

# *Billion-Way Resiliency for Extreme Scale Computing*

Seminar at German Research School for Simulation Sciences, Aachen

October 6<sup>th</sup>, 2014

Kento Sato

Lawrence Livermore National Laboratory



LLNL-PRES-662034

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



# Failures on HPC systems

- Exponential growth in computational power
  - Enables finer grained simulations with shorter period time
- Overall failure rate increase accordingly because of the increasing system size
- 191 failures out of 5-million node-hours
  - A production application of Laser-plasma interaction code (pF3D)
  - Hera, Atlas and Coastal clusters @LLNL

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

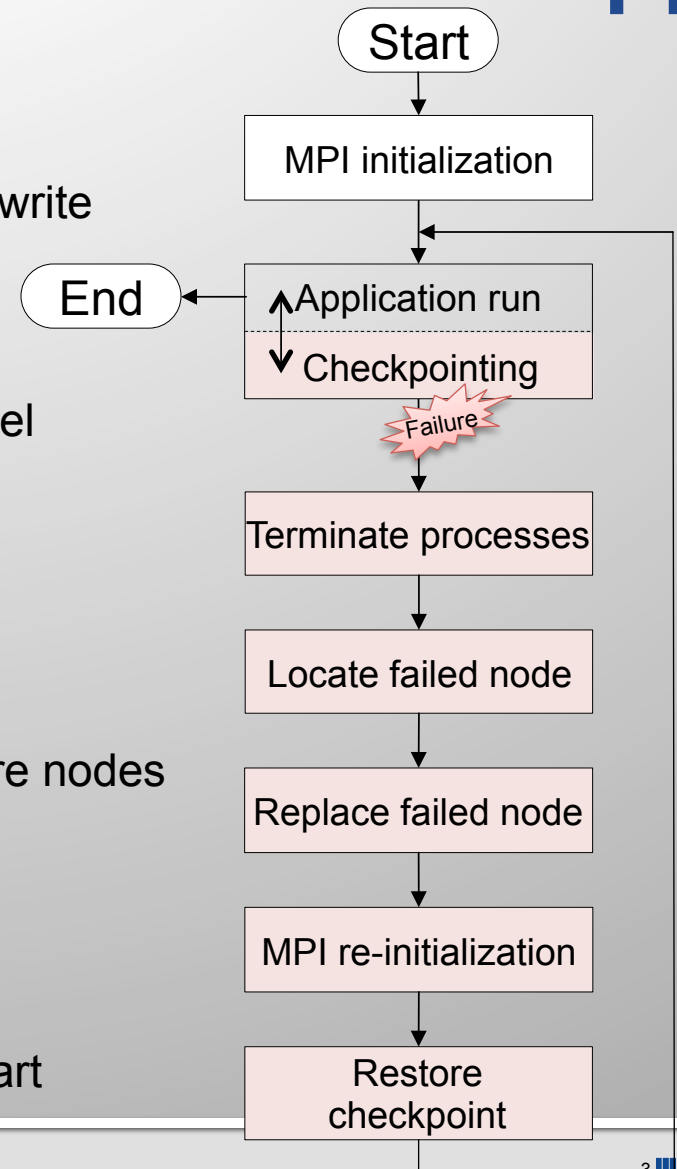
	1,000 nodes	10,000 nodes	100,000 nodes
MTBF	1.2 days (Measured)	2.9 hours (Estimation)	17 minutes (Estimation)

Source: A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System (SC 10)"

- Will be difficult for applications to continuously run for a long time without fault tolerance at extreme scale

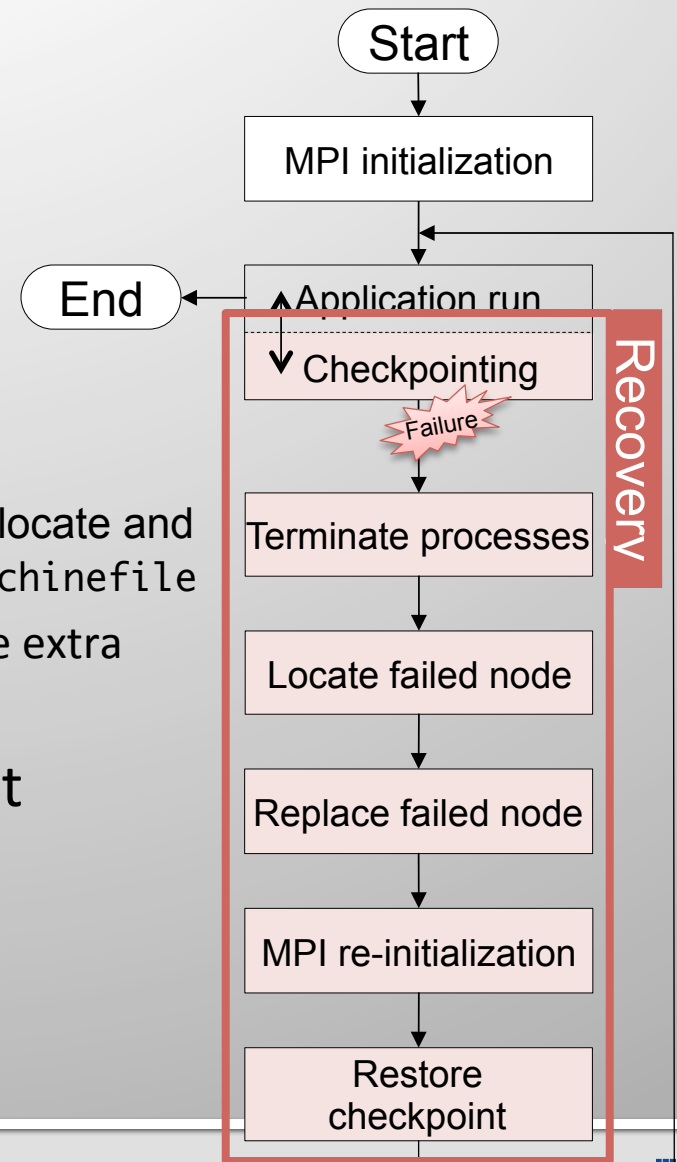
# Conventional fault tolerance in MPI apps

- 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

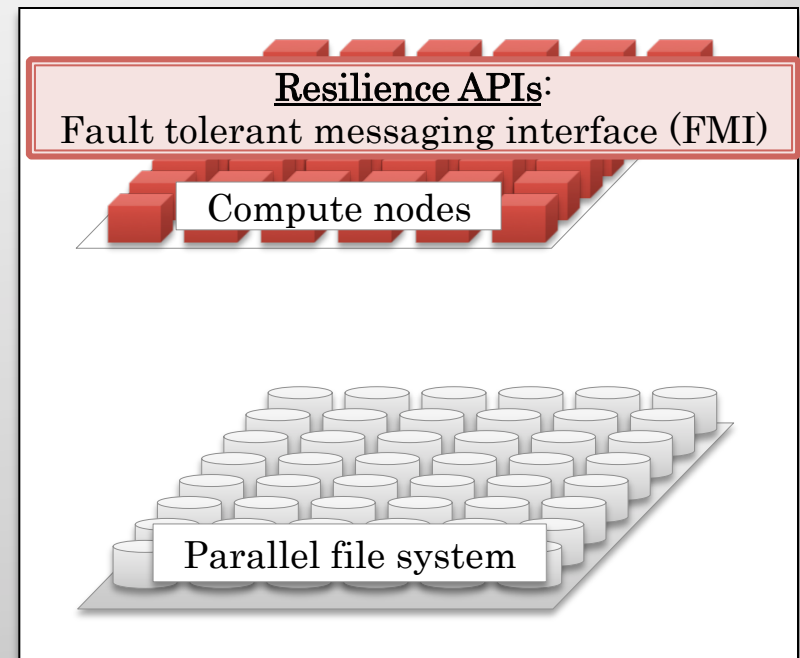
- Failure rate will increase in future extreme scale systems
- Applications will use more time for recovery
  - Whenever a failure occurs, users manually locate and replace the failed nodes with spare nodes via machinefile
  - The manual recovery operations may introduce extra overhead and human errors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





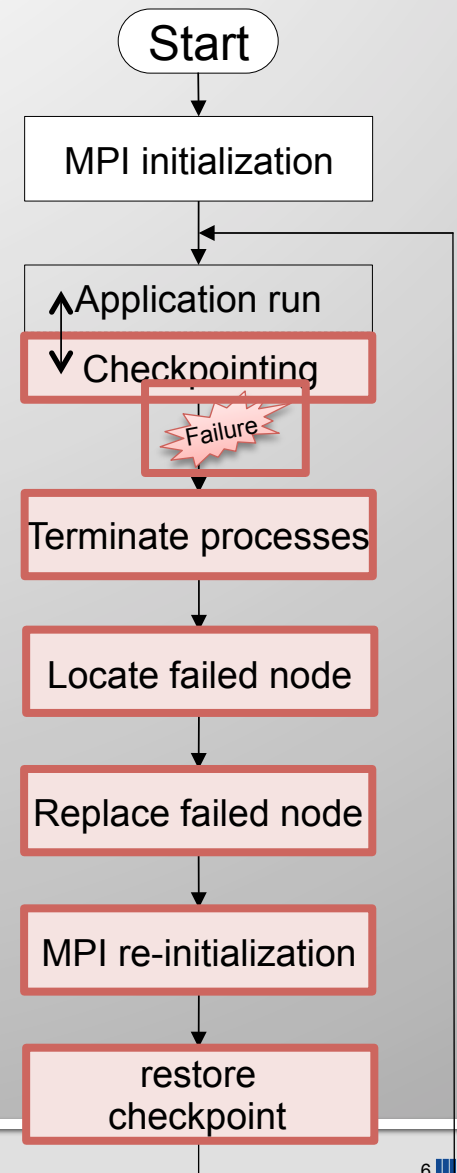
# Resilience APIs, Architecture and the model

- Resilience APIs
  - ⇒ Fault tolerant messaging interface (FMI)



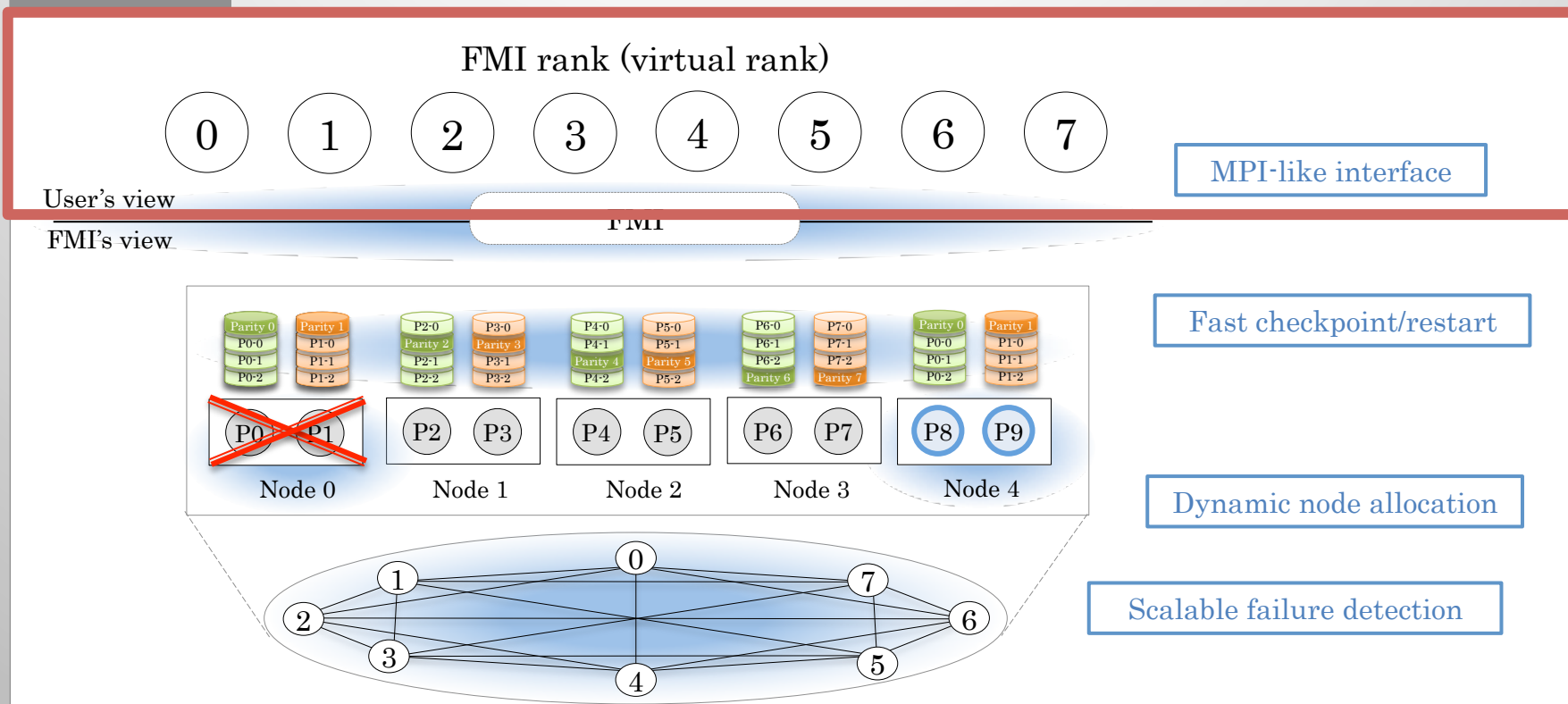
# Challenges for fast and transparent recovery

- 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 [IPDPS2014]

## FMI overview



- 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

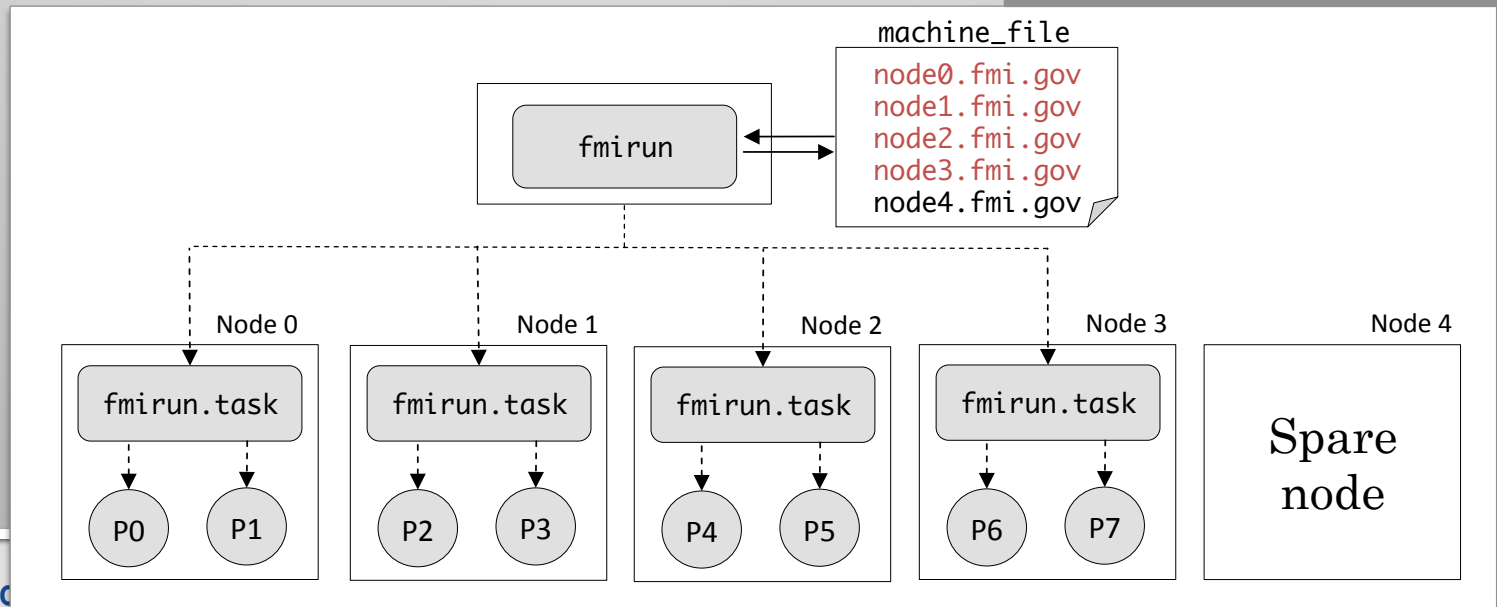
# How FMI applications work ?

## 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(...)

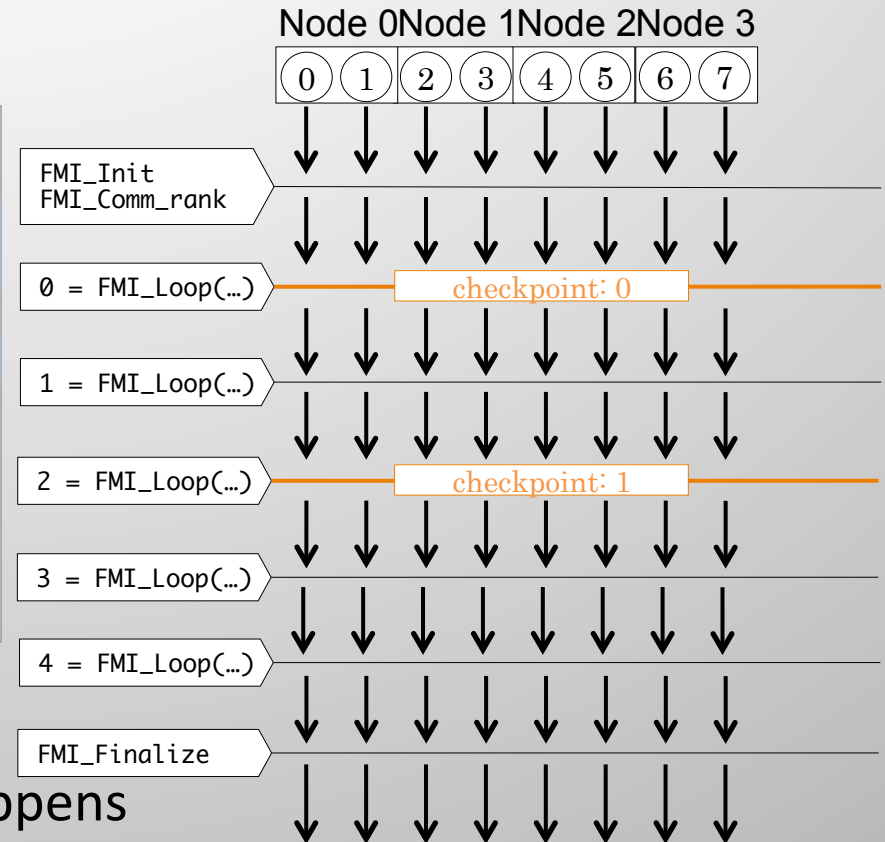
## Launch FMI processes



# 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

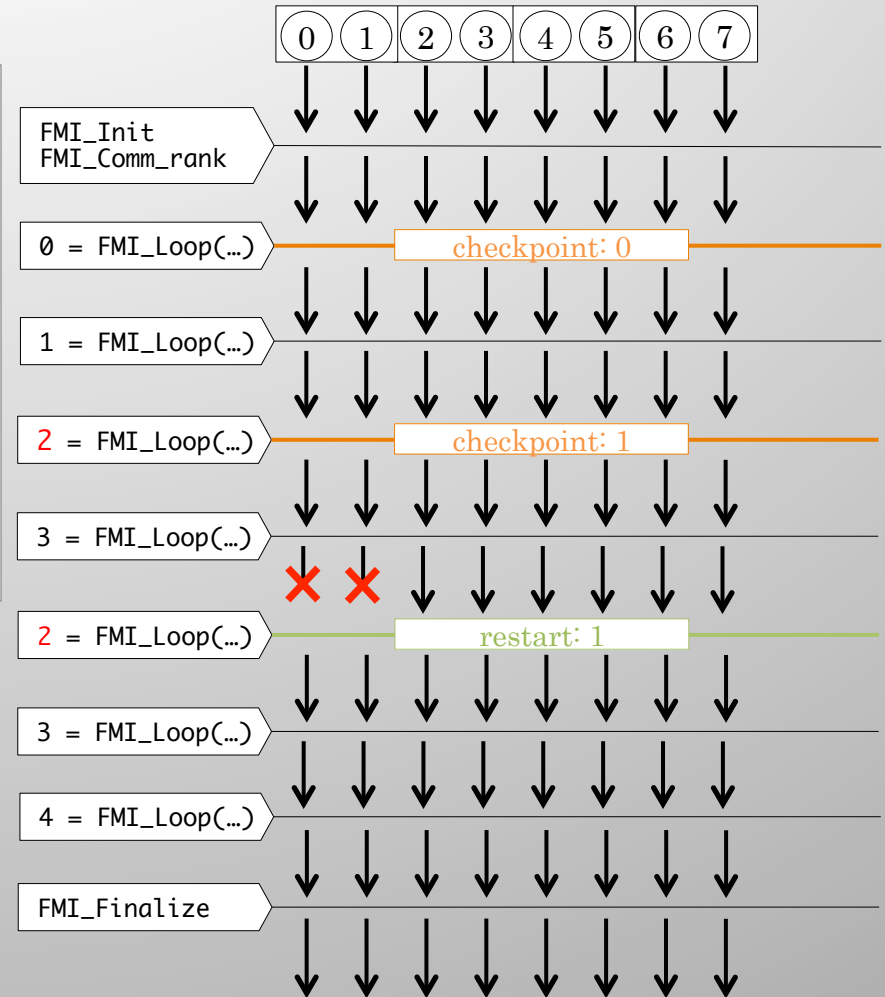


# User perspective : Failure

## 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 from the last checkpoint
  - 2<sup>th</sup> 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

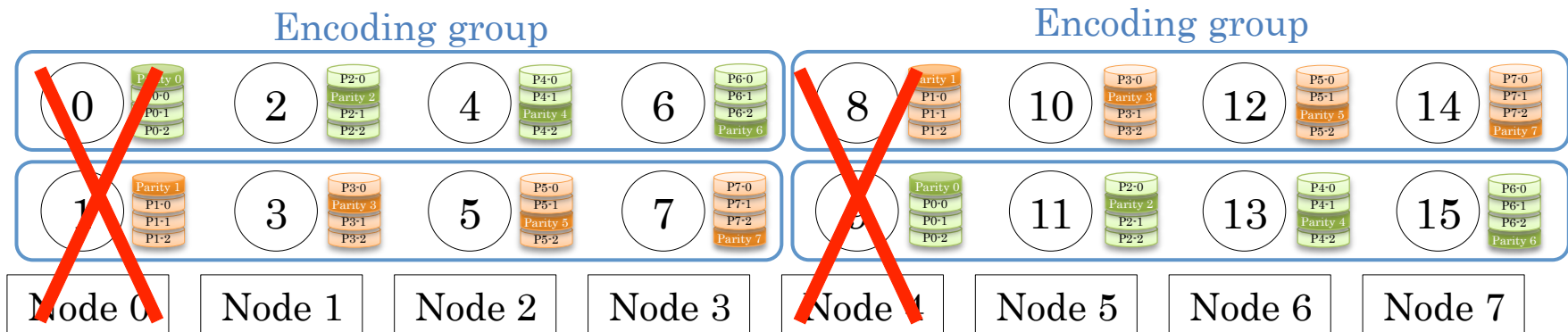
**sizes** : Array of sizes of each checkpointed variables

