Plan B: Interruption of Ongoing MPI Operations to Support Failure Recovery

Aurelien Bouteiller, George Bosilca, Jack J. Dongarra

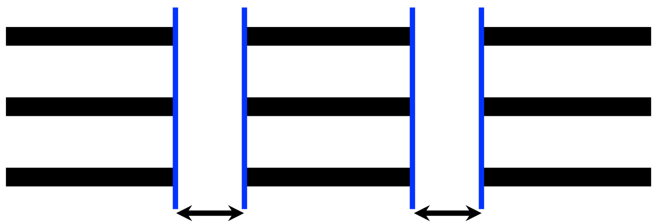
Euro MPI 2015
Bordeaux, France

Do we need fault tolerance?

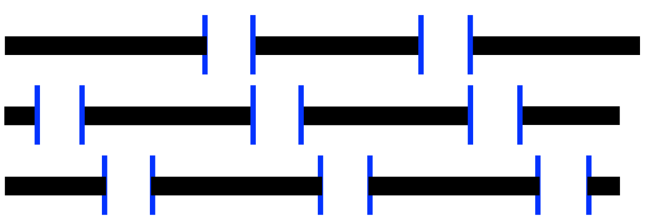
- No !
  - Hardware can take care of everything. And [of course] will !
    - The future tense is important !
- Meanwhile from a HPC viewpoint
  - Large platforms report several hard failures a day with tens/hundreds of applications to be rerun
  - ECC might not be enough to protect the data from Silent Data Corruptions
  - Future HPC platforms will grow in number of resources and by simple probabilistic deduction the frequency of faults will increase
- Parallel programming paradigms became mainstream, and HPC will not be the predominant target
  - What do we want MPI to be ?

Fault Tolerance techniques: 1/2
Rollback Recovery

Coordinated checkpoint
(with non-blocking, incremental checkpoints)



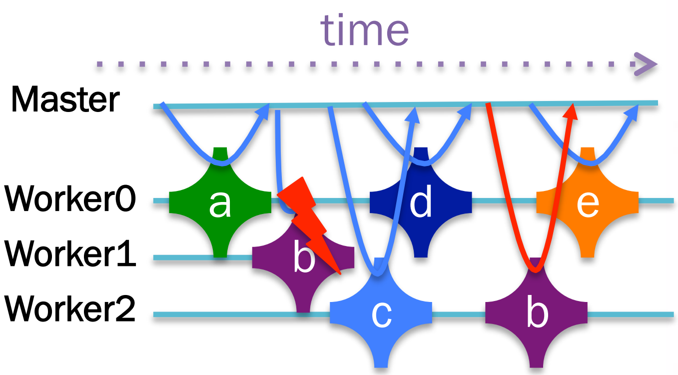
Uncoordinated checkpoint
(with non-blocking, incremental checkpoints)



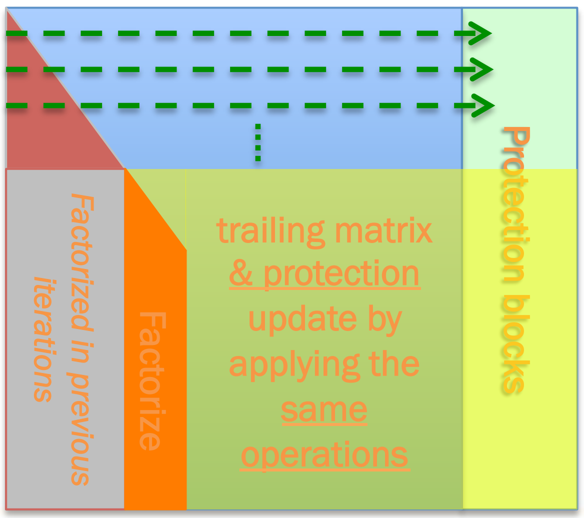
- Rollback recovery issues:
  - I/O overhead grows with scale (as MTBF declines)
    - Young/Dali Formulas used to compute optimal checkpoint interval
    - Results in too many preventive checkpoints
    - Eventually, more time spent doing checkpoints than real work
- Coordinated Checkpoint (legacy):
  - Low cost on communication
  - Coordinated recovery
- Uncoordinated Checkpoint:
  - Overhead on communication
  - Increased size of the checkpoint
  - Independent process recovery
    - Non faulty process continue progressing during recovery

Fault Tolerance Techniques 2/2
Forward Recovery

- Forward Recovery:
  - Any technique that permit the application to continue without rollback
    - Master-Worker with simple resubmission
    - Iterative methods, Naturally fault tolerant algorithms
    - Algorithm Based Fault Tolerance
    - Replication *(the only system level Forward Recovery)*
- No checkpoint I/O overhead
- No rollback, minimal loss of completed work
- May require (sometime expensive, like replicates) protection/recovery operations, *but generally still more scalable than checkpoint*
- Often requires in-depths algorithm rewrite (in contrast to automatic system based C/R)



