FMI: Fault Tolerant Messaging Interface for Fast and Transparent Recovery

Kento Sato[†1], Adam Moody[†2], Kathryn Mohror[†2], Todd Gamblin[†2], Bronis R. de Supinski[†2], Naoya Maruyama[†3] and Satoshi Matsuoka[†1]

†1 Tokyo Institute of Technology
†2 Lawrence Livermore National Laboratory
†3 RIKEN Advanced institute for Computational Science

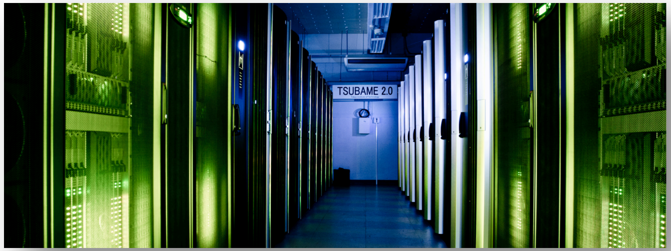
This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52- 07NA27344. LLNL-PRES-654621

Failures on HPC systems

Scientific discovery

Supercomputers enable larger and higher-fidelity simulations by communication libraries

The TSUBAME supercomputer



System failure

TSUBAME2.0 experienced 962 node failures for 1.5 years (MTBF = 13 hours)

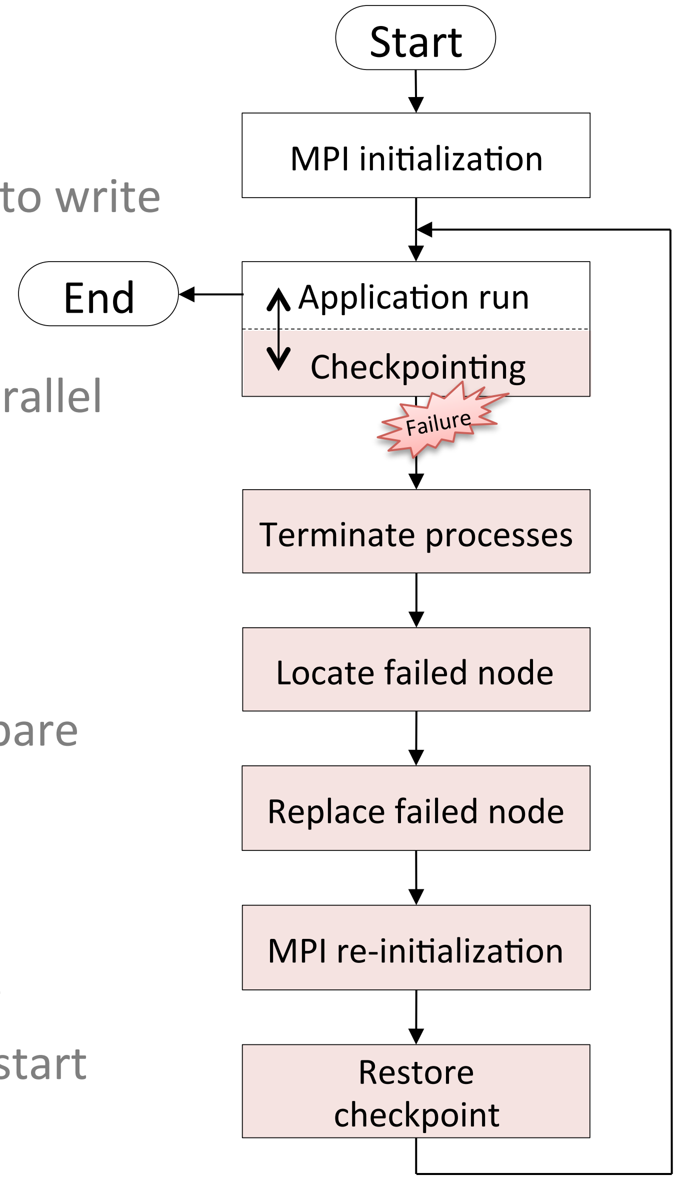
TSUBAME MTBF

Failure type	MTBF
PFS, Core switch	65.10 days
Rack	86.90 days
Edge switch	17.37 days
PSU	28.94 days
Compute node	0.658 days

Failures are already not exceptional but usual events

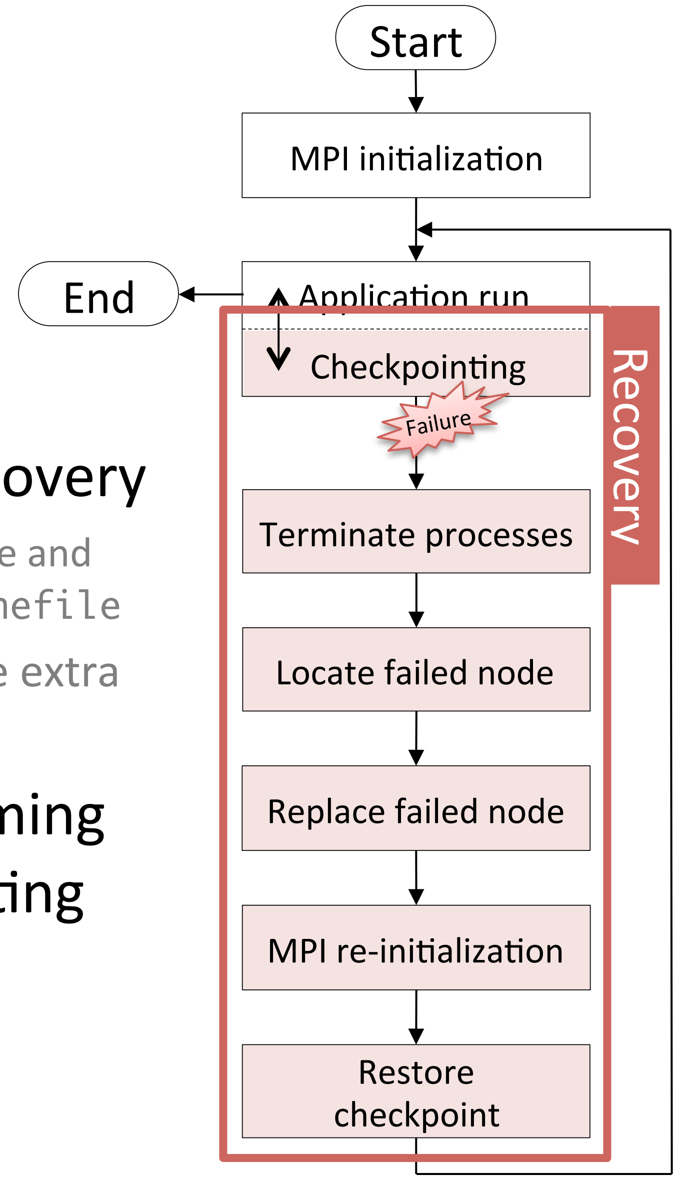
Conventional fault tolerance in MPI apps

- Checkpoint/Recovery (C/R)
  - Long running MPI applications are required to write checkpoints
- MPI
  - De-facto communication library enabling parallel computing
  - Standard MPI employs a fail-stop model
- When a failure occurs …
  - MPI terminates all processes
  - The user locate, replace failed nodes with spare nodes
  - Re-initialize MPI
  - Restore the last checkpoint
- The fail-stop model of MPI is quite simple
  - All processes synchronize at each step to restart



Requirement of fast and transparent recovery

- Failure rate will increase in future extreme scale systems
- Applications will use more time for recovery
  - Whenever a failure occurs, users manually locate and replace the failed nodes with spare nodes via `machinefile`
  - The manual recovery operations may introduce extra overhead and human errors
- Fast and transparent recovery is becoming more critical for extreme scale computing



Goal and Contributions

- Goal:
  - Fast and Transparent recovery for extreme scale computing

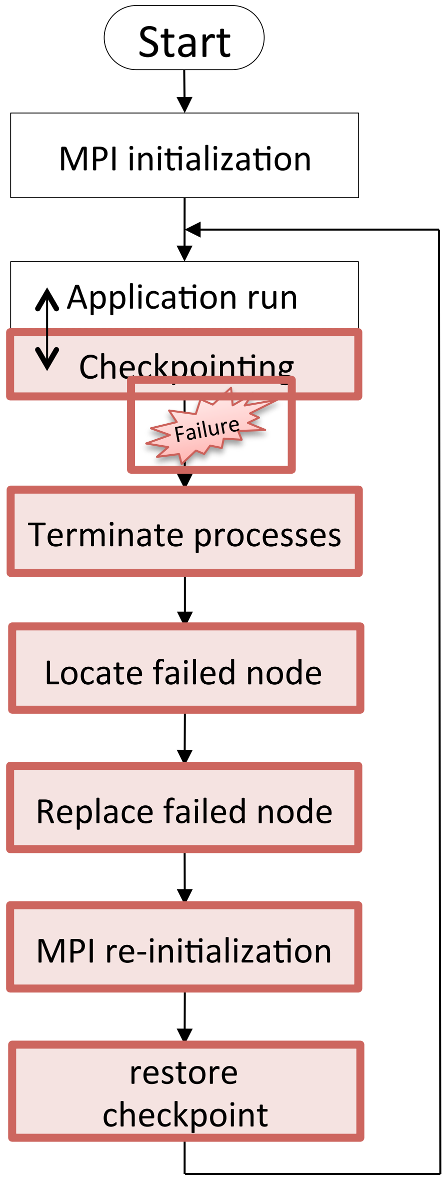
- Contributions:
  - We developed Fault Tolerant Messaging Interface (FMI) enabling fast and transparent recovery
  - Experimental results show FMI incurs only a **28% overhead** with a very high **MTBF of 1 minute**

Outline

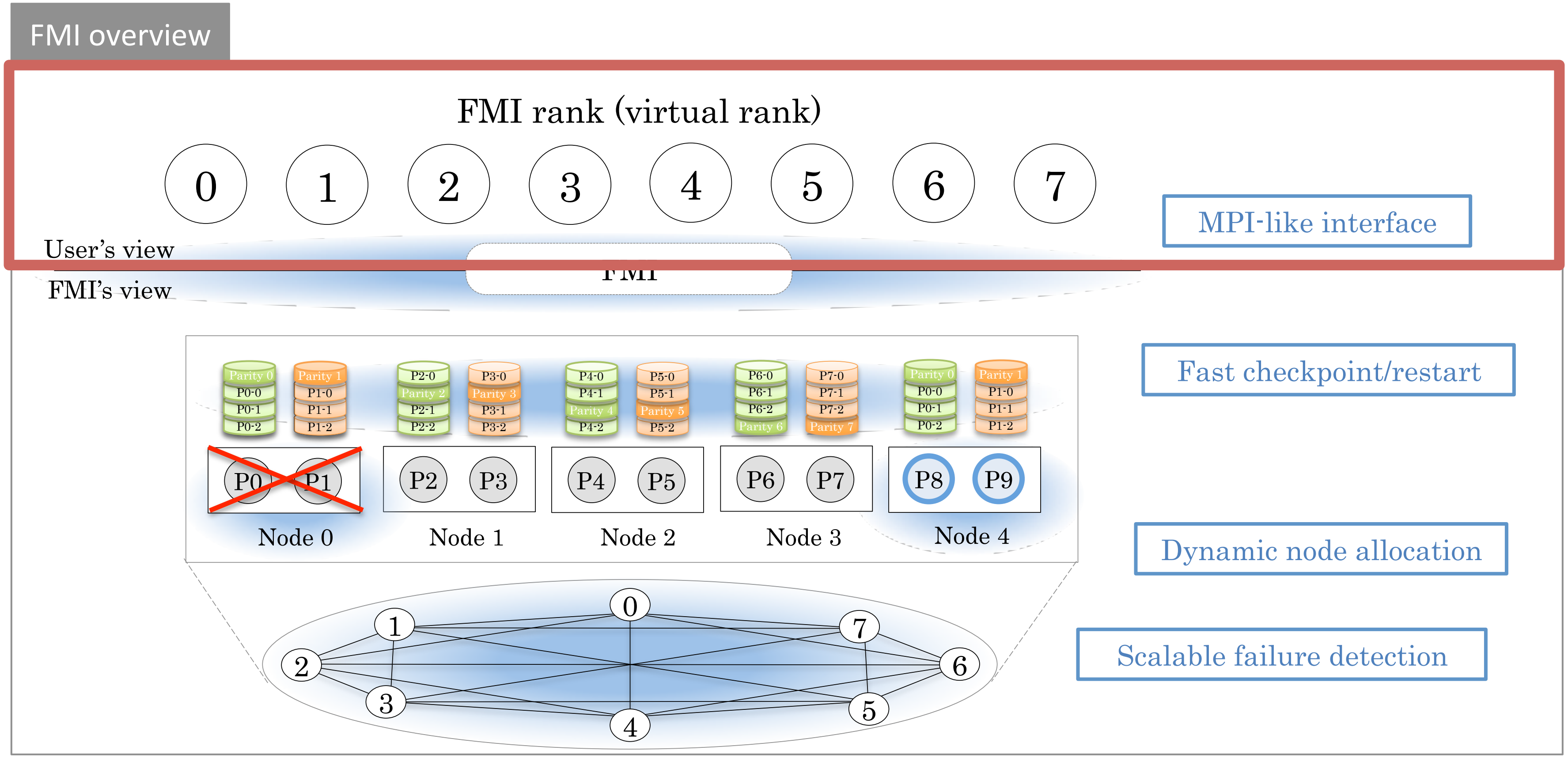
- Introduction
- Challenges for fast and transparent recovery
- FMI: Fault Tolerant Messaging Interface
  - User perspective
  - Internal implementation
- Evaluation
- Conclusion

