LLNL-TR-692704

LAWRENCE LIVERMORE NATIONAL LABORATORY

Fault Tolerance Assistant (FTA): An Exception Handling Programming Model for MPI Applications

A. Fang, I. Laguna, K. Sato, T. Islam, K. Mohror

May 23, 2016

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Fault Tolerance Assistant (FTA): An Exception Handling Programming Model for MPI Applications[*]

Aiman Fang
Department of Computer Science
University of Chicago
aimanf@cs.uchicago.edu

Ignacio Laguna, Kento Sato,
Tanzima Islam, Kathryn Mohror
Lawrence Livermore National Laboratory
{ilaguna, kento, tanzima, kathryn}@llnl.gov

ABSTRACT

Future high-performance computing systems may face frequent failures with their rapid increase in scale and complexity. Resilience to faults has become a major challenge for large-scale applications running on supercomputers, which demands fault tolerance support for prevalent MPI applications. Among failure scenarios, process failures are one of the most severe issues as they usually lead to termination of applications. However, the widely used MPI implementations do not provide mechanisms for fault tolerance. We propose FTA-MPI (Fault Tolerance Assistant MPI), a programming model that provides support for failure detection, failure notification and recovery. Specifically, FTA-MPI exploits a *try/catch* model that enables failure localization and transparent recovery of process failures in MPI applications. We demonstrate FTA-MPI with synthetic applications and a molecular dynamics code CoMD, and show that FTA-MPI provides high programmability for users and enables convenient and flexible recovery of process failures.

Keywords

Fault tolerance, MPI, exception handling, try/catch

1. INTRODUCTION

With the expected increase of mean time between failure (MTBF) in exascale systems [9, 5], along with more complex software and hardware stacks and runtimes, scientists require programming models that allow them to cope with failures efficiently and to increase productivity while developing large HPC applications. Although the MPI+X is expected to be widely used in exascale system—where X refers to node thread parallelism models, such as OpenMP—MPI does not provide mechanisms for fault tolerance: the standard specifies that if a failure occurs, the state of MPI is undefined, thus applications can do little more than abort. One of the grand challenges for exascale computing is therefore to provide practical fault-tolerance programming models for MPI applications.

Previous work has proposed fault-tolerant MPI libraries and interfaces [3, 8, 13]. For instance, the ULFM interface is a proposal under consideration to incorporate fault tolerance in the MPI Standard and provides functionality to deal with process failures, to repair communicators, and to propagate failures. In our attempt to make use of the programming interface that these approaches propose in large HPC programs, we have found the following limitations [10]:

- A substantial amount of code changes are required to use these interfaces. Code changes involve failure detection and fixing broken communicators; most applications do not have a central place where all communicators are created or repaired. Therefore programmers need to repeat many steps to enable resilience.
- *Failure localization* lacks sufficient support. Most models require checking returned error codes of MPI operations which can be cumbersome and can increase programming complexity, or provide error handlers. In the latter case, when a failure is detected (which can occur in an arbitrary MPI function call) an error handler is called. However, programmers do not know where in the code the failure originated or was detected from, hence limits flexible recovery.

To address these limitations, we propose FTA[1], a *try/-catch* programming model (see Figure 1) that allows transparent recovery of MPI communicators while providing failure localization guarantees—failures are detected and fixed within a user-defined code block. With FTA, programmers declare a *conversation* [4] (i.e., a set of MPI ranks that participate in executing a set of MPI calls) in a *try* code block. At the end of the conversation, all participating processes agree or disagree on a failure. If a failure is detected, FTA automatically executes recovery code, which involves repairing application-level state (e.g., by reading a checkpoint) and MPI-level state (e.g., repairing communicators).