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

returns iteration id

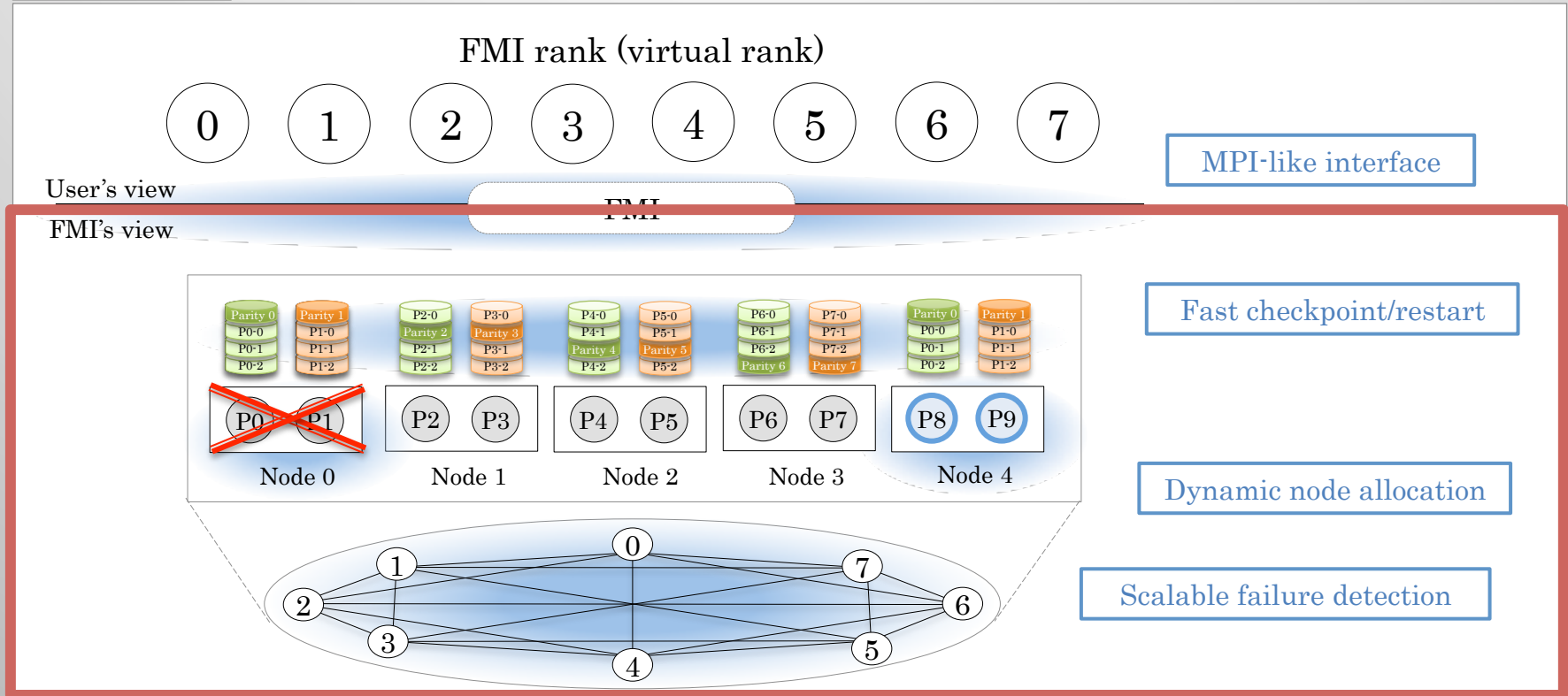
- FMI constructs in-memory RAID-5 across compute nodes
- Checkpoint group size
  - e.g.) group\_size = 4

## FMI checkpointing



# FMI: Fault Tolerant Messaging Interface

## FMI overview



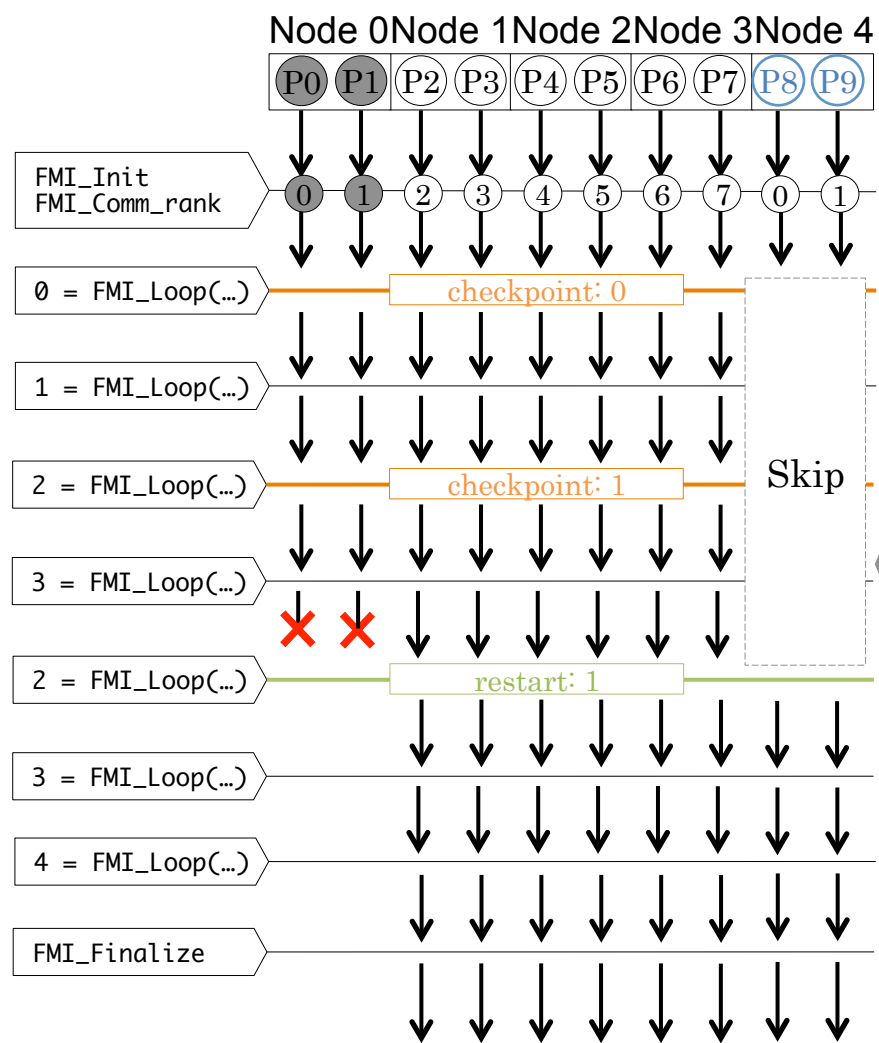
- FMI is an MPI-like survivable messaging interface
  - Scalable failure detection => Overlay network for failure detection
  - Dynamic node allocation => FMI ranks are virtualized
  - Fast checkpoint/restart => Diskless checkpoint/restart

# FMI's view

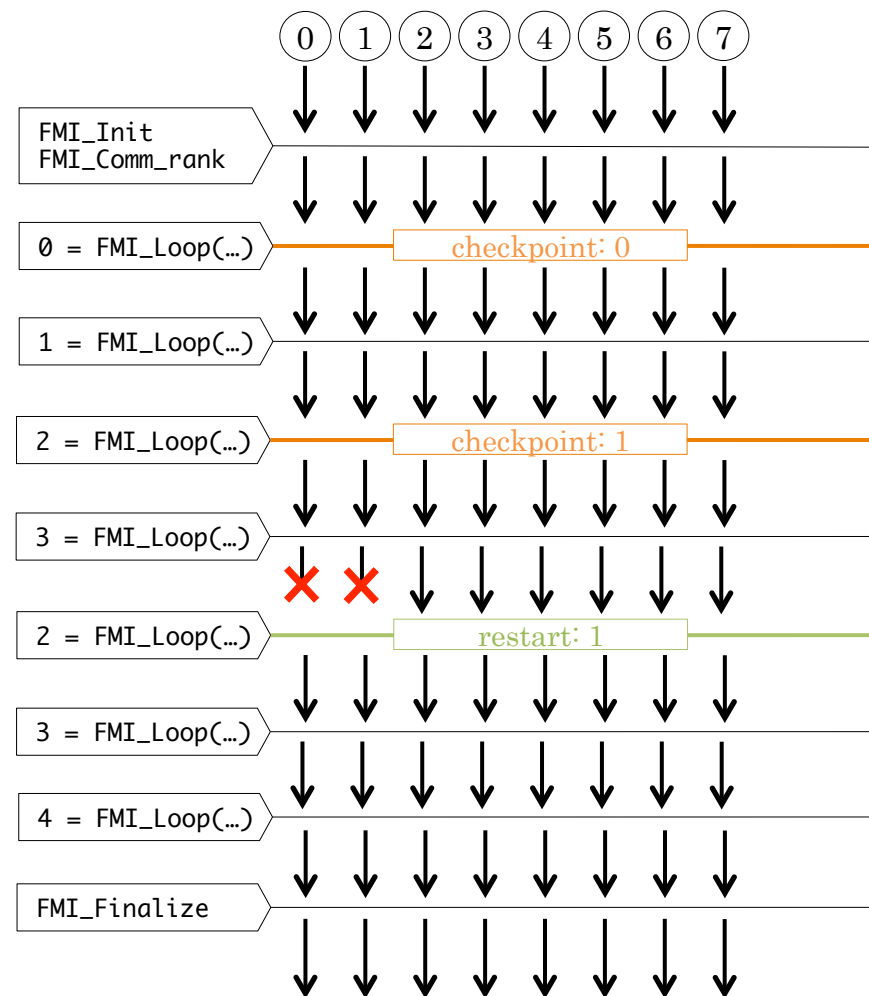
&

# User's view

## FMI's view

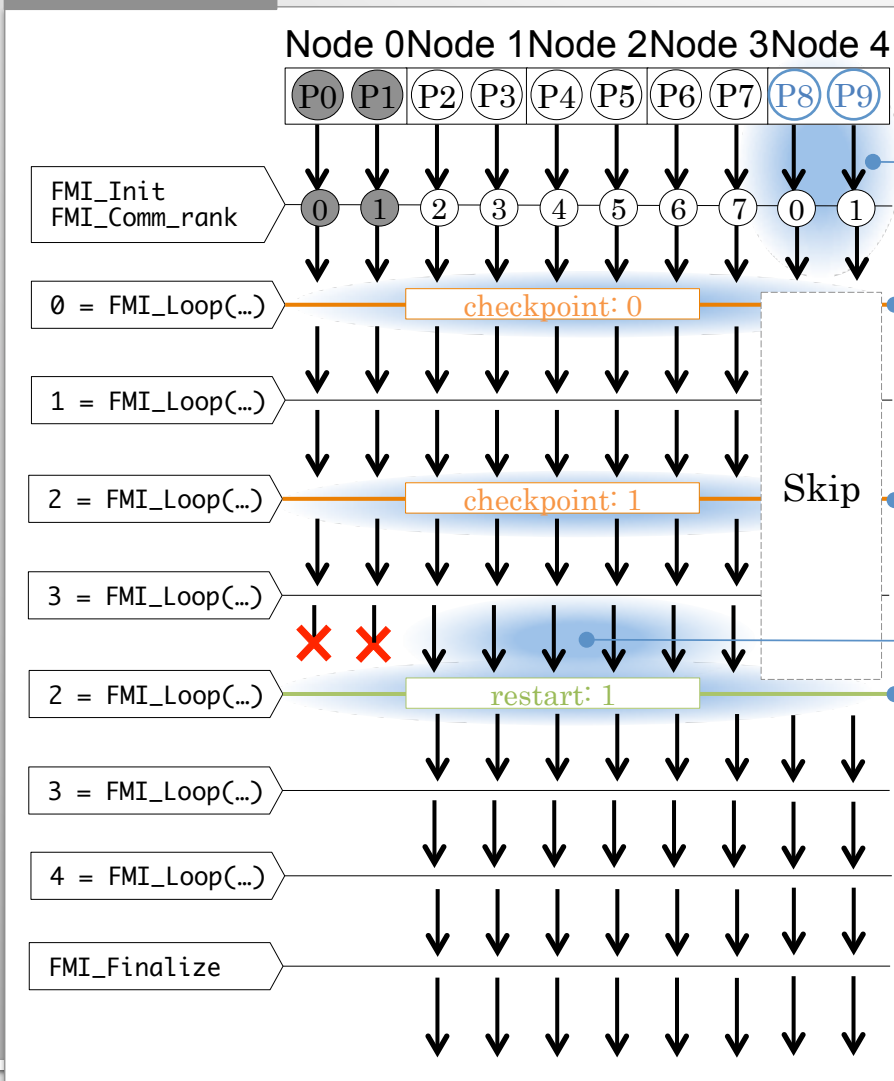


## User's view



# FMI's view

## FMI's view



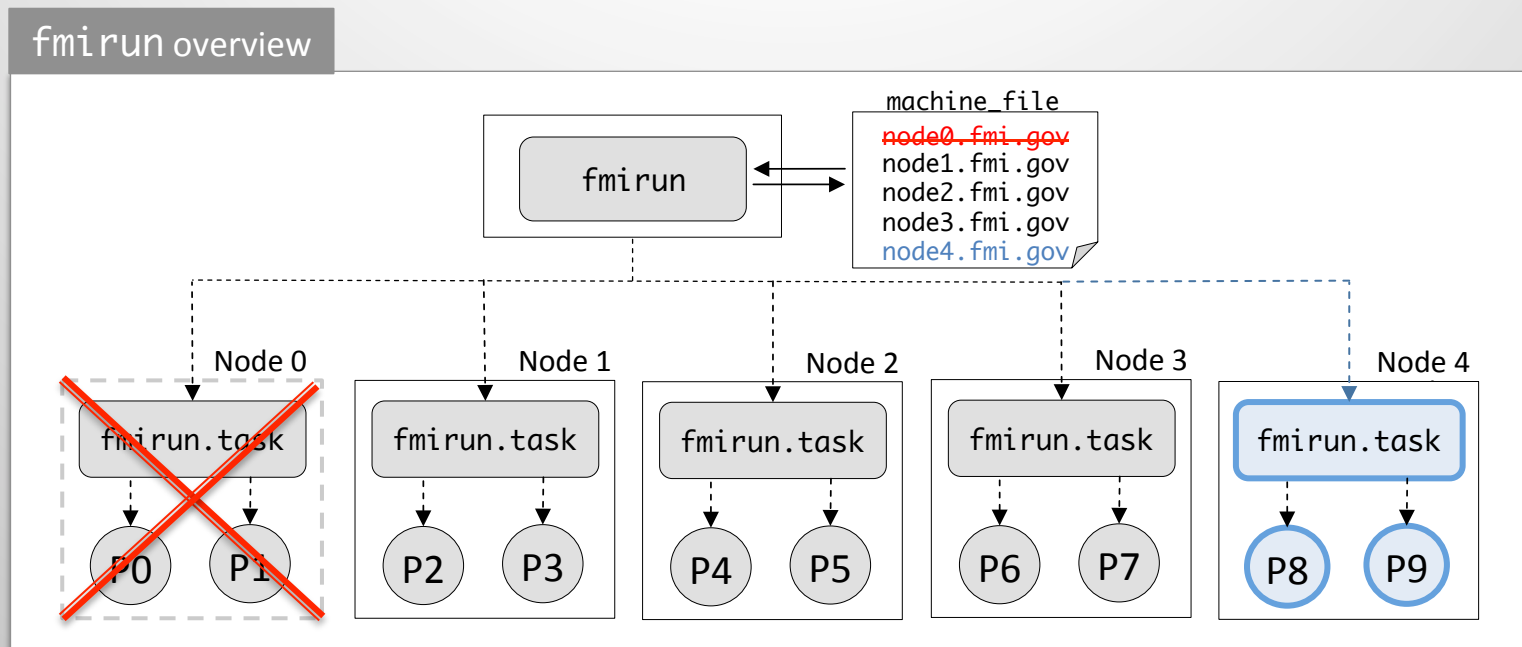
Transparent & Dynamic  
node allocation

Scalable failure detection &  
notification

Fast checkpoint/restart



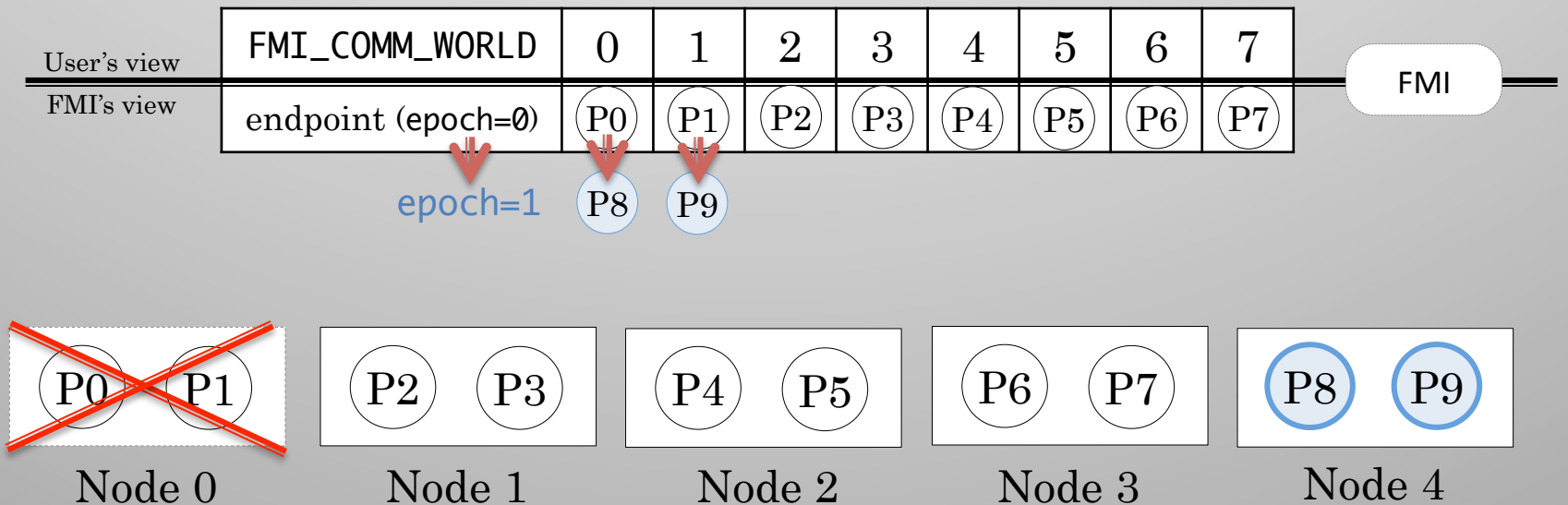
# Transparent and dynamic node allocation



- 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** broadcasts connection information (endpoint) of new processes, P8 and P9

# Transparent and dynamic node allocation (cont'd)

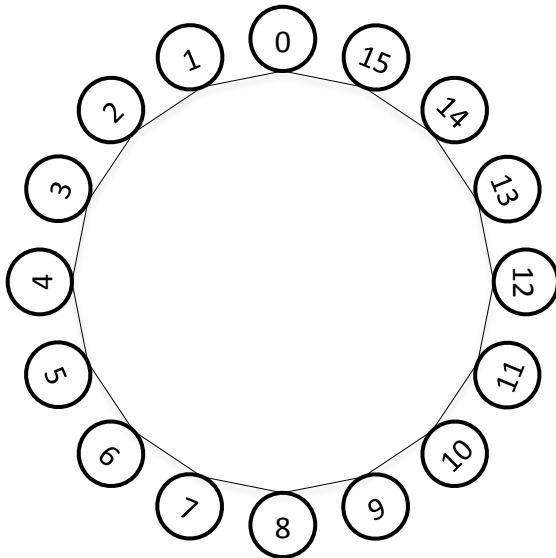
- In FMI, FMI\_COMM\_WORLD manages process mapping between FMI ranks and processes
  - Once receiving endpoints, the mapping table is updated (=> **bootstrapping**)
    - Applications can still use the same ranks
  - Then, increment a “**epoch**” number to be able to discard staled messages
    - After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



# Scalable failure detection

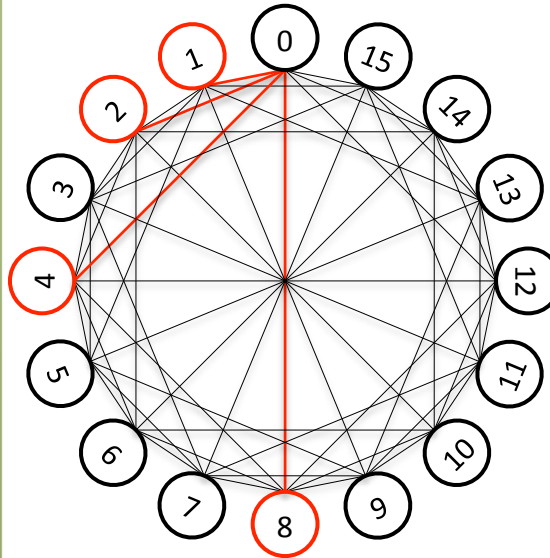
- FMI processes check if other processes are alive or not each other using overlay network
- Log-ring overlay network
  - Each FMI rank connects to  $2^k$ -hop neighbors ( $k=0,1,\dots$ )
  - e.g. ) FMI rank 0 connects to FMI rank 1, 2, 4 and 8
- Log-ring overlay is scalable for both construction and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

Ring overlay



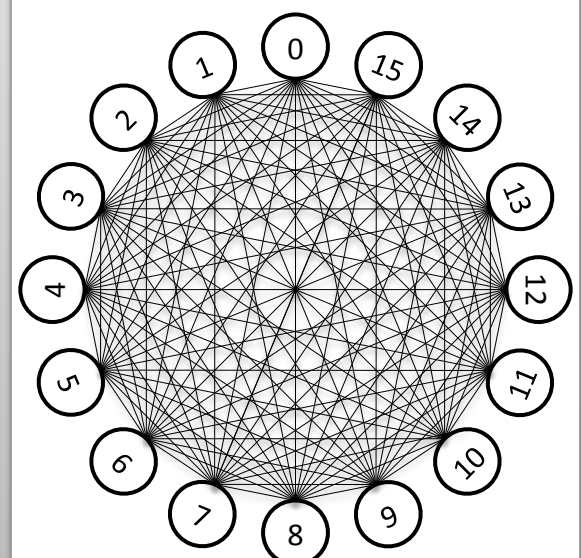
Construction:  $O(1)$   
Global detection:  $O(N)$

Log-ring overlay



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

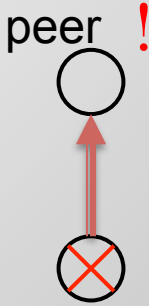
Complete overlay



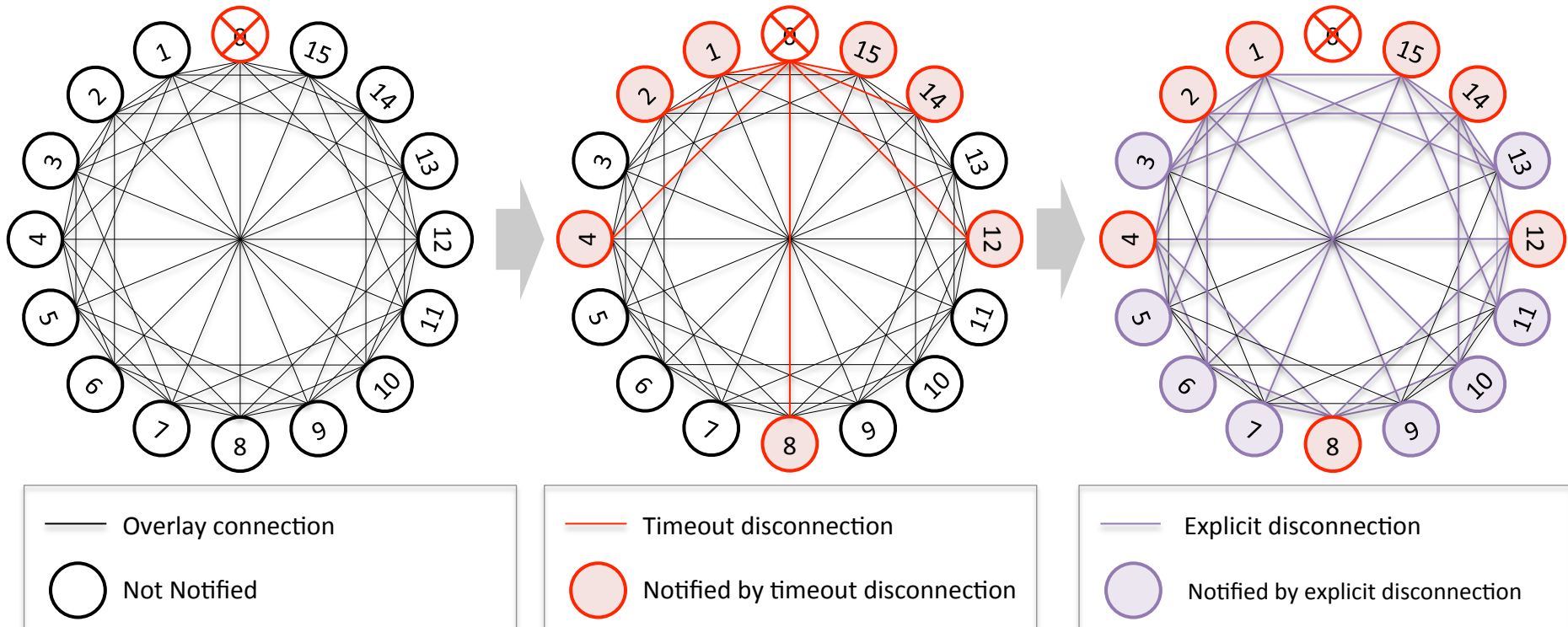
Construction:  $O(N)$   
Global detection:  $O(1)$

