Tree-Based Fault-Tolerant Collective Operations for MPI

Alexander Margolin
Department of Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

Amnon Barak
Department of Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

ABSTRACT

With the increase in size and complexity of high-performance computing systems, the probability of failures and the cost of recovery grow. Parallel applications running on these systems should be able to continue running in spite of node failures at arbitrary times. Collective operations are essential for many parallel MPI applications, and are often the first to detect such failures. This work presents tree-based fault-tolerant collective operations, which combine fault detection and recovery as an integral part each operation. We do this by extending existing tree-based algorithms, to allow for a collective operation to succeed despite failing nodes before or during its run. This differs from other approaches, where recovery takes place after a failure of such operations have failed. The paper includes a comparison between the performance of the proposed algorithm and other approaches, as well as a simulator-based analysis of performance at scale.

CCS CONCEPTS

• **Software and its engineering** → **Software fault tolerance**;

KEYWORDS

Fault-tolerance, Collective Operations, MPI, Allreduce

1 INTRODUCTION

With increasing computational power of HPC systems, it is not unreasonable to expect faults through the course of executing a parallel application. With this in mind, different approaches had been explored to anticipate, mitigate and overcome different types of faults. MPI Collective operations, unlike point-to-point communication, for example, typically span across a large number of processes. As a result, the effect of a fault during a collective operation is wide-spread and there is no simple mechanism for recovery. Common approaches, such as Checkpoint-Restart, require a considerable error propagation and recovery flow once a fault has been detected. Furthermore, fault detection typically relies on either an arbitrary communication timeout or a constant keep-alive mechanism, both of which may incur a significant penalty on the performance of a resilient application. Also, some fault-tolerance mechanisms require changes in the application code to make it resilient, and most also incur run-time overhead due to the use of CPU and other resources.

The MPI standard specifies little in excess of a dedicated return value for errors, and there are multiple outstanding proposals on how to address faults during MPI applications. Most of the work could be characterized as either "backward recovery" or "forward recovery": the former tries to restore a correct past state, and the latter would attempt to establish a new correct state. ULFM [2], a prominent proposal for forward recovery in MPI, presents a set of tools for an application to deal with faults once detected. It proposes an error handler, capable of propagating the knowledge of the fault among MPI processes, and run-time correction of communicators containing a faulty process.

This paper presents the Fault-Tolerant Collective Operations (FTCO) paradigm, and a parallel algorithm that applies it to tree-based collective operations in MPI. FTCO includes resilient versions of collective operations such as *Bcast*, *Reduce* and *Allreduce*, for any tree topology. The FTCO algorithm detects faults by a per-node calculated timeout and overcomes faults by excluding failed processes. It is intended for use-cases that can tolerate process faults, such as Monte Carlo method or PDE solvers. FTCO delivers a result to every live process, so that the application may keep running without handling those faults. The FTCO algorithm minimizes the fault-free performance penalty and shows a small increase in latency and messages per fault, regardless of job size. This offers a transparent and scalable forward recovery alternative to the costly legacy backward recovery mechanisms.

The paper is organized as follows: Section 2 presents the communication model for the Tree-based FTCO Algorithm; Section 3 describes its rationale and Section 4 its details. Section 5 presents the performance of FTCO. Related work and conclusions are shown in Sections 6 and 7.

2 THE COMMUNICATION MODEL

We use a communication model with discrete steps, where in each step the algorithm handles one message: either send or receive from an inbound message queue. The latency of any message is one or more steps. We assume the network topology is a fully connected graph, where every edge allows reliable full-duplex communication with no communication-related faults. For simplicity we also assume that one designated (**root**) node is immune to failure. However, the algorithm can include support for root failure.

To guarantee correctness and termination we use the fail-stop model, where a node may fail at some time $t \geq 0$, and after failure that node takes no action. Otherwise, a node could recover right before completion. Messages sent before the failure of a node are unaffected, and messages to that node are discarded since. We distinguish between *inactive* nodes, failing at $t = 0$, and run-time faults ($t > 0$). In our model a failure is detected using the message timeout. If a node does not get a response (ACK) within an expected period of time, it assumes that the responder is dead.

3 THE ALGORITHM RATIONALE

This section presents the rationale for our FTCO algorithm for collective operations based on tree topologies. We start by defining fault-tolerance for a collective operation. Let $FailureTime_i$ be the time when node i fails, where $FailureTime_i = \infty$ if the node never fails.

Definition 3.1. A *Fault-tolerant collective operation* is a collective operation with a finite completion time T, where the result either includes or excludes any node i for which $FailureTime_i < T$.

By applying Definition 3.1 to an *Allreduce* with a binary operator $\oplus$ (on each node's value); and both $FailureTime_i$ and $Value_i$ for each node i, we get:

Definition 3.2. Fault-tolerant Allreduce returns on every node $Value_{result} = Value_{j_1} \oplus ... \oplus Value_{j_n}$, for some $ReducedNodes = \{j_1, .., j_n\}$, so that $RemainingNodes \subseteq ReducedNodes \subseteq Nodes$ and $RemainingNodes = \{i | i \in Nodes \wedge FinishTime_i < FailureTime_i\}$.

Observe that the result of the fault-tolerant collective operation (Definition 3.1) is not necessarily deterministic, but is required to be consistent. For *Allreduce*, consistency means an equal result among all the active nodes. In other collective operations, e.g. *Reduce-Scatter*, this means that if a value from a failed node is excluded from the result, it will be disregarded on all nodes.

Message Flow

In order to detect faults, the algorithm sets a timeout for each sent message. The timeout serves as notice that the destination node is unresponsive and can be ignored. The types of messages that are passed between the nodes are:

(1) *DATA* - Passing a value along the tree.
(2) *KEEPALIVE* - Explicit fault detection (no data sent).
(3) *ACK* - Acknowledgment of *DATA* or *KEEPALIVE*.

The message flow of the fault-tolerant algorithm includes sending *DATA* along the tree, and expecting an *ACK* from each destination node. In the case of *Reduce*, first the node receives *DATA* messages from all its children; then it passes *DATA* to its parent, followed by *ACK* messages to all its children; and waits for an *ACK* from its parent. This flow is illustrated in Figure 1 for node 1. The flow for *Broadcast* starts by waiting for *DATA* from its parent, then it distributes *DATA* to the children followed by an *ACK* to the parent, see node 1 in Figure 2. The *ACK* send overlaps with the wait for the parent's *ACK*, making the latency equivalent to a fault-free tree traversal. The other alternative of first waiting for the parent's *ACK* would add the *ACK* tree traversal to the overall latency.

