2014 IEEE 28th International Parallel & Distributed Processing Symposium

FMI: Fault Tolerant Messaging Interface for Fast and Transparent Recovery

Kento Sato
Dep. of Mathematical and Computing Science
Tokyo Institute of Technology
2-12-1-W8-33, Ohokayama,
Meguro-ku, Tokyo 152-8552 Japan
Email: kent@matsulab.is.titech.ac.jp

Adam Moody, Kathryn Mohror,
Todd Gamblin and Bronis R. de Supinski
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551 USA
Email: {moody20, kathryn, tgamblin, bronis}@llnl.gov

Naoya Maruyama
Advanced Institute for Computational Science
RIKEN
7-1-26, Minatojima-minami-machi,
Chuo-ku, Kobe, Hyogo, 650-0047 Japan
Email: nmaruyama@riken.jp

Satoshi Matsuoka
Global Scientific Information and Computing Center
Tokyo Institute of Technology
2-12-1-W8-33, Ohokayama,
Meguro-ku, Tokyo 152-8552 Japan
Email: matsu@is.titech.ac.jp

***Abstract*—Future supercomputers built with more components will enable larger, higher-fidelity simulations, but at the cost of higher failure rates. Traditional approaches to mitigating failures, such as checkpoint/restart (C/R) to a parallel file system incur large overheads. On future, extreme-scale systems, it is unlikely that traditional C/R will recover a failed application before the next failure occurs. To address this problem, we present the Fault Tolerant Messaging Interface (FMI), which enables extremely low-latency recovery. FMI accomplishes this using a survivable communication runtime coupled with fast, in-memory C/R, and dynamic node allocation. FMI provides message-passing semantics similar to MPI, but applications written using FMI can run through failures. The FMI runtime software handles fault tolerance, including checkpointing application state, restarting failed processes, and allocating additional nodes when needed. Our tests show that FMI runs with similar failure-free performance as MPI, but FMI incurs only a 28% overhead with a very high mean time between failures of *1 minute*.**

***Keywords*-Fault tolerance; MPI; Checkpoint/Restart;**

I. Introduction

Advances in high performance computing (HPC) enable larger and higher-fidelity simulations, which are critical for scientific discovery. However, larger systems generally have larger numbers of processing elements and other components, which increase the overall system failure rate. If this trend continues, experts predict that mean time between failures (MTBF) for future systems may shrink to tens of minutes or hours [1]–[3]. Our prior analysis of failure rates on HPC systems showed that if we extrapolate the observed single-node failure rates to a system with 100,000 nodes, the estimated MTBF is 17 minutes [4]. Without changes in HPC trends or more reliable hardware, fault tolerant techniques will be critical for future, extreme-scale systems.

The Message Passing Interface (MPI) [5] is the de-facto HPC programming paradigm, but it employs a fail-stop model. On failure, all processes in the MPI job are terminated. MPI applications generally cope with failures using checkpoint/restart schemes (C/R). They periodically write their state to files on a reliable store, such as a parallel file system (PFS). When a failure occurs, the current job is terminated, and the application is relaunched as a new job that restarts from the last checkpoint. The approach is simple, but it incurs significant overheads due to high checkpoint and restart costs [6], [7].

In environments with high-frequency failures, it is critical that applications restart quickly, so that useful work can be done before the next failure occurs. For more efficient execution at extreme scale, there are four capabilities needed for resilience in such an environment: a messaging interface that can run through faults, fast in-memory or node-local checkpoint storage, fast failure detection, and a mechanism to dynamically allocate additional compute resources in the event of hardware failures, which terminates running processes.

Our approach to satisfy these requirements is the Fault Tolerant Messaging Interface (FMI), a survivable messaging interface that uses fast, transparent in-memory C/R and dynamic node allocation. With FMI, a developer writes an application using semantics similar to MPI. The FMI runtime ensures that the application runs through failures by handling the activities needed for fault tolerance. Our implementation of FMI has failure-free performance that is comparable with MPI. Experiments with a Poisson equation solver show that running with FMI incurs only a 28% overhead with a very high mean time between failures of *1 minute*.

Our key contributions are:

- a simplified programming model to enable fast, transparent C/R;
- implementation of a runtime that withstands process failures and allocates spare resources;
- a new overlay network structure called *log-ring* for scal-

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.126

able failure detection and notification;
- and demonstration of the fault tolerance and scalability of FMI even with a MTBF of 1 minute.

Our paper is organized as follows. We present characteristics of failures on large systems, and we identify critical resilience capabilities in Section II. In Section III, we introduce FMI, and we detail its implementation in Section IV. We describe our in-memory C/R strategy and the modeling in Section V. In Section VI, we present our experimental results. We detail related work in Section VII, and in Section VIII, we discuss the limitations of the current state of FMI and our future plans to mitigate them.

II. Background

Here, we first describe the types of failures observed on a number of large HPC systems. We then give background on the capabilities that are critical for fault tolerance on extreme-scale systems: a survivable messaging runtime, fast C/R, fast failure detection, and spare node allocation.

A. Characteristics of System Failures

We divide failures into two categories. A *recoverable* failure is one that can be remedied transparently by the hardware or operating system without terminating running processes. An *unrecoverable* failure causes the application to terminate. Examples of recoverable failures include single bit flips in DRAM and disk failures [8]. These failures occur frequently and are typically handled with hardware redundancy techniques such as ECC [9] or RAID [10]. Without hardware recovery techniques, these errors substantially degrade performance. Unrecoverable failures include CPU, motherboard, and power supply failures [8], [11]. These failures cause affected nodes to crash, terminating any running processes and losing the full contents of memory on the nodes.

In this work, we address unrecoverable failures, which we refer to as simply *failures*. Previous studies show that certain failures occur more often than others on large scale systems, and in particular, most failures affect a small portion of the system. For instance, 85% of job failures on Linux clusters at Lawrence Livermore National Laboratory (LLNL) affect at most one node [4]. Similarly, as reported in [11] and as shown in Table I, about 92% of failures affect a single node on TSUBAME2.0, and only about 5% of failures affect more than 4 nodes. For a more detailed breakdown of failure modes on TSUBAME2.0 from November 2010 to April 2012, see Figure 1 showing failure rates (the number of failures per second) of each component.