ABFT



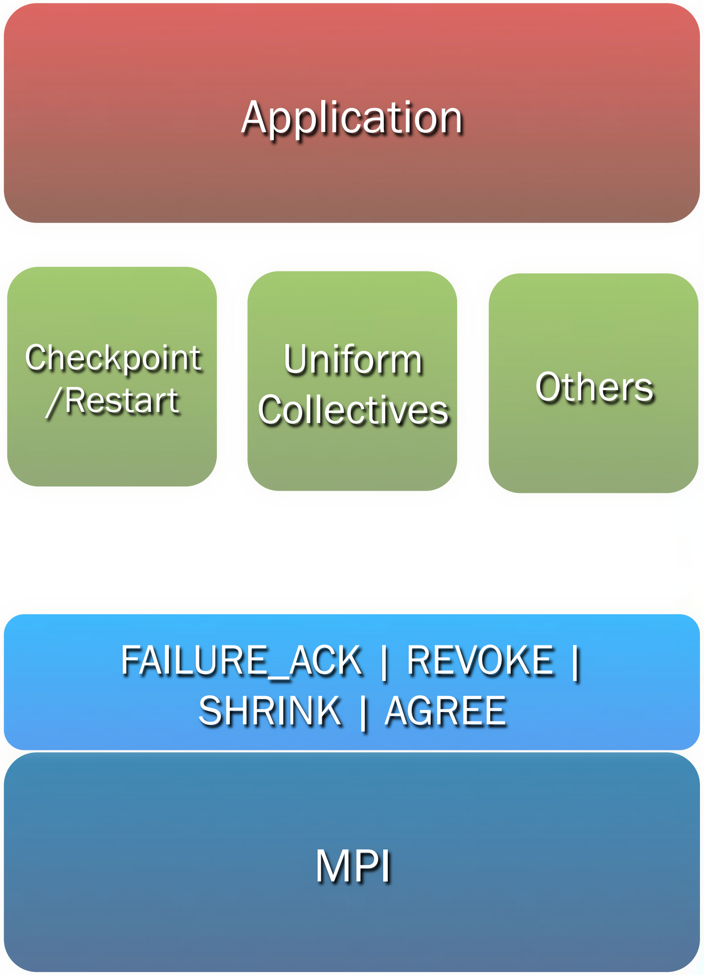
MPI-3: Fault Tolerance support

- We have algorithms (uncoordinated checkpoint, forward recovery), but they **expect MPI to continue to operate across failures**
  - MPI support of FT is non-existent
  - Prevents effective deployment of efficient, application specific approaches

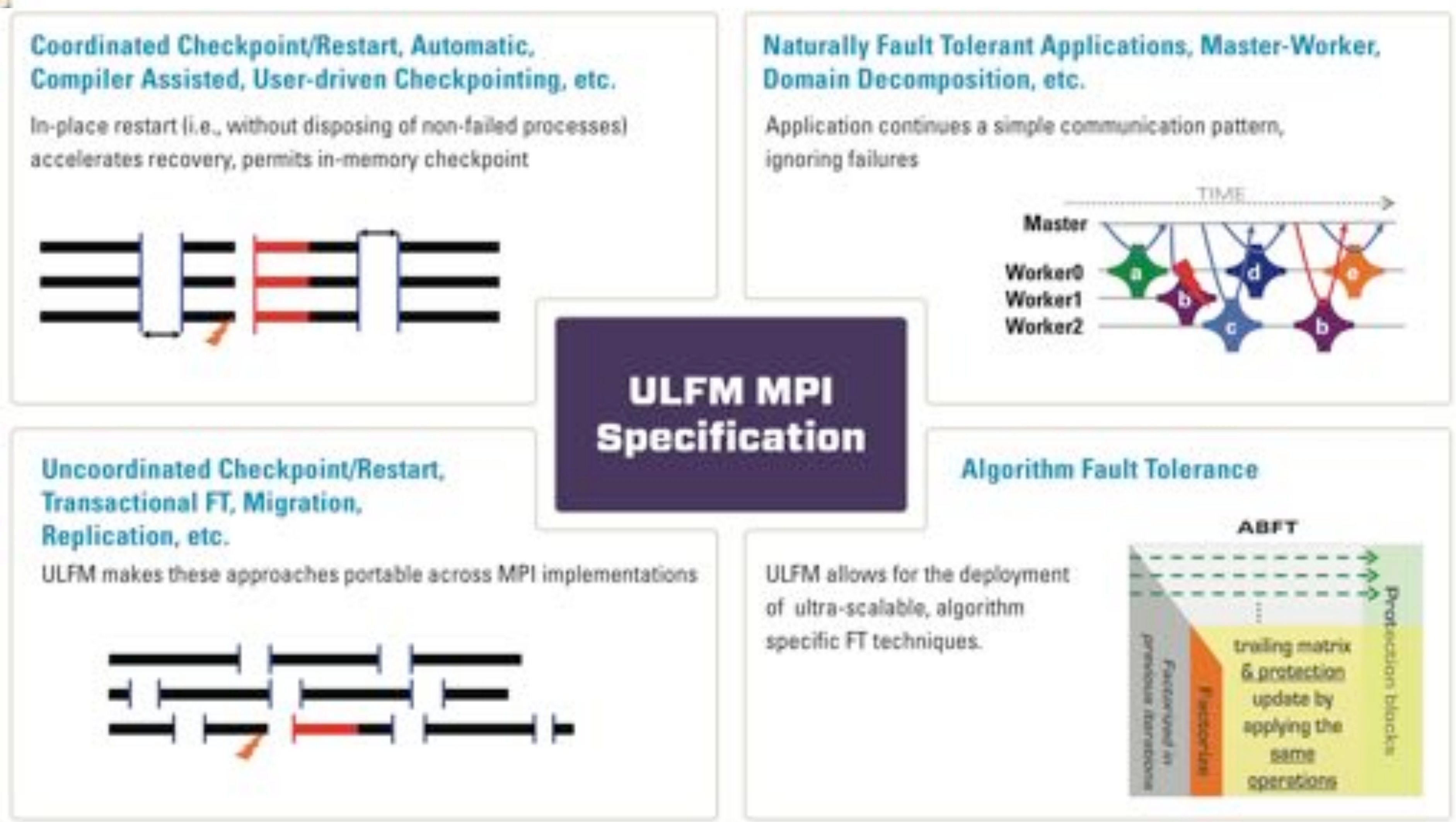
  - MPI_ERRORS_ARE_FATAL (default mode)
    - Application crashes at first failure
  - MPI_ERRORS_RETURN
    - Error returned to the user
    - State of MPI ***undefined***
      - *"…does not necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handler is to allow a user to issue user-defined error messages and take actions unrelated to MPI…An MPI implementation is free to allow MPI to continue after an error…"* (MPI-1.1, page 195)
      - *"Advice to implementors: A good quality implementation will, to the greatest possible extent, circumvent the impact of an error, so that normal processing can continue after an error handler was invoked."*