In the FTCO algorithm, failure is detected when a node sends either a *DATA* or a *KEEPALIVE* message to a peer and does not receive an *ACK* within the timeout period for that message. In such a case we circumvent the failed node, and resend the same message to either the parent or all the children of the failed node. Figure 3 illustrates this flow for *DATA*, and Figure 4 for *KEEPALIVE*. If there is a fault in several directly connected nodes, some nodes would send a message several levels up or down the tree. In the worst case, going down the tree may lead to sending messages to every node in the subtree before the operation is completed. Similarly, going up the tree sends one message per tree level and may eventually lead to sending directly to the root or any other node if the root fails.

Figure 1: Reduce - Correct Message Flow

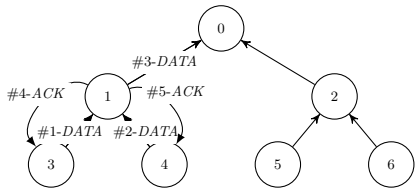


Figure 2: Broadcast - Correct Message Flow

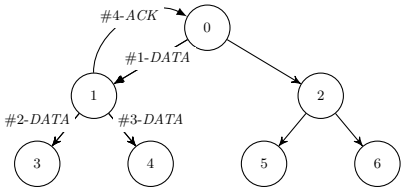


Figure 3: Message flow - *DATA* timeout

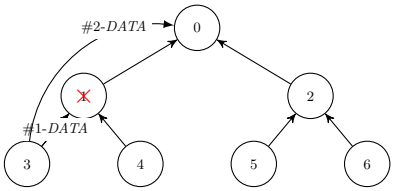
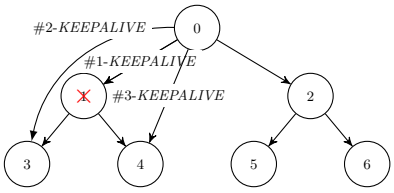


Figure 4: Message flow - *KEEPALIVE* timeout



Message Priorities

The essence of the algorithm is prioritization of *DATA* over other messages. Whenever *DATA* messages are present in the incoming queue, those are handled first, while other messages are processed only after the data was passed onward. This favors latency over and at the expense of fault detection time. The outcome of a run of our fault-tolerant algorithm without faults differs in latency by one *ACK* compared to the fault-free algorithm.

While our algorithm applies to any number of faults, in this paper we assume a small number of faults at the start and during the algorithm run. In general, *Allreduce* could also aggregate and distribute information about faults, in addition to reduction values, in order to reduce the detection time of faults in the next operations.

4 THE FTCO ALGORITHM DETAILS

The fault-tolerant algorithm consists of four parts: the main part invokes the tree traversal, the incoming message handler and the fault detection routine.

The **main** part is listed in Algorithm 1. The algorithm iteratively calls *tree_progress()* (Algorithm 3) to move the collective operation forward by send and receiving *DATA* messages. Algorithm 1 receives from *tree_progress()* an indication whether a message was sent during this call and also if the entire operation was completed. If neither occurred, for this iteration we choose a group of nodes to prioritize with a distance $service\_distance \in \mathbb{N}$ from the current node (*get_service_distance*() is shown in Appendix A). To complete this iteration, the algorithm attempts to sequentially perform the following tasks: (a) read an inbound message from a prioritized node (line 8); (b) check for faults on any of the prioritized nodes (line 9); (c) read any message from the inbound queue (line 10); and (d) check for faults on any relevant node (line 11). Once any of these tasks succeeded by sending or receiving a message, the iteration completes and the next iteration starts, unless Algorithm 3 indicates completion (line 5). Appendix B describes the case of process imbalance.

Algorithm 1: Fault-tolerant Collective Operation Main

```
1  int fault_tolerant_collective(initial_value)
       Input: The Value to perform the aggregation on
       Output: True if finished, False otherwise
2      Value ← initial_value;
3      is_done ← false;
4      while ¬is_done do
5          ⟨is_sent, is_done⟩ ←tree_progress(Value) ;
6          service_distance ← get_service_distance();
7          if ¬is_sent ∧ ¬is_done
8            ∧¬ handle_message_queue(service_distance)
9            ∧¬ fault_tolerance(service_distance)
10           ∧¬ handle_message_queue(0) then
11             fault_tolerance(0);
12     return Value;
```

The routine **handling incoming messages**, shown in Algorithm 2, prioritizes messages in the following order:

(1) *DATA* messages are aggregated or passed along the tree.
(2) *KEEPALIVE* messages are stored pending response.
(3) *ACK* messages indicate their sender has not failed.

The *should_handle* routine is used to test if a node is prioritized, i.e., matches the selected *service_distance* (0 is used as wild-card). We note that non-idempotent aggregation operators, such as addition, require a specific routine (instead of line 16 in Algorithm 2) so that each value is used only once. This routine could also support result reproducibility for the case of a non-associative aggregation.

Algorithm 2: Handling Incoming Messages

```
1  bool should_handle(node, service_distance)
2      if (service_distance = 0)
3        ∨(service_distance = distance(me, node)) then
4          return true;
5      return false;
6  void handle_message_queue(service_distance)
7      for type in [DATA,KEEPALIVE,ACK] do
8          for msg ∈ msg_queue do
9              if should_handle(msg.source,
10             service_distance) ∧ (msg.type = type) then
11                 switch type do
12                     case DATA do
13                         if msg.source ∈ Up ∧ |Up| > 1
                            then
14                             Value ← msg.value;
15                         else
16                             Value ← Value ⊕ msg.value;
17                     case KEEPALIVE do
18                         ToACK ←
                            ToACK ∪ {msg.source};
19                     case ACK do
20                         LastACK[msg.source] ← now;
21                 msg_queue ← msg_queue − {msg};
22                 return true;
23     return false;
```

For each collective operation, the **Tree Traversal** routine propagates the data along the respective tree. For example, Algorithm 3 demonstrates *Allreduce*. First each node waits for incoming *DATA* from all its children, and then sends the aggregated *DATA* to its parent. After that it waits to obtain the result from the parent, and then sends it to all its children. When a *DATA* message is sent or the next value is missing the routine ends. Once all the *DATA* messages were acknowledged, this collective operation is complete.

