Fault Tolerant Programming Abstractions and Failure Recovery Models for MPI Applications

Ignacio Laguna
Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

We use MPI workloads to design future machines

CORAL tier-1 benchmarks use MPI
CORAL is the recent DOE procurement to deliver next-generation (petaflops) supercomputers

75%

MPI is widely cited
Hits are returned by Google Scholar for the term “message passing interface”

46,600

Many implementations are available
C/C++, Java, Matlab, Python, R, ...

MPI+X will remain a common programming model
MOST NODE/PROCESS FAILURES SHOW UP IN MPI

MPI is the dominant “glue” for HPC applications

Examples:
- Application error (bug)
- Hardware error (soft error)
From the MPI standard:

- “... after an error is detected, the state of MPI is undefined”
- “MPI itself provides no mechanisms for handling processor failures.”

MPI doesn’t provide guaranties about failure detection and/or notifications

Resource manager kills the job (by default)
WHY TO INVEST IN FAULT TOLERANCE IN MPI?

1. MPI will continue to be used
2. Nice layer to detect failures
3. No resilience abstractions in the standard

Solution?
PUZZLE PIECES OF THE PROBLEM

Roadmap of the talk

1. Problem Description
   - Why adding FT to MPI is difficult?
   - Challenges & areas of concern

2. Approaches
   - Current solutions to the problem
   - Proposals in the MPI forum

3. Experimental Evaluation
   - Modeling & simulation
   - Early evaluation results

4. Lessons Learned
   - Where do we go from here?
   - Summary
The devil is on the details…

**FIXING A FAILED MPI RANK TRANSPARENTLY IS HARD**

Ideal fault-tolerance strategy:

*Replace transparently a failed process*

Some implementation questions / considerations:

1. How to bring a new MPI process up-to-date?
2. How to handle in-transit messages and operations?
3. Where to re-inject control in the application?

*This is difficult to implement correctly and efficiently in MPI libraries*
MOST CODES ASSUME NO ERROR CHECKING
Reasoning about error propagation in a complex code is hard

**Ideal world**

```
for (...) {
    err = MPI_Isend();
    if (err) recover();
}
for (...) {
    err = MPI_Irecv();
    if (err) recover();
}
err = MPI_Waitall();
if (err) recover();
err = MPI_Barrier();
if (err) recover();
```

**Real world**

```
for (...) {
    MPI_Isend();
    for (...) {
        MPI_Irecv();
        if (err) recover();
    }
}
MPI_Waitall();
MPI_Barrier();
```

MPI programs don’t check for errors
Fault detection that rely on error codes would be hard to use

*Most codes will recover from failures via checkpoint/restart*
OPEN CHALLENGES AND QUESTIONS

What failures to consider in the MPI standard?
- Node / process failures?
- Communication errors?
- Silent errors?

Should the application continue executing after a failure? How?
- Forward vs. backward recovery

Fault-tolerant APIs that don’t involve much code changes

Should fault tolerance be provided as a library?
PUZZLE PIECES OF THE PROBLEM
Roadmap of the talk

1 Problem Description
- Why adding FT to MPI is difficult?
- Challenges & areas of concern

2 Approaches
- Current solutions to the problem
- Proposals in the MPI forum

3 Experimental Evaluation
- Modeling & simulation
- Early evaluation results

4 Lessons Learned
- Where do we go from here?
- Summary
POSSIBLE SOLUTIONS TO THE PROBLEM
Resilient programming abstractions for MPI

1. ULFM: User level failure mitigation
   Local shrinking recovery strategy

2. Reinit interface
   Global non-shrinking recovery strategy

3. Fault tolerant libraries
   e.g., Local Failure Local Recovery (LFLR)

4. ?
ULFM: USER LEVEL FAILURE MITIGATION
Current proposal for MPI 4.0

Shrinking recovery strategy

Shrinking recovery: the available resources after a failure are shrunk or reduced

Focus on process failures
- Communication that involves a failed process would fail

Communicators can be revoked
- Enables fault propagation

Communicators can be shrunk
- Code must create new communicators with fewer processes

New error codes:
MPI_ERR_PROC_FAILED

New MPI calls:
- MPI_COMM_REVOKE
- MPI_COMM_SHRINK
- MPI_COMM_AGREE
- MPI_COMM_FAILURE_ACK
PROS AND CONS OF ULFM

Works well for master-slave codes
- Only few processes need to know of a failure

Difficult to use in bulk synchronous codes
- All processes need to know of failures (global recovery)
- Codes must rollback to a previous checkpoint

Most codes cannot handle shrinking recovery
- Cannot re-decompose problem in fewer processes
- Requires load balancing

Master-slave
- Some may rollback

Bulk synchronous
- Everyone must rollback
With ULFM, faults are “eventually” delivered to the application.