TABLE I: TSUBAME2.0 Failure Types

Failure type	Affected nodes	Failures per year	MTBF
PFS, Core switch	1408	5.61	65.10 days
Rack	32	4.20	86.90 days
Edge switch	16	21.02	17.37 days
PSU	4	12.61	28.94 days
Compute node	1	554.10	0.658 days

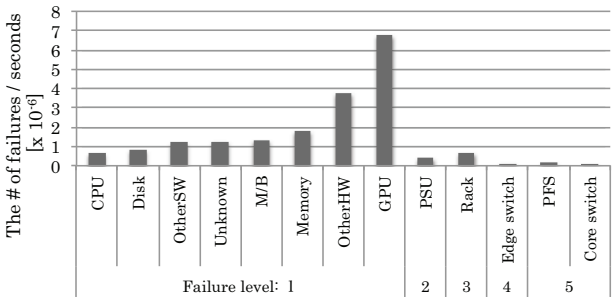


Fig. 1: TSUBAME2.0 Failure Breakdown

B. Critical Capabilities for Fault Tolerance

As described in the previous section, most failures only affect a small portion of the system, so the vast majority of processes and connections are still valid after a failure. It is inefficient for the runtime to tear all of this down only to immediately relaunch and reconnect it all. Launching large sets of processes, loading executables and libraries from shared file systems, and bootstrapping connections between those processes takes non-trivial amounts of time. All of this motivates the need for a survivable messaging runtime system. Such a system should be able to maintain processes and connections that are unaffected by the failure while starting and integrating replacement processes as needed.

However, a survivable messaging runtime itself is not sufficient for fault tolerant computing at extreme scales; a node failure destroys part of a parallel application's state. Today, applications checkpoint to a reliable PFS to mitigate node failures, and while this is sufficient for small systems, it incurs high overheads at extreme scale.

Multilevel C/R is a proven approach to lower these overheads [4], [12]. In this approach, checkpoints are placed in RAM or other node-local storage, and encoding techniques are used to protect data against common failures such as single-node failures. Only a select few checkpoints are copied to the PFS to guard against more catastrophic failures. To distinguish between these two types of checkpoints, the former are called *level-1 checkpoints* and the latter are called *level-2 checkpoints*. Since most failures only affect a small portion of the system, simple encoding schemes are often sufficient to recover lost data, and node-local storage provides fast, scalable performance. The net effect is that multilevel checkpointing gains an advantage by making the common case fast.

In a survivable model, processes that do not fail are not terminated; they simply keep running. The runtime system is responsible for starting new processes to replace those that failed and for providing a mechanism to incorporate the processes into the already running job, including acquisition of additional compute nodes if node failures occurred.

One solution is to request additional nodes in the allocation, e.g., request 70 compute nodes for a 64-node job, reserving 6 additional nodes as spares. A difficulty with this approach

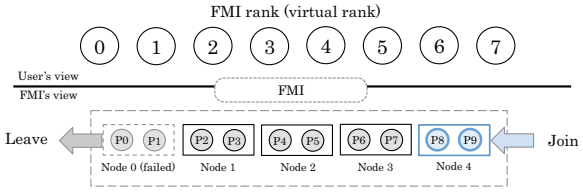


Fig. 2: Overview of FMI

```
1: int main(int *argc, char *argv[]) {
2:   FMI_Init(&argc, &argv);
3:   while ((n = FMI_Loop(...)) < numloop) {
4:     /*user program*/
5:   }
6:   FMI_Finalize();
7: }
```

Fig. 3: FMI example code

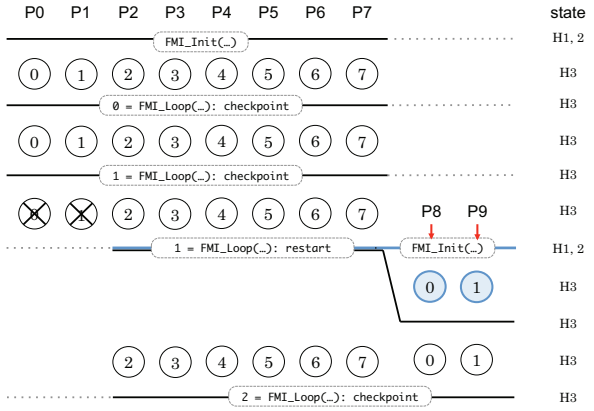


Fig. 4: Example: P0(rank=0) and P1(rank=1) fail after loop_id=1. P8 and P9 start from loop_id=1 as rank 0 and 1 each, and the other processes retry loop_id=1

is correctly estimating the number of spare nodes needed by a job, which requires knowledge of the failure characteristics of the machine and the behavior of the application. Another solution is to request compute nodes from the resource manager. This method may incur a high overhead if the job has to wait for spare nodes to become available. This overhead is reduced if the resource manager keeps a reserve of spare nodes specifically for fault tolerance. Either way, to support fast restart, it is critical to have access to spare resources.

Finally, one needs a fast, scalable mechanism to detect and react to failures. All of the above mechanisms matter little if the time to detect a failure overwhelms the cost to restart the job. Thus, we take these properties as required capabilities for our design of FMI: a survivable messaging runtime, fast C/R, scalable failure detection, notification, and spare node allocation.

III. FMI Programming Model

In this section, we describe the FMI programming model with an overview and then with an example.

A. Overview

With FMI, an application developer writes an application with MPI semantics, and FMI ensures that the application is agnostic to failure. The FMI runtime software handles the fault tolerance activities, including fast checkpointing of application state, restarting failed processes on failure, restoring application state, and allocating additional nodes when needed.

In Figure 2, we give an overview of FMI. Each process in an FMI application has an *FMI rank* as in MPI. But unlike MPI, an FMI rank is virtual and not bound to a particular process (Px) on a physical node. FMI may change the mapping of FMI ranks to processes to hide underlying hardware failures, transparently to the application. FMI also provides a capability for compute nodes to join or leave the job dynamically, primarily to replace failed nodes with spare nodes. Although our current prototype of FMI has limitations (See Section VIII), FMI transparently intercepts MPI calls, so that existing MPI applications can run on top of FMI with minimal code changes or without any code changes if users want to run their application with the fault tolerance capabilities disabled.

