

# Fault Tolerant Programming Abstractions and Failure Recovery Models for MPI Applications

Ignacio Laguna  
Center for Applied Scientific Computing



*Salishan Conference on High-speed Computing, Apr 27-30, 2015*

# MPI IS WIDELY USED, AND WILL CONTINUE TO BE...

We use MPI workloads to design future machines

```
90 static void setup_gather_scatter(void)
91 {
92     int i, j;
93     ptrdiff_t off;
94
95     MPI_Gather(local_ni, rnk, FFTW_MPI_PTRDIFF_T,
96             all_local_ni, rnk, FFTW_MPI_PTRDIFF_T,
97             0, MPI_COMM_WORLD);
98     MPI_Bcast(all_local_ni, rnk*n_pes, FFTW_MPI_PTRDIFF_T, 0, MPI_COMM_WORLD);
99     MPI_Gather(local_starti, rnk, FFTW_MPI_PTRDIFF_T,
100            all_local_starti, rnk, FFTW_MPI_PTRDIFF_T,
101            0, MPI_COMM_WORLD);
102     MPI_Bcast(all_local_starti, rnk*n_pes, FFTW_MPI_PTRDIFF_T, 0, MPI_COMM_WORLD);
103
104     MPI_Gather(local_no, rnk, FFTW_MPI_PTRDIFF_T,
105            all_local_no, rnk, FFTW_MPI_PTRDIFF_T,
106            0, MPI_COMM_WORLD);
107     MPI_Bcast(all_local_no, rnk*n_pes, FFTW_MPI_PTRDIFF_T, 0, MPI_COMM_WORLD);
108     MPI_Gather(local_starto, rnk, FFTW_MPI_PTRDIFF_T,
109            all_local_starto, rnk, FFTW_MPI_PTRDIFF_T,
110            0, MPI_COMM_WORLD);
111     MPI_Bcast(all_local_starto, rnk*n_pes, FFTW_MPI_PTRDIFF_T, 0, MPI_COMM_WORLD);
112
113     off = 0;
114     for (i = 0; i < n_pes; ++i) {
115         ptrdiff_t N = vni;
116         for (j = 0; j < rnk; ++j)
117             N *= all_local_ni[i * rnk + j];
118         isend_cnt[i] = N;
119         isend_off[i] = off;
120         off += N;
121     }
122     intot = off;
123     all_local_in_alloc = 1;
124
125     istrides[rnk - 1] = vni;
126     for (j = rnk - 2; j >= 0; --j)
127         istrides[j] = total_ni[j + 1] * istrides[j + 1];
128 }
```

