REINIT: A Simple and Efficient Fault-Tolerance Model for MPI Applications

4th Annual MVAPICH User Group (MUG) Meeting, Aug 15-17, 2016, Columbus, Ohio, USA

Ignacio Laguna
Center for Applied Scientific Computing (CASC)

In collaboration with:

Sourav Chakraborty, Khaled Hamidouche, Hari Subramoni, Dhabaleswar K. (DK) Panda

Murali Emani, Tanzima Islam, Kathryn Mohror, Adam Moody, Kento Sato, Martin Schulz

LLNL-PRES-688119
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Fault-Tolerance Solutions for HPC Applications

Our focus

Application

Examples: MPI FT programming models, ABFT, C/R
+ better view of what needs to be protected/recovered
– can be hard to integrate in large codes

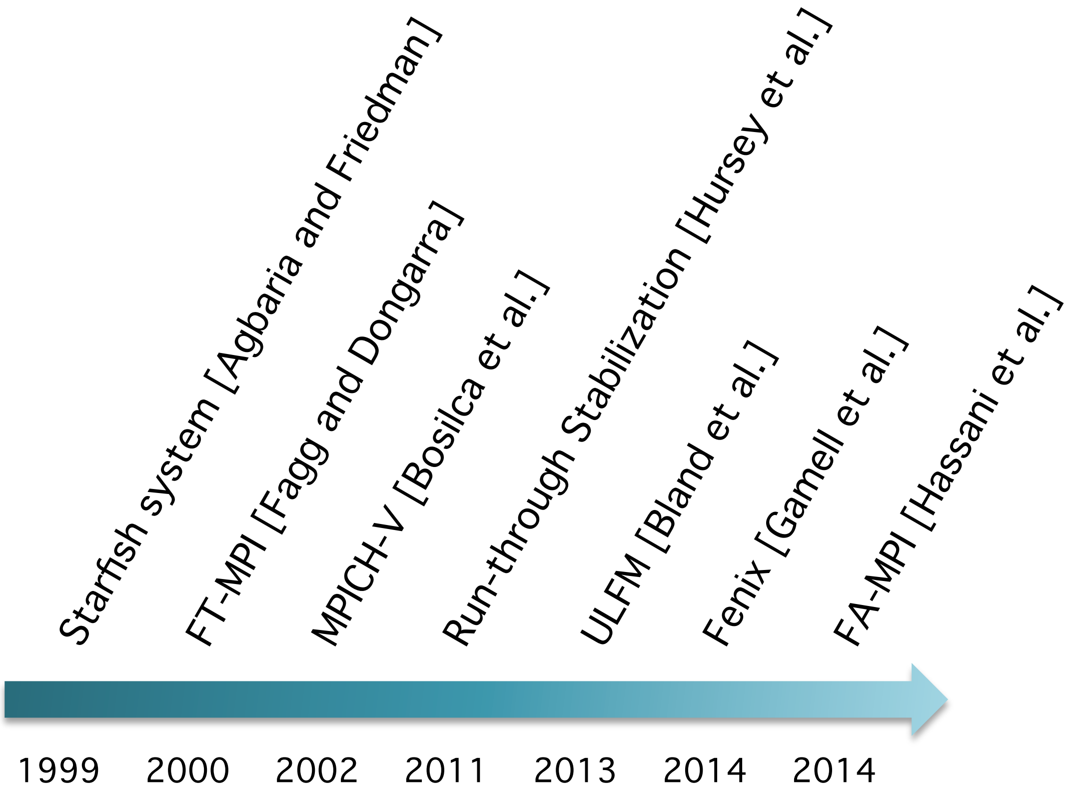
System

Examples: System-level checkpoints, message logging
+ don't require code changes, application agnostic
– overhead can be high, know little about applications

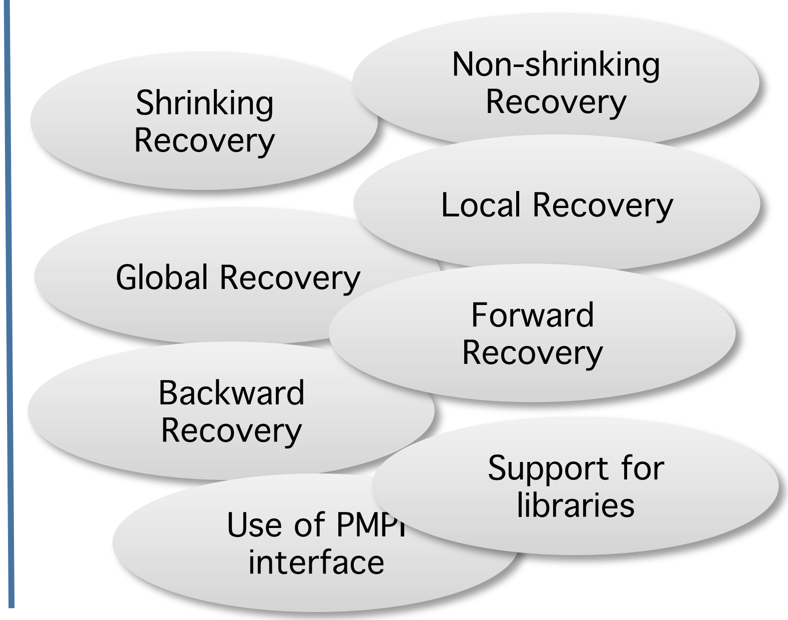
Hardware

Examples: Hardware replication, parity, ECC
+ don't require code changes, system/app agnostic
– can be very expensive (power, performance)

Several FT Programming Models for MPI have been Proposed…



There are many paradigms & features



How do we group or classify them all?