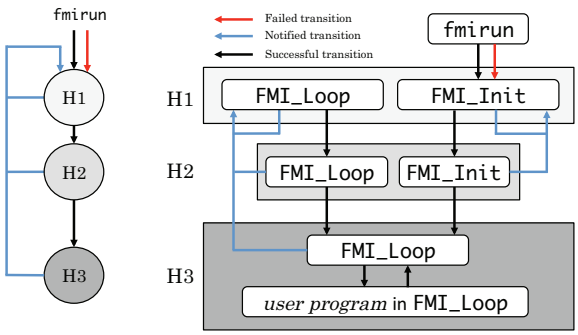
B. Writing an FMI Application

Writing a fault tolerant application is usually a complex ordeal, especially if there are a large number of dependencies across processes. With FMI, application developers simply write their code with MPI semantics and fault tolerance is provided by FMI. Figure 3 shows an example main loop for a code using FMI. The primary difference between the FMI and MPI programming models is that FMI provides the function `FMI_Loop` that synchronizes the application, writes checkpoints, or rolls back and restarts as needed. This single function call makes an application fault tolerant:

`int FMI_Loop(void** ckpts, size_t* sizes, int len).`

The parameter `ckpts` is an array of pointers to variables that contain data that needs to be checkpointed. If a failure occurred during the previous loop iteration, the last good values of the variables replace the values in `ckpts` to roll back to the last checkpoint. The parameter `sizes` is an array of sizes of each checkpointed variable, and `len` is the length of the arrays. `FMI_Loop` returns the loop iteration count (`loop_id`) incremented from 0 regardless of whether a checkpoint was written during this loop or not. However, if `FMI_Loop` rolls back and restores the last checkpoint, it returns the `loop_id` during which the last checkpoint was written.

When `FMI_Loop` is called the first time at the beginning of the execution, it writes checkpoints in memory using `memcpy` to minimize checkpoint time, and applies erasure encoding to the checkpoints using XOR encoding for level-1 checkpointing (See Section V). When completed, `FMI_Loop` guarantees that an application can continue to run even on a failure within the loop as long as any failures that occur are recoverable by the level-1 checkpoint. After the first call, `FMI_Loop` writes checkpoints at an interval specified by an *interval* environmental variable. Alternatively, if a user specifies an *MTBF*



(a) High level view (b) Low level view

Fig. 5: Process states for FMI

environmental variable, Currently FMI dynamically auto-tunes the checkpoint interval to maximize efficiency according to the MTBF based on Vaidya's model [13]. Future versions of FMI will support multilevel C/R, and optimize the intervals based on our multilevel C/R models [4], [11].

Figure 4 shows an example where `FMI_Loop` writes checkpoints every loop, i.e. $interval$=1. If a failure occurs (e.g., after `loop_id` = 1), all FMI ranks are notified of the failure by FMI (See Section IV-C), and all FMI communication calls return an error until recovery is performed in `FMI_Loop`. Then, the FMI process management program (`fmirun`) transparently allocates another node and spawns new processes (P8, P9 in the example) on the node to keep the number of FMI ranks constant. After all FMI ranks reach `FMI_Loop`, `FMI_Loop` restores the values of the checkpointed variables from `loop_id` = 1 and returns `loop_id` = 1. All recovery operations are transparent to the application, and all processes are simply rolled back to the last good state.

IV. FMI System Design

In this section, we detail our implementation of FMI. We describe our methods for keeping track of process states, managing dynamic node allocation, and joining new processes into the running application; our new overlay network design called *log-ring* for scalable failure detection and notification; and our method for transparently recovering communicators.

A. Process State Management

FMI manages the states of all processes, tracking whether or not processes are running successfully, and synchronizing for recovery when a failure occurs. Figure 5 shows a high level and low level view of transitions of process states. There are three process states in FMI: *Bootstrapping* (H1), *Connecting* (H2), and *Running* (H3).

The H1 and H2 states involve launching processes and establishing internal FMI communication networks, while the H3 state represents the running state of the application. In the H1 state, `fmirun` launches the FMI ranks (See Section IV-B), which then gather connection information (*endpoints*) to establish a dedicated low-latency communication network, similar to Open MPI's Matching Transfer Layer [14]. FMI

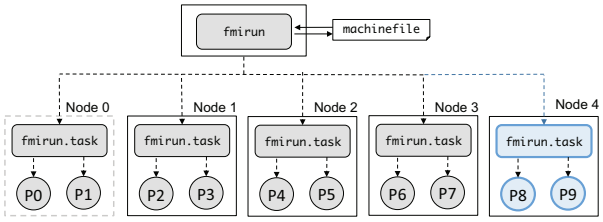


Fig. 6: FMI process management

uses *PMGR* [15], which provides a scalable communication interface for bootstrapping MPI jobs and exchanging messages via TCP/IP. On success, the processes transition to the H2 state, where the FMI ranks create a *log-ring* overlay network for scalable failure detection (See Section IV-C). If both H1 and H2 states succeed, the processes transition to H3. In the H3 state, the processes execute the application code, with the addition of `FMI_Loop`, that performs fault tolerance activities as described in Section III-B.

If any processes terminate because of a failure, `fmirun` launches new processes to replace them; they begin in the H1 state via the *Failed transition* path. The non-failed processes transition from their current state back to H1 on the *Notified transition* path. Thus, all processes transition to H1, then update endpoints to transparently recover communicators (See Section IV-D). On success, all processes transition to H2 and then H3 on *Successful transition* paths.

Figure 5(b) shows details of the states and transitions, and how failed ranks join the running non-failed processes. H1 and H2 are synchronizing states because they involve collective communications. Newly launched processes in H1 execute `FMI_Init`. Non-failed processes block in `FMI_Loop` until the new processes are bootstrapped and endpoints in the internal communication network are established. Then all processes transition to H2 to rebuild the *log-ring* network. If a process is notified of failure during `FMI_Loop`, non-failed processes abort all C/R operations, then internally transition to H1. Following this, the application computation begins from the previous iteration or the iteration with the last good checkpoint.

B. Hierarchical Process Management

Figure 6 shows an overview of the hierarchical structure of FMI process management. The master process `fmirun` is at the top level. `fmirun` has similar functionality to `mpirun` in MPI, but also manages processes during recovery in the event of failure. `fmirun` spawns `fmirun.task` processes on each node, which are at the second level of the hierarchy. Each `fmirun.task` calls `fork`/`exec` to launch a user program (`Px`) and manages the processes on its own local node.

