**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
Fault-Tolerance Solutions for HPC Applications

Our focus

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

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

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
- Non-shrinking Recovery
- Local Recovery
- Global Recovery
- Forward Recovery
- Backward Recovery
- Support for libraries
- Use of PMPI interface

Starfish system [Agbaria and Friedman]
FT-MPI [Fagg and Dongarra]
MPICH-V [Bosilca et al.]
Run-through Stabilization [Hursey et al.]
Fenix [Gamell et al.]
FA-MPI [Hassani et al.]


How do we group or classify them all?
Classification of Existing FT Programming Models

Granularity of control / detection

Fine

Medium

Coarse

Full Restart

A

Error Code Checking

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

B

Try Blocks

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

C

main () {
    restart_from_here();
    MPI_Operation1();
    MPI_Operation2();
    MPI_Operation3();
    MPI_Operation4();
    return 0;
}
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

---

Lawrence Livermore National Laboratory
LLNL-PRES-688119
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)
- CC is adopted by the NIST Structured Testing Methodology
Cyclomatic Complexity Metric

Which code is easier to understand and easier to test?

**Code with 10 assignments**

```
1   a = ...
2   b = ...
3   c = ...
4   ...
5   ...
6   ...
7   ...
8   ...
9   ...
10  j = ...
```

One execution path

\[ CC = 0 + 1 = 1 \]

**Code with 10 if conditions**

```
1   if (cond1) {
2     if (cond2) {
3       if (cond3) {
4         ...
5         ...
6         ...
7         ...
8         ...
9         ...
10        if (cond10) {
```

More than 1,000 execution paths!

\[ CC = 10 + 1 = 11 \]

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

\[ CC = \text{decisions} + 1 \]

The recommended value for CC in software engineering and industry is 10
Example: Cyclomatic Complexity with the Error Code Checking Model

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 */
        }
    }
}

*Cyclomatic Complexity = 2

Refinement original loop

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

*Cyclomatic Complexity = 8

6 additional conditions

*ULFM example (taken from ULFM documentation)
What is the Complexity of MPI Applications?

Cyclomatic Complexity

10

5

FT code CC

Applications CC

Ideal Case
- Low complexity in applications
- Overall CC below suggested limit

Acceptable
- Moderate complexity in applications
- Overall CC on the suggested limit

Bad Case
- High complexity in applications
- CC exceeds suggested limit

Horrible

?
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

Most (77%) applications have already a high degree of complexity
Our Solution Space: **Low Programming Complexity**

**Fine**

- Error Code Checking
  - An error occurs during `MPI_Operation1()`, and `recovery()` is called.

**Medium**

- Try Blocks
  - `TRY {
    MPI_Operation1();
    MPI_Operation2();
  }
  TRY {
    MPI_Operation3();
    MPI_Operation4();
  }

**Coarse**

- Full Restart
  - `main() {
    restart_from_here();
    MPI_Operation1();
    MPI_Operation2();
    MPI_Operation3();
    MPI_Operation4();
    return 0;
  }

Granularity of control / detection:

- High programming complexity
- Medium programming complexity
- Low programming complexity

Our focus: Low programming complexity
Design Goals of the Reinit Interface

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

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

/* 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

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);

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

- Process 0
  - state=New
  - Initialize
  - state=Restarted

- Process 1
  - state=New
  - Initialize
  - state=Restarted

- Process 2
  - state=New
  - Initialize
  - state=Restarted

- Process 3
  - state=New
  - Initialize
  - state=Added
  - Restart
  - Wait for others

Out-of-band Failure Notification
Failure Detection and Notification in SLURM

- Local slurmd detects process failure
- Send signal to processes
- Broadcast notification to slurmds
- Send failure notification to slurmd

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

- Job Restart
- MPI_Reinit

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.
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