Classification of Existing FT Programming Models

Fine — **Medium** — **Coarse**

Granularity of control / detection →

Error Code Checking

```
err = MPI_Operation1();
if (err) {
  recovery();
}
```

A

Try Blocks

```
TRY {
  MPI_Operation1();
  MPI_Operation2();
}
TRY {
  MPI_Operation3();
  MPI_Operation4();
}
```

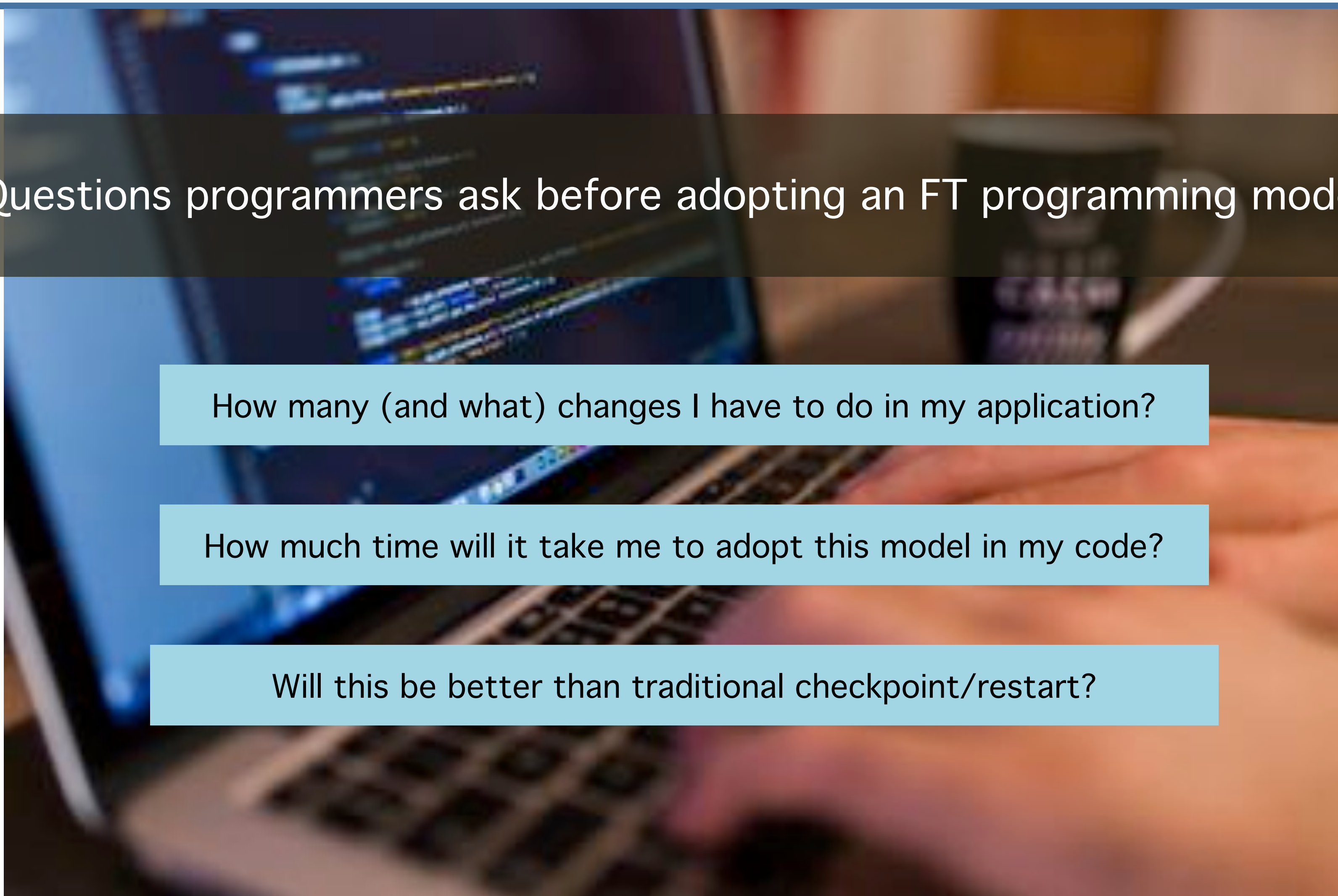
B

Full Restart

```
main () {
  restart_from_here();

  MPI_Operation1();

  MPI_Operation2();

  MPI_Operation3();

  MPI_Operation4();
  return 0;
}
```

C

Programmability and Usability can be Major Concerns



Questions programmers ask before adopting an FT programming model

How many (and what) changes I have to do in my application?

How much time will it take me to adopt this model in my code?

Will this be better than traditional checkpoint/restart?

But....How to Measure Programmability or Usability?