Challenges for fast and transparent recovery

- Scalable failure detection
  - When recovering from a failure, all processes need to be notified
- Survivable messaging interface
  - At extreme scale, even termination and Initialization of processes will be expensive
  - Not terminating non-failed processes is important
- Transparent and dynamic node allocation
  - Manually locating, and replacing failed nodes will introduce extra overhead and human errors
- Fast checkpoint/restart



FMI: Fault Tolerant Messaging Interface



- FMI is a survivable messaging interface providing MPI-like interface
  - Scalable failure detection => Overlay network
  - Dynamic node allocation => FMI ranks are virtualized
  - Fast checkpoint/restart => Diskless checkpoint/restart

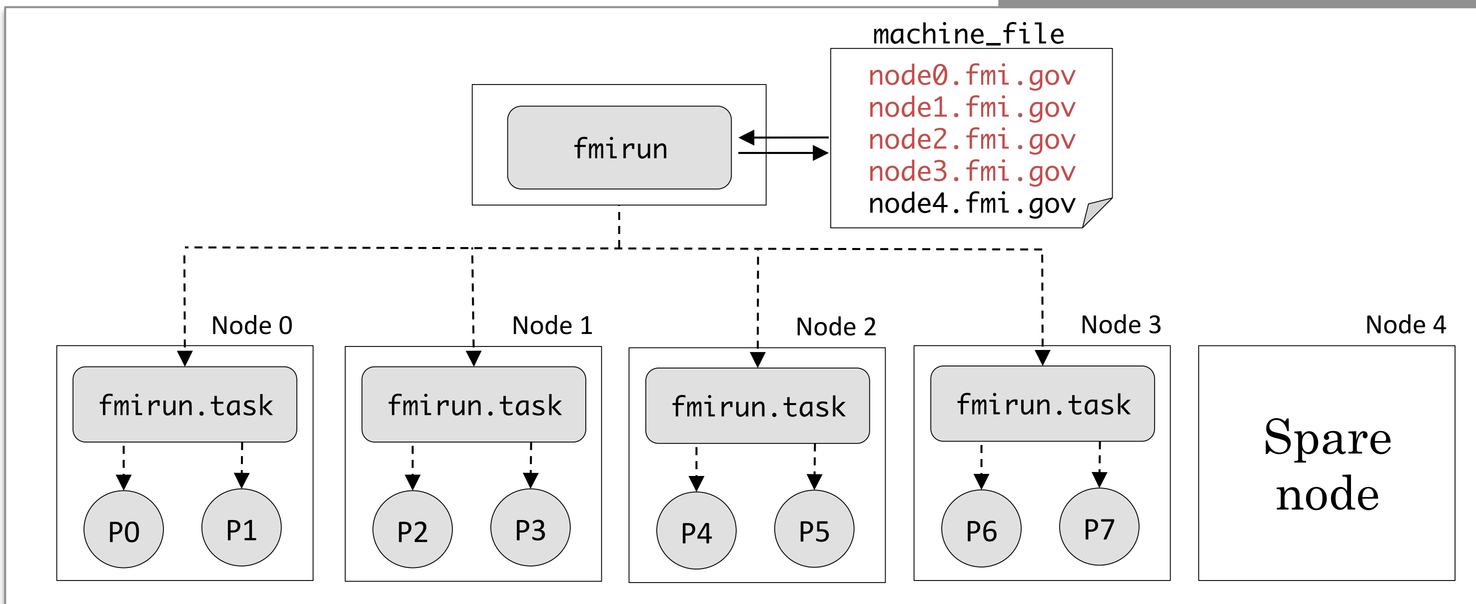
How FMI applications work ?

FMI example code

```
int main (int *argc, char *argv[]) {
  FMI_Init(&argc, &argv);
  FMI_Comm_rank(FMI_COMM_WORLD, &rank);
  /* Application's initialization */
  while ((n = FMI_Loop(…)) < numloop) {
    /* Application's program */
  }
  /* Application's finalization */
  FMI_Finalize();
}
```

- FMI_Loop enables transparent recovery and roll-back on a failure
  - Periodically write a checkpoint
  - Restore the last checkpoint on a failure
- Processes are launched via `fmirun`
  - `fmirun` spawns `fmirun.task` on each node
  - `fmirun.task` calls fork/exec a user program
  - `fmirun` broadcasts connection information (endpoints) for FMI_init(…)

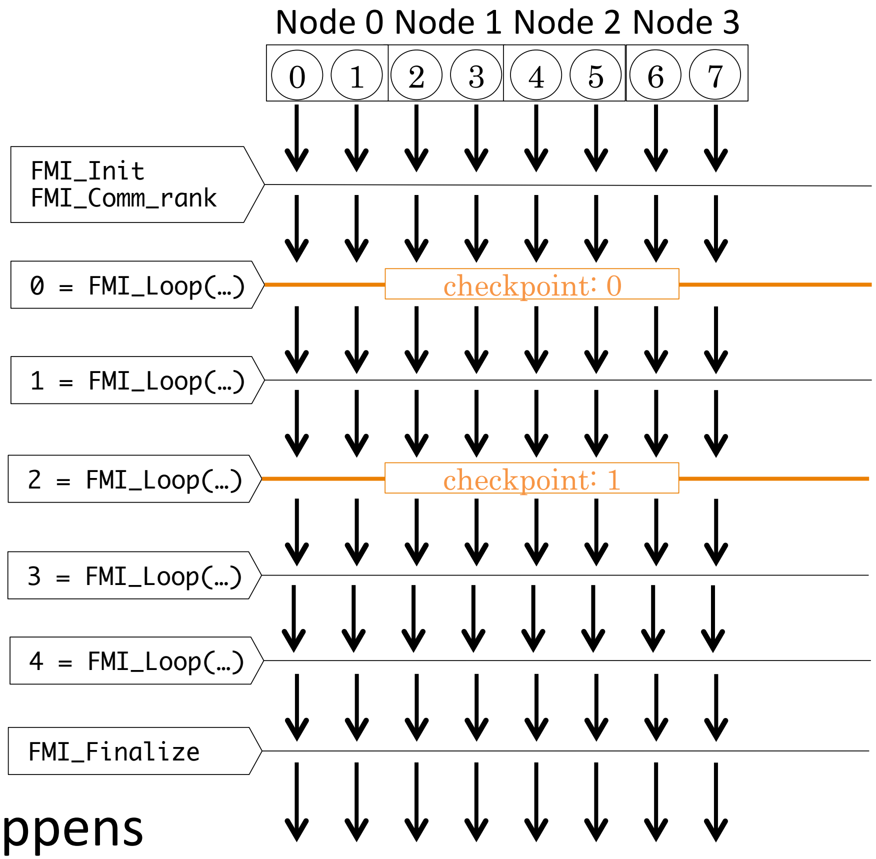
Launch FMI processes



User perspective: No failures

FMI example code

```
int main (int *argc, char *argv[]) {
  FMI_Init(&argc, &argv);
  FMI_Comm_rank(FMI_COMM_WORLD, &rank);
  /* Application's initialization */
  while ((n = FMI_Loop(…)) < 4) {
    /* Application's program */
  }
  /* Application's finalization */
  FMI_Finalize();
}
```

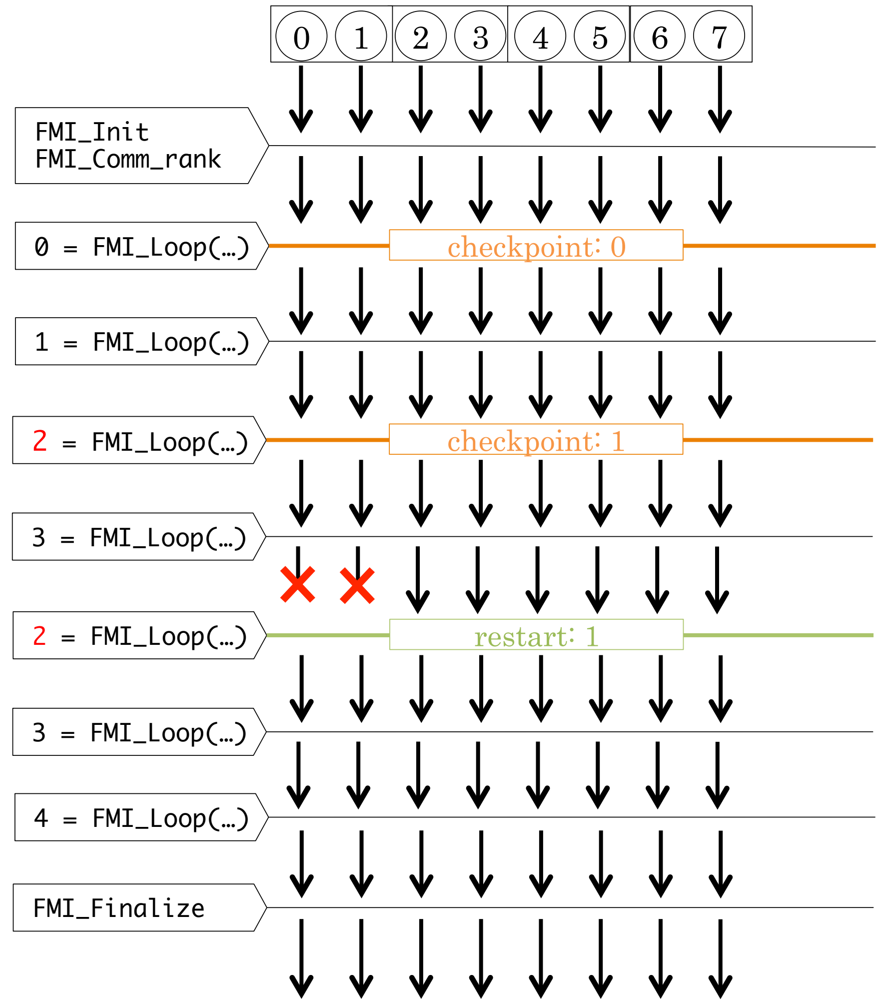


- User perspective when no failures happens
- Iterations: 4
- Checkpoint frequency: Every 2 iterations
- FMI_Loop returns incremented iteration id

