APIs, Architecture and Modeling for Extreme Scale Resilience

Dagstuhl Seminar: Resilience in Exascale Computing

9/30/2014

Kento Sato

Lawrence Livermore National Laboratory

LLNL-PRES-661421

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

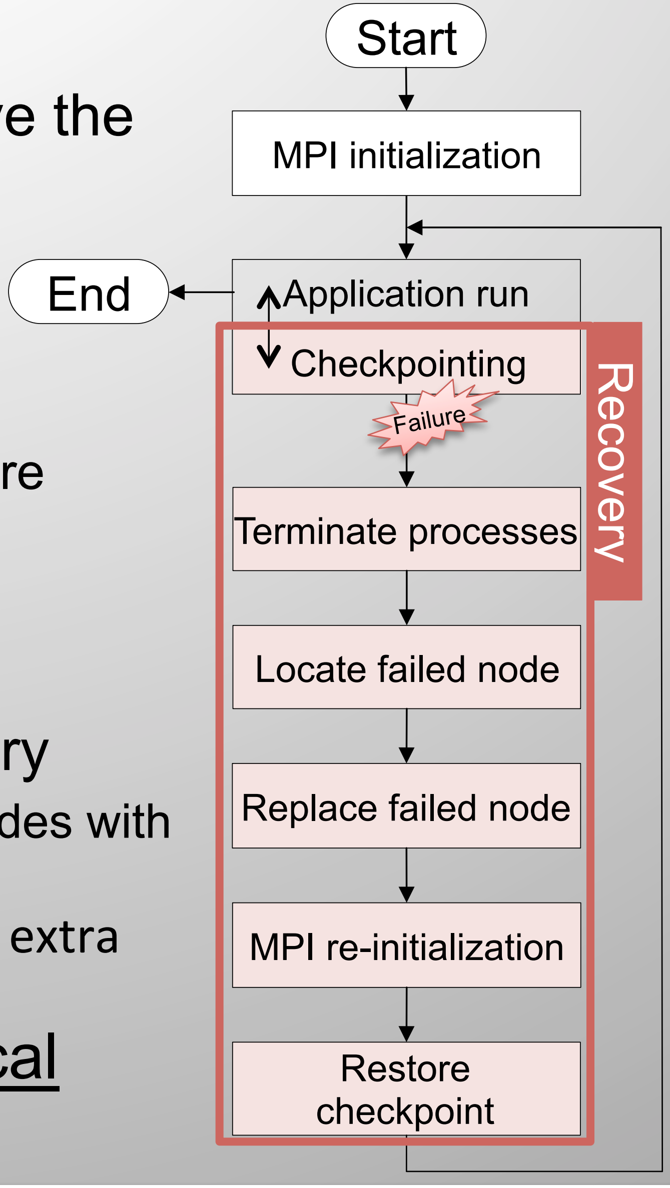
Failures on HPC systems

- System resilience 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 => MTBF: 1.2 day
    - C.f.) TSUBAME2.0 => MTBF: a day
- In extreme scale, failure rate will increase
- Now, HPC systems must consider failures as usual events

Motivation to resilience APIs

- Current MPI implementation does not have the capabilities
  - 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
- Applications will use more time for recovery
  - Users manually locate and replace the failed nodes with spare nodes via `machinefile`
  - The manual recovery operations may introduce extra overhead and human errors

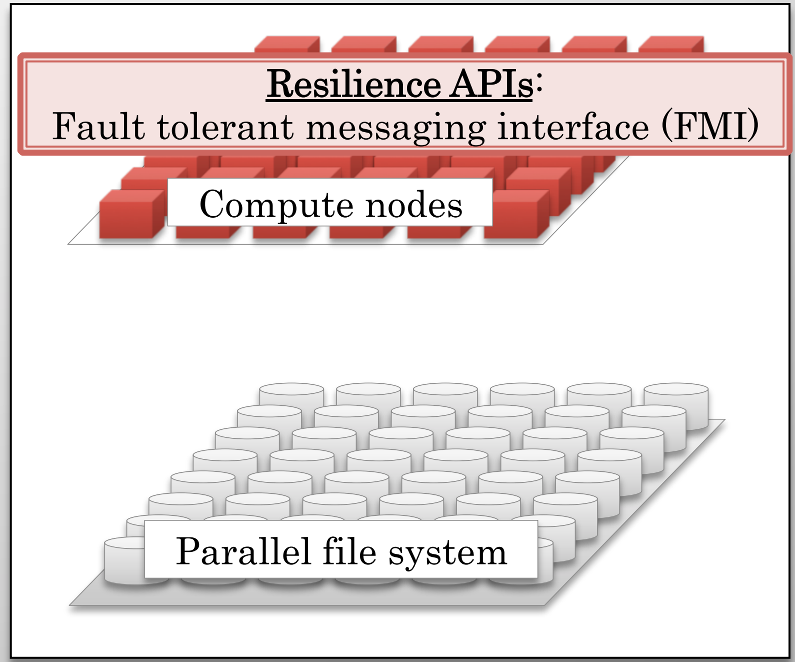
⇒ APIs to handle the failures are critical