- Lines of code
- Number of files modified (or functions, ...)
- Time spent modifying the code
- ...
- ...
- ...
- Cyclomatic complexity
  - McCabe '76, '89, Gill et al. '91, Lanning et al. '94, Kozlov et al. '08

Cyclomatic Complexity in Software Engineering

- Programs with high complexity have higher rates of bugs
- Programs with high complexity are more difficult to maintain and test
  - Lanning et al., Computer (1994)
  - Kozlov et al., Journal of Software Maintenance and Evolution (2008)
- CC is adopted by the NIST Structured Testing Methodology
  - A. H. Watson, T. J. McCabe, and D. R. Wallace. *Structured testing: A testing methodology using the cyclomatic complexity metric* (1996).

Cyclomatic Complexity Metric

Which code is easier to understand and easier to test?

Code with 10 assignments

```
a = …
b = …
c = …
…
…
…
…
…
…
j = …
```

One execution path

$$CC = 0 + 1 = 1$$

Code with 10 **`if`** conditions

```
if (cond1) {
    if (cond2) {
        if (cond3) {
            …
            …
            …
            …
            …
            …
            if (cond10) {
```

More than 1,000 execution paths!

$$CC = 10 + 1 = 11$$

Cyclomatic complexity (CC) measures number of decisions in a program

$$CC = decisions + 1$$

The recommended value for CC in software engineering and industry is 10

Example:
Cyclomatic Complexity with the Error Code Checking Model

Refinement original loop

```
while(error > threshold) {
  /* perform computation */
  MPI_Allreduce(..., comm);
}
```

Cyclomatic Complexity = 2

Fault-tolerant loop using return error code checking*

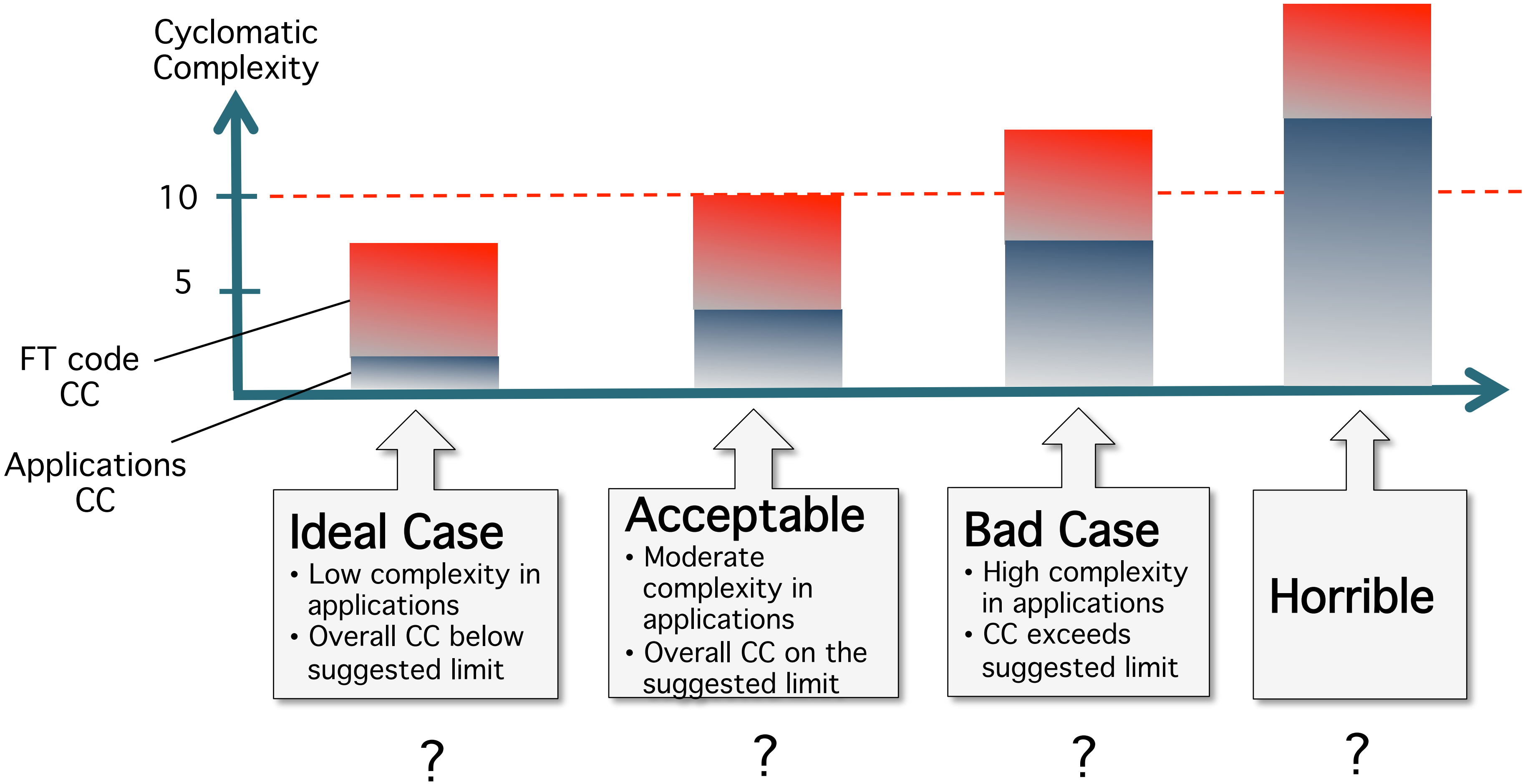
```
while(error > threshold) {
  rc = MPI_Allreduce(..., comm);
  if( (FAILED_PROCESS == rc) ||
      (FAILED_COMMUNICATOR == rc) ||
      (error <= threshold) ) {

    if(FAILED_PROCESS == rc )
      MPI_Comm_revoke(comm);

    allgood = (rc == MPI_SUCCESS);
    rc = MPI_Comm_agree(comm, &allgood);
    if( rc == FAILED_PROCESS ||
        !allsucceeded ) {
      /* repair communicator */
    }
  }
}
```