If any `fmirun.task` receives an unsuccessful exit signal from a child process, `fmirun.task` kills any other running child processes, and exits with `EXIT_FAILURE`. When `fmirun` receives an exit signal from an `fmirun.task`, `fmirun` attempts to find spare nodes to replace those that failed in the `machinelist` file; if no spare nodes are found, or if there are

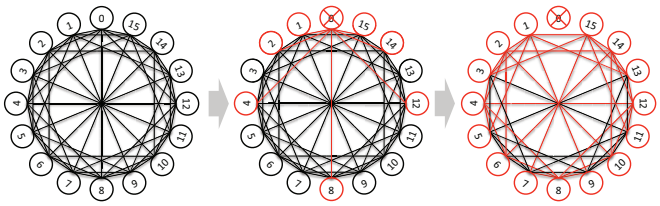


Fig. 7: Left: Structure of the *log-ring* overlay network ($n = 16$). Middle: If process 0 fails, processes 1, 2, 4, 8, 12, 14, and 15 are notified by ibverbs. Right: All processes are notified with 2 hops.

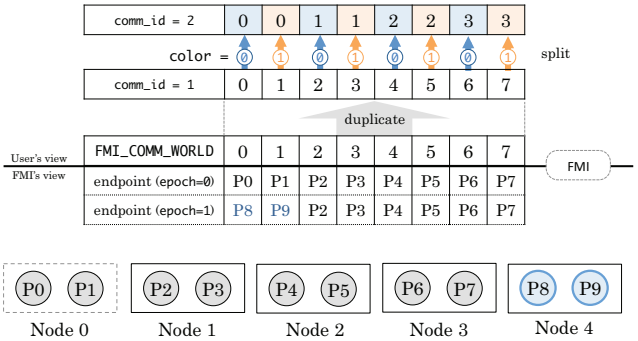


Fig. 8: Transparent communicator recovery

not enough to replace all that failed, $fmirun$ waits until new nodes are allocated from the resource manager. It then spawns any lost `fmirun.task` processes onto the spare nodes or the new nodes.

In our design, the master process (`fmirun`) becomes a single point of failure. However, because the MTBF of a single node in HPC systems is an order of years [4], [16], the failure rate for `fmirun` is negligibly small. That said, we plan to explore distributed management designs in future work.

C. Scalable Failure Detection

On failure, all surviving processes need to be notified so that the recovery process can begin, and restore consistent checkpoints across all processes. However, not all low-level communication libraries include a failure detection capability. For example, the Performance Scaled Messaging (PSM) library, a low-latency communication library for QLogic Infiniband, returns an error if there is a failure during connection establishment. However, once the connection is established, successive communication calls (e.g., sends or receives), do not return any errors even in the event of a peer failure.

One approach for detecting failures is that when `fmirun` receives a `EXIT_FAILURE` signal from an `fmirun.task`, `fmirun` could send notification signals to all other `fmirun.task` processes. However, the time complexity of this approach is $\mathcal{O}(N)$ in the number of compute nodes, which is not scalable.

Our approach to failure detection is a distributed method using a *log-ring* overlay network across the FMI ranks which can propagate failure notification in $\lceil \lfloor \log_2(n) \rfloor / 2 \rceil$ messages. We use the Infiniband Verbs API, *ibverbs*, a low-level communication library for Infiniband. The ibverbs library includes an event driven error notification capability, such that, if a process fails, all processes connected to the failed process can detect it by catching the error event.

The structure of the overlay network is critical for scalable failure detection. One option is a completely connected graph, where each process connects to all the other processes. In this option, notification of failure to all n processes occurs in $\mathcal{O}(1)$ steps. However, establishing the complete graph overlay network is $\mathcal{O}(n)$. In contrast, if we establish a ring overlay network, the connection cost is $\mathcal{O}(1)$, but propagation of failure notification to all processes is $\mathcal{O}(n)$.

To achieve a good balance between the overlay establishment cost and the global detection cost, we propose a *log-ring* overlay network. In a *log-ring* overlay network, each process makes $\log_2(n)$ connections with neighbors that are 2^k hops away in the FMI rank space ($2^k < n$). Figure 7 shows an example *log-ring* overlay network with $n = 16$ processes. In the example overlay, process 0 connects to 1, 2, 4, and 8 ($\log_2(16)$=4 connections), and receives connections from processes 8, 12, 14 and 15 (left in Figure 7). If process 0 fails, processes 1, 2, 4, 8, 12, 14 and 15 receive a disconnection event from ibverbs (middle in Figure 7). Once these processes receive the disconnection event, they explicitly close their other existing connections to propagate the failure notification. In this way, the failure notification reaches all processes with 2 hops (= $\lceil \lfloor \log_2(16) \rfloor / 2 \rceil$) in the example overlay network (right in Figure 7). In general, the *log-ring* overlay network can propagate failure notification across all processes with $\lceil \lfloor \log_2(n) \rfloor / 2 \rceil$ hops. Thus, the overlay network establishment and global failure notification are of order $\mathcal{O}(\log(n))$ and scalable.

The value of k in $\log_k(n)$ connections is a tunable parameter in FMI. Changing its value can change the overlay establishment and the global detection costs. As we show in Section VI-A, the establishment cost is negligible even with $k = 2$, which has maximal connections in the log-ring. Thus, we use $k = 2$ as the default value, but we leave the optimization of k for future work.

D. Transparent Communicator Recovery

One of obstacles of fault tolerant parallel computing is recovering communicators used in message passing. In FMI, communicators are a mapping between FMI ranks and physical processes, so recovering communicators is necessary for communication with the newly launched processes after recovery. Because FMI virtualizes the rank-to-process mapping, it can transparently recover communicators. Figure 8 shows an example where the application duplicates `FMI_COMM_WORLD`, and splits the communicator into two communicators. If a failure occurs on Node 0 and P0 and P1 terminate, FMI changes the process mapping by updating connection information (endpoint). In this example, FMI transparently updates the endpoints of FMI ranks 0 and 1 to P8, P9 respectively during the H1 bootstrapping state (See Figure 5(b)).