The **Fault detection and recovery** routine, shown in Algorithm 4, describes the method for detecting and overcoming node failures. This routine consists of three ordered elements: overcoming faults by skipping failed nodes; responding with an *ACK* to outstanding keep-alive requests; and sending *KEEPALIVE* messages to nodes suspected as failed. The routine returns once any of these elements sent or received a message. Algorithm 4 relies on the following three functions,

Algorithm 3: Allreduce Tree Traversal

```
1  bool[2] tree_progress()
       Data: The Value and the set of nodes arrived so far,
             Up_sent/Down_sent are where data was sent, initially ∅
       Output: One boolean to indicate if a message was
               sent, another if the traversal is complete
2      if Down ⊈ Arrived then
3          return ⟨false, false⟩
4      for x ∈ Up \ Up_sent do
5          Send Value to x as DATA;
6          Up_sent ← Up_sent ∪ {x};
7          return ⟨true, false⟩
8      if Up ⊈ Arrived then
9          return ⟨false, false⟩
10     for x ∈ Down \ Down_sent do
11         Send Value to x as DATA;
12         LastSent[x] ← now;
13         Down_sent ← Down_sent ∪ {x};
14         return ⟨true, false⟩
15     return ⟨false, true⟩
```

Algorithm 4: Fault Detection and Recovery

```
1  bool fault_tolerance(service_distance)
       Output: True if a message was handled, False
               otherwise
2      for x ∈ (Up ∪ Down) do
3          if should_handle(x, service_distance)
4            ∧(LastACK[x] < LastSent[x])
5            ∧((now − LastSent[x]) > f_timeout(me, x))
             then
6              ⟨Up_x, Down_x⟩ ← f_tree(x);
7              if (x ∈ Up) then
8                  Up ← (Up_x ∪ (Up − {x}));
9              else
10                 Down ← (Down_x ∪ (Down − {x}));
11             return false;
12     for x ∈ (Up ∪ Down) do
13         if should_handle(x, service_distance)
14           ∧(x ∈ ToACK) then
15             send ACK to x;
16             ToACK ← ToACK − {x};
17             return true;
18     for x ∈ (Up ∪ Down) do
19         if should_handle(x, service_distance)
20           ∧(x ∉ Arrived)
21           ∧(now > f_ETA(x))
22           ∧((now − LastSent[x]) > f_timeout(me, x))
             then
23             send KEEPALIVE to x;
24             LastSent[x] ← now;
25             return true;
26     return false;
```

which calculate certain parameters without traversing the tree:

- $f_{tree}(n) \rightarrow < Down, Up >$, where $n \in Nodes$, to calculate each node's parent and children prior to any operations, based on the tree topology, also used for Algorithm 3.
- $f_{ETA}(n) \rightarrow \mathbb{N}$, to calculate the expected arrival time of data to node n.
- $f_{timeout}(source, destination) \rightarrow \mathbb{N}$, to calculate the message timeout for fault detection, based on the message source and destination nodes.

Data Propagation Disruptions

In the FTCO algorithm, we call a "disruption" a distinct pattern of node failures that stop the *DATA* propagation. Consider a chain of at least four consecutive tree nodes, in which the third node has failed. The second node receives a *DATA* message from the first node, then it sends the *DATA* to the third node, followed an *ACK* to the first, and then it fails. In this scenario the fourth node would never have received the data, even though it has not failed.

In order to overcome disruptions, each node calculates its expected data arrival time based on its location in the tree, assuming no nodes fail (f_{ETA}). For *Allreduce* two such time values are used: one for the subtree data and the other for the final result. If either of these times elapses before data arrives, a node switches from passive waiting (no messages sent) to active fault-detection: *KEEPALIVE* messages are sent to every suspected node at some time interval until the data arrives. A node is suspected if it was expected but had not sent *DATA* before the expected arrival time.

In the previous example, the fourth node would send a *KEEPALIVE* messages to the third node once the waiting period elapses (f_{ETA}). In the absence of an *ACK* ($f_{timeout}$ steps) from the third node, it will send a *KEEPALIVE* to the second node. After another timeout, it will send a *KEEPALIVE* to the first node, which responds with *DATA* and resumes the operation. The result of the operation will exclude the failed nodes.

This approach for overcoming disruptions requires the root, to remain active throughout the collective operation. Separating the role of supporting fault-tolerance from the role of the root node can allow for the root node to complete the operation, in accordance with the MPI semantics. We note that FTCO can function without the fault-immune root assumption: upon fault the root would be replaced by all its children (extending Algorithm 3).

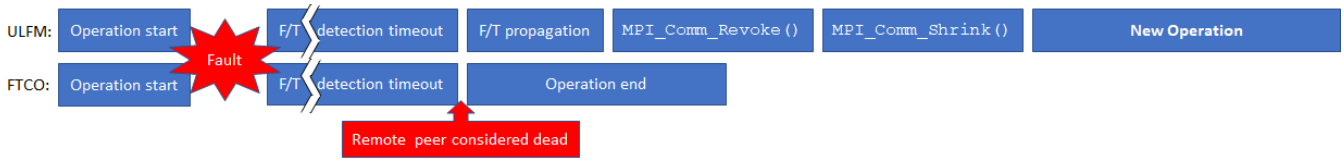


Figure 5: The steps of fault tolerance in collective operations

5 PERFORMANCE EVALUATION

This section presents the performance of the tree-based FTCO algorithm. We extended the Open-MPI implementation of ULFM with an FTCO option and measured the latency of the *Allreduce* operation, with and without faults. We also used a simulator to measure the corresponding performance for large numbers of nodes and faults in our formal communication model.

5.1 Experimental Results

We begin by comparing the step performed by FTCO and ULFM when a failure is encountered. Figure 5 illustrates (not to scale) the steps performed in each method along the timeline of a collective operation, with respect to fault tolerance. Both methods start by running the collective operation (leftmost block) until a fault occurs, followed by a timeout of the fault detection mechanism. From this point onwards, the two methods differ. ULFM propagates the fault notification, revokes the failed collective operation, repairs the communicator and restarts the operation. In contrast, after detecting a failure, our Fault-Tolerant Collective Operation (FTCO) algorithm resumes the collective operation without the failed nodes. FTCO is transparent to applications, whereas in ULFM users control how the communicator is repaired. We note that if necessary, FTCO could pass similar information to the application.