Requirements for MPI standardization of FT

- Expressive, simple to use
  - Support legacy code, backward compatible
  - Enable users to port their code simply
  - Support a variety of FT models and approaches
- Minimal (ideally **zero**) **impact** on failure free performance
  - No global knowledge of failures
  - No supplementary communications to maintain global state
  - Realistic memory requirements
- Simple to implement
  - Minimal (or **zero**) **changes to existing functions**
  - Limited number of new functions
  - Consider thread safety when designing the API



Application Recovery Patterns



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

ULFM: API extensions to “repair MPI”

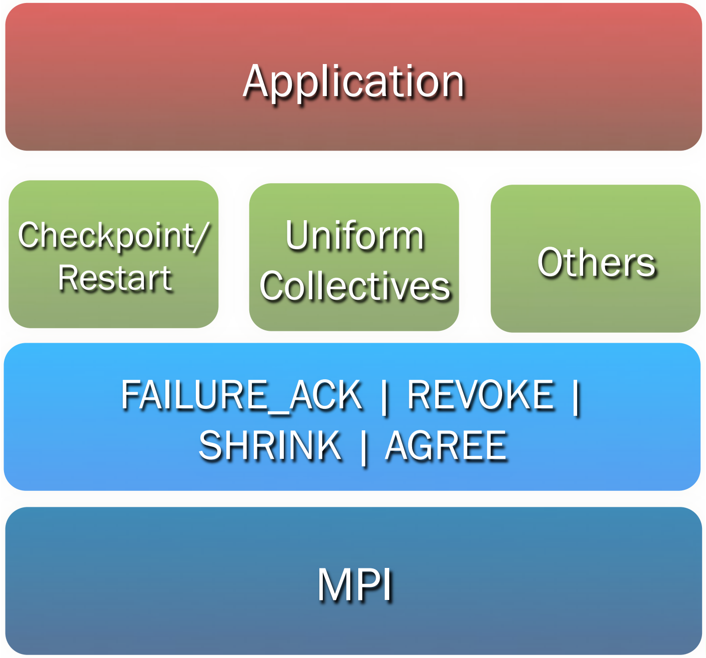
User Level Failure Mitigation: a set of MPI interface extensions to enable MPI programs to restore MPI communication capabilities disabled by failures

- Flexible:
  - Must accommodate all application recovery patterns
  - No particular model favored
  - Application directs recovery, pays only the necessary cost
- Performance:
  - Protective actions outside of critical path / communication routines
  - Unmodified collective, rendez-vous, RMA algorithms
  - Encourages a reactive programming style (diminish failure free overhead)
- Productivity:
  - Backward compatible with non-FT applications
  - A few simple concepts enable all types FT support (hard and soft failures)
  - Key concepts to support abstract models, libraries, languages, runtimes, etc

Minimal Feature Set for a Resilient 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

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

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

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

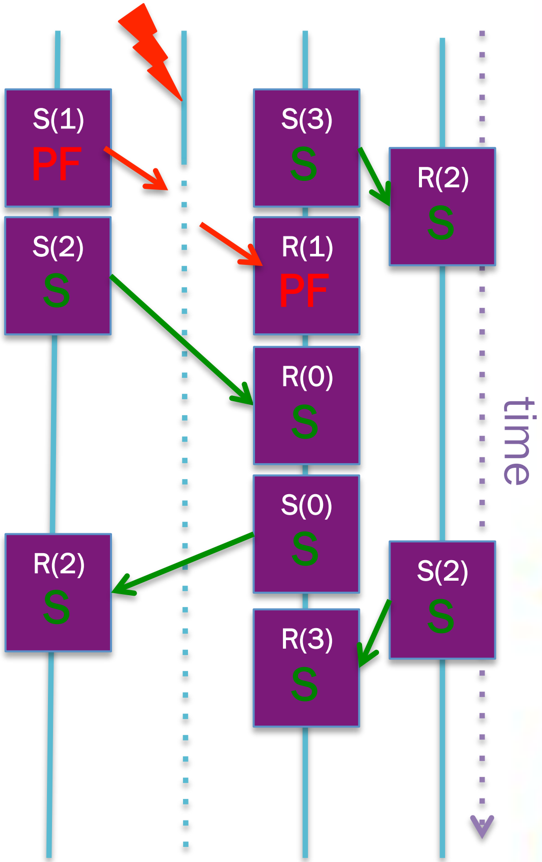
Notification

Propagation

Recovery

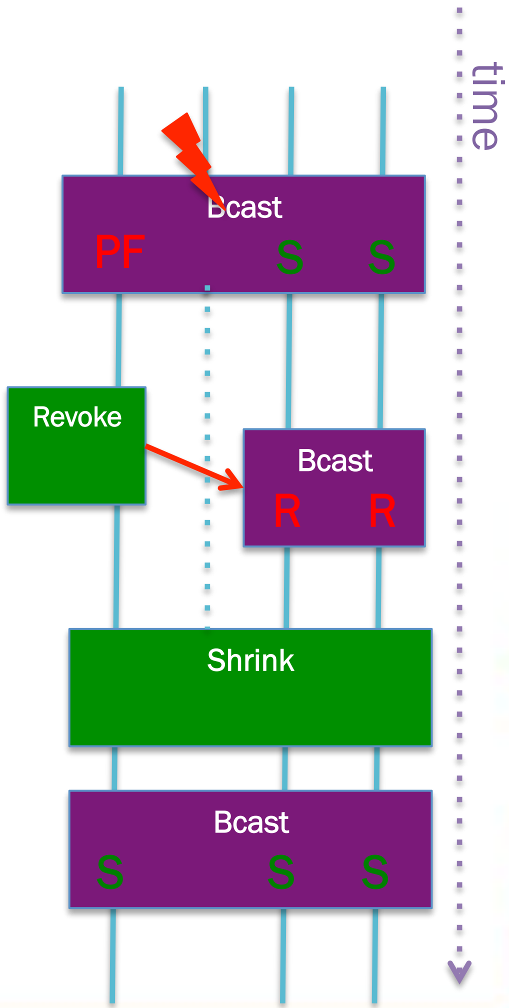
Errors are visible only for operations that can't complete