# Scalable failure detection (cont'd)

- Log-ring overlay network using ibverbs (constructed in FMI\_init(...))
  - 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

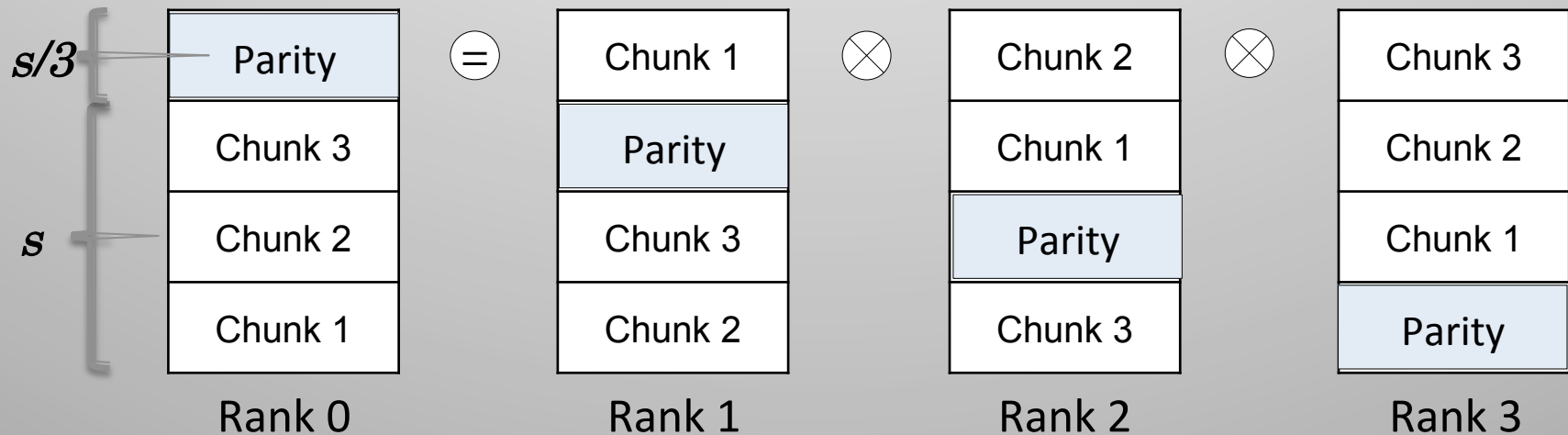


Example of global failure notification



# In-memory XOR checkpoint/restart algorithm

- XOR checkpoint/restart algorithm
  1. Write checkpoint using memcpy
  2. Divides into chunks, and allocate memory for parity 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).

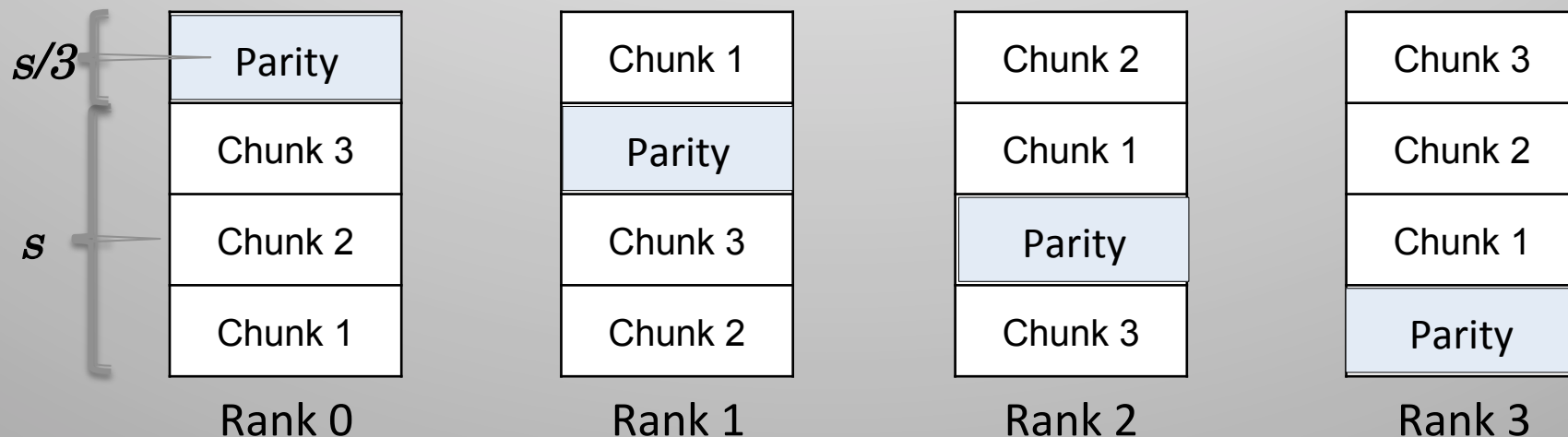


# In-memory XOR checkpoint/restart model

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

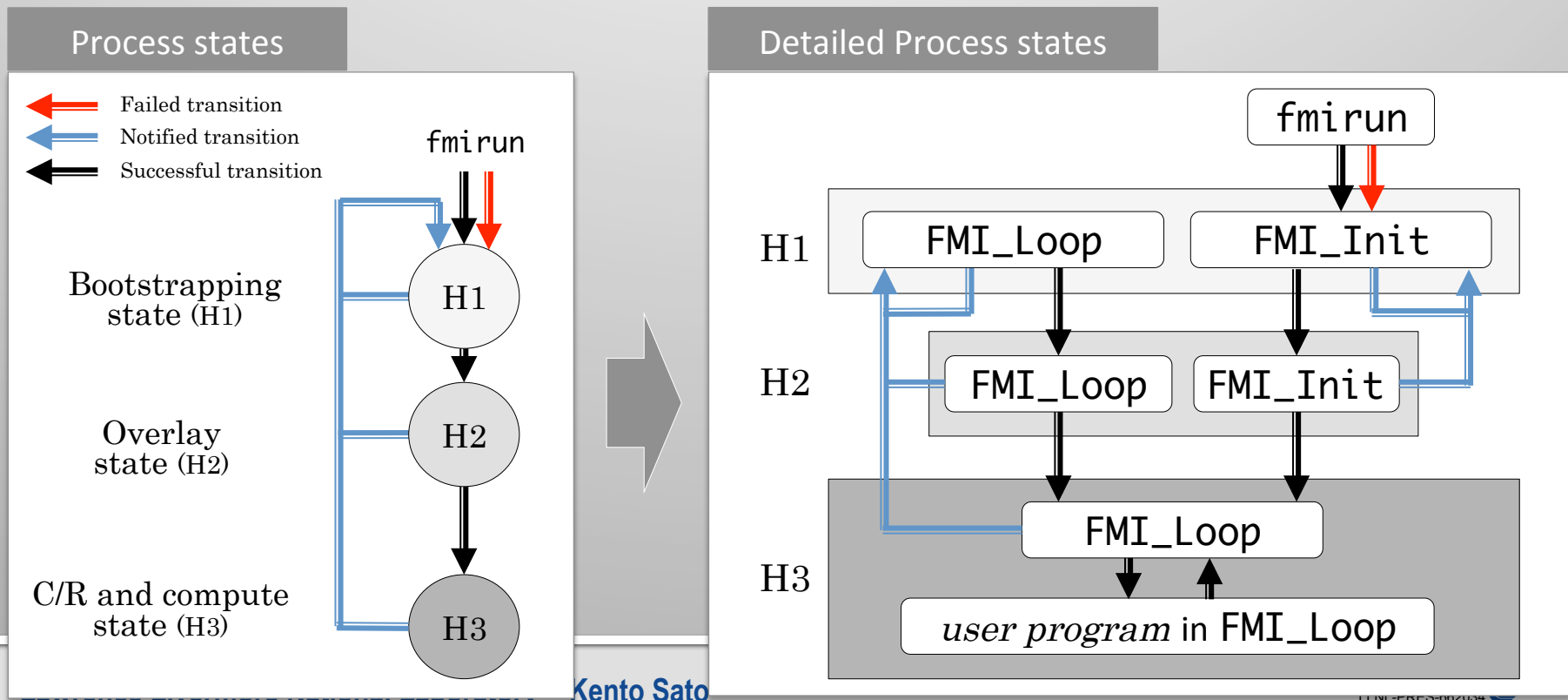
$s$ : ckpt size,  $n$ : group size,  $mem\_bw$ : memory bandwidth,  $net\_bw$ : network bandwidth

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



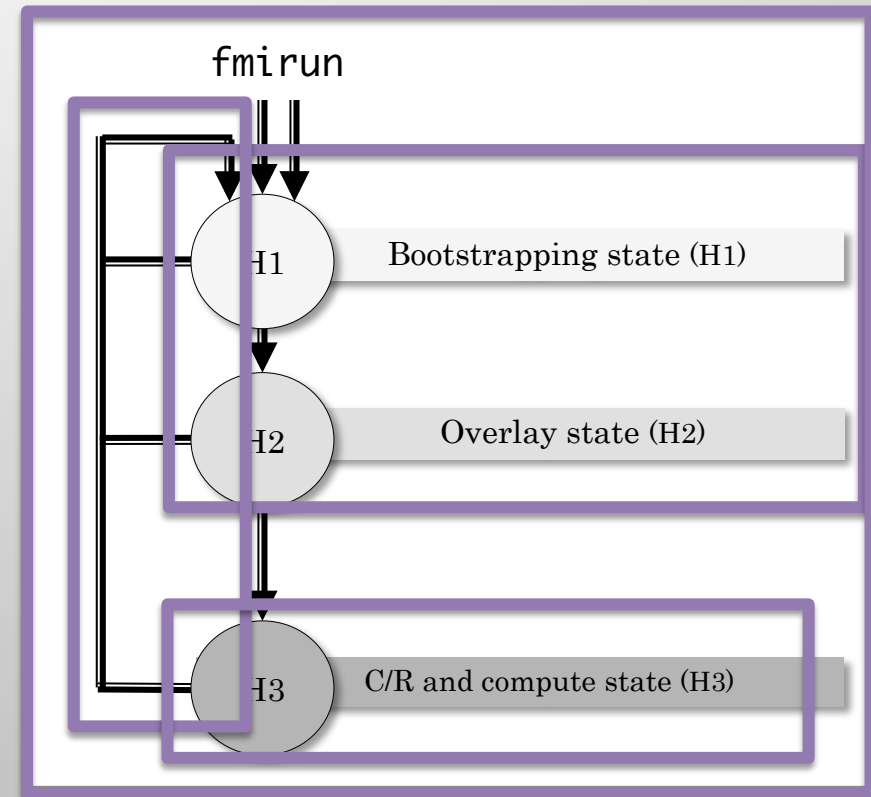
# Process state manage

- 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



# Experimental environment

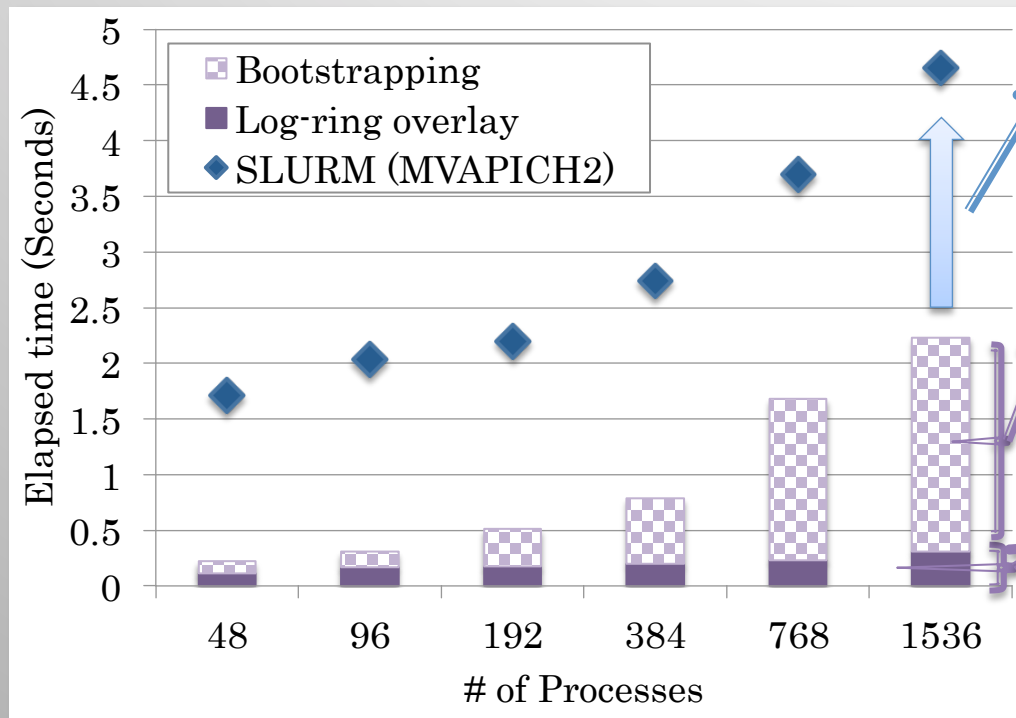
- Sierra cluster @LLNL

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 `mpi run` 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  
Current FMI do only minimal initialization to start an application

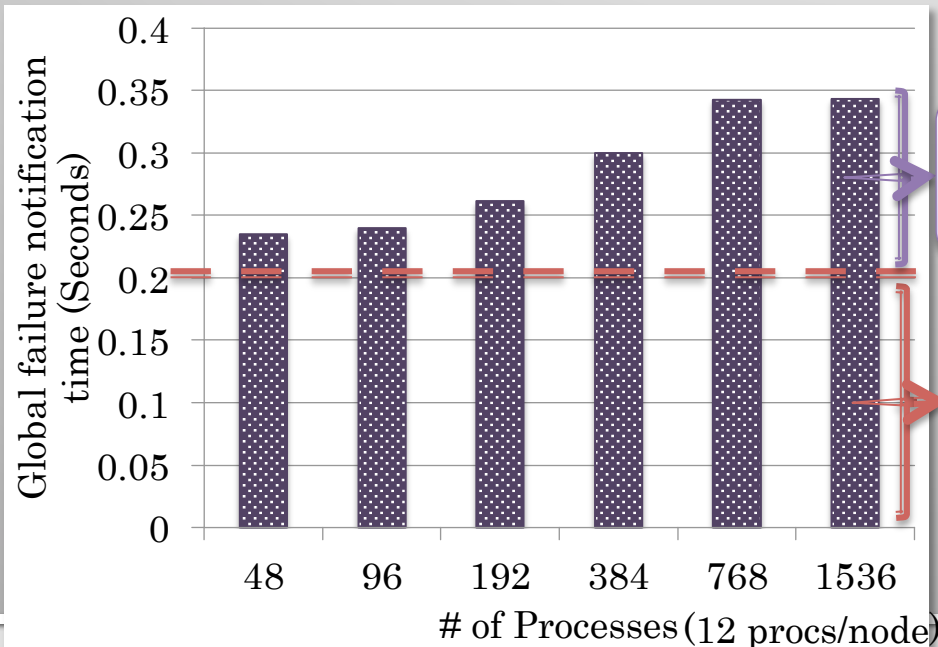
Log-ring construction time is small  
the overlay construction time is  $O(\log(n))$

MPI Initialization: MVAPICH2 MPI\_Init(...) launched by srun



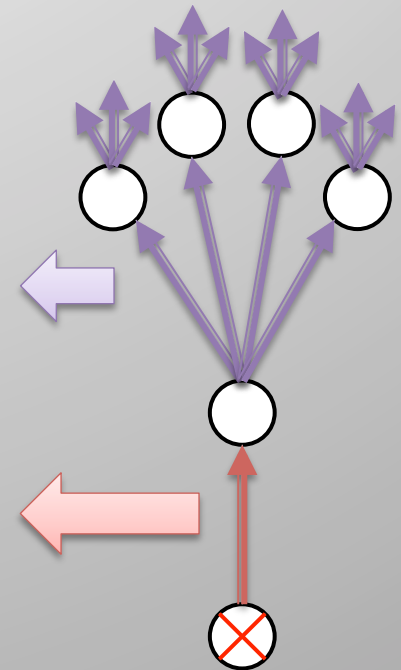
# FMI failure detection time

- 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



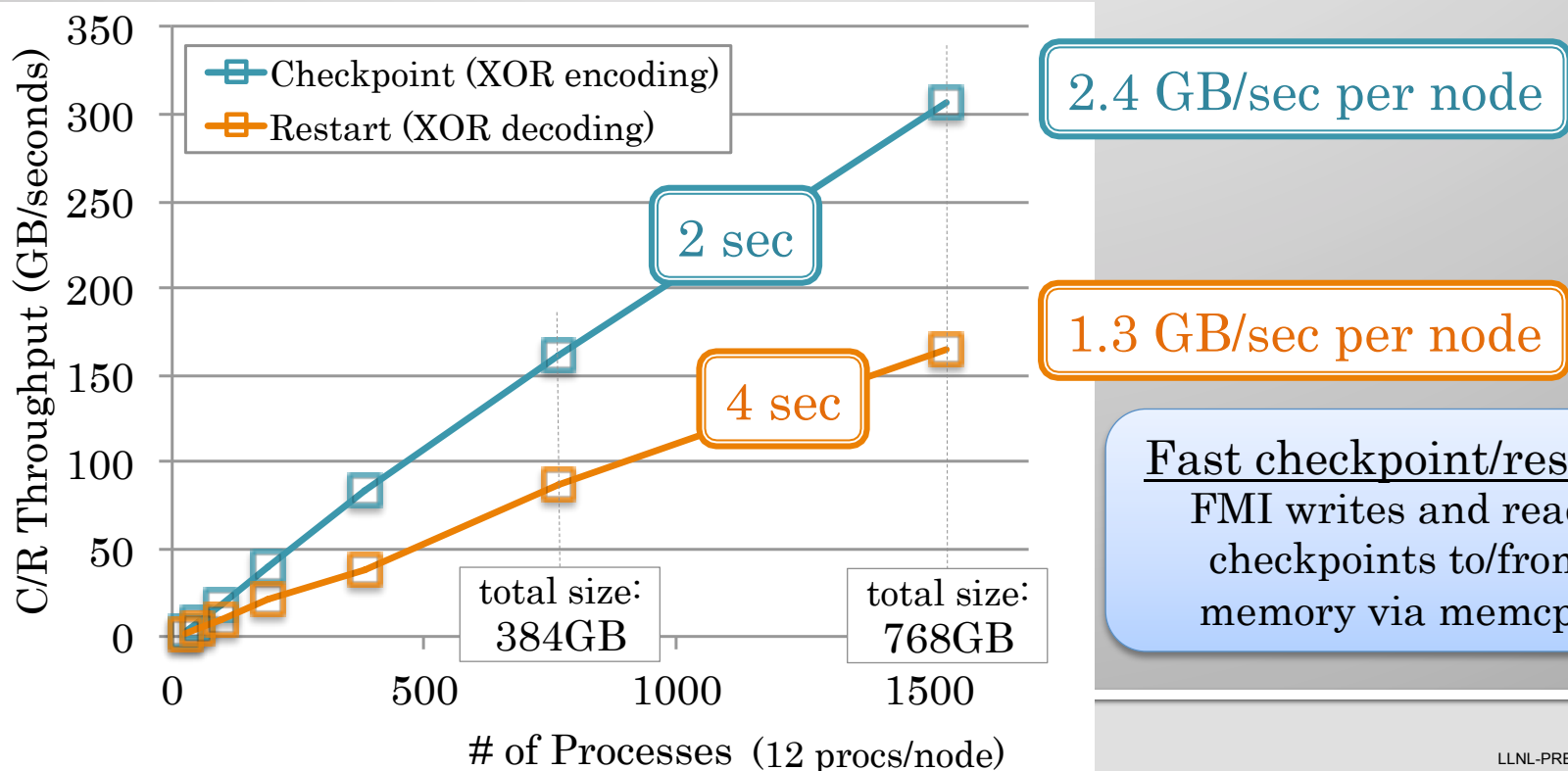
Explicit disconnection  
Exponentially  
propagate notification

