication structure is a local, small operation (on the order of a few microseconds). The cost of the collective validate operation is beyond the scope of this paper and left to future work for further analysis.

One alternative to rebalancing at the bottom of MPI_Comm_validate_all is to rebalance after notification of every process failure. This may potentially reduce the time to complete the MPI_Comm_validate_all function by shifting the rebalance operation to the failure notification operation. In the case of new failures detected at validate time (due to the synchronization of the failure detectors) the rebalance would occur at this time anyway, negating the performance benefits. Additionally, in the case of many process failures, the rebalance operation would be called once for each process failure instead of once at validate time, a step necessary to continuing to use collectives on this communicator. With these cases in mind, we chose to only rebalance at the bottom of the collective validate when the same failure set is know to all ranks.

IV. Results

The following analysis used a prototype of the run-through stabilization proposal based on the development trunk of the Open MPI implementation of the MPI standard [4]. We created a new component of the coll Modular Component Architecture (MCA) [11] framework based on the basic component, called ftbasic. The ftbasic component contains the fault aware versions of the collectives in the basic component. By separating the fault unaware and fault aware collectives into different components we were able to switch between them at runtime to compare their performance.

In these tests, we used 64 nodes of a 128 node, Dual AMD 2.0 GHz Dual-Core Opteron machine with 4 GB of memory per compute node. Compute nodes are connected with gigabit Ethernet and InfiniBand. Only the Ethernet (tcp) and shared memory (sm) Open MPI network drivers (BTL components) were used for these tests since they are the only fully supported interconnects provided by the prototype at this time.

Our analysis focuses on the performance of MPI_Barrier since it is a latency sensitive collective operation, and will best illustrate the performance impact of the various fault aware collective design choices. The implementation of MPI_Barrier uses a binomial tree to gather and broadcast control information. In testing, specific ranks are forcibly terminated before the performance testing by sending them the `SIGKILL` signal. Ranks were selected for termination in rank order starting at rank `1` to incur maximal tree imbalance as the number of failures increases. For example, `4` failures will be represented by the failure of ranks `1`, `2`, `3` and `4`, similar to the illustration in Figure 2.

Fig. 4: Barrier performance with no failures.

The basic component provides the baseline, fault unaware performance. Baseline performance was assessed using a communicator of size $N - F$ where N is the number of processes in the job, and F is the number of failures in the job. The fault unaware performance represents the target performance for the fault aware variations. Care was taken to place processes in the same physical location on the machine during the baseline runs as in the fault aware testing, so that the results are not skewed due to the position of a process in the system.

The ftbasic component provides three implementations of the fault aware collectives able to be selected at runtime via MCA parameters. The *rerouted* implementation checks for and recursively routes around recognized failures in the communicator. The *lookup avoiding* implementation determines the proper routing at validate time avoiding both the process state lookup and the recursive descent, but still uses a potentially unbalanced tree structure. The *rebalanced* implementation rebalances the tree at validate time in addition to avoiding the state lookups and recursive descent.

Figure 4 shows the performance of the barrier operation for each of the design variations when there are no failures. This figure illustrates that as scale increases the fault aware algorithm designs are able to achieve performance equal within 1% of the baseline performance.

Figure 5 shows the performance of a 256 process job as the number of failures is increased. As the number of failures increases the overhead due to load imbalance becomes more significant. After about 8 process failures the time to complete the barrier operation starts to grow substantially. The growth continues until it reaches half of the job size, at which point the unbalanced tree becomes flat, as shown for 16 processes with 8 failures in Figure 2. Once the tree becomes flat the number of outgoing edges to the root are reduced by each subsequent failure. As the number of outgoing edges decrease the performance starts to return to nearly the performance of the rebalanced version.

Figure 5 also shows the additional overhead of the recursive nature of the rerouting technique as compared with the lookup