In order to compare between ULFM and FTCO, we introduced an option to perform ULFM's keep-alive only upon demand, i.e., when two nodes communicate within a collective operation. Specifically, keep-alive messages in FTCO are sent only along the edges of the tree and only when data is expected. We also introduced "grand-parent" nodes to the tree topology calculation, to provide fall-back when parent nodes are suspected (based on the keep-alive) to have failed. A detailed description of these modifications can be found in Appendix D.

In a tree topology, multiple faults could be either serial or parallel: parallel faults occur in different sub-trees, thus their recovery time may overlap, while serial faults are handled one after the other. Table 1 shows the FTCO latency of the *Allreduce* operation for different combinations of offline faults and increasing number of processes. In the table, each figure represents the average longest time (in seconds) for a process to complete the same *Allreduce* call.

From the table it can be seen that the latency with FTCO is proportional to the number of serial faults: two parallel faults are approximately the same as a single fault; and adding a third parallel fault to two serial faults does not change the latency. We note that the 2 seconds are the current default ULFM timeout period. This timeout is added to the overall latency, depending on the combination of the faults. The FTCO algorithm includes a method for calculating the timeout for each sent message, depending on the distance between the message source and destination along the tree (see Appendix A).

No. of Proc.	No Faults	1 Fault	2 Parallel Faults	2 Serial Faults	3 Mixed Faults
64	0.0005	2.0095	2.0094	4.0194	4.0194
128	0.0006	2.0123	2.0123	4.0221	4.0222
256	0.0007	2.0121	2.0123	4.0220	4.0219
384	0.0017	2.0114	2.0323	4.0404	4.0217
512	0.0024	2.0343	2.0110	4.0421	4.0422

Table 1: FTCO *Allreduce* latency (in seconds)

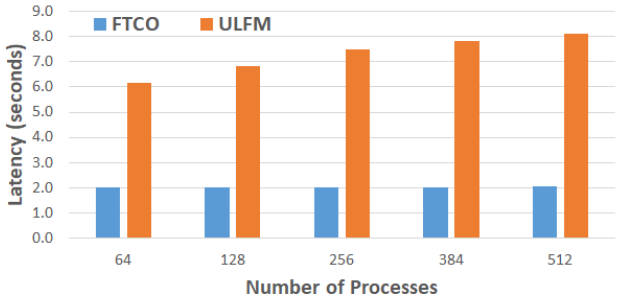


Figure 6: The latency cost of one fault (in seconds)

Figure 6 shows the latencies of FTCO and ULFM for an *Allreduce* operation with a single fault and an increasing number of processes. In both cases we used the same parameters, e.g., fault detection timeouts and tree topology. Each test started by injecting a fault to one process and then attempted an *Allreduce* on multiple communicators with that process. The resulting overall latency includes all the steps shown in Figure 5, which in the case of ULFM includes calls to *MPIX_Comm_Revoke* and *MPIX_Comm_Shrink*, followed by another *Allreduce* attempt. We note that when running without faults, the mature ULFM package shows up to 50% better latency than our FTCO prototype implementation.

5.2 Formal Model and Results

We now present a formal model for estimating the performance of FTCO with a large numbers of nodes and faults. For the formal analysis we use the *LogP* model [4] in a discrete environment. This model has been shown to predict

the network latency of a protocol based on four parameters: the maximal Latency (L), Overhead (o), Gap (g), and the number of Processors (P), where P is the number of nodes. For simplicity, we set $o = 1$ as the unit of time (**step**), and assume L is a multiple of o ($o < L$). At each step each node can either send or receive a single message. For small messages it has been shown [9] that often $g \ll o$ and thus can be neglected. A message sent from one node arrives after $L + o$ steps into the incoming queue of another node. The receiver can handle one incoming message from the queue at each step, which includes the reduction operation if applicable. We assume that all the nodes use the same unit of time, noting that our model does not quantify various reasons for delays in real systems, such as system noise or packet retransmission, which in our case are reflected in the L/o ratio. We note that FTCO can be part of a hierarchical collective operation, where intra-node communication is faster and less likely to fail, and thus it is excluded in this model.

To simulate tree-based collective operation algorithms at scale, we implemented a simulator application. The parameters of the simulator include the number of nodes and the tree topology; the service-distance function; the number of offline (inactive) nodes and run-time node failures. The results include the latency, in discrete steps, which could be used to estimate the latency for a specific cluster based on *LogP* parameters. The results also include the work (sent messages) per node and the maximal queue length in any node. Simulations were performed for a cluster with $P = 1K - 64K$ nodes; with 0,1,10 and 100 inactive nodes or run-time faults.

5.2.1 The Tree Topologies. We ran simulations on trees with K-ary and K-nomial topologies. K-ary tree has a fixed radix k, while a K-nomial tree has a variable radix and a parameter k - which determines the radix at each node. For example, in a broadcast, a k-nomial tree is produced by forwarding up to $k - 1$ messages from each node to its children in each iteration, until all nodes are reached. K-nomial trees are commonly used for large-scale reductions [13]. The standard topology for collective operations, e.g., in MPI, is a single-root tree. Another useful topology which improves the latency, the *multi-root*, uses a collection of trees with fully-connected roots.

We selected the network latency $L = 10$ and message send/receive overhead $o = 1$. Based on the results of Papadopoulou et al. [14], the small message send rate is $6.5 * 10^6$ per Sec., (implying $o = 1.3 * 10^{-7}$ Sec.), and P2P latency of $L = 1.53 * 10^{-6}$ Sec., over Infiniband™.

We found that for a given k, the K-nomial topology outperforms K-ary. We also found the optimal K-nomial parameter k that produces the minimal latency of the *Broadcast*, *Reduce* and *Allreduce* for the above parameters without faults or process imbalance. Figure 7 shows (in the two topmost lines) the latency for *k-nomial* (single-root and multi-root) topologies with the optimal parameter k. The bottom line shows the latency for the optimal tree.