Timeout disconnection  
about 200 ms



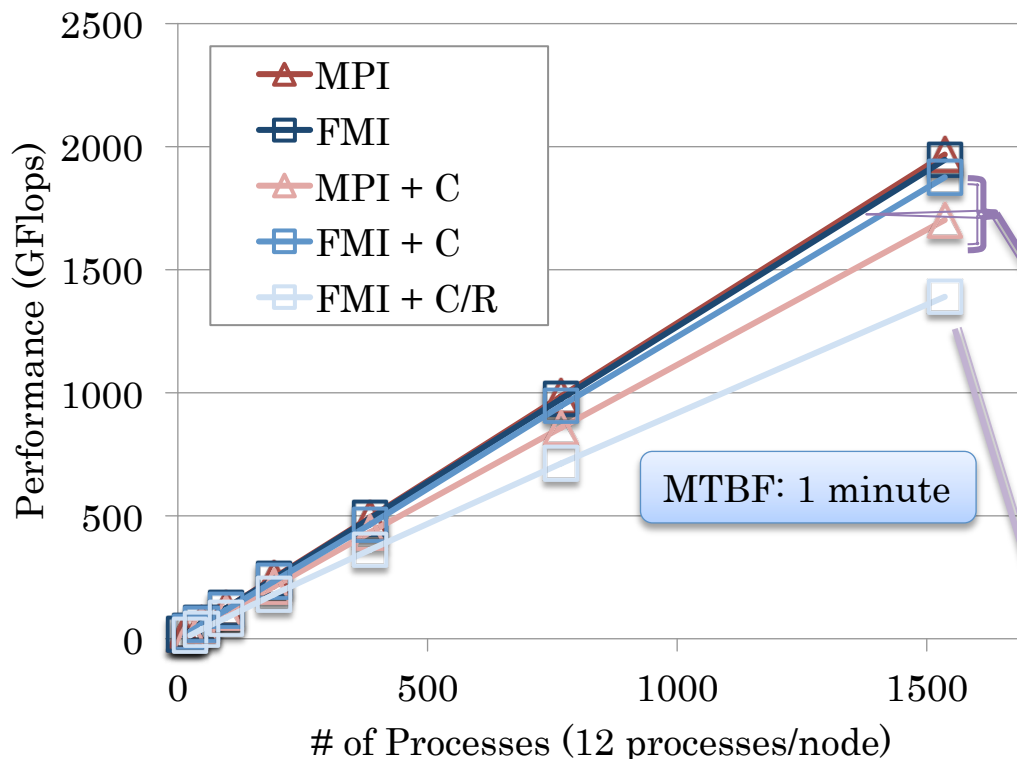
# FMI Checkpoint/Restart throughput

- 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



# 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

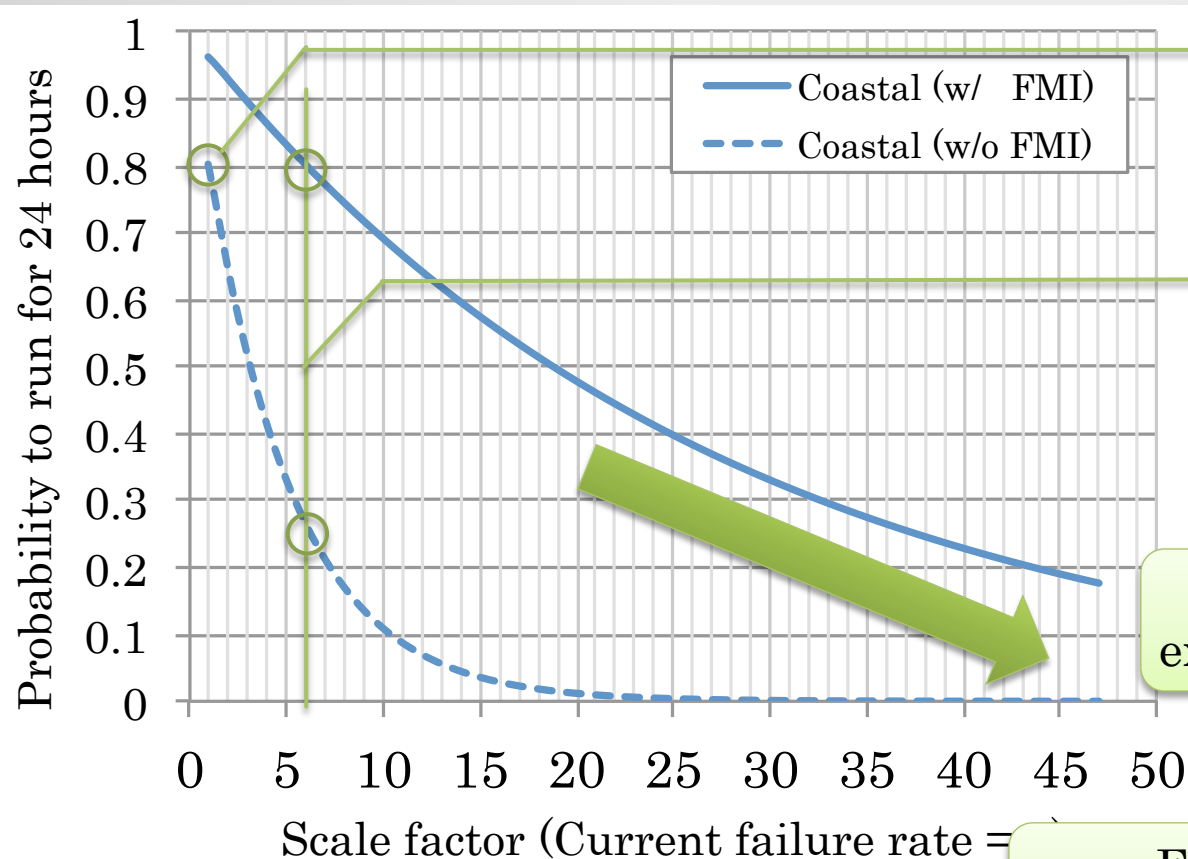
Failure analysis on Coastal cluster

	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

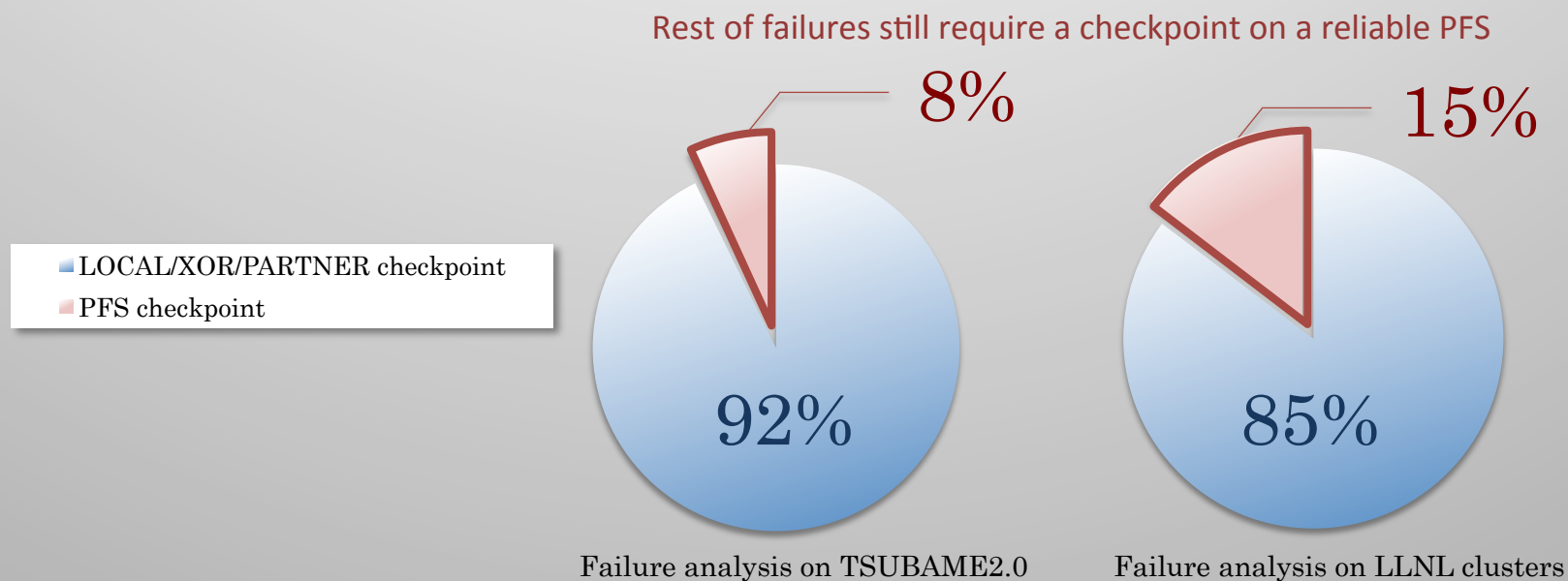
FMI execution: 80%  
non-FMI execution: 25%

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

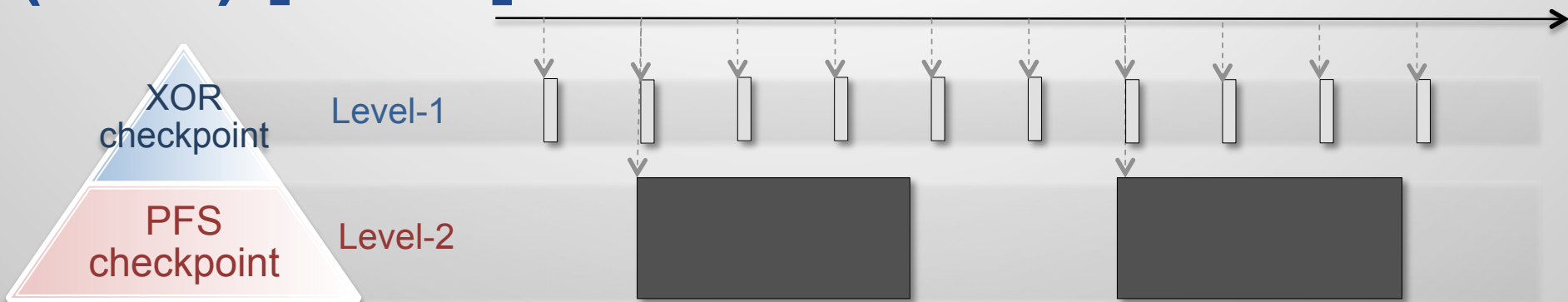
Future FMI will support  
async. multi-level checkpoint/restart

# Single node failure is common

- 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



# Asynchronous multi-level checkpointing (MLC) [SC12]



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
  - XOR 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

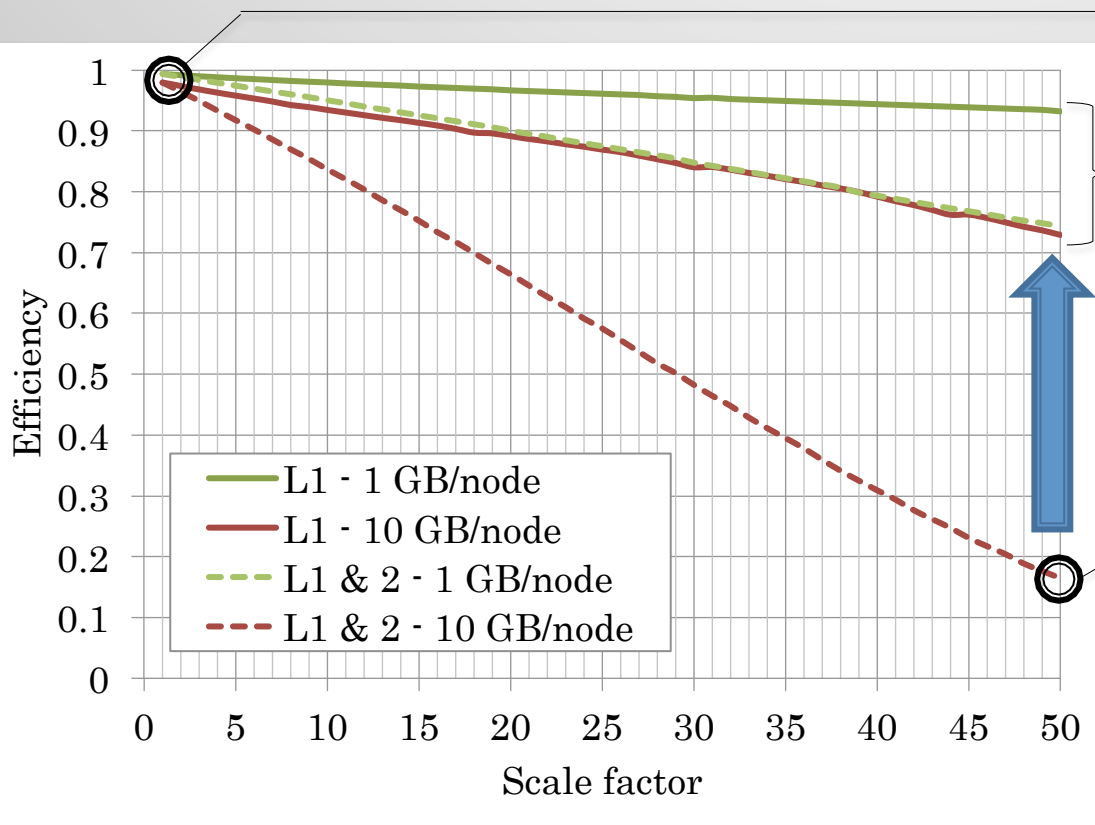
	MTBF	Failure rate
L1 failure	130 hours	$2.13^{-6}$
L2 failure	650 hours	$4.27^{-7}$

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



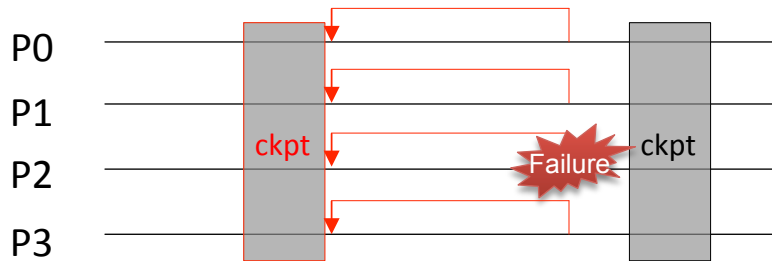
High efficiency with current failure rate

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

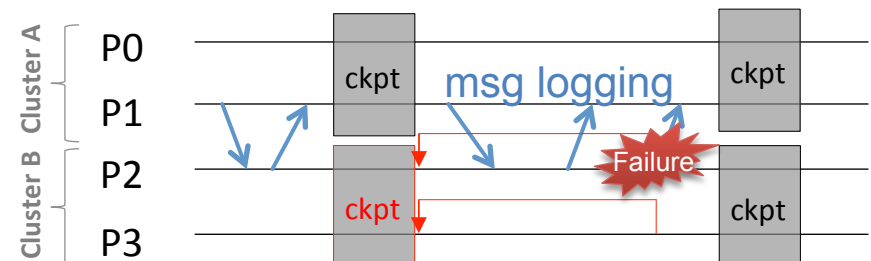
If both L1 & L2 failure rate increase, and checkpoint size is large, efficiency drops rapidly

# Uncoordinated C/R + MLC

Coordinated C/R

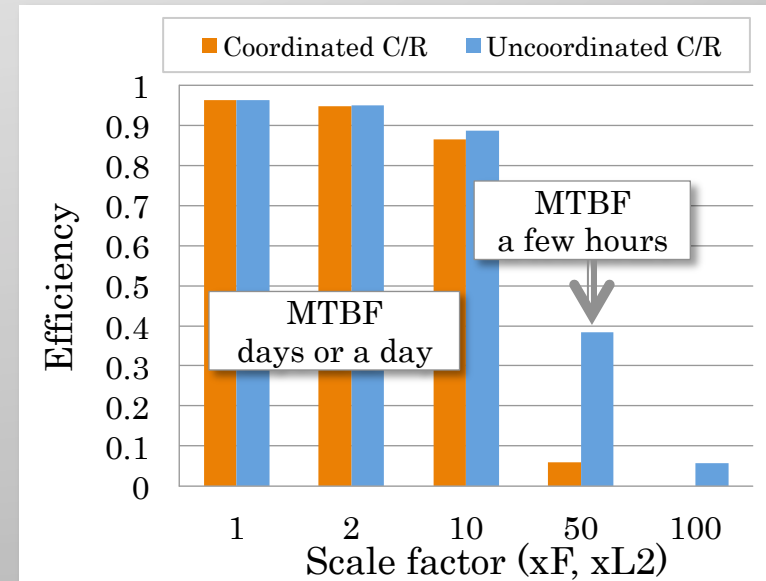


Uncoordinated C/R



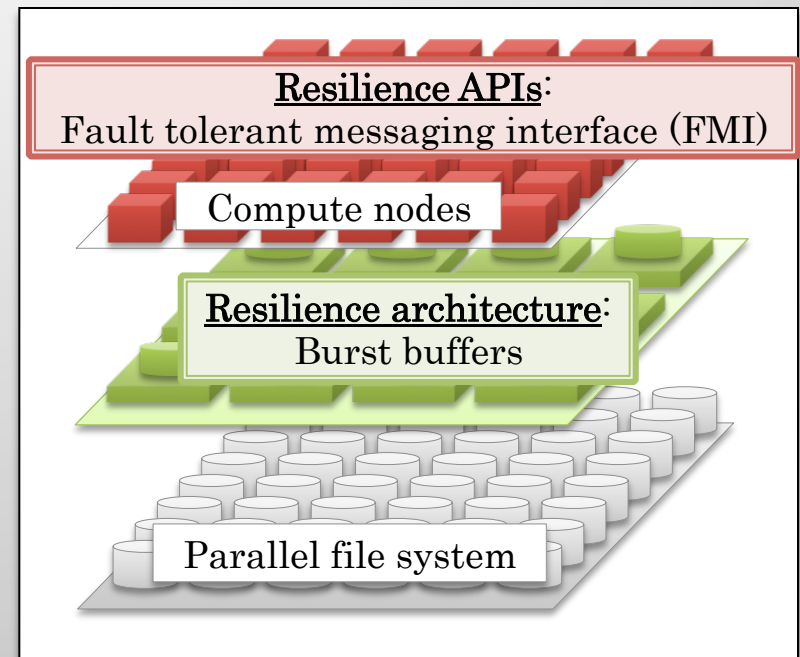
- Coordinated C/R
  - All processes globally synchronize before taking checkpoints and restart on a failure
  - Restart overhead
- Uncoordinated C/R
  - Create clusters, and log messages exchanged between clusters
  - Message logging overhead is incurred, but rolling-back only a cluster can restart the execution on a failure

⇒ MLC + Uncoordinated C/R (Software-level) approaches may be limited at extreme scale



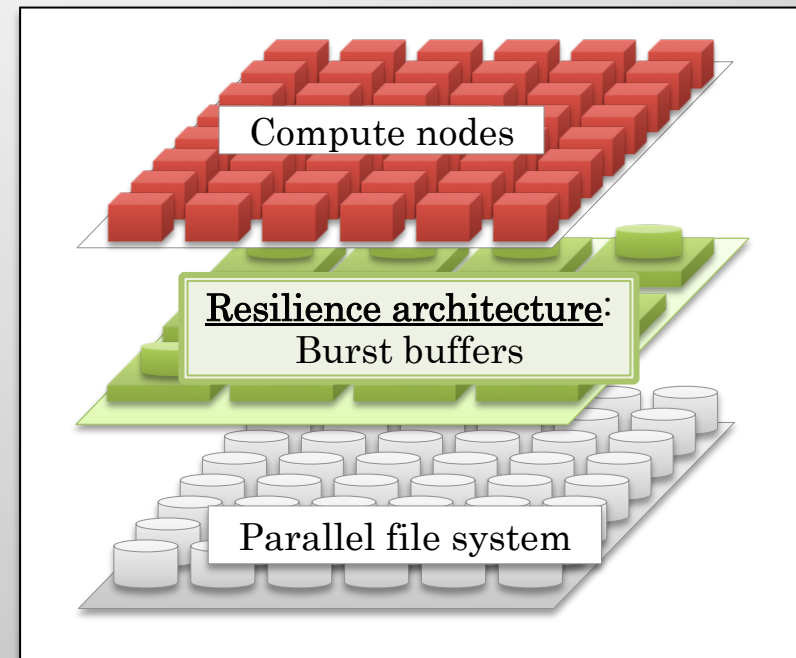
# Resilience APIs, Architecture and the model

- Resilience APIs
  - In near future, applications must have capabilities of handling failures as usual events
  - ⇒ Fault tolerant messaging interface (FMI) [IPDPS2014]
- Resilience architecture and model
  - Software level approaches are not enough
  - ⇒ Architecture using *Burst buffer* [CCGrid2014]



# Burst buffer storage architecture

- Burst buffer
  - A new tier in storage hierarchies
  - Absorb bursty I/O requests from applications
  - Fill performance gap between node-local storage and PFSs in both latency and bandwidth
- If you write checkpoints to burst buffers,
  - Faster checkpoint/restart time than PFS
  - More reliable than storing on compute nodes



# Checkpoint/Restart (Software-Lv.)

- Idea of Checkpoint/Restart

- Checkpoint

- Periodically save snapshots of an application state to PFS

- Restart

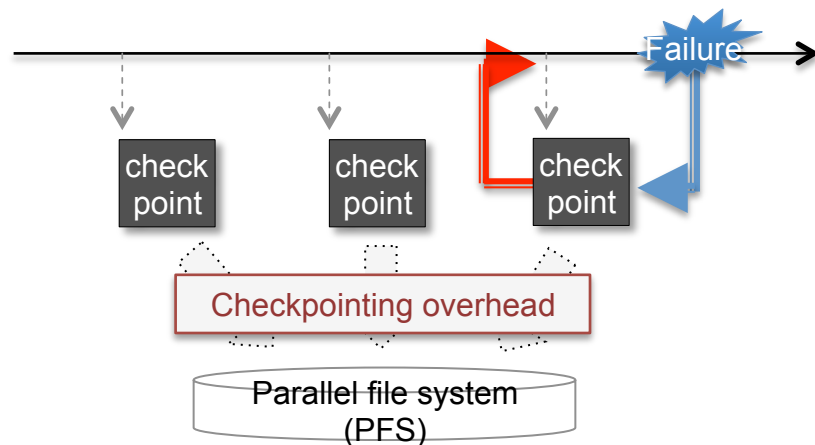
- On a failure, restart the execution from the latest checkpoint

- Improved Checkpoint/Restart

- Multi-level checkpointing [1]
- Asynchronous checkpointing [2]
- In-memory diskless checkpointing [3]

- We found that software-level approaches may be limited in increasing resiliency at extreme scale

Checkpoint/Restart



[1] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System (SC 10)

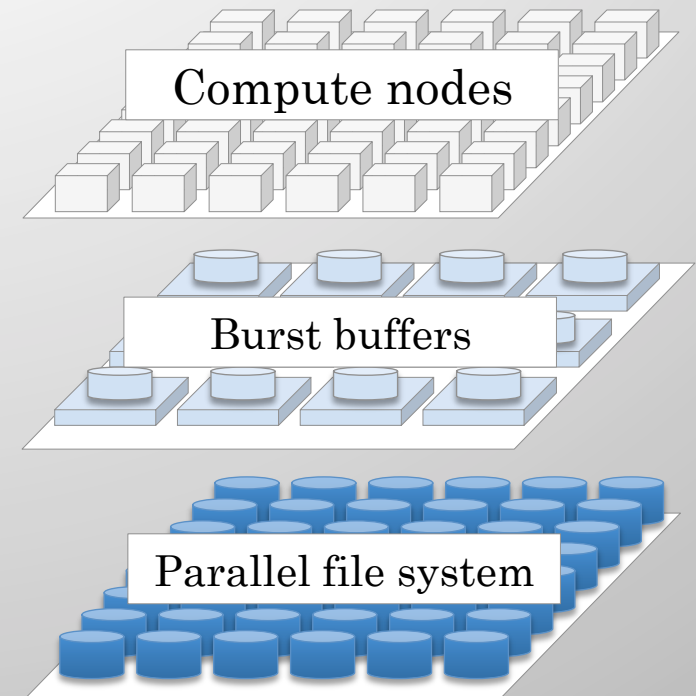
[2] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama and Satoshi Matsuoka, "Design and Modeling of a Non-blocking Checkpointing System", SC12

[3] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama and Satoshi Matsuoka, "FMI: Fault Tolerant Messaging Interface for Fast and Transparent Recovery", IPDPS2014

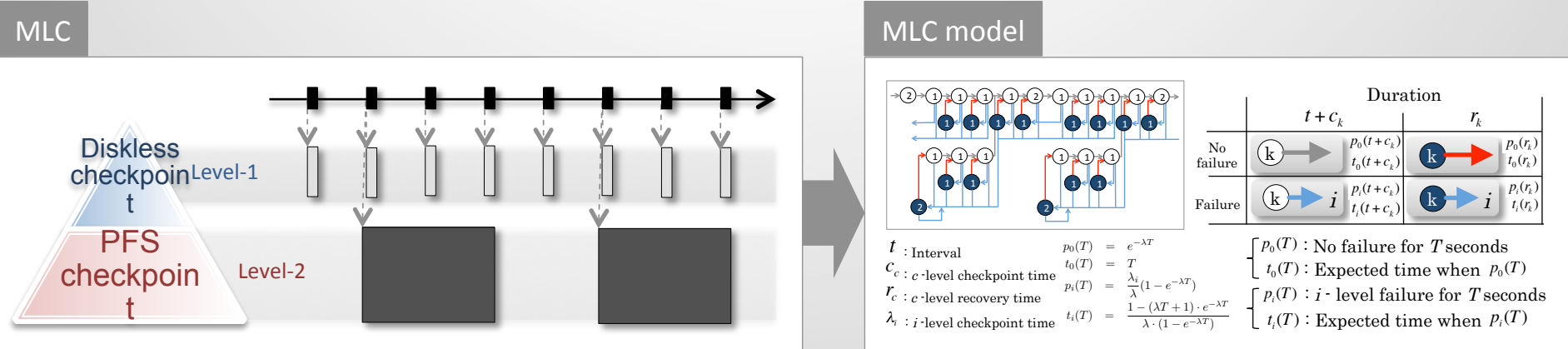
# Storage architectures

- We consider architecture-level approaches
- Burst buffer
  - A new tier in storage hierarchies
  - Absorb bursty I/O requests from applications
  - Fill performance gap between node-local storage and PFSs in both latency and bandwidth
- If you write checkpoints to burst buffers,
  - Faster checkpoint/restart time than PFS
  - More reliable than storing on compute nodes

