REINIT: Providing Backward Recovery for Bulk-Synchronous Codes by MPI Re-initialization

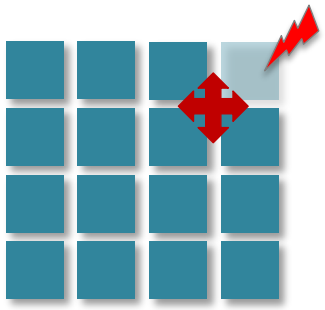
Ignacio Laguna, Giorgis Georgakoudis
LLNL

Presented at:
MPI Fault Tolerance Working Group

Lawrence Livermore National Laboratory

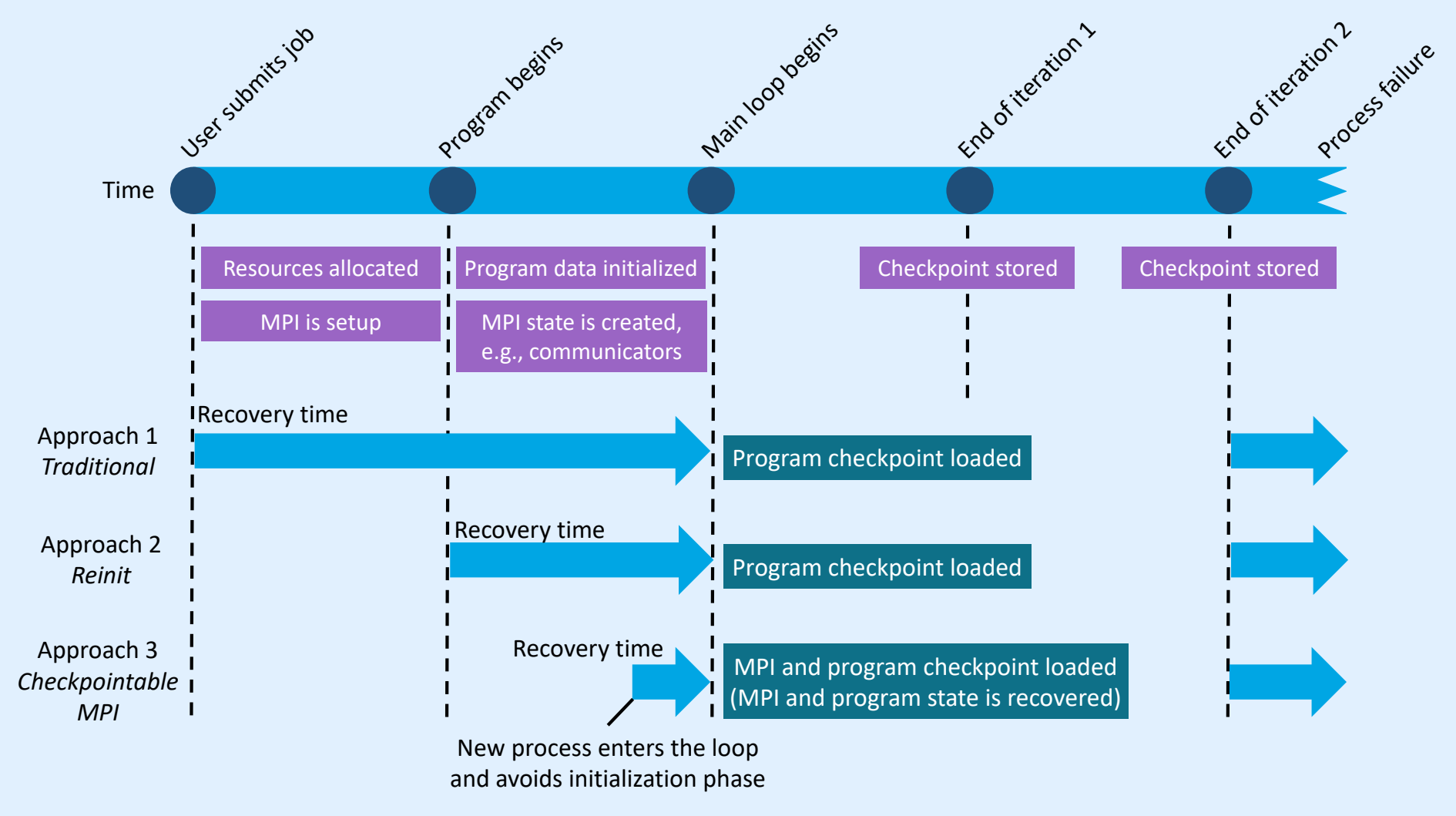
LLNL-PRES-LLNL-PRES-730179

Fault Tolerance Principles in Bulk Synchronous Applications



- A process failure typically propagates all other processes
  - On master/slave codes failures can be mitigated locally (usually)
- Most BSP codes use checkpoint/restart as a strategy to survive failures
- Question: **how can we make C/R for these codes faster?**

