Fault Tolerant MPI

Implementation, user's stories, use cases, performance

Aurelien Bouteiller (FT Working Group)

ICL UT INNOVATIVE COMPUTING LABORATORY THE UNIVERSITY of TENNESSEE

ULFM: API extensions to “repair MPI”

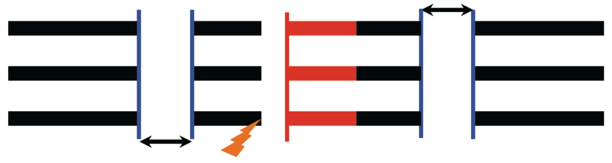
User Level Failure Mitigation: a set of MPI interface extensions to enable MPI programs to restore MPI communication capabilities disabled by failures

- 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 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 FT support
  - Key concepts to support abstract models, libraries, languages, runtimes, etc

Application Recovery Patterns

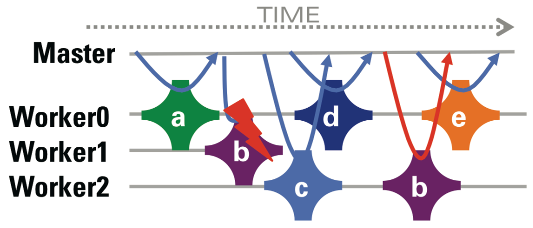
Coordinated Checkpoint/Restart, Automatic, Compiler Assisted, User-driven Checkpointing, etc.

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.

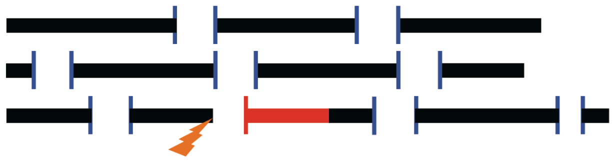
Application continues a simple communication pattern, ignoring failures



ULFM MPI Specification

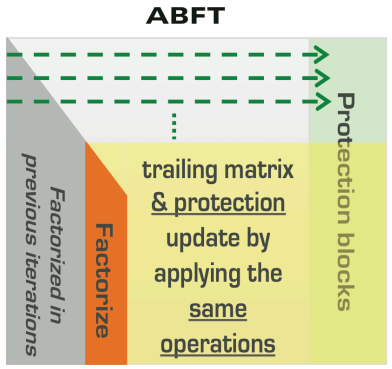
Uncoordinated Checkpoint/Restart, Transactional FT, Migration, Replication, etc.

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

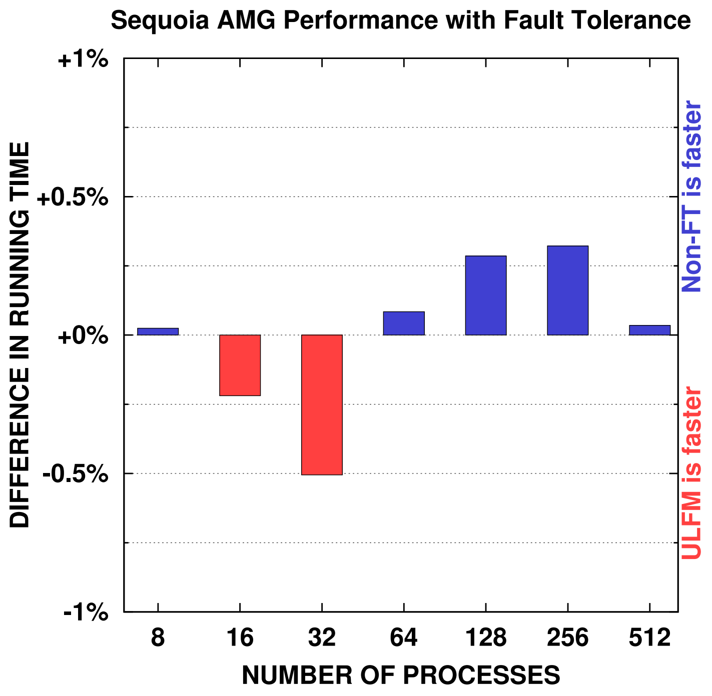
Minimal Feature Set for FT

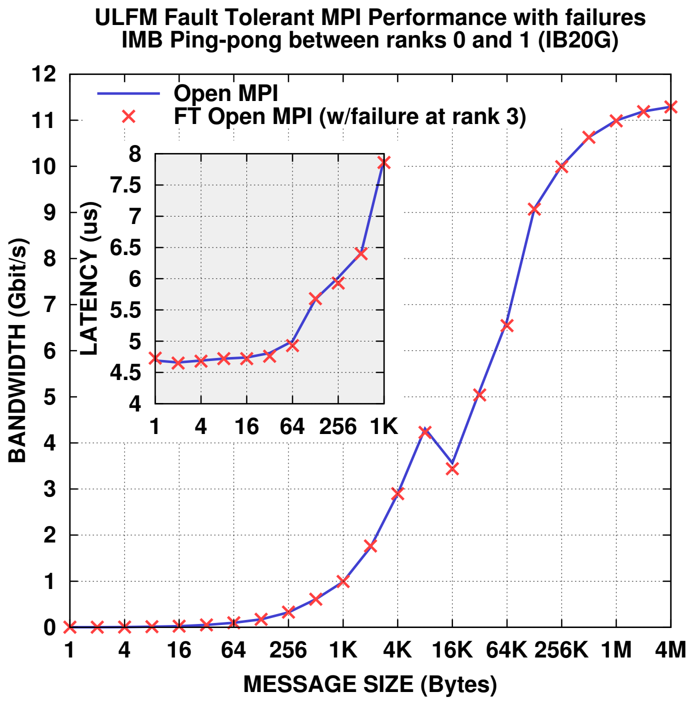
- Failure Notification
- Error Propagation
- Error Recovery

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

Implementation in Open MPI

- It works! Performance is good!





Sequoia AMG is an unstructured physics mesh application with a complex communication pattern that employs both point-to-point and collective operations. Its failure free performance is unchanged whether it is deployed with ULFM or normal Open MPI.

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.