From the figure it can be seen that the latency of the multi-root topology is lower than that of a single-root. Thus,

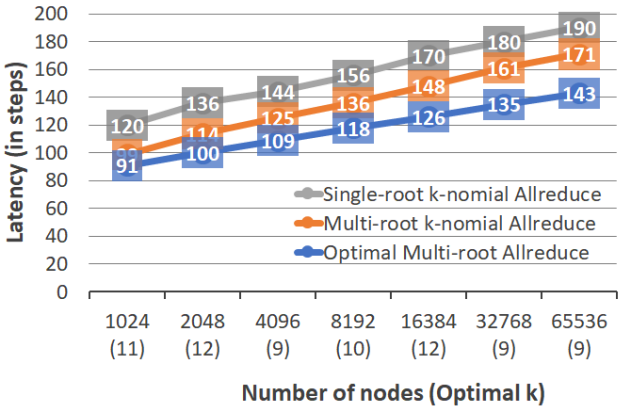


Figure 7: Latency of Fault-free Collective Operations

in the rest of this section we measure the *Allreduce* over multi-root k-nomial topology, with the matching optimal k for the above range of nodes.

5.2.2 The Cost of Fault-tolerance. The cost of fault-tolerance depends on the number of nodes, the time and location of the fault(s); and the service-distance function. For example, as explained in Section A, the message timeout depends on the locations of the source and failed destination nodes.

Figure 8 presents the average increase in the latency (penalty) per inactive node (compared to the fault-free run), over 10000 runs with random placements of the inactive nodes. The corresponding average messages sent per node is presented in Figure 9.

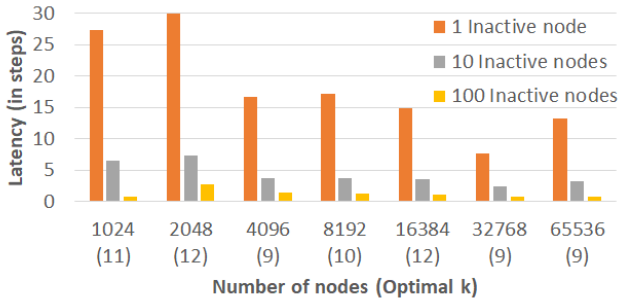


Figure 8: Latency penalty per fault of *Allreduce*

In Figure 8, for each number of nodes and the optimal radix, the leftmost bar presents the latency penalty for 1 inactive node; the middle bar for 10 and the rightmost bar for 100 inactive nodes. From the figure it can be seen that the penalty decreases as the number of inactive nodes is increased. This is due to the random placement of failed nodes, allowing the algorithm to perform fault detection and recovery in parallel. In an ideal placement of inactive nodes the penalty would have been in inverse proportion to their number, compared to a single inactive node. However, due to possible combinations of inactive nodes within close proximity, the actual penalty is higher.

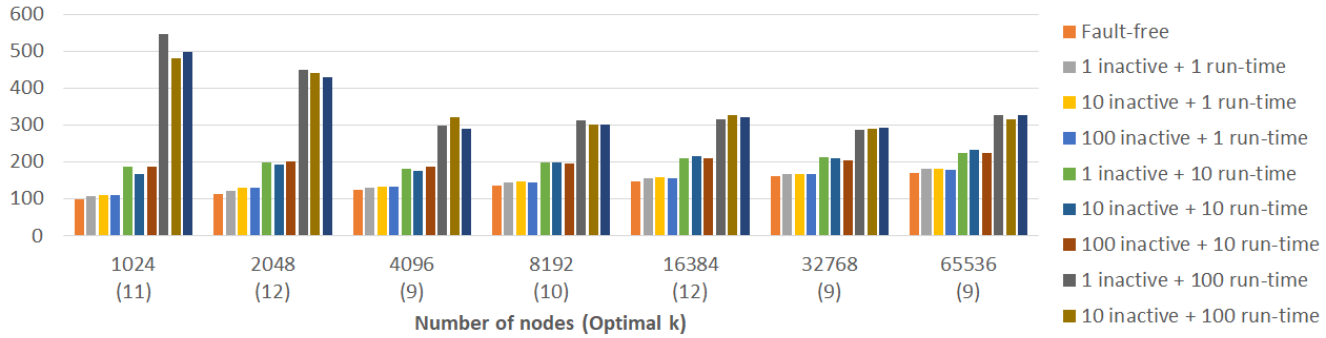


Figure 10: Total latency for *Allreduce* for different combinations of faults

As the number of nodes increases, the change in the penalty is related to the expected value of the fault detection timeout for a randomly placed inactive node. Observe that adding a new level to the tree, with a significant number of nodes, reduces the average penalty, e.g. the reduction between 2K and 4K, since inactive leaves have a low degree and timeout. In contrast, adding new nodes without increasing tree height increases the average penalty of the original nodes. We note that in a randomized algorithm, such as Corrected-Gossip [7], such phenomena does not occur due the lack of a fixed topology.

Figure 9 shows the average increase in the number of messages in all the nodes, for each inactive node. For example, with 1K nodes and one inactive node, each of the 1023 nodes sends on average 0.5 more messages during a run. We note that in a fault-free run, each node sends on average 3 messages: each node sends *DATA* and *ACK* both up and down the tree, but the operation completes before the leaf nodes, which are most of the nodes, send the last *ACK* message.

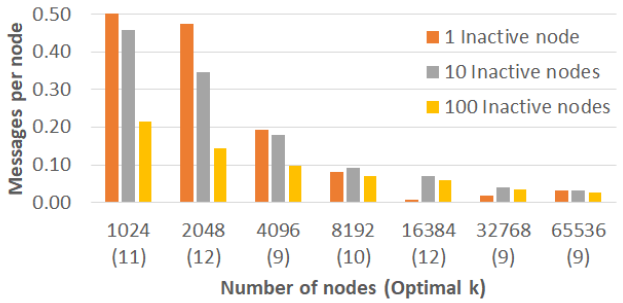


Figure 9: Work penalty per fault of *Allreduce*

5.2.3 Inactive Nodes and Run-time Faults. Figure 10 compares the latency overhead when there are inactive nodes and also run-time faults. From the figure it can be seen that the overhead incurred by run-time faults is more significant then that of the inactive nodes. Even a single fault delays in the collective operation, causing *KEEPALIVE* and *ACK* messages to be sent from every node. This triggers the detection of other faults across the tree at that time. In a practical implementation there would be very few offline nodes in a collective operation call, since the detection of a fault would mark the failed node for further calls. The longest incoming queue on any node was also measured, and during these runs it was at most 130 pending messages, justifying the assumption that this resource is not a limiting factor.