Resilience APIs, Architecture and the model

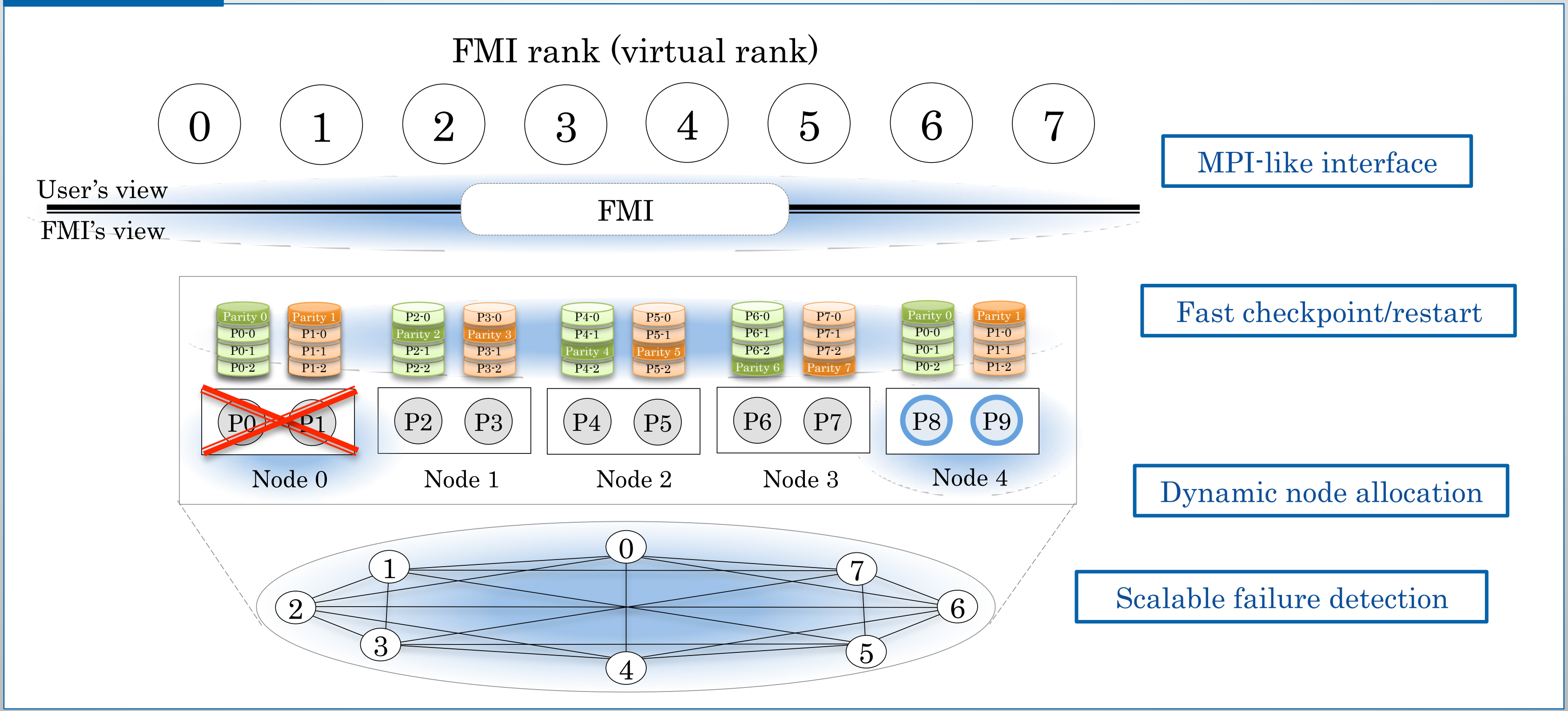
- Resilience APIs

  ⇒ Fault tolerant messaging interface (FMI)



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 ⇒ In-memory diskless checkpoint/restart

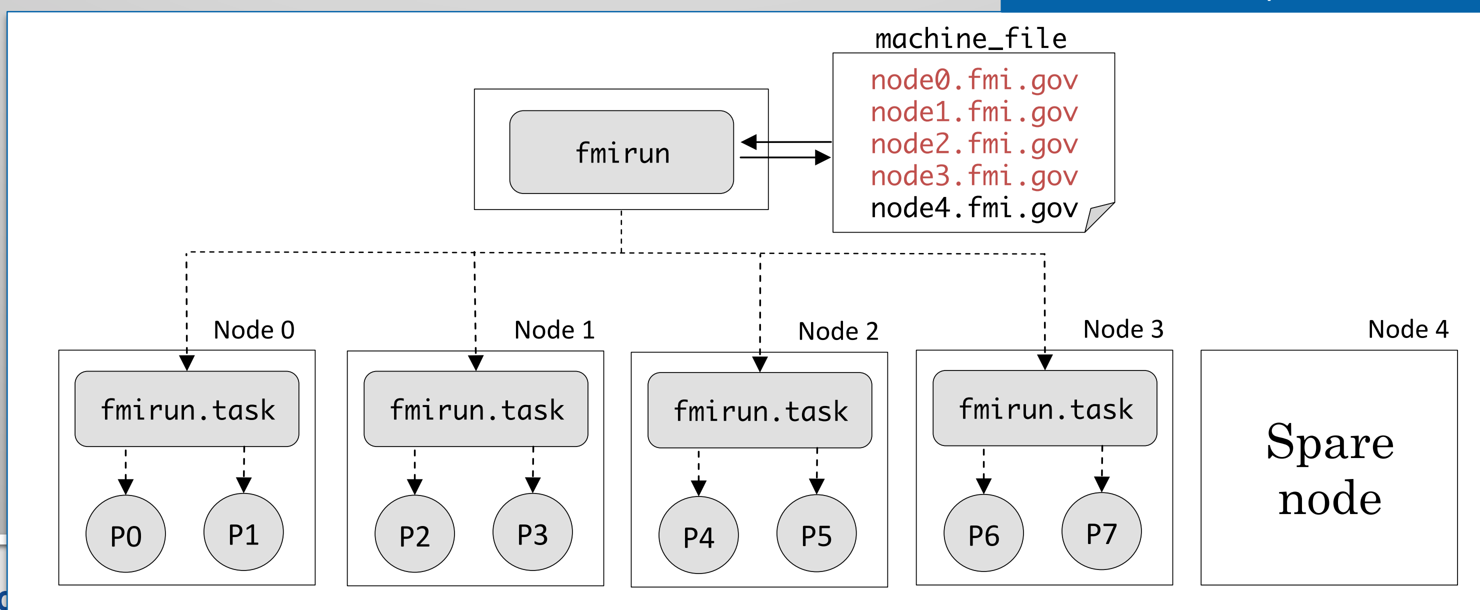
How FMI applications work ?

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(…)