User perspective : Failure

FMI example code

```
int main (int *argc, char *argv[]) {
  FMI_Init(&argc, &argv);
  FMI_Comm_rank(FMI_COMM_WORLD, &rank);
  /* Application's initialization */
  while ((n = FMI_Loop(…)) < 4) {
    /* Application's program */
  }
  /* Application's finalization */
  FMI_Finalize();
}
```



- Transparently migrate FMI rank 0 & 1 to a spare node
- Restart form the last checkpoint
  - 2th checkpoint at iteration 2
- With FMI, applications still use the same series of ranks even after failures

FMI_Loop

```
int FMI_Loop(void **ckpt, size_t *sizes, int len)
```

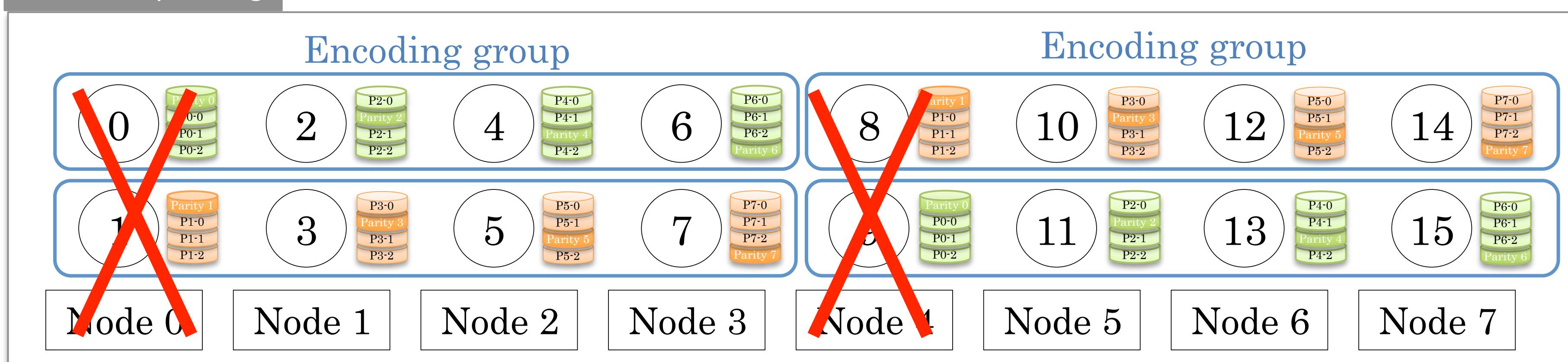
- `ckpt` : Array of pointers to variables containing data that needs to be checkpointed
- `sizes`: Array of sizes of each checkpointed variables
- `len` : Length of arrays, `ckpt` and `sizes`

returns iteration id

- Checkpoint interval
  - Fixed mode: Writing checkpoints every specified iterations
  - Adaptive mode: Checkpoint interval is optimized to maximize efficiency based on Vaidya's model*
- FMI constructs in-memory RAID-5
- Checkpoint group size
  - e.g.) group_size = 4

*N. H. Vaidya, "On Checkpoint Latency,"

FMI checkpointing



FMI_Loop

```
int FMI_Loop(void **ckpt, size_t *sizes, int len)
```

`ckpt` : Array of pointers to variables containing data that needs to be checkpointed

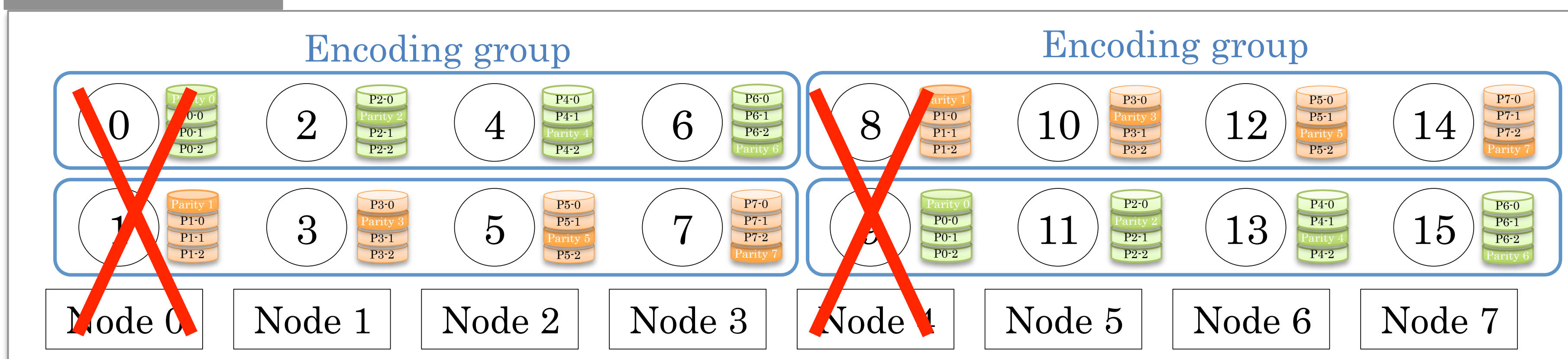
`sizes`: Array of sizes of each checkpointed variables

`len` : Length of arrays, `ckpt` and `sizes`

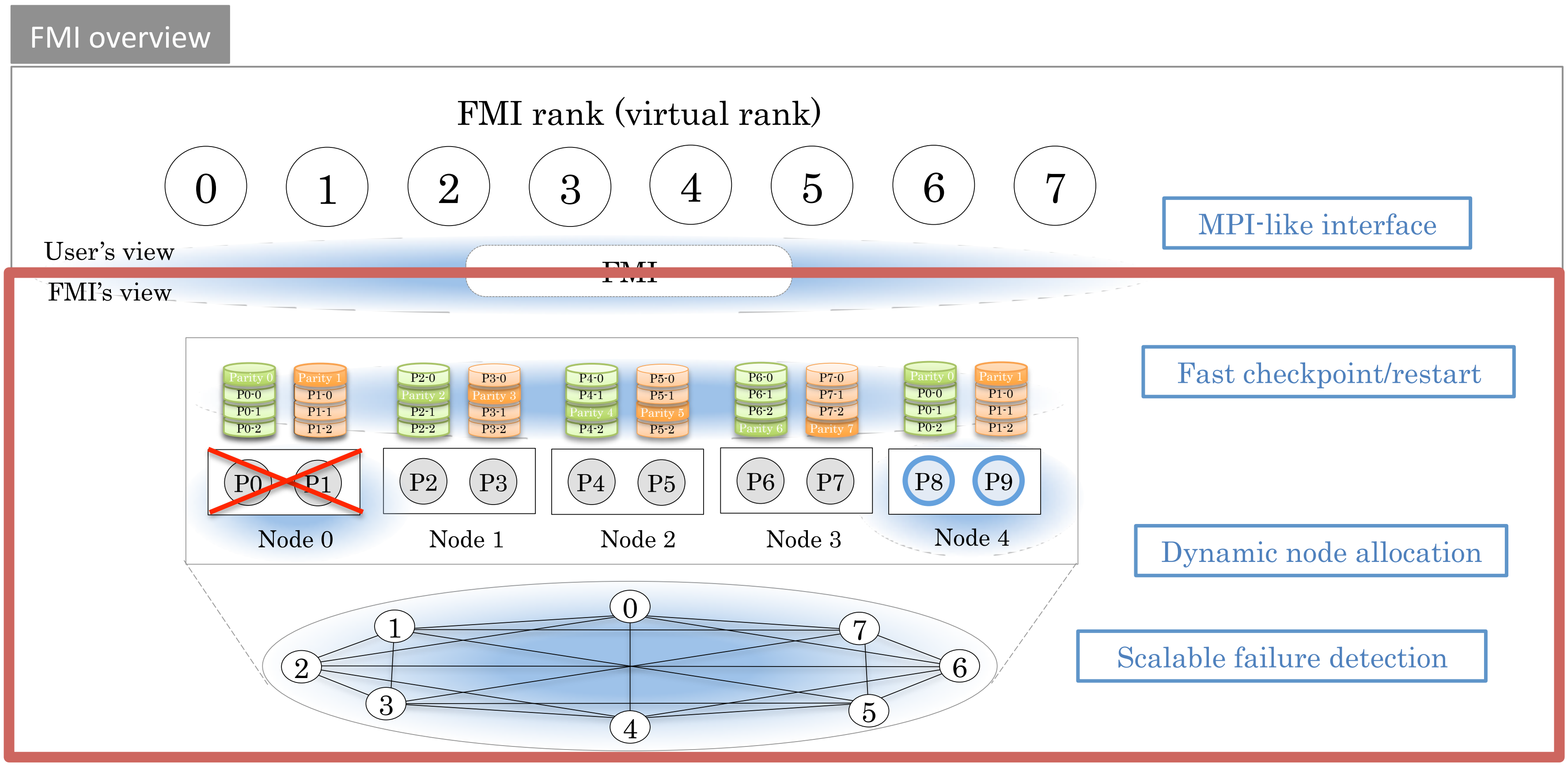
`returns iteration id`

- FMI constructs in-memory RAID-5 across compute nodes
- Checkpoint group size
  - e.g.) group_size = 4