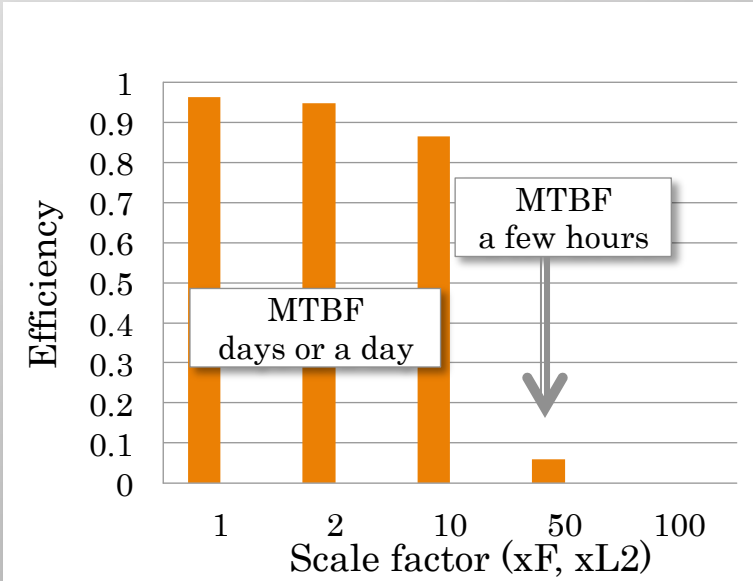
- However,...
  - Adding burst buffer nodes may increase total system size, and failure rates accordingly
    - It's not clear if burst buffers improve overall system efficiency
  - Because burst buffers also connect to networks, the burst buffers may still be a bottleneck



# Multi-level Checkpoint/Restart (MLC/R) [SC10, 12]



- MLC hierarchically use storage levels
  - Diskless checkpoint: Frequent for one node for a few node failure
  - PFS checkpoint: Less frequent and asynchronous for multi-node failure
- Our evaluation showed system efficiency drops to less than 10% when MTBF is a few hours



[1] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System (SC 10)

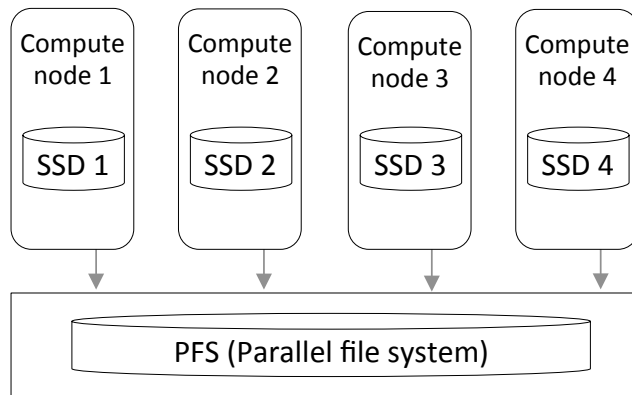
[2] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama and Satoshi Matsuoka, "Design and Modeling of a Non-blocking Checkpointing System", SC12



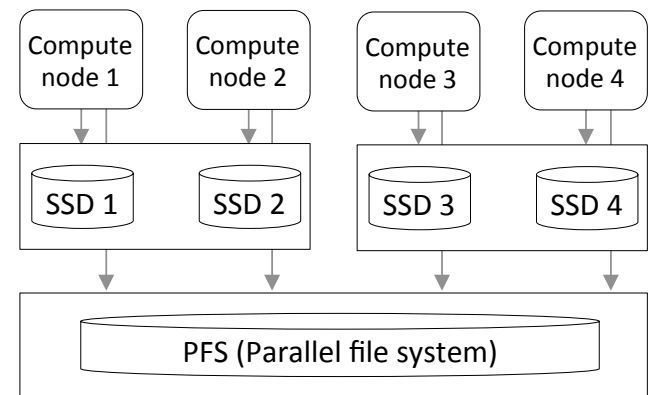
# Storage designs

- Addition to the software-level approaches, we also explore two architecture-level approaches
  - Flat buffer system:
    - Current storage system
  - Burst buffer system:
    - Separated buffer space

Flat buffer system



Burst buffer system



# Flat Buffer Systems

- Design concept

- Each compute node has its dedicated node-local storage
- Scalable with increasing number of compute nodes

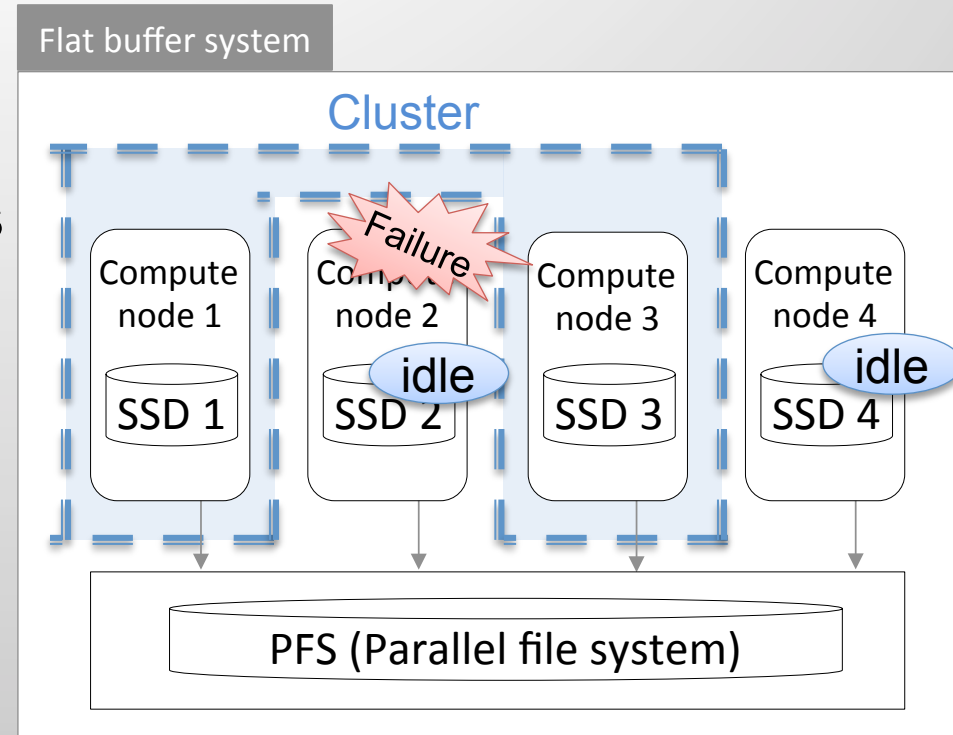
- This design has drawbacks:

1. Unreliable checkpoint storage

e.g.) If compute node 2 fails, a checkpoint on SSD 2 will be lost because SSD 2 is physically attached to the failed compute node 2

2. Inefficient utilization of storage resources on uncoordinated checkpointing

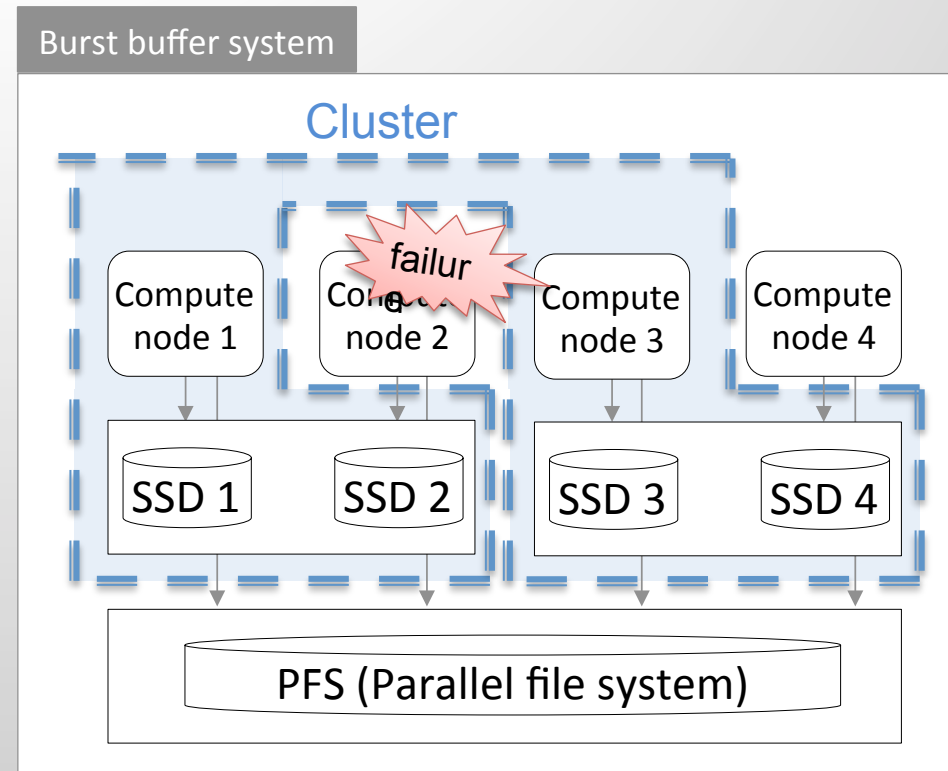
e.g.) If compute node 1 & 3 are in a same cluster, and restart from a failure, the bandwidth of SSD 2 & 4 will not be utilized



# Burst Buffer Systems

- Design concept

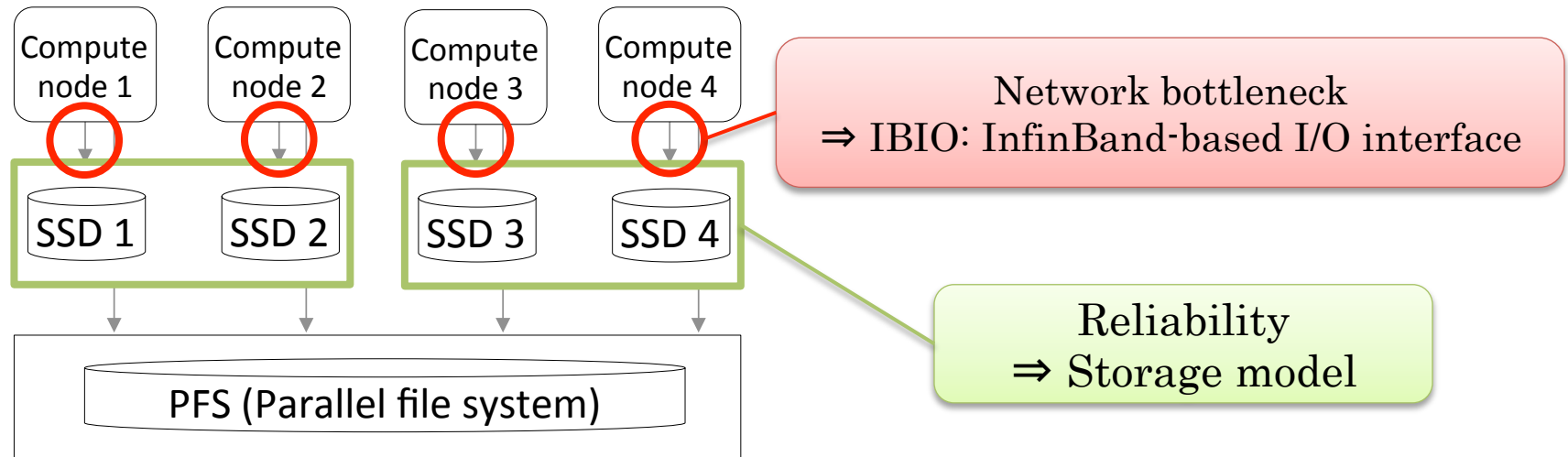
- A burst buffer is a storage space to bridge the gap in latency and bandwidth between node-local storage and the PFS
- Shared by a subset of compute nodes



- Although additional nodes are required, several advantages
  1. More Reliable because burst buffers are located on a smaller # of nodes  
e.g.) Even if compute node 2 fails, a checkpoint of compute node 2 is accessible from the other compute node 1
  2. Efficient utilization of storage resources on uncoordinated checkpointing  
e.g.) if compute node 1 and 3 are in a same cluster, and both restart from a failure, the processes can utilize all SSD bandwidth unlike a flat buffer system

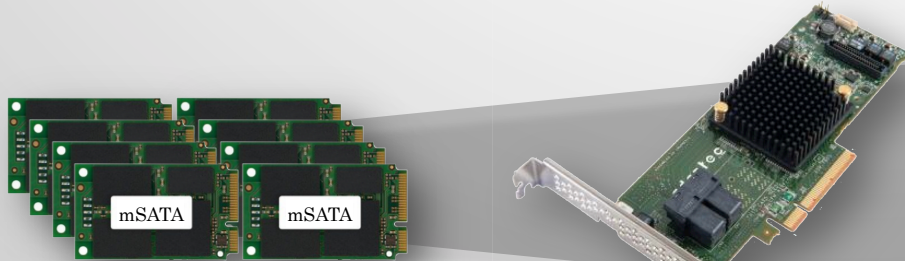
# Challenges for using burst buffers

## Challenges for using burst buffer system



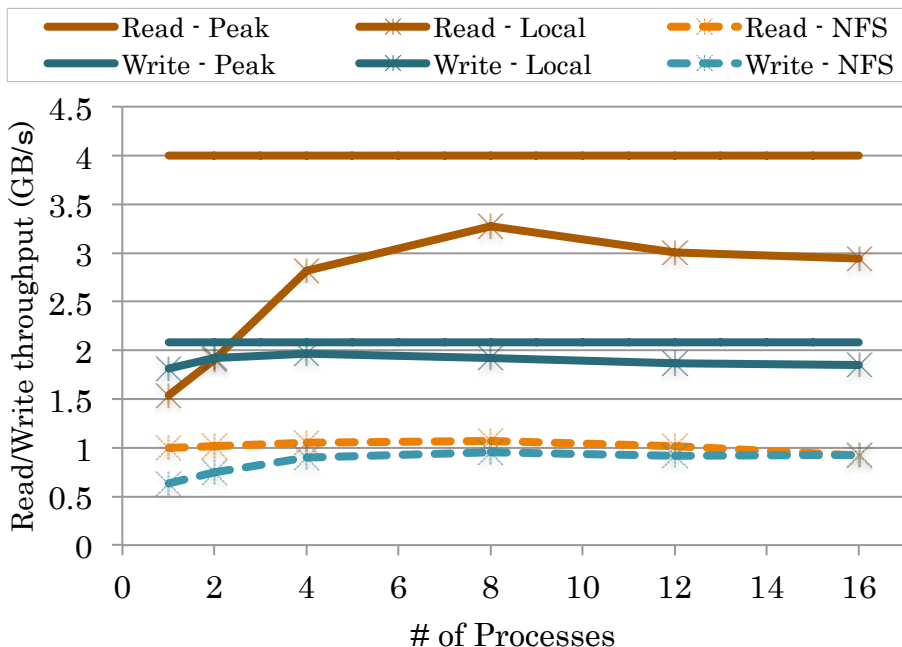
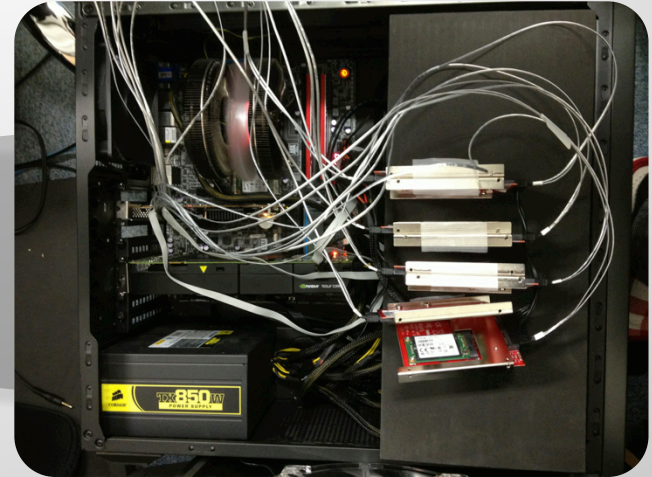
- Exploiting storage bandwidth of burst buffers
  - Burst buffers are connected to networks, networks can be bottleneck
- Analyzing reliability of systems with burst buffers
  - Adding burst buffer nodes increase total system size, and increase overall failure rate
  - System efficiency may decrease

# Burst buffer prototype multi-mSATA High I/O BW & cost



mSATA  $\times$  8  
(Read: 500MB/s, Write: 260MB/s)

Adaptec RAID  $\times$  1



## Node specification

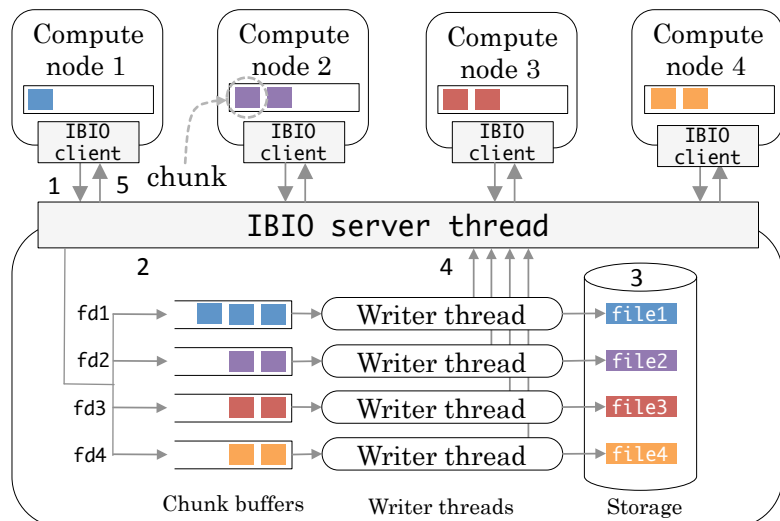
CPU	Intel Core i7-3770K CPU (3.50GHz x 4 cores)
Memory	Cetus DDR3-1600 (16GB)
M/B	GIGABYTE GA-Z77X-UD5H
SSD	Crucial m4 msata 256GB CT256M4SSD3 (Peak read: 500MB/s, Peak write: 260MB/s)
SATA converter	KOUTECH IO-ASS110 mSATA to 2.5' SATA Device Converter with Metal Fram
RAID Card	Adaptec RAID 7805Q ASR-7805Q Single

Interconnect :Mellanox FDR HCA (Model No.: MCX354A-FCBT)

# IBIO: InfiniBand-based I/O interface

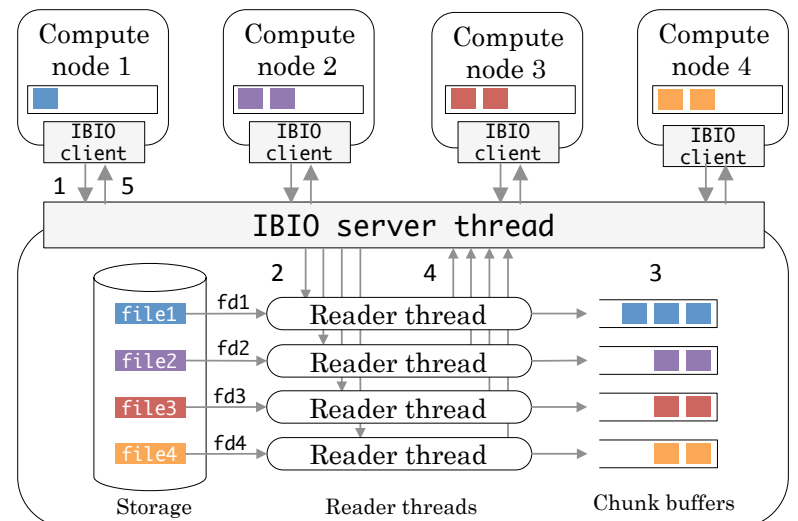
- Provide POSIX I/O-like interfaces
  - open, read, write and close
  - Client can open any files on any servers
    - `open("hostname:/path/to/file", mode)`
- IBIO use ibverbs for communication between clients and servers
  - Exploit network bandwidth of infiniband

## IBIO write



IBIO write: four IBIO clients and one IBIO server

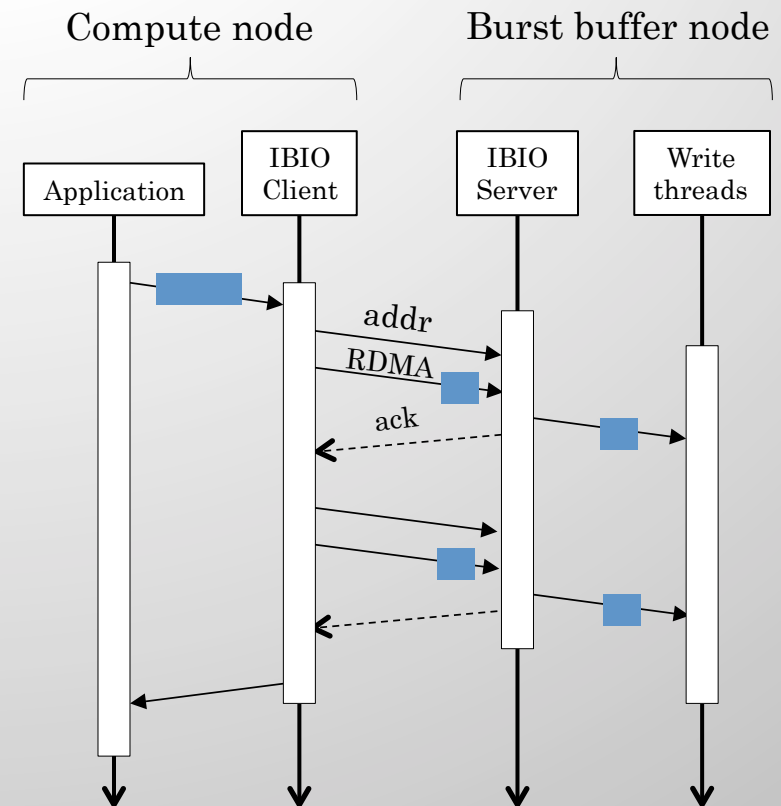
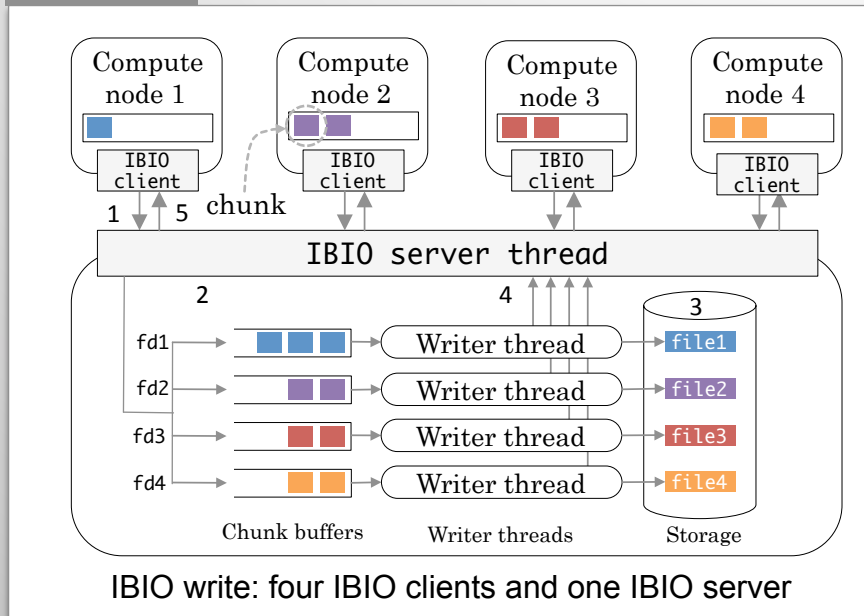
## IBIO read