Failures	Baseline	Fault Aware		No Lookup		Rebalance	
0	858.74	854.48	(0.5 %)	852.46	(0.7 %)	851.26	(0.9 %)
1	880.83	860.93	(2.3 %)	860.37	(2.3 %)	872.86	(0.9 %)
2	875.06	867.91	(0.8 %)	867.12	(0.9 %)	862.74	(1.4 %)
4	837.35	840.88	(-0.4 %)	828.98	(1.0 %)	842.76	(-0.6 %)
8	832.62	910.20	(-9.3 %)	885.57	(-6.4 %)	837.15	(-0.5 %)
16	825.11	1317.16	(-59.6 %)	1244.39	(-50.8 %)	833.61	(-1.0 %)
32	815.06	2275.98	(-179.2 %)	2111.94	(-159.1 %)	823.07	(-1.0 %)
64	806.89	3245.06	(-302.2 %)	2978.67	(-269.2 %)	814.99	(-1.0 %)
96	789.39	3323.22	(-321.0 %)	2925.42	(-270.6 %)	795.53	(-0.8 %)
128	787.71	3396.60	(-331.2 %)	2869.00	(-264.2 %)	787.59	(0.0 %)
160	685.89	2572.30	(-275.0 %)	2002.26	(-191.9 %)	690.38	(-0.7 %)
192	653.65	1780.34	(-172.4 %)	1127.87	(-72.5 %)	659.10	(-0.8 %)
224	518.93	1105.24	(-113.0 %)	522.44	(-0.7 %)	522.01	(-0.6 %)

TABLE I: Barrier performance analysis with up to 224 process failures for a 256 process job. Times in microseconds.

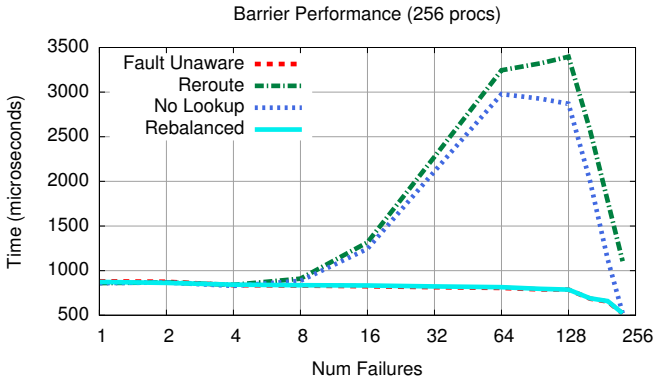

Fig. 5: Barrier performance with up to 224 process failures for a 256 process job.

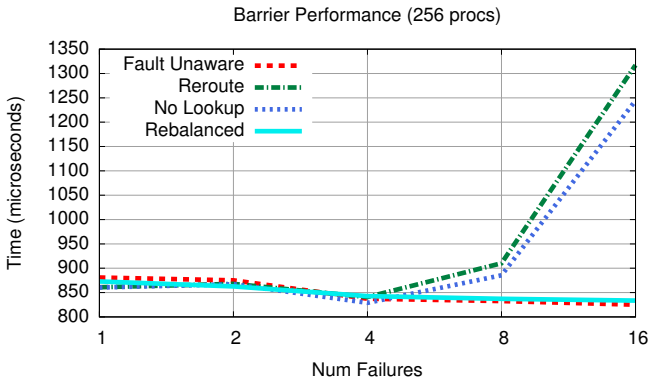

Fig. 6: Barrier performance with up to 16 failures for a 256 process job.

avoiding performance. As the number of failures increases the depth of the recursion to find the alive children of a descendant increases. The depth of recursion and the frequent state lookups contribute to this difference in performance, though the recursion overhead was the main contributor to the performance overhead.

Figure 5 and Table I also show that the rebalanced approach achieves performance within 1% of the baseline, fault unaware version. Figure 6 focuses the graph on just failures between 1 and 16 processes. This figure shows that the performance gains from rebalancing start to become noticeable after 4 process failures, and avoiding the lookup time is always beneficial.

V. Related Work

Applications have already started to experiment with integrating 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 [12]. ABFT techniques typically require data encoding, algorithm redesign, and diskless checkpointing [13] in addition to a fault tolerant message passing environment (e.g., MPI). Although matrix operations have been the focus of much of the research into ABFT [14], [15], [16], there has also been research in other domains [17].

Related to ABFT is natural fault tolerance techniques [18], [19]. Natural fault tolerance techniques focus on algorithms that can withstand the loss of a process and still return an approximately correct answer, usually without the use of data encoding or checkpointing. So natural fault tolerance can be viewed as a more general form of ABFT.