Another problem to address is that stale messages could be received if they are sent before a failure and not yet received

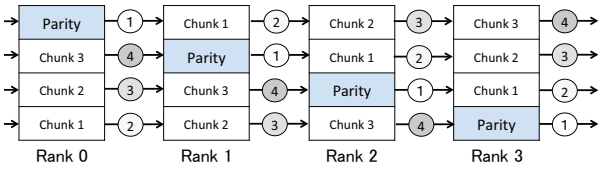


Fig. 9: XOR encoding algorithm: The circled numbers are the steps of sending/receiving parity

by the target rank. For example, if process A sends a message before a failure, but process B does not receive it before the failure, it may receive the stale message when it executes the receive operation after recovery. To address the problem, FMI increments an `epoch` variable after each recovery, and discards all messages that arrive with an older `epoch` value.

V. Fast and Scalable In-Memory C/R

With FMI, an application can continue to run even if a failure occurs. However, if a node fails, we may lose needed simulation data from processes on the failed node, so C/R is critical and must be scalable to be effective at large scales. Because the most common failures affect only a single or a few nodes [4], [11], multilevel C/R is effective and scalable.

A. Implementation

FMI employs the same level-1 checkpoint and encoding algorithm as the Scalable Checkpoint/Restart library (SCR) [4]. However, while SCR requires a file system interface for storing checkpoints, FMI writes checkpoints directly to memory without involving a file system for faster C/R throughput. Unlike with MPI, FMI does not terminate non-failed processes on a failure, and in-memory checkpoint data is not flushed.

At initialization, FMI splits ranks into XOR encoding groups (XOR group) with ranks in each group distributed across nodes. Because the common failure affects a single node, FMI ensures that each rank in the same node belongs to a different XOR group. For example, when processes are launched as in Figure 6, FMI splits the ranks as shown in Figure 8. Figure 9 shows the encoding algorithm for one XOR group. First, for an XOR group size of n, FMI divides a checkpoint into $n-1$ chunks, and allocates an additional parity chunk initialized with zeros. Each rank sends the parity chunk to its "right-hand" neighbor, and receives from its "left-hand" neighbor, and calculates "`parity` $\wedge=$ `chunk 1`". In general, each rank sends and receives a parity chunk, and computes "`parity` $\wedge=$ `chunk k`" at step k (k is circled number in Figure 9). Thus, each rank receives back the encoded parity chunk after n steps. When FMI restores a checkpoint, FMI decodes it with the same algorithm as the encoding, and then a newly launched rank collects the decoded checkpoint chunks from the other ranks.

B. Performance Model

When FMI writes s bytes of checkpoint data and encodes it, s bytes are copied by `memcpy`, $s + \frac{s}{n-1}$ bytes are transferred, and s bytes are encoded in total. Therefore, the time for C/R can be modeled as:

$$\frac{s}{mem\_bw} + \frac{s + s/(n-1)}{net\_bw} + \frac{s}{mem\_bw}$$

where mem_bw is memory bandwidth, and net_bw is network bandwidth. Because the XOR operation is memory-bound, the time becomes $\frac{s}{mem\_bw}$. When restoring a checkpoint, a newly launched rank collects the decoded checkpoint chunks from the other ranks at the end (Gather in Figure 11), so $\frac{s}{net\_bw}$ is added for restart.

The model tells us that the C/R time is constant regardless of the total number of processes. Thus, the our in-memory XOR C/R is scalable as well as fast.

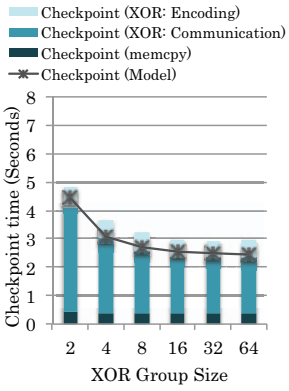


Fig. 10: XOR checkpoint time

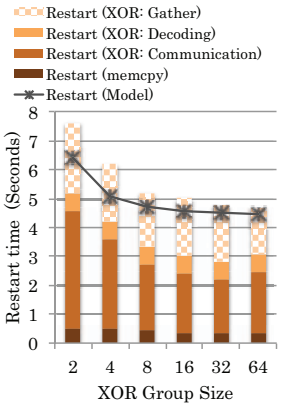


Fig. 11: XOR restart time

C. XOR Group Size Tuning

FMI directly uses memory to store checkpoints. Thus, reducing memory consumption while maintaining resiliency is important. If an XOR group size is small, memory consumption and C/R time become large. For large XOR group sizes, resiliency decreases because the XOR C/R encoding is tolerant to only a single rank failure in a XOR group. We performed experiments to evaluate the trade-offs of C/R time and XOR group size.

Figures 10 and 11 show the checkpoint and restart times where the checkpoint size is 6GB per node. For the memory and network bandwidths in the model, we use the peak bandwidth of the Sierra cluster at LLNL in Table II. We find that the C/R time starts to saturate at an XOR group size of 16 nodes. For this XOR group size, the parity chunk size is only 6.6 % of the full checkpoint size. Thus, we use 16 nodes for the XOR group size in the rest of our experiments.

VI. Experimental Results

To evaluate the performance and resiliency of FMI, we measured several benchmarks with FMI, and predict the performance of an FMI application run at extreme scale. We ran our experiments on the Sierra cluster at LLNL. The details of Sierra are in Table II. Because FMI follows the messaging semantics of MPI, we want to compare the performance of

TABLE II: Sierra Cluster Specification

Nodes	1,856 compute nodes (1,944 nodes in total)
CPU	2.8 GHz Intel Xeon EP X5660 × 2 (12 cores in total)
Memory	24 GB (Peak CPU memory bandwidth: 32 GB/s)
Interconnect	QLogic InfiniBand QDR

FMI with an MPI implementation. For those experiments, we used MVAPICH2 version 1.2 running on top of SLURM [17].

A. FMI Performance

TABLE III: Ping-Poing Performance of MPI and FMI

	1-byte Latency	Bandwidth (8MB)
MPI	3.555 usec	3.227 GB/s
FMI	3.573 usec	3.211 GB/s

We measured the point-to-point communication performance on Sierra, and compare FMI to MVAPICH2. Table III shows the ping-pong communication latency for 1-byte messages, and bandwidth for a message size of 8 MB. Because FMI can intercept MPI calls, we compiled the same ping-pong source for both MPI and FMI. The results show that FMI has very similar performance compared to MPI for both the latency and the bandwidth. The overhead for providing fault tolerance in FMI is negligibly small for messaging.