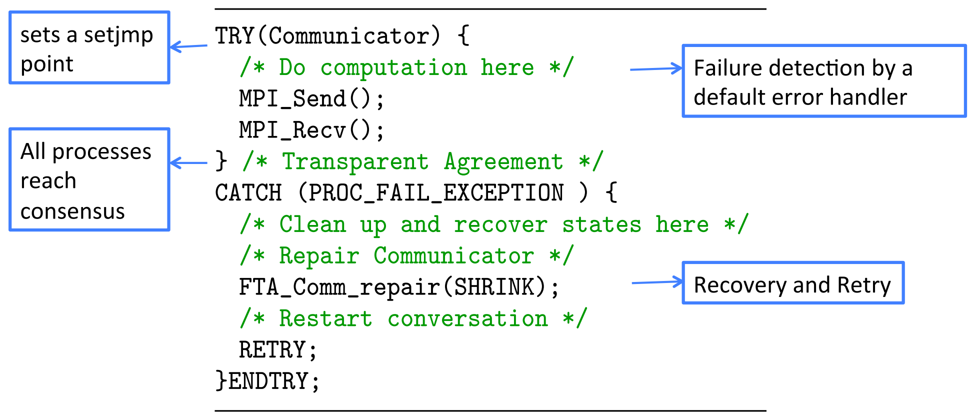
Hassani et al. proposed FA-MPI [7], a transactional resilience scheme for MPI. This work shares some of the ideas of FTA, for example, it allows *transactions*, which are similar in nature to FTA conversations; however, it works only for non-blocking MPI operations. In addition, FA-MPI requires users to write code to recover MPI state, such as

```
TRY(Communicator) { // MPI processes enter the try block
  // Execute MPI calls
} // Agree on whether or not a failure occured
CATCH (PROC_FAIL_EXCEPTION ) {
  // Receover MPI and application state
}ENDTRY;
```

Figure 1: Try/catch programing model that FTA proposes

[*]This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-TR-692704).

[1]Fault Tolerance Assistant


sets a setjmp point
Failure detection by a default error handler
All processes reach consensus
Recovery and Retry

```
TRY(Communicator) {
  /* Do computation here */
  MPI_Send();
  MPI_Recv();
} /* Transparent Agreement */
CATCH (PROC_FAIL_EXCEPTION ) {
  /* Clean up and recover states here */
  /* Repair Communicator */
  FTA_Comm_repair(SHRINK);
  /* Restart conversation */
  RETRY;
}ENDTRY;
```

Figure 2: FTA exception handling (try/catch) model and primitives

communicators. The goal of FTA is to perform this recovery transparently.

Two failure recovery models can be used in MPI: *shrinking recovery*, in which the number of resources (i.e., MPI processes and nodes) are reduced after a failure, or *non-shrinking recovery*, in which failed processes and nodes are replaced so that applications can continue with the original number of resources [10]. FTA supports both shrinking and non-shrinking recovery models.

We have implemented a prototype of FTA in Open MPI and have tested it in a synthetic and in an mini HPC application, CoMD [1]. In the rest of the paper we describe the design challenges and implementation details of FTA.

2. FTA DESIGN

The objective of FTA is to isolate the scope of failures and enable flexible shrinking/non-shrinking recovery with minor changes to applications. We deploy the *try/catch* mechanism to locate where failures occur. In addition, we design communicator management, which intelligently repairs all broken communicators upon failures.

2.1 Exception Handling (Try/Catch) Model

Exception handling is a classical mechanism of programming languages to build fault-tolerant programs [11]. It allows programmers to specify the recovery procedure when an exception is captured. The overview of FTA model and primitives are illustrated in Figure 2.

A challenge of exception handling in distributed systems is the complexity of asynchronous interacting activities. The idea of *Coordinated Atomic Actions* (CA actions) or *conversations* [4] has been proposed to control such complexity. In FTA, a **conversation** starts with a `TRY` statement, which sets a `setjmp` point for retry. The conversation encloses the interactions of a group of processes, that is, activities within one communicator and its subsets. Therefore, the granularity of failure detection is a conversation. Process failures are recognized as exceptions and raised to all members of the associated communicator.

To detect process failures within a conversation, FTA sets the error handler of designated communicator for the conversation. When MPI operations fail due to process failures, a `PROC_FAIL_EXCEPTION` is raised by the error handler. Processes that detect the failure will skip the rest of work in the conversation and jump to `CATCH` block, while other processes continue execution. At the end of a conversation, all processes within the conversation reach consensus on whether or not a failure occurred. Therefore, failures are guaranteed to be acknowledged by all the non-failed processes of the communicator at the end of a conversation. In our prototype implementation, we use the `MPI_Comm_agree` function provided by ULFM to perform the consensus. We expect that if the FTA programming model is incorporated in the MPI standard, other consensus protocol implementations can be used in the MPI library.