When it comes to extending the fault semantics of the MPI standard, the run-through stabilization proposal used in this paper is closely related to the FT-MPI project. FT-MPI is an MPI-1 implementation that extended the MPI communicator states and modified the MPI communicator construction functions [20]. Fault tolerant MPI applications use these extensions to stabilize MPI communicators and, optionally, recover failed processes by relaunching them from the original binary and rejoining them into the MPI communicator. The run-through stabilization proposal behaves similar to FT-MPI's blank communicator mode, where failed processes are replaced by MPI_PROC_NULL. Additionally, the two proposals 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 the same order. In contrast, the run-through stabilization proposal

preserves all communicators and groups. Additionally, FT-MPI required that every process failure be recognized globally by all alive processes in order to complete the recovery stage. In the run-through stabilization proposal, process failures can be recognized locally, and on a per-communicator basis. These two differences allow the run-through stabilization proposal to more flexibly support libraries, and, by allowing for localized failure recognition, open the door to more scalable fault tolerant solutions. However, the run-through stabilization proposal does not, at the moment, handle process recovery and rejoining recovered processes to existing communicators.

Application developers rely on optimized collective algorithms to efficiently use large scale HPC systems. There exists a substantial body of collective optimization research that use a variety of communication patterns that may be tuned to specific system designs [21], [22], [23], [24]. Though most of these algorithms are fault unaware, only the FT-MPI and Adaptive MPI (AMPI) projects provide fault aware collective variations. Though FT-MPI provides a set of tuned collective operations [25], [23], after code inspection (of version 1.0.1 [26]) it was determined that only the linear, not tree based or tuned, algorithms were used when the blank communication mode was enabled. In FT-MPI, a dense *shadow* communicator, associated with every communicator, is used for collective operations internally. In the designs presented in this paper, instead of a shadow communicator a structure containing a reference to the root, parent, and list of children is associated with a communicator for use in the collective operation. In the lookup avoiding and rebalanced designs, the binomial tree based algorithms use this structure to determine the communication pattern over the original communicator. The recursively rerouted design routes around recognized failed processes in the tree during the collective operation. Since FT-MPI does not provide fault aware, tree-based collective operations we were not able to directly compare the designs beyond this analysis.

The AMPI project provides fault tolerant collective operations that can operate across process migration activities based on a k-ary tree communication structure [27]. AMPI requires that all failures be predictable so that effected processes can be migrated before a failure. As a result the collectives are able to rely on every process continuing to participate in the collective across the process migration, and need not account for permanent process failures, as in the run-through stabilization scenario presented in this paper. The collective algorithms provided by AMPI use a technique similar to the recursive rerouting technique when handling a predicted fault inside an active collective operation. Once the operation is complete, AMPI then rebalances the communication topology for successive operations. The performance results shown in [27] complement the rerouting and rebalancing results presented in this paper, though this paper focuses on maintaining the performance even when processes are permanently failed.

The MPI/FT and C^3 projects approached MPI collective operations in a slightly different manner than the FT-MPI and AMPI projects. The model based approach of the MPI/FT project provides customized solutions to a few different application execution models. This project requires that either failed processes be replaced (i.e., from a checkpoint or spare process) or collective operations are prohibited, as in the manager/worker model [28]. These restrictions indicate that the MPI/FT project did not implement fault aware MPI collective algorithms that handle permanently failed processes in the communicator.

The C^3 project elaborates on the challenges of handling collective operations at the application level to support application level checkpointing [29]. As with the MPI/FT project, the C^3 project replaces failed processes by restarting the application from the last stable checkpoint, so the collective algorithms also do not have to explicitly handle permanently failed processes in the communicator.

VI. Conclusion

Collective operations are an important component of many scalable HPC applications, and the focus of many years of algorithmic optimization research. Unfortunately, most of this research does not account for emergent and existing process failure. A conventional approach to building fault aware collectives is to recursively route around failed processes. This paper demonstrated that such an approach can lead to significant performance degradation, up to four times at a relatively small scale. We explored two alternative approaches, lookup avoiding and rebalancing, that yielded performance benefits over the recursive rerouting approach. The lookup avoiding approach was able to improve the performance in all tests by avoiding the recursive descent and shifting the check for recognized failures from every collective operation to once at the end of the collective MPI_Comm_validate_all function. The rebalancing approach further improved this by maintaining the balanced tree and achieved performance within 1% of the fault unaware collective algorithm regardless of the number of failures. Though the analysis used a binomial tree communication pattern, the design techniques described are applicable to other collective communication topologies.

As future work, we intend to investigate fault aware versions of more advanced collective algorithms. Some such algorithms use alternative communication structures that adjust to the network and node topologies of a given system. Additionally, we intend to investigate scalable implementation variations of the collective MPI_Comm_validate_all operation which plays a key role in the run-through stabilization proposal. Beyond these items, we will continue to work on extending the prototype to support the full MPI standard on a wider variety of platforms. The prototype development will match the pace of the development of the fault tolerance proposal being generated by the MPI Forum's Fault Tolerance Working Group.