IBIO read: four IBIO clients and one IBIO server

# IBIO write/read

## IBIO write

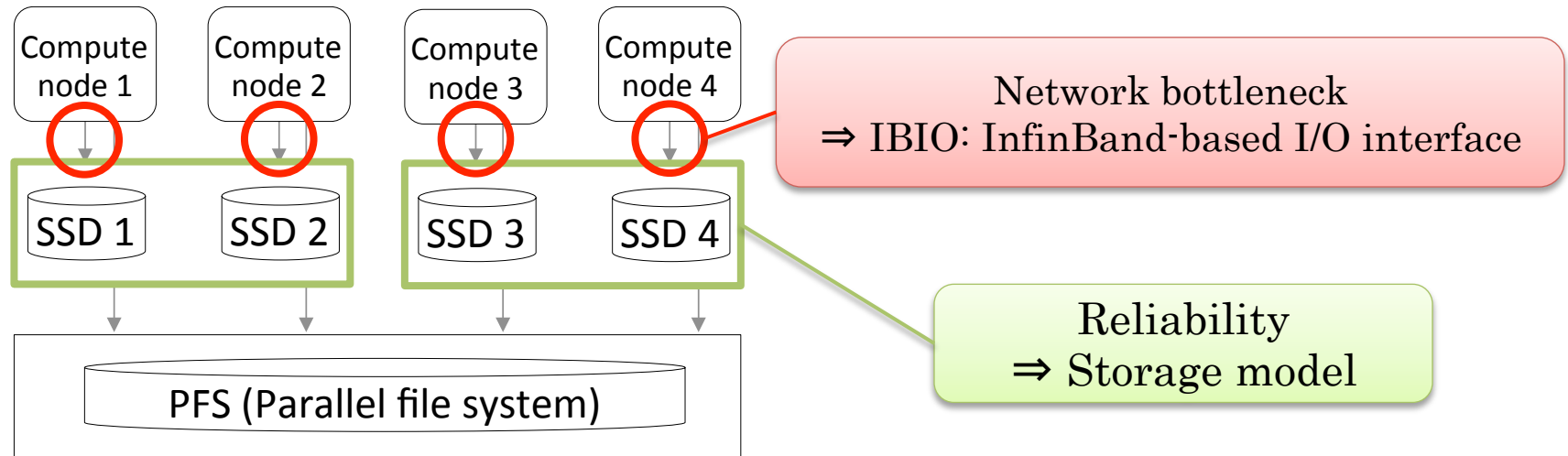


- IBIO write
  1. Application call IBIO client function with data to write
  2. IBIO client divides the data into chunks, then send the address to IBIO server for RDMA
  3. IBIO server issues RDMA read to the address, and reply ack
  4. Continues until all chunks are sent, and return to application
  5. Writer threads asynchronously write received data to storage
- IBIO read
  - Reads chunks by reader threads and send to clients in the same way as IBIO write by using RDMA



# Challenges for using burst buffers

## Challenges for using burst buffer system



- Exploiting storage bandwidth of burst buffers
  - Burst buffers are connected to networks, networks can be bottleneck
- Analyzing reliability of systems with burst buffers
  - Adding burst buffer nodes increase total system size
  - System efficiency may decrease due to Increased overall failure by added burst buffers

# Modeling overview

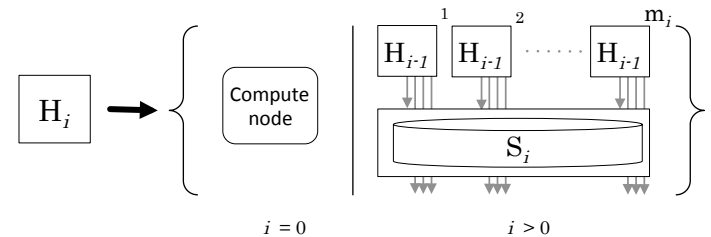
- To find out the best checkpoint/restart strategy for systems with burst buffers, we model checkpointing strategies

## C/R strategy model

$$O_i = \begin{cases} C_i + E_i & (\text{Sync.}) \\ I_i & (\text{Async.}) \end{cases} \quad L_i = C_i + E_i$$

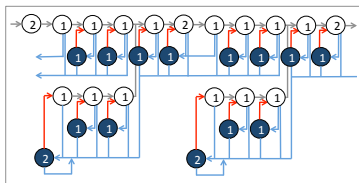
$$C_i \text{ or } R_i = \frac{\langle \text{C/R data size / node} \rangle \times \langle \# \text{ of C/R nodes per } S_i^* \rangle}{\langle \text{write perf. ( } w_i \text{) } \rangle \text{ or } \langle \text{read perf. ( } r_i \text{) } \rangle}$$

## Recursive structured storage model



Storage Model:  $H_N \{m_1, m_2, \dots, m_N\}$

## MLC model [2]



$t$  : Interval  
 $c_c$  : c-level checkpoint time  
 $r_c$  : c-level recovery time  
 $\lambda_i$  : i-level checkpoint time

$$\begin{aligned}
 p_0(T) &= e^{-\lambda T} \\
 t_0(T) &= T \\
 p_i(T) &= \frac{\lambda_i (1 - e^{-\lambda T})}{\lambda} \\
 t_i(T) &= \frac{1 - (\lambda T + 1) \cdot e^{-\lambda T}}{\lambda \cdot (1 - e^{-\lambda T})}
 \end{aligned}$$

	Duration	
	$t + c_k$	$r_k$
No failure		
Failure		

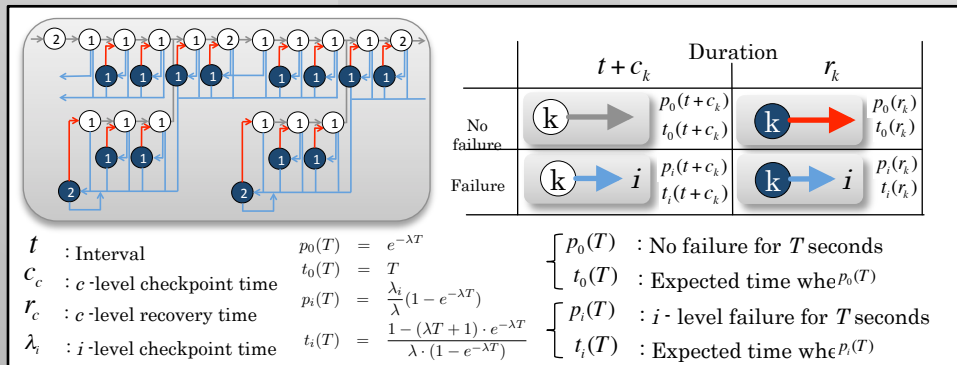
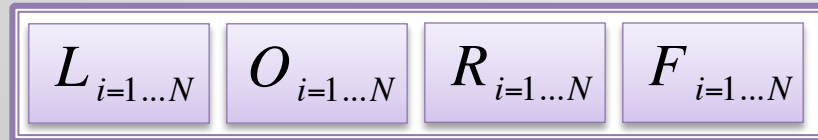
$p_0(T)$  : No failure for  $T$  seconds  
 $t_0(T)$  : Expected time when  $p_0(T)$   
 $p_i(T)$  : i-level failure for  $T$  seconds  
 $t_i(T)$  : Expected time when  $p_i(T)$

## Efficiency

Fraction of time an application spends only in useful computation

# Multi-level Asynchronous C/R Model [SC12]

- Optimize checkpoint intervals and compute checkpoint/restart “*Efficiency*” using Markov model
  - Vertex: Compute state OR Checkpointing state OR Recovery state
  - Edge: Completion of each state



- Input: Each level of
  - $L_i$ : Checkpoint Latency
  - $O_i$ : Checkpoint overhead
  - $R_i$ : Restart time
  - $F_i$ : Failure rate
- Output: “*Efficiency*”

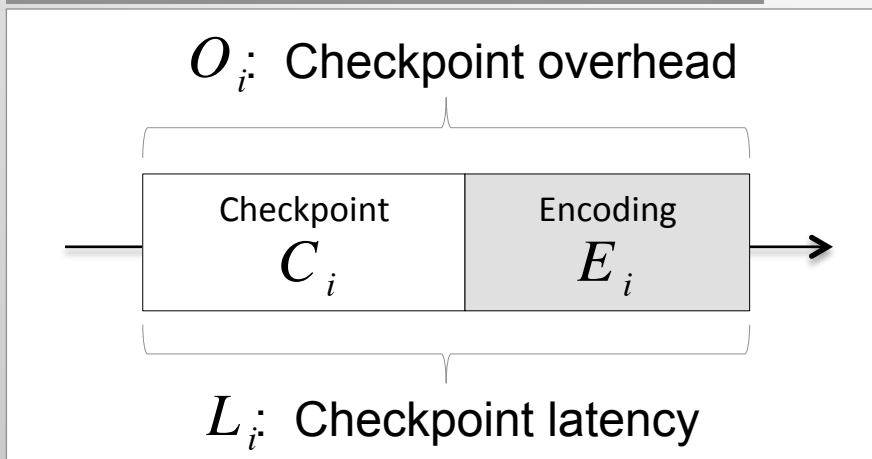
*Efficiency*

Efficiency

Fraction of time an application spends only in computation in optimal checkpoint interval

# Modeling of C/R Strategies

## Synchronous checkpointing (Diskless C/R)

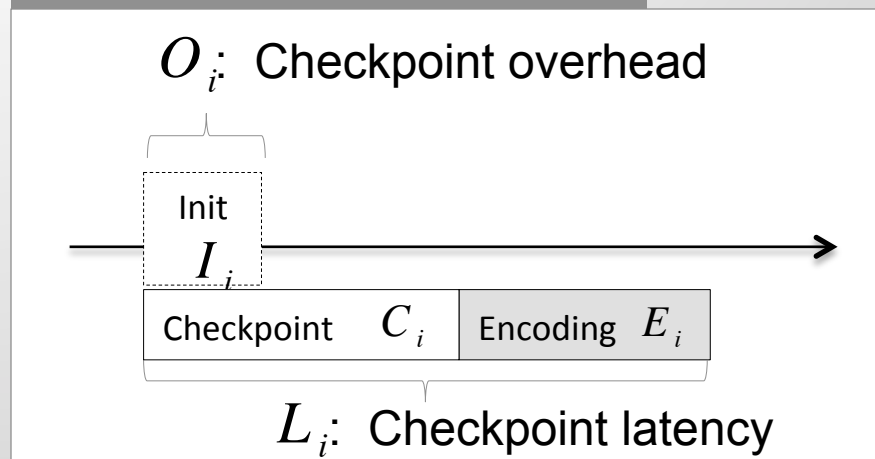


- $L_i$ : Checkpoint Latency
  - Time to complete a checkpoint ( $C_i$ ) and encoding ( $E_i$ )

$$L_i = C_i + E_i$$

- $C_i$  &  $R_i$ : Checkpoint/Restart time

## Asynchronous checkpointing (PFS)



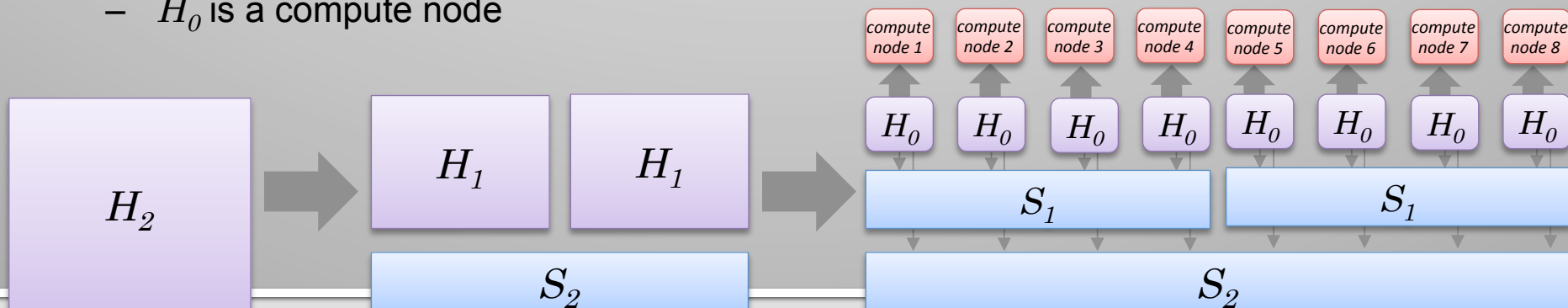
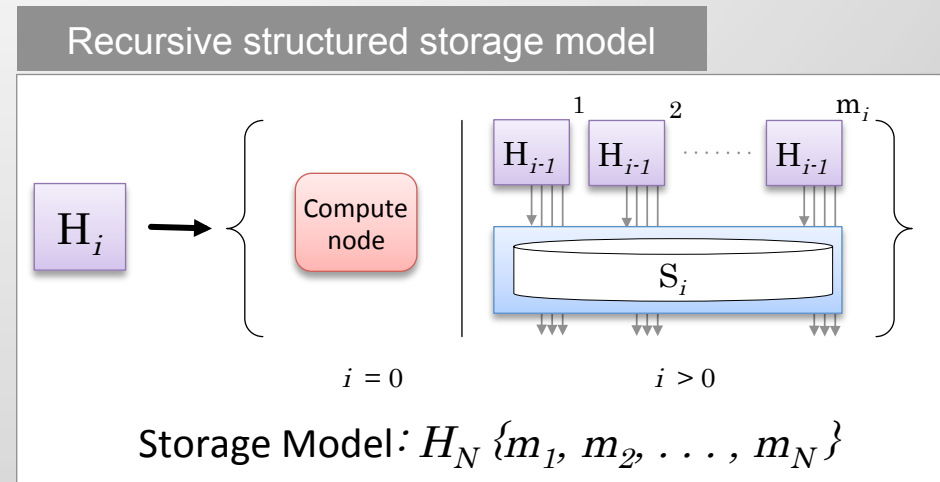
- $O_i$ : Checkpoint overhead
  - The increased execution time of an application

$$O_i = \begin{cases} C_i + E_i & \text{(Sync.)} \\ I_i & \text{(Async.)} \end{cases}$$

$$C_i \text{ or } R_i = \frac{\langle \text{C/R data size / node} \rangle \times \langle \# \text{ of C/R nodes per } S_i^* \rangle}{\langle \text{write perf. ( } w_i \text{ ) } \rangle \text{ or } \langle \text{read perf. ( } r_i \text{ ) } \rangle}$$

# Recursive Structured Storage Model

- Generalization of storage architectures with "context-free grammar"
  - A tier  $i$  hierarchical entity ( $H_i$ ), has a storage ( $S_i$ ) shared by ( $m_i$ ) upper hierarchical entities ( $H_{i-1}$ )
  - $H_{i=0}$  is a compute node
  - $H_N \{m_1, m_2, \dots, m_N\}$
- e.g. )  $H_2 \{4, 2\}$ 
  - $H_2$  has an  $S_2$  shared by 2  $H_1$
  - $H_1$  has an  $S_1$  shared by 4  $H_0$
  - $H_0$  is a compute node



# Recursive Structured Storage Model (cont'd)

- The number of nodes accessing to  $S_i$

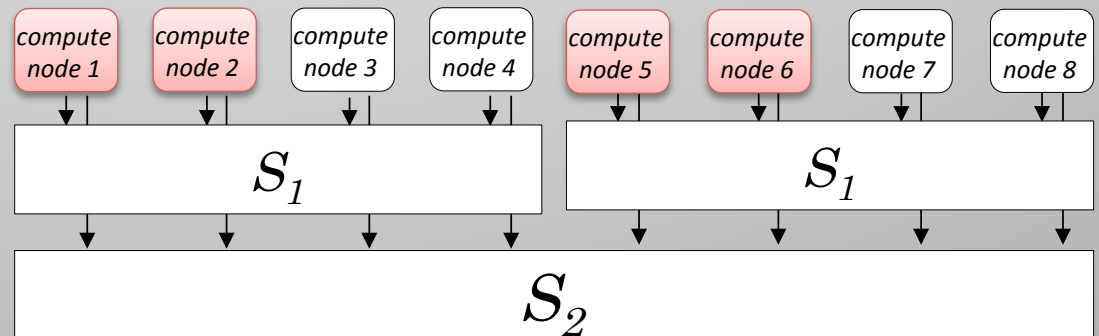
$$\langle \# \text{ of C/R nodes per } S_i \rangle = \frac{K}{\langle \# \text{ of } S_i \rangle}$$

$$\langle \# \text{ of } S_i \rangle = \begin{cases} II^N_{k=i+1} m_k & (i < N) \\ 1 & (i = N) \end{cases}$$

$K$ : C/R cluster size

- e.g. )  $K = 4$

- # of C/R nodes per  $S_1$ 
  - $4/2 = 2$  nodes
- # of C/R nodes per  $S_2$ 
  - $4/1 = 4$  nodes



# Evaluation

- IBIO performance
- Simulation

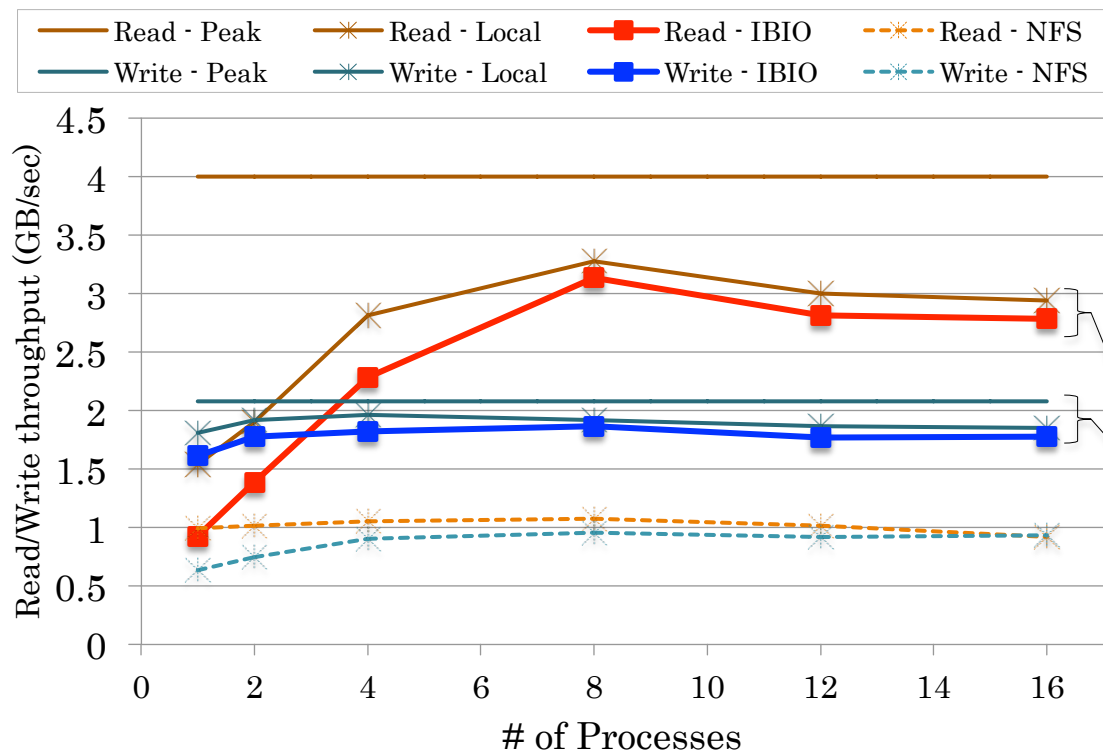


# Sequential IBIO read/write performance

- Set chunk size to 64MB for both IBIO and NFS to maximize the throughputs

mSATA × 8  
(Read: 500MB/s,  
Write: 260MB/s)

Adaptec RAID × 1



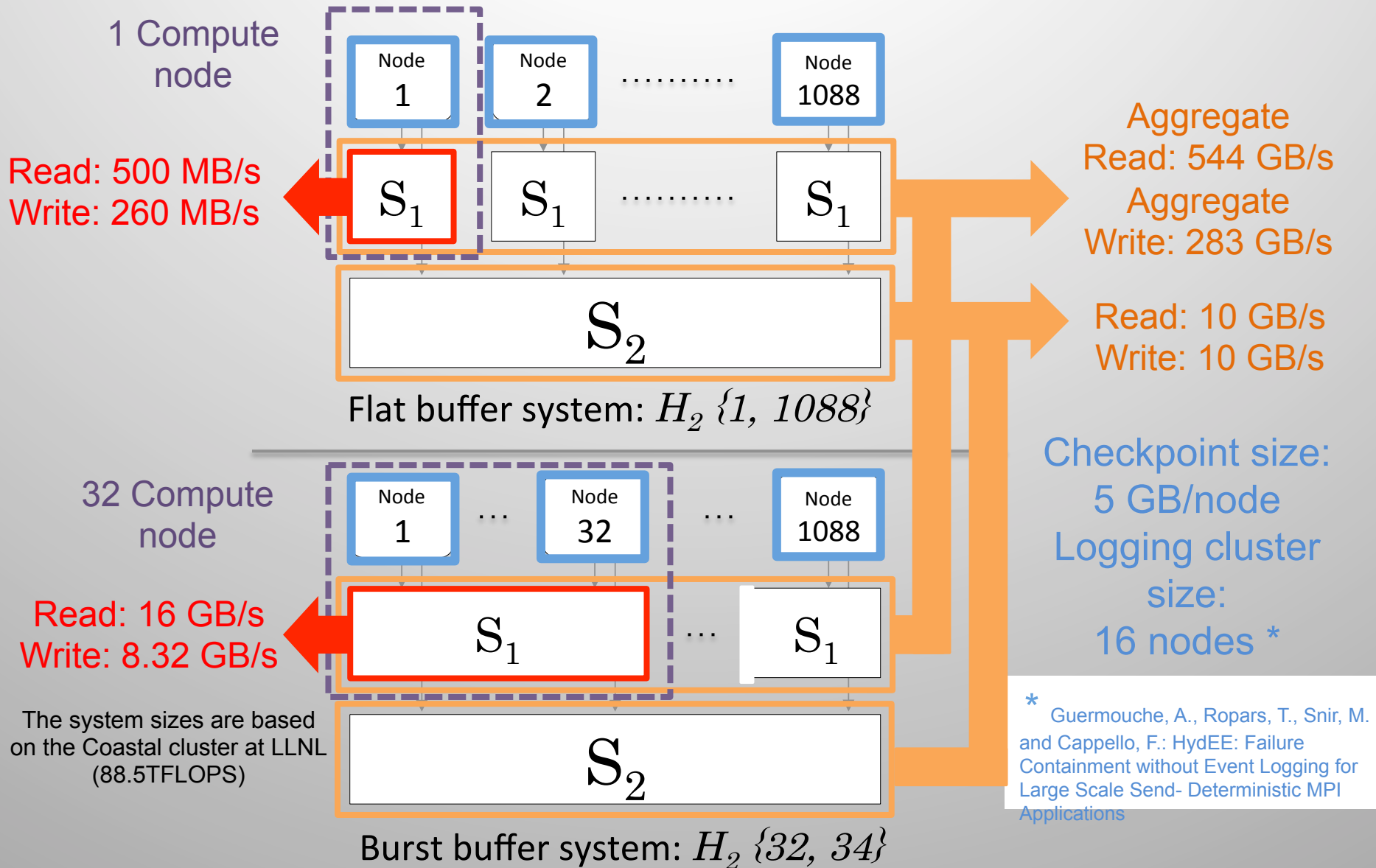
## Node specification

CPU	Intel Core i7-3770K CPU (3.50GHz x 4 cores)
Memory	Cetus DDR3-1600 (16GB)
M/B	GIGABYTE GA-Z77X-UD5H
SSD	Crucial m4 msata 256GB CT256M4SSD3 (Peak read: 500MB/s, Peak write: 260MB/s)
SATA converter	KOUTECH IO-ASS110 mSATA to 2.5' SATA Device Converter with Metal Fram
RAID Card	Adaptec RAID 7805Q ASR-7805Q Single