Thanks for CREST, Riken support

User activities

- ORNL: Molecular Dynamic simulation
  - Employs coordinated user-level C/R, in place restart with Shrink
- UAB: transactional FT programming model
- Tsukuba: Phalanx Master-worker framework
- Georgia University: Wang Landau Polymer Freezing and Collapse
  - Employs two-level communication scheme with group checkpoints
  - Upon failure, the tightly coupled group restarts from checkpoint, the other distant groups continue undisturbed
- Sandia: Sparse solver
- INRIA: Sparse PDE solver
- Cray: CREST miniapps, PDE solver Schwartz, PPStee (Mesh, automotive), HemeLB (Lattice Boltzmann)
- UTK: FTLA (dense Linear Algebra)
  - Employs ABFT
  - FTQR returns an error to the app, App calls new BLACS repair constructs (spawn new processes with MPI_COMM_SPAWN), and re-enters FTQR to resume (ABFT recovery embedded)
- ETH Zurich: Monte-Carlo
  - Upon failure, shrink the global communicator (that contains spares) to recreate the same domain decomposition, restart MC with same rank mapping as before

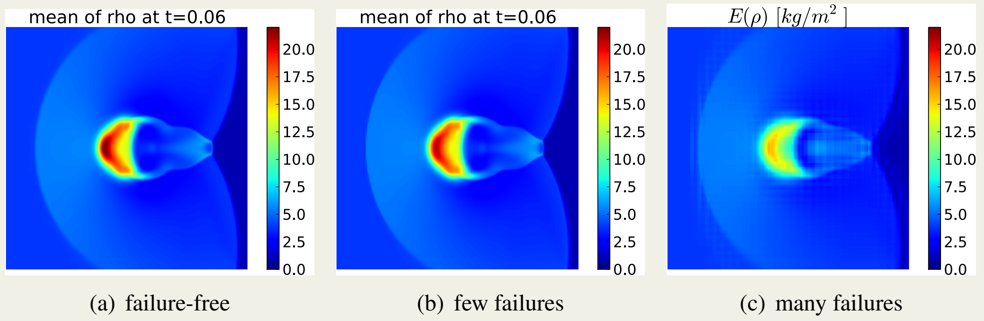


Figure 5. Results of the FT-MLMC implementation for three different failure scenarios.

Credits: ETH Zurich

Applications

HemeLB

PPSTee

Asynchronous optimized Schwarz methods

Collaborative Research into Exascale Systemware, Tools & Applications

- **Applications chosen as representative sample of HPC**
- **Providing feedback as co-design vehicles**
- **Subset appropriate to test Fault Tolerant MPI**

Applications

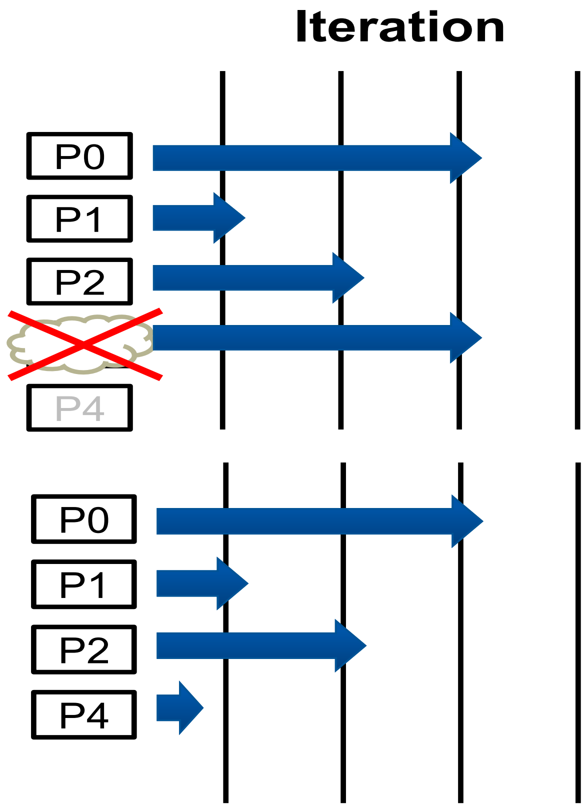
Asynchronous Schwarz Methods

PDE Solver with independent local Iteration Counters
Ecole Centrale Paris

- **Domain decomposition with independent, asynchronous boundary exchange**
- **Converges independent of local iteration counters**

Processor fails:

➢ **Re-initialize the substitute processor with initial solution and continue solving**