The recovery procedure is located in a `CATCH` block which usually includes code for cleaning up states, repairing communicators (transparently facilitated by FTA), and data redistribution (if needed). A `RETRY` statement—essentially a `longjmp` function—can be used to restart the `TRY` block. Finally, the `TRY/CATCH` pair is completed with the statement `ENDTRY`. Our initial prototype uses macros to implement language statements, such as `TRY` and `CATCH`.

2.2 Communicator Management

A challenge in try/catch MPI programming models is repairing communicators. Applications may have a set of communicators with various relationships, such as parent/child, overlapping (i.e., they have mutual MPI ranks), and isolated, depending on how communicators are created. In large and complex MPI applications, identifying and repairing broken communicators is a demanding task since applications do not usually have a central place where all communicators are created. Moreover, using the repaired communicator (which usually comes with a different handler) in the right code location is difficult. For instance, a shrinking operation on the broken communicator `COMM_1` will return `COMM_2`, which should be semantically used as `COMM_1` by applications.

FTA provides the ability to automatically repair all communicators through communicator management, requiring no effort from users. Programmers only need to specify the recovery mode of communicators—either shrinking or non-shrinking. In shrinking mode, all communicators exclude the failed processes, while in non-shrinking mode, pre-allocated spare processes substitute failed ones.

The design of FTA communicator management and automatic communicator repair is illustrated in Figure 3. FTA

