Resilient applications using MPI-level constructs

MPI Forum March 2-5, 2015

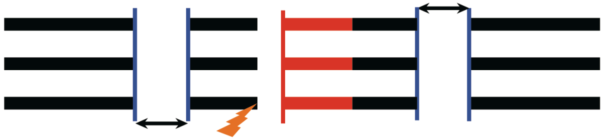
Aurélien Bouteiller

ICL UT
INNOVATIVE
COMPUTING LABORATORY
THE UNIVERSITY of TENNESSEE

An API for diverse FT approaches

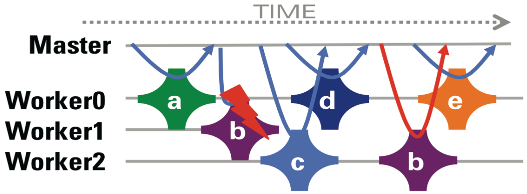
Coordinated Checkpoint/Restart, Automatic, Compiler Assisted, User-driven Checkpointing, etc.

In-place restart (i.e., without disposing of non-failed processes) accelerates recovery, permits in-memory checkpoint



Naturally Fault Tolerant Applications, Master-Worker, Domain Decomposition, etc.

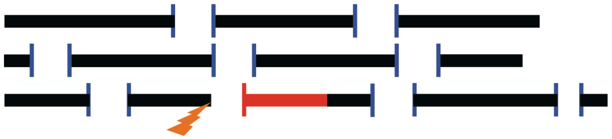
Application continues a simple communication pattern, ignoring failures



ULFM MPI Specification

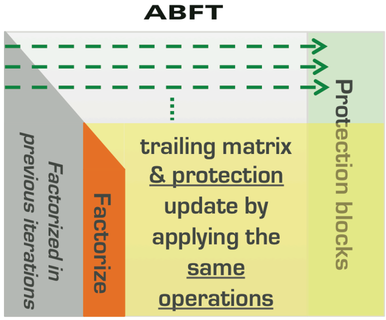
Uncoordinated Checkpoint/Restart, Transactional FT, Migration, Replication, etc.

ULFM makes these approaches portable across MPI implementations



Algorithm Fault Tolerance

ULFM allows for the deployment of ultra-scalable, algorithm specific FT techniques.

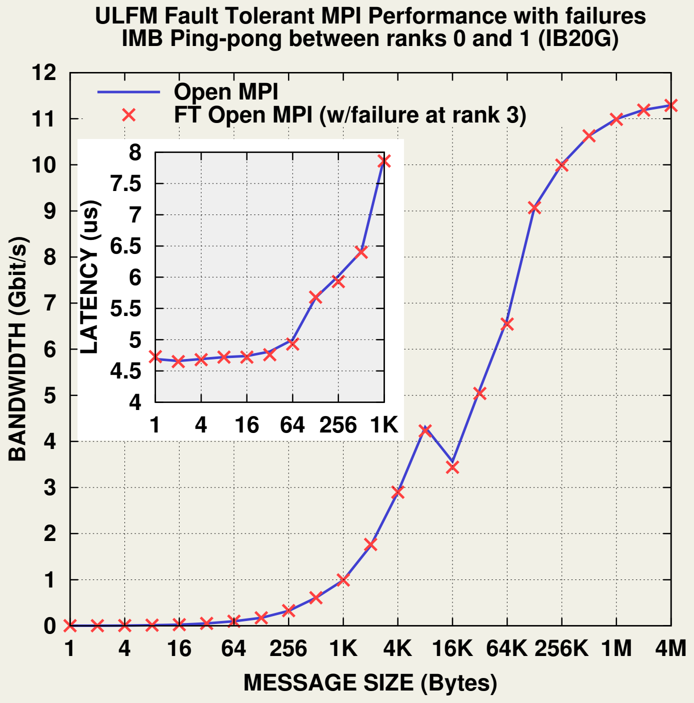

ABFT
Factorized in previous iterations
Factorize
trailing matrix & protection update by applying the same operations
Protection blocks

User Level Failure Mitigation: a set of MPI interface extensions to enable MPI programs to restore MPI communication capabilities disabled by failures

ULFM MPI: Software Infrastructure

- Implementation in Open MPI available
  - ANL working on MPICH implementation, close to release
- Very good performance w/o failures
- Optimization and performance improvements of critical recovery routines are close to release
  - New revoke
  - New Agreement

Performance w/failures



The failure of rank 3 is detected and managed by rank 2 during the 512 bytes message test. The connectivity and bandwidth between rank 0 and rank 1 are unaffected by failure handling activities at rank 2.

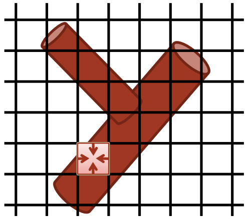
Applications

HemeLB

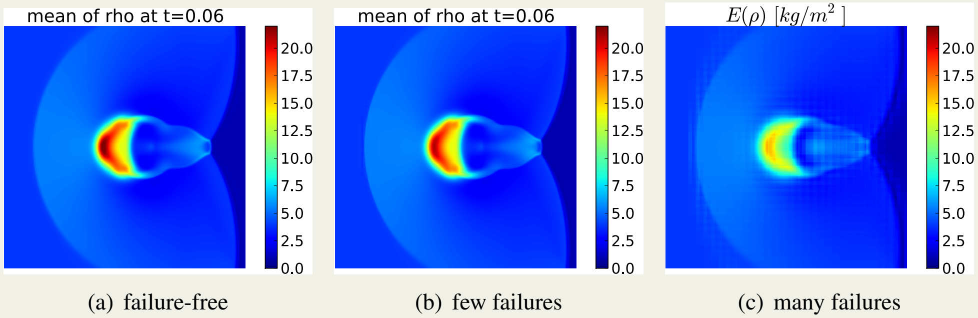
Lattice Boltzmann Flow Solver
University College London

Processor fails
- Re-initialize substitute processor with average mass flow, velocity from neighbors
  passable error in domain size and magnitude if real solution sufficiently smooth

Long running computations
- Small errors can be eliminated by numerical procedure



Credits: ETH Zurich



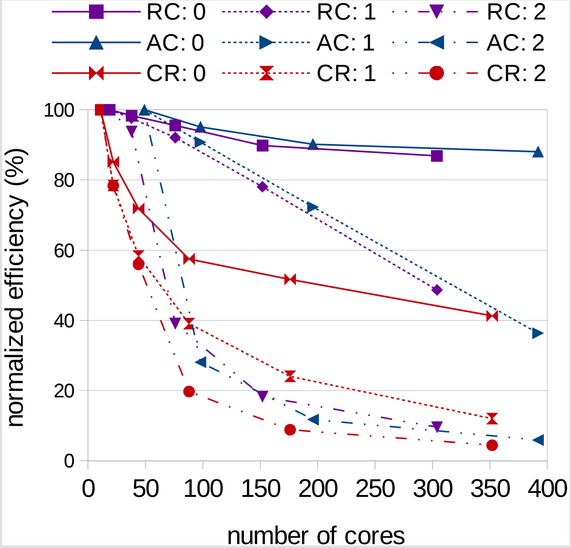
(a) failure-free (b) few failures (c) many failures

Figure 5. Results of the FT-MLMC implementation for three different failure scenarios.