75%

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

46,600

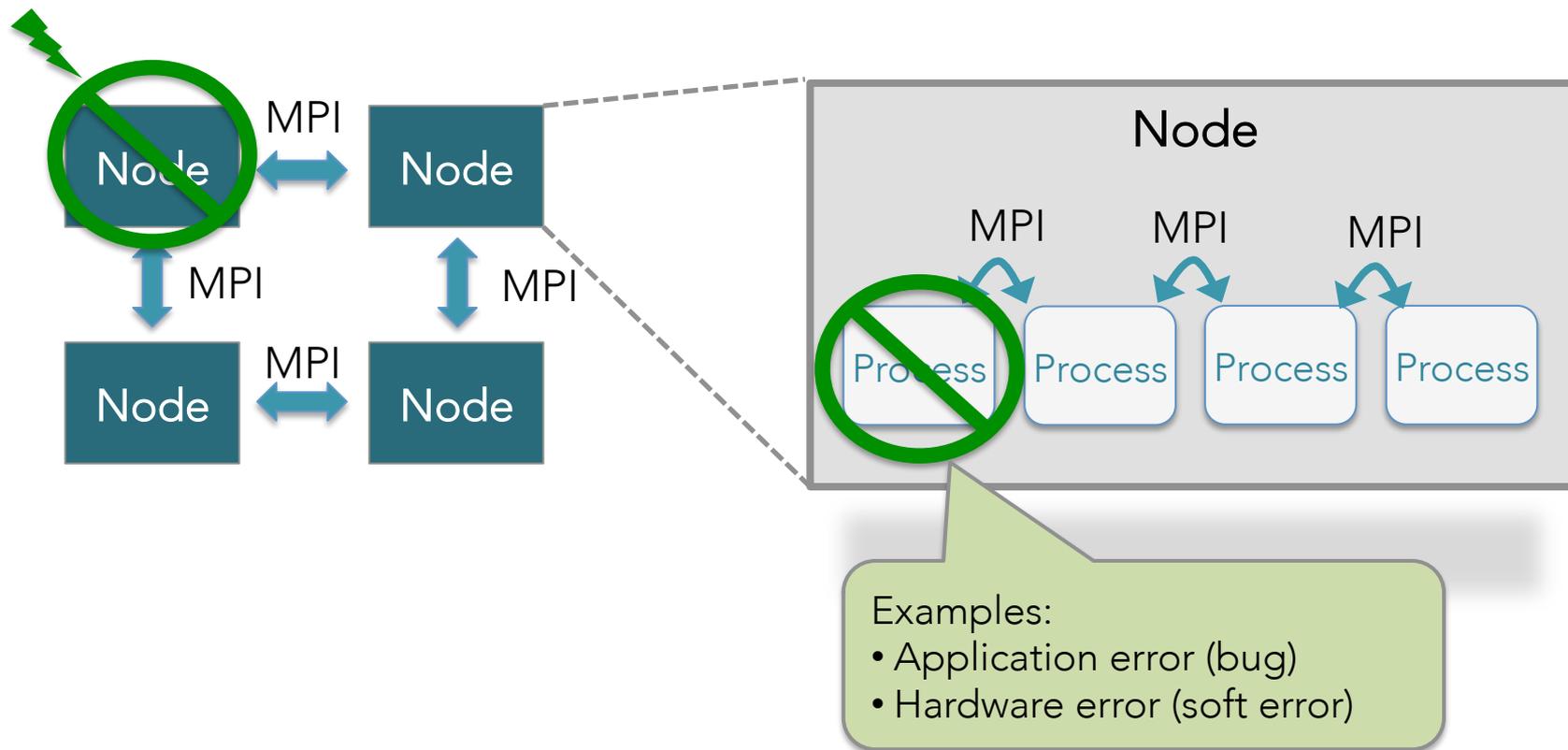
MPI is widely cited  
Hits are returned by Google Scholar for the term "message passing interface"

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

MPI+X will remain a common programming model

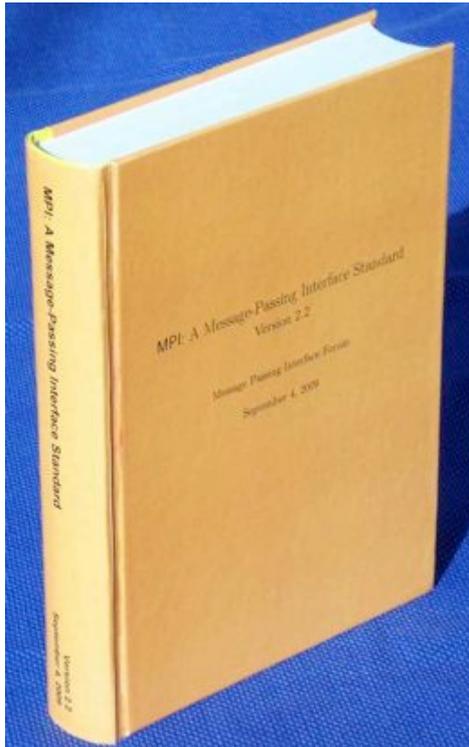
# MOST NODE/PROCESS FAILURES SHOW UP IN MPI