Because failure rates are expected to increase at extreme scale, C/R for failure recovery must be fast and scalable. To evaluate the scalability of C/R in FMI, we ran a benchmark which writes checkpoints (6 GB/node), and then recovers using the checkpoints. Figure 12 shows the C/R throughput including XOR encoding and decoding. The checkpoint time of FMI is fairly scalable because the checkpointing and encoding times are constant regardless of the total number of nodes. Also, because FMI writes and reads checkpoints to and from memory, the throughputs are high. FMI achieves 2.4 GB/sec checkpointing throughput per node, and 1.3 GB/sec restart throughput per node. On a restart, newly launched processes gather the restored checkpoint chunks from the other processes in the XOR group after the decoding as in Figure 11, so the restart throughput is lower than that of checkpointing.

Fast and scalable failure detection time and reinitialization time (H1 and H2 states) are critical in environments with high failure rates. Figure 13 shows the time for all processes to be notified of failure with the *log-ring* overlay. In this experiment, we inject a failure by sending a signal to kill a process in between two checkpoints to measure averaged performance. For example, if we write checkpoints after 10, 20, 30 seconds, we inject failures after 15, 25, 35 seconds. later. As shown, the global detection is scalable because the log-ring propagates the notification in logarithmic time. When a process terminates, ibverbs waits approximately 0.2 seconds before closing the connection to the terminated process. Therefore there is a constant overhead of 0.2 seconds before the notification starts to propagate in the *log-ring*.

FMI establishes the log-ring overlay network (H2 states) on the recovery. The initialization must be fast and scalable for fast recovery. In Figure 14, we show the initialization time for MVAPICH2 and FMI. For FMI, this is time spent in the H1 and H2 states. We compare the time in `FMI_Init` with that in MVAPICH2's `MPI_Init`. We see that the time to build the *log-ring* (H2 state) is small and scalable, because each process only connects to $\log_2 n$ other processes. The FMI bootsrapping time (H1 state) is about two times faster than that of MVAPICH2. The current prototype of FMI has limited capabilities compared to MPI. A smaller number of messages are exchanged in FMI initialization than in MVAPICH2, which results in faster bootstrapping. However, we expect that if FMI evolves to support more capabilities, it will also exchange more messages and its initialization time will approach that of MVAPICH2.

B. Application Performance with FMI

To investigate the impact of FMI on the performance of an actual application run, we used a Poisson equation solver, the Himeno benchmark [18]. Himeno is a stencil application in which each grid point is iteratively updated using only neighbor points. The computational pattern frequently appears in numerical simulation codes for solving partial differential equations. Himeno uses point-to-point communications and one Allreduce at the end of each iteration.

Figure 15 shows the performance of Himeno compared with MPI using SCR [4]. The FLOPS metric is computed based on time spent in application code making useful progress. For example, if an application fails at time t_1, and rolls back to time t_0, the FLOPS metric does not include the lost computation done to restore the application back to the state at t_1. We configured SCR to write checkpoints to `tmpfs` and optimize the checkpoint interval of both SCR and FMI with Vaidya's model [13] based on configured MTBF of 1 minute, and measured checkpointing time.

Because the point-to-point communication performance of FMI and MVAPICH2 are nearly the same (Table III), the performance of Himeno is nearly the same for FMI and MPI if we do not write any checkpoints during the execution (FMI & MPI in Figure 15). For checkpointing, SCR writes to memory via a file system (MPI + C), while FMI writes checkpoints directly to memory using `memcpy` (FMI + C). Thus, FMI exhibits higher performance by 10.3 % with the same memory consumption as MPI when checkpointing is enabled. We also injected failures into Himeno to see the impact of killing a process with a MTBF of 1 minute during the execution. Even with the very high failure rate, we found that Himeno incurred only a 28% overhead with FMI. Because the FMI C/R time is constant regardless of the total number of nodes according to performance model in Section V-B, we expect FMI to scale to a much larger number of nodes.

C. Resiliency with FMI

FMI applications can continue to run as long as all failures are recoverable. To investigate how long an application can

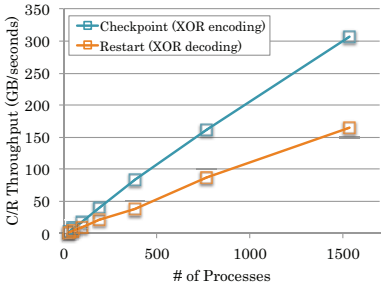


Fig. 12: Checkpoint/Restart scalability with 6 GB/node checkpoints, 12 processes/node

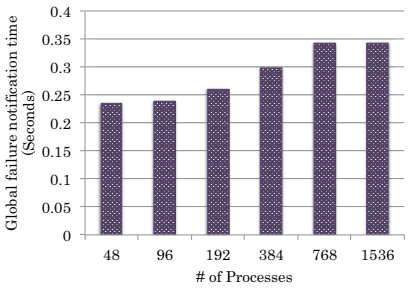


Fig. 13: Failure notification time with *log-ring* overlay network

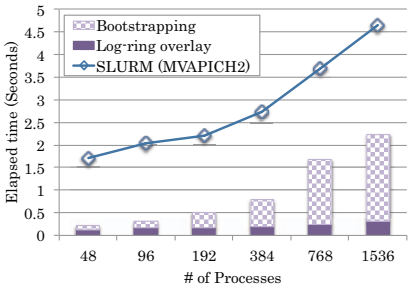


Fig. 14: MPI_Init vs. FMI_Init

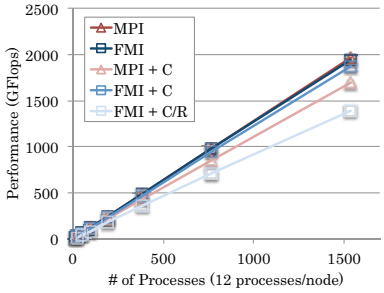


Fig. 15: Himeno benchmark (Checkpoint size: 821 MB/node, MTBF: 1 minute)

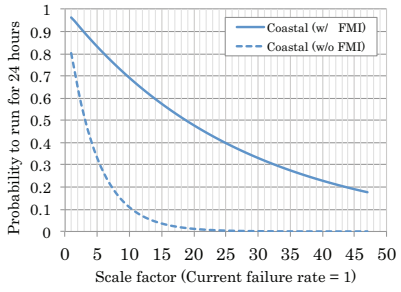


Fig. 16: Probability to continuously run for 24 hours

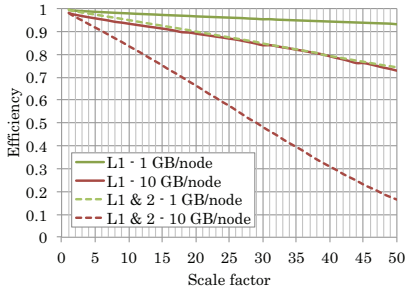


Fig. 17: Efficiency of multilevel C/R under increasing failure rate and L2 C/R time

run continuously with or without FMI, we simulated an application running at extreme scale. If we assume failures occur according to Poisson's distribution, the probability that an application runs for time T continuously is $e^{-\lambda \cdot T}$ where λ is the unrecoverable failure rate.