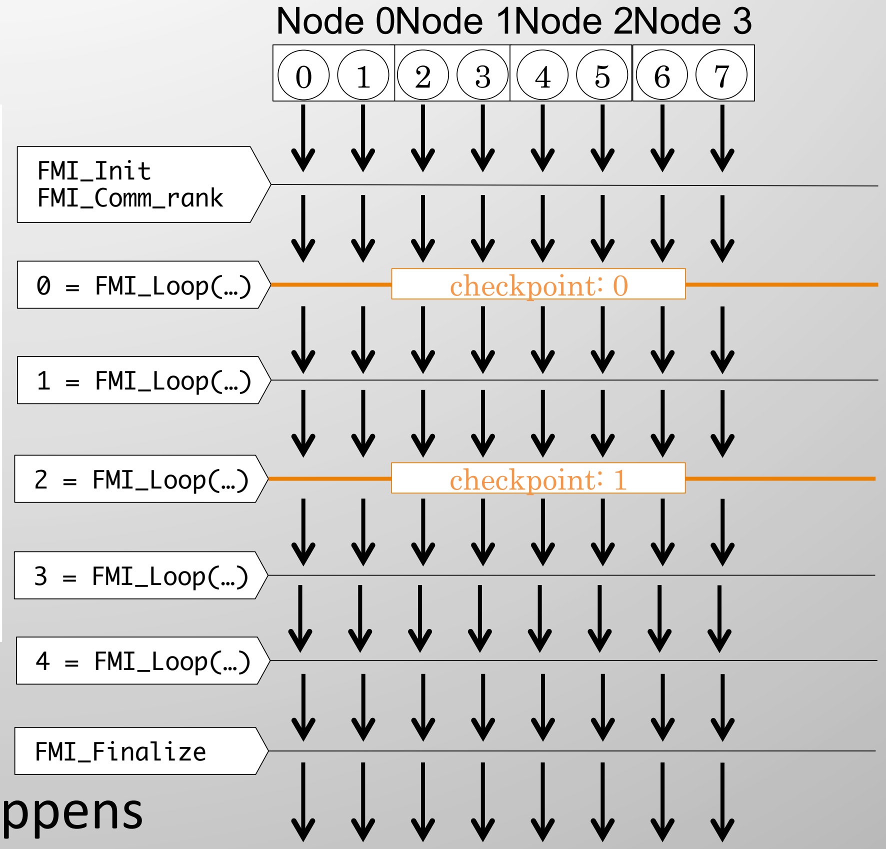
Launch FMI processes



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();
}
```

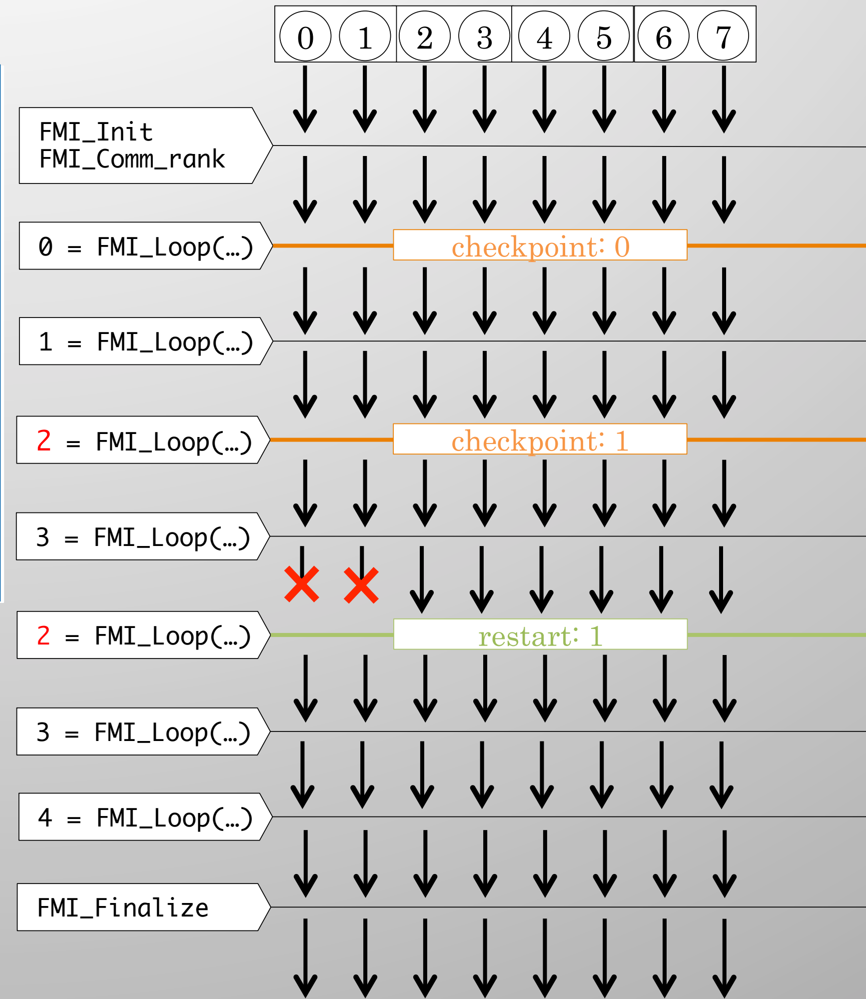


- User perspective when no failures happens
- Iterations: 4
- Checkpoint frequency: Every 2 iterations
- FMI_Loop returns incremented iteration id

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

Resilience API: FMI_Loop

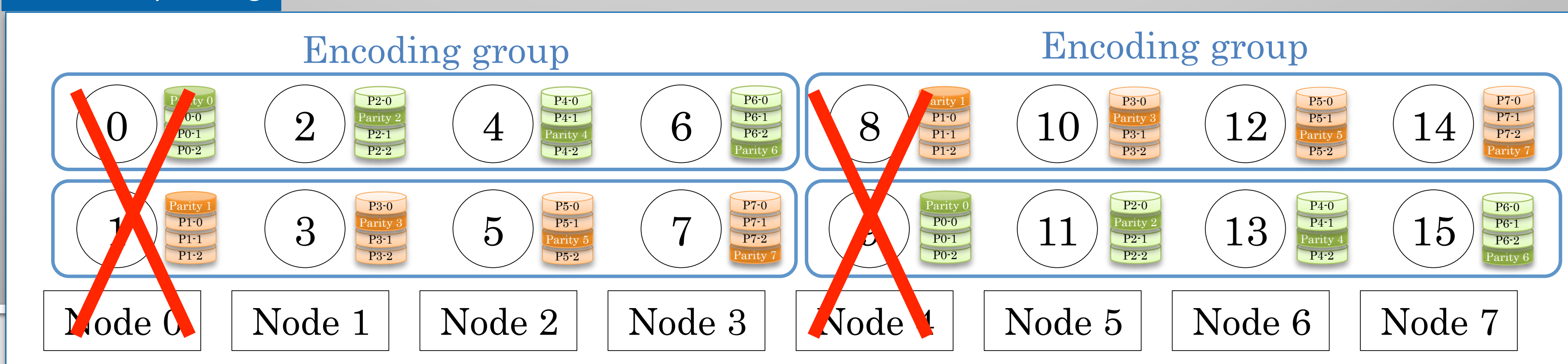
```
int FMI_Loop(void **ckpt, size_t *sizes, int len)
```

- `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`

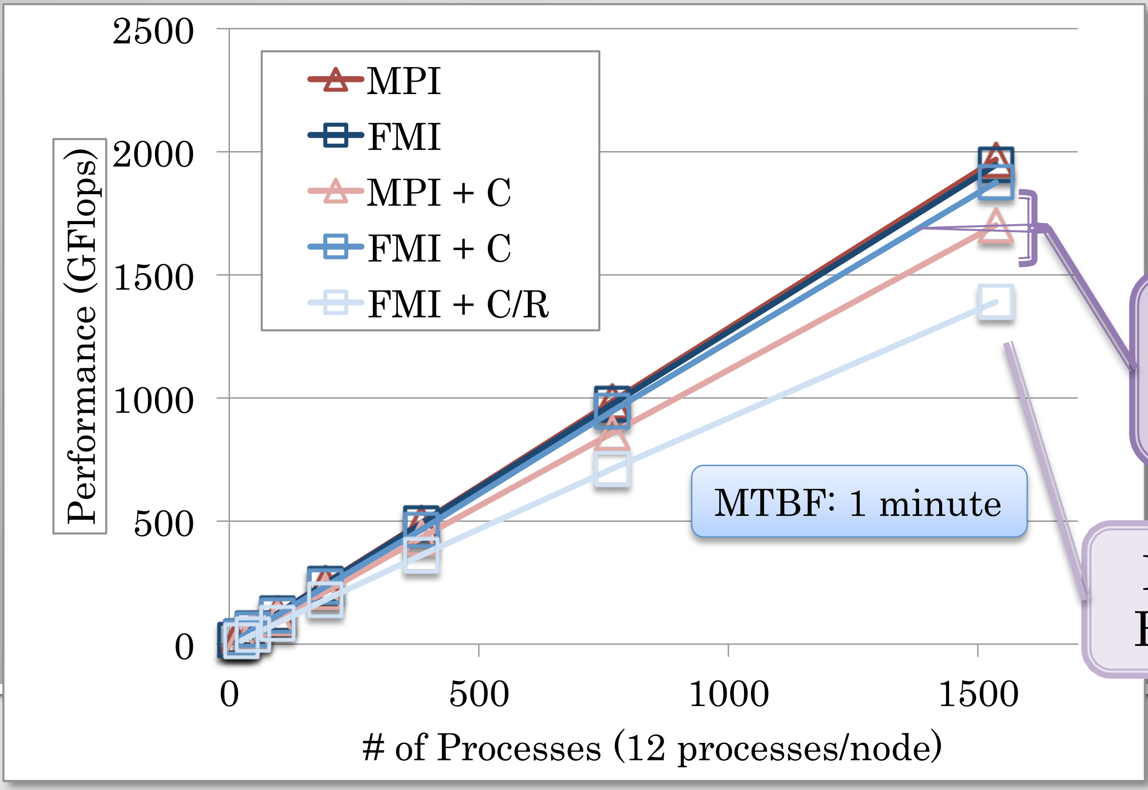
- FMI constructs in-memory RAID-5 across compute nodes
- Checkpoint group size
  - e.g.) group_size = 4

FMI checkpointing



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



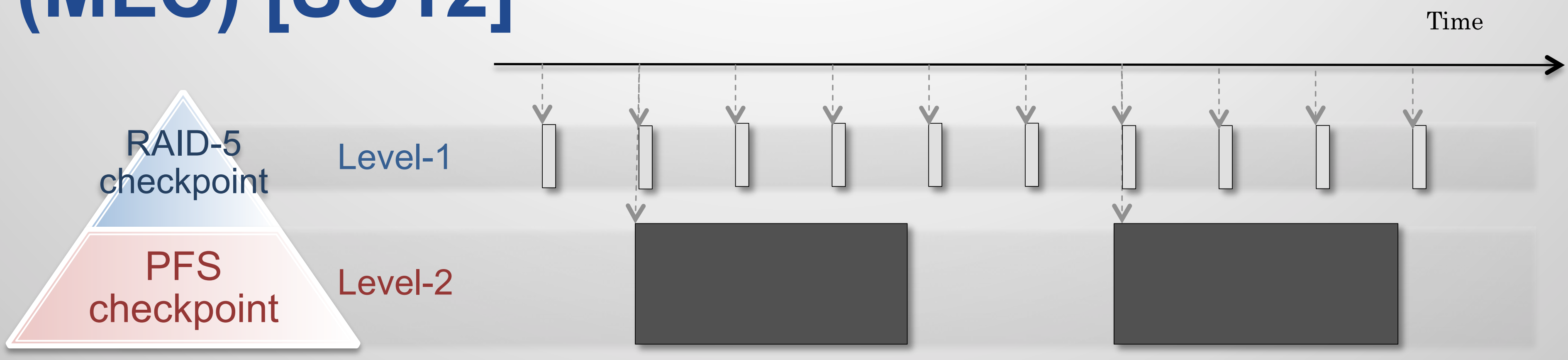
P2P communication performance

	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

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