Approaches to Return MPI State Back to Previous State



Key Concepts of Reinit

- Automatically restart MPI after a failure
  - Programmers are not involved in failure detection and/or repairing the state of MPI
- The resulting state is the same as the state after MPI_Init returns
  - All communicators (except for MPI_COMM_WORLD) are deleted
  - All requests, and messages in queue are deleted
  - Connect/accept is usually not used in much in BSP codes
- Since the job is not killed, this allows optimizations to make failure recovery faster
  - Checkpoints can be loaded from memory
  - Connections of alive processes can be re-used
  - Eliminates job startup time (in some systems it could be high)
  - May eliminate data initialization

Avoiding Data Initialization for Alive Processes

```
int main()
{
   MPI_Init();

   /* Some time spent in
      data initialization (e.g., loading
      files) */

   while (error < threshold)
   {

      saveCheckpoint();
   }

   MPI_Finalize();
}
```

If MPI state (e.g., communicators) is not created in this phase, we want to restart from here

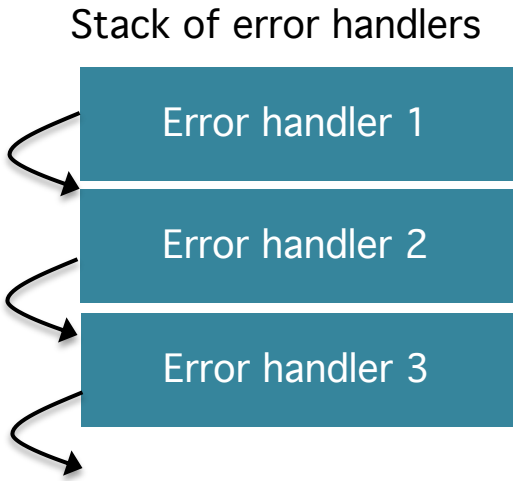
Original Interface

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

```
/* 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);
```