- Operations that can't complete return ERR_PROC_FAILED
  - State of MPI objects unchanged (communicators, etc)
  - Repeating the same operation has the same outcome
- Operations that can be completed return MPI_SUCCESS
  - Pt-2-pt operations between non failed ranks can continue

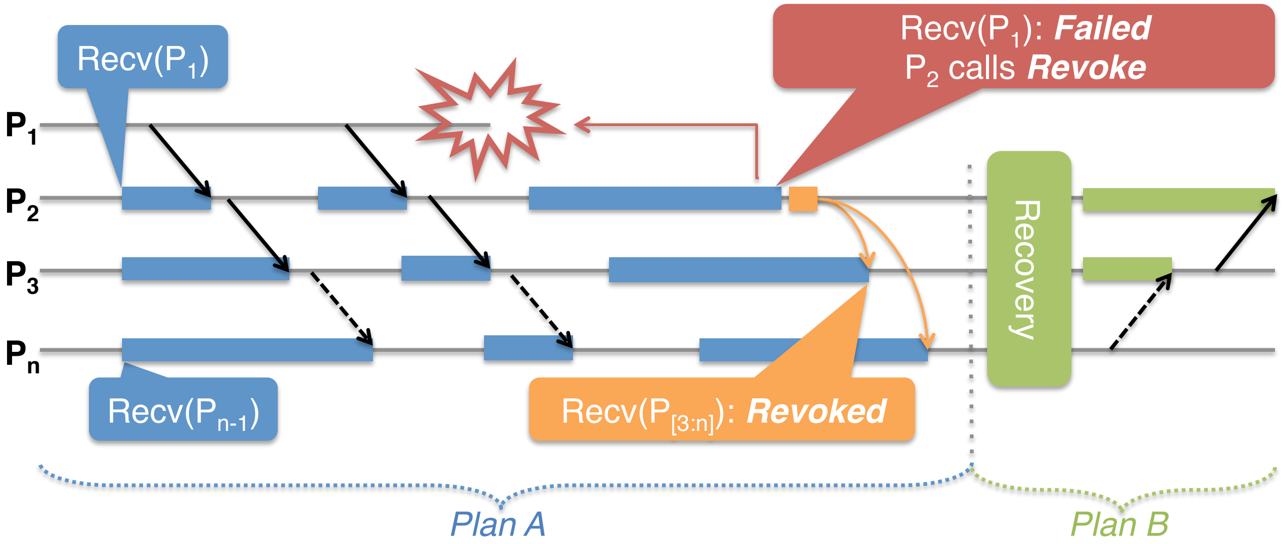


Incoherent global state and resolution

- Operations that **can't complete** return **ERR_PROC_FAILED**
- Operations that **can be completed** return **MPI_SUCCESS**
  - local semantic is respected (that is buffer content is defined), it **does not indicate success at other ranks**!
  - New constructs **MPI_Comm_revoke resolves inconsistencies** introduced by failures



Resolving transitive dependencies




```
proc_failed_err_handler(MPI_Comm comm, int err) {
   if(err == MPI_ERR_PROC_FAILED ||
      err == MPI_ERR_REVOKED ) {
     if(err == MPI_ERR_PROC_FAILED) 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 Communications