6 additional conditions

*ULFM example (taken from ULFM documentation)

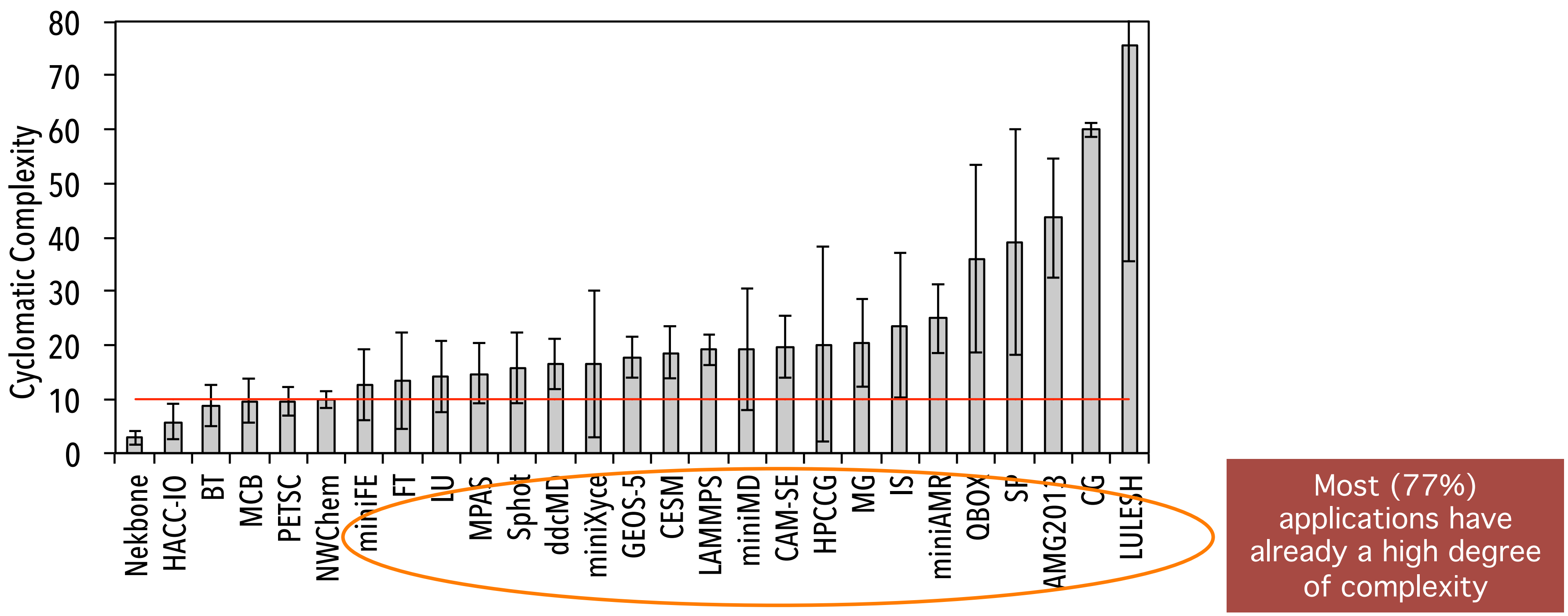
Cyclomatic Complexity = 8

What is the Complexity of MPI Applications?