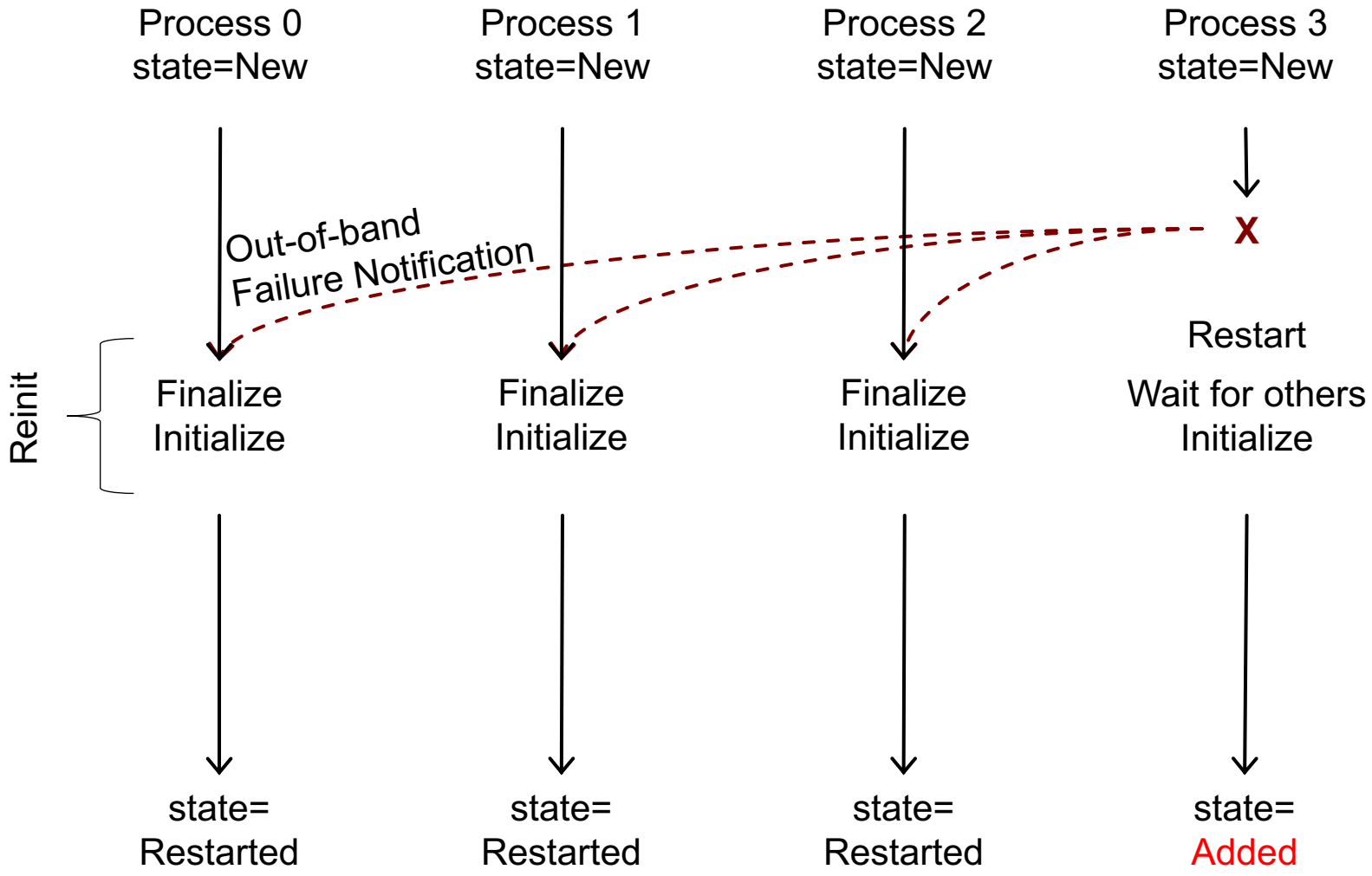
For more details:
https://github.com/tgamblin/mpi-resilience/

Simplified Example Program

```
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();
}
```

For more details:
https://github.com/tgamblin/mpi-resilience/

Execution Flow of Reinit



Possible New Interface

Description of the Alternative Reinit Interface

```
/* Initialization routines */
typedef enum {
    MPI_START_NEW,         // New process
    MPI_START_REINITED,    // Process is alive, re-inited after another process faulted
    MPI_START_RESTARTED    // Process has faulted and is restarted
} MPI_Reinit_state;

/* Set the process re-init point */
int MPI_Reinit();

/* Get the process re-init state  */
int MPI_Reinit_state
    (MPI_Reinit_state *state);
```