Applications

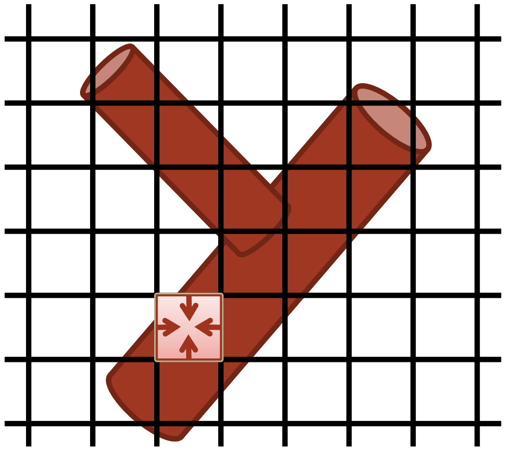
HemeLB

Lattice Boltzmann Flow Solver
University College London

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

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

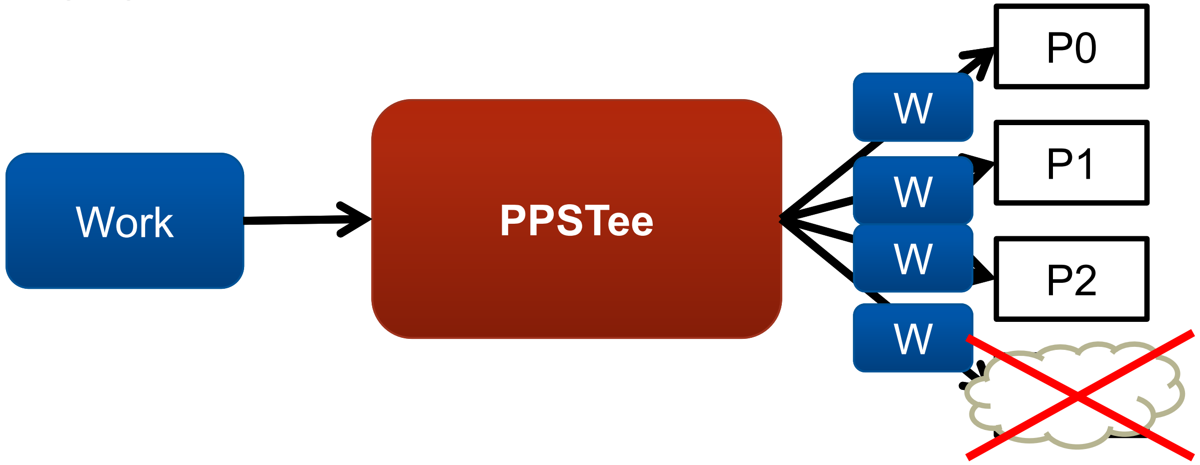


Applications

PPSTee

Pre-processing Steering Interface
German Aerospace Center (DLR)

- **Load data used to steer dynamic re-meshing**
  - Core simulation
  - Post-processing
  - Visualization



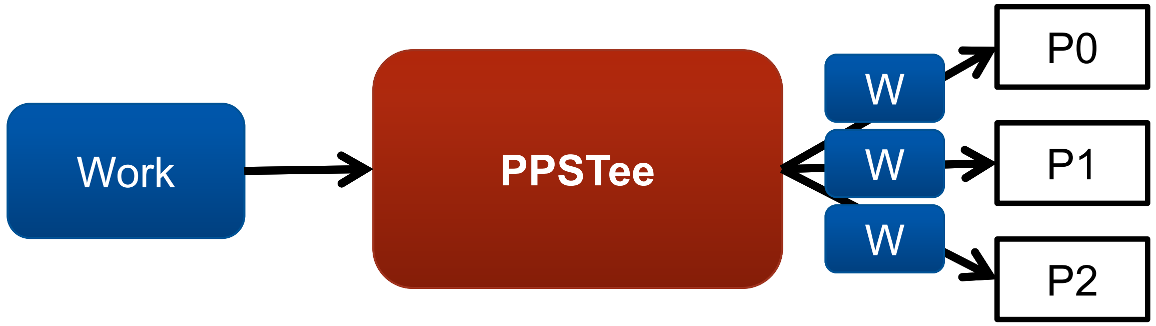
Applications

PPSTee

Pre-processing Steering Interface
German Aerospace Center (DLR)

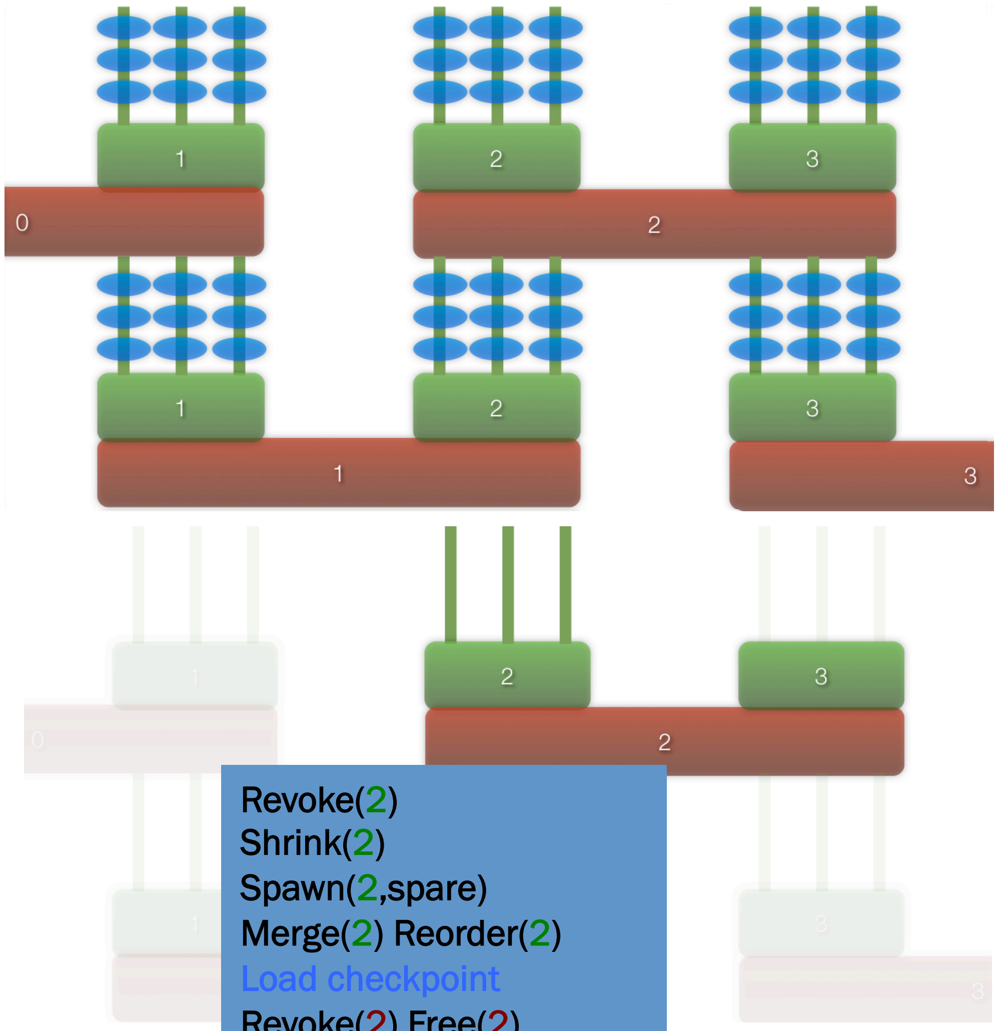
Processor fails:

- **Trigger re-distribution of work on remaining processors**
  (weight=0 on the failed processor)