  - RAID-5 checkpoint: Frequent for one node or a few node failure
  - PFS checkpoint: Less frequent and asynchronous for multi-node failure
- Our previous work model the asynchronous MLC

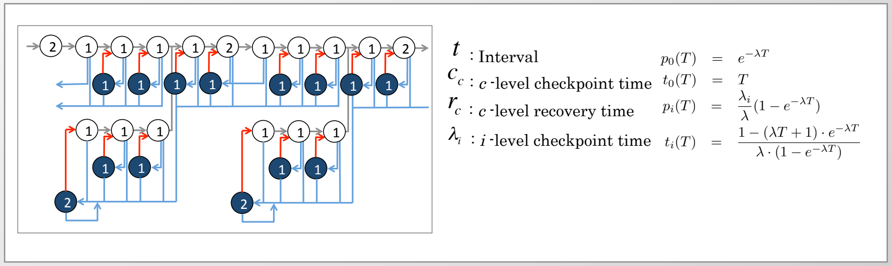
Failure analysis on Coastal cluster

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

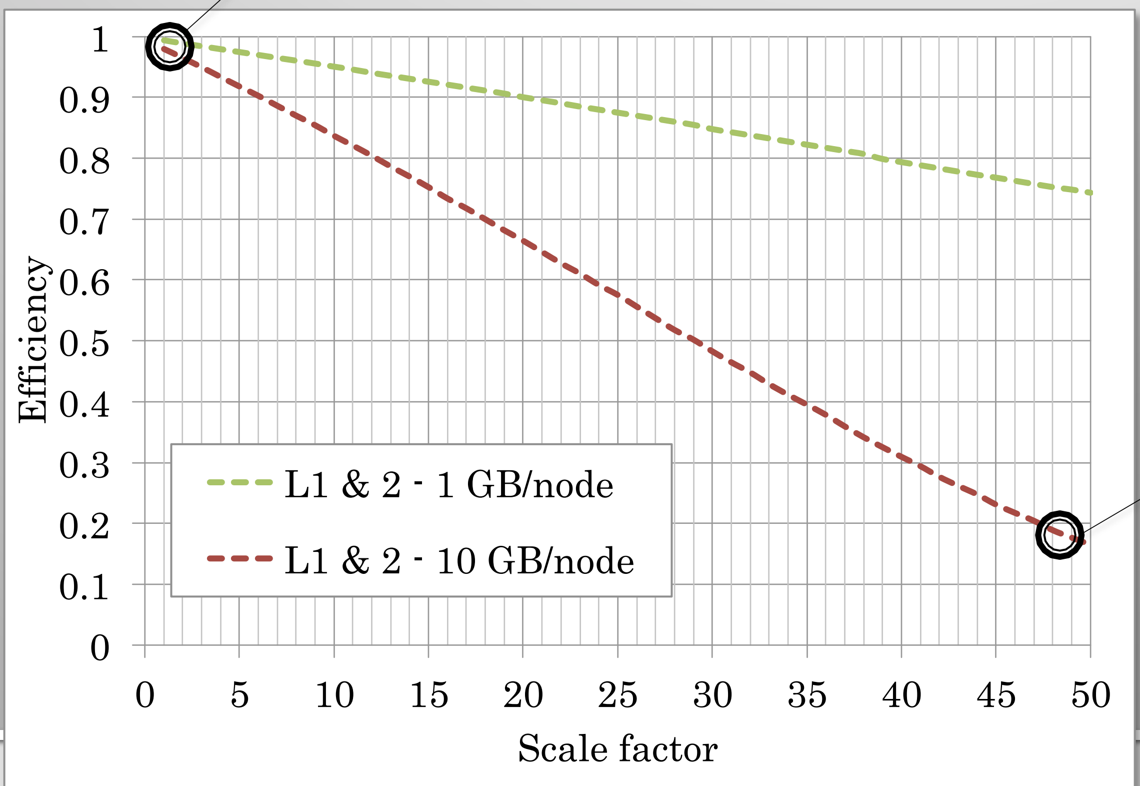
Source: A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10).

Simulation based on Asynchronous MLC

Async. MLC (Multi-level C/R) model



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



High efficiency with current failure rate

If both L1 & L2 failure rate increase, and checkpoint size is large, efficiency decrease faster

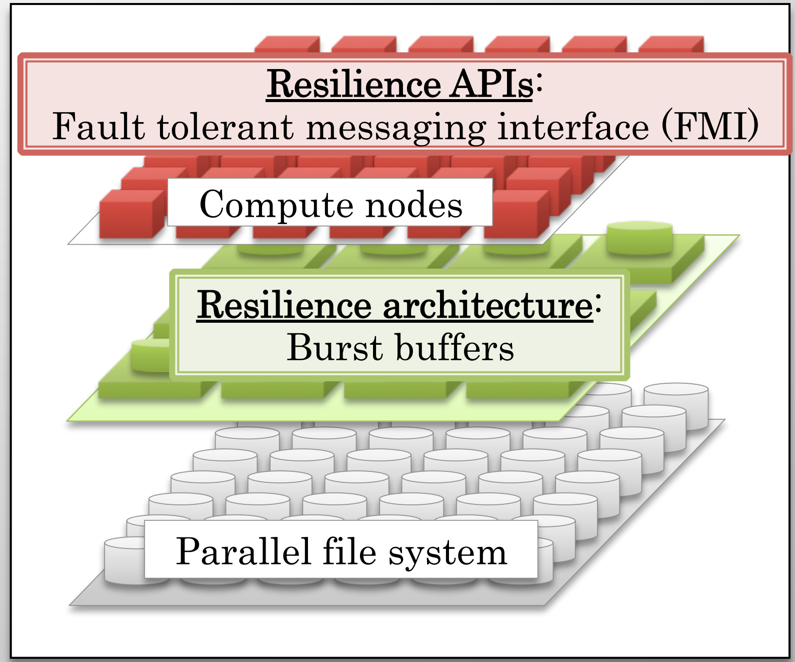
Resilience APIs, Architecture and the model

- Resilience APIs
  - In near future, applications must have capabilities of handling failures as usual events

  ⇒ Fault tolerant messaging interface (FMI)