5.2.4 Process Imbalance. Process arrival time imbalance, or process imbalance, is caused by different application workloads between nodes - making collective operations start later on some nodes. For such node i we define the $StartTime_i$ as the delay, in steps, before the collective operation starts on that node (compared to the baseline $t = 0$). For the entire collective operation, we denote the following times:

- $StartTime_{first}$ - The first node starts
- $StartTime_{last}$ - The last live node starts
- $RootTime$ - One node knows the final result (the root)
- $FinishTime_{first}$ - First node finishes
- $FinishTime_{last}$ - Last non-root node finishes

In addition to the simple model where all nodes start simultaneously, where $StartTime_{first} = StartTime_{last} = 0$, we also analyze the case of process arrival time imbalance. We measured the effect of the imbalanced process arrival time (spread) on the performance, as well as on the process departure time, in a cluster with 64K nodes. Extensions of the FTCO algorithm for process imbalance are shown in Appendix B. We considered two probability distributions for the process arrival times: a uniform and a normal distributions, with $StartTime_{first} = 0$. For both distributions, Table 2 shows the average latency and work (sent messages per node) to complete the operation for an average input spread ranging up to 8K steps. It also includes the longest incoming message queue length throughout any of the runs, and the resulting output spread ($FinishTime_{last} - FinishTime_{first}$).

Table 2 shows several characteristics of tree-based algorithms. The Collective operation latency (measured as $FinishTime_{last} - StartTime_{last}$) is shown to decrease slightly as the spread increases, thus "absorbing" the process imbalance [16]. This is due to progress achieved before the last process arrives, making *DATA* message from the last node travel faster up the tree. Meanwhile, the late arrival triggers

	Uniform				Normal			
Input Spread	Latency	Work	Output Spread	Max. Len.	Latency	Work	Output Spread	Max. Len.
0	171	3	43	9	171	3	43	9
256	151.8	15.3	43.4	70	141.1	14.8	49.5	93
512	150.0	23.0	43.2	71	140.4	21.5	51.6	83
1024	148.8	35.9	43.7	74	139.8	33.7	52.5	80
2048	147.0	62.0	44.0	71	139.0	58.9	52.9	78
4096	145.3	114.4	44.0	74	137.4	108.1	53.5	75
8192	144.0	219.2	44.2	76	139.9	208.8	53.5	81
16384	143.3	428.8	44.8	74	138.1	408.0	53.6	76

Table 2: Process Imbalance Effects on 64K Nodes

the active fault detection on waiting nodes, causing the per-node work to increase. Lastly, the increasing output spread is unusual for tree-based algorithms, and is the result of the multi-root topology: as the time between completions on different roots grows, so does the completion times between their sub-trees.

6 RELATED WORK

Algorithm-based fault tolerance (ABFT) refers to algorithms which include fault detection or recovery. For example, matrix computation algorithms could recover from faults by way of "hot-replacement" [19]. One common fault detection method is based on the communication timeouts, such as the logical ring topology proposed by Bosilca et al. [3], sending periodic keep-alive messages in parallel with application execution. Fault-aware MPI [5] (FA-MPI) represents another approach for applications to address faults by defining "transactions" which could be either committed or rolled-back in the case of a fault, comparable to ULFM.

ABFT for tree-based collective operations was presented by Graham et al. [10], proposing modifications to the tree topology to overcome node failure, similarly to this paper. They compare several design approaches for "fault-aware" collective operations, namely *rerouting*, *lookup avoiding* and *rebalancing*. Their work describes explicit MPI failure detection by calling *MPI_Comm_validate_rank* or *MPI_Comm_validate_all*, while our work combines the fault detection implicitly inside the collective operation algorithms. The approach taken to "repair" trees is similar to the *rerouting* alternative, and replaces the dependence on MPI with an internal fault detection model, but our algorithm changes the tree independently on each node detecting a fault.

Herault et al. [6] also demonstrate the replacement of nodes where a fault was detected, as part of an algorithm for scalable consensus. Their work targets the ULFM use case, specifically the agreement in MPI_Comm_Shrink serves as an example. While the method of using adjacent nodes for replacement is similar, our work shows how this method could be integrated into the collective operation, rather than being explicitly invoked to recover from an unrelated fault. Another transparent approach for fault tolerance has been presented by Kalim et al. [11]: SLIM introduces resilience on the transport level, so that some faults could be resolved without involving upper layers, including the application.

Multiple studies of MPI applications consider fault-tolerance, and some include the performance overhead of fault-tolerance in the overall evaluation. For example, Amatya et al. [1] evaluate the use in ULFM in several Deep Learning applications and analyze the impact of a single fault, including the reloading of data; the duration of communicator shrink and the way computation converges. Work by Rizzi et al. [18] presents a PDE solver application that uses ULFM by disregarding faults, as they show a small amount of faults has relatively little impact on the application result. This suggests there are various applications where FTCO could be applied.

Algorithms for collective operations are often evaluated by using the *LogP* model, such as the Barrier analysis by Hoefler et al. [8]. Tree-based algorithms could also be evaluated in other models, such as *Hockney*, or *PLogP*, as presented by Pješivac-Grbović et al. [17]. Process imbalance and its effect on collective operation algorithms was shown by Marendic et al. [12] and Parsons et al. [15], along with proposals to mitigate its impact.

Hoefler et al. [7] presents an alternative in the form of gossip algorithms, under the same formal model as paper employs, for a reliable *Broadcast* collective operation. Their formal bound, coupled with simulator results, show that combining randomness (gossip phase) and determinism (correction phase) can yield a significant improvement over other algorithms in latency and message count.

7 CONCLUSIONS AND FUTURE WORK

Collective operations are a key component of the MPI standard and most MPI applications, but fault-tolerance remains an open issue. The increase in the number of faults and the implied recovery costs in large systems suggest that parallel applications should incorporate means for fault-tolerance to run at scale. This work presented fault-tolerant collective operations, a paradigm where the faults are detected and handled as part of the collective operation. FTCO is suitable for applications that that can tolerate process faults and partial results.