Side-by-side Examples of the Original and Alternative Interfaces

Original

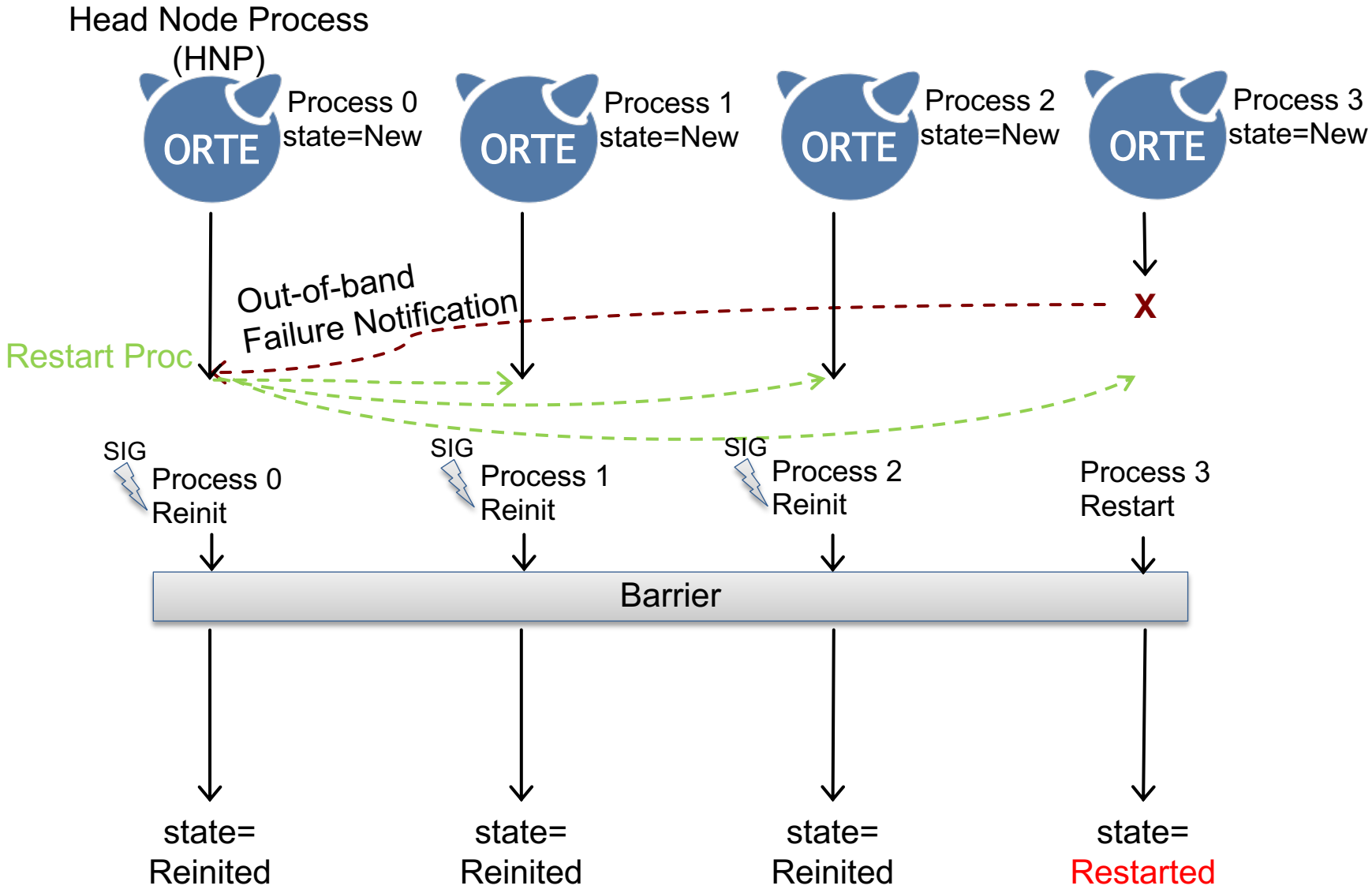
```
int resilient_main
(int argc, char **argv, MPI_Start_state start_state)
{
  /* Recover using checkpoint (in memory or file?) */
  /* Do computation */
  /* Store checkpoint */
}

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

  // Register application cleanup handler
  MPI_Cleanup_handler_push(cleanup_handler);
  // Entry point for resilient MPI program
  MPI_Reinit(&argc, &argv, resilient_main);

  MPI_Finalize();
}
```

Alternative

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

  // Register application cleanup handler
  MPI_Cleanup_handler_push(cleanup_handler);
  // Re-init point for resilient MPI program
  MPI_Reinit();

  MPI_Reinit_state state;
  // Get re-init state
  MPI_Reinit_state(&state);

  if( state == MPI_START_NEW )
    /* Do something if needed */
  else if( state == MPI_START_REINITED )
    /* Recover using checkpoint (in memory?) */
  else if( state == MPI_START_RESTARTED )
    /* Recover using checkpoint (from file?) */

  /* Do computation */
  /* Store checkpoint */

  MPI_Finalize();
}
```

Implementation

Runtime Flow of Reinit (based on OpenMPI)



MPI_Reinit Implementations

Old Interface: Function

```
void sighandler(int signo)
{
     siglongjmp(env, 1);
}

int MPI_Reinit
(int argc, char **argv, MPI_Restart_point point)
{
  sigsetjmp(env, 1);
  if( state == REINITED )
    <barrier>

  point(argc, argv, state);

  return MPI_SUCCESS;
}
```

Alternative Interface: Macro

```
void sighandler(int signo)
{
     siglongjmp(env, 1);
}

#define MPI_Reinit()     \
{                        \
  sigsetjmp(env, 1);     \
  MPI_Reinit_internal(); \
}

void MPI_Reinit_internal()
{
  state = MPI_Reinit_state()
  if( state == REINITED )
    barrier
}
```

Lawrence Livermore
National Laboratory