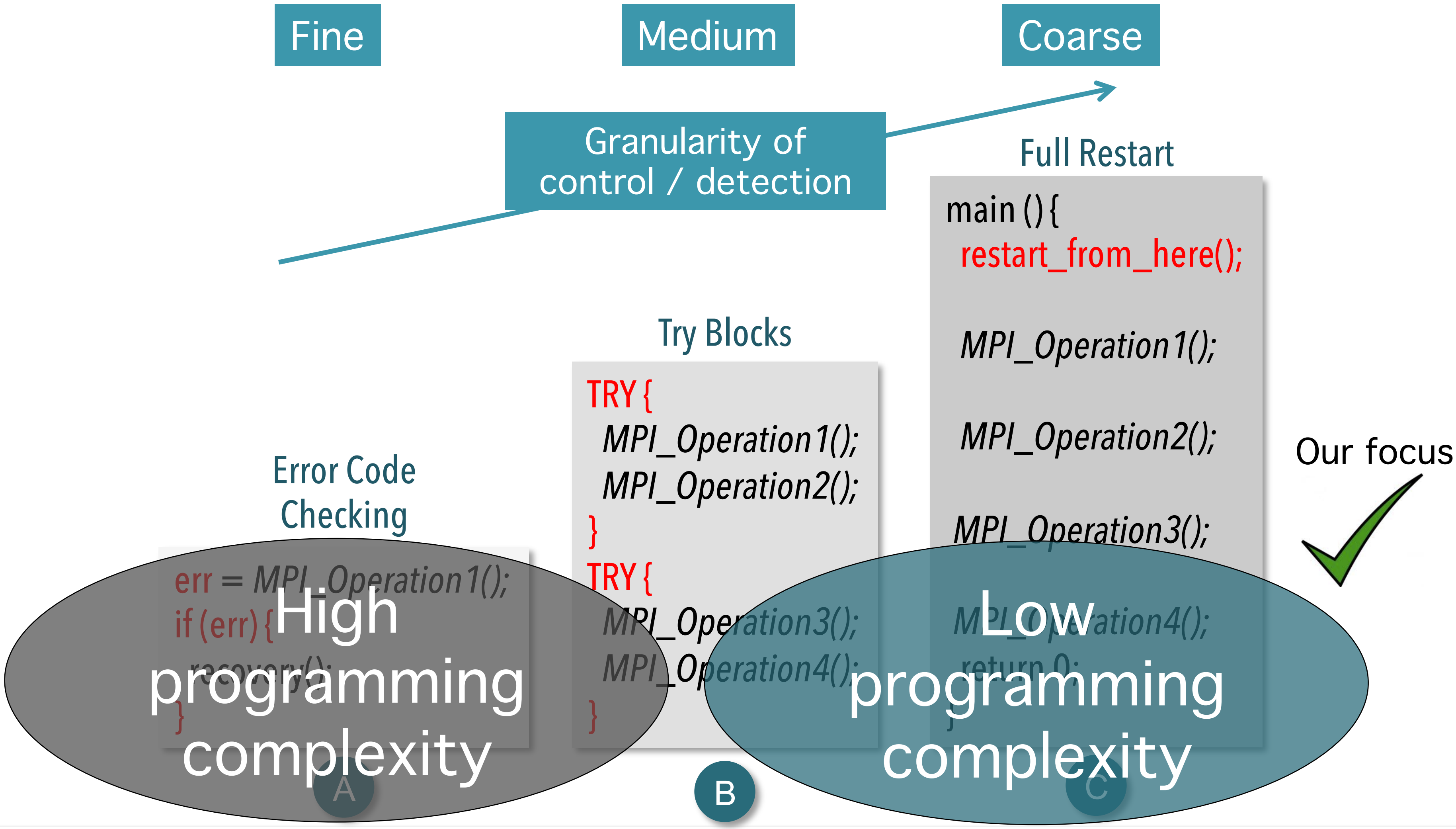


Study of Cyclomatic Complexity of MPI Applications

- Conducted analysis on a large number of MPI applications
- Measured CC of functions that use MPI communication routines
  - Analyzed over 2,300 functions



Our Solution Space: *Low Programming Complexity*



Design Goals of the Reinit Interface

1. Simple to program interface
- Support current fault-tolerance programming practices
- Checkpoint/Restart

2. MPI library cleans up its state (not the application)
- Provide state similar to MPI_Init
- All communicators are gone (except MPI_COMM_WORLD)

3. Close interaction between MPI & resource manager
- More efficient reparation of failed resources
- Faster recovery time

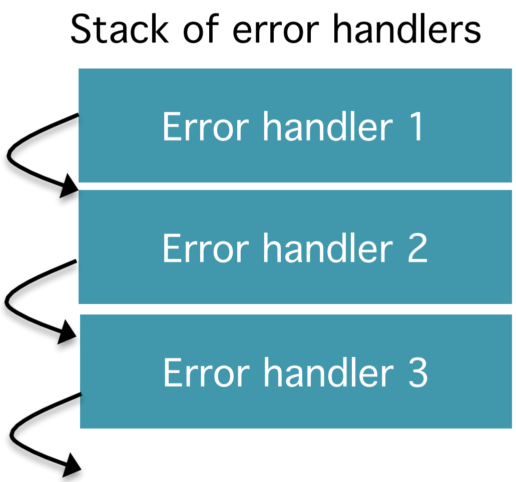
4. Mechanism to clean up libraries
- FIFO stack of error handlers
- Libraries and applications provide their own handlers

Description of the Reinit Interface

```c
/* Initialization routines */
typedef enum {
    MPI_START_NEW,          // Fresh process
    MPI_START_RESTARTED,    // Restarted after fault
    MPI_START_ADDED         // Replaced process
} MPI_Start_state;

/* Application entry point */
typedef void (*MPI_Restart_point)
    (int argc, char **argv, MPI_Start_state state);

int MPI_Reinit
    (int argc, char **argv, MPI_Restart_point point);
```