- **ORNL:** Molecular Dynamic simulation, C/R in memory with Shrink
- **UAB:** transactional FT programming model
- **Tsukuba:** Phalanx Master-worker framework
- **Georgia University:** Wang Landau Polymer Freezing and Collapse, localized subdomain C/R restart
- **Sandia, INRIA, Cray:** PDE sparse solver
- **Cray:** CREST miniapps, PDE solver Schwartz, PPStee (Mesh, automotive), HemeLB (Lattice Boltzmann)
- **ETH Zurich:** Monte-Carlo, on failure the global communicator (that contains spares) is shrunk, ranks reordered to recreate the same domain decomposition
- …

Tens of papers using ULFM last year alone.

12 Results: Scalability



- results on OPL cluster, max. resolution of 2^{13}
- in terms of absolute time, CR is always more longer (however, uses fewer processes)
- RC and AC also show best scalability
- plots for 2 failures erratic due to high overheads in β version of ULFM MPI

OPL cluster node: 2x6 cores Xeon5670, QDR IB

RC=Replication/resampling
AC=Alternate recombination
CR=Checkpoint/Restart

Part rationale, part examples

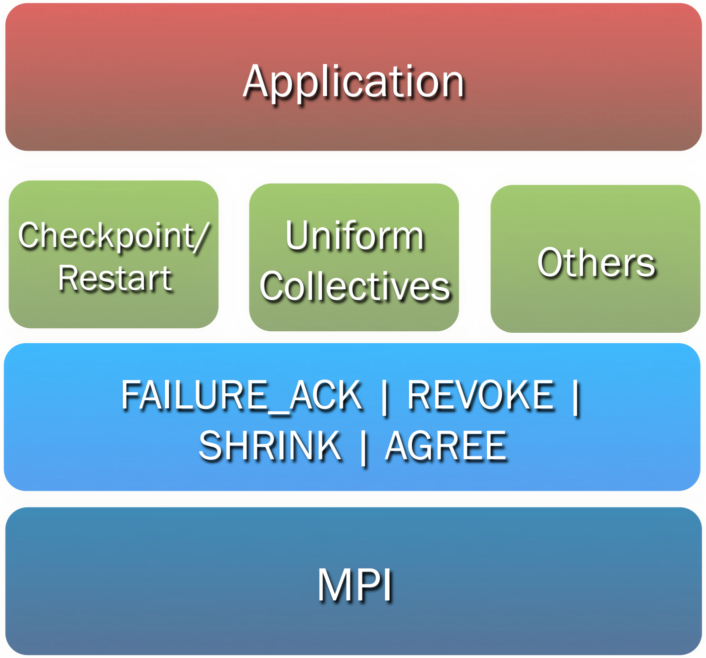
ULFM MPI API

Summary of existing functions

- MPI_Comm_create_errhandler(errh, errhandler_fct)
  - Declare an error handler with the MPI library
- MPI_Comm_set_errhandler(comm, errh)
  - Attach a declared error handler to a communicator
  - Newly created communicators inherits the error handler that is associated with their parent
  - Predefined error handlers:
    - MPI_ERRORS_ARE_FATAL (default)
    - MPI_ERRORS_RETURN

Minimal Feature Set for FT MPI

- Failure Notification
- Error Propagation
- Error Recovery



Not all recovery strategies require all of these features, that's why the interface splits notification, propagation and recovery.

ULFM is not a recovery strategy, but a minimalistic set of building blocks for more complex recovery strategies.

Integration with existing mechanisms

- New error codes to deal with failures
  - MPI_ERROR_PROC_FAILED: report that the operation discovered a newly dead process. Returned from all blocking function, and all completion functions.
  - MPI_ERROR_PROC_FAILED_PENDING: report that a non-blocking MPI_ANY_SOURCE potential sender has been discovered dead.
  - MPI_ERROR_REVOKED: a communicator has been declared improper for further communications. All future communications on this communicator will raise the same error code, with the exception of a handful of recovery functions
- Is that all?
  - Matching order (MPI_ANY_SOURCE), collective communications

Summary of new functions

- MPI_Comm_failure_ack(comm)
  - Resumes matching for MPI_ANY_SOURCE
- MPI_Comm_failure_get_acked(comm, &group)
  - Returns to the user the group of processes acknowledged to have failed

Notification

- MPI_Comm_revoke(comm)
  - Non-collective, interrupts all operations on comm (future or active, at all ranks) by raising MPI_ERR_REVOKED

Propagation

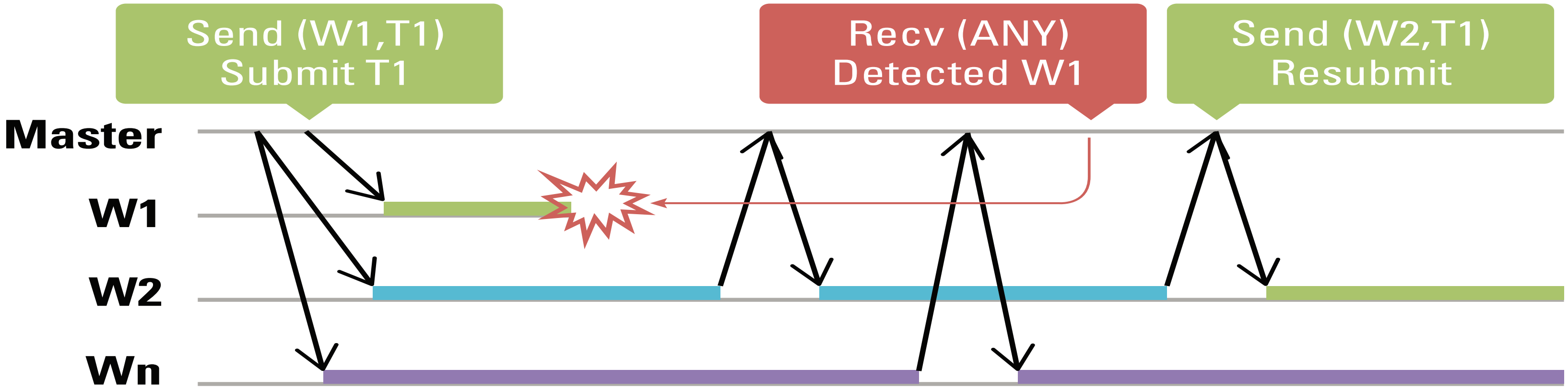
- MPI_Comm_shrink(comm, &newcomm)
  - Collective, creates a new communicator without failed processes (identical at all ranks)
- MPI_Comm_agree(comm, &mask)
  - Agree on the AND value on binary mask, ignoring failed processes (reliable AllReduce)

