An Evaluation of User Level Failure Mitigation Support in MPI

W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, J.J. Dongarra
Innovative Computing Laboratory @ The University of Tennessee
Oak Ridge National Laboratory

Acknowledgements: the MPI Forum members, and especially participants to the MPI-3 FT working group

EuroMPI'12, Wien
Apocalypse?

Figure 1: Modeled application efficiency with and without state machine replication for a 168-hour application, 5-year per-socket MTBF, and 15 minute checkpoint times. Shaded region corresponds to possible socket counts for an exascale class machine [4].

Source: Ferreira et al. SC’11

- Rollback recovery efficiency is threatened at Exascale
  - *We are considering replication* as a serious option, this is disheartening
    - On a sweet-sour note, replication negates *only* ~2yr of Moore’s law…

![Image with Lego figures representing 'NUMA', 'Heterogeneity', 'Jitter', 'Failures']
Issues with Rollback Recovery

- 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:
  - Some cost on communication 😞
  - **Independent process recovery 😊**
    - Non faulty process continue progressing during recovery
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, no loss of completed work
- May require (sometime expensive, like replicates) protection/recovery operations, but still generally more scalable than checkpoint 😊
- Often requires in-depths algorithm rewrite (in contrast to automatic system based C/R) 😞
Algorithm Based Fault Tolerance Performance

- without failure
  - cost of ABFT comes from extra flops (to update checksums)
  - Cost decrease with scale (divided by P when using PxQ processes)
  - Goes to 0% cost at scale
  - Much better than replication, obviously

- Cost of a failure
  - Still competitive with non FT QR (~5% overhead compared to non-FT, with a failure!)
  - Each failure costs ~2% of runtime, bound to decrease with scale

- Productivity cost
  - Application code is heavily modified (as deep as the algorithmic level)
Algorithm Based Fault Tolerance Performance

- without failure
  - cost of ABFT comes from extra flops (to update checksums)
  - Cost decrease with scale (divided by P when using PxQ processes)
  - Goes to 0% cost at scale
  - Much better than replication, obviously

- Cost of a failure
  - Still competitive with non FT QR (~5% overhead compared to non-FT, with a failure!)
  - Each failure costs ~2% of runtime
  - Bound to decrease with scale

- Productivity cost
  - Application code is heavily modified (as deep as the algorithmic level)
The FT techniques spectrum

- All these FT methods **suppose that MPI continues to operate across failures**
  - Advanced Rollback Recovery *(uncoordinated checkpoint)*
  - Forward Recovery:
    - Master-Worker with simple resubmission
    - Iterative methods, Naturally fault tolerant algorithms
    - Algorithm Based Fault Tolerance
    - Replication *(the only system level Forward Recovery)*

---

**Uncoordinated checkpoint (with non-blocking, incremental checkpoints)**

---

**ABFT**

- Factorization in previous iterations
- Trailing matrix & protection update by applying the same operations
- Protection blocks
MPI-2: Fault Tolerance support

- We have the algorithms, but...
  - MPI support of FT is non-existent
  - Prevents effective deployment of ABFT

- **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.”
Non standard extensions for Fault Tolerant MPI

• Example: FT-MPI - a fault tolerant MPI implementation (designed ~2000)
  • Proven research tool
    • Effective support of Forward Recovery, many success stories
      • ABFT dense Linear Algebra
      • CG, Iterative methods, etc.
    • …but…
  • Supports only TCP, does not support modern networks
  • Not installed by default on your system, may not compile on your flavor of HPC system
• Non standard, FT-MPI adapted code is not portable
Requirements for MPI standardization of FT

- **Expressive**, simple to use
  - Enable users to port their code simply, support a variety of FT models and approaches
  - Support legacy code, **backward compatible**
- **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 interfaces
Fault Tolerant MPI: the ULFM proposition for standardization

- 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
Example with MPI\_ANY\_SOURCE: Master-worker

```c
MPI_Irecv_init(comm, ANY\_SOURCE, work\_done)

While(more\_work && workers) {
    submit\_work(worker[i++ % workers])
    rc = MPI\_Test(work\_done)
    if(MPI\_SUCCESS != RC)
    {
        MPI\_COMM\_FAILURE\_ACK(comm)
        MPI\_COMM\_FAILURE\_GET\_ACKED(comm, i)
        worker[i] = worker[workers--]
        resubmit\_work(worker[i], i)
    }
}
```
Fault Tolerant MPI: the ULFM proposition for standardization

- 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 Comm_Revoke resolves inconsistencies introduced by failures
Summary of new functions and impact on implementation