MPI is the dominant "glue" for HPC applications



# MPI DOES NOT PROVIDE FAULT TOLERANCE

Failures are not an option in MPI



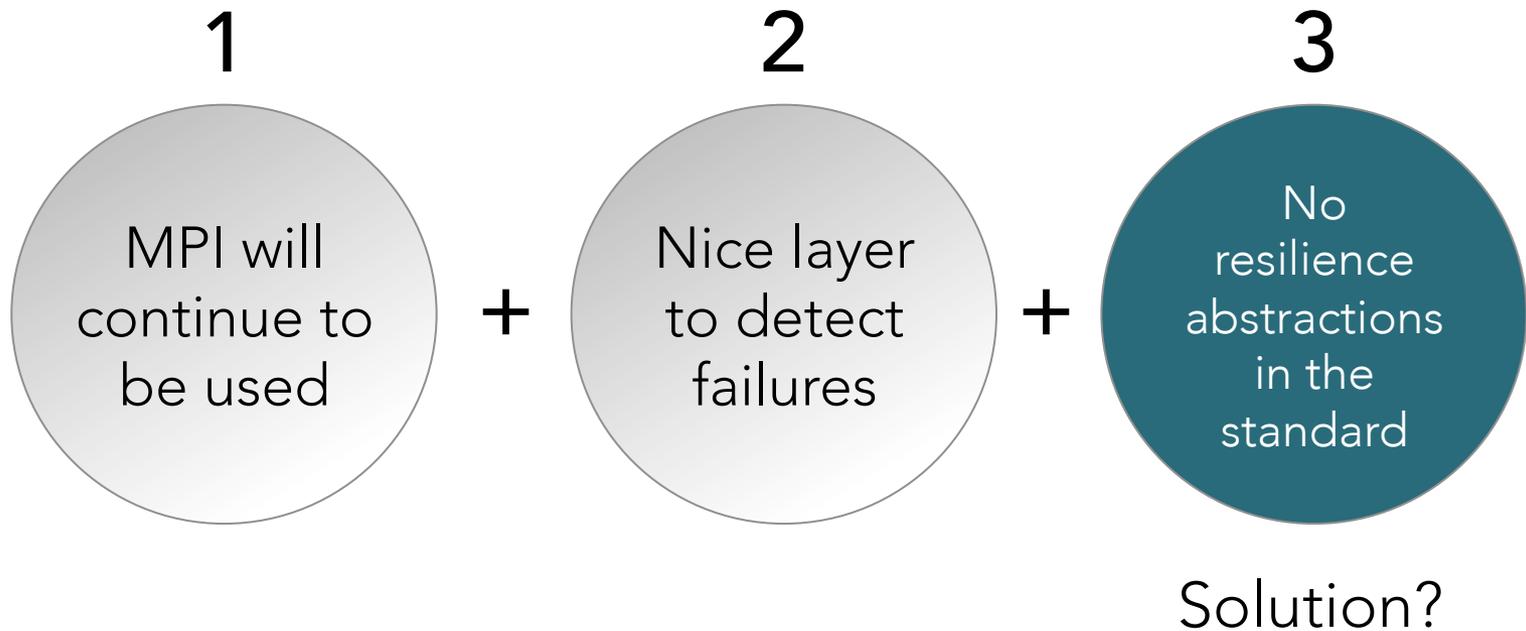
From the MPI standard:

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

MPI doesn't provide guarantees about failure detection and/or notifications

Resource manager kills the job (by default)

# WHY TO INVEST IN FAULT TOLERANCE IN MPI?



# 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

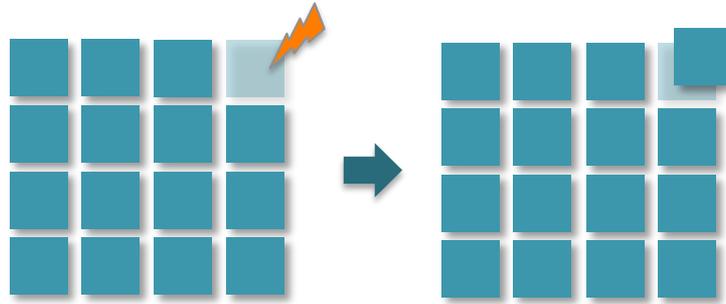


# FIXING A FAILED MPI RANK TRANSPARENTLY IS HARD

The devil is on the details...

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();  
  
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

ULFM: User level failure mitigation  
Local shrinking recovery strategy



*Reinit* interface  
Global non-shrinking recovery strategy



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



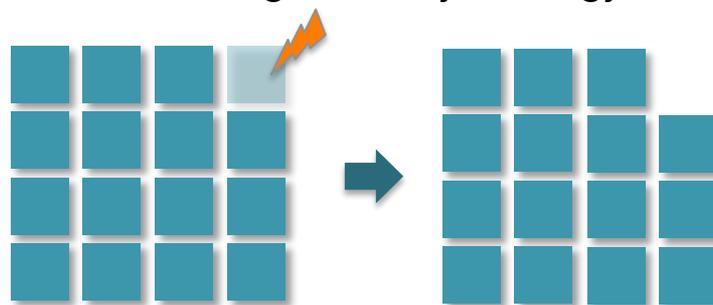
?



