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

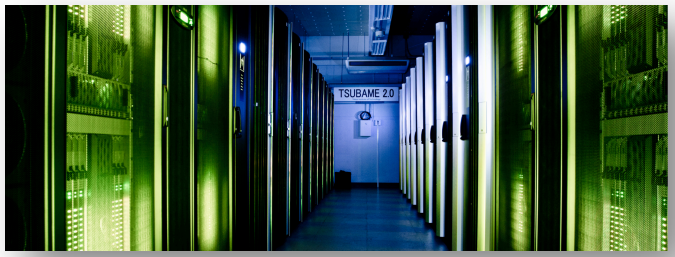
This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52- 07NA27344. LLNL-PRES-654621

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)

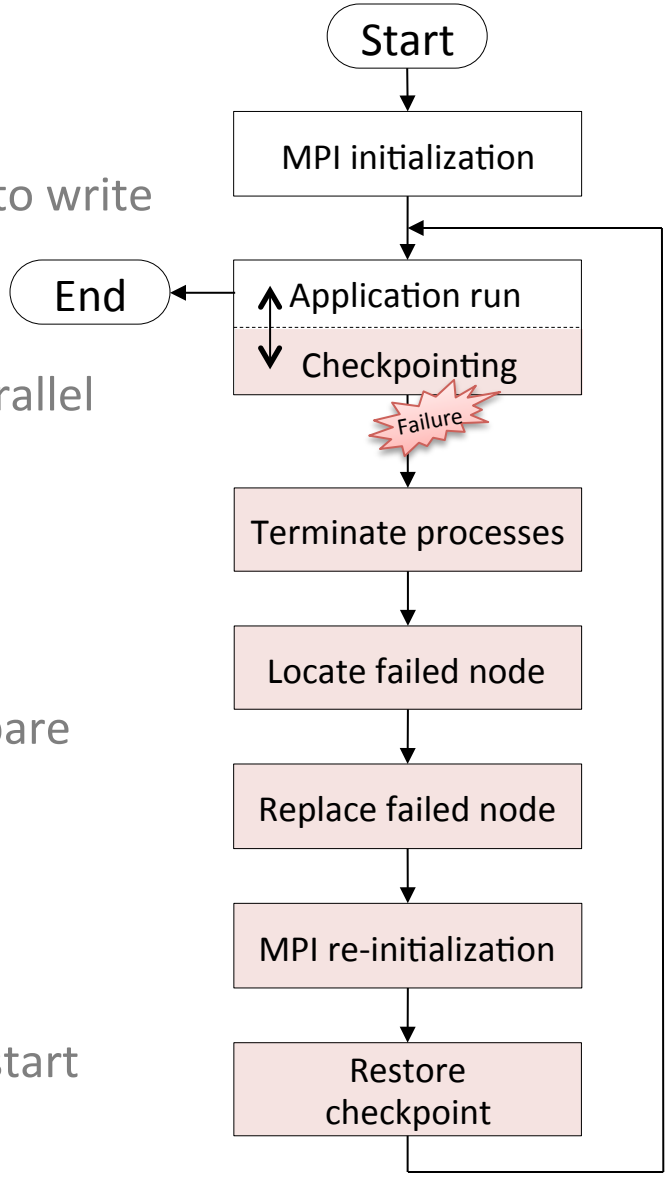
TSUBAME MTBF

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

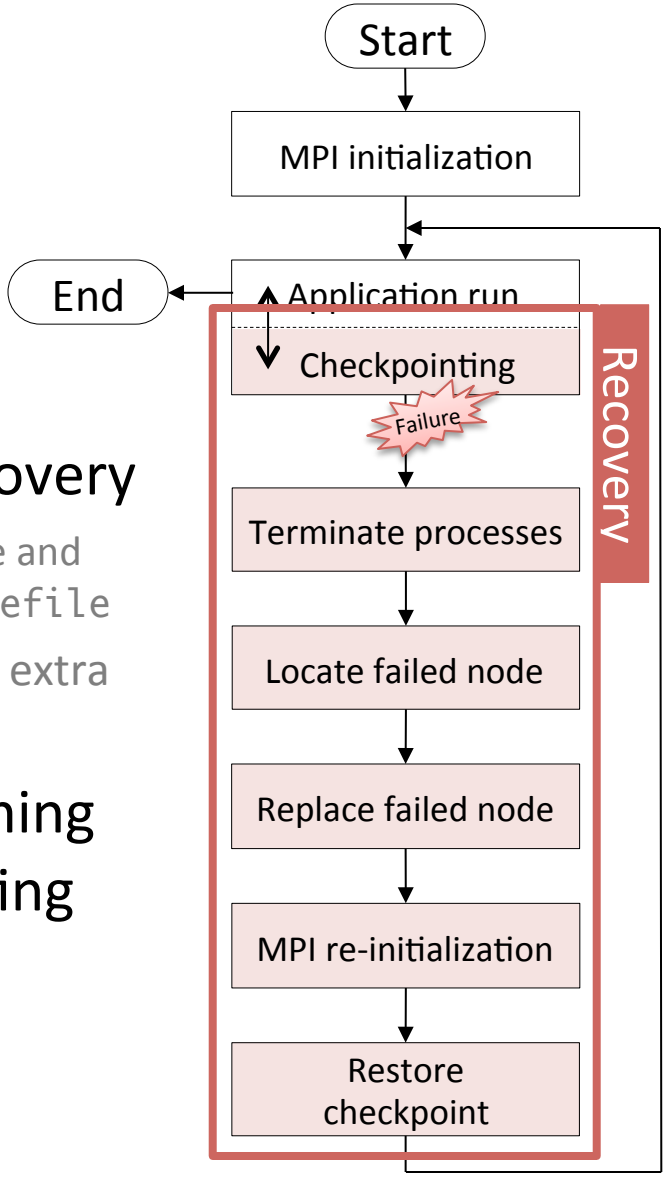
Conventional fault tolerance in MPI apps

- 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

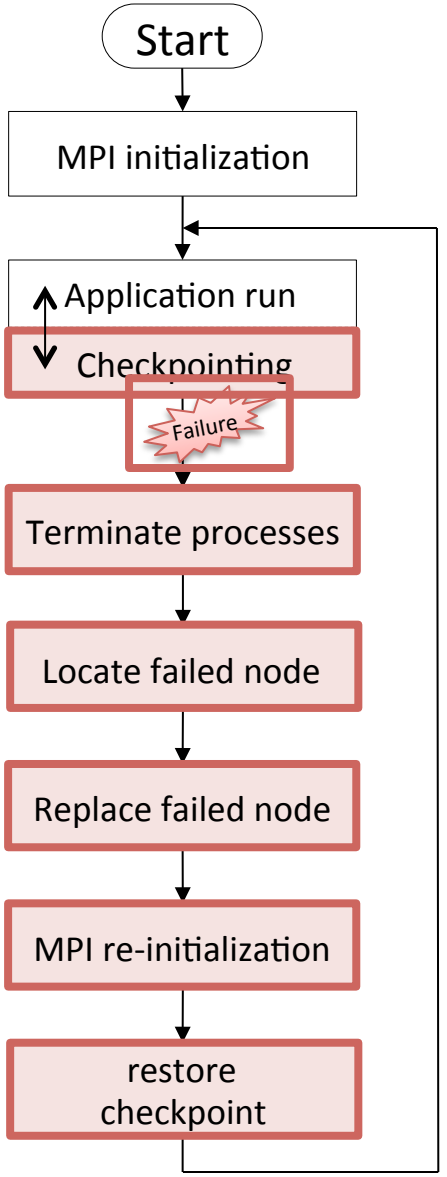
- Contributions:
  - We developed Fault Tolerant Messaging Interface (FMI) enabling fast and transparent recovery
  - Experimental results show FMI incurs only a **28% overhead** with a very high **MTBF of 1 minute**

Outline

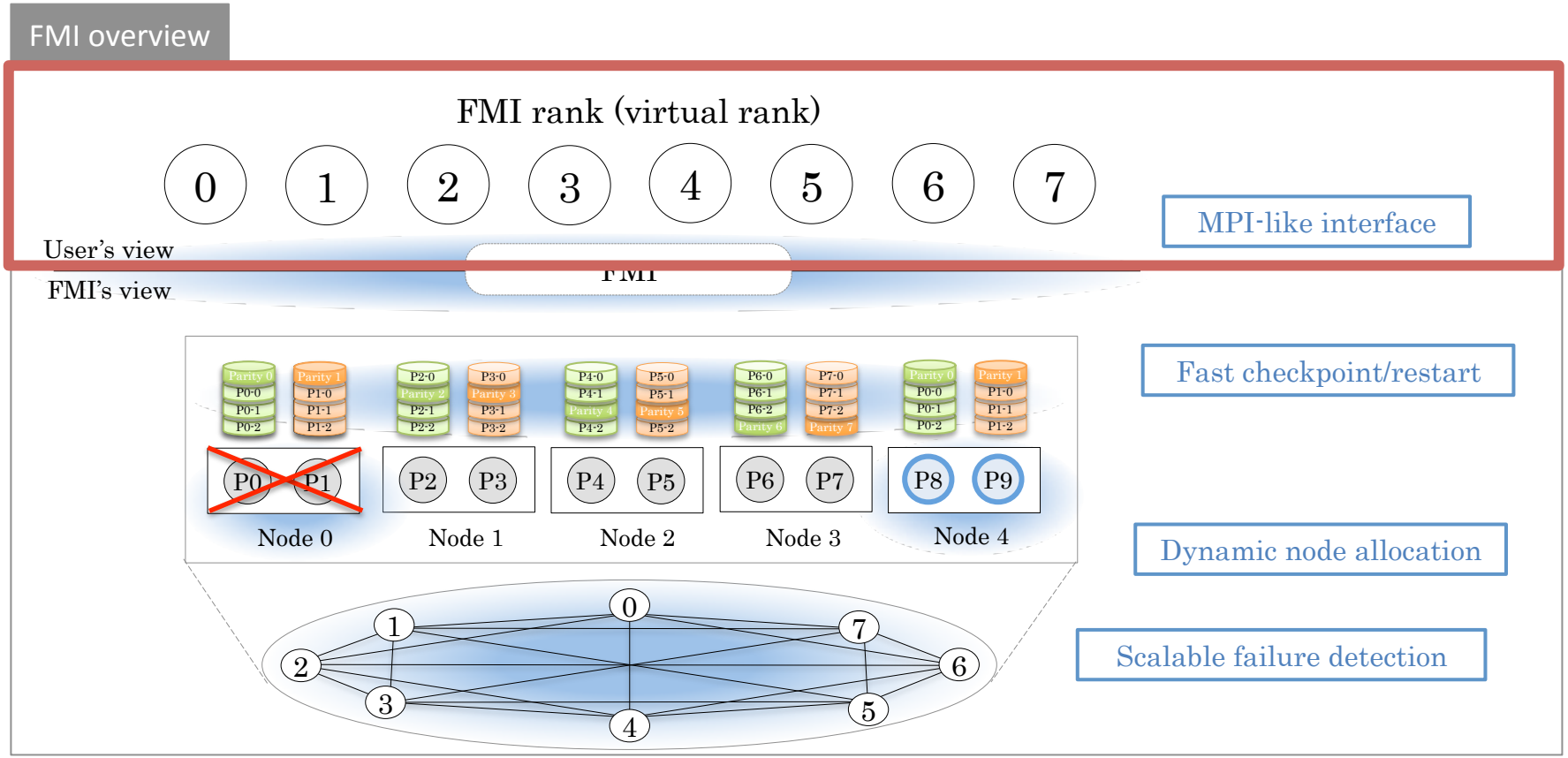
- Introduction
- Challenges for fast and transparent recovery
- FMI: Fault Tolerant Messaging Interface
  - User perspective
  - Internal implementation
- Evaluation
- Conclusion

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

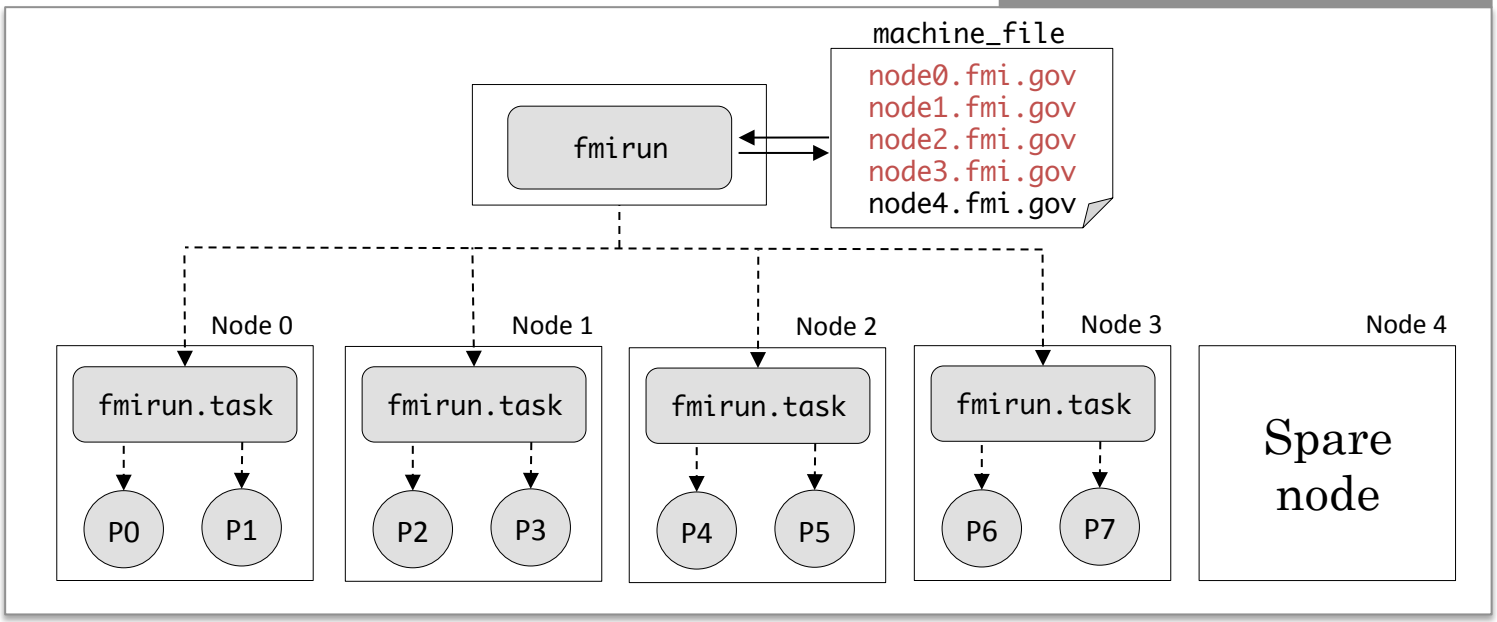
How FMI applications work ?

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(…)