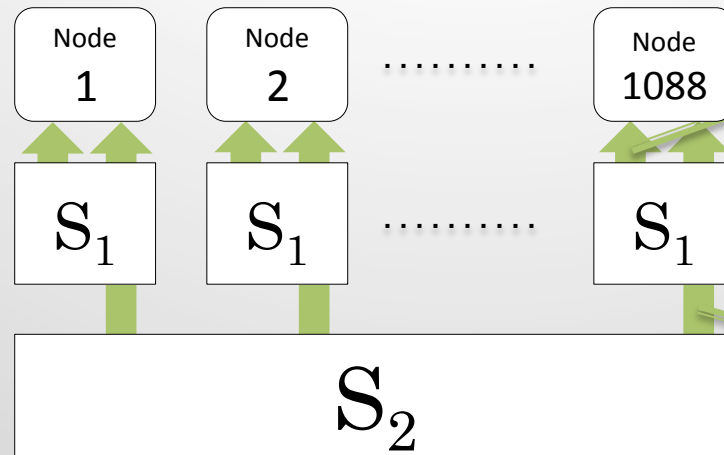
Interconnect :Mellanox FDR HCA (Model No.: MCX354A-FCBT)

IBIO achieve the same remote read/write performance as the local read/write performance by using RDMA

# Experimental setup



# Experimental setup



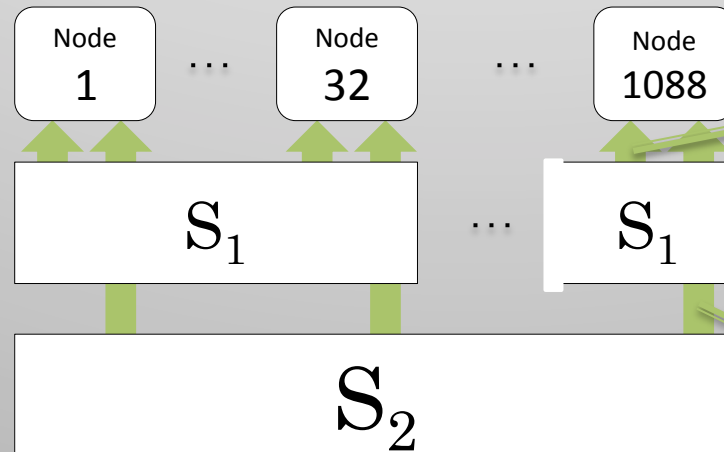
Flat buffer system:  $H_2 \{1, 1088\}$

Level 1  
(XOR checkpoint required)  
 $2.14 \times 10^{-6}$

Level 2  
(PFS checkpoint required)  
 $4.28 \times 10^{-7}$

Estimated failure rates are based on failure analysis on the Coastal cluster at LLNL (88.5TFLOPS) [1]

[1] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System (SC 10)



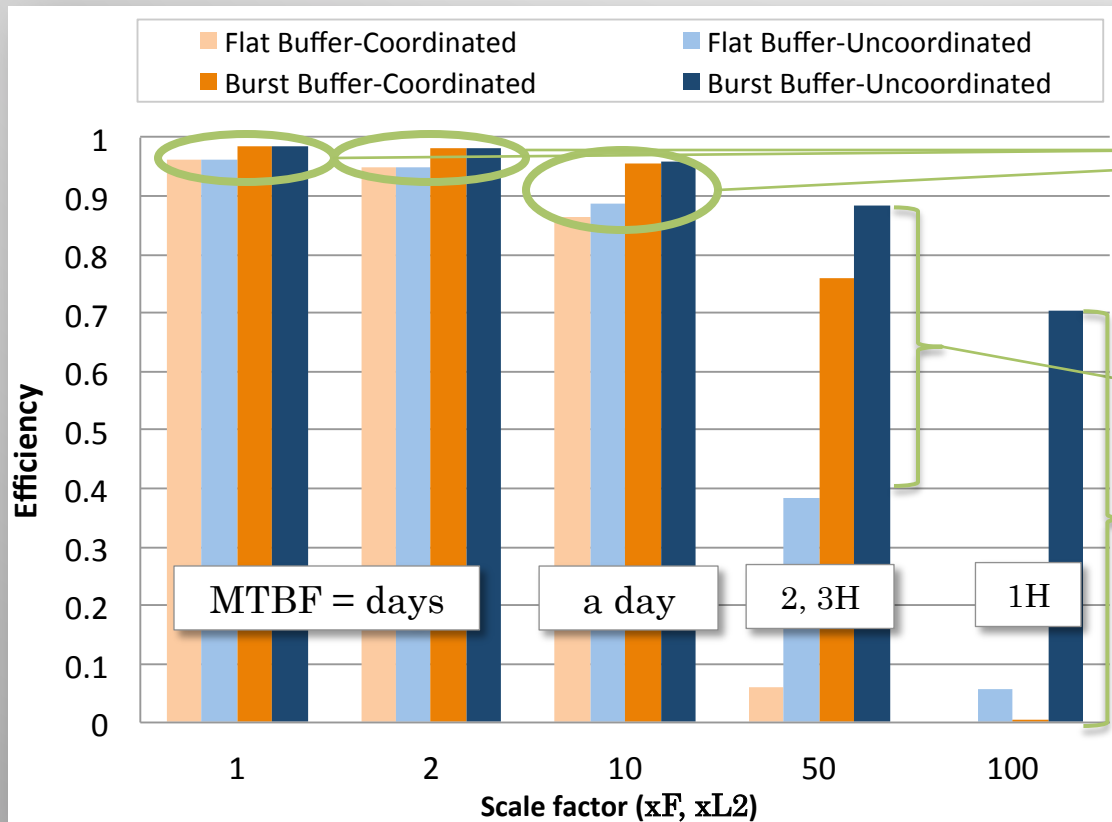
Burst buffer system:  $H_2 \{32, 34\}$

Level 1  
(XOR checkpoint required)  
 $2.63 \times 10^{-6}$

Level 2  
(PFS checkpoint required)  
 $1.33 \times 10^{-8}$

# Efficiency with Increasing Failure Rates and Checkpoint Costs

- Assuming there is no message logging overhead



In days or a day of MTBF, there is no big efficiency differences

In a few hours of MTBF, with burst buffers, systems can still achieve high efficiency

Even in a hour of MTBF, with uncoordinated, systems can still achieve 70% efficiency

⇒ Partial restart accelerate recovery time from burst buffers and PFS checkpoint

# Allowable Message Logging overhead

Message logging overhead allowed in uncoordinated checkpointing to achieve a higher efficiency than coordinated checkpointing

Flat buffer		Burst buffer	
scale factor	Allowable message logging overhead	scale factor	Allowable message logging overhead
1	0.0232%	1	0.00435%
2	0.0929%	2	0.0175%
10	2.45%	10	0.468%
50	84.5%	50	42.0%
100	$\approx 100$	100	99.9%

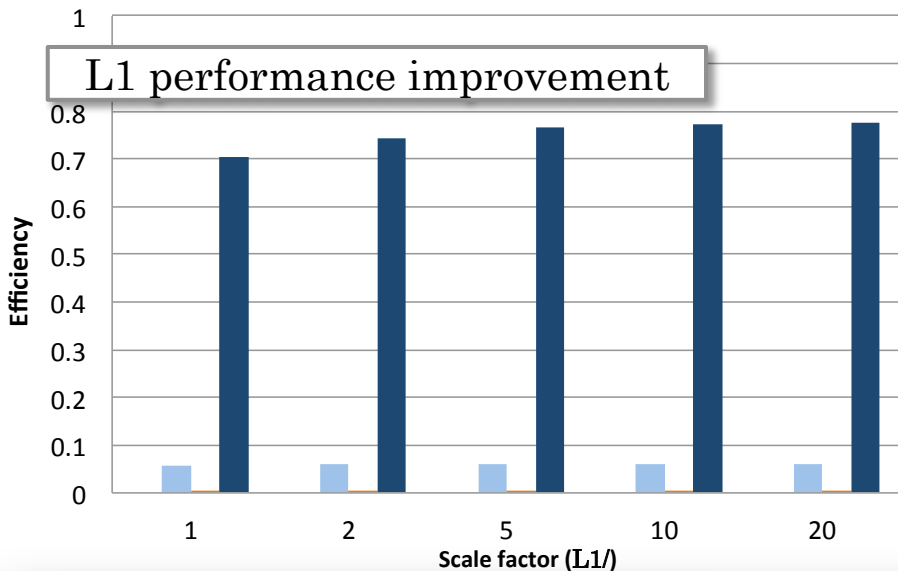
- Logging overhead must be relatively small, less than a few percent in days or a day of MTBF
  - In a few hours or a hour, very high message logging overheads are tolerated

⇒ Uncoordinated checkpointing can be more effective on future systems

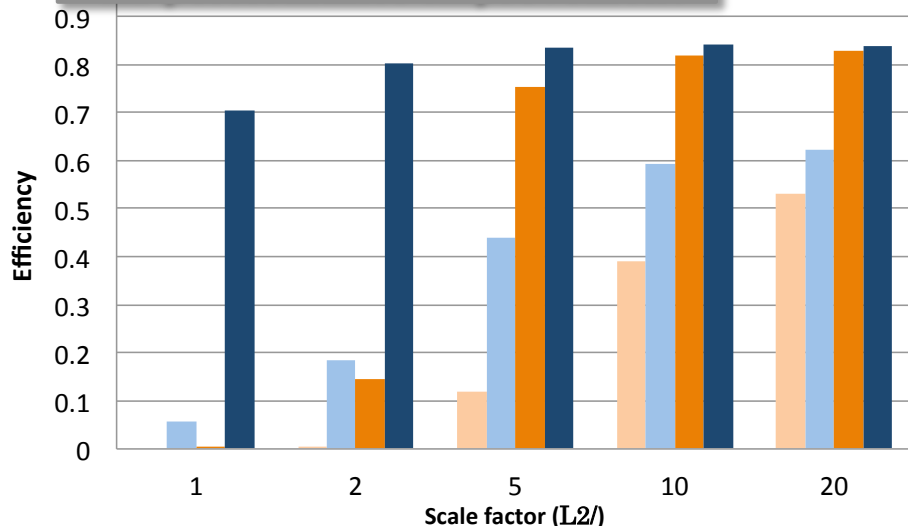
# Effect of Improving Storage Performance

Flat Buffer-Coordinated      Flat Buffer-Uncoordinated  
Burst Buffer-Coordinated      Burst Buffer-Uncoordinated

L1 performance improvement



L2 performance improvement



To see which storage impact to efficiency, we increase performance of level-1 and level-2 storage while keeping MTBF a hour

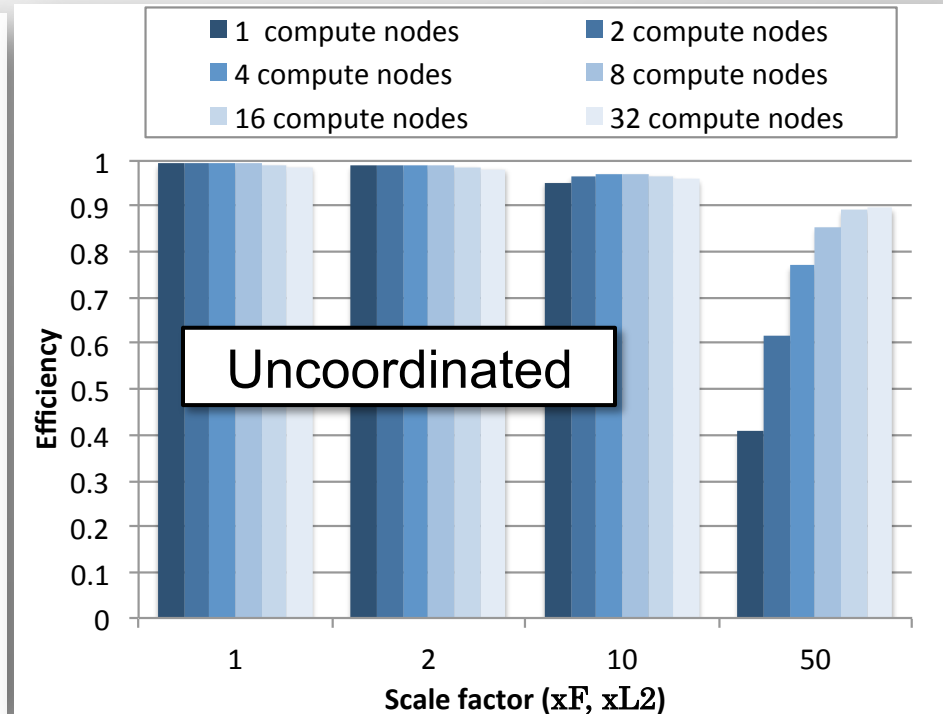
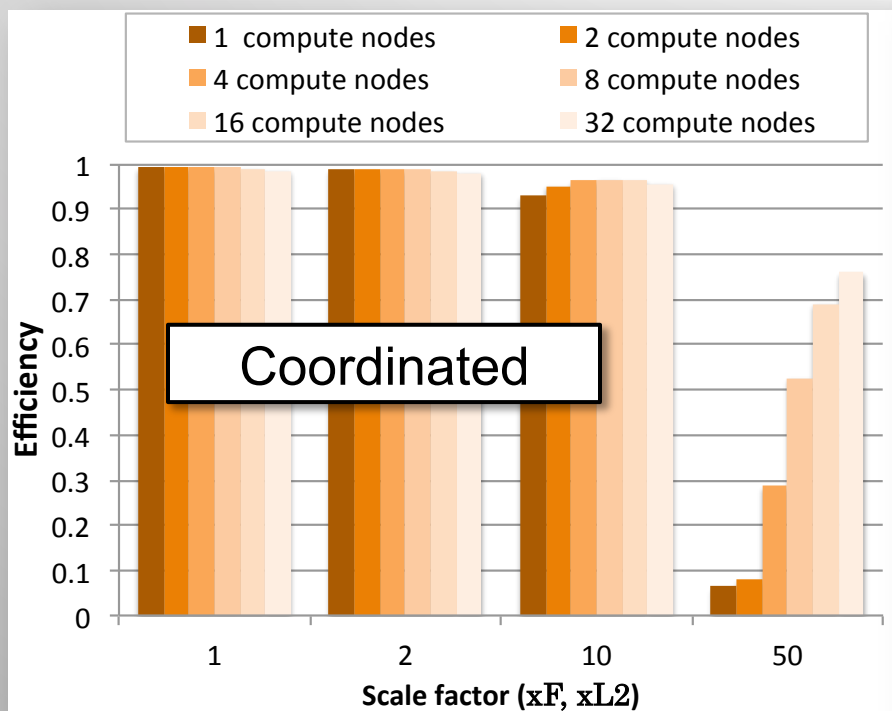
Improvement of level-1 storage performance does not impact efficiency for both flat buffer and burst buffer systems

Increasing the performance of the PFS does impact system efficiency

L2 C/R overhead is a major cause of degrading efficiency, so reducing level-2 failure rate and improving level-2 C/R is critical on future systems

# Ratio of Compute nodes to Burst Buffer nodes

Another thing to consider when building a burst buffer system is the ratio of compute nodes to burst buffer nodes



- The ratio is not important matter when MTBF is from a day to days
- When MTBF is a few hours, a larger number of burst buffer nodes decreases efficiency  
⇒ Adding additional burst buffer nodes increases the failure rate which degrades system efficiency more than the efficiency gained by the increased bandwidth



# Towards resilient extreme scale computing

## 1. Burst buffers

- Burst buffers are beneficial for C/R at extreme scale

## 2. Uncoordinated C/R

- When MTBF is days or a day, uncoordinated C/R may not be effective
- If MTBF is a few hours or less, will be effective

## 3. Level-2 failure, and Level-2 performance

- Reducing Level-2 failure and increasing Level-2 performance are critical to improve overall system efficiency

## 4. Fewer number of burst buffers

- Adding additional burst buffer nodes increases the failure rate
- May degrades system efficiency more than the efficiency gained by the increased bandwidth
- We need to be careful a trade-off between I/O performance and reliability of burst buffers

# Conclusion

- Fault tolerance is critical at extreme scale
  - Both C/R strategy and storage design are important
- We developed IBIO to maximize remote access to burst buffers, and modeled C/R strategy and storage design
- We listed up key factors to build resilient systems based on our evaluations
- We expect our findings can benefit system designers to create efficient and cost-effective systems

# NEEDS FOR REDUCTION IN CHECKPOINT TIME

## Checkpoint/Restart

- Store the data of memory in the disk
- High I/O cost

### On TSUBAME2.5

Memory capacity : about 100TB  
I/O throughput : about 20GB/s



Checkpoint time : **about 80min**

## Reduce MTBF(Mean Time Between Failure) by expansion in scale of HPC systems

- MTBF is **over 30min** by trial calculation On a exascale computer [※1]



If **MTBF < Checkpoint time**

Application may not be able to run !



Needs for reduction in checkpoint time !

There are methods of reduction in checkpoint cost, incremental checkpoint etc., but we **compress** checkpoint

※1 : Peter Kogge, Editor & Study Lead (2008)

# LOSSLESS AND LOSSY COMPRESSION

## Features of lossless

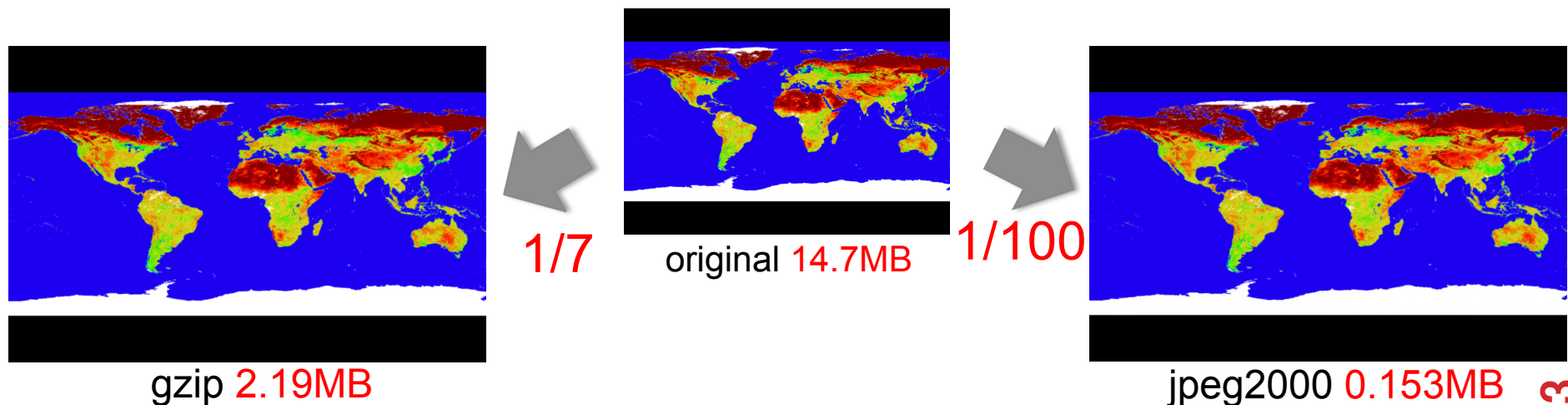
- Decompress a data **without a loss**
- **Low compression rate** without bias
  - Scientific data has a randomness

## Features of lossy

- **High compression rate**
- introduce **an error**

## About introducing an error

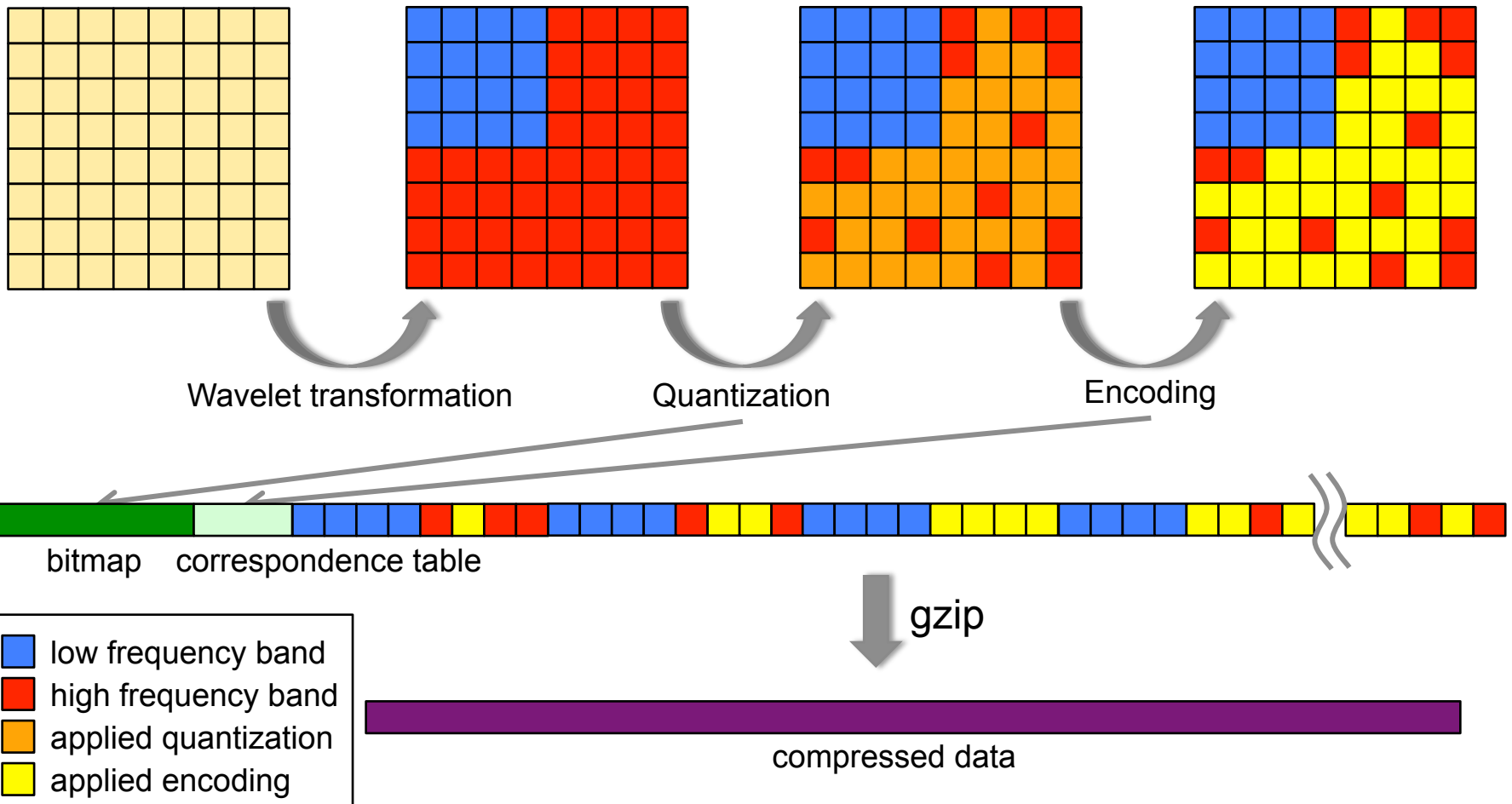
- Possibility of getting equal quality result with introducing an error
- Don't apply lossy compression to a data that must not have an error(pointer etc.)