Recovery

Failure Discovery

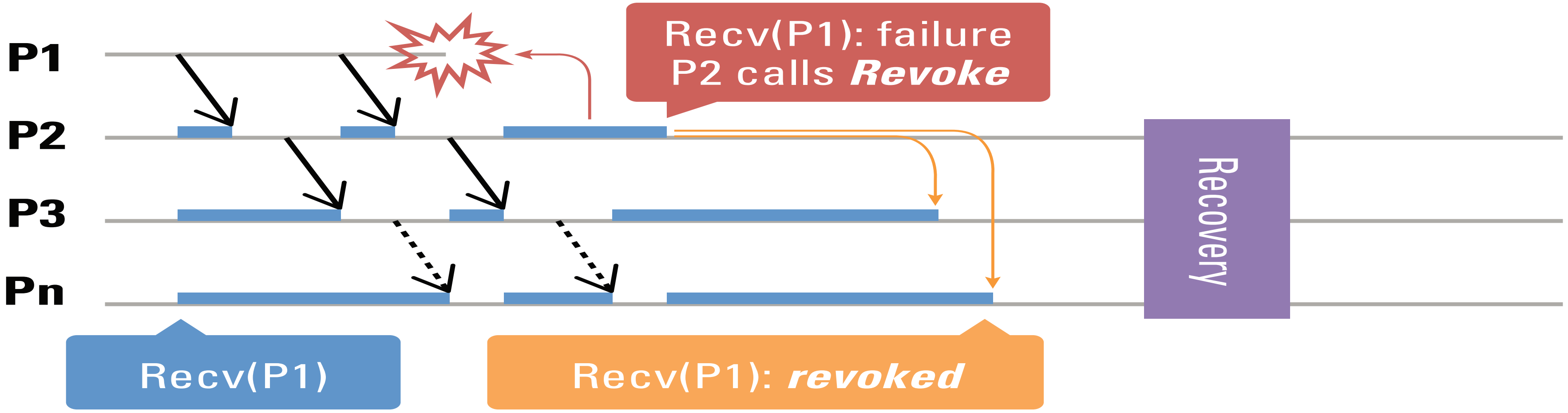
- Discovery of failures is *local* (different processes may know of different failures)
- MPI_COMM_FAILURE_ACK(comm)
  - This local operation gives the users a way to acknowledge all locally notified failures on comm. After the call, unmatched MPI_ANY_SOURCE receive operations proceed without further raising MPI_ERR_PROC_FAILED_PENDING due to those acknowledged failures.
- MPI_COMM_FAILURE_GET_ACKED(comm, &grp)
  - This local operation returns the group *grp* of processes, from the communicator comm, that have been locally acknowledged as failed by preceding calls to MPI_COMM_FAILURE_ACK.
- Employing the combination ack/get_acked, a process can obtain the list of all failed ranks (as seen from its local perspective)

Continuing through errors



- Error notifications do not break MPI
  - App can continue to communicate on the communicator
  - More errors may be raised if the op cannot complete (typically, most collective ops are expected to fail), but p2p between non-failed processes works
- In this Master-Worker example, we can continue w/o recovery!
  - Master sees a worker failed
  - Resubmit the lost work unit onto another worker
  - Quietly continue

Resolving transitive dependencies



```
proc_failed_err_handler(MPI_Comm comm, int err) {
  if(err == MPI_ERR_PROC_FAILED ||
     err == MPI_ERR_REVOKED ) {
    MPI_Comm_revoke(comm);
    recovery(comm);
  }
}
ft_transitive_deps(void) {
  for(i=0; i<nbrecv; i++) {
    if(myrank>0) MPI_Irecv(buff, count, datatype,
                      myrank-1, tag, comm, &req);
    if(myrank<n) MPI_Send(buff2, count, datatype,
                      myrank+1, tag, comm, &req);
  }
}
```

- P1 fails
- P2 raises an error and wants to change comm pattern to do application recovery
- but P3..Pn are stuck in their posted recv
- P2 can unlock them with **Revoke** ☺
- P3..Pn join P2 in the recovery

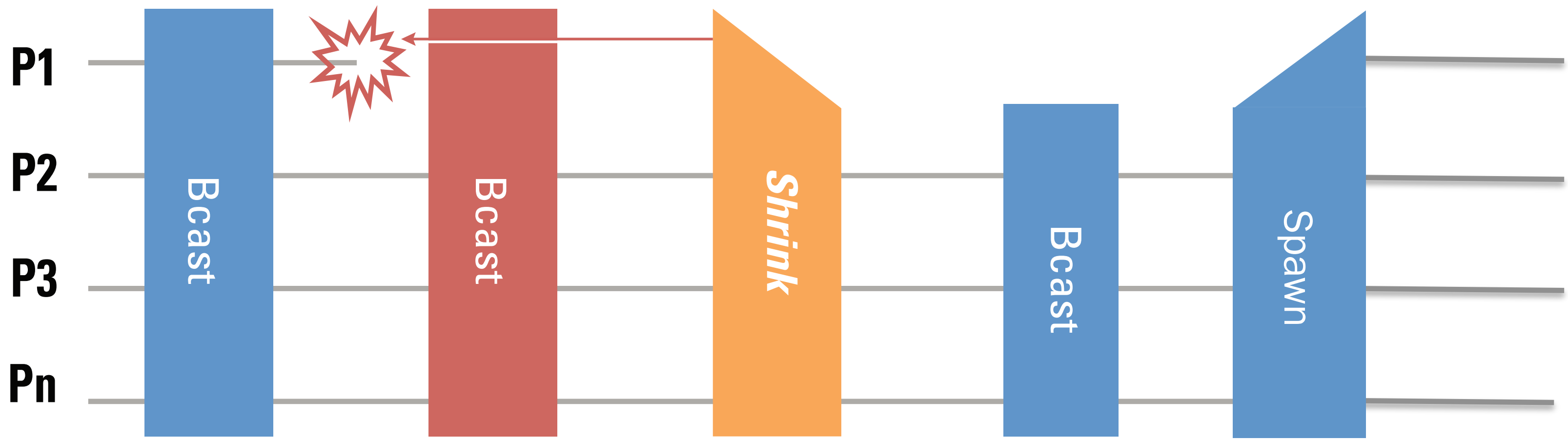
Errors and Collective Operations

```
proc_failed_err_handler(MPI_Comm comm, int err) {
  if(err == MPI_ERR_PROC_FAILED ||
     err == MPI_ERR_REVOKED ) recovery(comm);
}

deadlocking_collectives(void) {
  for(i=0; i<nbrecv; i++) {
     MPI_Bcast(buff, count, datatype, 0, comm);
  }
}
```