Cleanup Stack Mechanisms

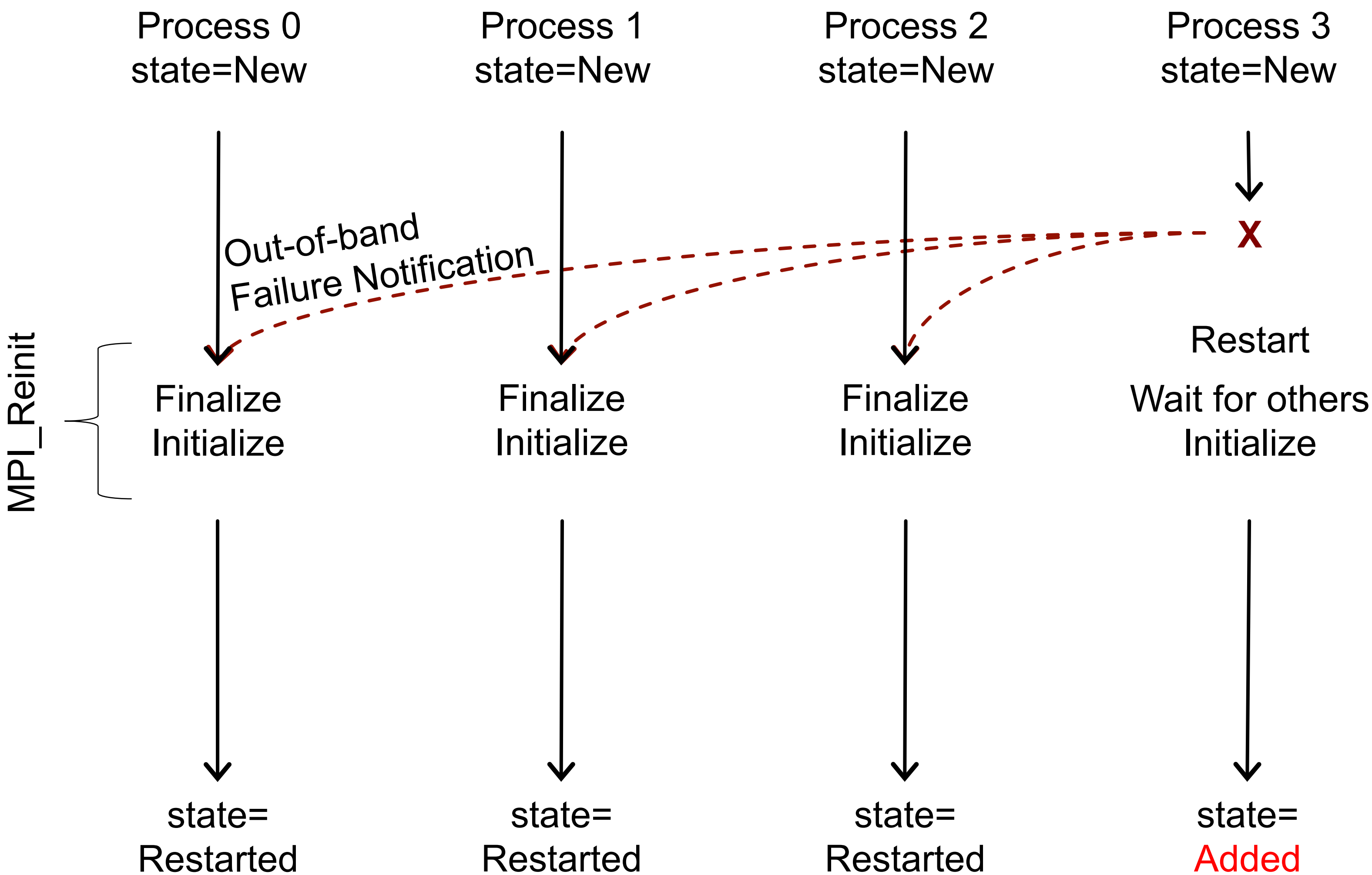
```
/* Cleanup routines */
typedef int (*MPI_Cleanup_handler) (
  MPI_Start_state start,
  void *state);

int MPI_Cleanup_handler_push (
  MPI_Cleanup_handler handler,
  void *state);

int MPI_Cleanup_handler_pop (
  MPI_Cleanup_handler *handler,
  void **state);
```