- **MPI_Comm_failure_ack** *(comm)*
  - Resumes matching for MPI_ANY_SOURCE
  - Implementation: easy
  - Requirement: memory linear with number of failures + 1 index per communicator
- **MPI_Comm_failure_get_acked** *(comm, &group)*
  - Returns to the user the group of processes acknowledged to have failed
  - Implementation: easy
  - Requirement: none
- **MPI_Comm_revoke** *(comm)*
  - Non-collective, interrupts all operations on comm (future or active, at all ranks) by raising MPI_ERR_REVOKED
  - Implementation: moderate/difficult
  - Requirement: reliable broadcast, modification in the matching logic, supplementary condition on the latency path, one extra boolean per communicator
- **MPI_Comm_shrink** *(comm, &newcomm)*
  - Collective, creates a new communicator without failed processes (identical at all ranks)
  - Implementation: moderate
  - Requirement: agreement algorithm (many available in the literature)
- **MPI_Comm_agree** *(comm, &bool)*
  - Agree on the AND value on bool, ignoring failed processes (reliable AllReduce)
  - Implementation: moderate (but identical to the agreement in shrink)
  - Requirement: agreement algorithm
Summary of new functions and impact on implementation

- **MPI_Comm_failure_ack**(comm)
  - Resumes matching for MPI_ANY_SOURCE
  - Implementation: easy
  - Requirement: memory linear with number of failures + 1 index per communicator

- **MPI_Comm_failure_get_acked**(comm, &group)
  - Returns to the user the group of processes acknowledged to have failed
  - Implementation: easy
  - Requirement: none

- **MPI_Comm_revoke**(comm)
  - Non-collective, interrupts all operations on comm (future or active, at all ranks) by raising MPI_ERR_REVOKED
  - Implementation: moderate/difficult
  - Requirement: reliable broadcast, modification in the matching logic, supplementary condition on the latency path, one extra boolean per communicator

- **MPI_Comm_shrink**(comm, &newcomm)
  - Collective, creates a new communicator without failed processes (identical at all ranks)
  - Implementation: moderate
  - Requirement: agreement algorithm (many available in the literature)

- **MPI_Comm_agree**(comm, &bool)
  - Agree on the AND value on bool, ignoring failed processes (reliable AllReduce)
  - Implementation: moderate (but identical to the agreement in shrink)
  - Requirement: agreement algorithm
Summary of new functions and impact on implementation

- **MPI_Comm_failure_ack**(comm)
  - Resumes matching for MPI_ANY_SOURCE
  - Implementation: easy
  - Requirement: memory linear with number of failures + 1 index per communicator

- **MPI_Comm_failure_get_acked**(comm, &group)
  - Returns to the user the group of processes acknowledged to have failed
  - Implementation: easy
  - Requirement: none

- **MPI_Comm_revoke**(comm)
  - Non-collective, interrupts all operations on comm (future or active, at all ranks) by raising MPI_ERR_REVOKED
  - Implementation: moderate/difficult
  - Requirement: reliable broadcast, modification in the matching logic, supplementary condition on the latency path, one extra boolean per communicator

- **MPI_Comm_shrink**(comm, &newcomm)
  - Collective, creates a new communicator without failed processes (identical at all ranks)
  - Implementation: moderate
  - Requirement: agreement algorithm (many available in the literature)

- **MPI_Comm_agree**(comm, &bool)
  - Agree on the AND value on bool, ignoring failed processes (reliable AllReduce)
  - Implementation: moderate (but identical to the agreement in shrink)
  - Requirement: agreement algorithm
ULFM Performance in Open MPI

- Code complexity is limited
  - New features are contained: minor changes to the existing MPI layer
  - more changes in the runtime (to avoid abort behavior)
- No performance difference measured on micro-benchmarks
  - Point-to-point latency/bandwidth unchanged
    - Even on shared-memory
  - Collective communications unchanged
    - Code unchanged
    - Overhead below standard deviation
    - Enabling FT may even increase performance on some runs, wohoo!

### Table 1

<table>
<thead>
<tr>
<th>Interconnect</th>
<th>Vanilla</th>
<th>Std. Dev.</th>
<th>Enabled</th>
<th>Std. Dev.</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>10.2564</td>
<td>0.0946</td>
<td>10.2776</td>
<td>0.1065</td>
<td>0.0212</td>
</tr>
<tr>
<td>OpenIB</td>
<td>4.9637</td>
<td>0.0018</td>
<td>4.9650</td>
<td>0.0022</td>
<td>0.0013</td>
</tr>
</tbody>
</table>

### Table 2

<table>
<thead>
<tr>
<th>Interconnect</th>
<th>Vanilla</th>
<th>Std. Dev.</th>
<th>Enabled</th>
<th>Std. Dev.</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>6,311.38</td>
<td>14.42</td>
<td>6,302.75</td>
<td>10.72</td>
<td>-8.63</td>
</tr>
<tr>
<td>OpenIB</td>
<td>9,688.85</td>
<td>3.29</td>
<td>9,689.13</td>
<td>3.77</td>
<td>0.28</td>
</tr>
</tbody>
</table>
ULFM Performance in Open MPI