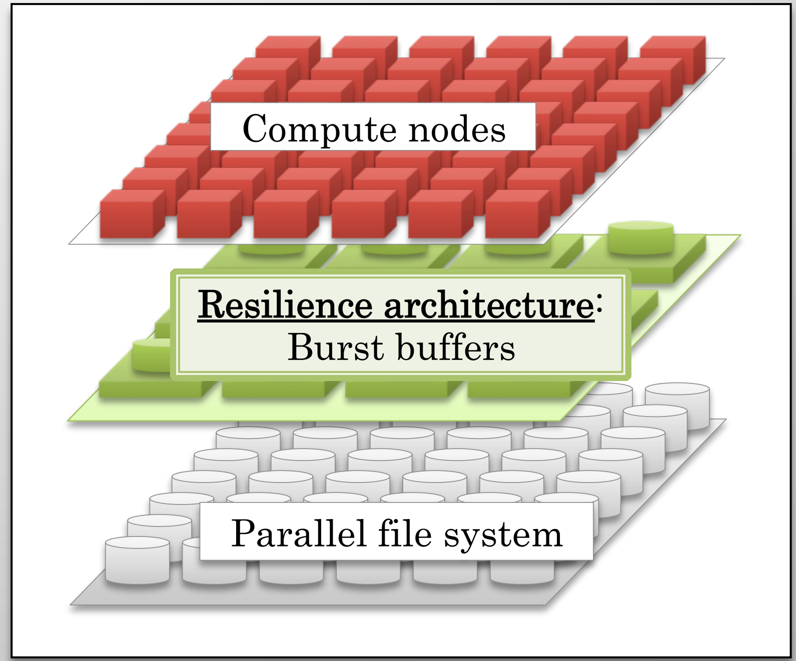
- Resilience architecture and model
  - Software level approaches are not enough

  ⇒ Architecture using *Burst buffer*



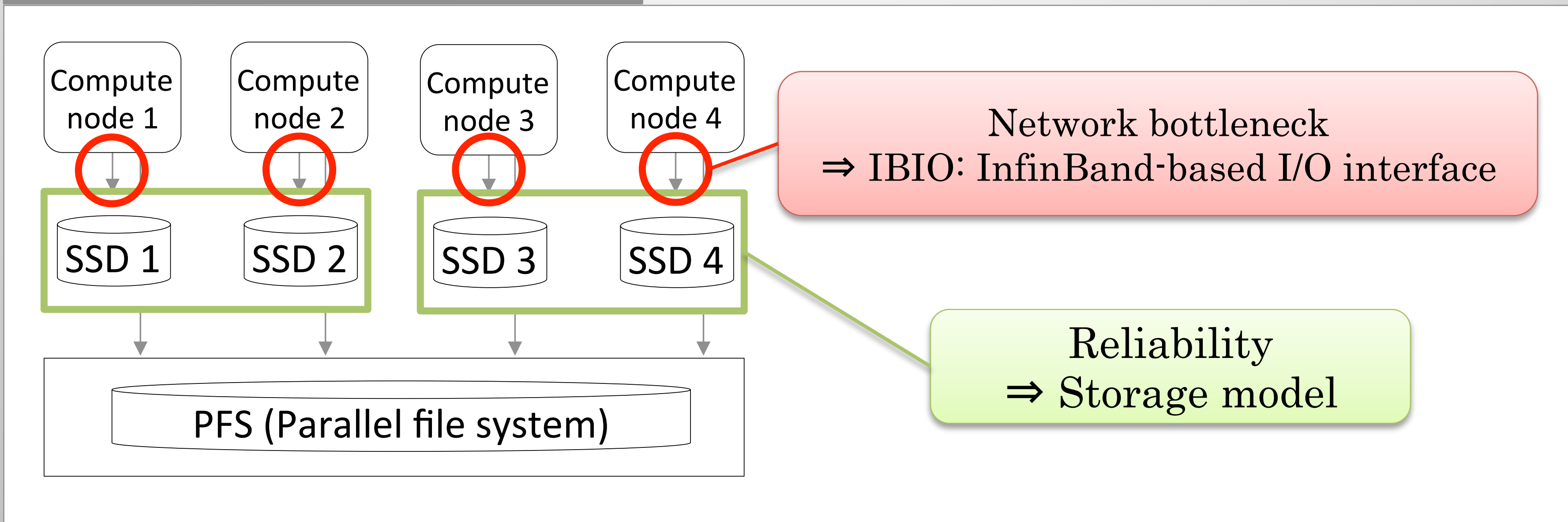
Burst buffer storage architecture

- 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



Burst buffer storage architecture (cont'd)

Challenges for using burst buffer system

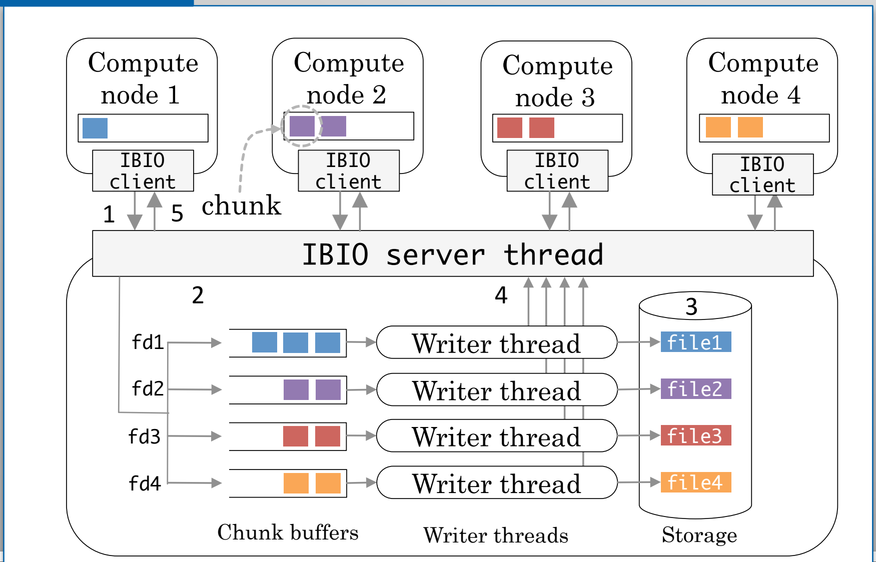


- Exploiting storage bandwidth of burst buffers
  - Burst buffers are connected to networks, networks can be bottleneck
- Analyzing reliability of systems with burst buffers
  - Adding burst buffer nodes increase total system size
  - System efficiency may decrease due to Increased overall failure by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

APIs for burst buffers: InfiniBand-based I/O interface (IBIO)

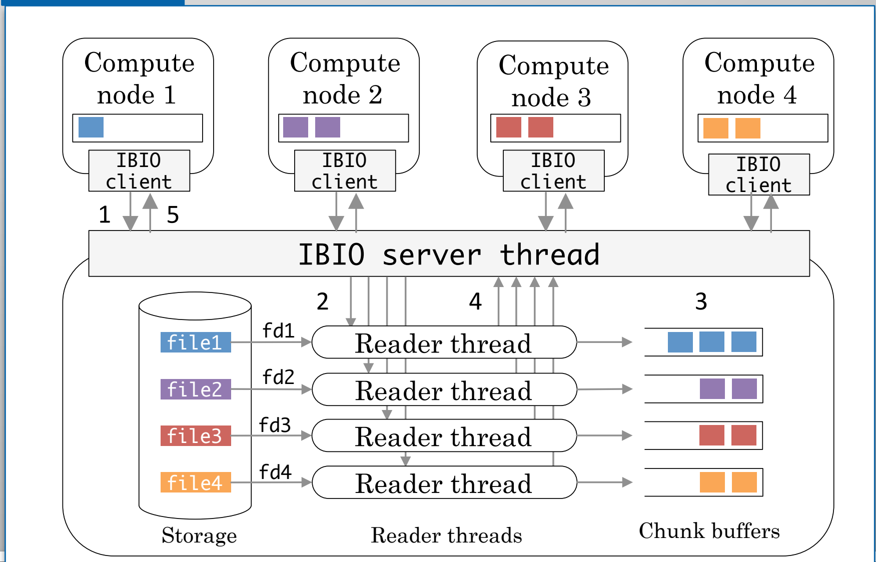
- Provide POSIX-like I/O interfaces
  - Open, read, write and close operations