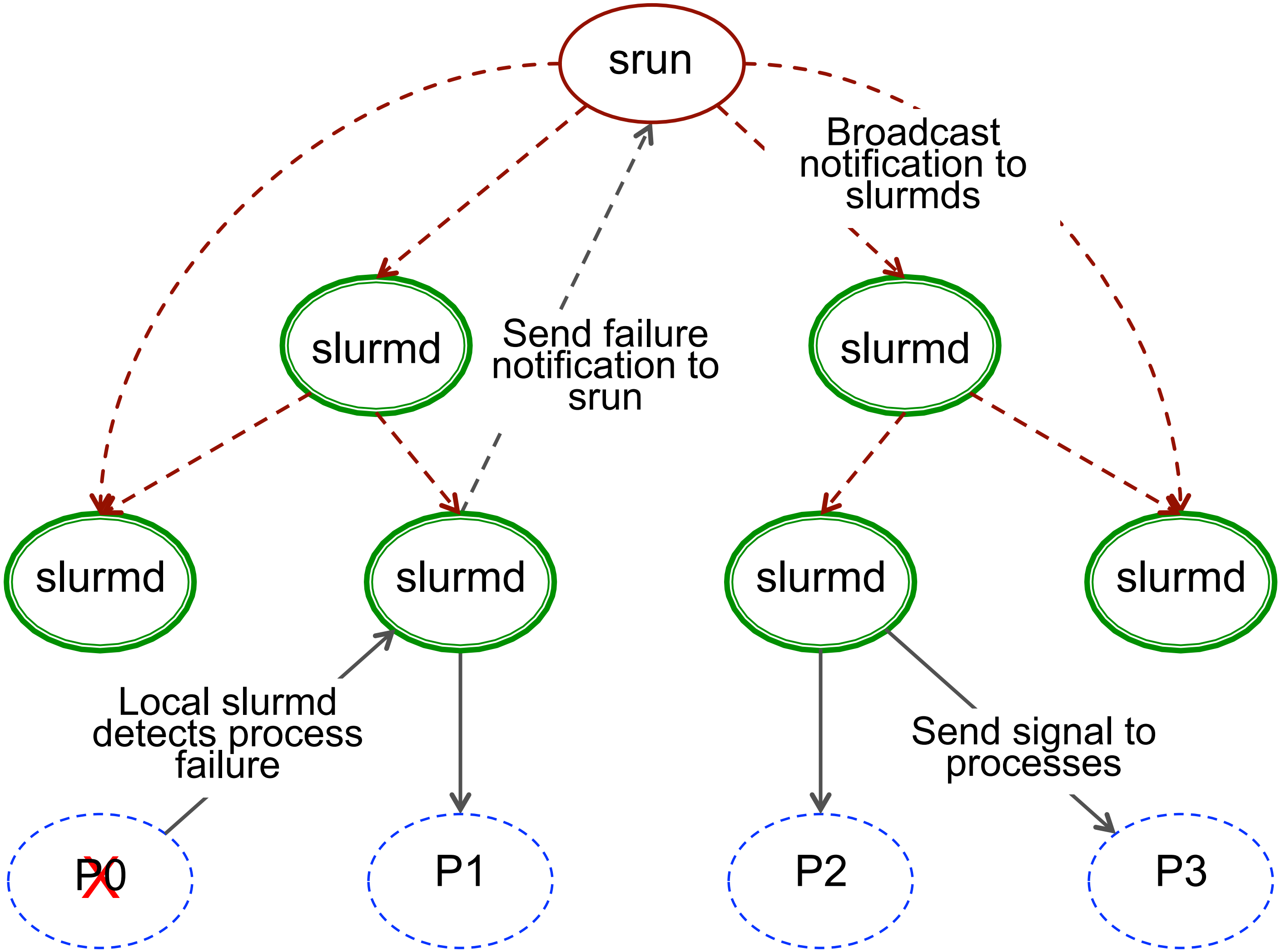
Example Program

```c
int cleanup_handler (MPI_Start_state, void *);

int resilient_main (int argc, char **argv,
     MPI_Start_state start_state)
{
  /* Recover using checkpoint */
  /* Do computation */
  /* Store checkpoint */
}

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);

  MPI_Cleanup_handler_push(cleanup_handler);  // Register application cleanup handler

  MPI_Reinit(&argc, &argv, resilient_main);   // Entry point for resilient MPI program

  MPI_Finalize();
}
```

Execution Flow of Reinit



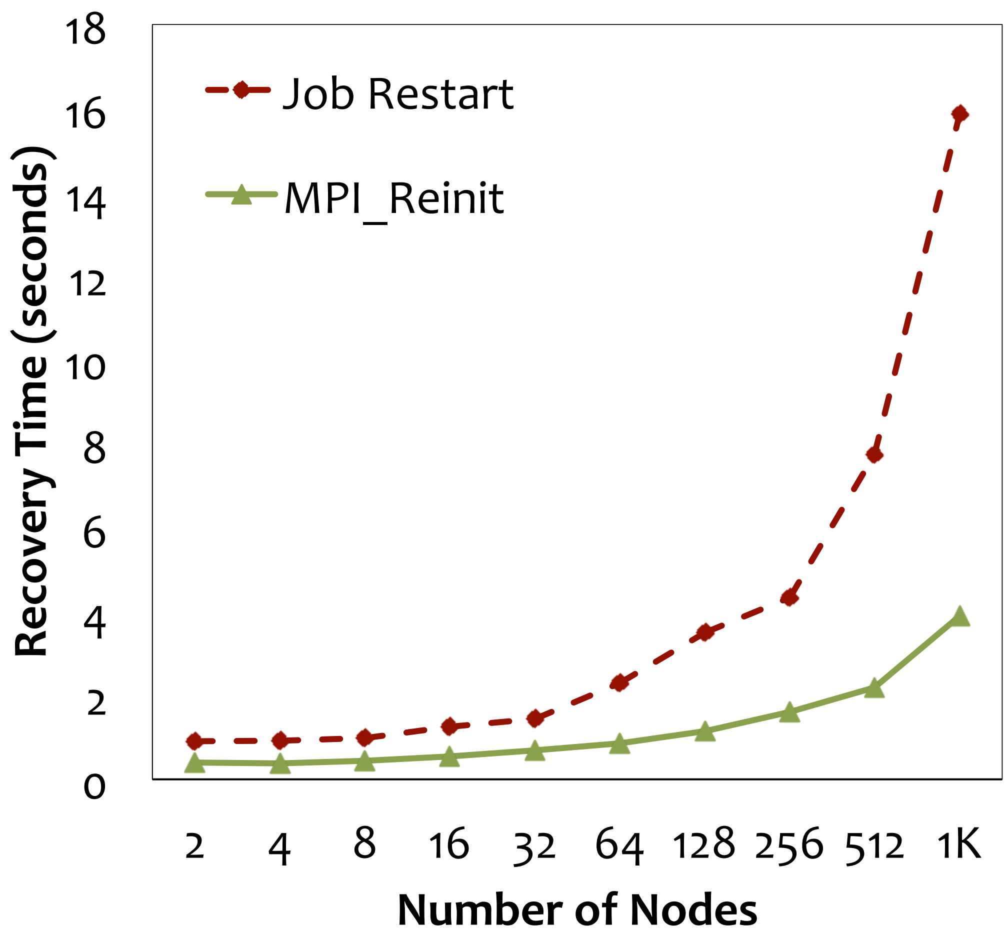
Failure Detection and Notification in SLURM