# 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

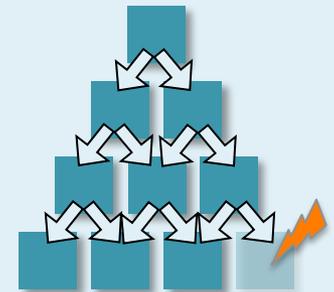
Difficult to use in bulk synchronous codes

- All processes need to know of failures (global recovery)
- Codes must rollback to a previous checkpoint

Most codes cannot handle shrinking recovery

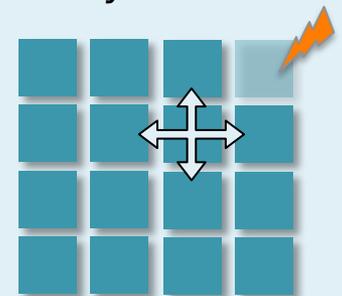
- Cannot re-decompose problem in fewer processes
- Requires load balancing

Master-slave



Some may  
rollback

Bulk synchronous



Everyone must  
rollback

# DELAYED DETECTION IS DIFFICULT TO USE FOR ALGORITHMS THAT USE NON-BLOCKING OPERATIONS

## Data exchange patten

```
for (i=0; i < nsends; ++i) {  
    /* computation */  
    MPI_Isend(...);  
}
```

```
for (i=0; i < nrecvs; ++i) {  
    /* computation */  
    MPI_Irecv(...);  
}
```

```
MPI_Waitall(...);
```

```
/* computation */
```

```
MPI_Barrier(...);
```

 Failure?

 Failure?

 Failure?

Delayed  
detection?

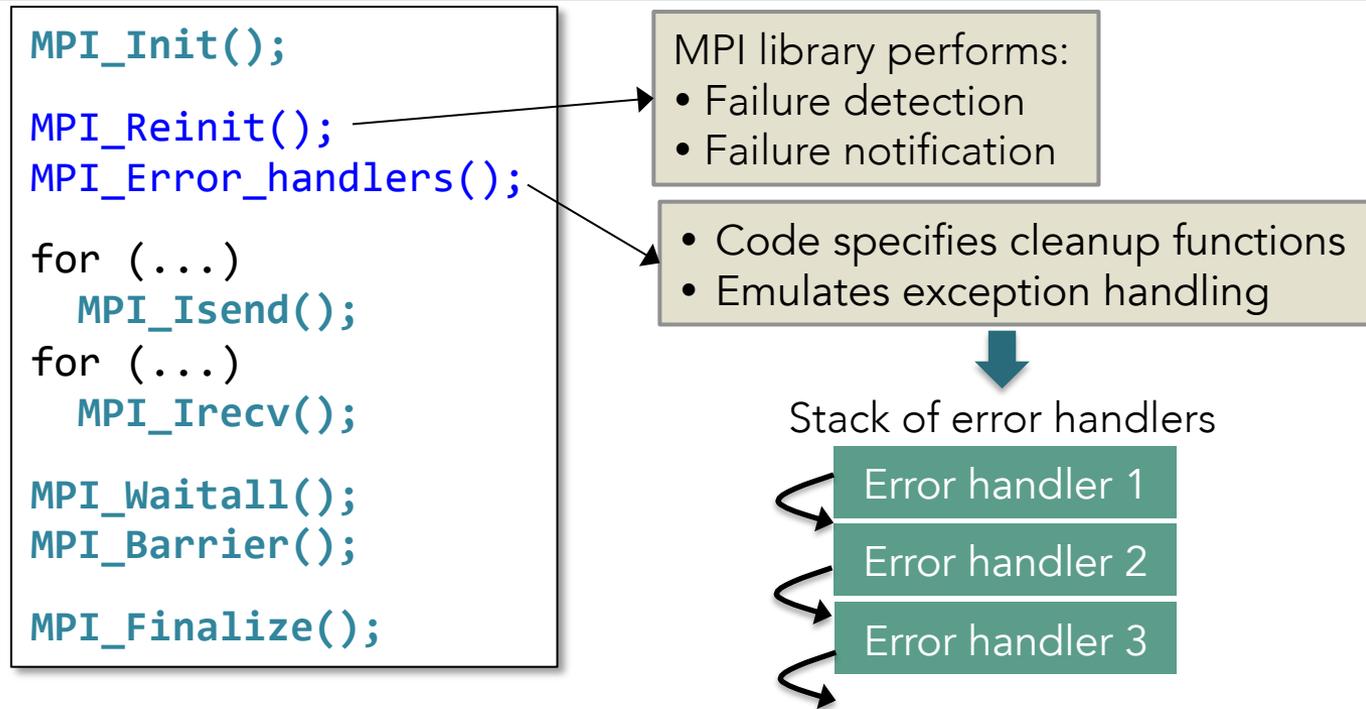
Where in the loop do  
we re-inject control?