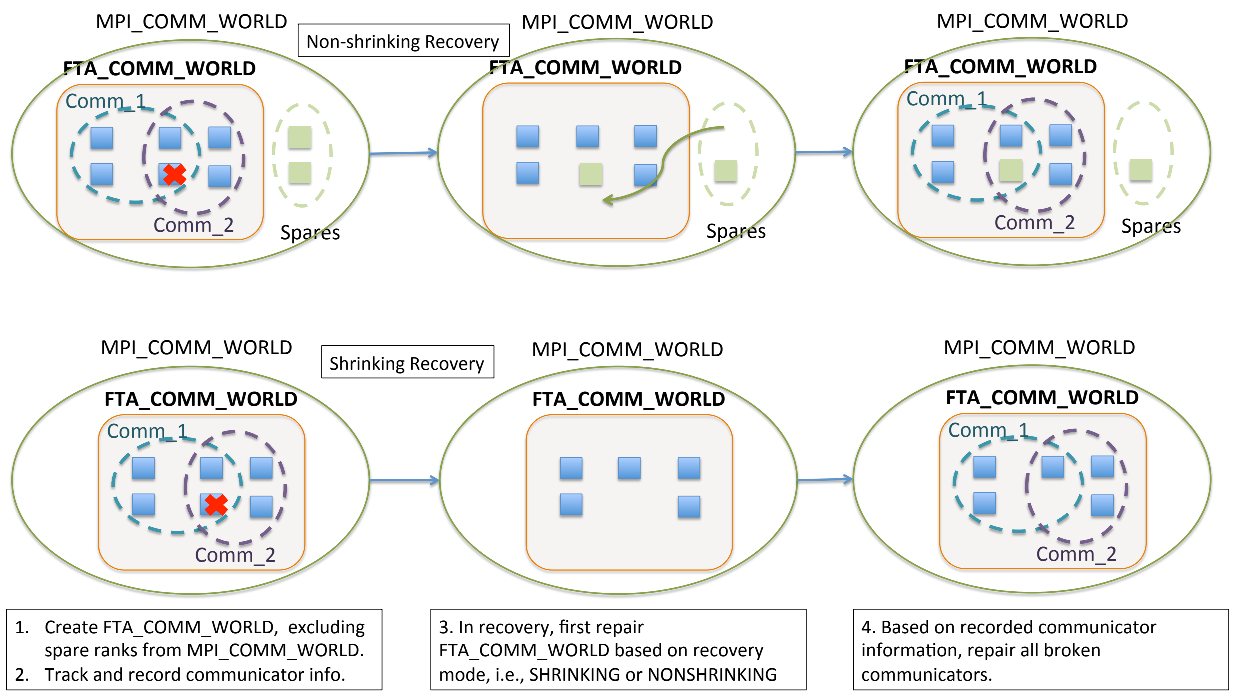

MPI_COMM_WORLD
Non-shrinking Recovery
FTA_COMM_WORLD
Comm_1
Comm_2
Spares
Shrinking Recovery
1. Create FTA_COMM_WORLD, excluding spare ranks from MPI_COMM_WORLD.
2. Track and record communicator info.
3. In recovery, first repair FTA_COMM_WORLD based on recovery mode, i.e., SHRINKING or NONSHRINKING
4. Based on recorded communicator information, repair all broken communicators.

Figure 3: Communicator management and automatic communicator repair

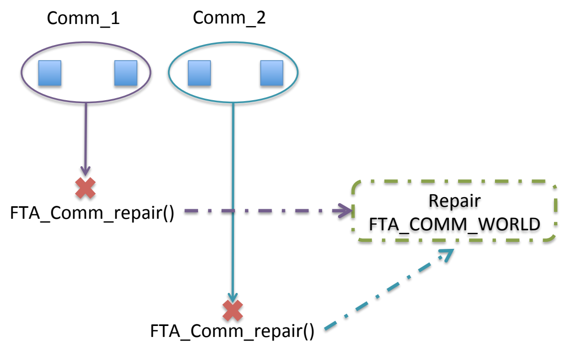


Figure 4: Asynchronous failure detection and recovery model

first creates the `FTA_COMM_WORLD`, which excludes the pre-allocated spare processes from `MPI_COMM_WORLD`. Without spare processes, `FTA_COMM_WORLD` is essentially a duplicate. Applications are required to use `FTA_COMM_WORLD` instead of `MPI_COMM_WORLD`. FTA exploits PMPI routines to replace MPI routines that are related to communicators operations. Therefore FTA is able to track the creation of any communicator by trapping all communicator creation operations, and recording the corresponding information, which includes a pointer to the communicator handler and associated ranks (see Figure 5). This information incurs negligible memory overhead. FTA maintains the mapping and lineage of communicators.

During recovery, FTA first recovers `FTA_COMM_WORLD` through the interface `FTA_Comm_repair` that is provided to users and it takes recovery mode as the parameter. Given the stored communicator and failed rank information, FTA finds all broken communicators and revokes them. Second, FTA constructs new communicators with the same set of process ranks. By assigning the pointer of communicator handler to the new communicator, FTA allows applications to use the original communicator handler in the application. This feature of FTA allows applications to keep same semantics and alleviate the burden on programmers to adjust algorithms after recovery.

2.3 Recovery

In distributed parallel programs, processes assigned in different communicators perform work concurrently. Our model guarantees that processes in the same conversation will be notified of failures synchronously at the end of the `TRY` block. However, failure notification may occur asynchronously in different communicators, and processes in the communicator that encounter the failure first need to wait for other processes to perform global repair before continuing. For instance in Figure 4, there are two disjoint communicators. A failure occurs in `COMM_1`, which sees the failure first. `COMM_1` calls `FTA_Comm_repari()` and waits for a global repair of `FTA_COMM_WORLD`. However, without intervention `COMM_2` may still be able to continue work until the point that members in `COMM_2` interact with those in `COMM_1`. In the worst case, members in `COMM_2` will see the failure at the final termination phase of program. `COMM_1` waits for `COMM_2` to participate in the world repair procedure, and after its completion, `COMM_1` and `COMM_2` restart locally.