Experimental Evaluation

- Implementation of Reinit in SLURM-2.6.5 + MVAPICH2-2.1
- Experimental system
  - Sierra cluster @ LLNL
  - Intel Xeon 6-core EP X5660
  - 12 Cores per Node
- Single process failure scenario

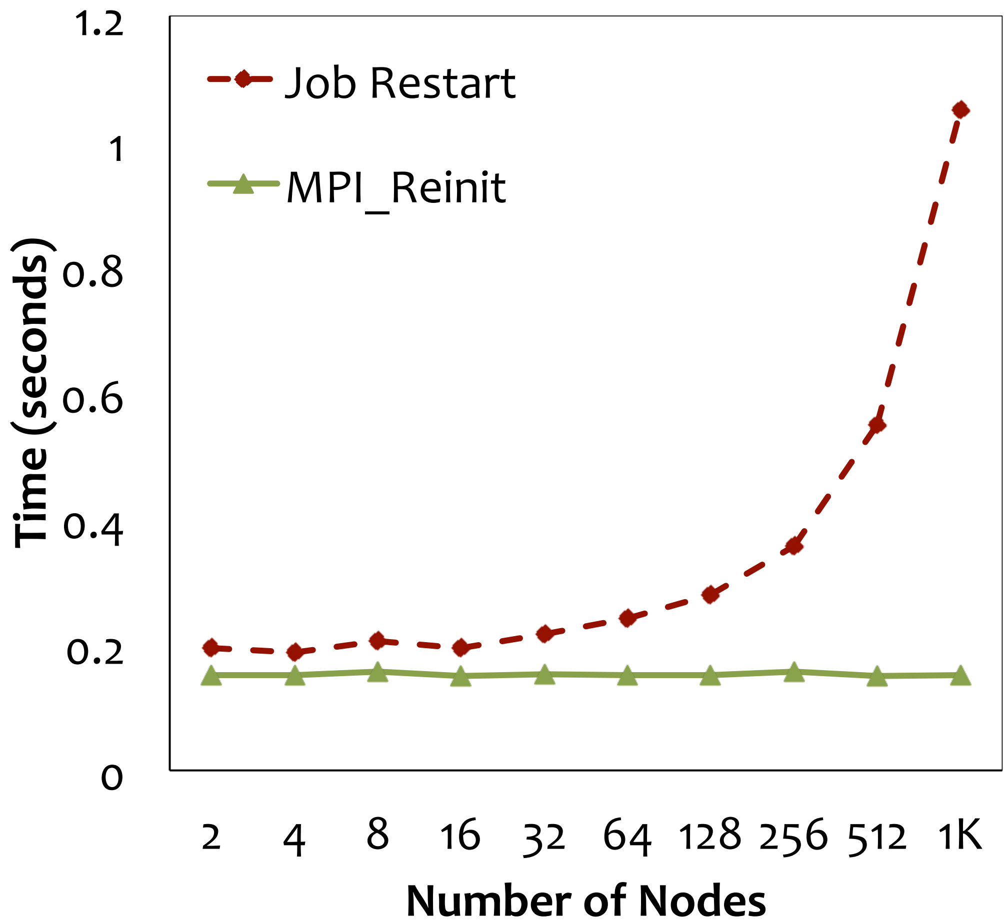
Recovery Time with MPI_Reinit Function



Less than 4 seconds to recover with 1K nodes, 12K processes

Recovery with REINIT is 4 times faster than Job restart

Time to Restore a 100 MB Checkpoint



Job restart forces each process to load checkpoints from persistent storage

Only the failed processes need to reload for REINIT

REINIT is 7 times faster than Job restart with 1K nodes, 12K processes

Summary

- Programming complexity can be a major impediment in adopting FT programing models for MPI applications
- We propose Reinit for low programing complexity and high scalability
  - Supports current FT programing practices (checkpoint/restart)
  - Close integration with resource manager (faster recovery)
  - Simple library and application cleanup
- Current implementation in MVAPICH + SLURM
- Future Work:
  - Support for node failures
  - Code release

Thanks to the Team Members!

Ohio State University (OSU)

 Sourav Chakraborty

 Khaled Hamidouche

 Hari Subramoni

 Dhabaleswar K. (DK) Panda

LLNL

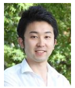
 Murali Emani

 Tanzima Islam

 Ignacio Laguna

 Kathryn Mohror

 Adam Moody

 Kento Sato

 Martin Schulz

Lawrence Livermore National Laboratory