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



## Advantages

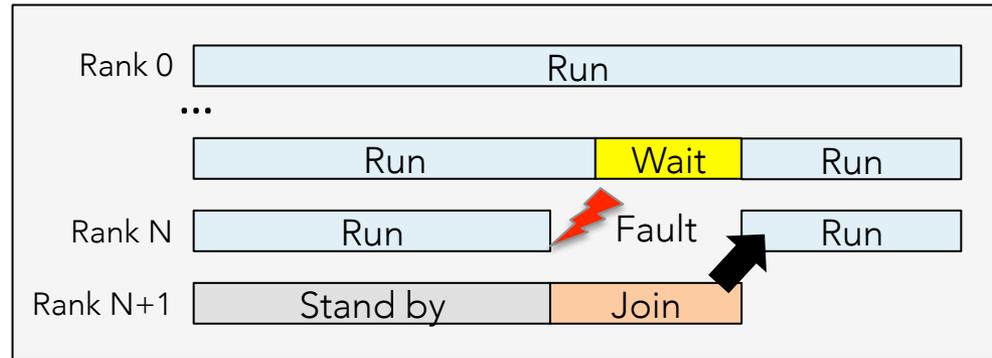
- Job is not killed
- Faster checkpoint/restart

## 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)*



**Reference:** Keita Teranishi and Michael A. Heroux. *Toward Local Failure Local Recovery Resilience Model using MPI-ULFM*, EuroMPI/ASIA '14.

**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

ULFM: User level failure mitigation  
Local shrinking recovery strategy



*Reinit* interface  
Global non-shrinking recovery strategy



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



? Don't integrate fault tolerance into MPI  
Rely in Checkpoint/Restart



# 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

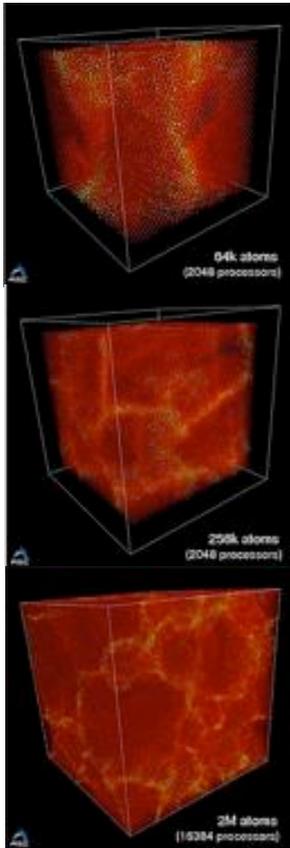
- Modeling & simulation
- Early evaluation results