- Exceptions are raised only at ranks where the Bcast couldn’t succeed (lax consistency)
  - In a tree-based Bcast, only the subtree under the failed process sees the failure
  - Other ranks succeed and proceed to the next Bcast
  - Ranks that couldn’t complete enter “recovery”, do not match the Bcast posted at other ranks => MPI_Comm_revoke(comm) interrupts unmatched Bcast and forces an exception (and triggers recovery) at all ranks

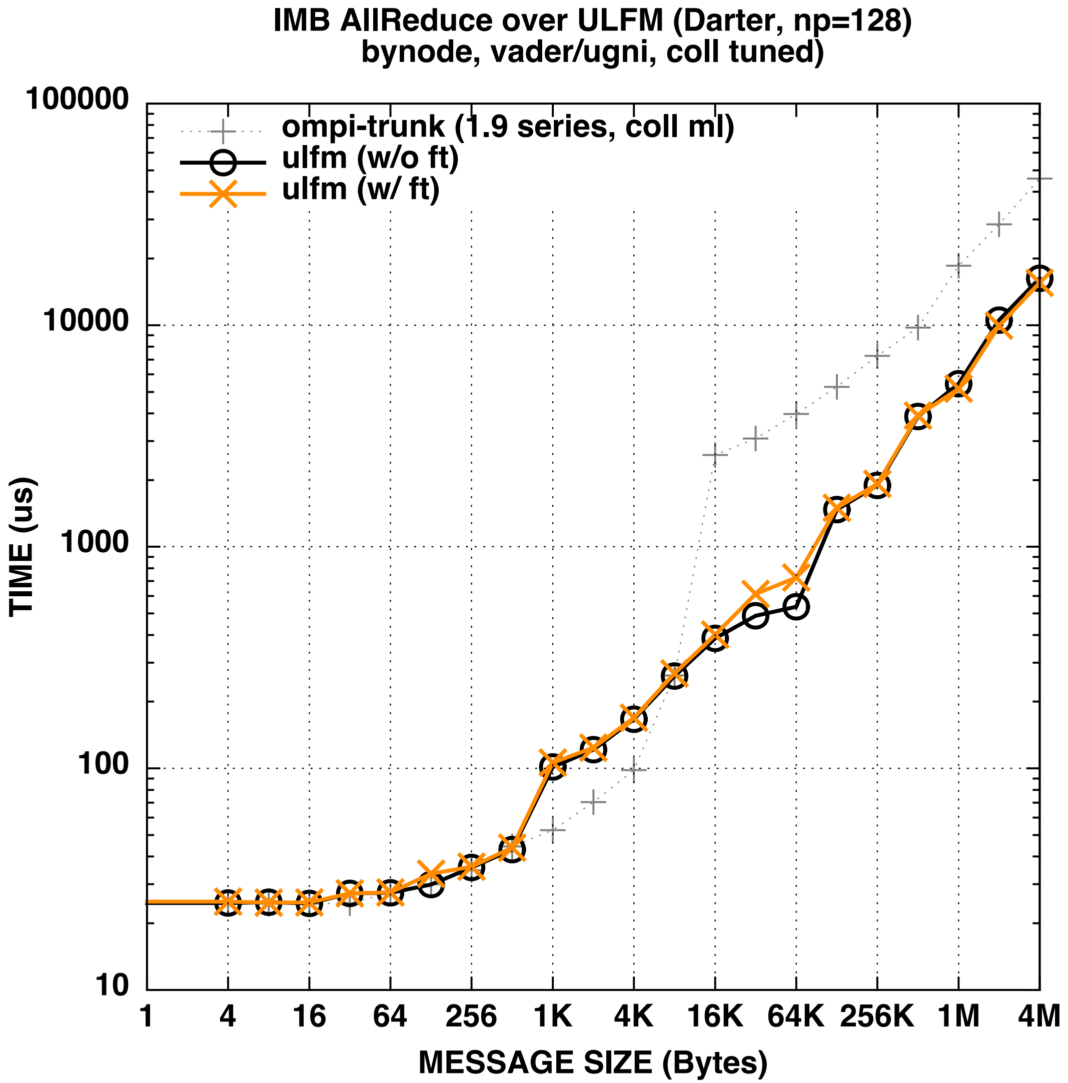
Full Recovery



- Restores full communication capability (all collective ops, etc).
- MPI_COMM_SHRINK(comm, newcomm)
  - Creates a new communicator excluding failed processes
  - New failures are absorbed during the operation
  - The communicator can be restored to full size with MPI_COMM_SPAWN

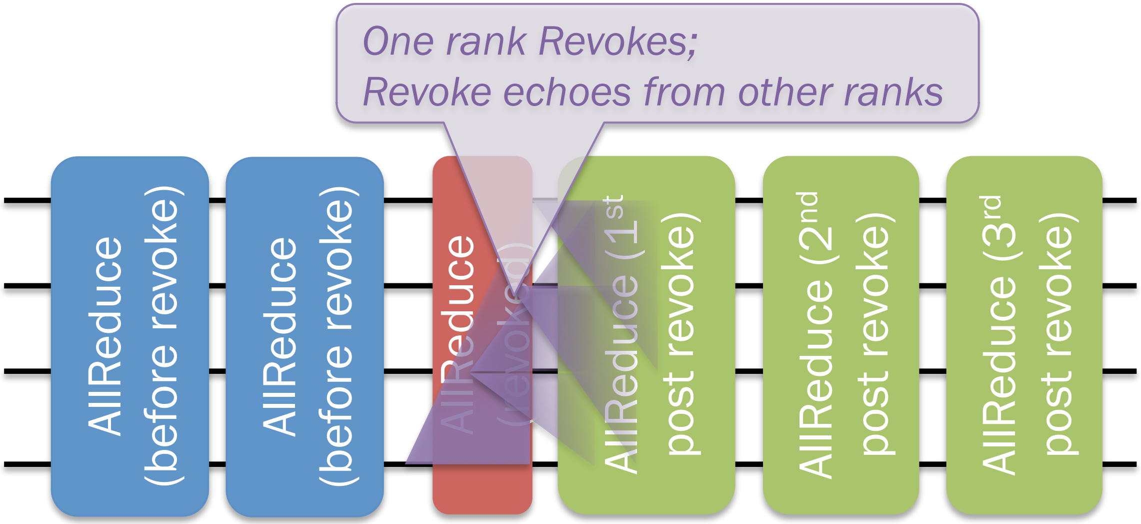
OPEN MPI IMPLEMENTATION UPDATE

Collective and p2p



- Systematic verification of correct behavior for "tuned" collective module under failure cases
- Backport of latest trunk "tuned" collective component completed
- Backport of latest "basic" collective module in progress
- Vader BTL (shared memory transport) from trunk importation: in progress (main benefit is better support for xpmem)