The paradigm presented in this work could be extended to other existing algorithms for collective operations, such as "recursive doubling". It would also be interesting to measure the performance of parallel applications that use FTCO on large-scale systems.

Finally, an FTCO library would enable research into fault-tolerance of higher-level algorithms: how system faults effect the performance and accuracy of applications such as numeric solvers. Furthermore, applications could define custom fault-recovery methods with minimal changes, such as replacing missing processes with some predefined default value or some extrapolation based on previous data, thus benefiting applications in a low fault-rate system.

8 ACKNOWLEDGMENTS

This research was supported in part by grants from Dr. and Mrs. Silverston, Cambridge, the UK; and from the DFG, German Priority Programme 1648 "Software for Exascale Computing" (SPPEXA), project FFMK.

REFERENCES

[1] V. Amatya, A. Vishnu, C. Siegel, and J. Daily. 2017. What Does Fault Tolerant Deep Learning Need from MPI?. In *Proc. European MPI Users' Group Meeting (EuroMPI)*. 13:1–13:11.

[2] W. Bland, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra. 2012. *A Proposal for User-Level Failure Mitigation in the MPI-3 Standard*. Technical Report.

[3] G. Bosilca, A. Bouteiller, A. Guermouche, T. Herault, Y. Robert, P. Sens, and J. Dongarra. 2016. Failure Detection and Propagation in HPC Systems. In *Proc. Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. 27:1–27:11.

[4] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. 1993. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. Sym. on Principles and Practice of Parallel Programming (PPoPP)*. 1–12.

[5] A. Hassani, A. Skjellum, R. Brightwell, and P.V. Bangalore. 2014. Comparing, Contrasting, Generalizing, and Integrating Two Current Designs for Fault-Tolerant MPI. In *Proc. European MPI Users' Group Meeting*. 63:63–63:68.

[6] T. Herault, A. Bouteiller, G. Bosilca, M. Gamell, K. Teranishi, M. Parashar, and J. Dongarra. 2015. Practical Scalable Consensus for Pseudo-synchronous Distributed Systems. In *Proc. Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. 31 1–31 12.

[7] T. Hoefler, A. Barak, A. Shiloh, and Z. Drezner. 2017. Corrected Gossip Algorithms for Fast Reliable Broadcast on Unreliable Systems. In *Proc. Intl. Parallel and Distributed Processing Sym. (IPDPS)*. 357–366.

[8] T. Hoefler, L. Cerquetti, T. Mehlan, F. Mietke, and W. Rehm. 2005. A practical approach to the rating of barrier algorithms using the LogP model and Open MPI. In *Proc. Intl. Conf. on Parallel Processing Workshops (ICPPW)*. 562–569.

[9] T. Hoefler and W. Rehm. 2005. A Communication Model for Small Messages with InfiniBand. 32–41.

[10] J. Hursey and R. L. Graham. 2011. Preserving Collective Performance across Process Failure for a Fault Tolerant MPI. In *Proc. Intl. Parallel and Distributed Processing Sym. (IPDPS)*. 1208–1215.

[11] U. Kalim, M.K. Gardner, and W. Feng. 2017. A Non-Invasive Approach for Realizing Resilience in MPI. In *Proc. Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*. 1–8.

[12] P. Marendic, J. Lemeire, D. Vucinic, and P. Schelkens. 2016. A novel MPI reduction algorithm resilient to imbalances in process arrival times. *The Journal of Supercomputing* 72, 5 (2016), 1973–2013.

[13] A. Moody, J. Fernandez, F. Petrini, and D.K. Panda. 2003. Scalable NIC-based Reduction on Large-scale Clusters. In *Proc. Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. 59–59.

[14] N. Papadopoulou, L. Oden, and P. Balaji. 2017. A Performance Study of UCX over InfiniBand. In *Proc. Intl. Sym. on Cluster and Cloud and Grid Computing (CCGRID)*. 345–354.

[15] B.S. Parsons and V.S. Pai. 2015. Exploiting Process Imbalance to Improve MPI Collective Operations in Hierarchical Systems. In *Proc. Intl. Conf. on Supercomputing (ICS)*. 57–66.

[16] I.B. Peng, S. Markidis, and E. Laure. 2015. The Cost of Synchronizing Imbalanced Processes in Message Passing Systems. In *IEEE Intl. Conf. on Cluster Computing*. 408–417.

[17] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G.E. Fagg, E. Gabriel, and J. Dongarra. 2007. Performance analysis of MPI collective operations. *Cluster Computing* 10, 2 (2007), 127–143.

[18] F. Rizzi, K. Morris, K. Sargsyan, P. Mycek, C. Safta, B.J. Debusschere, O.P. Le Maître, and O.M. Knio. 2016. ULFM-MPI Implementation of a Resilient Task-Based Partial Differential Equations Preconditioner. In *Proc. Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*.

[19] R. Wang, E. Yao, M. Chen, G. Tan, P. Balaji, and D. Buntinas. 2011. Building algorithmically nonstop fault tolerant MPI programs. In *Proc. Intl. Conf. on High Performance Computing*. 1–9.

APPENDIX

A SERVICE-DISTANCE FUNCTIONS

To complete the description of the algorithm, we first define the *"service-distance function"* used in Algorithm 1 line 6:

Definition A.1. For each step of the algorithm, the *Service-distance function* $f : \mathbb{N} \to \mathbb{N}$ determines the distance of the nodes to be handled.