FMI checkpointing

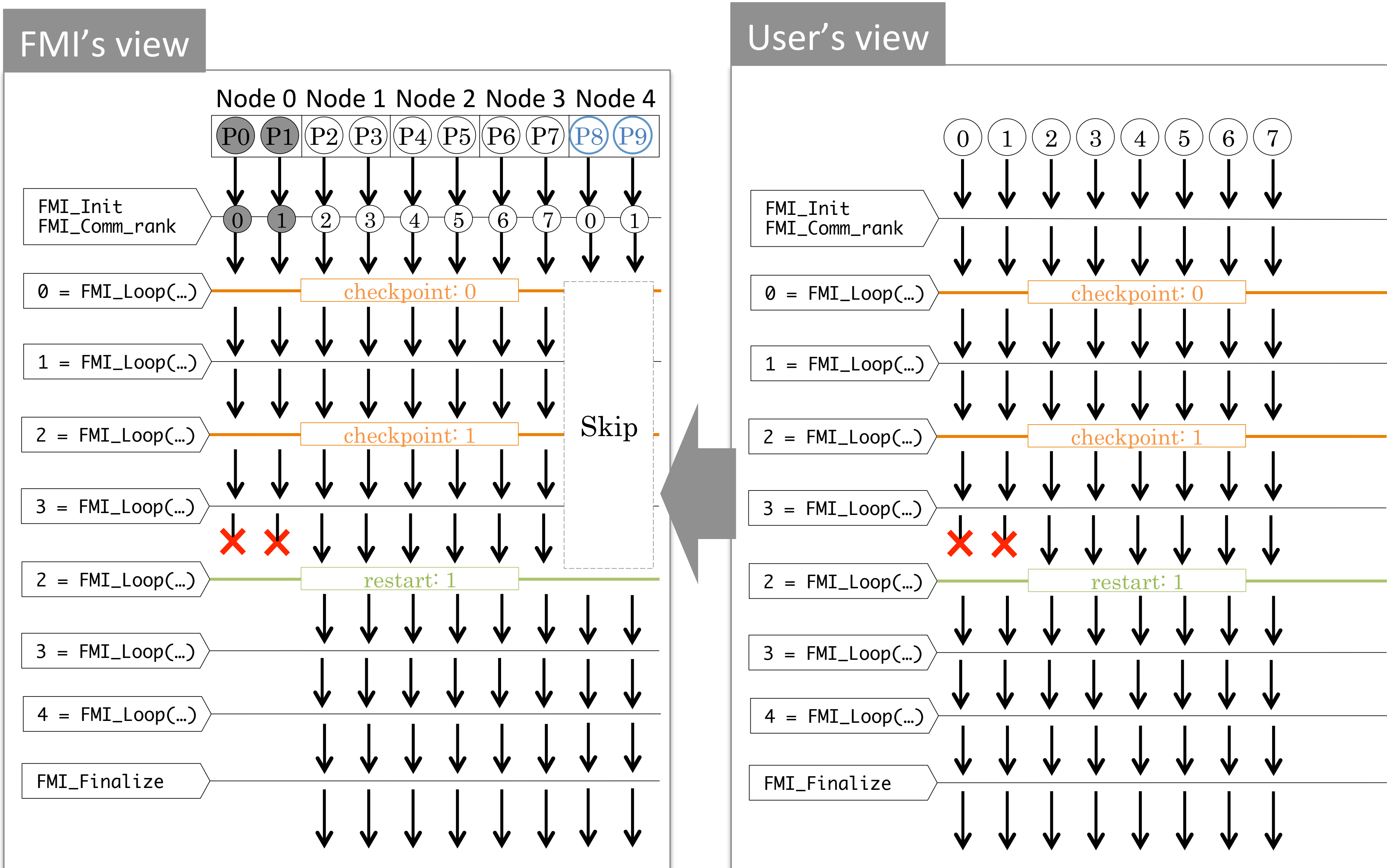


FMI: Fault Tolerant Messaging Interface

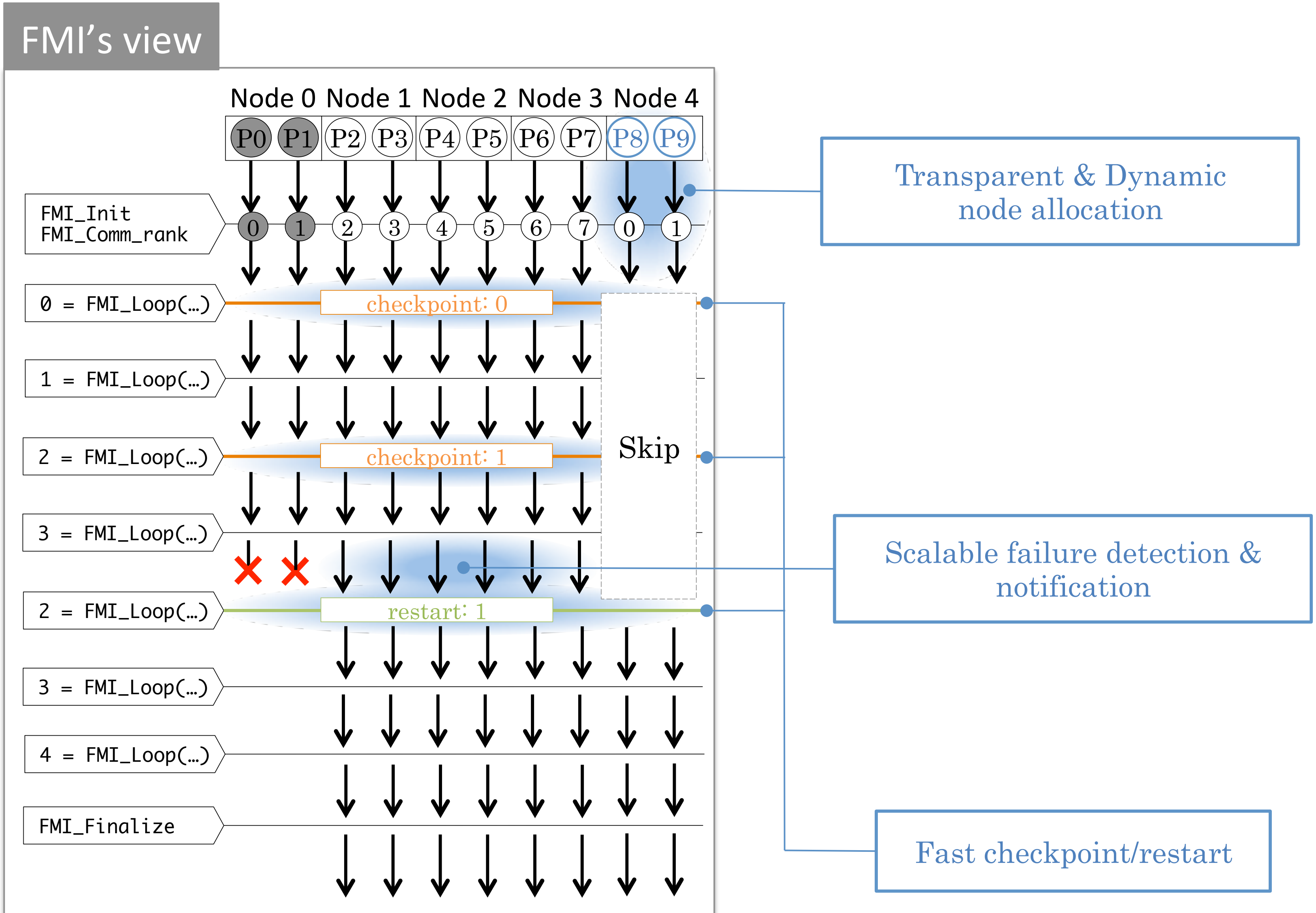


- FMI is an MPI-like survivable messaging interface
  - Scalable failure detection => Overlay network for failure detection
  - Dynamic node allocation => FMI ranks are virtualized
  - Fast checkpoint/restart => Diskless checkpoint/restart

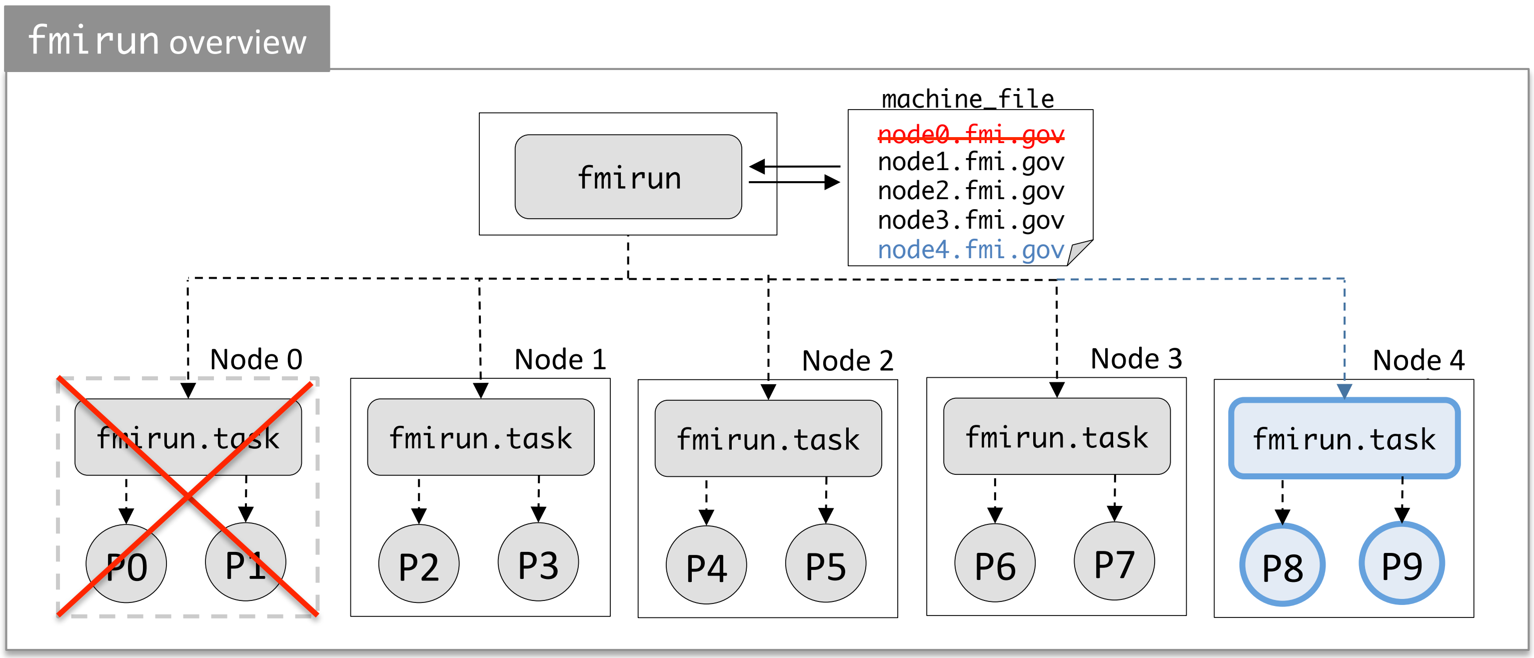
FMI's view & User's view



FMI's view


FMI's view
Node 0 Node 1 Node 2 Node 3 Node 4
P0 P1 P2 P3 P4 P5 P6 P7 P8 P9
FMI_Init
FMI_Comm_rank
0 1 2 3 4 5 6 7 0 1
0 = FMI_Loop(…)
checkpoint: 0
1 = FMI_Loop(…)
2 = FMI_Loop(…)
checkpoint: 1
3 = FMI_Loop(…)
Skip
2 = FMI_Loop(…)
restart: 1
3 = FMI_Loop(…)
4 = FMI_Loop(…)
FMI_Finalize
Transparent & Dynamic node allocation
Scalable failure detection & notification
Fast checkpoint/restart

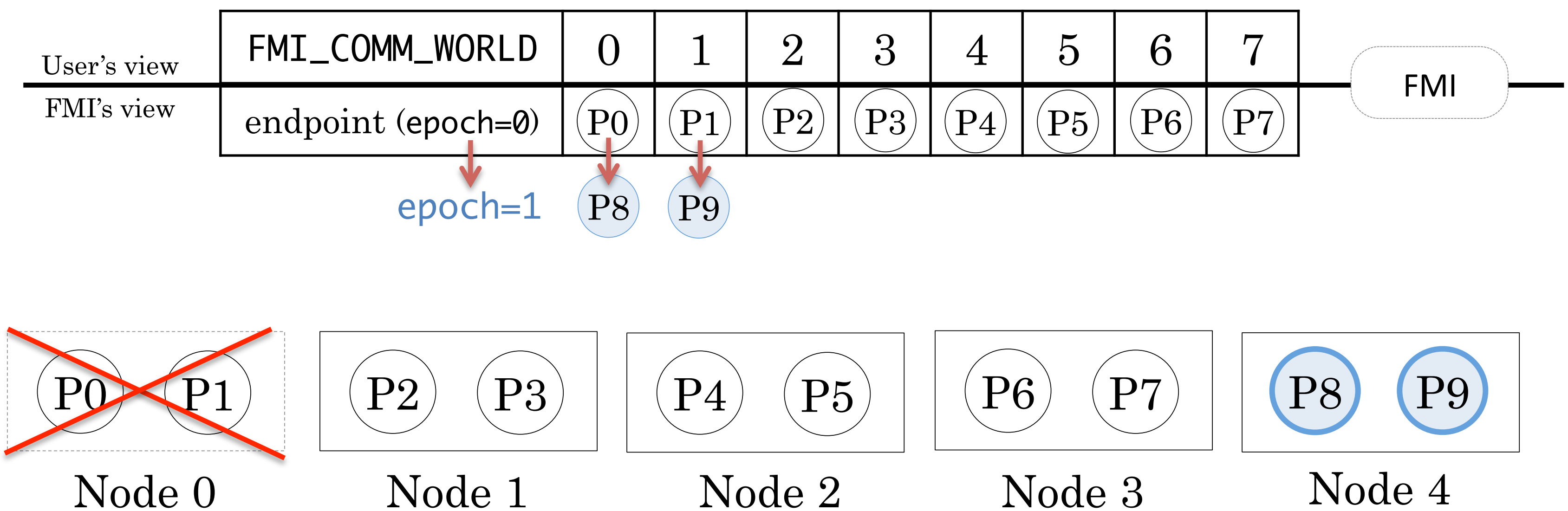
Transparent and dynamic node allocation



- If fmirun.task receives an unsuccessful exit signal from a child process
  - `fmirun.task` kills any other running child processes in the node, and exits with `EXIT_FAILURE`
- When fmirun receives the EXIT_FAILURE from the fmirun.task,
  - `fmirun` attempts to find spare nodes to replace the failed nodes in the `machine_file`
  - `fmirun` spawns new processes on the spare nodes
- `fmirun` boradcasts connection information (endpoint) of new processes, P8 and P9