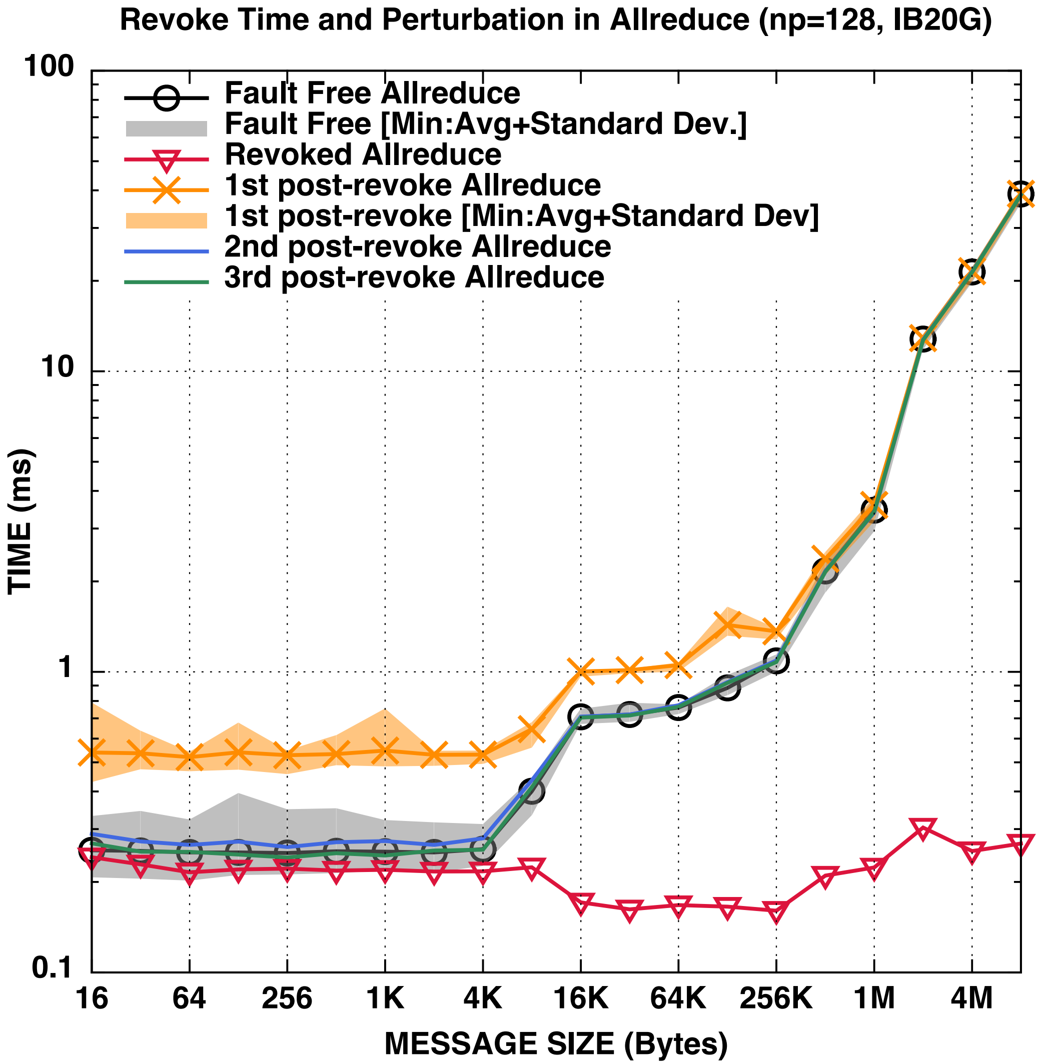
Evaluating Revoke Cost


One rank Revokes;
Revoke echoes from other ranks
AllReduce (before revoke)
AllReduce (before revoke)
AllReduce (revoked)
AllReduce (1st post revoke)
AllReduce (2nd post revoke)
AllReduce (3rd post revoke)

1. The cost of Revoke at the initial caller is essentially 0 (immediate operation, completes in the background)
2. But, even after a Revoke has delivered to all ranks, the "revoke tokens" are still circulating on the network

- Two duplicate of MPI_COMM_WORLD: blue, green
- On the blue communicator:
  - Repeat allreduce (measure baseline time)
  - At some iteration, one rank revokes the blue communicator
  - Measure the time it takes for the last allreduce to be revoked at all ranks
- Immediately after, on the green communicator
  - Repeat allreduce (this comm is not revoked, no deads, so everything works w/o errors)
  - Measure the time it takes for the first, second, third, allreduce, until the noise generated by background revoke cannot be observed