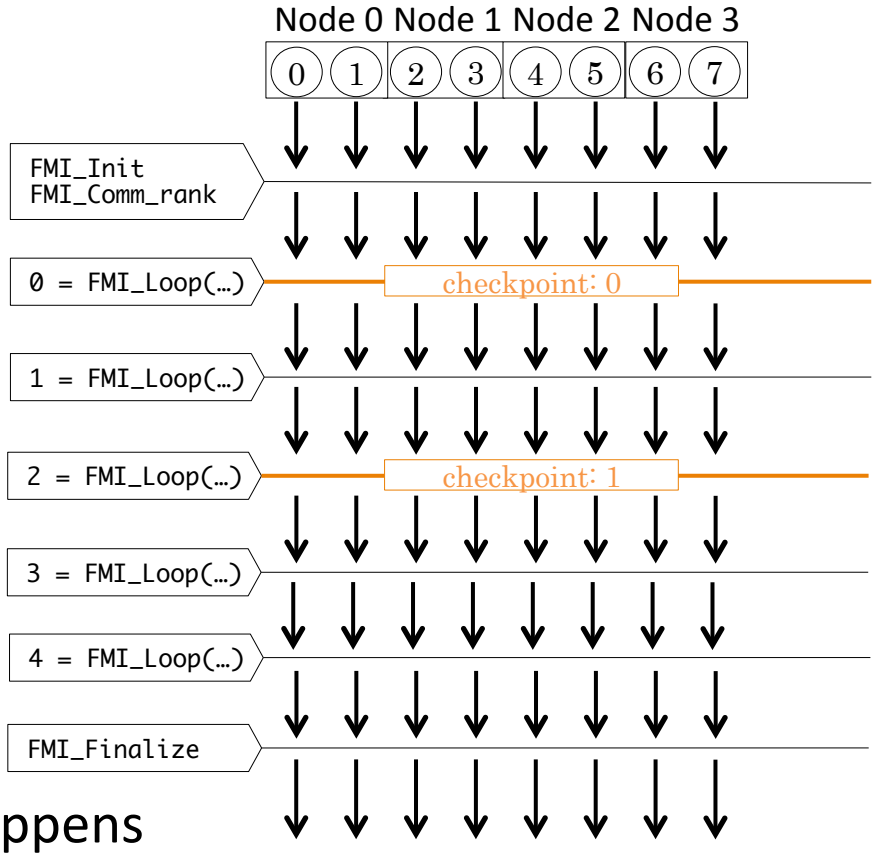
Launch FMI processes



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();
}
```

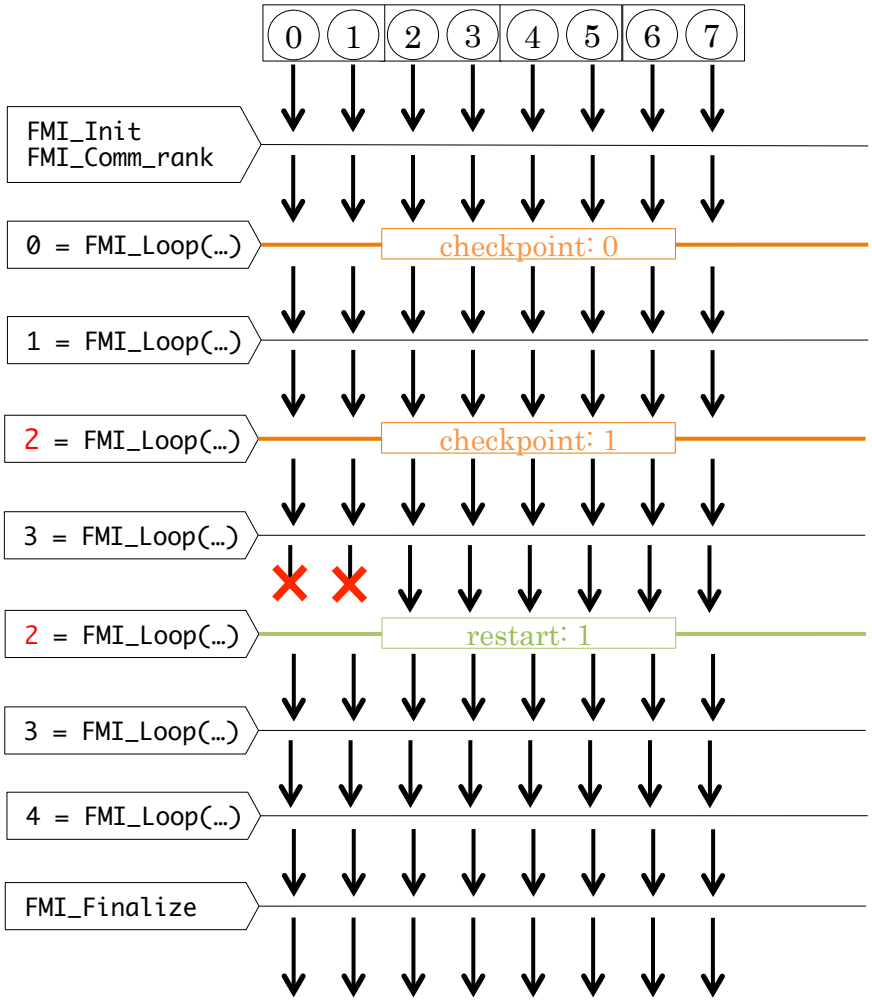


- User perspective when no failures happens
- Iterations: 4
- Checkpoint frequency: Every 2 iterations
- FMI_Loop returns incremented iteration id

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

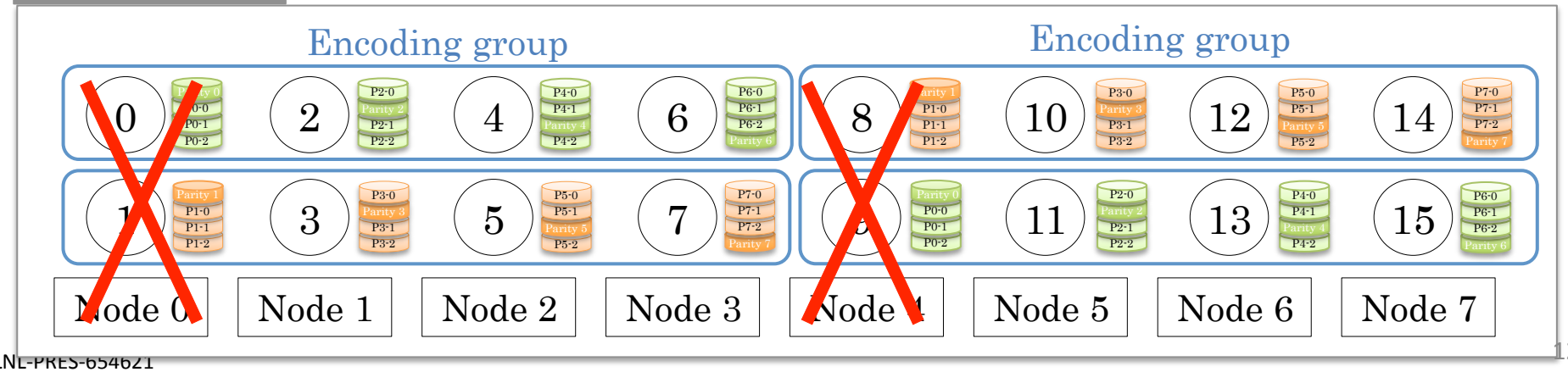
`len` : Length of arrays, `ckpt` and `sizes`

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

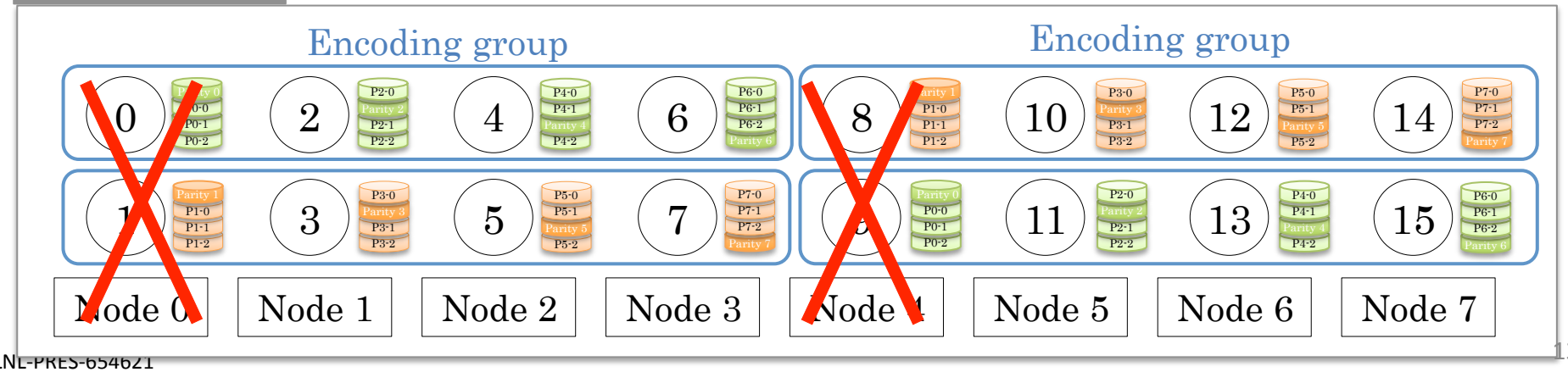
```
int FMI_Loop(void **ckpt, size_t *sizes, int len)
```

- `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`

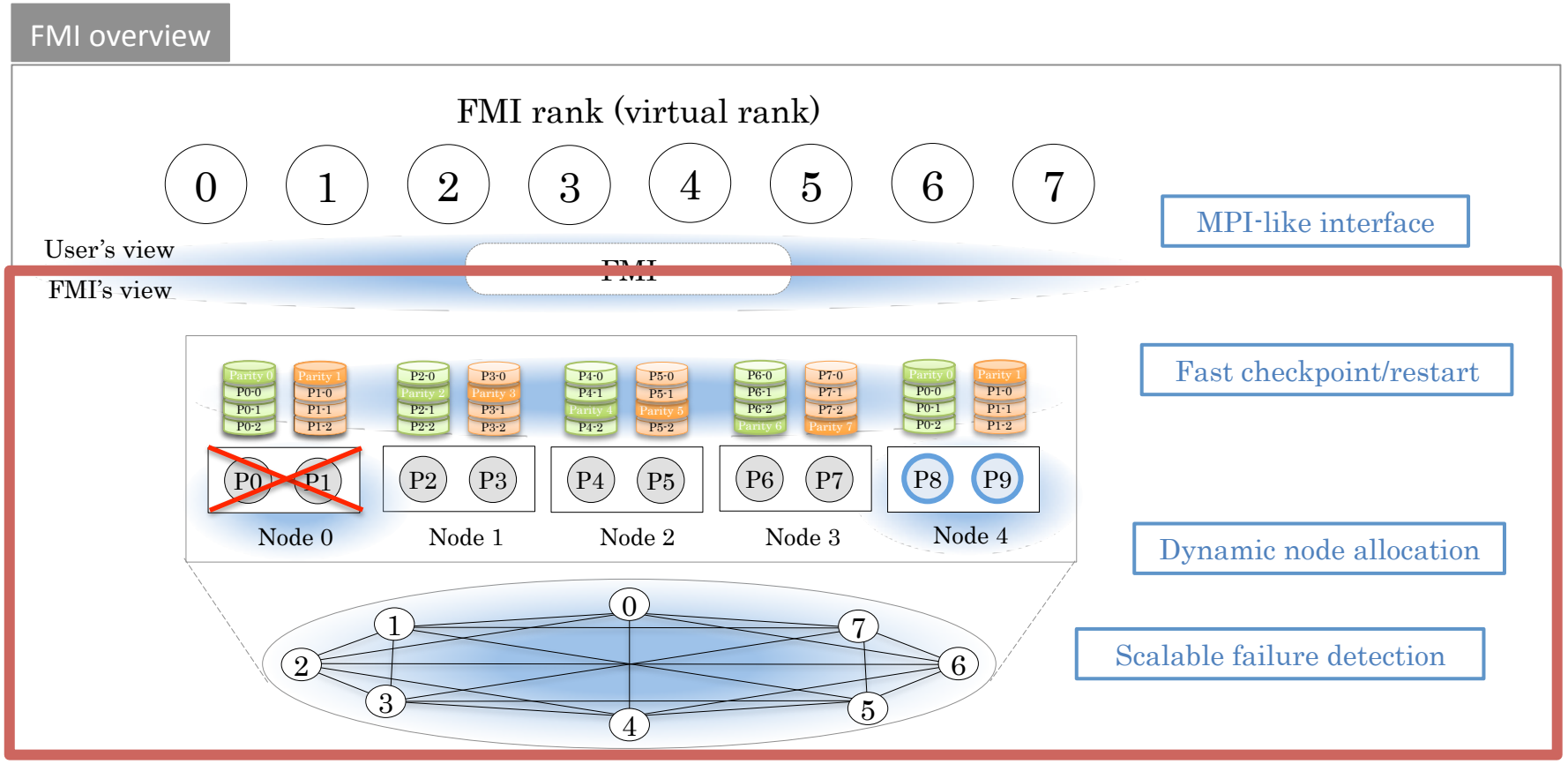
returns iteration id

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

FMI checkpointing

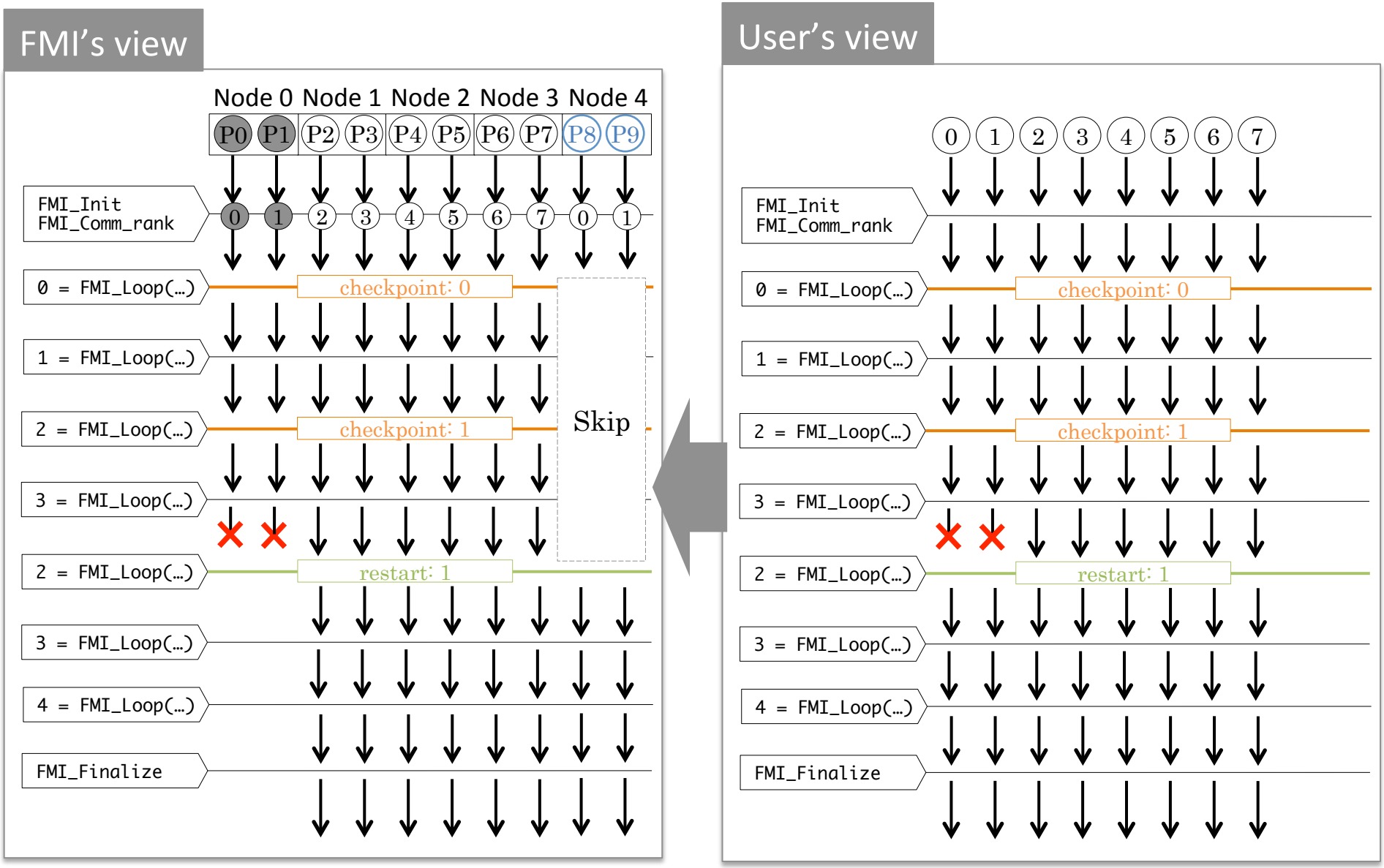


FMI: Fault Tolerant Messaging Interface

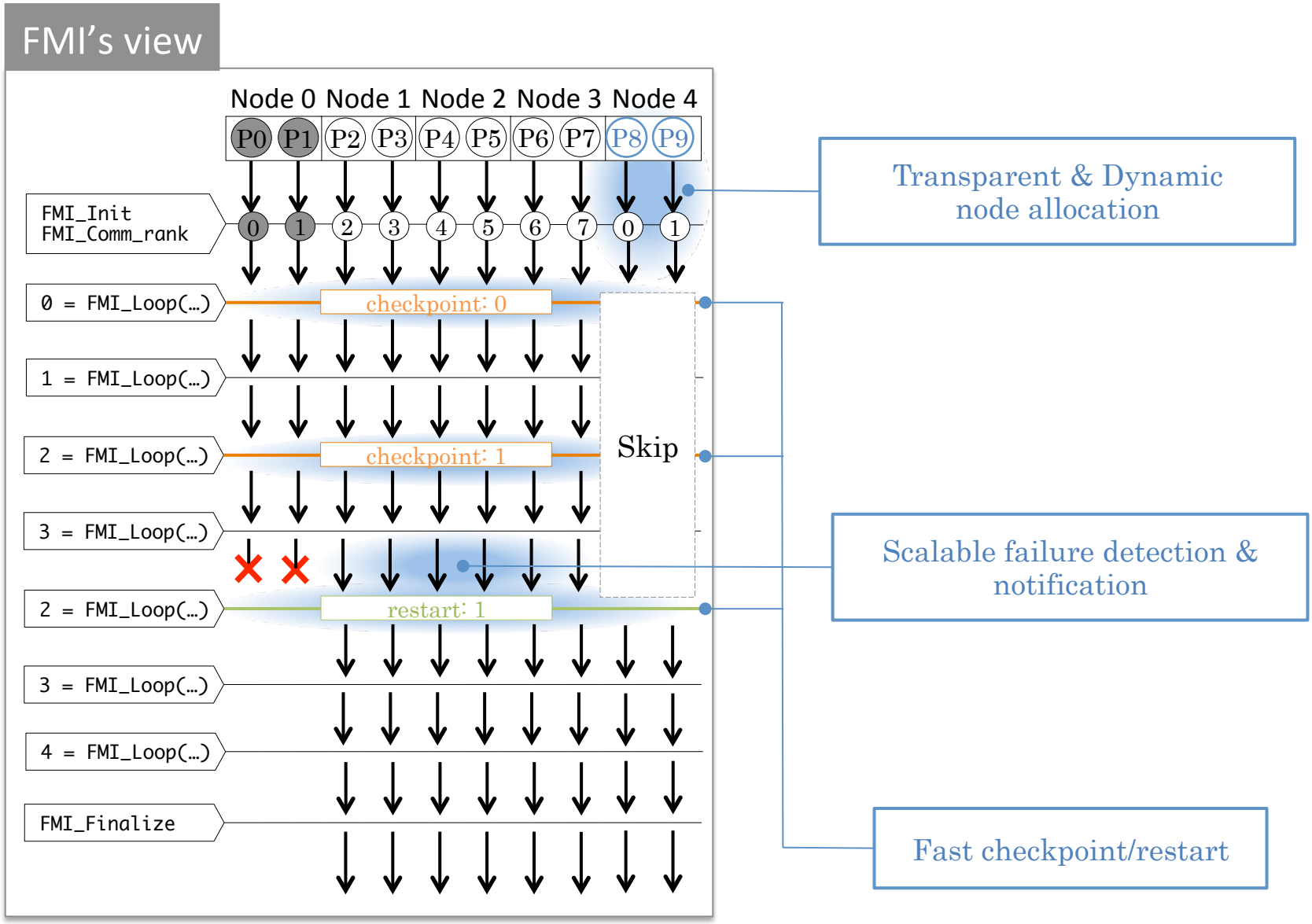


- FMI is an MPI-like survivable messaging interface
  - Scalable failure detection => Overlay network for failure detection
  - Dynamic node allocation => FMI ranks are virtualized
  - Fast checkpoint/restart => Diskless checkpoint/restart

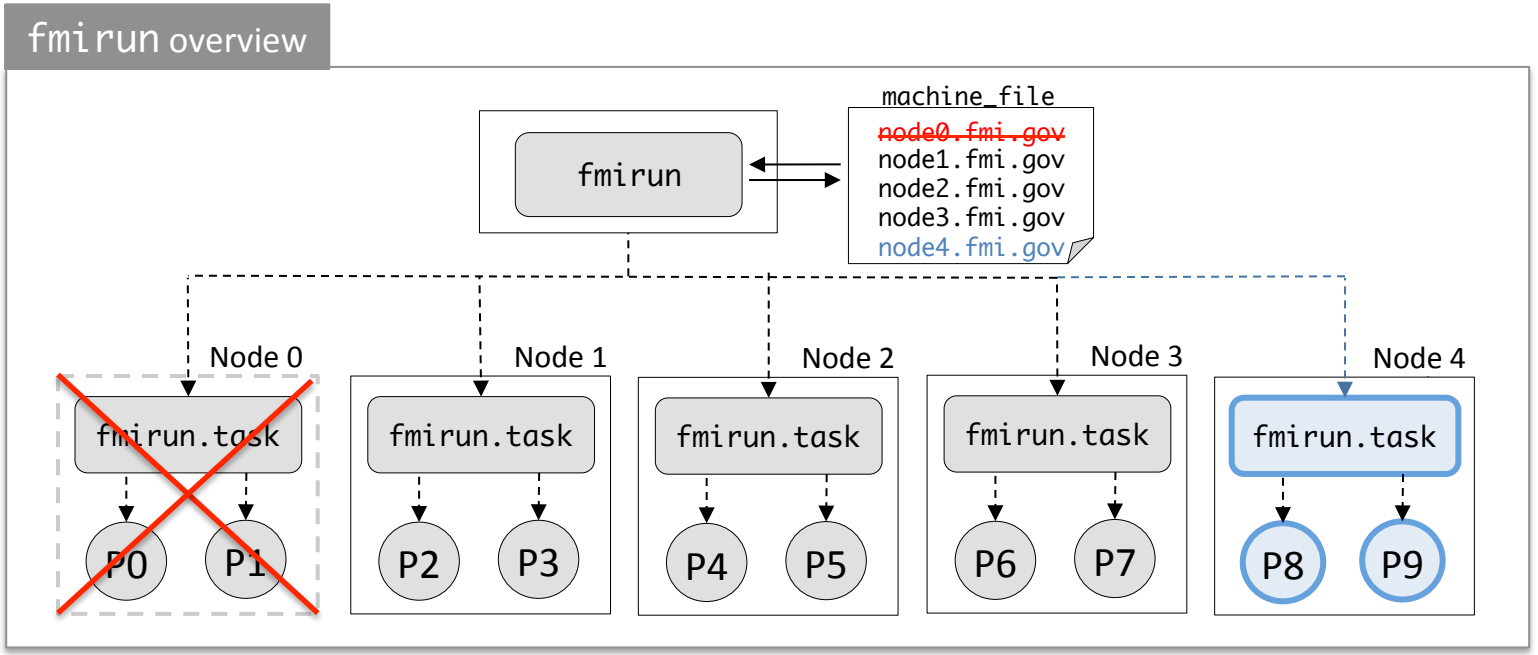
FMI's view & User's view



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
Skip
3 = FMI_Loop(…)
2 = FMI_Loop(…)
restart: 1
3 = FMI_Loop(…)
4 = FMI_Loop(…)
FMI_Finalize
Transparent & Dynamic node allocation
Scalable failure detection & notification
Fast checkpoint/restart

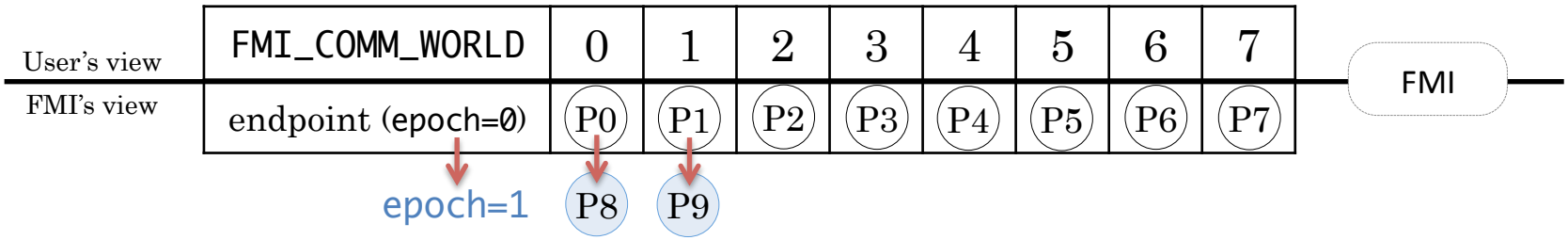
Transparent and dynamic node allocation

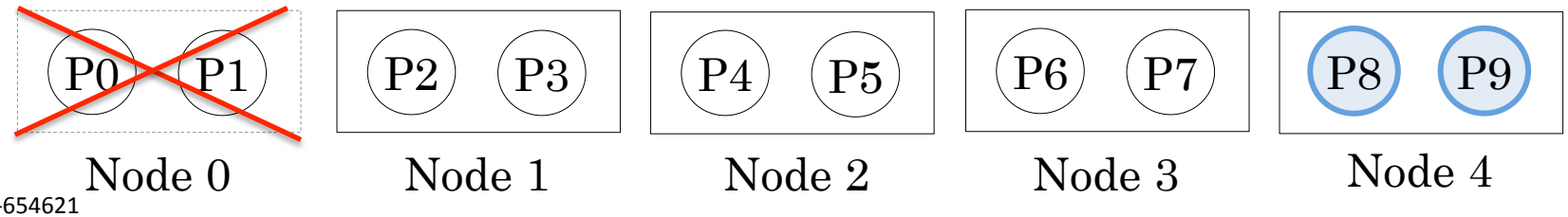


- If fmirun.task receives an unsuccessful exit signal from a child process
  - `fmirun.task` kills any other running child processes in the node, and exits with `EXIT_FAILURE`
- When fmirun receives the EXIT_FAILURE from the fmirun.task,
  - `fmirun` attempts to find spare nodes to replace the failed 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

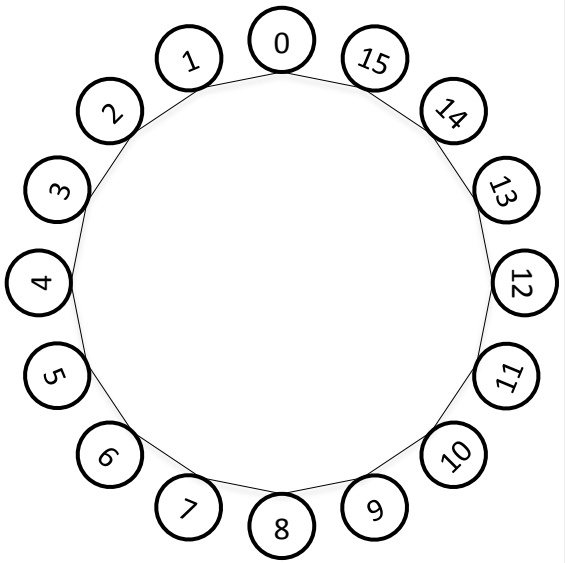




Scalable failure detection

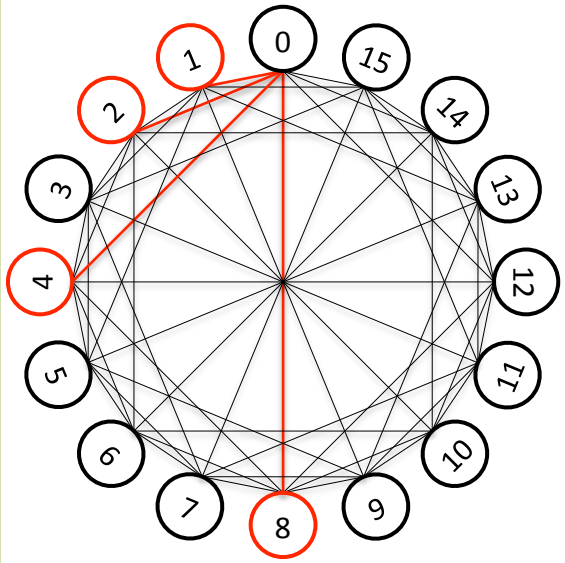
- FMI processes check if other processes are alive or not each other using overlay network
- Log-ring overlay network
  - Each FMI rank connects to 2^k-hop neighbors ($k = 0, 1 \ldots$)
  - e.g.) FMI rank 0 connects to FMI rank 1, 2, 4 and 8
- Log-ring overlay is scalable for both construction and detection

Ring overlay



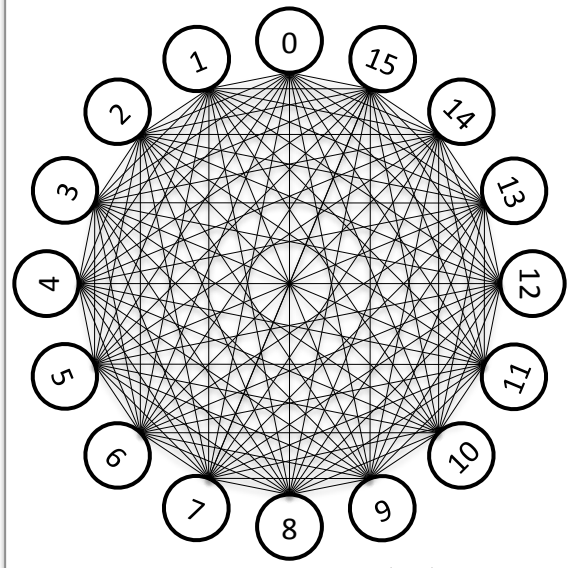
Construction: $O(1)$
Global detection: $O(N)$

Log-ring overlay



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

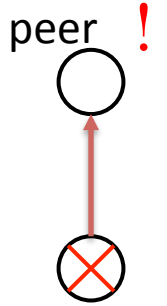
Complete overlay



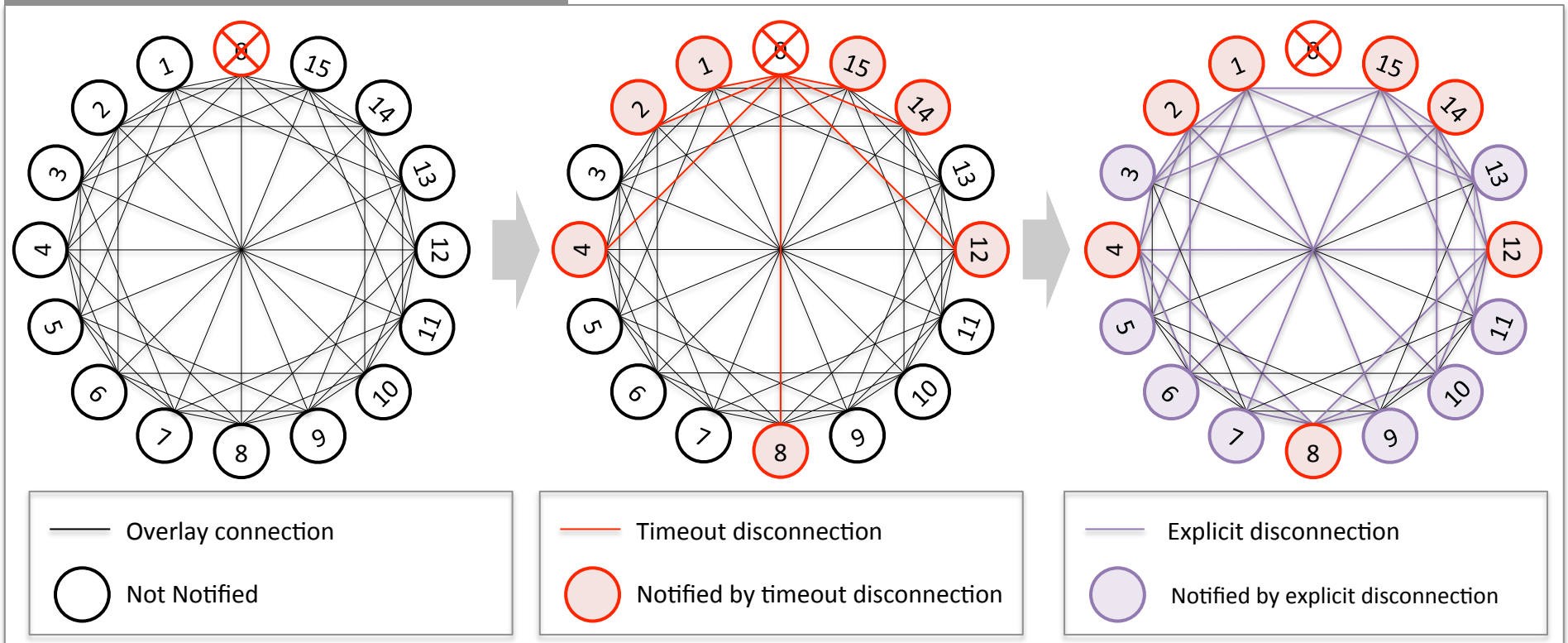
Construction: $O(N)$
Global detection: $O(1)$

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

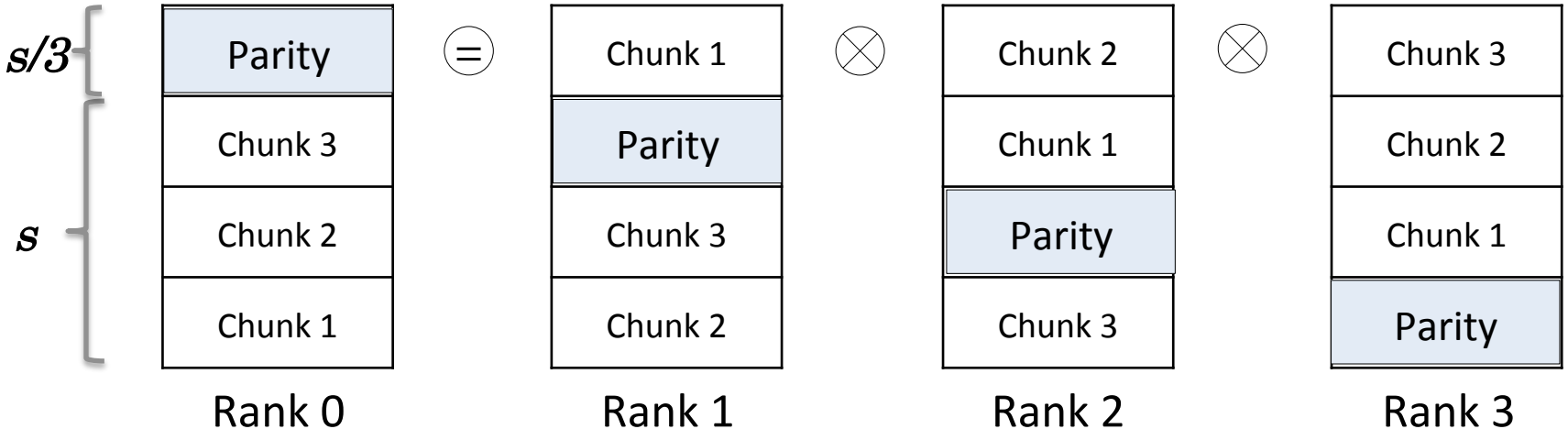