  - Client can open any files on any servers
    - `open("hostname:/path/to/file", mode)`
- IBIO use ibverbs for communication between clients and servers
  - Exploit network bandwidth of infiniBand

IBIO write



IBIO write: four IBIO clients and one IBIO server

IBIO read



IBIO read: four IBIO clients and one IBIO server

Resilience modeling overview

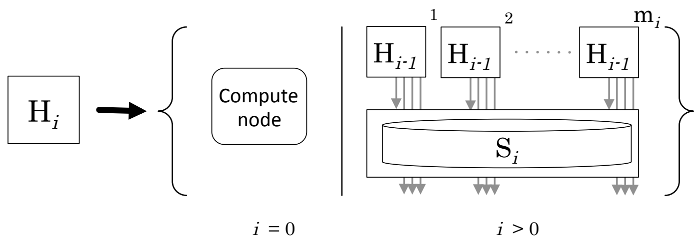
- To find out the best checkpoint/restart strategy for systems with burst buffers, we model checkpointing strategies

C/R strategy model

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

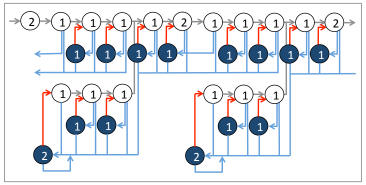
$$C_i \text{ or } R_i = \frac{\text{< C/R date size / node >} \times \text{<\# of C/R nodes per } S_i^* \text{>}}{\text{< write perf. (} w_i \text{) > or <read perf. (} r_i \text{) >}}$$

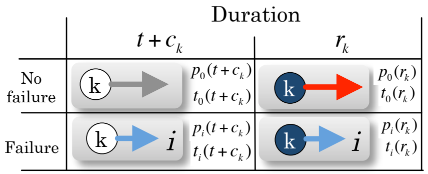
Recursive structured storage model



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

Async. MLC model [2]





t : Interval