## 4 Lessons Learned

- Where do we go from here?
- Summary



# TESTBED APPLICATION: ddcMD



Scalable molecular dynamics application

- Not a proxy / mini / benchmark code

Problem can be decomposed onto any number of processes

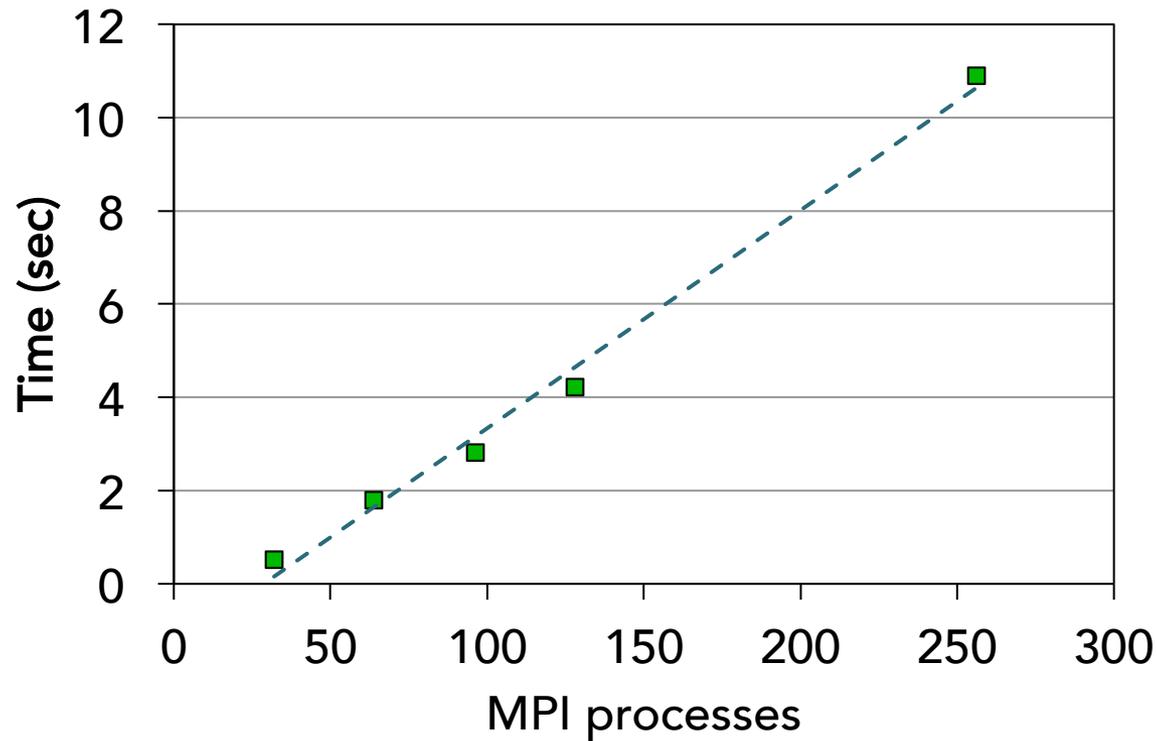
Includes load balancing

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

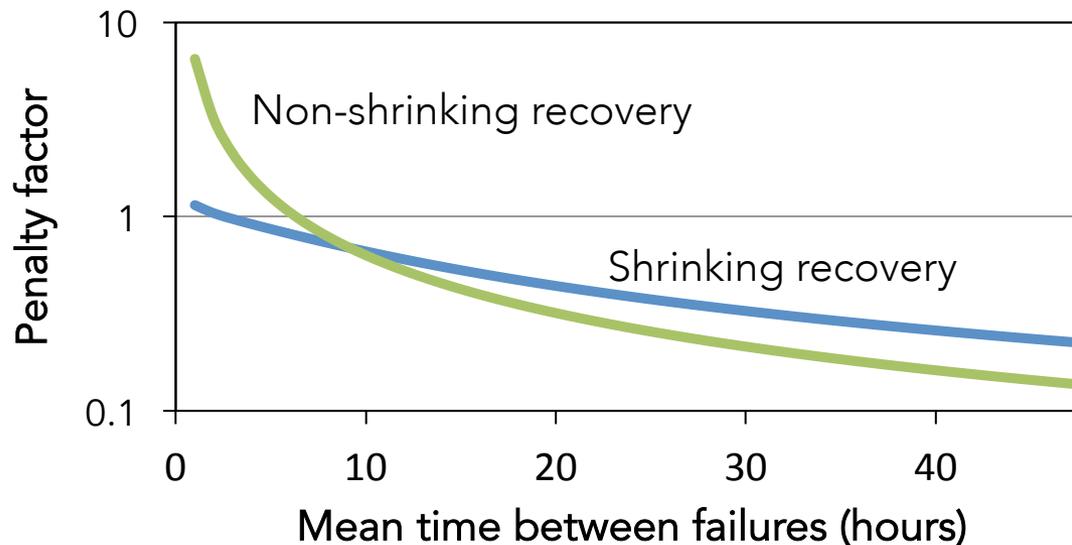
Time to shrink MPI\_COMM\_WORLD when a process fails