Example of global failure notification



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



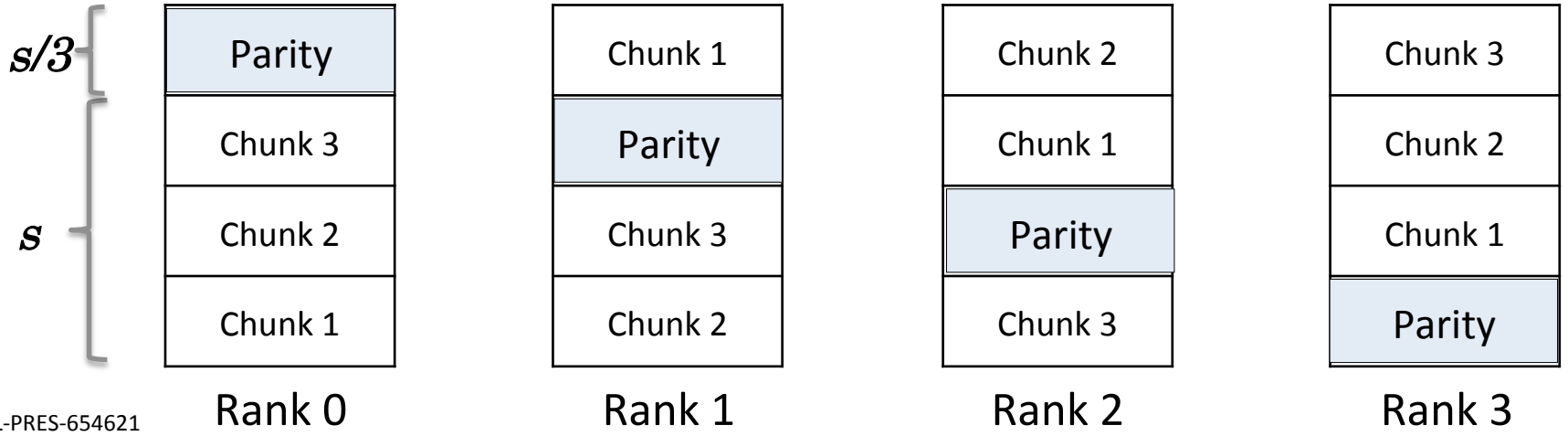
Source: A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10).

In-memory XOR checkpoint/restart model

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

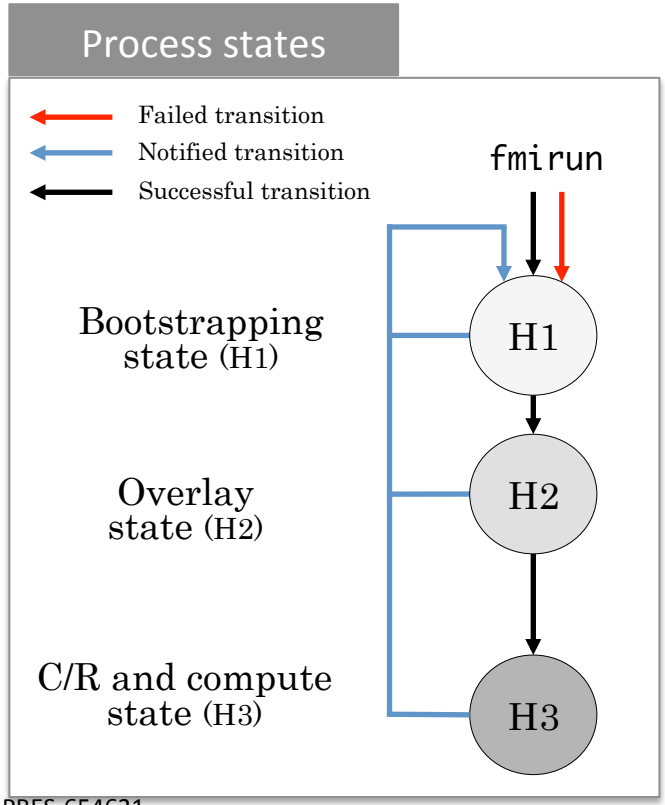
s : ckpt size, ***n*** : group size, ***mem_bw*** : memory bandwidth, ***net_bw*** : network bandwidth

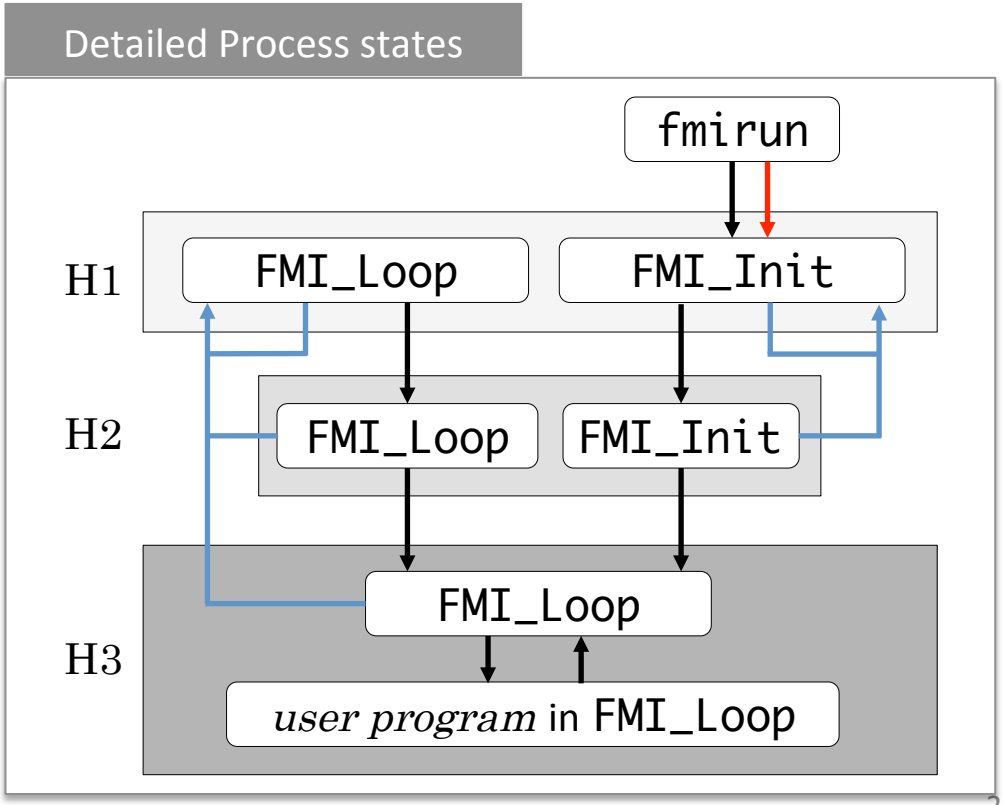
	memcpy	parity transfer	encoding	gathering
Checkpoint	$\frac{s}{mem\_bw}$	$\frac{s + s/(n-1)}{net\_bw}$	$\frac{s}{mem\_bw}$	
Restart	$\frac{s}{mem\_bw}$	$\frac{s + s/(n-1)}{net\_bw}$	$\frac{s}{mem\_bw}$	$\frac{s}{net\_bw}$



Process state manage

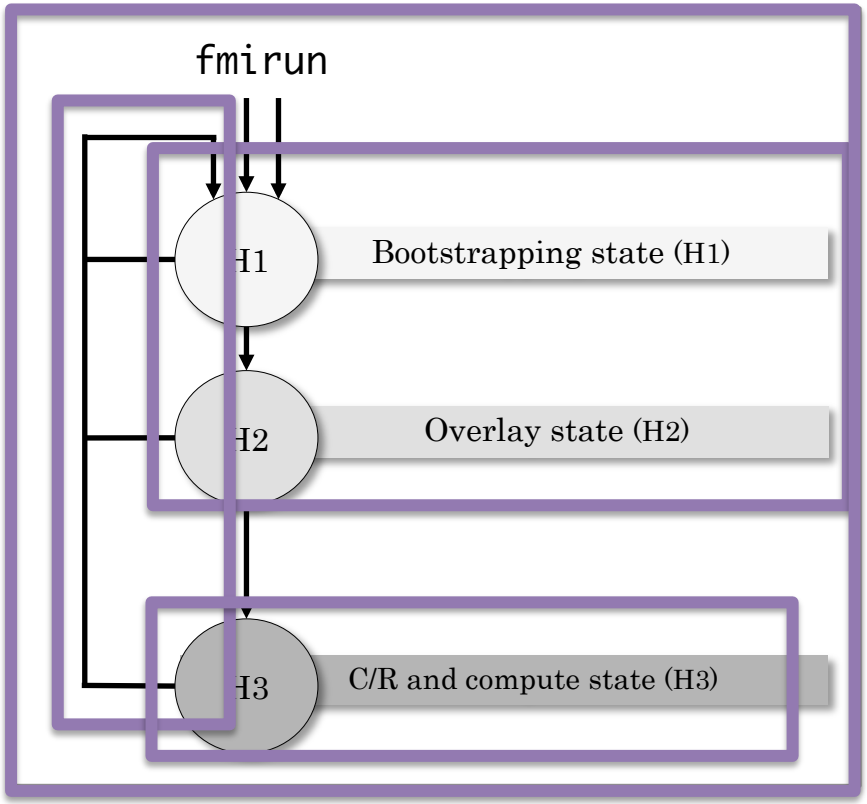
- FMI manages three states to make sure all processes to synchronously
  - H1: Bootstrap for endpoint, process mapping update, and epoch
  - H2: Construct overlay for scalable failure detection
  - H3: Do computation and checkpoint
- Whenever failures happens, all processes transitions to H1 to restart





Evaluations

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



Evaluations

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

Experimental environment

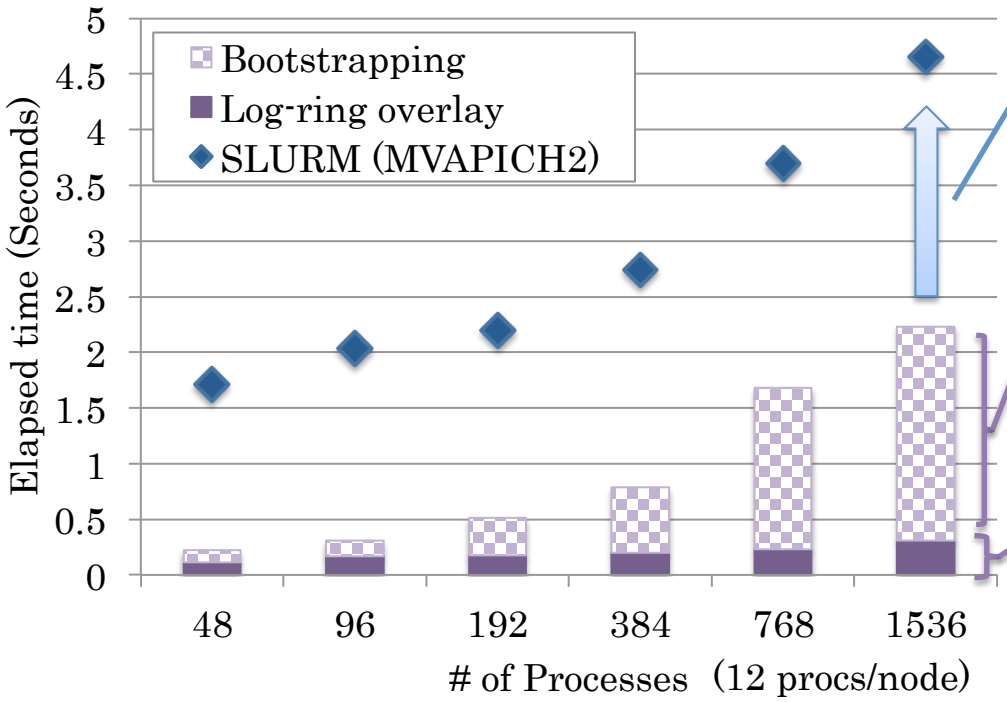
- Sierra cluster @LLNL

TABLE 4.1: Sierra Cluster Specification

Nodes	1,856 compute nodes (1,944 nodes in total)
CPU	2.8 GHz Intel Xeon EP X5660 × 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



Future FMI may reach the same initialization time as MPI one

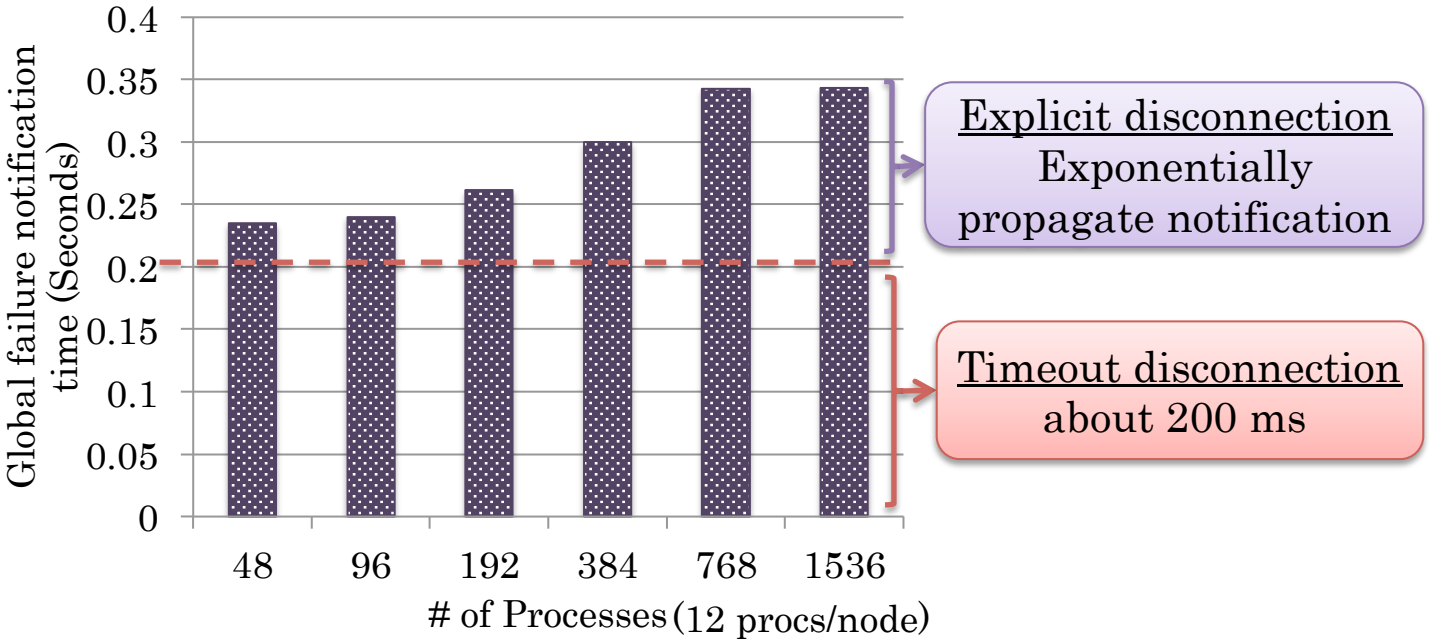
Bootstrapping time is also short
Currant FMI do only minimal initialization to start an application

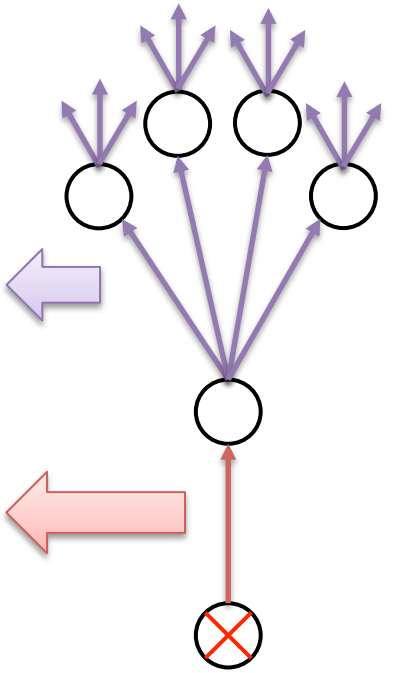
Log-ring construction time is small
the overlay construction time is *O(log(n))*

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

FMI failure detection time

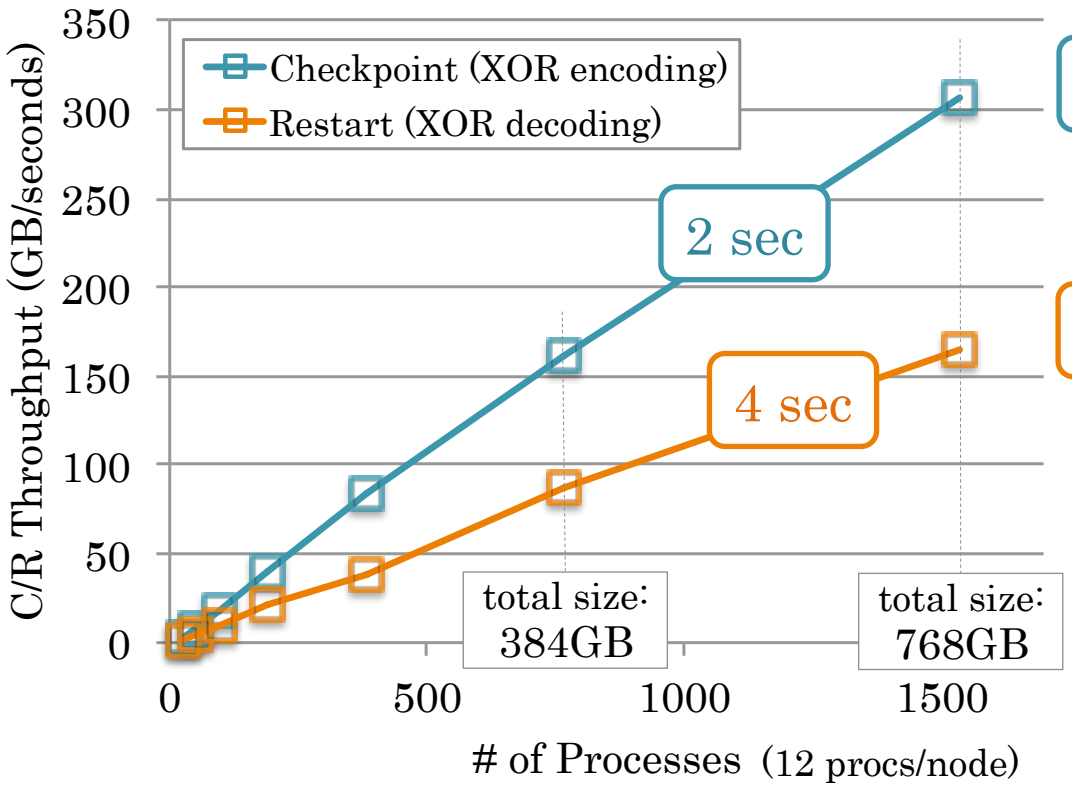
- We measured the time for all processes to be notified of a failure
  - Injected a failure by killing a process
- Once a process receive a disconnection event, the notification exponentially propagate
  - Time complexity: $O(log(N))$ to propagate





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



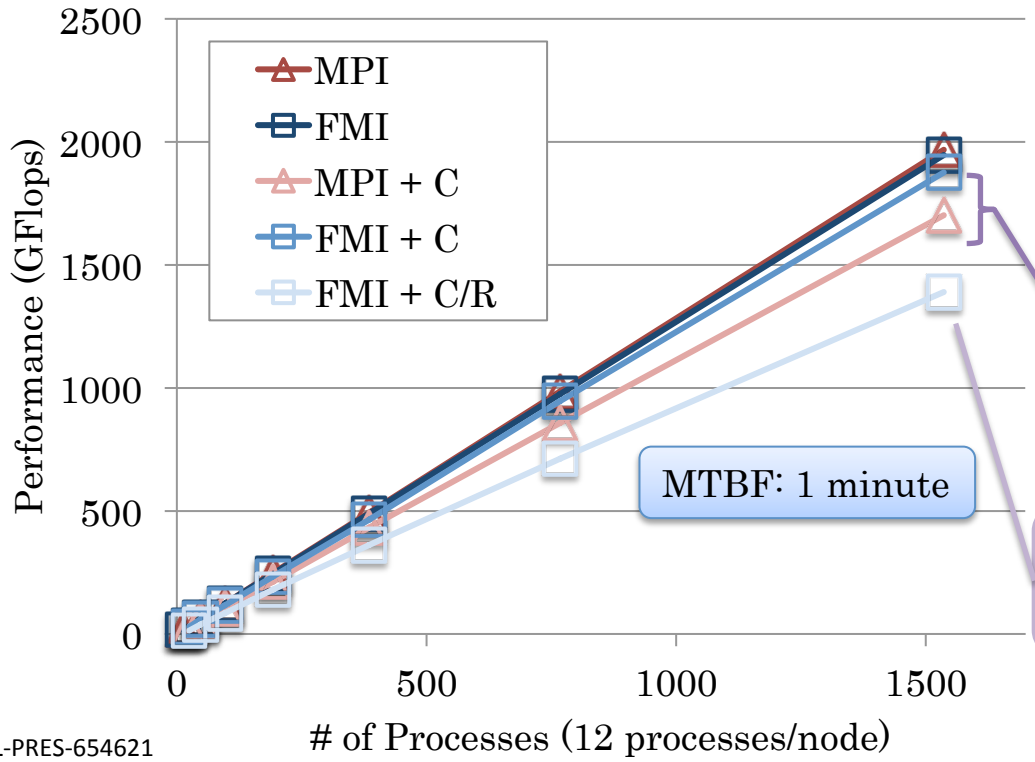
2.4 GB/sec per node

1.3 GB/sec per node

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

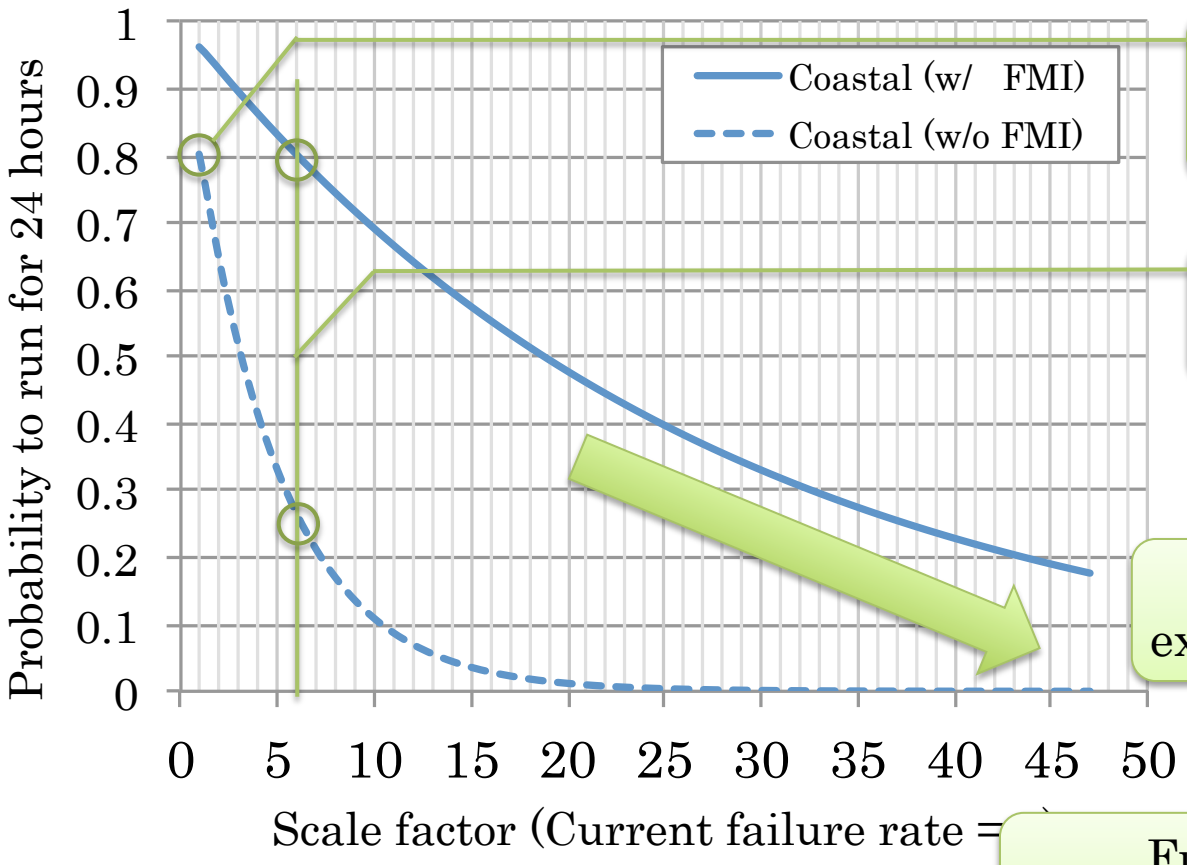
Failure analysis on Coastal cluster

	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



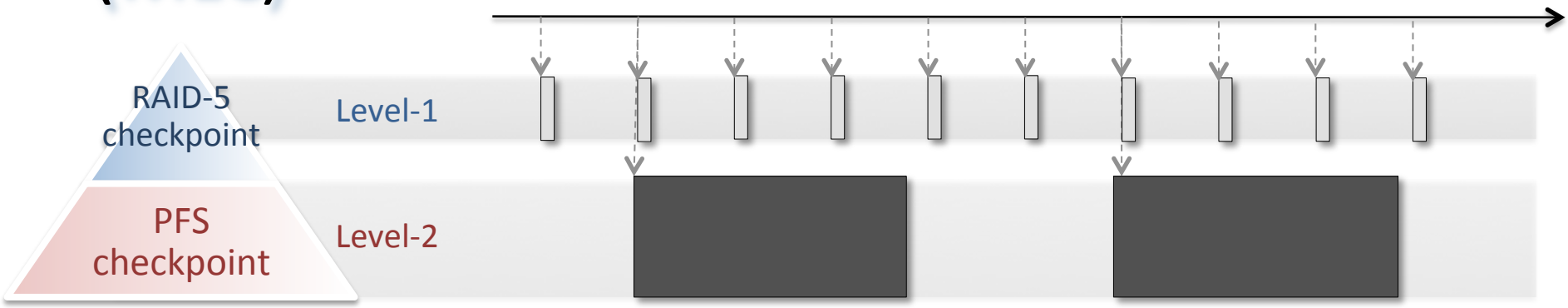
80% of probability to run for 24 hours on environment with current failure rate

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

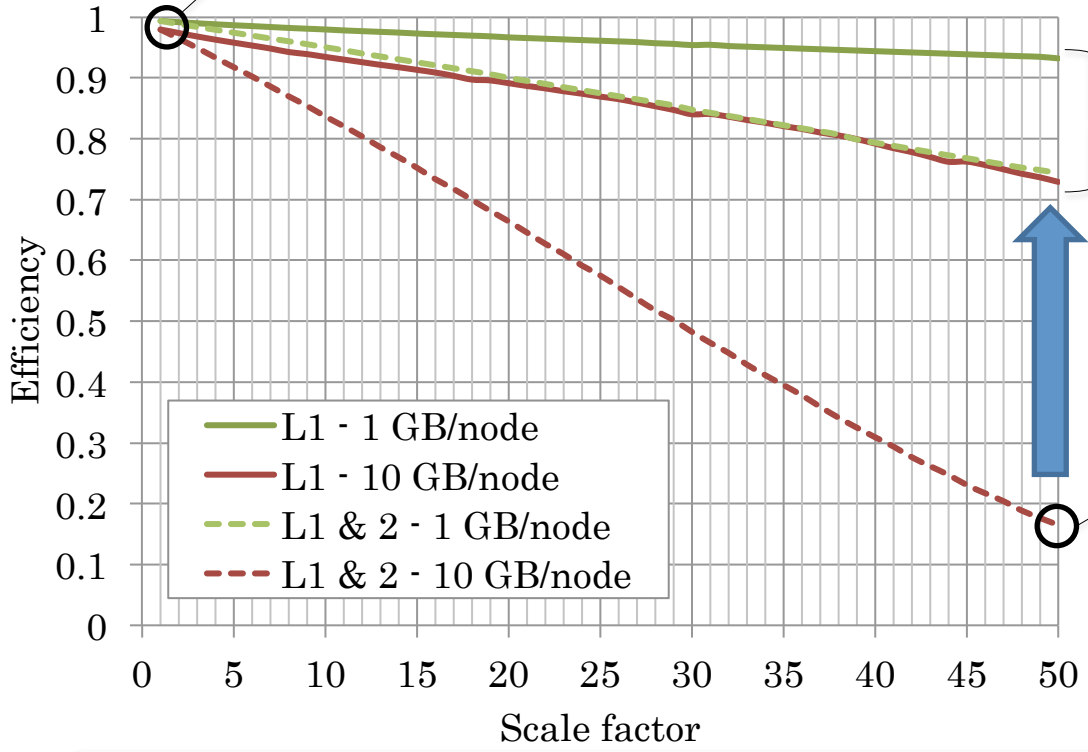
Failure analysis on Coastal cluster

	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