Figure 16 shows the probability to run continuously for 24 hours using failure rates from the LLNL failure analysis of the Coastal cluster [4], with a level-1 failure rate of 2.13^{-6} (MTBF = 130 hours) (recoverable by XOR encoding), and level-2 failure rate of 4.27^{-7} (MTBF = 650 hours) (unrecoverable failures). We increase the failure rates from the observed level-1 and 2 values by scale factors of 1 (observed values) to 50 to evaluate FMI's performance at larger scales. With FMI, 80% of executions can run for 24 hours with even 6× higher failure rates. At failure rates of 10× higher than today's, 70% of FMI executions can run continuously for 24 hours, while only 10% of non-FMI executions can do the same. Executions without FMI are terminated by any failures, while FMI executions are terminated by only level-2 failure. Thus, using FMI effectively decreases unrecoverable failure rate, λ, and thus the probability of long continuous runs is higher with FMI, even at very high failure rates. At a scale factor 50, the level-1 MTBF becomes 2.6 hours (= 130 hours / 50), which is a quite long MTBF for FMI. As shown in Figure 15, FMI achieves a only 28% overhead (72% of efficiency) even with MTBF of 1 minute. Also, in the absence of unrecoverable failures, an application can run with negligibly small overhead.

If FMI uses a multilevel C/R strategy and writes some checkpoints to the PFS (level-2 C/R) in addition to XOR C/R (level-1 C/R), level-2 failures can also be recoverable. Here, we predict the efficiency of using multilevel C/R with FMI, where efficiency is the ratio of time spent in useful computation only versus computation, C/R activities, and recomputation after recovery. As future systems become larger, we expect higher failure rates and total aggregate checkpoint sizes. Thus, to predict application efficiency at larger scales, we increase failure rates and checkpoint costs up to 50×, using the Coastal system as a base line. Because level-1 C/R time is constant regardless of the total number of nodes, we only increase level-2 C/R time. For level-2 C/R, we assume we write checkpoints asynchronously to the PFS using the framework and model developed in our prior work [16].

Figure 17 shows the efficiency of multilevel C/R in FMI. Because we are uncertain as to whether level-2 failure rates will increase at extreme scale, in our evaluation we increase only the level-1 failure rate (L1) or both the level-1 and 2 failure rates (L1,2) with different checkpoint sizes per node (1 or 10 GB/node). We estimate level-1 C/R time using the performance model in Section V-B. For level-2 C/R time, we use a PFS bandwidth of 50 GB/s, the bandwidth of the LLNL Lustre file system `/p/lscratchd`. We find that we can achieve fairly high efficiencies if future systems can keep current level-2 failure rates constant, or the size of checkpoints is small. However, if both level-1 and 2 failure rates increase and the checkpoint size is large, the efficiency drops down to under 2%. Thus, future systems must either decrease level-2 failure rates or increase PFS throughput to achieve high system efficiency.

VII. Related Work

Fault tolerant messaging runtimes will be critical for applications to be able to recover from failures on future systems. FT-MPI [19] and ULFM (User Level Fault Mitigation) [20] implement fault tolerant capabilities on top of MPI. In these libraries, failures are visible to applications, so users need to write fault tolerant codes, such as communicator recovery and C/R. In other efforts [21], [22], the libraries write checkpoints transparently on top of MPI, but recovery is not transparent to applications. Adaptive MPI [23], developed on top of Charm++ [24], provides transparent recovery using C/R. However, failed processes are relaunched on existing nodes instead of spare nodes, which can cause unbalanced process affinity, and performance degradation after recovery. In contrast, FMI supports these fault tolerance features transparently; applications are agnostic to failures. FMI also provides dynamic node allocation, which is not supported in the current of MPI.

To restore application state at extreme scale where failure rate are expected to be much higher, fast and scalable C/R techniques are indispensable. Diskless checkpointing [25], [26] is a scalable, and fast C/R method because checkpoints are written directly to memory instead of to a reliable PFS. However, the current version of MPI cannot exploit diskless checkpointing because MPI terminates all processes on a failure, and the in-memory checkpoint is lost. Because FMI combines a survivable messaging runtime and diskless checkpointing, FMI can achieve fast, scalable, and transparent recovery. Multilevel C/R [4], [12] is a more sophisticated C/R method. Multilevel checkpointing libraries utilize multiple tiers of storage, such as node-local storage and the PFS, by combining traditional C/R and diskless checkpointing. With Multilevel C/R, FMI can recover from any failures.

We developed the *log-ring* overlay network for fast global failure notification. The network topology itself is similar to Chord [27] in a P2P network. However, the purpose of our *log-ring* overlay network is to provide the functionality of global failure detection and notification.

To the best of our knowledge, FMI is the only messaging library providing a survivable messaging runtime coupled with fast, scalable, in-memory C/R, and dynamic node allocation, which is required for future fault tolerant extreme scale computing.

VIII. Limitations and Future Support

Although the current FMI prototype has demonstrated promising results, it not yet complete enough to support a broad range of applications. Here, we discuss the limitations of our prototype and how we plan to address them.

First, our prototype FMI implementation only supports a subset of MPI functions. For example, collective I/O, i.e., MPI_IO, is an important feature of MPI, because it is often used for C/R to the PFS. Checkpointing to a PFS can be very time consuming, especially at large scales. Additionally, an application can experience much higher failure rates of the PFS than average when there is high load on the PFS. Thus, a checkpoint may never complete due to frequent roll-backs. However, if we create parity data across nodes before initiating the MPI_IO operation, we can restore lost data and continue the I/O operation in the middle without starting over. Thus, support of MPI_IO in FMI is in our plans.