```
proc_failed_err_handler(MPI_Comm comm, int err) {
    if(err == MPI_ERR_PROC_FAILED ||
        err == MPI_ERR_REVOKED ) {
       if(err == MPI_ERR_PROC_FAILED) MPI_Comm_revoke(comm);
       recovery(comm);
    }
}
deadlocking_collectives(void) {
    for(i=0; i<nbrecv; i++)
        MPI_Bcast(buff, count, datatype, 0, comm);
}
```

- Lax consistency: Exceptions are raised only at ranks where the Bcast couldn't succeed
  - 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

Revoke is a critical operation that must be reliable and scalable

Contribution 1: MPI_Comm_revoke != Reliable Broadcast

- The revoke notification need to be propagated to all alive processes (almost like a reliable broadcast)
- In the context of MPI_Comm_revoke, the 4 defining qualities of a reliable broadcast (Termination, Validity, Integrity and Agreement) can be relaxed (non-uniform versions)
  - Agreement, Validity: once one process delivers *v*, then all processes delivers *v*. Revoke has a single state (revoked) and all processes will eventually converge their views.
  - Integrity: a message delivered at most once. The revoked communicator is immutable, so multiple deliveries is not an issue
  - Termination: Once a communicator is locally known as revoked no further propagation of the state change
- As we don't need uniform variants of the revoke operation, we are not bound to fully-connected overlay topologies (Hamiltonian is more than enough)

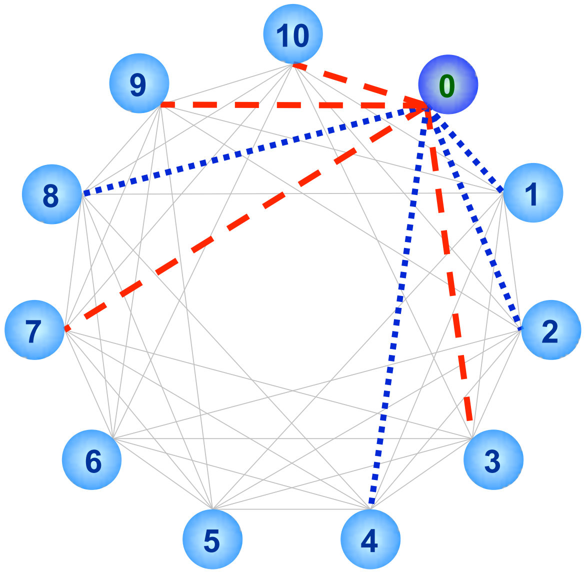
Contribution 2: Identifying a suitable underlying topology

- The basic behavior of a process: once it receives a revoke message for the first time it delivers it to all neighbors
  - The agreement property can only be guaranteed when failures do not disconnect the overlay graph