Transparent and dynamic node allocation (cont'd)

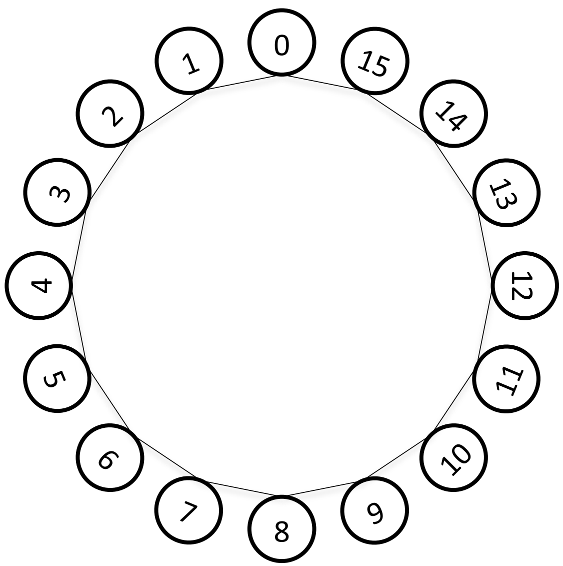
- In FMI, FMI_COMM_WORLD manages process mapping between FMI ranks and processes
  - Once receiving endpoints, the mapping table is updated (=> bootstrapping)
    - Applications can still use the same ranks
  - Then, increment a "epoch" number to be able to discard staled messages
    - After recovery, processes may receive old data which is sent before a failure happens



Scalable failure detection

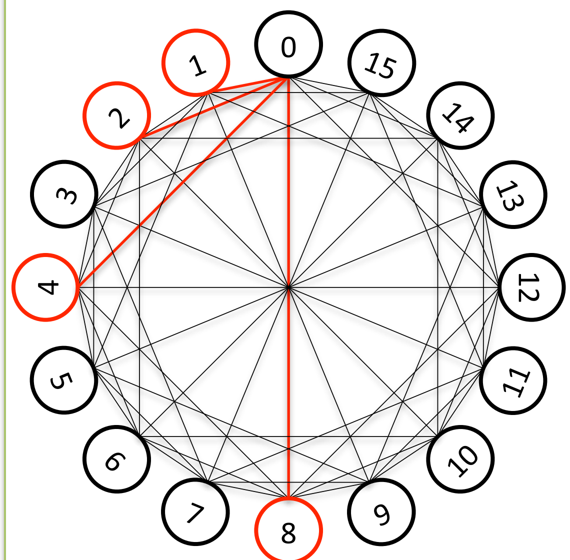
- FMI processes check if other processes are alive or not each other using overlay network
- Log-ring overlay network
  - Each FMI rank connects to 2^k-hop neighbors ($k = 0,1\ldots$)
  - e.g.) FMI rank 0 connects to FMI rank 1, 2, 4 and 8
- Log-ring overlay is scalable for both construction and detection

Ring overlay



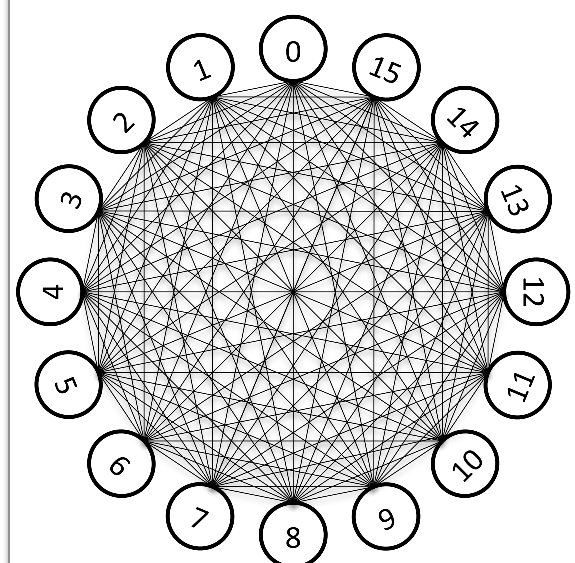
Construction: $O(1)$
Global detection: $O(N)$

Log-ring overlay



Construction: $O(\log N)$
Global detection: $O(\log N)$

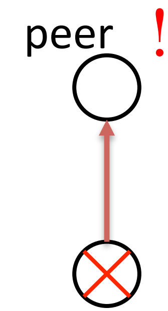
Complete overlay



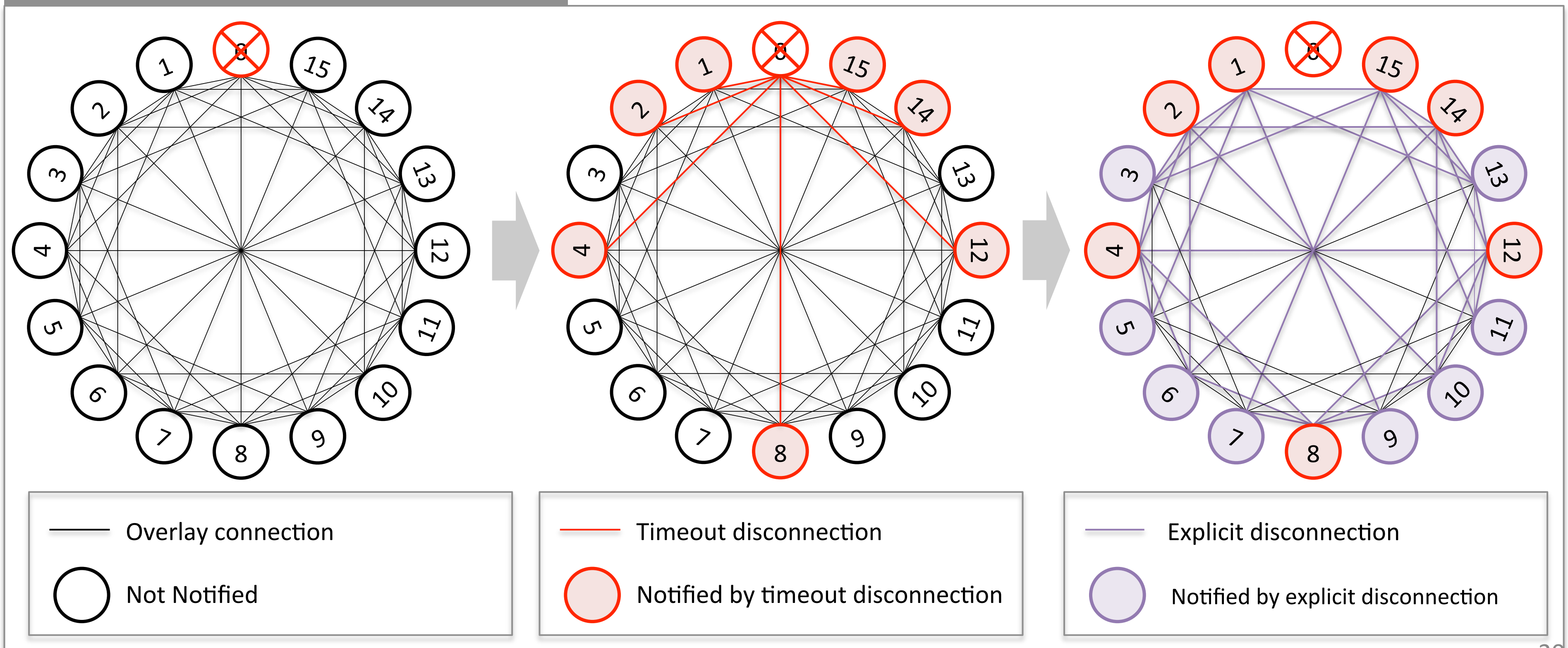
Construction: $O(N)$
Global detection: $O(1)$

Scalable failure detection (cont'd)

- Log-ring overlay network using ibverbs
  - Connection-based communication: if a process is terminated, the peer processes receive the disconnection event
- FMI global failure notification
  - When FMI processes receive disconnection events, the processes explicitly disconnect all of ibverbs connections

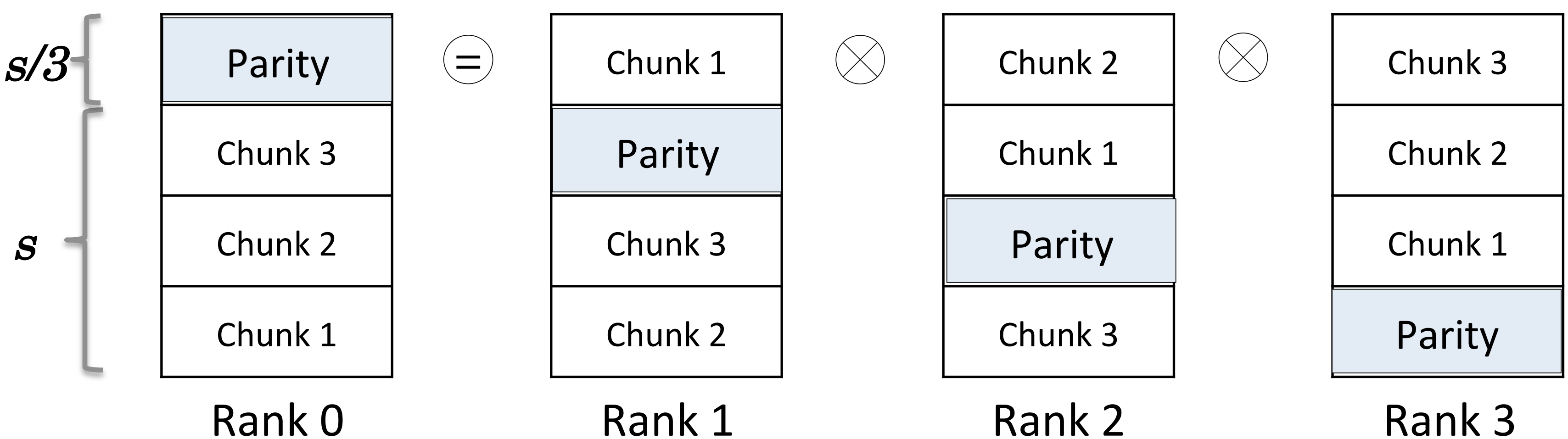


Example of global failure notification



In-memory XOR checkpoint/restart algorithm

- XOR checkpoint/restart algorithm
  1. Write checkpoint using `memcpy`
  2. Divides into chunks, and allocate memory for party data
  3. Send parity data to one neighbor, receive parity data from the other neighbor, and compute XOR
  4. Continue 3. until first parity come back
  5. (For restart) gather all restored data