U-GA: Wang-Landau polymer Freeze

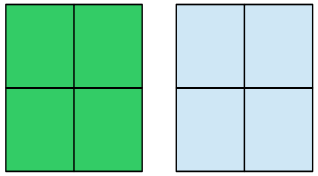
- Long independent computation on each processor
  - Dataset protected by small, cheap checkpoints (stored on neighbors)
- Periodically, an AllReduce on the communicator of the Energy window
- Immediately after, a Scatter and many pt2pt on the communicator linking neighboring energy windows

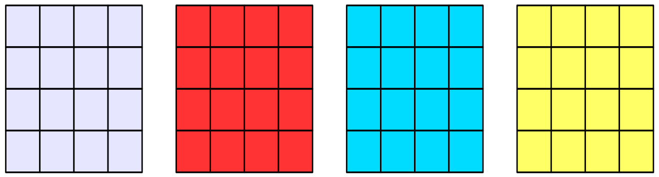


Revoke(2)
Shrink(2)
Spawn(2,spare)
Merge(2) Reorder(2)
Load checkpoint
Revoke(2) Free(2)
Connect(2) Accept(3)
Merge(2) Reorder

ETH-Zurich: Monte-Carlo PDE

- X is the solution to a stochastic PDE
- Each sample X^i is computed with a FVM solver
- MC error is determined by
  - stochastic error (depends on M)
  - discretization error (depends on the mesh-width h)
- A more accurate MC approximation requires more samples M and a finer mesh h

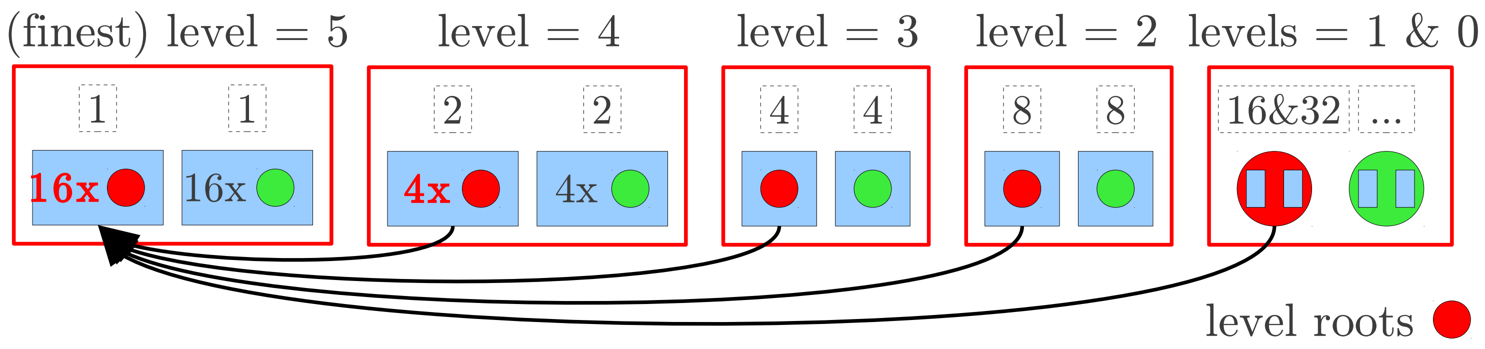




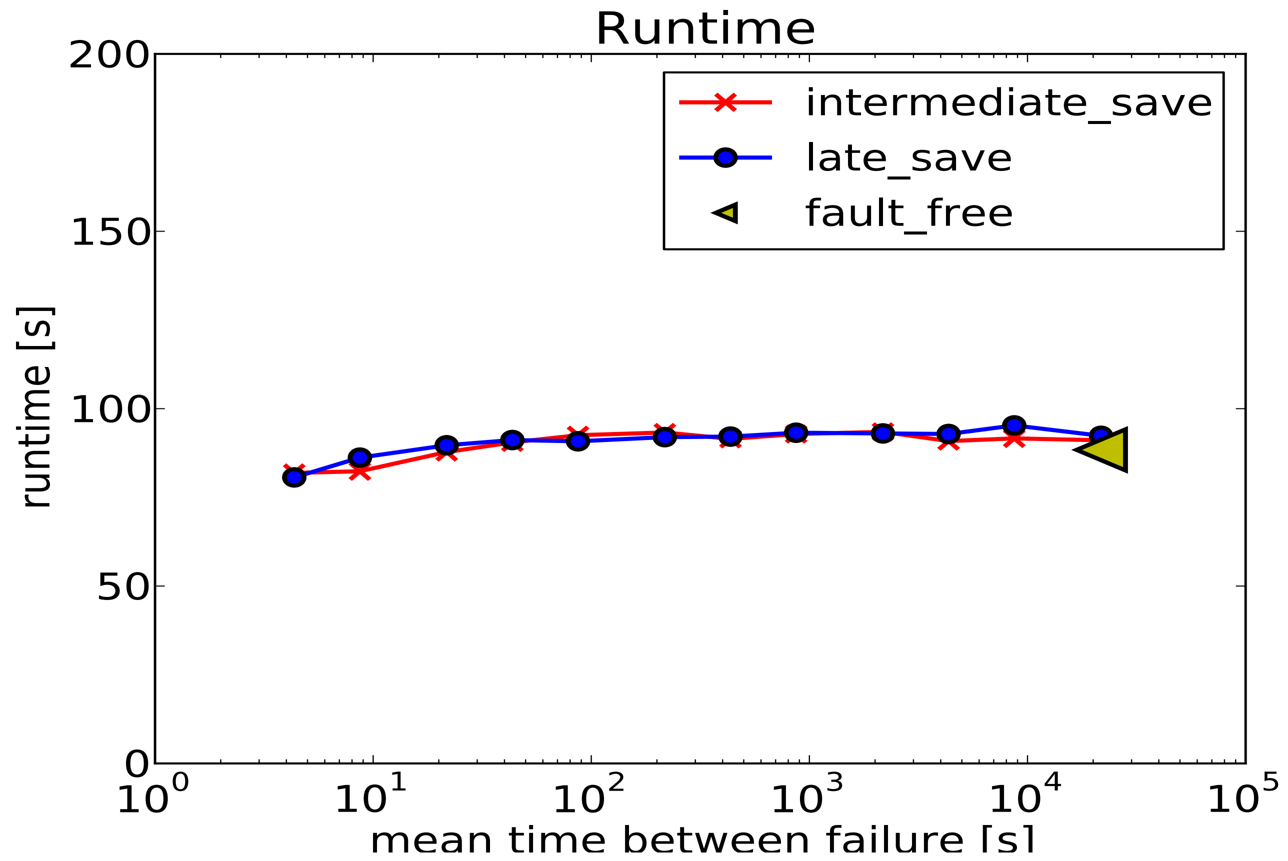
Fault Tolerant Monte Carlo:

- The number of samples M turns into a random variable $\hat{M}$
- $\|\mathbb{E}[X] - E_{\hat{M}}[X]\| \leq \mathbb{E}\left[\frac{1}{\sqrt{\hat{M}}}\right]\|X\|$

ETH-Zurich: Monte-Carlo PDE



- Try to collect the mean as in fault-free ALSVID-UQ
- Call MPI_BARRIER on MPI_COMM_WORLD at the end to discover failed processes
- non-uniform success of MPI_BARRIER: MPI_BARRIER is followed by MPI_COMM_AGREE
- In case of failure: (Re)assign the level roots and repeat the collection of the means



- Small overhead ($\approx 5\%$) in the run-time compared to the fault-free ALSVID-UQ
- Large samples are likely to abort in the presence of a high failure rate $\longrightarrow$ reduced runtime

AIST: Runtime System

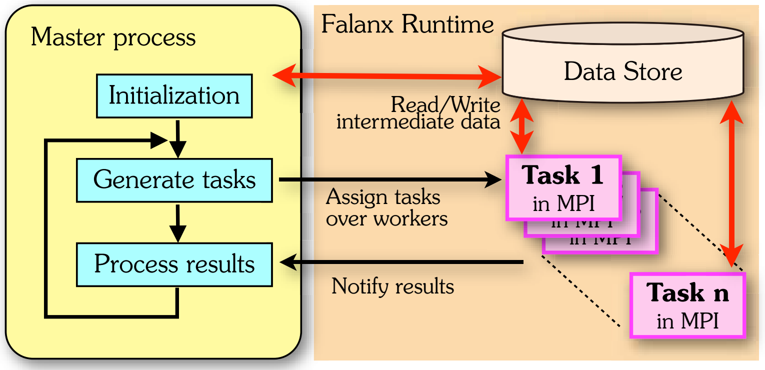


Figure 1: A schematic structure of a Falanx application.

- Falanx dataflow runtime
- Advanced Master-worker with parallel (MPI) worker jobs
- All complexity of MPI (and ULFM) masked inside the runtime ☺

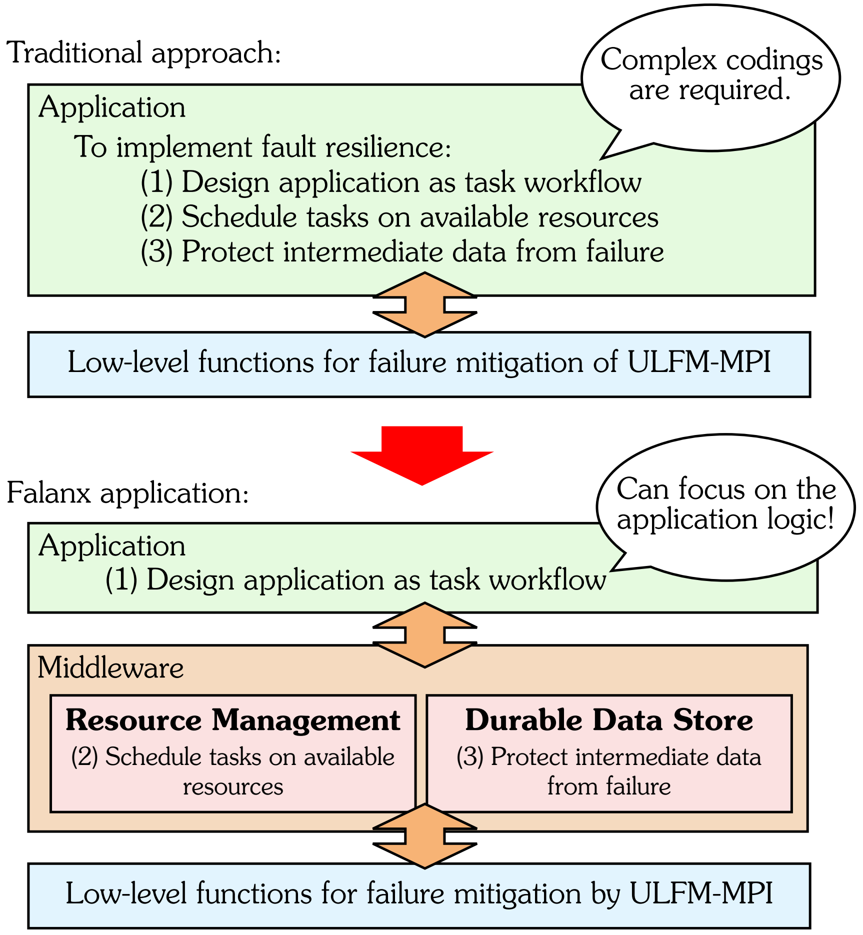

Traditional approach:
Complex codings are required.
Application
To implement fault resilience:
(1) Design application as task workflow
(2) Schedule tasks on available resources
(3) Protect intermediate data from failure
Low-level functions for failure mitigation of ULFM-MPI
Falanx application:
Can focus on the application logic!
Application
(1) Design application as task workflow
Middleware
Resource Management
(2) Schedule tasks on available resources
Durable Data Store
(3) Protect intermediate data from failure
Low-level functions for failure mitigation by ULFM-MPI

Figure 2: Implementation of fault resilience with and without Falanx.