- Fully connected topologies do have such a property but they scale poorly with the number of processes. In practice:
  - Number of messages quadratic
  - Resource exhaustion: too many simultaneously opened channels, too many unexpected messages or posted receives
- We need a better topology with small degree and diameter, hardened and bridgeless
  - Torus, HiC, CST, Hypercube, Chord (not good enough)

Binomial Graph (BMG)

- Undirected graph $G:=(V, E)$, $|V|=n$ (any size)
  - Node $i=\{0,1,2,\ldots,n-1\}$ has links to a set of nodes U
    - $U=\{i\pm1, i\pm2,\ldots, i\pm2^k \mid 2^k \leq n\}$ in a circular space
    - $U=\{ (i+1) \bmod n, (i+2) \bmod n,\ldots, (i+2^k) \bmod n \mid 2^k \leq n \}$ and

      $\{ (n+i-1) \bmod n, (n+i-2) \bmod n,\ldots, (n+i-2^k) \bmod n \mid 2^k \leq n \}$

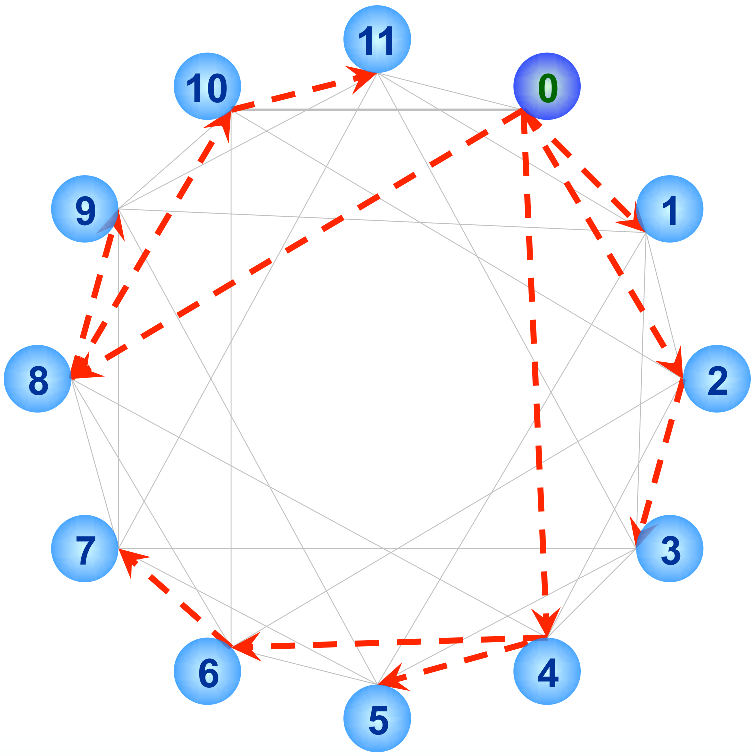


Belong to the connected Circulant graph family: biconnected, bridgeless, cyclic, Hamiltonian, LCF, regular, traceable, and vertex-transitive.

Angskun, T., Bosilca, G., Dongarra, J. "Binomial Graph: A Scalable and Fault- Tolerant Logical Network Topology," Proceedings of The Fifth International Symposium on Parallel and Distributed Processing and Applications (ISPA07), Springer, Niagara Falls, Canada, 2007

Binomial Graph (BMG)

- Merging all necessary links creates a binomial tree from each node in the graph.



Properties

1. Broadcast messages from any node within $\lceil \log_2(n) \rceil$ steps
2. Extremely difficult to bipartite
3. Easy to compute an alternate routing around failed processes
4. Interesting self-healing properties

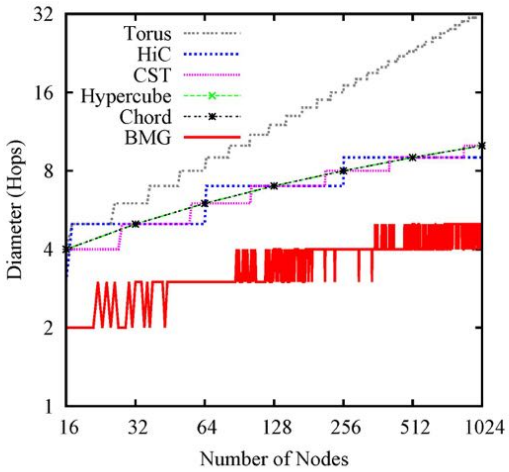
Basic Properties of BMG

- Degree δ (number of neighbors)