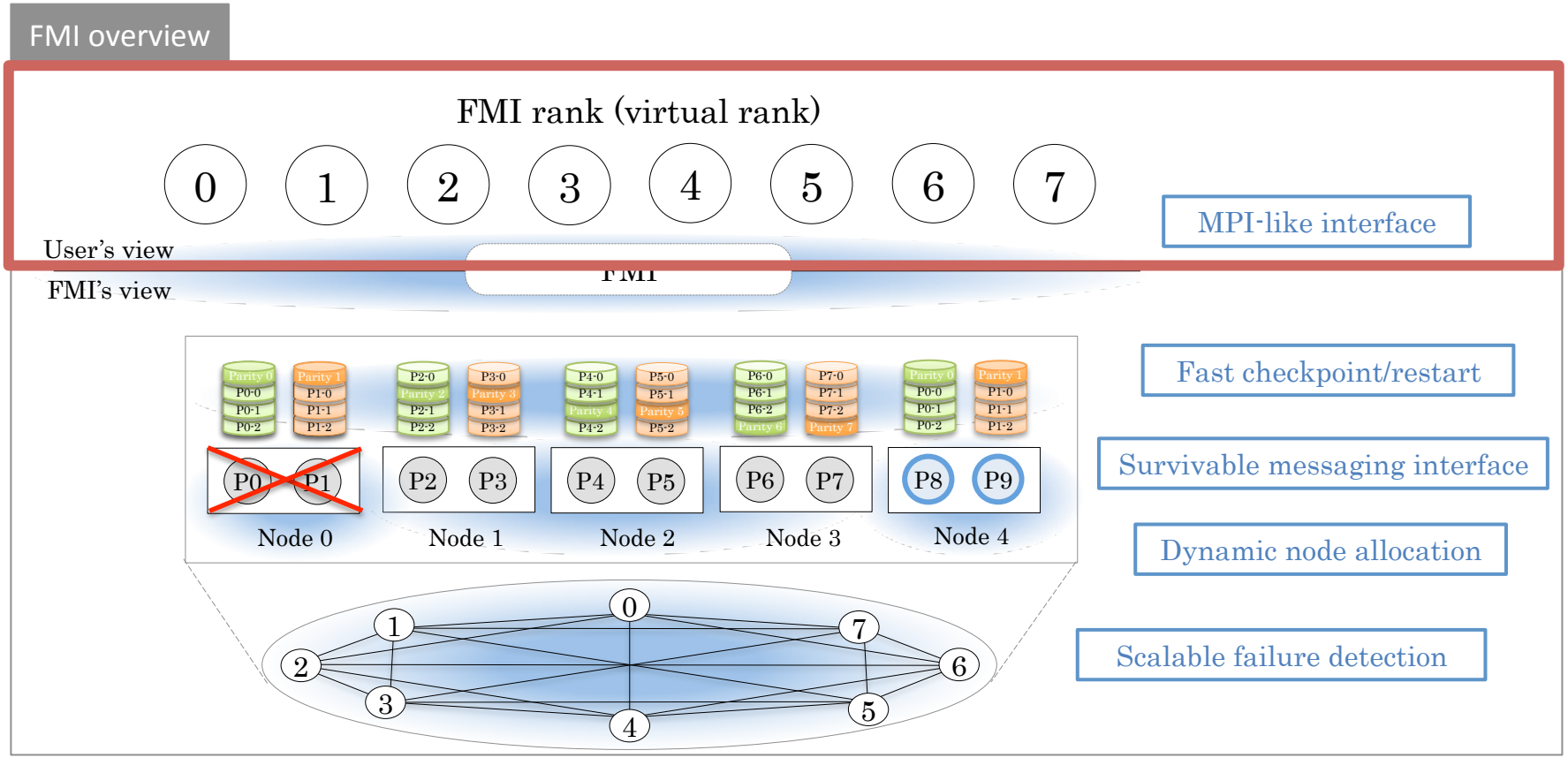
Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R de. Supinski,
Naoya Maruyama, Satoshi Matsuoka

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.

BACKUP

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

Major failures are compute node failures

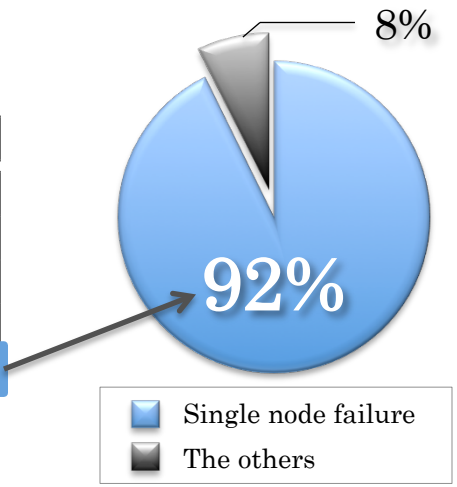
- Most of failures comes from one node, or can recover from XOR checkpoint
  - e.g. 1) TSUBAME2.0: 92% failures
  - e.g. 2) LLNL clusters: 85% failures
- The in-memory RAID-5 gives applications good enough resiliency

TSUBAME2.0 failure analysis

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

Period: 1.5 years (Nov 1st, 2010 ~ April 6th 2012)

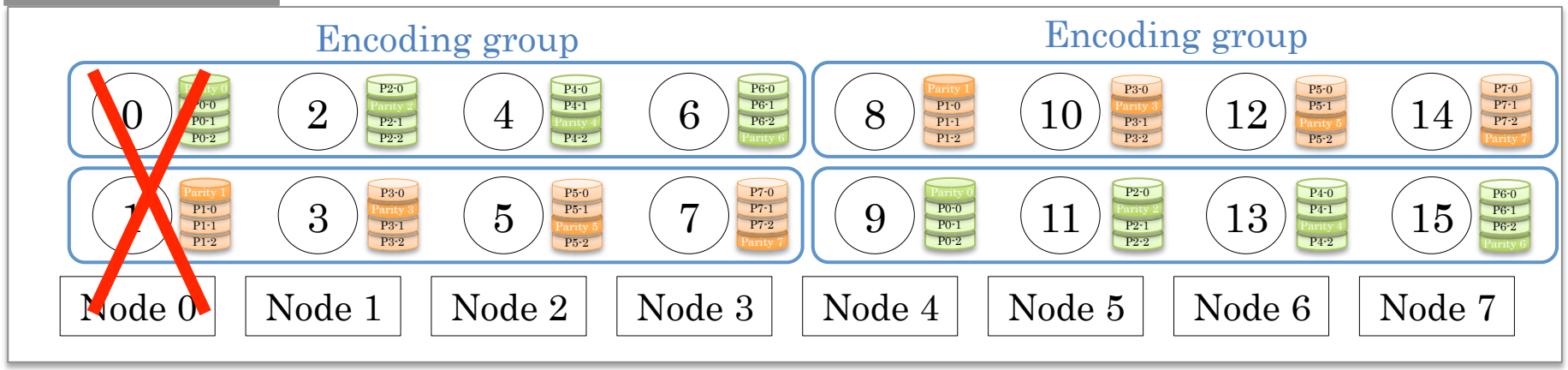


Source: K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka, "Design and Modeling of a Non-Blocking Checkpointing System," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ser. SC '12. Salt Lake City, Utah

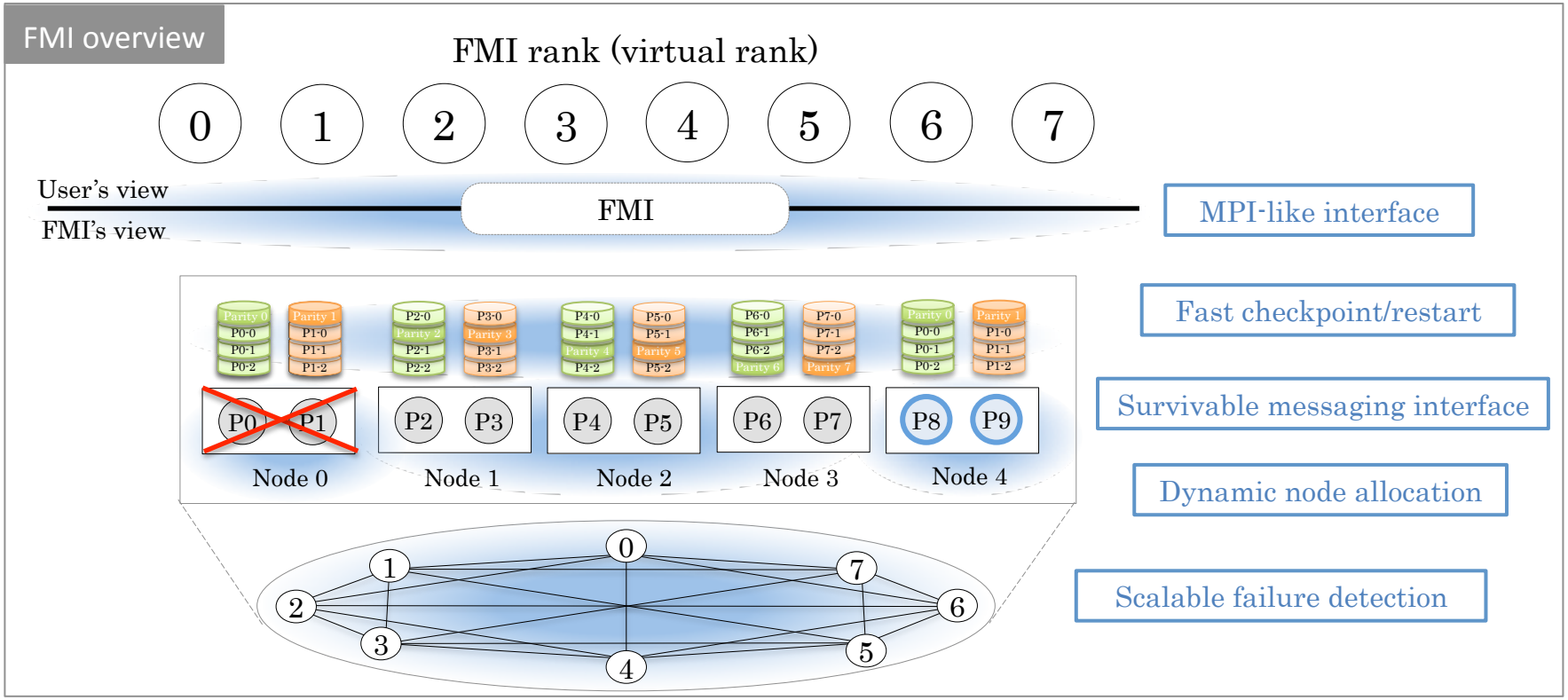
Fast checkpoint/restart

- In-memory checkpoint/restart using `memcpy`
  - FMI directly write checkpoints in in-memory space using `memcpy`
  - Then, create encoding groups across nodes, and construct in-memory RAID-5, which computes XOR parity data and distributes them across nodes
- Like RAID-5, FMI is tolerant to a single node failure with in a encoding group
- Unlike with MPI, FMI does not terminate non-failed processes on a failure, and in-memory checkpoint data is not flushed

FMI checkpointing



FMI: Fault Tolerant Messaging Interface

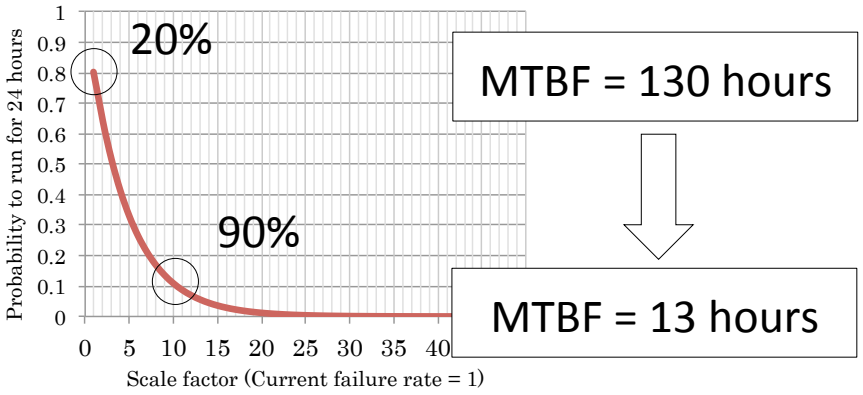


- FMI is MPI-like survivable messaging interface
- Scalable failure detection
  - When a failure occurs, all N processes can detect it or be notified by other processes on a failure with time complexity, *O(log(N))*

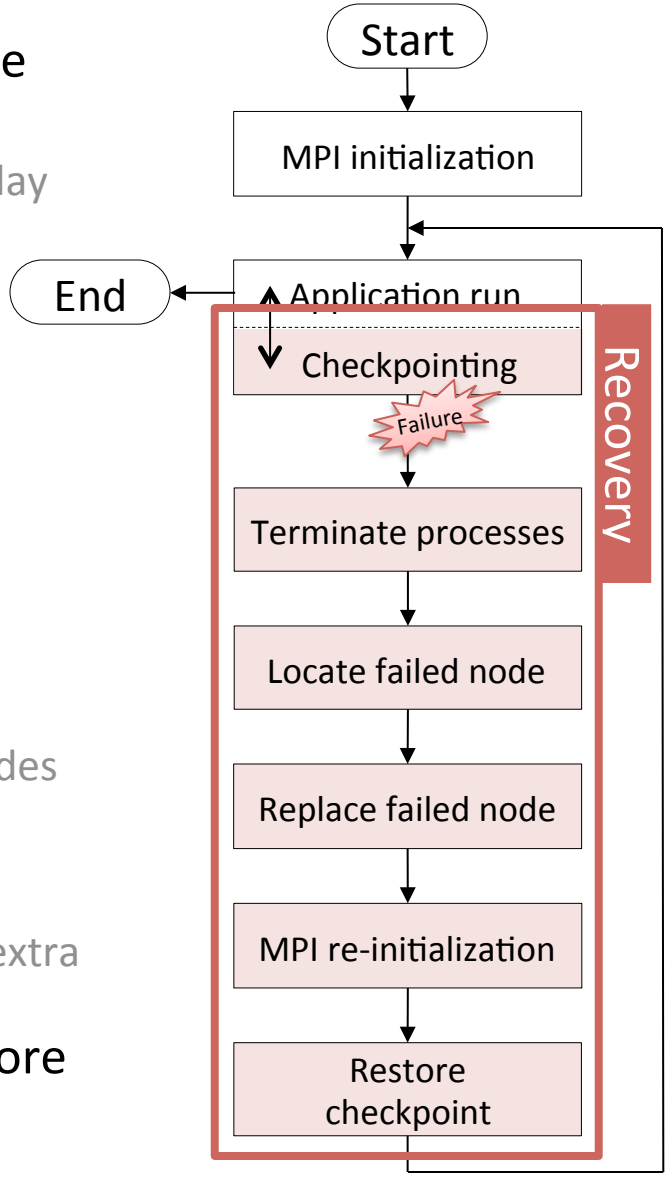
- Survivable messaging interface
  - On a failure, FMI does not terminate all FMI processes
  - FMI does not free any resources on non-failed nodes
- Dynamic node allocation
  - FMI transparently replace failed nodes with spare nodes
  - Applications can still use FMI rank (virtual rank)
- Fast checkpoint/restart
  - Scalable disk less checkpointing (XOR encoding)
  - Directly write checkpoints in a memory space

Requirement of fast and transparent recovery

- Failure rate will increase in future extreme scale systems
  - Failures are expected to occurs several times in a day



- Applications will use more time for recovery
  - If MPI does not return hostnames of all failed nodes on a failure, Users
    - Manually locate the failed nodes
    - Manually replace the failed 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



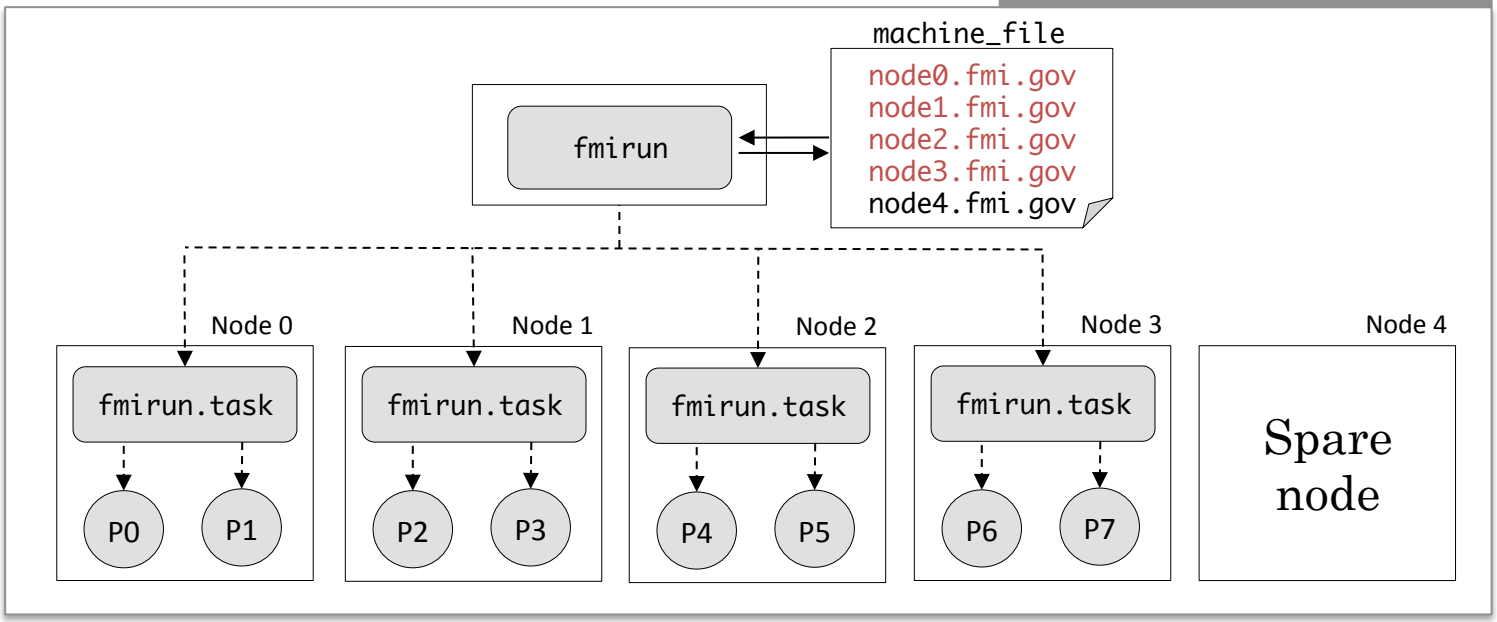
How FMI applications work ?

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();
}
```