Second, several applications dynamically split a communicator with nested loops in order to balance the workload across processes. Such applications change not only application state but also communicator state over the iterations. To support such applications, future versions of `FMI_Loop` will support C/R of communicators, and nested loops.

Third, our prototype does not support multilevel C/R. FMI cannot recover from multiple nodes failure within XOR group. Future versions of FMI will support multilevel C/R to be able to recover from any failures occurring on HPC systems.

FMI is an on-going project, and future FMI versions will remove the above limitations to support a wider range of applications, and become more resilient.

IX. Conclusion

From our analysis of failures on tera- and peta-scale systems, we have identified four critical capabilities for resilience with extreme-scale computing: a survivable messaging interface that can run through process faults, fast checkpoint/restart, fast failure detection, and a mechanism to dynamically allocate spare compute resources in the event of hardware failures. To satisfy these requirements, we designed and implemented the FMI, a survivable messaging runtime that uses in-memory C/R, scalable failure detection, and dynamic spare node allocation. With FMI, a developer writes applications using semantics similar to MPI, and the FMI runtime ensures that the application runs through failures by handling the activities needed for fault tolerance. Our implementation of FMI has performance comparable to MPI. Our experiments with a Poisson equation solver show that running with FMI incurs only a 28% overhead with a very high MTBF of 1 minute. By defining a simplified programming model and custom runtime, we find that FMI significantly improves resilience overheads compared to similar existing multilevel checkpointing methods and MPI implementations.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-645209). This work was also supported by Grant-in-Aid for Research Fellow of the Japan Society for the Promotion of Science (JSPS Fellows) 24008253, and Grant-in-Aid for Scientific Research S 23220003.

References

[1] B. Schroeder and G. A. Gibson, "Understanding Failures in Petascale Computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, pp. 012 022+, Jul. 2007. [Online]. Available: http://dx.doi.org/10.1088/1742-6596/78/1/012022

[2] A. Geist and C. Engelmann, "Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors," 2002.

[3] J. Daly *et al.*, "Inter-Agency Workshop on HPC Resilience at Extreme Scale," February 2012. [Online]. Available: http://institutes.lanl.gov/resilience/docs/Inter-AgencyResilienceReport.pdf

[4] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, Nov. 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/sc.2010.18

[5] "MPI Forum." [Online]. Available: http://www.mpi-forum.org/

[6] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-Forwarding Infrastructure for Petascale Architectures," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 153–162.

[7] R. Ross, J. Moreira, K. Cupps, and W. Pfeiffer, "Parallel I/O on the IBM Blue Gene/L System," Blue Gene/L Consortium Quarterly Newsletter, Tech. Rep., First Quarter, 2006.

[8] B. Schroeder and G. A. Gibson, "Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you?" in *Proceedings of the 5th USENIX conference on File and Storage Technologies*, ser. FAST '07. Berkeley, CA, USA: USENIX Association, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267903.1267904

[9] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: a state-of-the-art review," *IBM J. Res. Dev.*, vol. 28, no. 2, pp. 124–134, Mar. 1984. [Online]. Available: http://dx.doi.org/10.1147/rd.282.0124

[10] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '88. New York, NY, USA: ACM, 1988, pp. 109–116. [Online]. Available: http://dx.doi.org/10.1145/50202.50214

[11] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka, "Design and Modeling of a Non-Blocking Checkpointing System," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012. [Online]. Available: http://portal.acm.org/citation.cfm?id=2389022

[12] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka, "FTI: high performance Fault Tolerance Interface for hybrid systems," in *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, WS, USA, 2011.

[13] N. H. Vaidya, "On Checkpoint Latency," College Station, TX, USA, Tech. Rep., 1995. [Online]. Available: http://portal.acm.org/citation.cfm?id=892900

[14] R. L. Graham, R. Brightwell, B. Barrett, G. Bosilca, and Pjesivac-Grbović, "An Evaluation of Open MPI's Matching Transport Layer on the Cray XT," Oct 2007.

[15] "PMGR_COLLECTIVE." [Online]. Available: http://sourceforge.net/projects/pmgrcollective/

[16] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. de Supinski, N. Maruyama, and S. Matsuoka, "Design and Modeling of a Non-Blocking Checkpoint System," in *ATIP - A*CRC Workshop on Accelerator Technologies in High Performance Computing*, May 2012.

[17] A. Yoo, M. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Springer Berlin Heidelberg, 2003, vol. 2862, pp. 44–60. [Online]. Available: http://dx.doi.org/10.1007/10968987_3

[18] R. Himeno, "Himeno Benchmark," http://accc.riken.jp/HPC_e/himenobmt_e.html.

[19] G. E. Fagg and J. Dongarra, "FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World," in *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 2000, pp. 346–353. [Online]. Available: http://portal.acm.org/citation.cfm?id=746632

[20] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in mpi," in *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 193–203. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33518-1_24

[21] Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-transparent checkpoint/restart for mpi programs over infiniband," in *In ICPP 06: Proceedings of the 35th International Conference on Parallel Processing*. IEEE Computer Society, 2006, pp. 471–478.

[22] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine, "The lam/mpi checkpoint/restart framework: System-initiated checkpointing," in *in Proceedings, LACSI Symposium, Sante Fe*, 2003, pp. 479–493.

[23] C. Huang, O. Lawlor, and L. V. Kal, "Adaptive mpi," in *In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03*, 2003, pp. 306–322.

[24] G. Zheng, L. Shi, and L. V. Kale, "FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, ser. CLUSTER '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 93–103. [Online]. Available: http://portal.acm.org/citation.cfm?id=1111712

[25] L. A. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed Diskless Checkpoint for Large Scale Systems," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. IEEE, May 2010, pp. 63–72. [Online]. Available: http://dx.doi.org/10.1109/ccgrid.2010.40

[26] J. S. Plank, K. Li, and M. A. Puening, "Diskless Checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 972–986, Oct. 1998. [Online]. Available: http://dx.doi.org/10.1109/71.730527

[27] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160. [Online]. Available: http://doi.acm.org/10.1145/383059.383071