$$\delta = \begin{cases} (2 \times \lceil \log_2 n \rceil) - 1 & \text{For } n = 2^k\text{, where } k \in \mathbb{N} \\ (2 \times \lceil \log_2 n \rceil) - 2 & \text{For } n = 2^k + 2^j\text{, where } k, j \in \mathbb{N} \wedge k \neq j \\ 2 \times \lceil \log_2 n \rceil & \text{Otherwise} \end{cases}$$

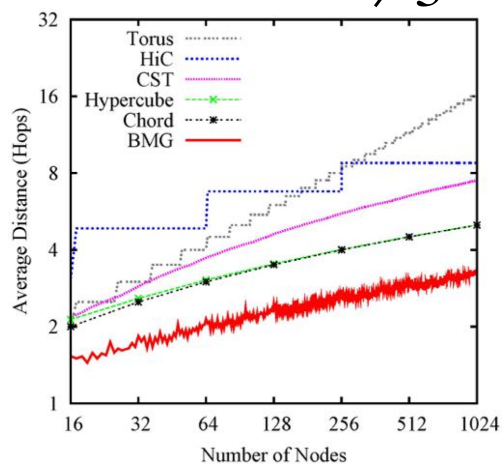
Diameter

$$(D) = O(\lceil \frac{\lceil \log_2(n) \rceil}{2} \rceil)$$

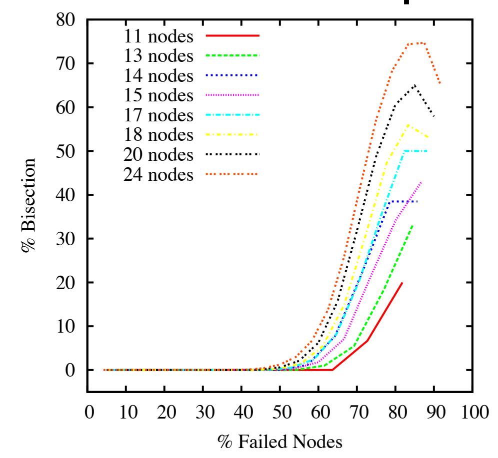


Average Distance

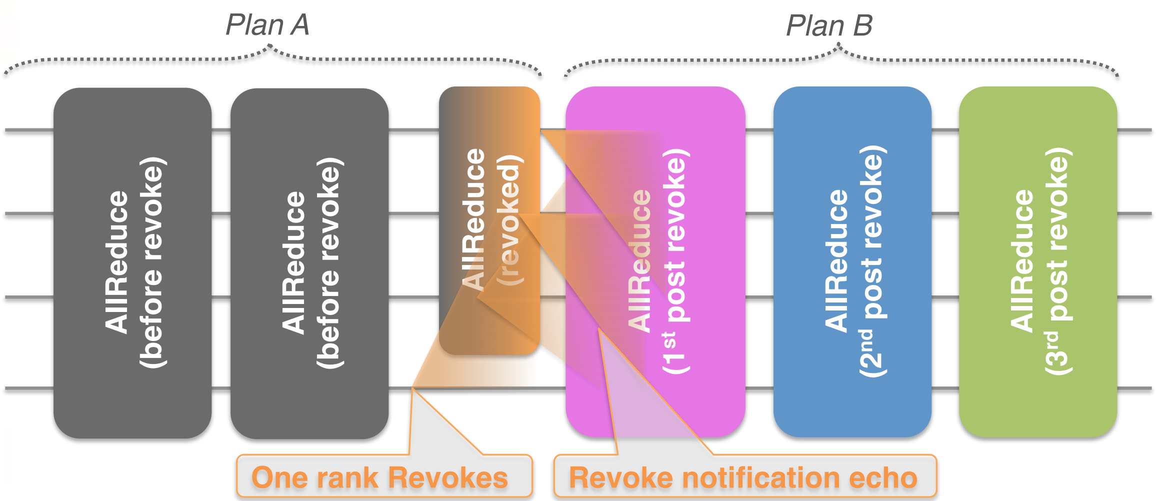
$$(\bar{d}) \approx \log_2(n) / 3$$



Bipartite vs. Failed relationship



Evaluating Revoke Cost

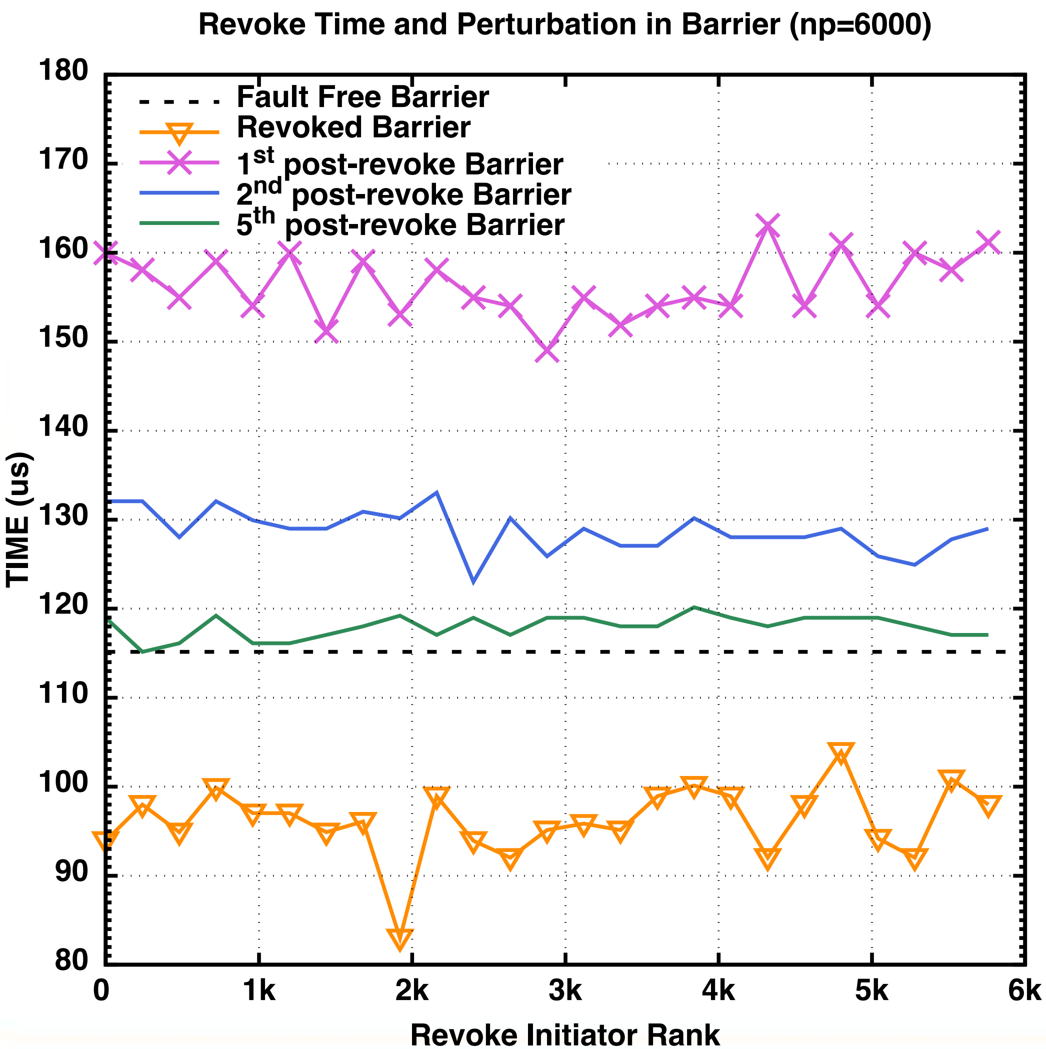


- The cost of Revoke cannot be measured directly. At the initial caller is essentially 0 (immediate operation, completes in the background)
- Instead we measure the impact of a revoke on subsequent operations
- Even after a Revoke has delivered to all ranks, the "revoke tokens" are still circulating on the network

- Two duplicate of MPI_COMM_WORLD:
- 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, … collective, until the background noise generated by revoke cannot be observed

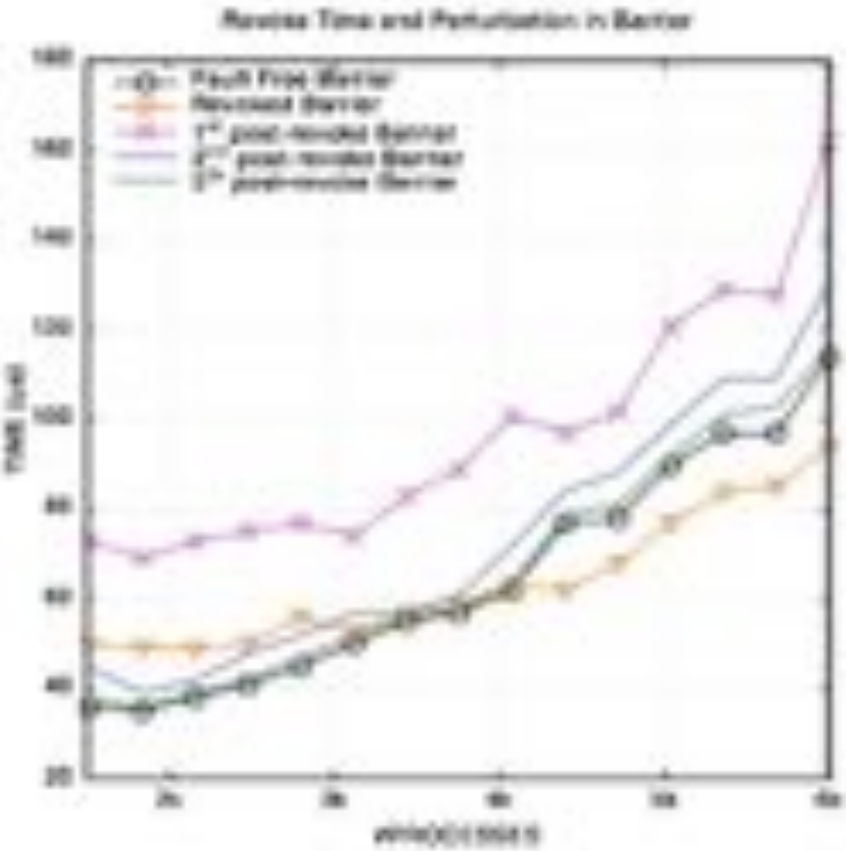
Darter platform, a Cray XC30 at NICS724 compute nodes with 2 x 2.6 GHz Intel 8-core XEON E5-2600 (Sandy Bridge), connected via a Cray Aries router with a bandwidth of 8GB/sec.

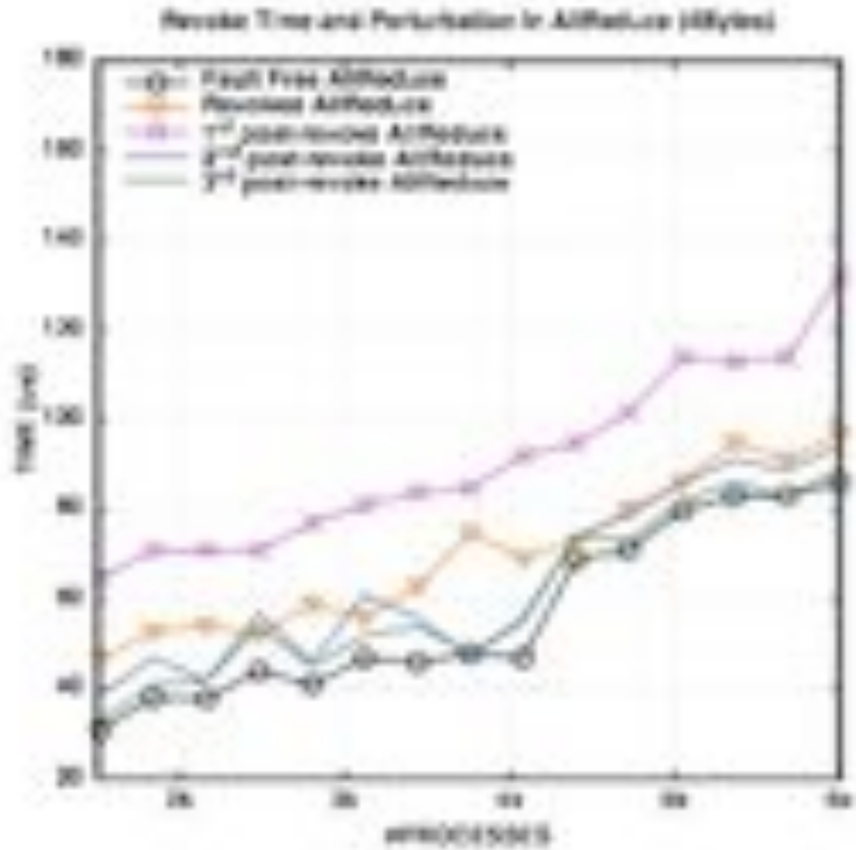
Evaluation: Initiator Location



- The underlying BMG topology is symmetric and reflects in the revoke which is independent of the initiator
- The performance of the first post-Revoke collective operation sustains some performance degradation resulting from the network jitter associated with the circulation of revoke tokens