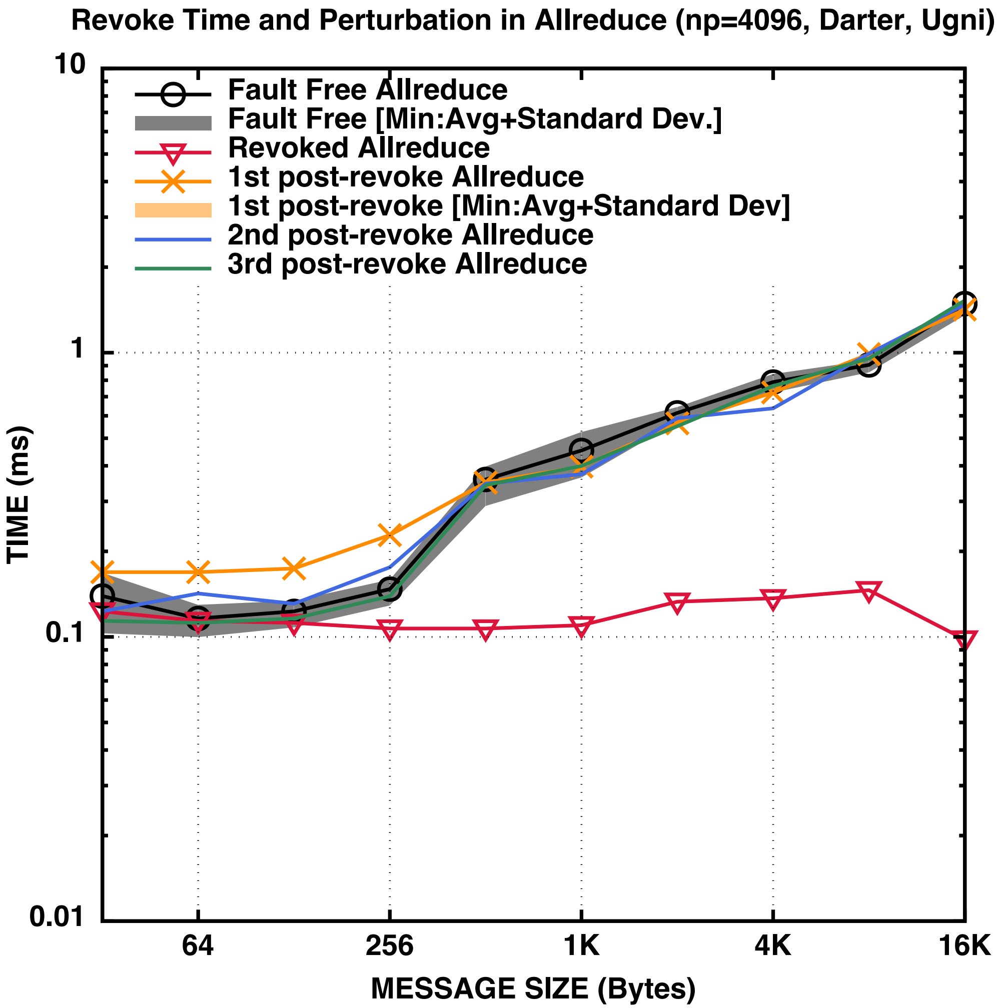
Cost of Revoke



- Propagation time for Revoke messages ~= small message allreduce latency
- After the revoke has propagated, noise continue for another small message allreduce latency
- Only the first allreduce is impacted

ICL UT

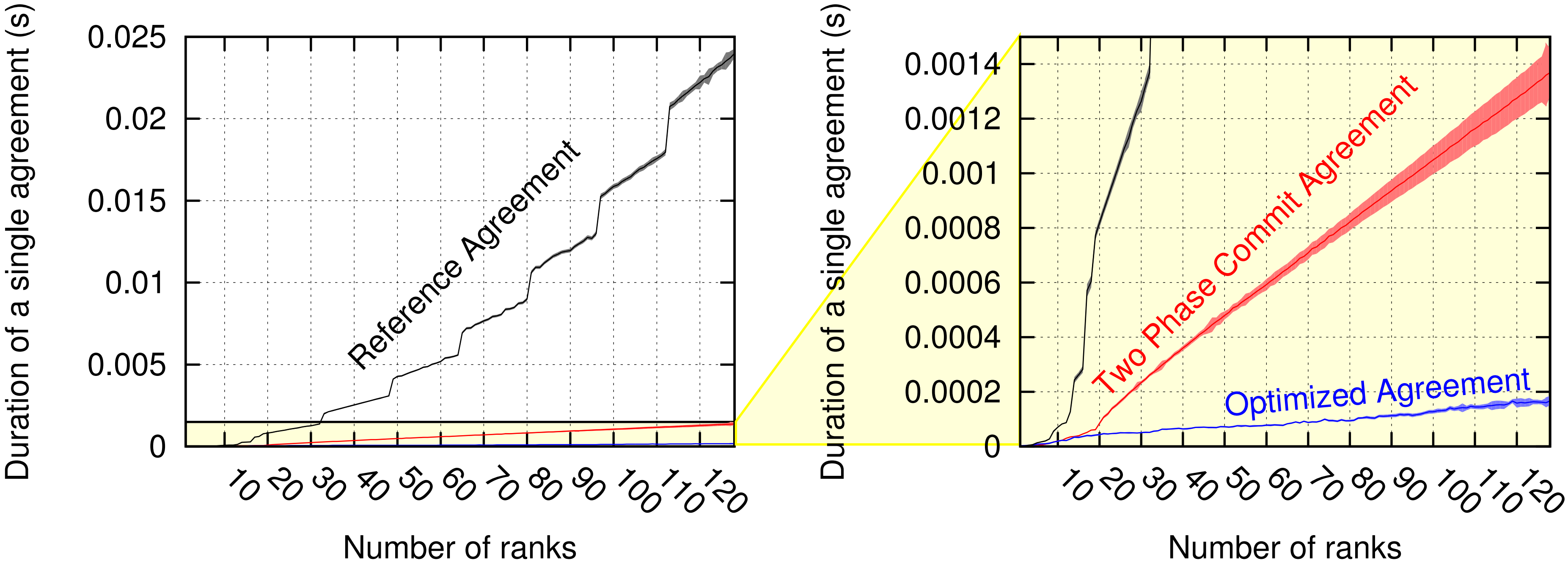
Cost of Revoke (Darter, 4k cores)



Same story at scale on Darter:

the noise of the agreement is invisible for as small as 512 msg size.