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

# REINIT PERFORMANCE MEASUREMENTS ARE PROMISING

Recovery time is reduced compared to traditional job restarts

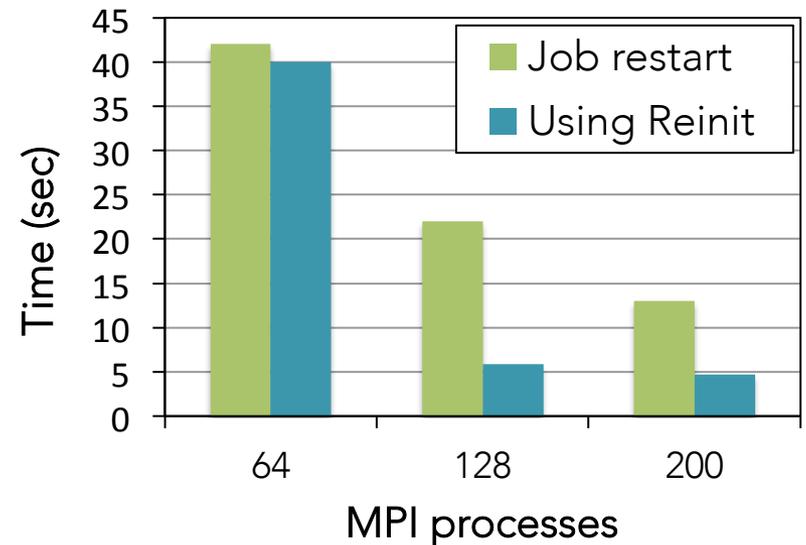
Prototype *Reinit* in Open MPI

Tests on Cray XC30 system (BTL network)

Applications:

- Lattice Boltzmann transport code (LBMv3)
- Molecular dynamics code (ddcMD)

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

# 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

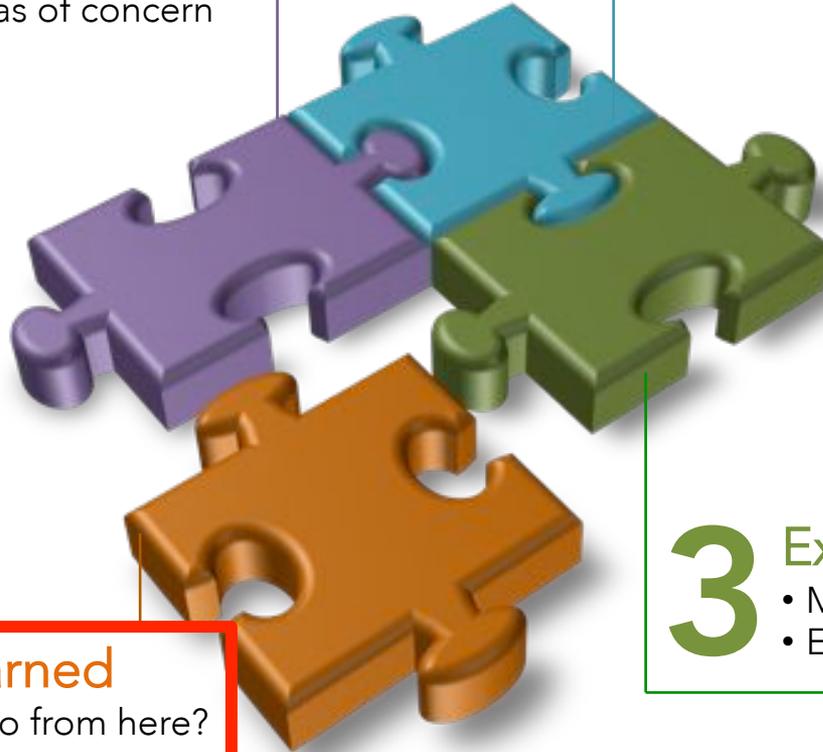
- Current solutions to the problem
- Proposals in the MPI forum