c_c : c-level checkpoint time
r_c : c-level recovery time
λ_i : i-level checkpoint time

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

$p_0(T)$: 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)

[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

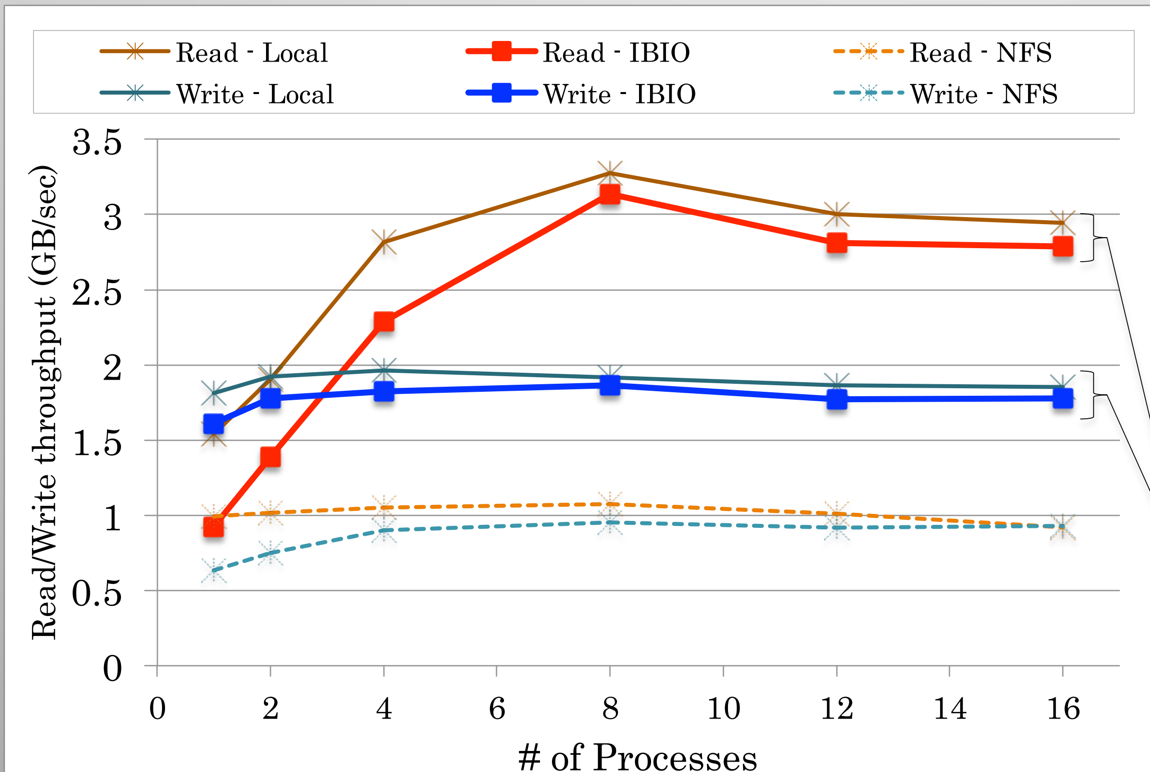
Sequential IBIO read/write performance

- 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

EBD I/O



Node specification

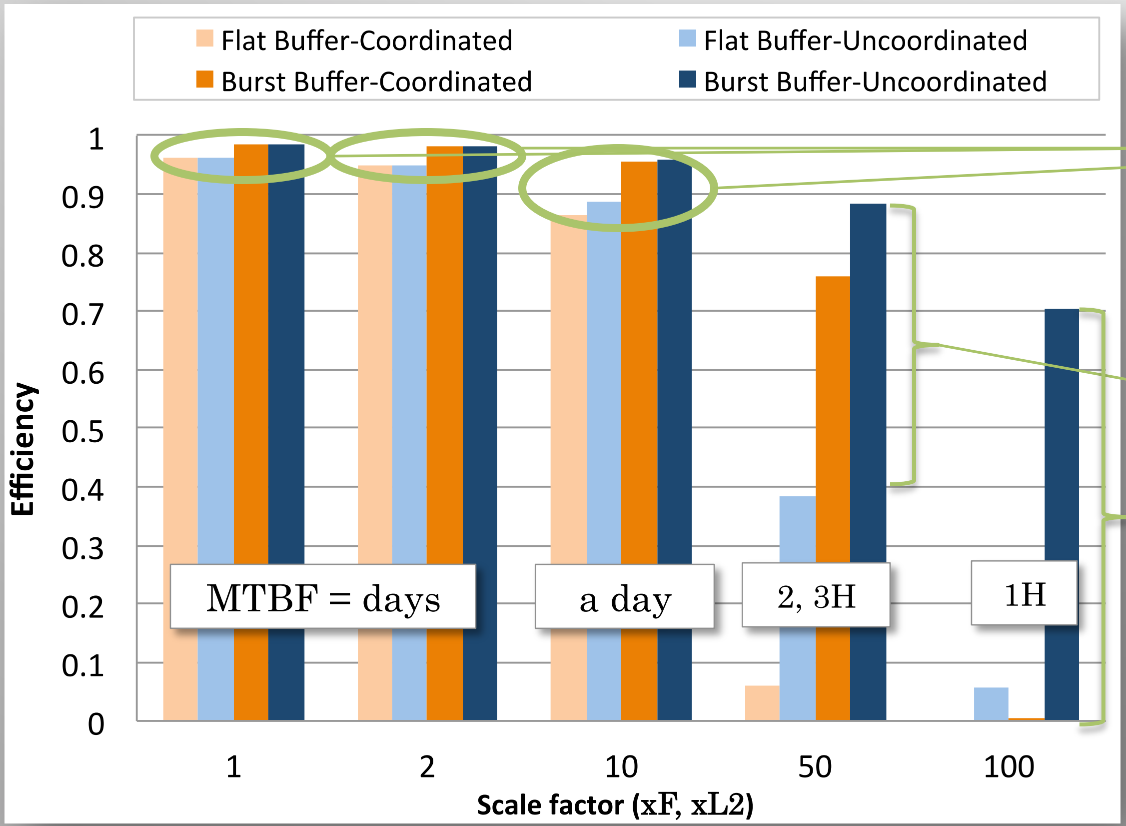
CPU	Intel Core i7-3770K CPU (3.50GHz x 4 cores)
Memory	Cetus DDR3-1600 (16GB)
M/B	GIGABYTE GA-Z77X-UD5H
SSD	Crucial m4 msata 256GB CT256M4SSD3 (Peak read: 500MB/s, Peak write: 260MB/s)
SATA converter	KOUTECH IO-ASS110 mSATA to 2.5' SATA Device Converter with Metal Fram
RAID Card	Adaptec RAID 7805Q ASR-7805Q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

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 can decrease 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	≈ 100	[illegible]	99.9%