Credits: Ikigamo et al. (AIST)

UAB: transactional model

```
communication initialization;
if restarted then
    load data from last checkpoint (optional);
end
repeat
    while more_work_to_do do
        MPI_TryBlock_start();
        computation, communication and/or I/O;
        wait for operations to finish;
        inject local errors;
        MPI_TryBlock_finish();
        if failure happened then
            isolate and mitigate the failure;
            if recovery_needed then break;
        end
        periodically checkpoint;
    end
    if recovery_needed then
        do recovery procedure;
    end
until more_work_to_do or restart_needed;
```

Algorithm 1: A basic application using FA-MPI

- Amin Hassani and Anthony Skjellum presented this idea 1 year ago
- Amin has implemented the core concepts of FA-MPI on top of ULFM MPI
- Provides a higher level abstraction than ULFM (but more targeted to a particular programming style)

Spawning replacement ranks 1/2

```
int MPIX_Comm_replace(MPI_Comm comm, MPI_Comm *newcomm) {
    MPI_Comm shrinked, spawned, merged;
    int rc, flag, flagr, nc, ns;

    redo:
        MPI_Comm_shrink(comm, &shrinked);
        MPI_Comm_size(comm, &nc); MPI_Comm_size(shrinked, &ns);
        rc = MPI_Comm_spawn(…, nc-ns, …, 0, shrinked, &spawned, …);
        flag = MPI_SUCCESS==rc;
        MPI_Comm_agree(shrinked, &flag);
        if( !flag ) {
            if(MPI_SUCCESS == rc) MPI_Comm_free(&spawned);
            MPI_Comm_free(&shrinked);
            goto redo;
        }
        rc = MPI_Intercomm_merge(spawned, 0, &merged);
        flagr = flag = MPI_SUCCESS==rc;
        MPI_Comm_agree(shrinked, &flag);
        MPI_Comm_agree(spawned, &flagr);
        if( !flag || !flagr ) {
            if(MPI_SUCCESS == rc) MPI_Comm_free(&merged);
            MPI_Comm_free(&spawned);
            MPI_Comm_free(&shrinked);
            goto redo;
        }
```

Spawning replacement ranks 2/2

```
int MPIX_Comm_replace(MPI_Comm comm, MPI_Comm *newcomm) {
    …
/* merged contains a replacement for comm, ranks are not ordered properly */
    int c_rank, s_rank;
    MPI_Comm_rank(comm, &c_rank);
    MPI_Comm_rank(shrinked, &s_rank);
    if( 0 == s_rank ) {
        MPI_Comm_grp c_grp, s_grp, f_grp; int nf;
        MPI_Comm_group(comm, &c_grp); MPI_Comm_group(shrinked, s_grp);
        MPI_Group_difference(c_grp, s_grp, &f_grp);
        MPI_Group_size(f_grp, &nf);
        for(int r_rank=0; r_rank<nf; r_rank++) {
            int f_rank;
            MPI_Group_translate_ranks(f_grp, 1, &r_rank, c_grp, f_rank);
            MPI_Send(&f_rank, 1, MPI_INT, r_rank, 0, spawned);
        }
    }
    rc = MPI_Comm_split(merged, 0, c_rank, newcomm);
    flag = (MPI_SUCCESS==rc);
    MPI_Comm_agree(merged, &flag);
    if( !flag ) { goto redo; } // (removed the Free clutter here)
```

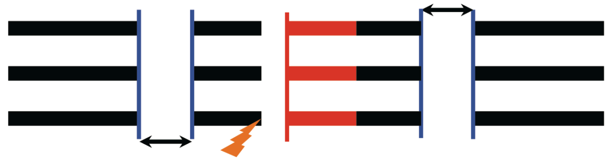
Example: in-memory C/R

```
int checkpoint_restart(MPI_Comm *comm) {
    int rc, flag;
    checkpoint_in_memory(); // store a local copy of my checkpoint
    rc = checkpoint_to(*comm, (myrank+1)%np); //store a copy on myrank+1
    flag = (MPI_SUCCESS==rc); MPI_Comm_agree(*comm, &flag);
    if( !flag ) { // if checkpoint fails, we need restart!
        MPI_Comm newcomm; int f_rank; int nf;
        MPI_Group c_grp, n_grp, f_grp;
redo:
        MPIX_Comm_replace(*comm, &newcomm);
        MPI_Comm_group(*comm, &c_grp); MPI_Comm_group(newgroup, &n_grp);
        MPI_Comm_difference(c_grp, n_grp, &f_grp);
        MPI_Group_size(f_grp, &nf);
        for(int i=0; i<nf; i++) {
            MPI_Group_translate_ranks(f_grp, 1, &i, c_grp, &f_rank);
            if( (myrank+np-1)%np == f_rank ) {
                serve_checkpoint_to(newcomm, f_rank);
            }
        }
        MPI_Group_free(&n_grp); MPI_Group_free(&c_grp); MPI_Group_free(&f_grp);
        rc = MPI_Barrier(newcomm);
        flag=(MPI_SUCCESS==rc); MPI_Comm_agree(*comm, &flag);
        if( !flag ) goto redo; // again, all free clutter not shown
        restart_from_memory(); // rollback from local memory
        MPI_Comm_free(comm);
        *comm = newcomm;
    }
```

Application Recovery Patterns

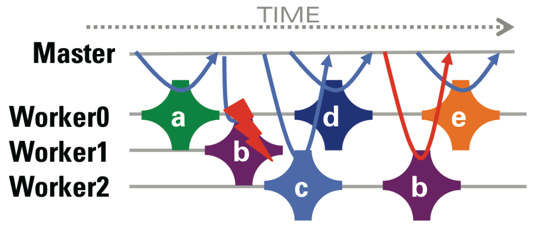
Coordinated Checkpoint/Restart, Automatic, Compiler Assisted, User-driven Checkpointing, etc.

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.