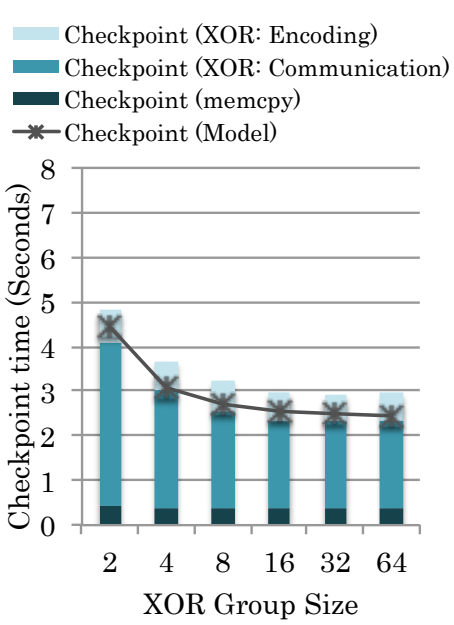
- FMI_Loop enables transparent recovery and roll-backs on a failure
  - Periodically write a checkpoint/
  - Restore the last checkpoint on a failure
- Processes are launched via `fmirun`
  - `fmirun` spawns `fmirun.task` on each node
  - `fmirun.task` calls fork/exec a user program
  - `fmirun` broadcasts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