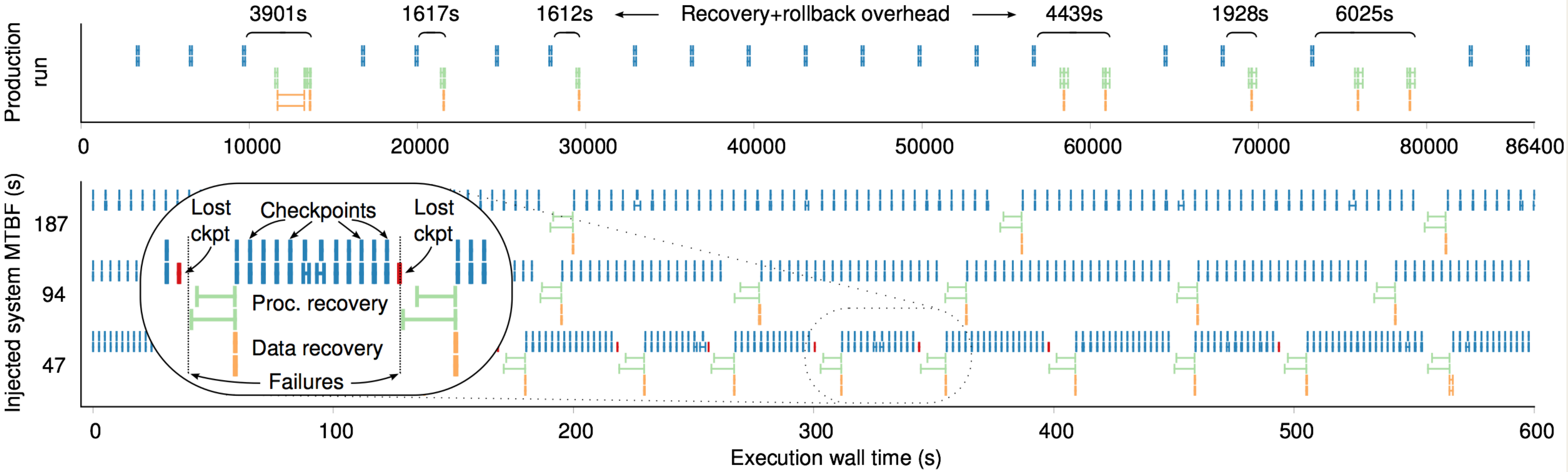
New Agreement



New Agreement with logarithmic complexity

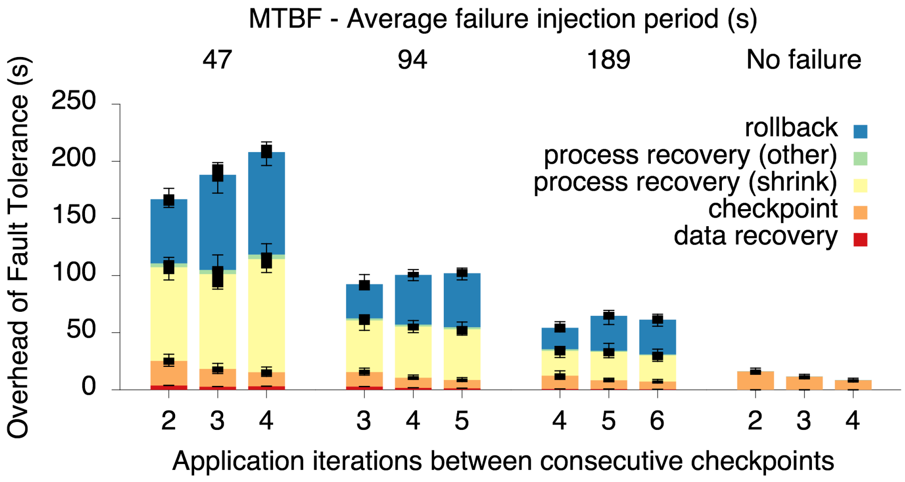
Will resolve previously reported bad performance at scale in MPI_Comm_shrink ☺

4. Recovering from **high-frequency failures**

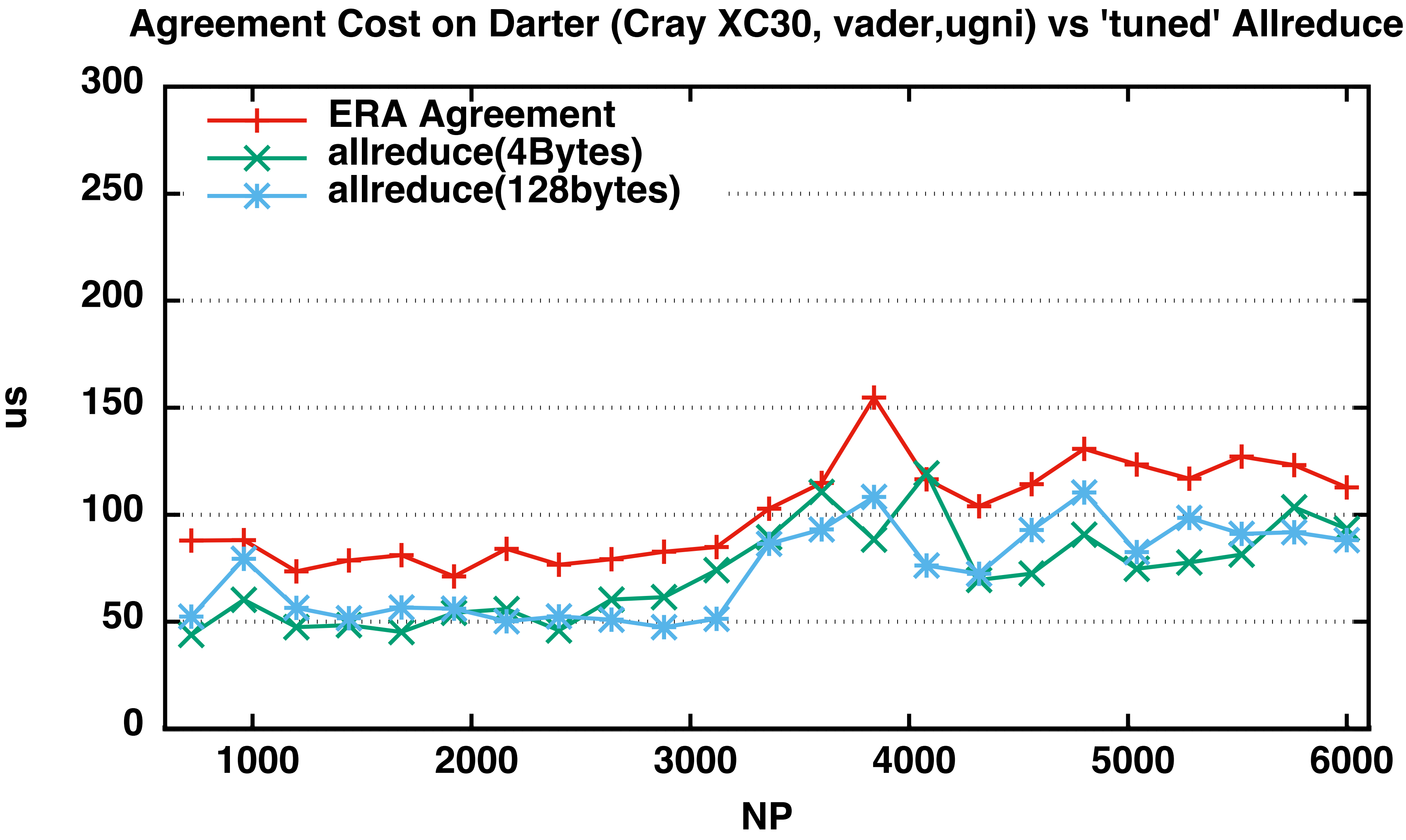


Conclusions:

- Online recovery allows the usage of in-memory checkpointing, O(0.1s)
- Efficient recovery from high frequency node failures, as exascale compels
- **With failures injected every 189, 94 and 47 seconds,** the total job run-time penalty is **10%, 15% and 31%, respectively**
  - *Note that current production runs' fault tolerance cost is 31%!*
- This can dramatically improve by optimizing ULFM shrink