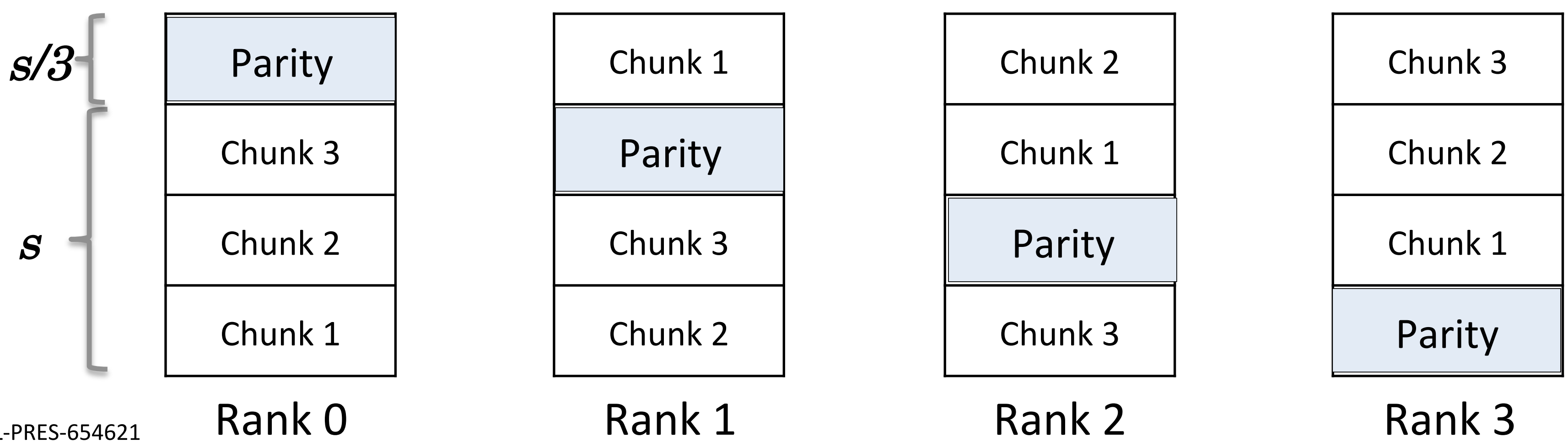
Source: A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10).

In-memory XOR checkpoint/restart model

- In-memory XOR checkpoint/restart time depends on only XOR group size

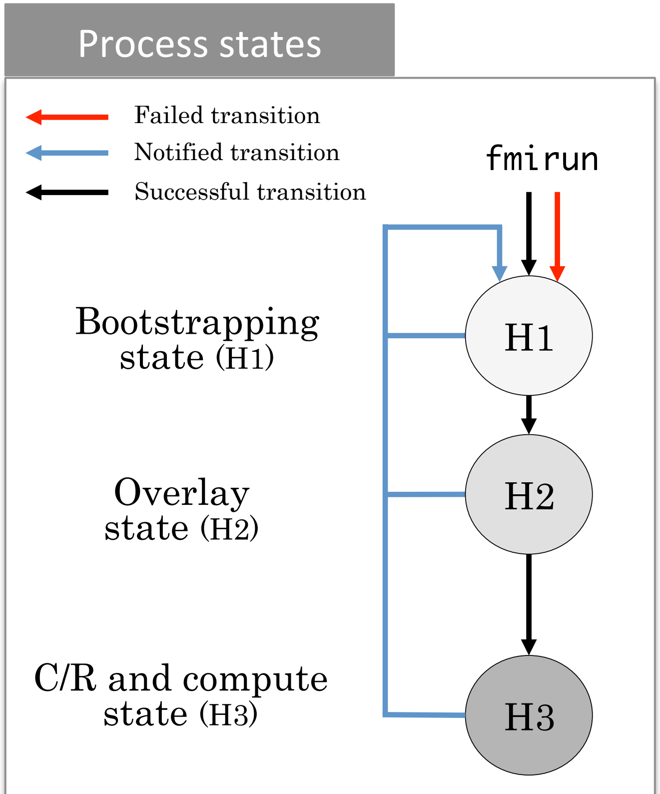
$\boldsymbol{s}$: ckpt size, $\boldsymbol{n}$: group size, $\boldsymbol{mem\_bw}$: memory bandwidth, $\boldsymbol{net\_bw}$: network bandwidth

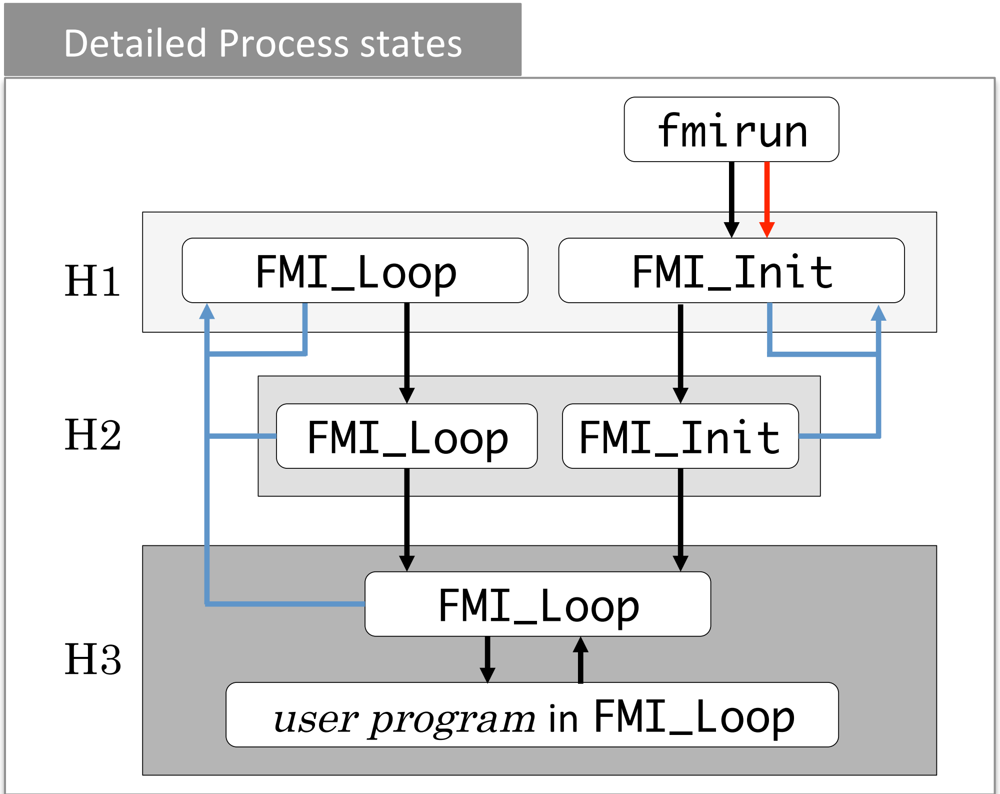
	memcpy	parity transfer	encoding	gathering
Checkpoint $=$	$\frac{s}{mem\_bw}$ $\oplus$	$\frac{s+s/(n-1)}{net\_bw}$ $\oplus$	$\frac{s}{mem\_bw}$	
Restart $=$	$\frac{s}{mem\_bw}$ $\oplus$	$\frac{s+s/(n-1)}{net\_bw}$ $\oplus$	$\frac{s}{mem\_bw}$ $\oplus$	$\frac{s}{net\_bw}$



Process state manage

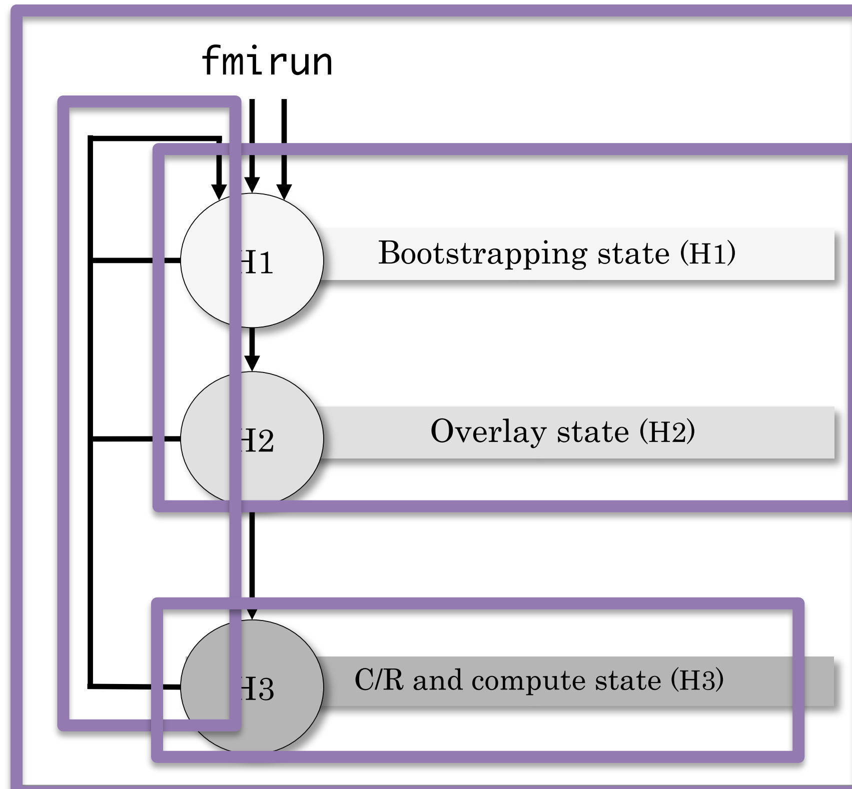
- FMI manages three states to make sure all processes to synchronously
  - H1: Bootstrap for endpoint, process mapping update, and epoch
  - H2: Construct overlay for scalable failure detection
  - H3: Do computation and checkpoint
- Whenever failures happens, all processes transitions to H1 to restart





Evaluations

- Initialization
  - FMI_Init time
- Detection
- Checkpoint/restart
- Benchmark run
- Simulations for extreme scale



LLNL-PRES-654621

Evaluations

- Initialization
  - FMI_Init time
- Detection
- Checkpoint/restart
- Benchmark run
- Simulations for extreme scale

LLNL-PRES-654621

Experimental environment

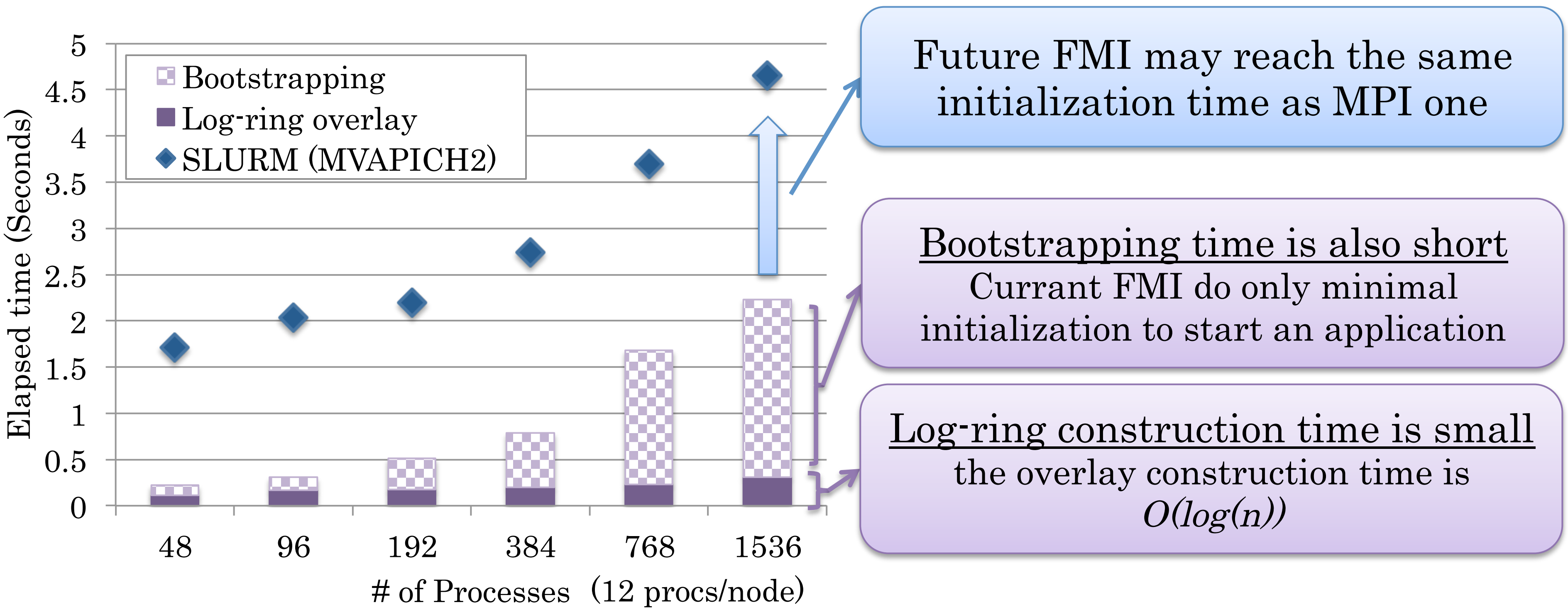
- Sierra cluster @LLNL

Table 4.1: Sierra Cluster Specification

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

- MPI: MVAPICH2 (1.2)
  - Runs on top of SLURM
  - `srun` instead of `mpirun` for launching MPI processes