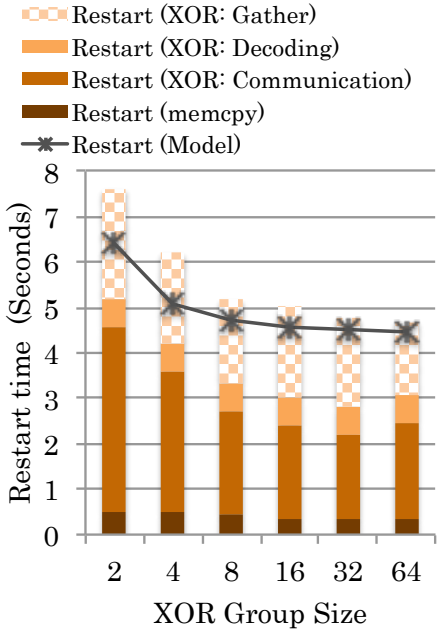
Launch FMI processes



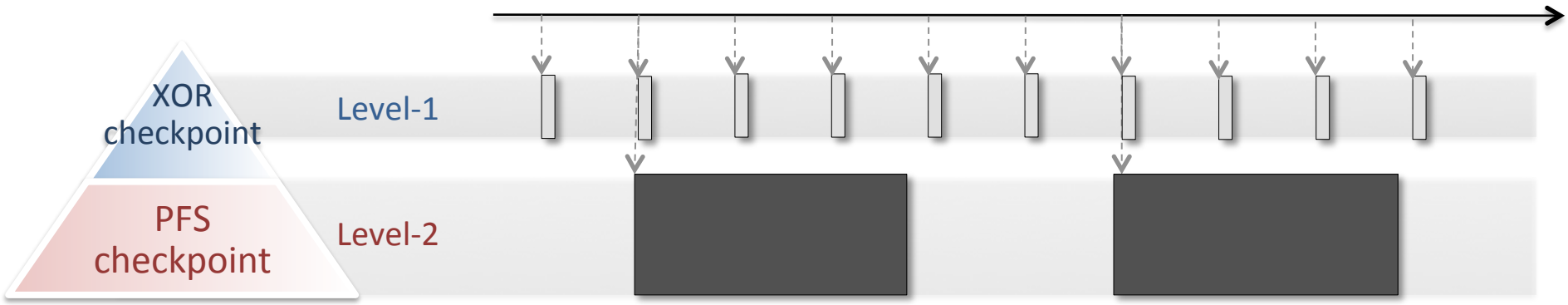
Checkpoint/Restart model

- Checkpoint size: 6GB per node
  - For the memory and network bandwidths in the model, we use the peak bandwidth of the Sierra cluster at LLNL
  - 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.

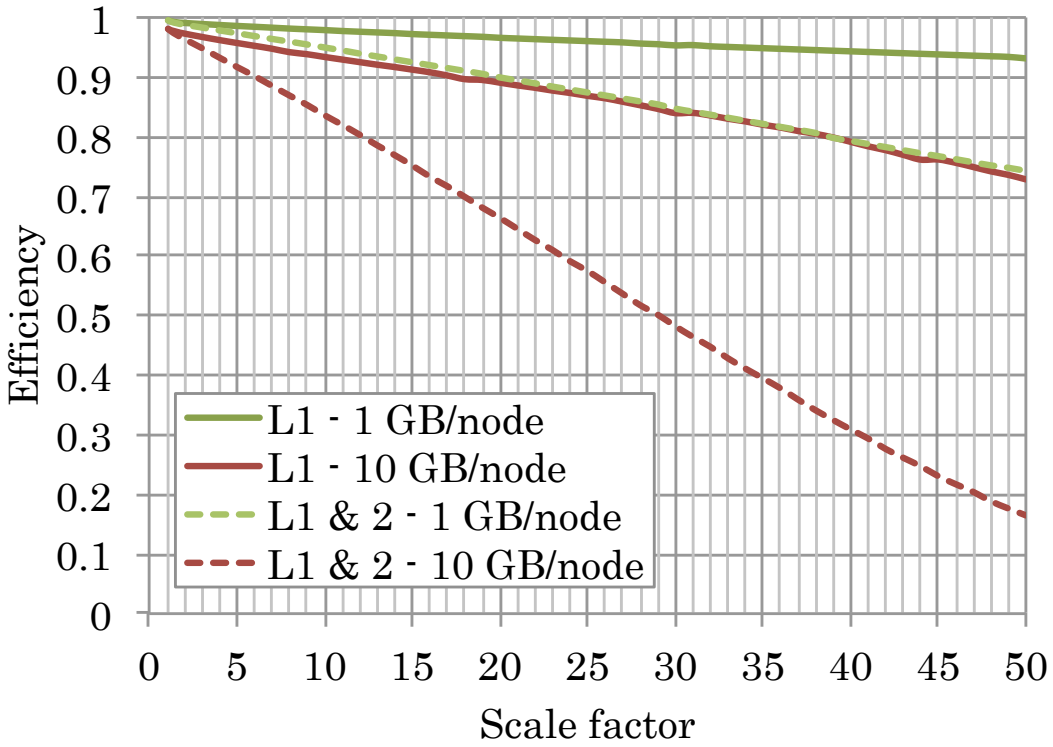




Asynchronous multi-level checkpointing

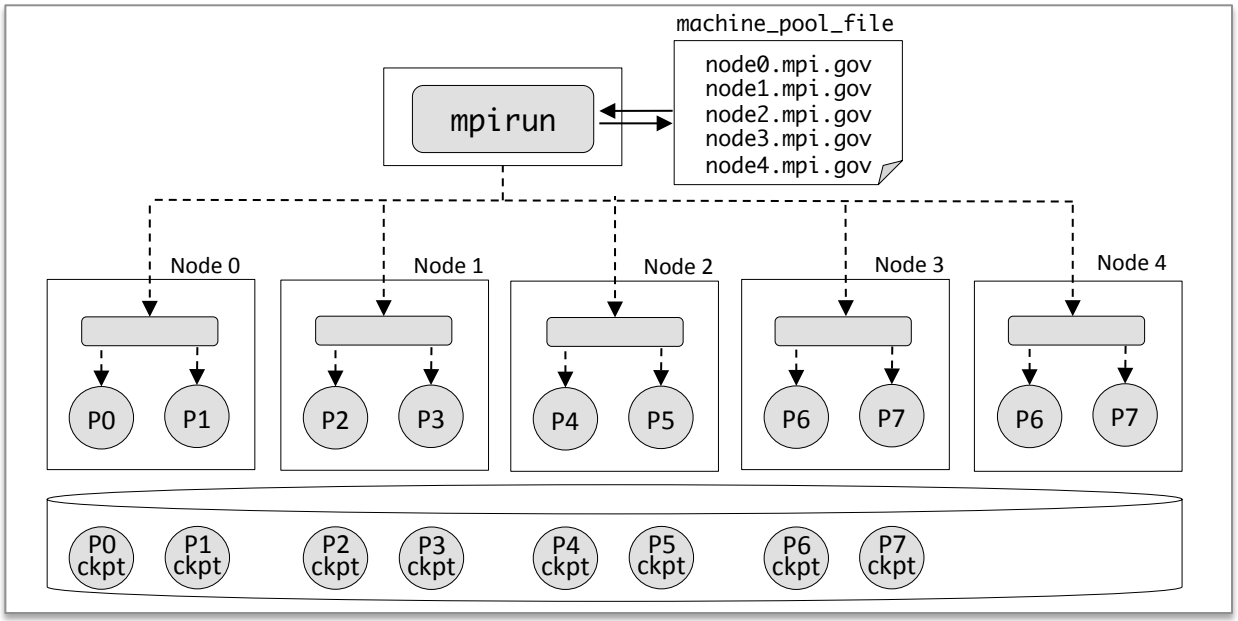


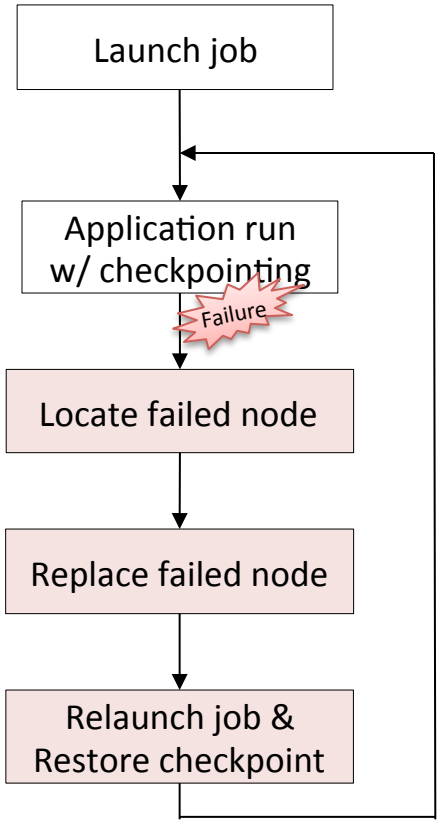
K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka, "Design and Modeling of a Non-Blocking Checkpointing System," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ser. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012



Conventional fault tolerance in MPI app.