3. EXAMPLES

Figure 6 shows how a bulk synchronous parallel (BSP) program might use our model. The application writes a checkpoint at the beginning, which has all the state the ap-

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm
    *newcomm)
{
   int ret = PMPI_Comm_create(comm, group, newcomm);
   /* set error handler */
   MPI_Comm_set_errhandler(*newcomm, MPI_ERRORS_RETURN);
   /* record <comm, group, ranks> */
   ...
   /* assign pointer to the communicator handler */
   ...
}
```

Figure 5: FTA wrapper: example use of profiling interface (PMPI) in FTA to track communicator creation

```
main()
{
   /* Initilize FTA which creates FTA_COMM_WORLD */
   FTA_Init();
   /* Initial checkpoint */
   WriteCkpt();
   for (i = 0; i < loop; i++) {
      TRY (COMM) {
         MPI_Send();
         MPI_Recv();
         WriteCkpt();
      }
      CATCH (PROC_FAIL_EXCEPTION) {
         /* Application clean up codes here */
         FTA_Comm_repair(SHRINK);
         /* Load data if needed */
         ReadCkpt();
         RETRY;
      }ENDTRY;
   }
   FTA_Finalize();
}
```

Figure 6: Using FTA for shrinking recovery in a synthetic application

```
main()
{
   FTA_Init();
   /* Initial checkpoint */
   WriteCkpt();
   for (i = 0; i < loop; i++) {
      TRY (FTA_COMM_WORLD) {
         /* If Restart, read checkpoint */
         if (FTA_RETRY) ReadCkpt();
         /* Molecular dynamics simulation */
         Simulation();
         WriteCkpt();
      }
      CATCH (PROC_FAIL_EXCEPTION) {
         FTA_Comm_repair(NONSHRINK);
         RETRY;
      }ENDTRY;
   }
   FTA_Finalize();
}
```

Figure 7: Using FTA for non-shrinking recovery in CoMD

plications needs to recover. In the `for` loop, a `TRY` block encloses computation and communication for communicator `COMM`, followed by a light-weight incremental checkpoint. If a failure exception is raised, the program calls `FTA_Comm_repair` to shrink `COMM`. Data redistribution may be needed for the surviving processes. The program calls RETRY to restart the `TRY` block.

Figure 7 illustrates FTA non-shrinking recovery for a molecular dynamics proxy-application, CoMD. In CoMD, we enclose each time step of the simulation in conversation for the communicator `FTA_COMM_WORLD`. If process failures occur during simulation, the exception is raised and followed by a non-shrinking recovery of `FTA_COMM_WORLD`. Spare processes will join the simulation loop by reading the checkpoints to initialize states and keep consistent with surviving processes. Note that allowing spare processes to join the loop only works when *try/catch* is used in the main function. We discuss challenges in supporting non-shrinking recovery in Section 5.

4. RELATED WORK

ULFM (User-Level Failure Mitification) [2] proposes a minimum set of extensions to MPI so that applications are able to deal with process failures and further to repair states of MPI. This set of APIs covers three important issues—failure detection, notification, and communicator repair which can be used to design various fault tolerance schemes. However the complexity of handling process failures still remain for applications such as challenge of code changes and failure localization problems that we identified in Section 1. Our work incorporate fault tolerance with *try/catch* mechanism, which not only provides programmability but also persists the recovery flexibility.

FA-MPI (Fault-Aware MPI) [7] proposes a transactional approach to address failure detection, isolation, mitigation and recovery for MPI-based applications. It provides a set of extensions to the MPI standard that applications can adopt to implement resilient transactions. The basic idea is to divide the codes into blocks, denoted as *TryBlocks*. Each *TryBlock* is a transaction containing a series of non-blocking MPI operations that bind to a communicator (of which any communicators inside a *TryBlocks* should be a subset). Failures are detected inside the block and notified to applications after *TryBlock*'s finish call. Applications then decide how to recover the block and need to handle issues such as inconsistency, communicator repair, etc.

FA-MPI restricts MPI to non-blocking communication operations because FA-MPI desires to check both local and remote status of operations to gurantee failure detection and consistency. Non-blocking MPI operations provide a request handle for tracking the status while blocking MPI operations lack such capability. Our work FTA-MPI shares the similarity with FA-MPI on isolating the scope of failures by grouping a set of operations. However FTA-MPI does not impose restrictions on blocking operations because FTA-MPI allows asynchronous failure detection and eventual consistent global repair. In addition, FTA-MPI alleviates the recovery burden of applications by automatically repairing communicators.