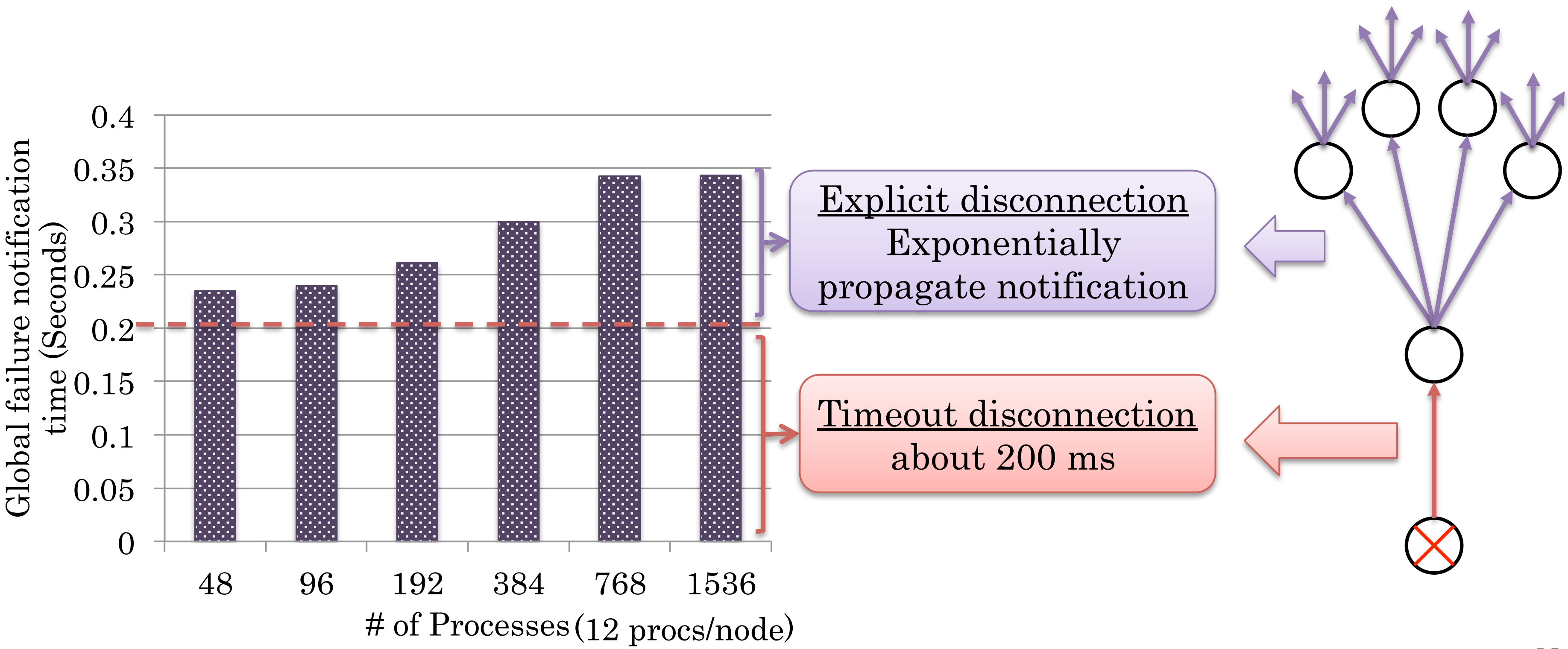
MPI_Init vs. FMI_Init time



MPI Initialization: MVAPICH2 MPI_Init(…) launched by `srun`

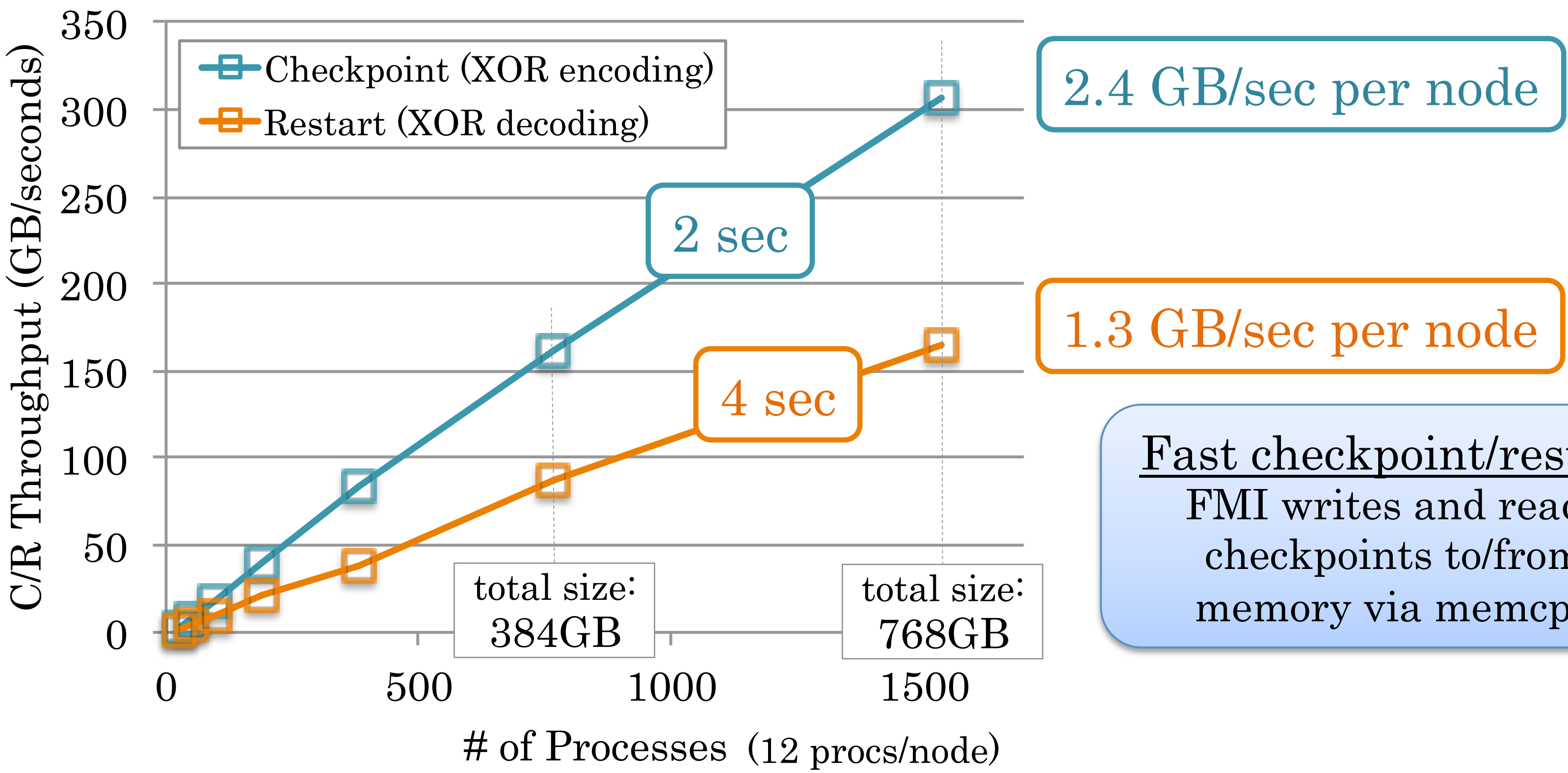
FMI failure detection time

- We measured the time for all processes to be notified of a failure
  - Injected a failure by killing a process
- Once a process receive a disconnection event, the notification exponentially propagate
  - Time complexity: $O\ (log(N))$ to propagate



FMI Checkpoint/Restart throughput

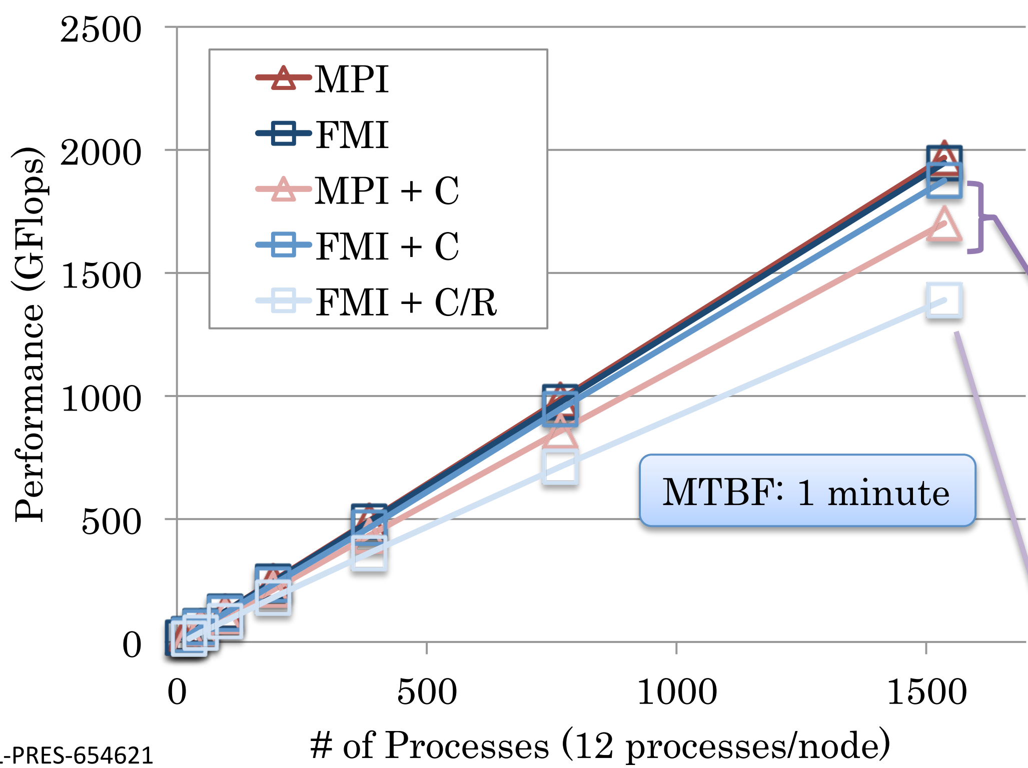
- Checkpoint size: 6GB/node
- The checkpoint/restart time of FMI is scalable
  - FMI directly write checkpoint to memory via memcpy
  - As in the model, the checkpointing and restart times are constant regardless of the total number of processes