Global recovery avoids this issue—all processes roll back to a known safe state.
REINIT INTERFACE
Global non-shrinking recovery strategy

MPI_Init();
MPI_Reinit();
MPI_Error_handlers();
for (...)  
  MPI_Isend();
for (...)  
  MPI_Irecv();
MPI_Waitall();
MPI_BARRIER();
MPI_Finalize();

MPI library performs:
• Failure detection
• Failure notification
• Code specifies cleanup functions
• Emulates exception handling

Stack of error handlers
Error handler 1
Error handler 2
Error handler 3

Advantages
• Job is not killed
• Faster checkpoint/restart

Disadvantages
• Difficult to clean up state of multithreaded code (OpenMP)
• Won’t work if application’s initialization takes too much time
Approach: use ULFM’s functionality to provide fault tolerance as a library

Example:  *Local Failure Local Recovery (LFLR)*

<table>
<thead>
<tr>
<th>Rank 0</th>
<th>Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Rank N</td>
<td>Run</td>
</tr>
<tr>
<td>Rank N+1</td>
<td>Stand by</td>
</tr>
</tbody>
</table>


**Advantages**

- Handles fault tolerance transparently

**Disadvantages**

- Applications cannot use other tools / libraries
- Inherits any performance issues and/or bottlenecks from ULFM
POSSIBLE SOLUTIONS TO THE PROBLEM
Resilient programming abstractions for MPI

1. ULFM: User level failure mitigation
   Local shrinking recovery strategy

2. Reinit interface
   Global non-shrinking recovery strategy

3. Fault tolerant libraries
   e.g., Local Failure Local Recovery (LFLR)

4. Don’t integrate fault tolerance into MPI
   Rely in Checkpoint/Restart
1 Problem Description
• Why adding FT to MPI is difficult?
• Challenges & areas of concern

2 Approaches
• Current solutions to the problem
• Proposals in the MPI forum

3 Experimental Evaluation
• Modeling & simulation
• Early evaluation results

4 Lessons Learned
• Where do we go from here?
• Summary
TESTBED APPLICATION: ddcMD

Scalable molecular dynamics application
- Not a proxy / mini / benchmark code

Problem can be decomposed onto any number of processes

Includes load balancing

Uses a few communicators
- Simplifies implementing shrinking recovery
- We have to shrink only one communicator (MPI_COMM_SHRINK)
ELIMINATING A PROCESS FROM A COMMUNICATOR TAKES TOO MUCH TIME

Time to shrink MPI_COMM_WORLD when a process fails

Open MPI 1.7, Sierra cluster at LLNL (InfiniBand)
SHRINKING RECOVERY IS ONLY USEFUL IN SOME CASES
Most codes will use non-shrinking recovery at large scale

Shrinking recovery only works when:
- Application can balance loads quickly after failures
- System experiences high failure rates
- Application can re-decompose problem on fewer processes/nodes

Most codes/systems don’t have these capabilities
Prototype *Reinit* in Open MPI

Tests on Cray XC30 system (BTL network)

Applications:
- Lattice Bolzmann transport code (LBMv3)
- Molecular dynamics code (ddcMD)

Recovery time is reduced compared to traditional job restarts.

Time to recover from a failure using *Reinit* versus a standard job restart.

**Insight**

With Reinit, we believe that data of recent checkpoints is likely cached in the filesystem buffers since the job is not killed.
Roadmap of the talk

1 Problem Description
- Why adding FT to MPI is difficult?
- Challenges & areas of concern

2 Approaches
- Current solutions to the problem
- Proposals in the MPI forum

3 Experimental Evaluation
- Modeling & simulation
- Early evaluation results

4 Lessons Learned
- Where do we go from here?
- Summary
SOME LESSONS LEARNED

- The MPI community should evaluate carefully the pros and cons of current fault-tolerant proposals
- It is important to consider a broad range of applications
- Pay special attention to legacy scalable codes (e.g., BSP)
- Viewing the problem only from the system perspective doesn’t work
- We must design interfaces after consulting with several users
FUTURE DIRECTIONS

How do we solve this problem?

…and only then we propose modifications to the MPI standard

1. Evaluate multiple resilient programming abstractions (other than ULFM and Reinit)

2. Test models on a broad range of applications

3. Evaluate not only performance, but also programmability
Acknowledgments
Smart people that contribute to this effort

Martin Schulz, LLNL

David Richards, LLNL

Bronis R. de Supinski, LLNL

Kathryn Mohror, LLNL

Todd Gamblin, LLNL

Howard Pritchard, LANL

Adam Moody, LLNL
Thank you!
ULFM IS SUITABLE ONLY FOR A SUBSET APPLICATIONS
It is hard to use ULFM in bulk synchronous codes

<table>
<thead>
<tr>
<th>Suitable for ULFM (easy to implement with few changes in the application)</th>
<th>Application can “naturally” support this model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shrinking Recovery</td>
<td>ULFM</td>
</tr>
<tr>
<td>Non-shrinking Recovery</td>
<td>ULFM</td>
</tr>
<tr>
<td>Local Recovery</td>
<td>ULFM</td>
</tr>
<tr>
<td>Global Recovery</td>
<td>ULFM</td>
</tr>
<tr>
<td>Backward Recovery</td>
<td>ULFM</td>
</tr>
<tr>
<td>Forward Recovery</td>
<td>ULFM</td>
</tr>
</tbody>
</table>

Applications

<table>
<thead>
<tr>
<th>Bulk synchronous</th>
<th>Master-slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shrinking Recovery</td>
<td>ULFM</td>
</tr>
<tr>
<td>Non-shrinking Recovery</td>
<td>APP</td>
</tr>
<tr>
<td>Local Recovery</td>
<td>ULFM</td>
</tr>
<tr>
<td>Global Recovery</td>
<td>APP</td>
</tr>
<tr>
<td>Backward Recovery</td>
<td>APP</td>
</tr>
<tr>
<td>Forward Recovery</td>
<td>APP</td>
</tr>
</tbody>
</table>

**REINIT SUPPORTS BACKWARD RECOVERY**
In contrast, the focus of ULFM is forward recovery

---

**Backward recovery**
Attempts to restart the application from a previously saved state

**Forward recovery**
Attempts to find a new state from which the application can continue.

---

**Reinit Interface**
- Restart from a checkpoint
- Get “fresh” MPI state

**ULFM**
- Fix communicators and continue
- Attempt to “fix” MPI state