## 3 Experimental Evaluation

- Modeling & simulation
- Early evaluation results

## 4 Lessons Learned

- Where do we go from here?
- Summary

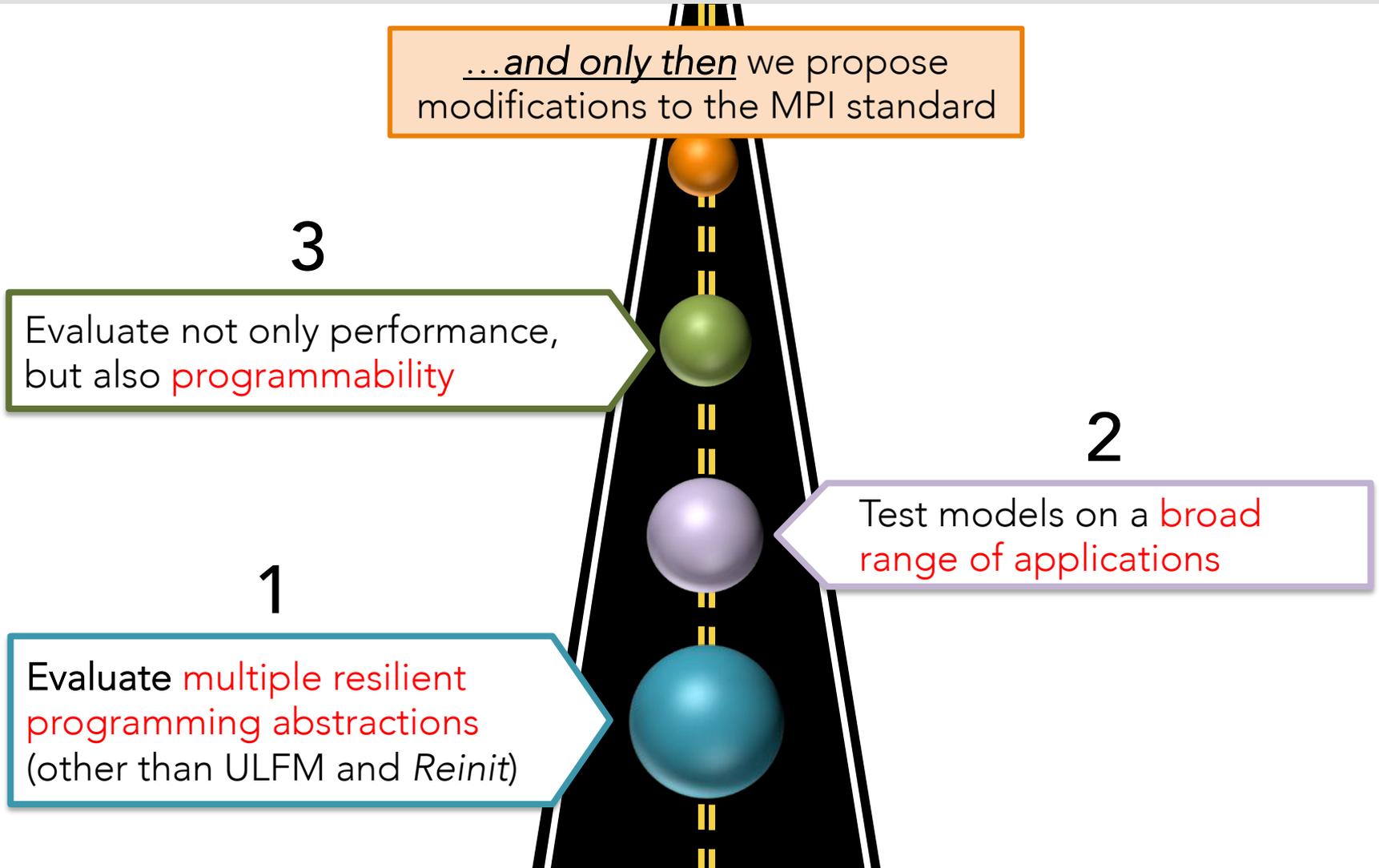


# SOME LESSONS LEARNED

- ❑ 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?



# 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

-  **ULFM** Suitable for ULFM (easy to implement with few changes in the application)
-  **APP** Application can “naturally” support this model

	Applications	
	 Bulk synchronous	Master-slave 
Shrinking Recovery		
Non-shrinking Recovery		
Local Recovery		
Global Recovery		
Backward Recovery		
Forward Recovery		

**Reference:** Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, Bronis R. de Supinski, “Evaluating User-Level Fault Tolerance for MPI Applications”, EuroMPI/ASIA, Kyoto, Japan, Sep 9-12, 2014.

# REINIT SUPPORTS BACKWARD RECOVERY

In contrast, the focus of ULFM is forward recovery



Time