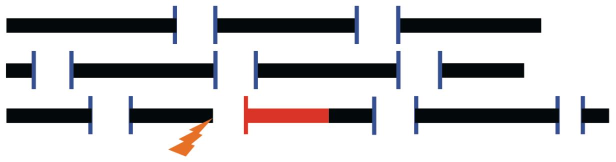
Application continues a simple communication pattern, ignoring failures



ULFM MPI Specification

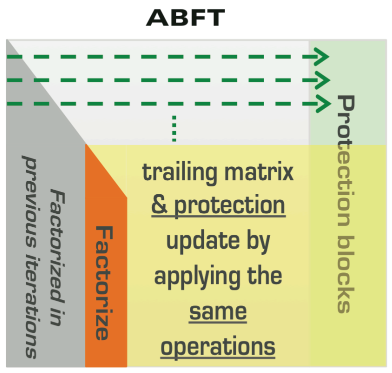
Uncoordinated Checkpoint/Restart, Transactional FT, Migration, Replication, etc.

ULFM makes these approaches portable across MPI implementations



Algorithm Fault Tolerance

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


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

Future directions

- Transient failures
  - Implementations can work around transient failures by “promoting” them to fail-stop
  - This proposal does not talk about them (on purpose, to leave room for future directions)
- KISS for C/R
  - This proposal supports C/R and restart with same proc number
  - It is however not necessarily easy to write with a deep call stack
  - We would like to explore addition of “revoke_all” that destroys all communicators (and possibly more of the MPI objects), to automate “wiping out” MPI state and reconstruct only MPI_COMM_WORLD
- Conditional init of FT, introspection
  - Turn on FT support only if MPI init has special parameters
  - Dependent on fate of external tickets (MPI_Init_with_info)

Thank you

To know more…

http://fault-tolerance.org/ulfm/

ICL UT INNOVATIVE COMPUTING LABORATORY THE UNIVERSITY of TENNESSEE

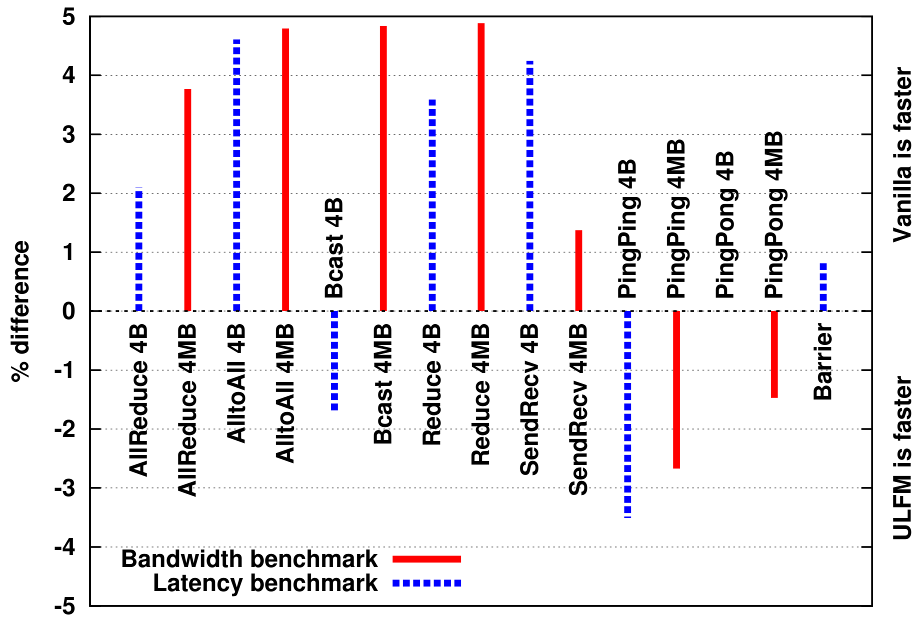
More performance: synthetic benchmarks

1-byte Latency (microseconds) (cache hot)

Interconnect	Vanilla	Std. Dev.	Enabled	Std. Dev.	Difference
Shared Memory	0.8008	0.0093	0.8016	0.0161	0.0008
TCP	10.2564	0.0946	10.2776	0.1065	0.0212
OpenIB	4.9637	0.0018	4.9650	0.0022	0.0013

Bandwidth (Mbps) (cache hot)

Interconnect	Vanilla	Std. Dev.	Enabled	Std. Dev.	Difference
Shared Memory	10,625.92	23.46	10,602.68	30.73	-23.24
TCP	6,311.38	14.42	6,302.75	10.72	-8.63
OpenIB	9,688.85	3.29	9,689.13	3.77	0.28

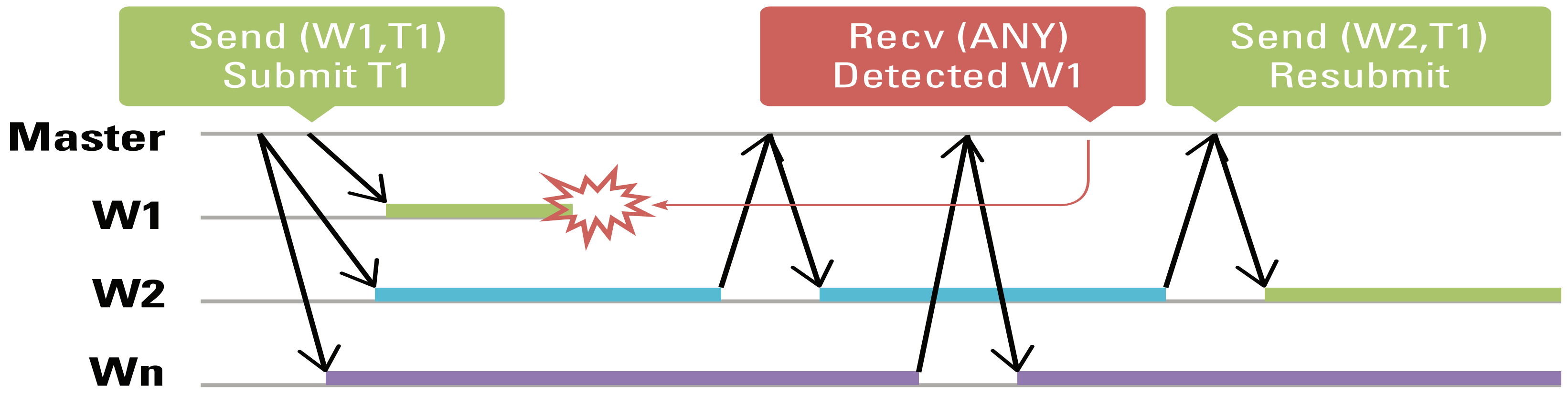


Collective communications:
48 core shared memory (very stressful)
Performance difference is less than std-deviation