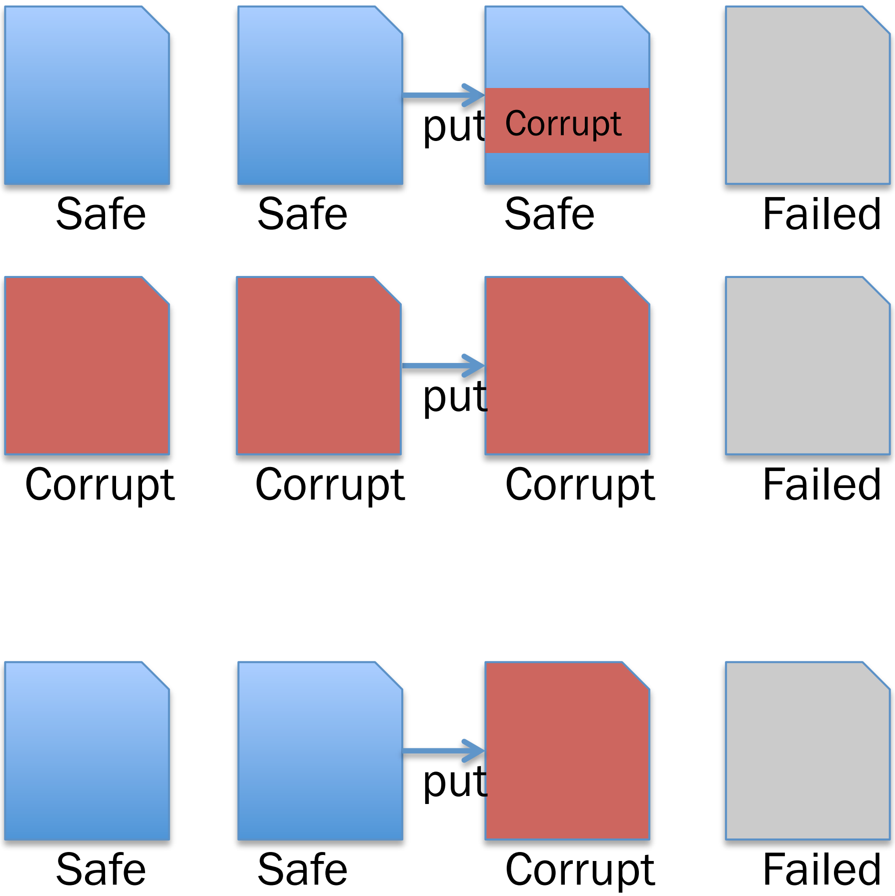
Agreement performance at scale



FT RMA ONGOING WORK AND DIRECTIONS

Post failure RMA Semantic

- March 2014:
  - A failure during an RMA operation damages only the memory specifically targeted by remote write operations
  - Considered too strong consistency, possibly hard to implement with hypothetical pathological hardware
- September/December 2014
  - A failure during an RMA operation damages all memory exposed by the window at all ranks
  - Considered too hard to used, limited usefulness
- New proposition (thanks Jeff Hammond)
  - A failure during an RMA operation damages all memory exposed by the window at a rank that has been the target of a remote write (since the last successful “epoch changing” operation
  - The goal is that data exposed through RMA do not become corrupt even when we do “double buffering” or similar techniques where the data/ checkpoint is “read only” during the epoch.

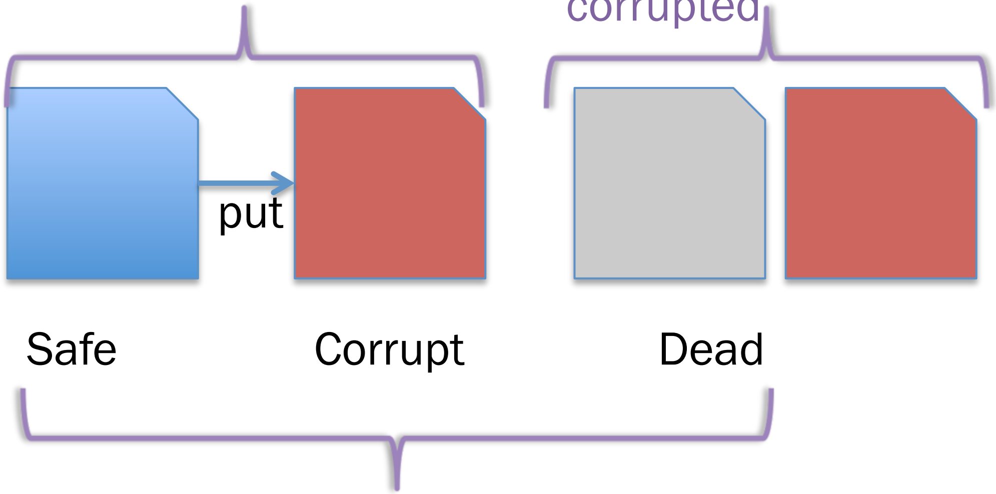


Safe Safe put Corrupt Safe Failed

Corrupt Corrupt put Corrupt Failed

Safe Safe put Corrupt Failed

RMA consistency, side cases



W1: no dead process, memory corrupted through exposure in W2

W3: true shared memory: no put, but true sharing with a dead => memory corrupted

put

Safe

Corrupt

Dead

W2: contains a dead process, put corrupts the target

- Memory exposed by multiple windows become corrupted in all windows where it is exposed (even though no errors may be raised in this window)
- True shared memory can become corrupted at ranks that have not failed in the window
  - It should remain “accessible/ addressable”, it is implementation’s business to make it so.

Tentative text

- \par When an operation on a window raises an exception related to
- -process failure, the state of all data held in memory exposed
- -by that window becomes undefined at all ranks.
- +process failure, the state of all data held in memory exposed by that
- +window becomes undefined at all ranks for which a one-sided
- +communication operation could have modified local data (an origin in
- +a remote read operation or a target in a remote write or accumulate operation), and the
- +operation completion has not been guaranteed by a successful
- +synchronization.
-
- \begin{users}
-     A high quality implementation may be able to limit the scope of the exposed
- -   memory that becomes undefined (for example, only the memory that has been
- -   targeted by a remote write, or has been an origin in a remote read).
- +   memory that becomes undefined (for example, only the memory addresses that have been
- +   targeted by a remote write, or have been an origin in a remote read).

Vengeance of the deads

- Possible issue with stalled (buffered?) RDMA messages
  - P0 posts a RDMA send/put to P1
  - P0 dies
  - P1 detects P0 is dead, marks requests as completed (in error), frees the window, the target buffer, etc.
  - Stall RDMA message from P0 gets delivered from network buffer, writes into the memory of P1 with no warning when it is not expected anymore
- Scenario is possible but of little practical relevance
  - Failure notification “faster” than RDMA message, really?...
  - In most transports (ugni, ib, etc), it is possible to remove the rdma key that exposes the memory (so the stall message is safely discarded)
  - K computer does not have this feature (yet), but it is being integrated as we speak (for security reasons, not for FT, because letting people write everywhere in your memory w/o checks is a bad idea anyway)
  - Even if the memory registration key cannot be removed from the hardware, an implementation can still use timeouts when clearing operations that are potentially releasing memory targeted by RDMA

Current position: the implementation must take care of it (either dropping the stall packets, or waiting long enough before reporting a process dead so that all stall buffers are guaranteed empty)

Want to participate?

- main forum ticket: https://svn.mpi-forum.org/trac/mpi-forum-web/attachment/ticket/323/
- Demand access to the standard draft development repo (with discussions bug tracking and milestone tickets, etc): https://bitbucket.org/bosilca/mpi3ft
- Open MPI implementation repo http://fault-tolerance.org