NR-MPI (Non-stop Fault Resilient MPI) [12] implements the semantics of FT-MPI and supports online recovery of `MPI_COMM_WORLD`. NR-MPI exploits the RMS (Resource Management System, i.e. SLURM) for fault tolerance resource management, failure detection and notification. Failures are detected by FD (Fault Detector) and FA (Failure Arbiter) in RMS. NR-MPI maintains two universe communicators and creates the world communicator from one of them. In a failure, NR-MPI reconstructs `MPI_COMM_WORLD` from the universe communicator using spare processes. Compared to NR-MPI, FTA-MPI supports both shrinking and non-shrinking recovery and handles recovery of all derived communicators of `MPI_COMM_WORLD`.

5. LIMITATIONS AND FUTURE WORK

FTA is work in progress. The following are some of the limitations and challenges that we plan to address as future work.

Efficient global failure detection. The FTA model assumes that at the end of a conversation, all the participating processes must agree on whether or not a failure occurred. This *agreement* can be implemented using distributed consensus protocols at the cost of communication overhead. Our FTA prototype uses the ULFM agreement functions for failure detection, but it can use other methods. In the future, we plan to investigate alternative failure detection and agreement protocols to reduce the overhead that is involved in each *try/catch* block.

Arbitrary non-shrinking recovery. Currently, FTA allows non-shrinking recovery only when a *try/catch* block is placed in the main function. This is because when a failure occurs at an arbitrary location in the call stack, there is no system-independent way to replace the failed process with a spare one. Most solutions to create an identical process (with arbitrary call-path depth) require system-level checkpointing, which can be incurs significant performance overhead. We will investigate how to provide a system-independent solution for this problem.

Nested error handling. A *try/catch* model should support nested failure handling, just as traditional exception handling models do in object-oriented programming. We have not explored fully the semantics and applications of nested failure handling in FTA; however, when provided, it would allow users to handle a variety of failures, including silent data corruption detected in the application inside a code block. This would allow to apply concepts, such as containment domains [6], in MPI applications.

6. SUMMARY

In this work we describe our initial design and prototype of FTA, a try/catch programing model for MPI applications. The goal of FTA is to assist programmers with failure localization—failures are confined to try blocks—and with repairing MPI communicators upon failures, using a try/-catch model. Despite being work in progress, we have tested our current framework of FTA in a mini application and have presented in this paper the main building blocks that could make general and flexible try/catch programming models a viable resilience solution for MPI programs.

7. REFERENCES

[1] CoMD: A Classical Molecular Dynamics Mini-app. http://codesign.lanl.gov/projects/exmatex.

[2] W. Bland. User level failure mitigation in mpi. In *Euro-Par 2012: Parallel Processing Workshops*, pages 499–504. Springer, 2012.

[3] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. An evaluation of User-Level Failure Mitigation support in MPI. *Computing*, 95(12):1171–1184, 2013.

[4] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *Software Engineering, IEEE Transactions on*, 1986.

[5] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 2009.

[6] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *SC*, 2012.

[7] A. Hassani, A. Skjellum, and R. Brightwell. Design and evaluation of fa-mpi, a transactional resilience scheme for non-blocking mpi. In *DSN*, 2014.

[8] J. Hursey et al. Run-through stabilization: An MPI proposal for process fault tolerance. In *Recent Advances in the Message Passing Interface*. Springer, 2011.

[9] J. Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, Feb. 2011.

[10] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, and B. R. de Supinski. Evaluating User-Level Fault Tolerance for MPI Applications. In *EuroMPI/ASIA*, page 57, 2014.

[11] A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors. *Advances in Exception Handling Techniques.* Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[12] G. Suo, Y. Lu, X. Liao, M. Xie, and H. Cao. Nr-mpi: A non-stop and fault resilient mpi. In *ICPADS*, 2013.

[13] K. Teranishi and M. A. Heroux. Toward Local Failure Local Recovery Resilience Model using MPI-ULFM. In *EuroMPI/ASIA*, 2014.