Failure Notification

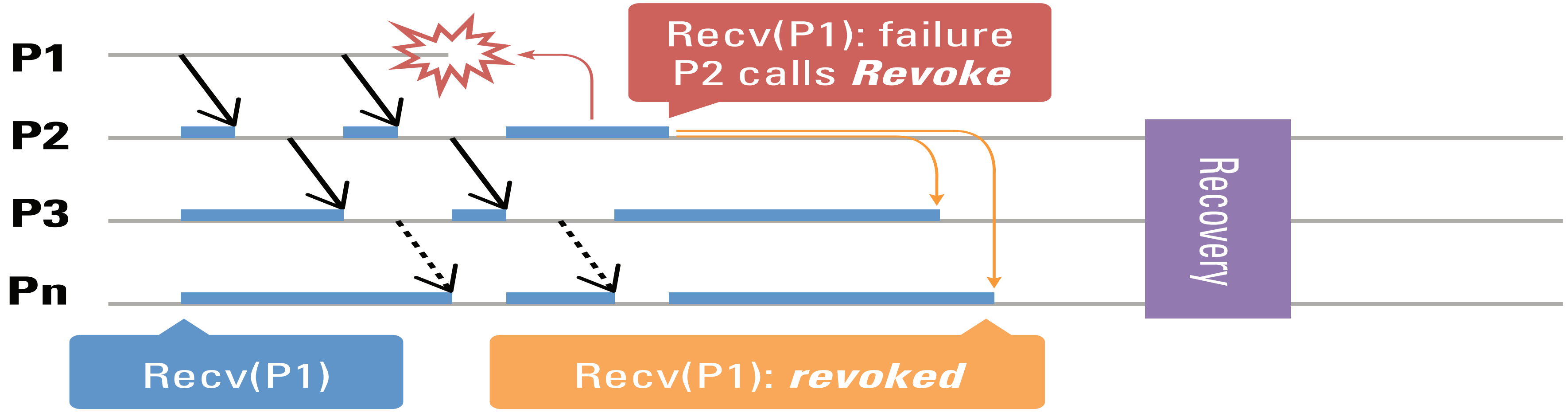
- Notification of failures is local only
  - New error MPI_ERR_PROC_FAILED Raised when a communication with a targeted process fails
- In an operation (collective), some process may succeed while other raise an error
  - Bcast might succeed for the top of the tree, but fail for some subtree rooted on a failed process
- ANY_SOURCE must raise an exception
  - the dead could be the expected sender
  - Raise error MPI_ERR_PROC_FAILED_PENDING, preserve matching order
  - The application can complete the recv later (MPI_COMM_FAILURE_ACK())
- Exceptions indicate an operation failed
  - To know what process failed, apps call MPI_COMM_FAILURE_ACK(), MPI_COMM_FAILURE_GET_ACKED()

App using notification only



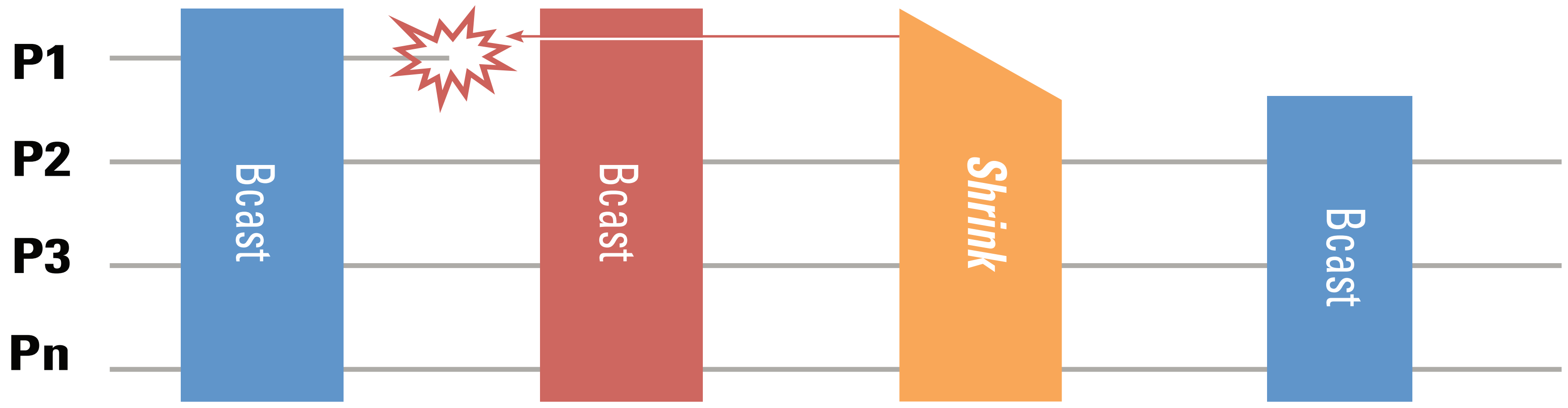
- Error notifications do not break MPI
  - App can continue to communicate on the communicator
  - More errors may be raised if the op cannot complete (typically, most collective ops are expected to fail), but p2p between non-failed processes works
- In this Master-Worker example, we can continue w/o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

App using propagation only



- Application does only p2p communications
- 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 unlocks them with Revoke
- P3..Pn join P2 in the new recovery p2p communication pattern

Error 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

Error Agreement

- When in need to decide if there is a failure and if the condition is recoverable (collectively)
  - MPI_COMM_AGREE(comm, flag)
    - Fault tolerant agreement over boolean flag
    - Unexpected failures (not acknowledged before the call) raise MPI_ERR_PROC_FAILED
    - The flag can be used to compute a user condition, even when there are failures in comm
- Can be used as a global failure detector