(citation of images : <http://svs.gsfc.nasa.gov/vis/a000000/a002400/a002478/>)

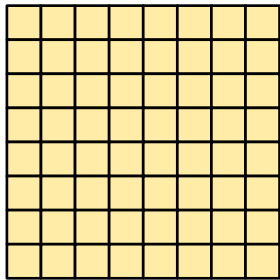
# PROPOSAL APPROACH: LOSSY COMPRESSION WITH WAVELET

We apply wavelet transformation, quantization and encoding to a target data, then compress a data that is stored in proposal output format with *gzip*

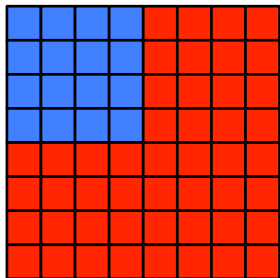


# PROPOSAL APPROACH: LOSSY COMPRESSION WITH WAVELET

## Wavelet transformation

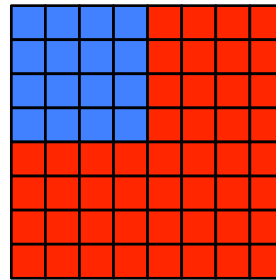


Divide original data into two subbands

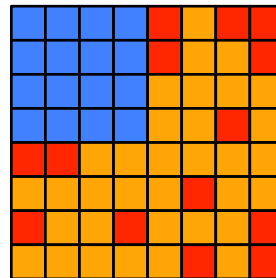


■ use average  
■ use difference  
(most of these  
is close to zero)

## Quantization

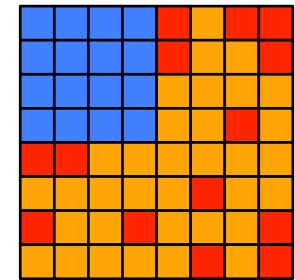


Round *the red values* into  $n$  kind of values

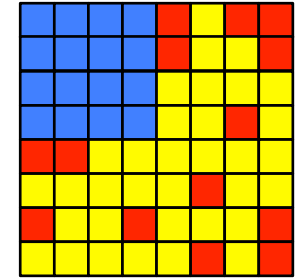


■  $n$  kind of values  
( $n = 2^0 \sim 2^7$ )

## Encoding



Store the *float, double* value to *char* value



■ Data size reduces to  
1/4 or 1/8 at this point

# EVALUATION ENVIRONMENT

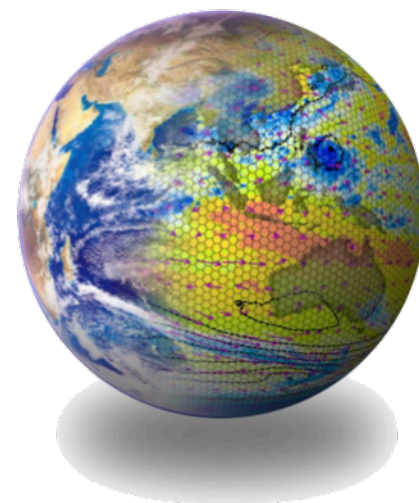
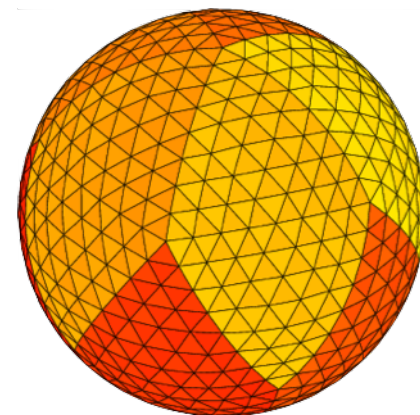
**We apply our approach to climate simulation NICAM[M.Satoh, 2008]**

- Target physical quantity are pressure, temperature and velocity.
  - 3Darray, double precision, 1156\*82\*2
- The data is uniform in initial state
  - apply the method after 720 step from initial state?

**We evaluate compression time and rate and error, while changing the number of division ( $2^0 \sim 2^7$ )**

Machine spec

CPU	Intel Core i7-3930K 6 cores 3.20GHz
Memory size	16GB





# EVALUATION OF COMPRESSION TIME

## An assumption about compression time

- I/O throughput...20GB/s
- Checkpoint size that each process has...about 1.5MB
  - Total checkpoint size...about  $(1.5 \times \# \text{ of parallelism})\text{MB}$

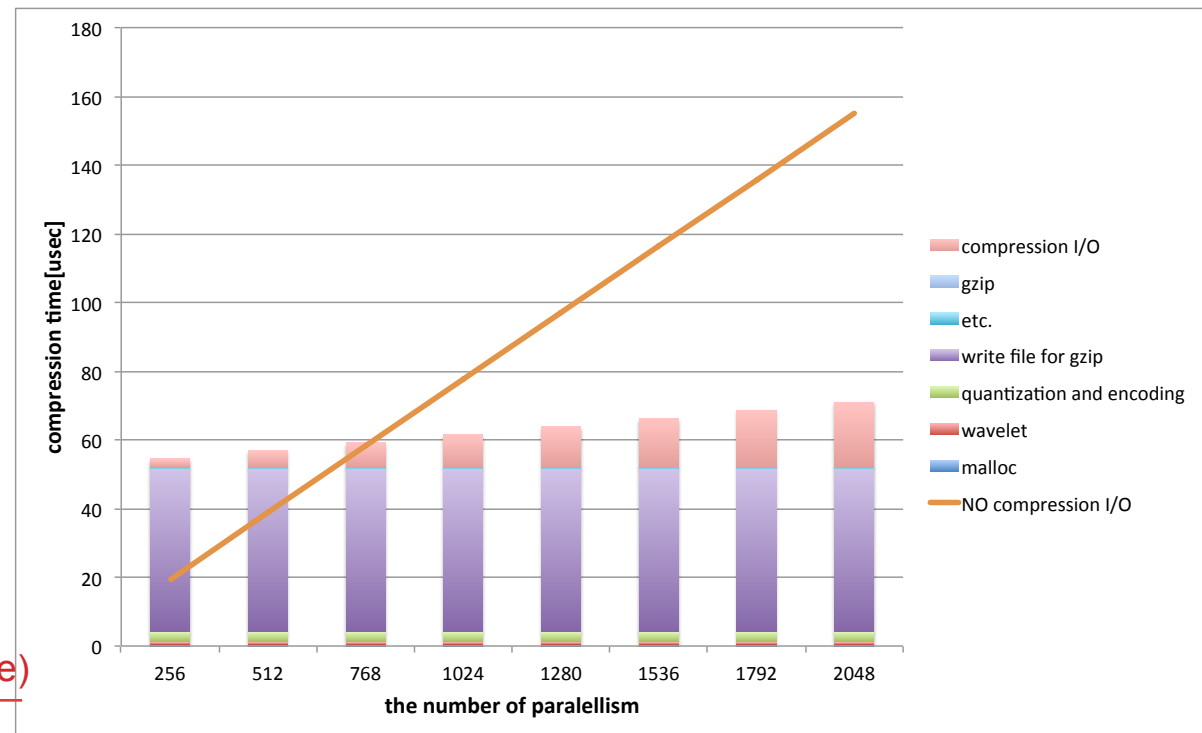
## Actual survey

- Compression time
- Compression rate

## Calculation from assumption

- I/O time

$$\frac{\text{Total checkpoint size}(\times \text{compression rate})}{\text{I/O Throughput}}$$



# EVALUATION OF COMPRESSION TIME

## An assumption about compression time

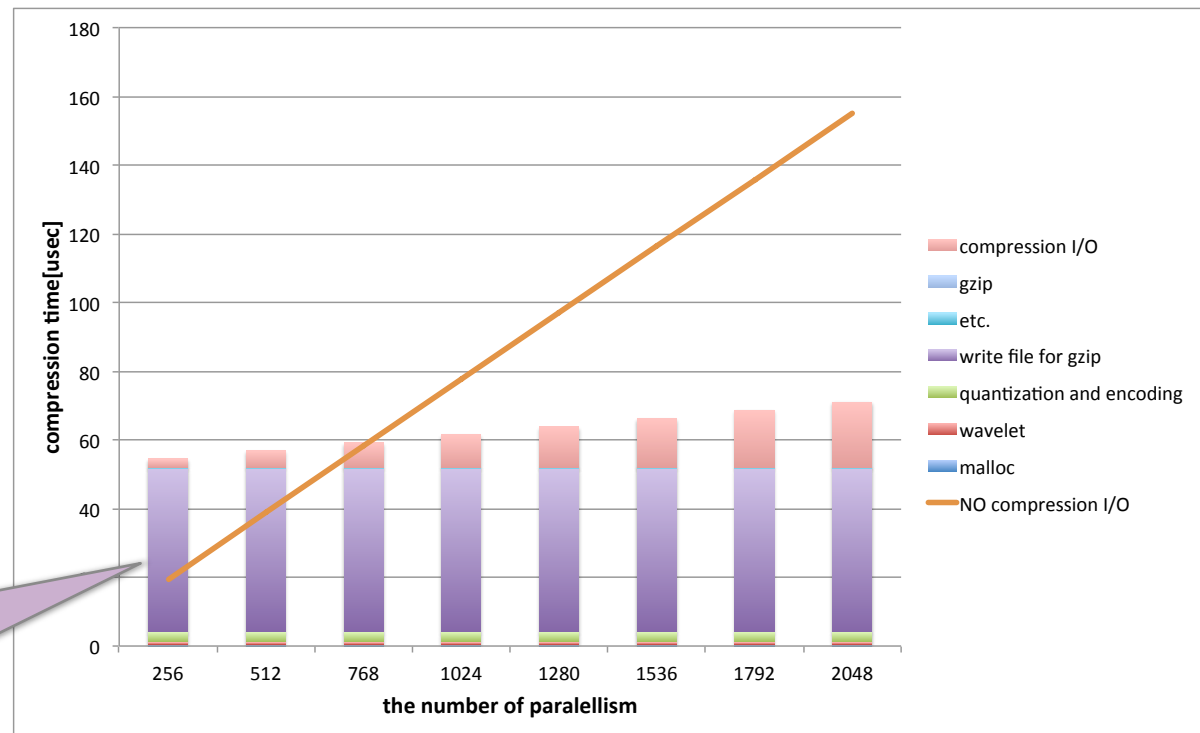
- I/O throughput...**20GB/s**
- Checkpoint size that each process has...**about 1.5MB**  
→ Total checkpoint size...**about  $(1.5 \times \# \text{ of parallelism})\text{MB}$**

## Problems of gzip

- Computational complexity
- Needs for writing files



Write time can be cut if we apply gzip to the data internally



# EVALUATION OF COMPRESSION TIME

## An assumption about compression time

- I/O throughput...**20GB/s**
- Checkpoint size that each process has...**about 1.5MB**  
→ Total checkpoint size...**about  $(1.5 \times \# \text{ of parallelism})\text{MB}$**

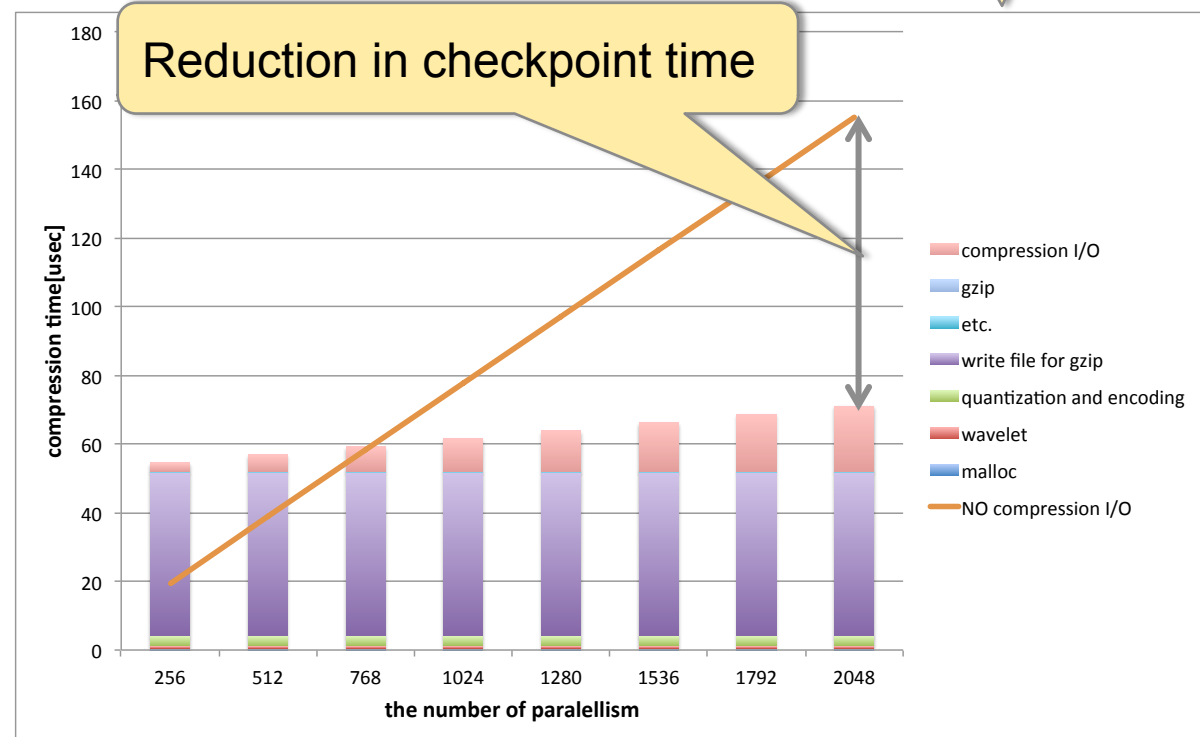
I/O time reduces by **about 70%**, if compression time is negligible by increasing # of parallelism

Each process compress 1.5MB data in spite of # of parallelism

- Compression time is constant
- I/O time depends on total checkpoint size

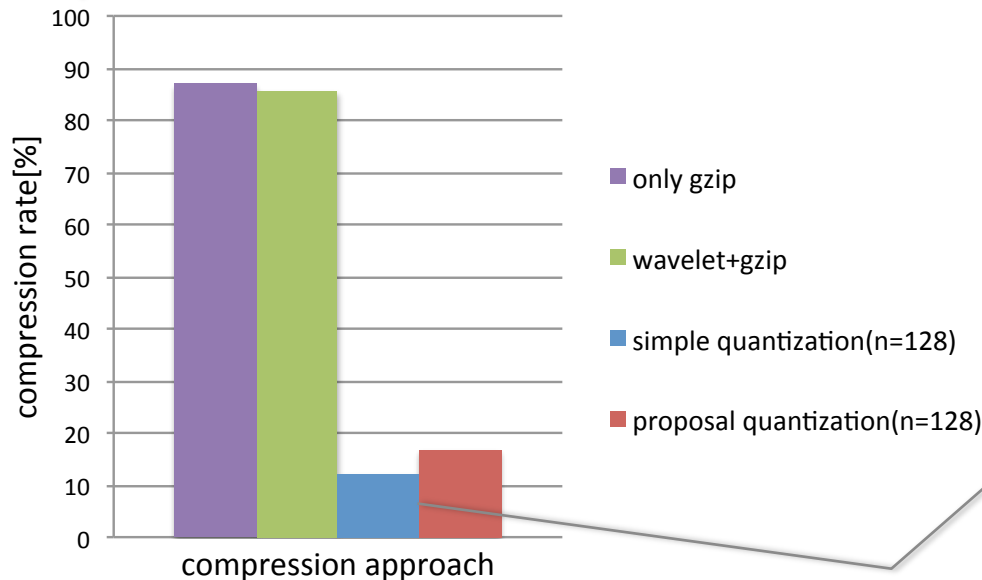
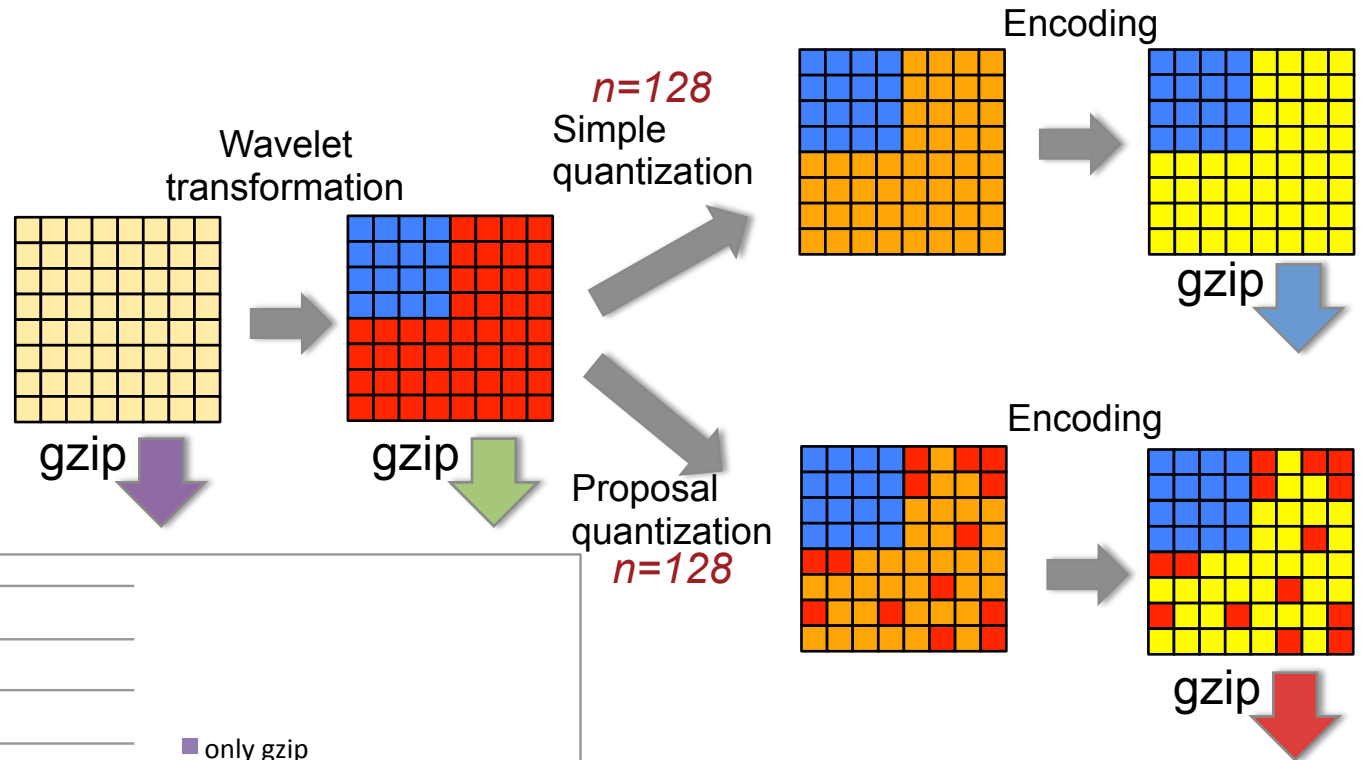


Our approach takes advantage when # of parallelism increases



# COMPARISON TO WITHOUT OUR APPROACH

In comparison with only gzip, our approach reduces checkpoint size by **75%**



Simple quantization achieves better compression rate, but larger error than proposal quantization

# EVALUATION OF ERROR

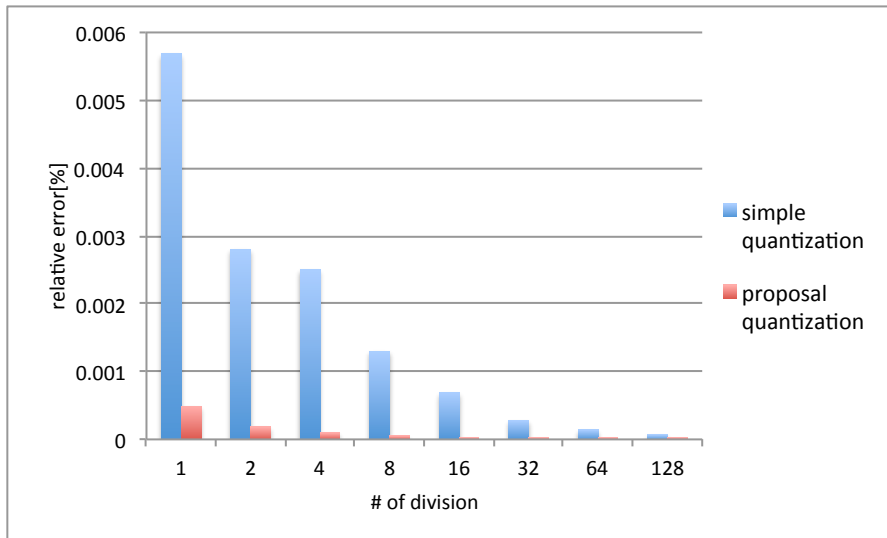
$$RE_i = \frac{x_i - \tilde{x}_i}{\max_j \{x_j\} - \min_j \{x_j\}}$$

Reduce an error with # of division( $n$ ) increasing

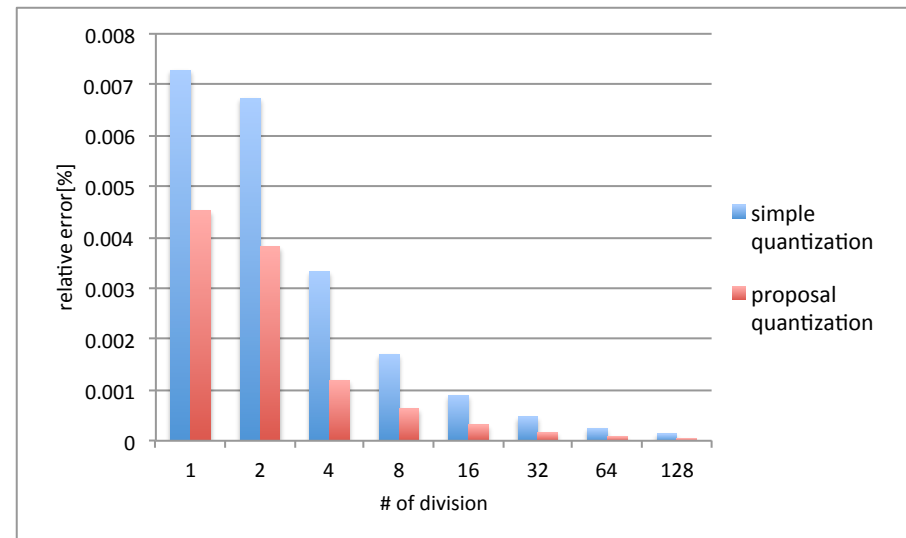
- An error reduce by **about 98%** at  $n = 128$  compared to  $n = 1$

**Our quantization reduce an error in comparison with simple one**

- A degree of reduction of an error is different depending on arrays



An average error on pressure array



An average error on temperature error

On all variables, maximum error is within **5%**

# Summary

- Resilience APIs
  - Resilient APIs in MPI is critical for fast and transparent recovery in HPC applications
- Resilient Architecture
  - Burst buffers Burst buffers are beneficial for C/R at extreme scale
  - Uncoordinated C/R
    - When MTBF is days or a day, uncoordinated C/R may not be effective
    - If MTBF is a few hours or less, will be effective
  - Level-2 failure, and Level-2 performance
    - Reducing Level-2 failure and increasing Level-2 performance are critical to improve overall system efficiency
  - Fewer number of burst buffers
    - Adding additional burst buffer nodes increases the failure rate
    - May degrades system efficiency more than the efficiency gained by the increased bandwidth
    - We need to be careful a trade-off between I/O performance and reliability of burst buffers
- Lossy data compression
  - Preliminary, but promising

# Q & A

## Speaker

Kento Sato  
Lawrence Livermore National Laboratory  
kento@llnl.gov

## External collaborators

Satoshi Matsuoka, Tokyo Tech  
Naoya Maruyama, RIKEN AICS