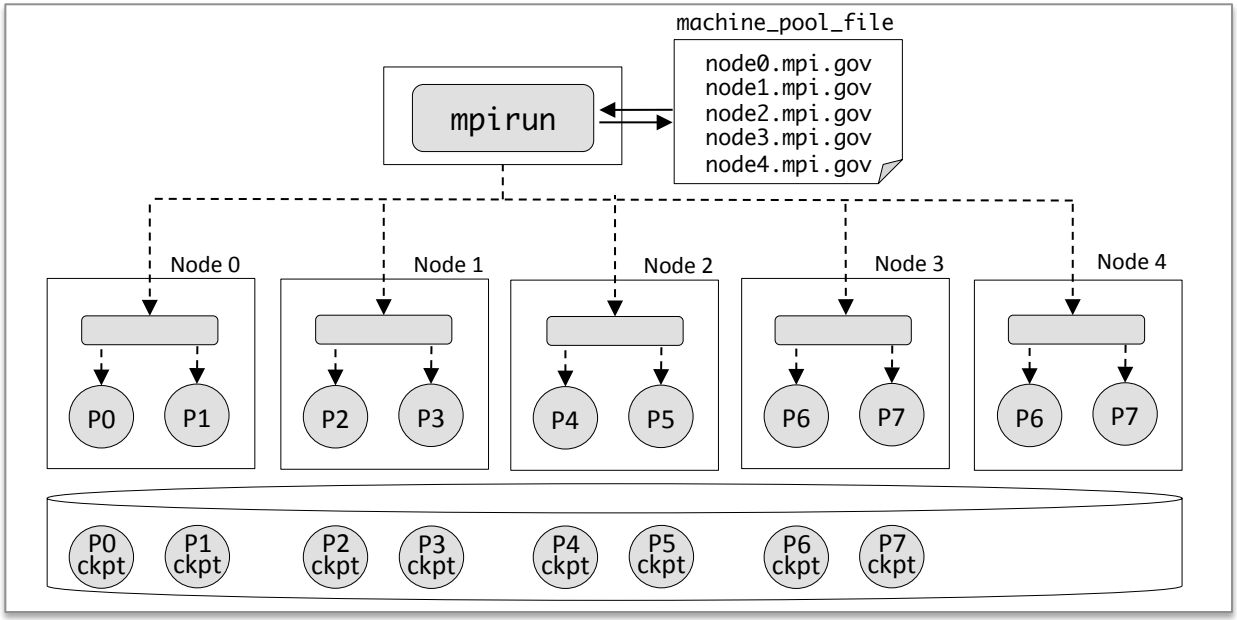
- Long running MPI applications usually write checkpoints
- When a failure occurs, the MPI application is terminated, and relaunched as a new job that restarts from the last checkpoint
- This approach is quite simple

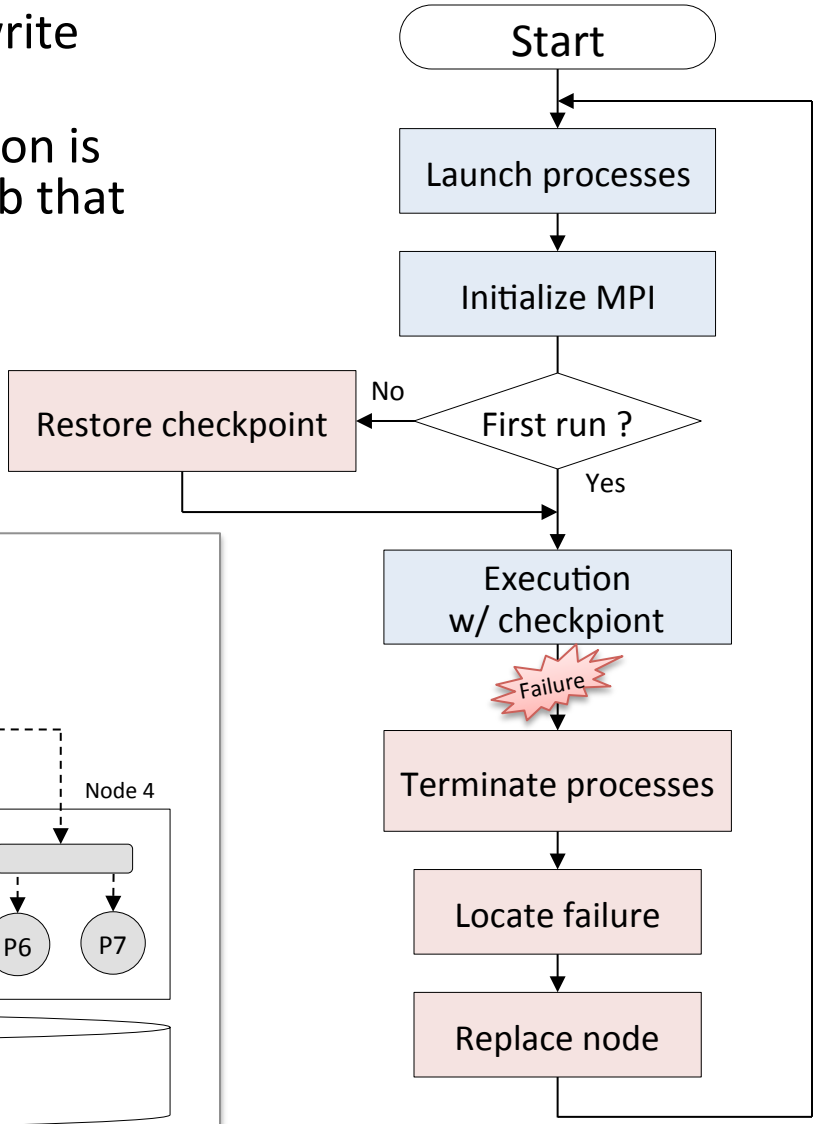




Conventional fault tolerance in MPI app.

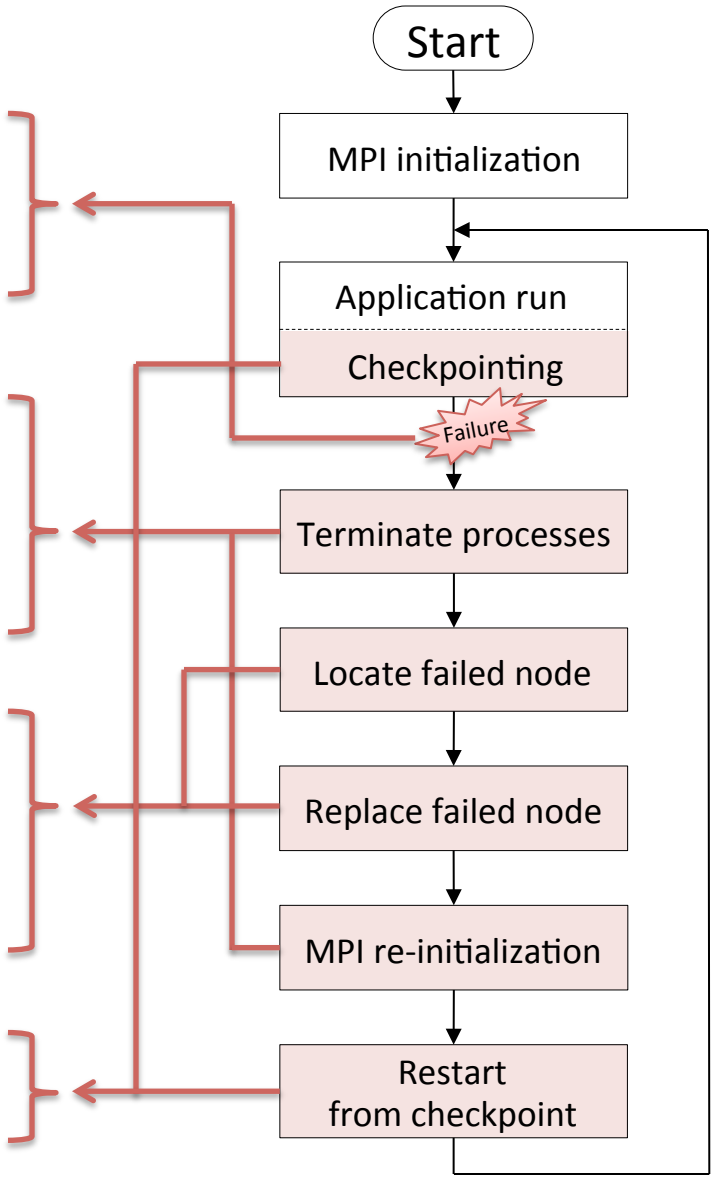
- Long running MPI applications usually write checkpoints
- When a failure occurs, the MPI application is terminated, and relaunched as a new job that restarts from the last checkpoint
- This approach is quite simple





Challenges for fast and transparent recovery

- Scalable failure detection
  - When recovery, all processes need to be notified the failure
- Survivable messaging interface
  - Termination and Initialization of processes will be expensive at extreme scale
- Dynamic node allocation
  - Manual allocation of spare nodes may incur extra overhead or human errors
- Fast checkpoint/restart

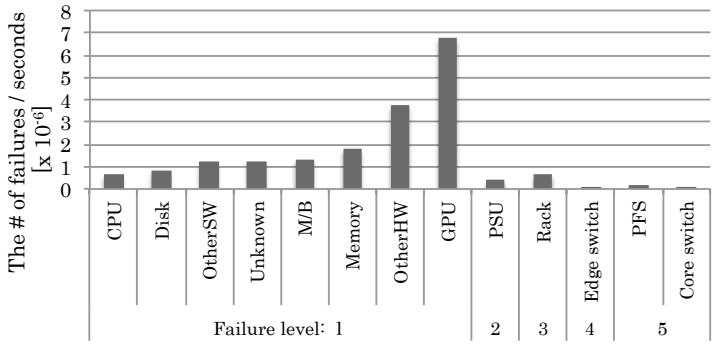


Failure analysis

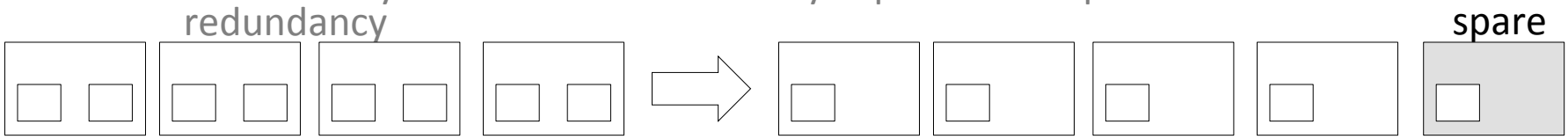
- Several common failures are transparently recovered by hardware and software level because these failure can occur frequently
  - Memory: bit flip by ECC (hardware)
  - Disk: disk failure by RAID (hardware or software)
  - Network: packet loss by TCP/IP (software)
- Not all failures are recoverable
  - GPU, CPU, M/B, Power supply => terminate processes
  - These failure are not exceptional nowadays

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



  - Redundancy of these resource is very expensive => spare node rather than redundancy



Major failures are compute node failures

- Most of failures comes from one node, or can recover from XOR checkpoint
  - e.g. 1) TSUBAME2.0: 92% failures
  - e.g. 2) LLNL clusters: 85% failures

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

Clusters	Coastal	Hera	Atlas	Total
Time span	Oct 09 - Mar 10	Nov 08 - Nov 09	May 08 - Oct 09	
Number of jobs	135	455	281	871
Node hours	2,830,803	1,428,547	1,370,583	5,629,933
Total failures	24	87	80	191
`LOCAL` required	2 (08%)	36 (41%)	21 (26%)	59 (31%)
`PARTNER`/`XOR` required	18 (75%)	32 (37%)	54 (68%)	104 (54%)
`Lustre` required	4 (17%)	19 (22%)	5 (06%)	28 (15%)

How to restore applications data ?

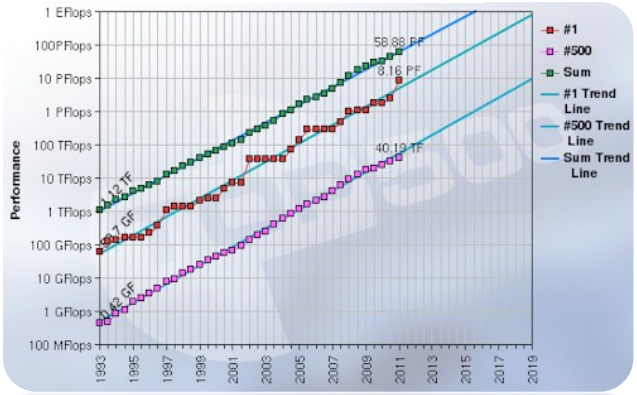
FMI example code

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

- `int FMI_Loop(void **ckpt, int *sizes, int len)`
  - `ckpt` : Array of pointers to data for checkpoint
  - `sizes`: Array of sizes of each checkpoint
  - `len` : Length of ckpt and sizes
- Checkpointing scheme: In-memory XOR encoding across distributed memory
- Current FMI only support a single loop
  - Future FMI will support multiple and nested loops

Failures on HPC systems

- System resiliency is critical for future extreme-scale computing
- 191 failures out of 5-million node-hours
  - A production application using Laser-plasma interaction code (pF3D)
  - Hera, Atlas and Coastal clusters @LLNL



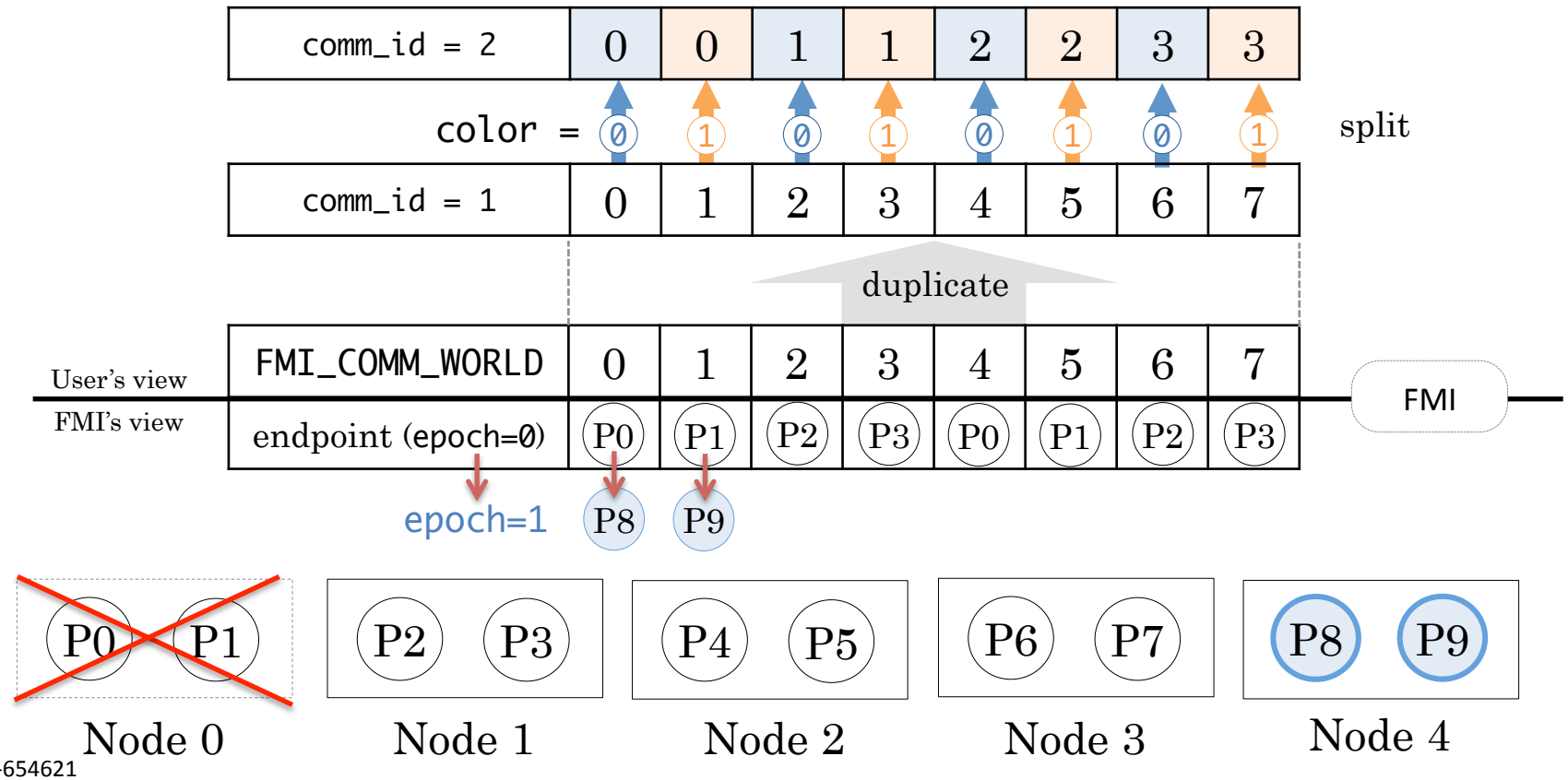
Estimated MTBF (w/o hardware reliability improvement in future)

	1,000 nodes	10,000 nodes	100,000 nodes
MTBF	1.2 days (Measured)	2.9 hours (Estimation)	17 minutes (Estimation)

- Difficult to continuously run for a long time without fault tolerance at extreme scale