Fast checkpoint/restart
FMI writes and reads checkpoints to/from memory via memcpy

Application runtime with failures

- Benchmark: Poisson's equation solver using Jacobi iteration method
  - Stencil application benchmark
  - MPI_Isend, MPI_Irecv, MPI_Wait and MPI_Allreduce within a single iteration
- For MPI, we use the SCR library for checkpointing
  - Since MPI is not survivable messaging interface, we write checkpoint memory on tmpfs
- Checkpoint interval is optimized by Vaidya's model for FMI and MPI



P2P communication performance

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

FMI directly writes checkpoints via memcpy, and can exploit the bandwidth

Even with the high failure rate, FMI incurs only a 28% overhead

Simulations for extreme scale

- FMI applications can continue to run as long as all failures are recoverable. To investigate how long an application can
- run continuously with or without FMI, we simulated an application running at extreme scale.
- Types of failures
  - L1 failure: Recoverable by FMI
  - L2 failure: Unrecoverable by FMI
- We scale out failure rates, evaluate
  1. How long applications can continuously run;
  2. efficiency at extreme scale

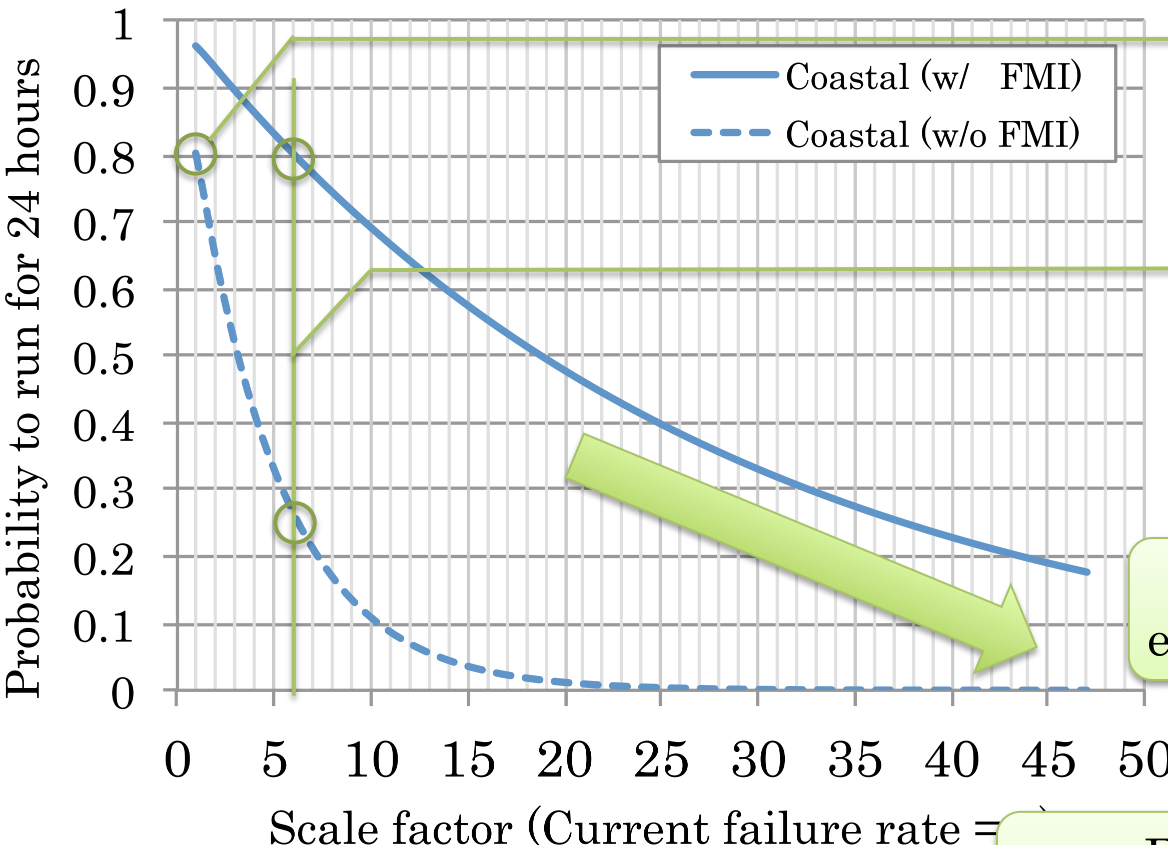
Failure analysis on Coastal cluster

	MTBF	Failure rate
L1 failure	130 hours	2.13^{-6}
L2 failure	650 hours	4.27^{-7}

Source: 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 (SC 10).

Probability to run for 24 hours

- With FMI, application continuously run for longer time



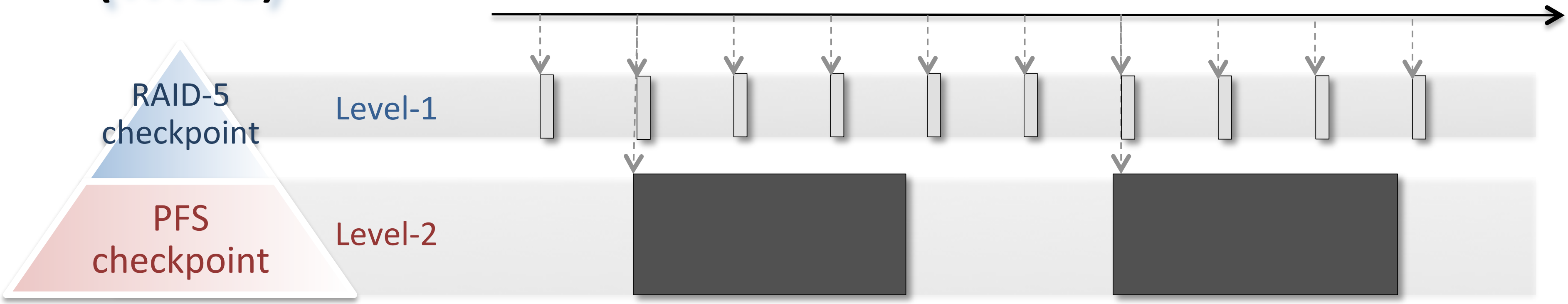
80% of probability to run for 24 hours on environment with current failure rate

FMI execution: 80%
non-FMI execution: 25%

Even with FMI, most of executions cannot run for 24H

Future FMI will support async. multi-level checkpoint/restart

Asynchronous multi-level checkpointing (MLC)



Source: 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

- Asynchronous MLC is a technique for achieving high reliability while reducing checkpointing overhead
- Asynchronous MLC Use storage levels hierarchically
  - RAID-5 checkpoint: Frequent for one node for a few node failure
  - PFS checkpoint: Less frequent and asynchronous for multi-node failure
- Our previous work model the asynchronous MLC

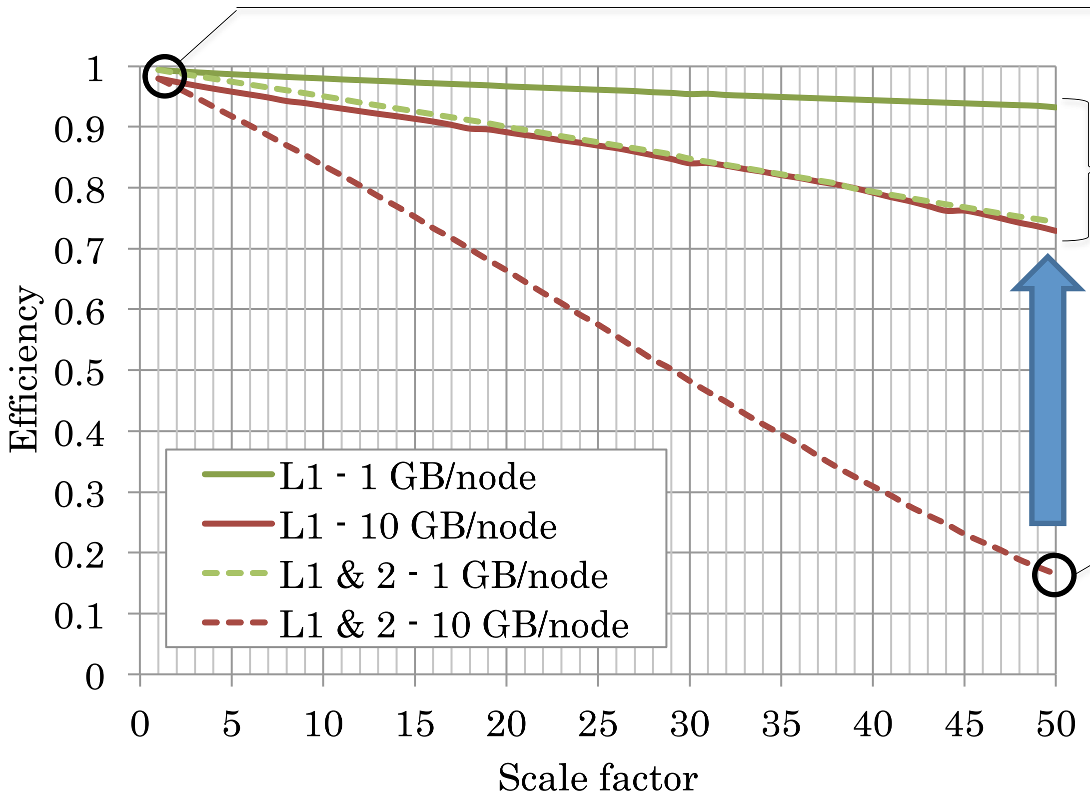
Failure analysis on Coastal cluster

	MTBF	Failure rate
L1 failure	130 hours	2.13^{-6}
L2 failure	650 hours	4.27^{-7}

Source: 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 (SC 10).

Efficiency with FMI + Asynchronous MLC

- Checkpoint size: 1 and 10 GB/node
- We increase L1 and L1 & L2 failure rates



High efficiency with current failure rate

FMI + Asynchronous MLC achieve high efficiency even with much higher failure rate

If both L1 & L2 failure rate increase, and checkpoint size is large, efficiency drops rapidly

Kento Sato, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama and Satoshi Matsuoka, "A User-level InfiniBand-based File System and Checkpoint Strategy for Burst Buffers", CCGrid2014 (May 23th, Best Paper Session)

Limitation and Future support

- FMI is an on-going project, several limitations exist
- Limited MPI functions
  - The current FMI implementation only supports a subset of MPI functions.
  - e.g.) MPI_IO
- C/R of communicators
  - Several applications dynamically split a communicator in order to balance the workloads across processes
  - Such applications change not only application state but also communicator state over the iterations
- Multi-level C/R
  - Future versions of FMI will support multilevel C/R to be able to recover from any failures occurring on HPC systems.

Conclusion

- We developed Fault Tolerant Messaging Interface (FMI) for fast and transparent recovery
  - Scalable failure detection
  - Survivable messaging interface
  - Dynamic node allocation
  - Fast checkpoint/restart

- Experimental results show FMI incurs only a **28% overhead** with a very high **MTBF of 1 minute**
  - The result presents good prospect to implement resilience capability on top of other fault tolerant MPIs (e.g. ULFM & NR-MPI)

Q & A

Speaker:

Kento Sato (佐藤 賢斗)

kent@matsulab.is.titech.ac.jp

Tokyo Institute of Technology (Tokyo Tech)

Research Fellow of the Japan Society for the Promotion of Science

http://matsu-www.is.titech.ac.jp/~kent/index_en.html

Collaborators

Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R de. Supinski,
Naoya Maruyama, Satoshi Matsuoka

Acknowledgement

This work was performed under the auspices of the U.S. Depart- ment 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.