- After the fifth Barrier (approximately 700μs), the application is fully resynchronized, and the Revoke reliable broadcast has completely terminated, therefore leaving the application free from observable jitter.

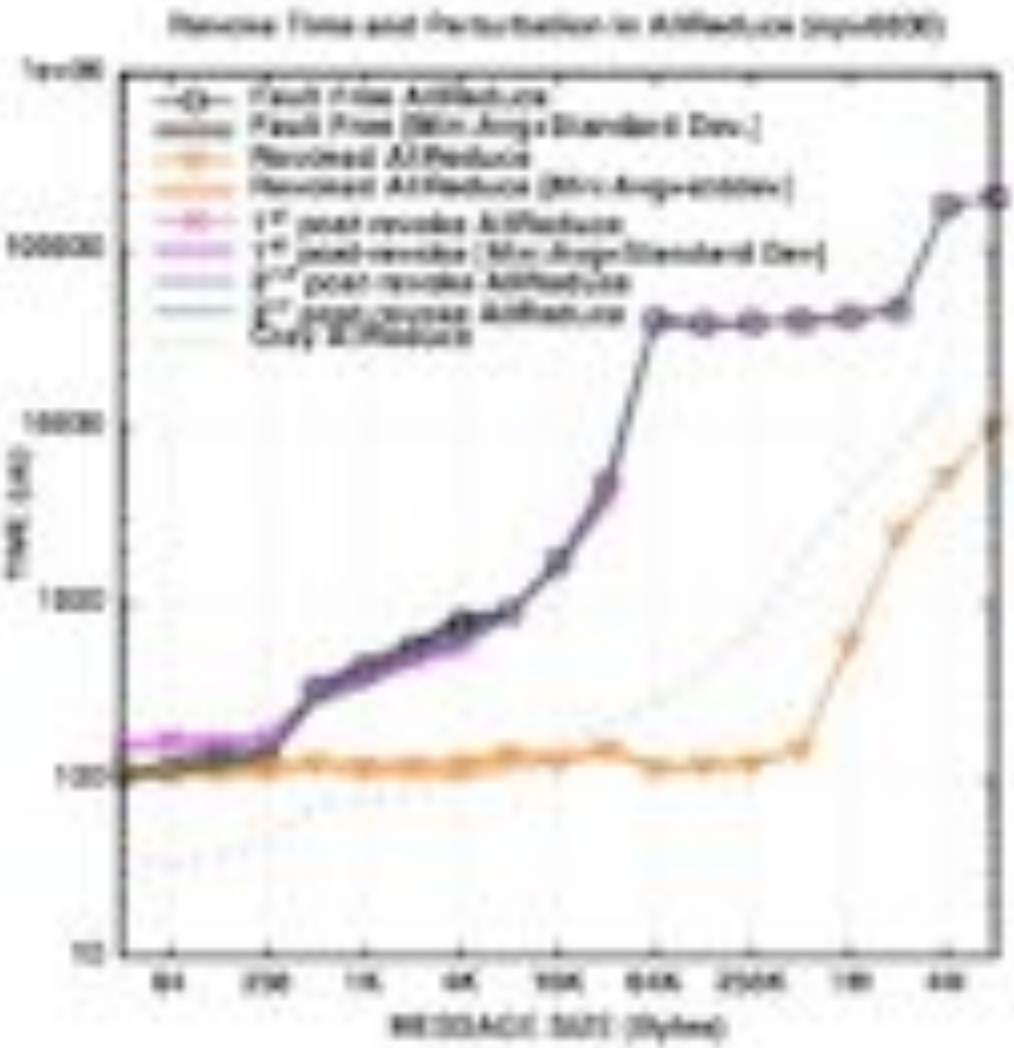
Evaluation: Collective pattern





Performance of post-Revoke collective communications follows the same scalability trend as the pre-Revoke operations, even those impacted by jitter.

Evaluation: Message Size



- Propagation time for Revoke messages ~= small message allreduce latency
- After the revoke has propagated, noise continue for another small message allreduce latency
- Performance penalty only visible for small message operations and only for a short duration.

Conclusion

- ULFM is not a fault management approach
  - It's a toolbox to build higher-level application/domain specific techniques
  - Critical to improve the scalability and performance of the ULFM constructs
    - detection / revoke / agreement*
- There are now viable alternatives to handling the faults by C/R
  - HPC applications can definitively benefit
  - This makes MPI a suitable programming environment for domains outside HPC

* Herault, T., Bouteiller, A., Bosilca, G., Gamell, M., Teranishi, K., Parashar, M., Dongarra, J. "Practical Scalable Consensus for Pseudo-Synchronous Distributed Systems," SuperComputing, Austin, TX, November, 2015

More info, resources

http://fault-tolerance.org/

- Standard draft document
  - https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/323
- Prototype implementation available
  - Version 1.0 based on Open MPI 1.6 released early September 2015
    https://bitbucket.org/icldistcomp/ulfm
  - Full communicator-based (point-to-point and all flavors of collectives) support
  - Network support IB, uGNI, TCP, SM
  - RMA, I/O in progress