The selection of the service-distance function determines the message timeout function $f_{timeout}$. We examined several alternatives and found that a reasonable choice is to select an $f_{timeout}$ proportional to the distance between the source and the destination nodes, as follows:

Definition A.2. A *Nepotistic service-distance function* is a service-distance function $f(i)$ that satisfies the following two conditions:

(1) No starvation (every distance repeats infinitely), so that $Pr(f(i) = d \mid i > step) > 0 \quad \forall step, d \in \mathbb{N}$;
(2) Nepotism (favor closer nodes):
$\forall step, d_1, d_2 \in \mathbb{N} : (d_1 < d_2) \Rightarrow Pr(f(i) = d_1 \mid i > step) \geq Pr(f(i) = d_2 \mid i > step)$

We propose two such functions: one is randomized and simple to compute, but for formal validation of correctness we also define a deterministic alternative. The *Randomized Service-Distance (RSD)* function based on the uniform distribution: $f_{RSD}(x) = \lceil \log_r(Y) \rceil$, where $Y \sim U\{0, 1\}$ and $0 < r \leq \frac{1}{2}$. The Deterministic Intervals Service-Distance (DISD) function f_{DISD} consists of intervals with a fixed number of steps, allocated in a repeating pattern: half of each interval is allocated to handle messages from nodes of distance 1; quarter to nodes of distance 2; eighth to nodes of distance 3 and so on. We set the number of steps in each interval to be four times the number of nodes of distance 1, so that the *DATA* and *ACK* from each node of distance 1 are handled within one interval. We note that for our service-distance function, the timeout $f_{timeout}$ is proportional to the degree of destination node and the distance between the source and destination nodes. Simulator results show negligible difference in latency between the two functions.

B PROCESS IMBALANCE

We assume a node with a delayed start time has some limited functionality, to distinguish node failure from late arrival due to process imbalance. Before entering the collective operation a node is assumed to respond with *ACK* to *DATA* and *KEEPALIVE*, retain arriving *DATA*, but not send any *DATA*. In order to maintain algorithm correctness with respect to timeouts determining node failure, when multiple messages arrive at once, such a node must respond with the same ratio as if it had arrived at the collective operation. The assumptions on incoming message queue allow for it to store incoming data messages until arrival time. Algorithm 5 demonstrated the functionality when $0 \leq t < StartTime$, followed by the start of Algorithm 1 (Main part).

Algorithm 5: Collective Operation Imbalance

```
1 void pre_collective_functionality(initial_value)
2     while now < StartTime do
3         service_distance ← get_service_distance();
4         if ¬ handle_message_queue(service_distance)
5           ∧¬ fault_tolerance(service_distance)
6           ∧¬ handle_message_queue(0) then
7             fault_tolerance(0);
8     return fault_tolerant_collective(initial_value);
```

C SIMULATOR REPRODUCIBILITY

C.1 Description

This section contains information about the simulator for the formal results. It was used to obtain the results for large numbers of nodes and faults, including process imbalance, in Section 5.2. The simulator uses MPI only to launch multiple randomized tests and aggregate the results - each test simulates discrete steps for a large number of nodes within a single process.

C.1.1 *Simulator Check-list.*

- **Algorithm:** FTCO
- **Program:** Simulator application in C
- **Compilation:** C compiler
- **Run-time environment:** OpenMPI
- **Execution:** MPI job with a subset of input parameters
- **Output:** CSV file, listing results by input parameters
- **Experiment workflow:** Build the code, run the simulation with different input parameters
- **Publicly available?:** Yes, on github.

C.1.2 Software dependencies. Implementation of MPI v1.0 standard or above (only simple collectives are used).

C.2 Experiment workflow

Once the simulator has been built, we ran it with increasing node counts and different sets of parameters. The simulator supports control of the following variables: L/o ratio (in steps), tree topology and radix (N-ary or k-nomial, single-root or multi-root), number of nodes and faults (inactive nodes or offline faults), process imbalance distribution and the number of test iterations to run (for randomized fault configurations). For each such test, the result include the statistics on the following metrics: latency (until the last node finishes, in steps), input and output spread (process imbalance of the collective operation start and finish), the number of messages and the longest inbound message queue length on any node during the test.

C.3 Evaluation and expected result

To validate the simulator results, we compare it to known bounds for some tree topologies. Knowing the number of nodes, the tree radix and the latency (in steps) between each two nodes, it is possible to calculate the expected result from the simulator. The paper also presents the rationale for the results of randomized tests.

D MPI FTCO REPRODUCIBILITY

D.1 Description

This section contains information about the OpenMPI extension, used for the experimental results. It was used to obtain the results for the comparison with ULFM in Section 5.1. The extension includes these main changes:

- Change tree topology calculation to include "grand-parent" nodes
- Change ULFM keep-alive mechanism to contact peer processes only upon request, rather than constantly contacting some pre-calculated peer
- Change *MPI_Reduce* and *MPI_Allreduce* to fall-back to "grand-parent" nodes when failure was detected by ULFM on the parent node
- Add support for a "temporary fault" in a process, to allow multiple test iterations within the same MPI job

We also modified an existing ULFM benchmark, called *benchrevoke*, to time *MPI_Reduce*/*MPI_Allreduce* in the presence of faults in any configuration (named it *benchtolerate*).

D.1.1 *OpenMPI Extension Check-list.*

- **Algorithm:** FTCO vs. ULFM
- **Program:** *benchtolerate (see description)*
- **Compilation:** *OpenMPI's C/C++ compilers*
- **Run-time environment:** *OpenMPI v4.1.0*
- **Hardware:** *32 machines, Dual-socket Octa-core Intel Xeon E5-2650 v2 at 2.6GHz*
- **Execution:** *MPI job with different fault configurations*
- **Output:** *Latency measurements for multiple iterations*
- **Experiment workflow:** *Build OpenMPI with the extension, then compare runs with the extension (FTCO) and without it (ULFM).*
- **Publicly available?:** *Yes, implementation and benchmarks.*

D.1.2 Software dependencies. A customized OpenMPI, cloned from the ULFM master branch (v2.0), along with our modifications, was used for measuring the latency of the collective operations.

D.2 Experiment workflow

For several fault configurations, we compared the latency of collective operations on multiple communicators cloned from *MPI_COMM_WORLD*. The *MPI_Allreduce*, with either FTCO or the original ULFM implementation, used a separate thread (which improves fault detection latency), TCP communication and a binary tree topology. This is the only topology shared between the generic Reduce and Broadcast implementations, and for *MPI_Reduce* the extension supports the other topologies as well. Example command-line: *mpirun –bind-to core -mca mpi_ft_detector_thread true -mca mpi_ft_detect_local true -mca mpi_ft_detector_rdma_heartbeat false -mca coll_tuned_use_dynamic_rules true -mca coll_tuned_allreduce_algorithm 2 -mca coll_tuned_reduce_algorithm 4 -mca coll_tuned_bcast_algorithm 5 -am ft-enable-mpi ./benchtolerate*