- Application performance
  - Sequoia AMG: Multigrid solver for unstructured mesh
  - Application is not fault tolerant
  - Running FT/No FT MPI
  - No measured performance difference in any of the computing blocks

Fig. 2. Comparison of the vanilla and ULFM versions of Open MPI running Sequoia-AMG at different scales (Smoky).
ULFM Recovery performance

- Benchmark loops Barrier
- Failures are injected at random ranks
- We measure
  - Time for Barrier to return PROC_FAILED or REVOKED
  - Time to do MPI_Revoke
  - Time to repair the communicator with MPI_Shrink

To measure the impact of the prototype on a real application, we used the Sequoia AMG benchmark\(^6\). This MPI intensive benchmark is an Algebraic Mult-Grid (AMG) linear system solver for unstructured mesh physics. A weak scaling study was conducted up to 512 processes following the problem Set 5. In Figure 2, we compare the time slicing of three main phases (Solve, Setup, and SStruct) of the benchmark, with, side by side, the vanilla version of the Open MPI implementation, and the ULFM enabled one. The application itself is not fault tolerant and does not use the features proposed in ULFM. The goal of this benchmark is to demonstrate that a careful implementation of the proposed semantic does not impact the performance of the MPI implementation, and ultimately leaves the behavior and performance of legacy applications unchanged. The results show that the performance difference is negligible.

![Fig. 2. Comparison of the vanilla and ULFM versions of Open MPI running Sequoia-AMG at different scales (Smoky).](image)

To assess the overheads of recovery constructs, we developed a synthetic benchmark that mimics the behavior of a typical fixed-size tightly-coupled fault-tolerant application. Unlike a normal application it performs an infinite loop, where each iteration contains a failure and the corresponding recovery procedure. Each iteration consists of 5 phases: in the first phase (Detection), all processes but a designated victim enter a Barrier on the intracommunicator. The victim dies, and the failure detection mechanism makes all surviving processes exit the Barrier, some with an error code. In Phase 2 (Revoke), the surviving processes that detected a process-failure related error during the previous phase invoke the new construct `MPI_COMM_REVOKE`. Then they proceed to Phase 3 (Shrink) where the intracommunicator is shrunk using `MPI_COMM_SHRINK`. The two other phases serve to repair a full-size intracommunicator using spawn and intercommunicator merge operations to allow the benchmark to proceed to the next round. In Figure 3, we present the timing of each phase, averaged upon 50 iterations of the benchmark loop, for a varying number of processes on the Smoky machine. We focus on the three points related to ULFM: failure detection, revoke and shrink. The failure detection is mildly impacted by the scale. In the prototype\(^6\) https://asc.llnl.gov/sequoia/benchmarks/#amg

![Fig. 3. Evaluation of the Fault Injection Benchmark with full recovery at different scales (Smoky).](image)
Conclusion

• Many FT approaches depend on MPI to continue operating after a failure!
  • Master-worker, Naturally fault tolerant, iterative methods can benefit for free!
  • ABFT methods have great scaling potential but are more intrusive
  • Not a silver bullet, each algorithm/application has to care for FT
  • Even automatic approaches (replication, uncoordinated checkpoint) require it
  • MPI-3.0 released, but the Forum will continue to work on FT for inclusion in MPI-next standard revision
• User Level Failure Mitigation: a proposal for standardized MPI API changes to support FT
  • Existing MPI functions unchanged (reuse old definitions of MPI_ERR_RETURNS exception handlers)
  • Code complexity is small, implementation is reasonable
  • Impact on performance is not measurable
• Ongoing/Future works
  • Porting varied applications to the proposal (SPMD, master-worker, ABFT, etc.)
    • Ongoing efforts: Wang Landau Polymer Collapse (Appl. Checkpoint + independent multiple communicator recovery)
  • Support higher level FT Models (uncoordinated C/R, transactions, etc.)
Real Example: Wang Landau Polymer Freezing and Collapse

- Collaboration with UGA
- Energy range (temperature) sliced
- Groups of processors focus on an energy window
  - Periodical Allgather inside groups
- Energy window overlap
- Periodically, energy window exchange condition
- Repeat until convergence
Real Example: Wang Landau Polymer Freezing and Collapse

- Collaboration with UGA
- Energy range (temperature) sliced
- Groups of processors focus on an energy window
  - Periodical Allgather inside groups
- Energy window overlap
- Periodically, energy window exchange condition
- Repeat until convergence
Real Example: Wang Landau Polymer Freezing and Collapse

- 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
Real Example: Wang Landau Polymer Freezing and Collapse

- 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
Real Example: Wang Landau Polymer Freezing and Collapse

- 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
Real Example: Wang Landau Polymer Freezing and Collapse

- 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
```