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. The Odin machine at Indiana University was provided by grant EIA-0202048 from the National Science Foundation.

REFERENCES

[1] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, 2007.

[2] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[3] Message Passing Interface Forum, "MPI: A Message Passing Interface," in *Proceedings of Supercomputing '93*. IEEE Computer Society Press, November 1993, pp. 878–883.

[4] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[5] Fault Tolerance Working Group, "Run-though stabilization interfaces and semantics," svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_stabilization.

[6] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[7] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[8] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[9] M. Barborak, A. Dahbura, and M. Malek, "The consensus problem in fault-tolerant computing," *ACM Computing Surveys*, vol. 25, no. 2, pp. 171–220, 1993.

[10] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on reliable software*. New York, NY, USA: ACM Press, 1975, pp. 437–449.

[11] J. M. Squyres and A. Lumsdaine, "The component architecture of Open MPI: Enabling third-party collective algorithms," in *Proceedings of the 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, V. Getov and T. Kielmann, Eds. St. Malo, France: Springer, July 2004, pp. 167–185.

[12] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. 33, no. 6, pp. 518–528, 1984.

[13] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, October 1998.

[14] Z. Chen and J. Dongarra, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1628–1641, 2008.

[15] Y. Du, P. Wang, H. Fu, J. Jia, H. Zhou, and X. Yang, "Building single fault survivable parallel algorithms for matrix operations using redundant parallel computation," *International Conference on Computer and Information Technology*, pp. 285–290, 2007.

[16] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra, "Recovery patterns for iterative methods in a parallel unstable environment," *SIAM Journal of Scientific Computing*, vol. 30, no. 1, pp. 102–116, 2007.

[17] H. Ltaief, E. Gabriel, and M. Garbey, "Fault tolerant algorithms for heat transfer problems," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 663–677, 2008.

[18] C. Engelmann and A. Geist, "Super-scalable algorithms for computing on 100,000 processors," in *Proceedings of International Conference on Computational Science (ICCS)*, vol. 3514, no. 1, May 2005, pp. 313–320.

[19] A. Geist and C. Engelmann, "Development of naturally fault tolerant algortihms for computing on 100,000 processors," *Journal of Parallel and Distributed Computing*, 2002.

[20] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, and J. J. Dongarra, "Process fault-tolerance: Semantics, design and applications for high performance computing," *International Journal for High Performance Applications and Supercomputing*, vol. 19, no. 4, pp. 465–478, 2005.

[21] T. Kielmann, R. Hofman, H. Bal, A. Plaat, and R. Bhoedjang, "MagPIe: MPI's collective communication operations for clustered wide area systems," *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on principles and practice of parallel programming*, vol. 34, no. 8, pp. 131–140, Aug 1999.

[22] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "On optimizing collective communication," *IEEE International Conference on Cluster Computing*, pp. 145 – 155, 2004.

[23] G. Fagg, G. Bosilca, J. Pješivac-Grbović, T. Angskun, and J. Dongarra, "Tuned: An Open MPI collective communications component," *Distributed and Parallel Systems*, pp. 67–72, Jan 2007.

[24] A. Faraj, X. Yuan, and D. Lowenthal, "STAR-MPI: self tuned adaptive routines for MPI collective operations," *Proceedings of the 20th annual international conference on supercomputing*, Jan 2006.

[25] G. E. Fagg, T. Angskun, G. Bosilca, J. Pješivac-Grbović, and J. Dongarra, "Scalable fault tolerant MPI: extending the recovery algorithm," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 366/2005, pp. 67–75, Jan 2005.

[26] Innovative Computing Laboratory, "FT-MPI version 1.0.1," http://icl.cs.utk.edu/ftmpi/software.

[27] S. Chakravorty, C. Mendes, and L. Kalé, "Proactive fault tolerance in MPI applications via task migration," *High Performance Computing*, vol. 4297, pp. 485–496, Jan 2006.

[28] R. Batchu, Y. S. Dandass, A. Skjellum, and M. Beddhu, "MPI/FT: a model-based approach to low-overhead fault tolerant message-passing middleware," *Cluster Computing*, vol. 7, no. 4, pp. 303–315, Jan 2004.

[29] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Collective operations in application-level fault-tolerant MPI," *Proceedings of the 17th annual international conference on Supercomputing*, Jan 2003.