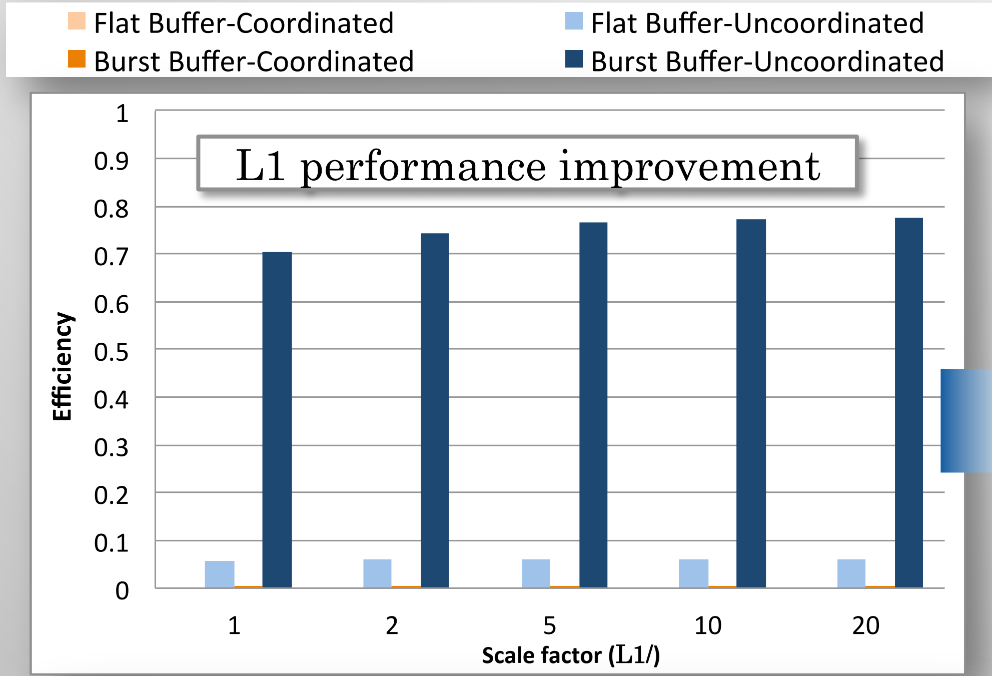
Coordinated

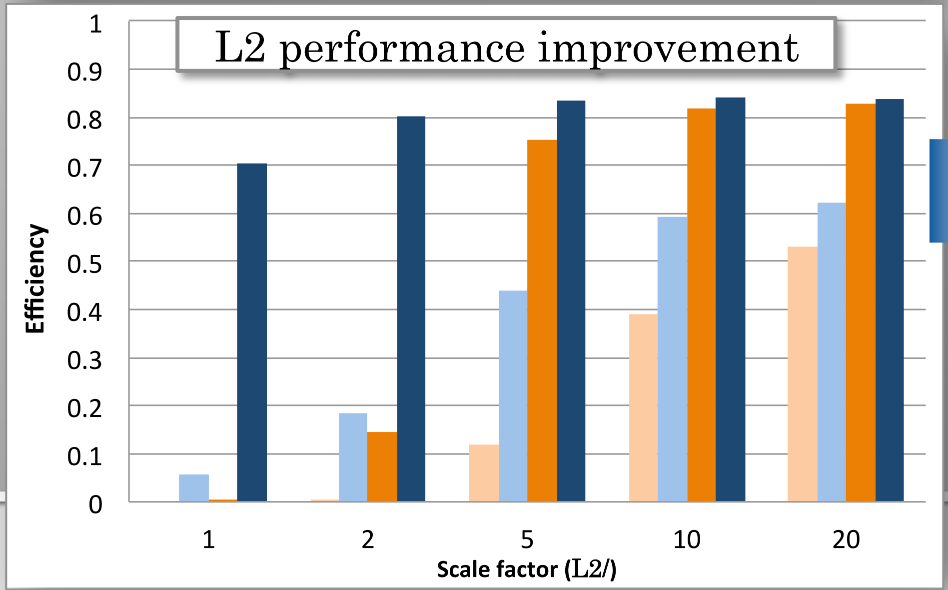
Uncoordinated

- 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





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

Summary: Towards extreme scale resiliency

- Resilient APIs
  - Resilient APIs in MPI is critical for fast and transparent recovery in HPC applications
  - In-memory C/R by FMI incurs only a 28% overhead even with the high failure rate
  - Software-level solution may not enough at extreme scale
- Resilient Architecture
  - 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(PFS) performance
    - Reducing Level-2 failure, increasing Level-2 (PFS) performance are critical to improve overall system efficiency

Q & A

Speaker

Kento Sato
Lawrence Livermore National Laboratory
kento@llnl.gov

External collaborators

Satoshi Matsuoka, Tokyo Tech
Naoya Maruyama, RIKEN AICS

Lawrence Livermore National Laboratory