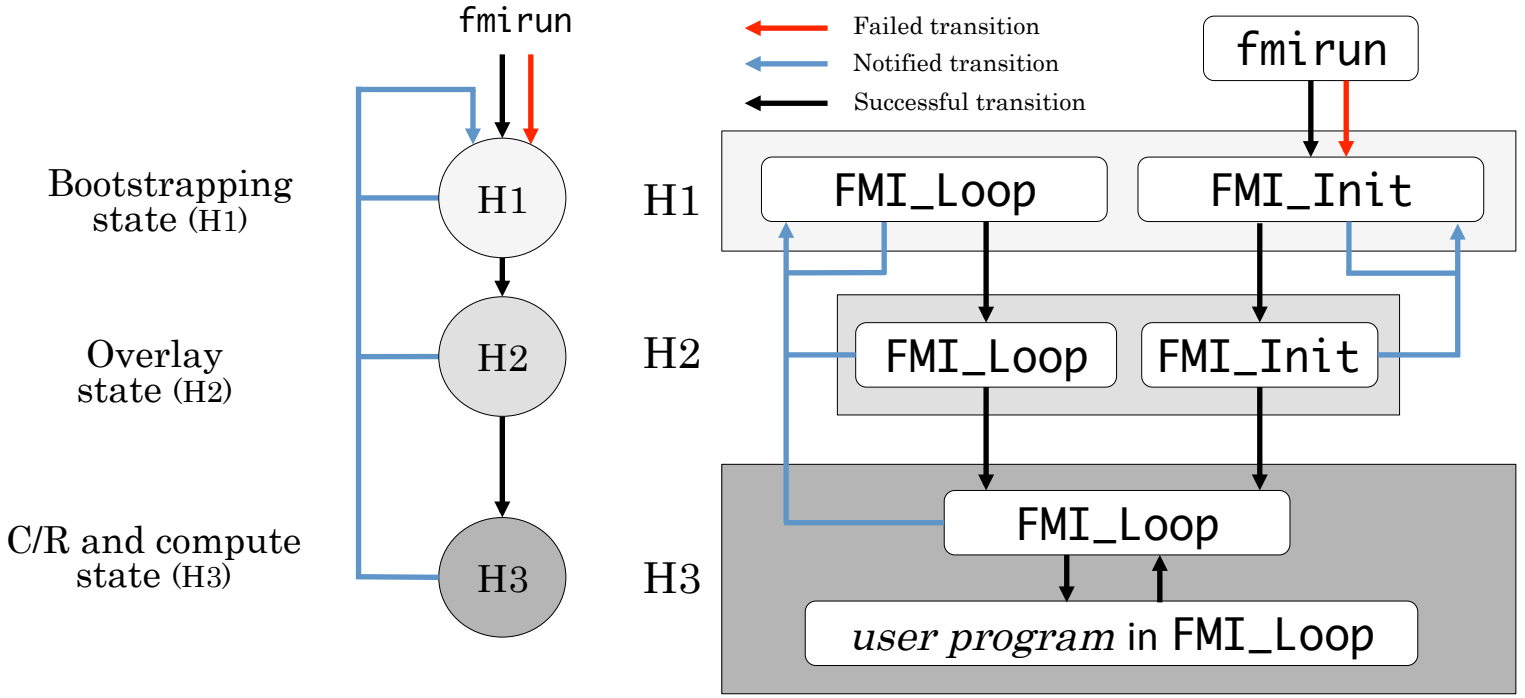
Survivable messaging interface & Dynamic node allocation (cont'd)

- Bootstrapping
  1. Update process mapping between physical processes and virtual rank
  2. replace endpoint information
  3. increment epoch to be able to discard staled messages



Process state manage

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

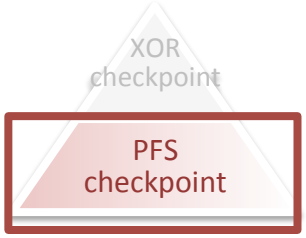


Existing fault tolerant messaging interface

	FT-MPI (ULFM)	SCR	Charm++
Survivable	Yes		Yes
Failure detection (scalability)	Yes (w/ explicit call)		Yes (not scalable)
Auto-recovery	No		Yes
MPI Compatibility	Yes	Yes	Yes (w/o FT feature)
Scalable checkpoint		Yes	Yes (Only partner)

⇒ A survivable messaging interface (w/ scalable detection, recovery, and checkpoint/restart) is critical

Checkpoint/Restart

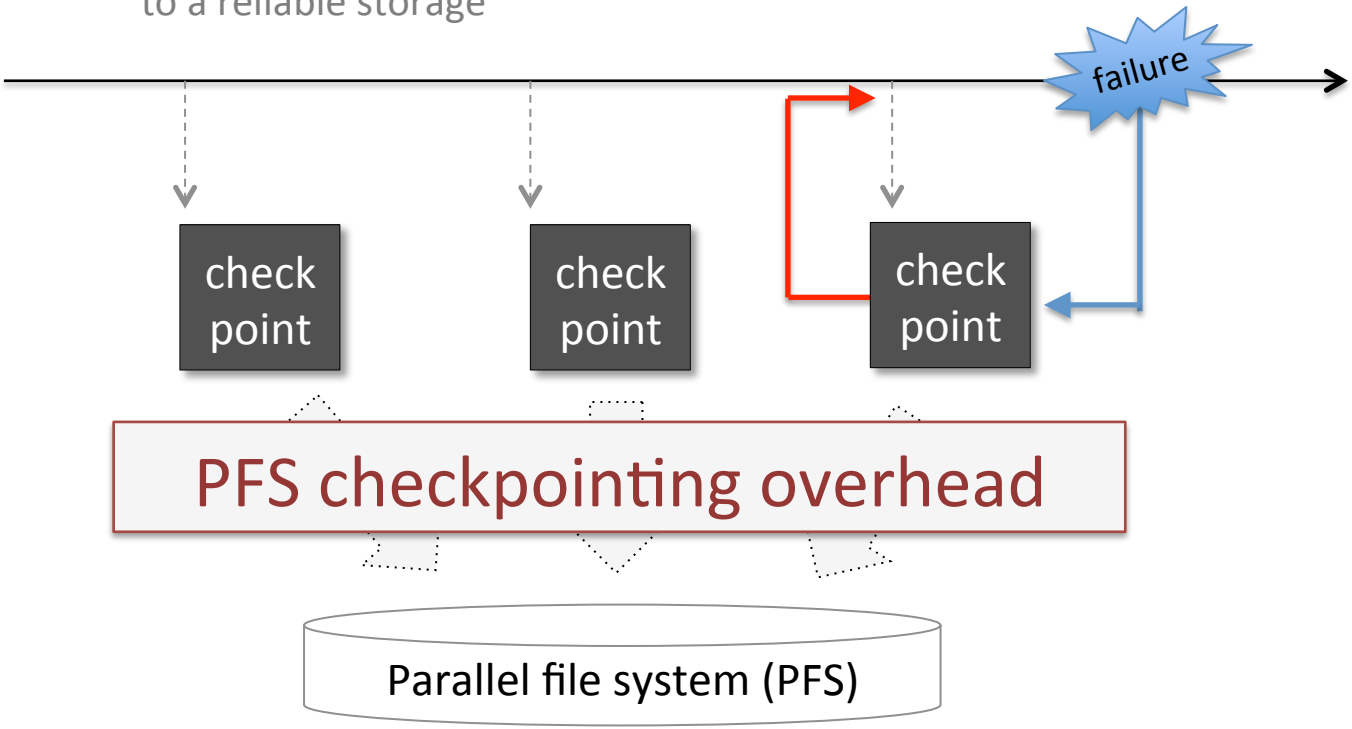


Checkpoint

Periodically save a snapshot of an application state to a reliable storage

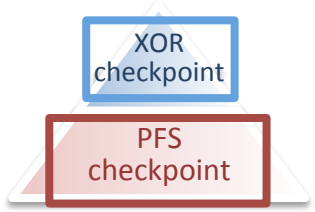
Restart

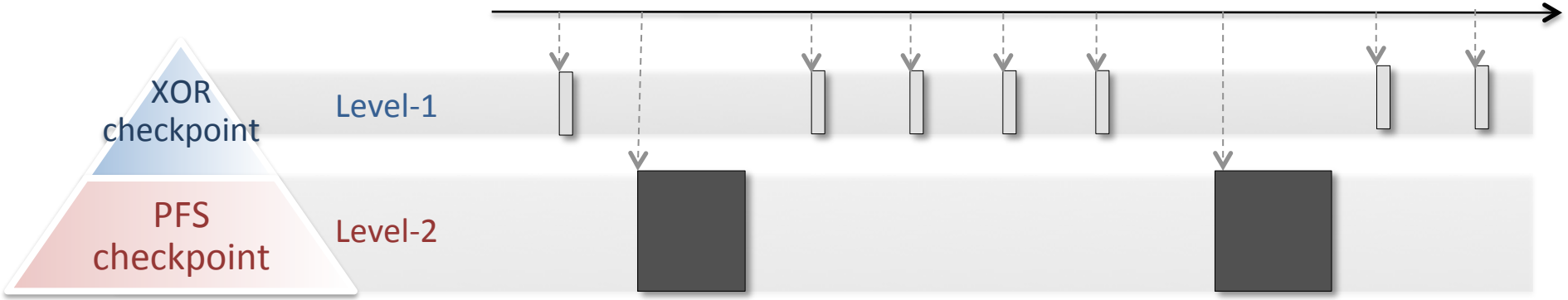
On a failure, restart the execution from the latest checkpoint



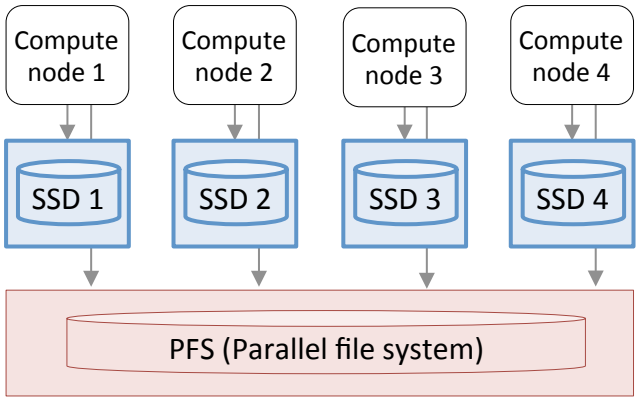
Mostly these checkpoints are stored in a PFS

Multi-level checkpointing (MLC)

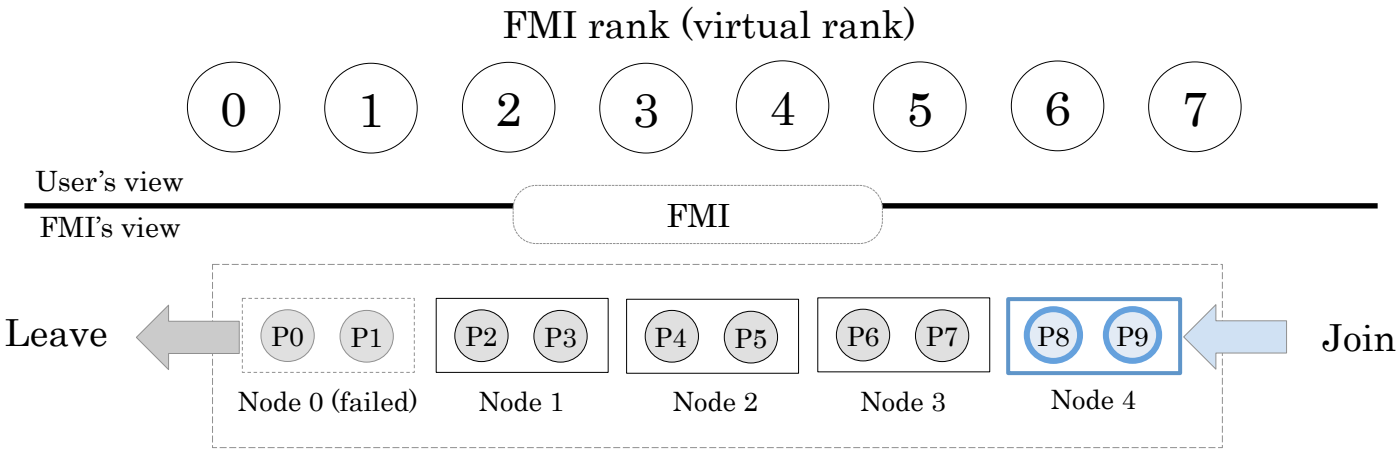


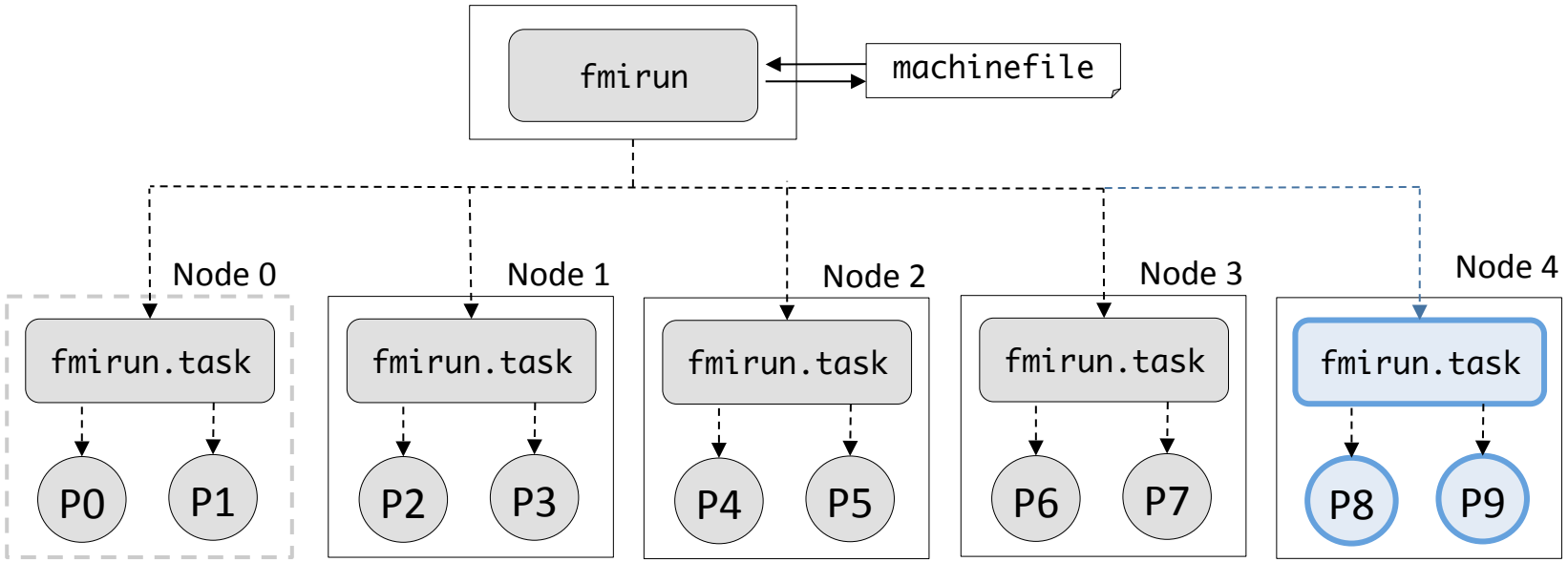


- Use storage levels hierarchically
  - XOR checkpoint: Frequently
    - for one node or a few node failure
  - PFS checkpoint: Less frequently
    - for multi-node failure



- Restart overhead
  - 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.
- Fast checkpoint/restart
  - 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
- Detection


FMI rank (virtual rank)
0
1
2
3
4
5
6
7
User's view
FMI's view
FMI
Leave
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
Join
Node 0 (failed)
Node 1
Node 2
Node 3
Node 4


fmirun
machinefile
Node 0
Node 1
Node 2
Node 3
Node 4
fmirun.task
fmirun.task
fmirun.task
fmirun.task
fmirun.task
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9

	Coastal	Hera	Atlas	Total
Batch	1088	800	1072	
debugs	16	16	16	
login	8	6	8	
total	1152	864		
LOCAL	(2.1359306e-07, 4681800.0, 'sec', 78030.0, 'min', 1300.5, 'H', 54.1875, 'D’)	(5.7142857e-06, 175000.0, 'sec', 2916, 'min', 48.611111, 'H', 2.02546296, 'D’)	(4.5644235e-06, 219085, 'sec', 3651, 'min', 60.8571428, 'H', 2.53571428, 'D’)	
XOR	(1.9223375e-06, 520200, 'sec', 8670, 'min', 144.5, 'H', 6.02083, 'D')	(5.079365e-06, 196874, 'sec', 3281, 'min', 54.6875, 'H', 2.2786458, 'D')	(1.1737089e-05, 85200, 'sec', 1420, 'min', 23.6666666, 'H', 0.9861111, 'D')	
L1	(2.135930624973e-06, 468180, 'sec', 7803, 'min', 130.05, 'H', 5.41875, 'D')	(1.0793650793e-05, 92647, 'sec', 1544, 'min', 25.7352941, 'H', 1.0723039215, 'D')	(1.63015127803e-05, 61344, 'sec', 1022, 'min', 17.04, 'H', 0.71, 'D')	
Lustre (L2)	(4.2718612e-07, 2340900.0, 'sec', 39015.0, 'min', 650.25, 'H', 27.09375, 'D')	(3.0158730e-06, 331578, 'sec', 5526, 'min', 92.1052631, 'H', 3.83771929, 'D')	(1.086767e-06, 920159, 'sec', 15335, 'min', 255.59999, 'H', 10.6499999, 'D')	

Table 2: pF3D failures on three different clusters

Clusters	Coastal	Hera	Atlas	Total
Time span	Oct 09 - Mar 10	Nov 08 - Nov 09	May 08 - Oct 09	
Number of jobs	135	455	281	871
Node hours	2,830,803	1,428,547	1,370,583	5,629,933
Total failures	24	87	80	191
`LOCAL` required	2 (08%)	36 (41%)	21 (26%)	59 (31%)
`PARTNER`/`XOR` required	18 (75%)	32 (37%)	54 (68%)	104 (54%)
`Lustre` required	4 (17%)	19 (22%)	5 (06%)	28 (15%)