TOHOKU UNIVERSITY

Automatic Parameter Tuning for Efficient Checkpointing

WSSP28 @ HLRS

Hiroyuki Takizawa (Tohoku University)

<takizawa@tohoku.ac.jp>

Muhammad Alfian Amrizal, Kazuhiko Komatsu, and Ryusuke Egawa

Outline

- Introduction
- Checkpointing methods
- Application-level incremental checkpointing with auto-tuning
- Evaluation and discussions
- Conclusions

Background

- Checkpointing is to dump a whole memory image of a running process to a checkpoint file, from which the process can restart.
  - One of the most intensive I/O operations in HPC applications
    - The I/O performance will not increase at the same pace as the computation performance.
    - The ratio between the I/O performance and the computation performance will be larger in the future system.
  - Future system will require more frequent checkpointing.
    - Future system will consist of much more hardware components, resulting in a higher probability of facing a failure during execution.

  → The checkpointing overhead could dominate the total execution time.

We need to reduce the checkpointing overhead to efficiently use future computing systems.

→ Various approaches have been proposed so far.

Incremental checkpointing

Application-level checkpointing

This Work

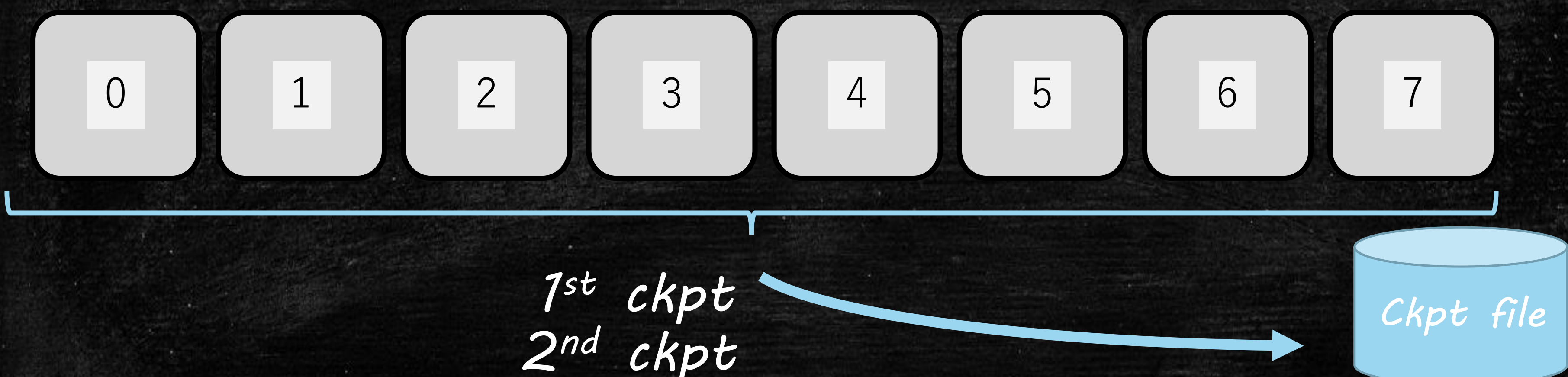
- **A combination of application-level ckpt and incremental ckpt.**
  - *Automatic parameter tuning (**auto-tuning**) is also employed to reduce the overhead.*
  - *A simple API, **Appicpr**, is provided as a prototype implementation of the proposed approach.*

Outline

- Introduction
- Checkpointing methods
- Application-level incremental checkpointing with auto-tuning
- Evaluation and discussions
- Conclusions

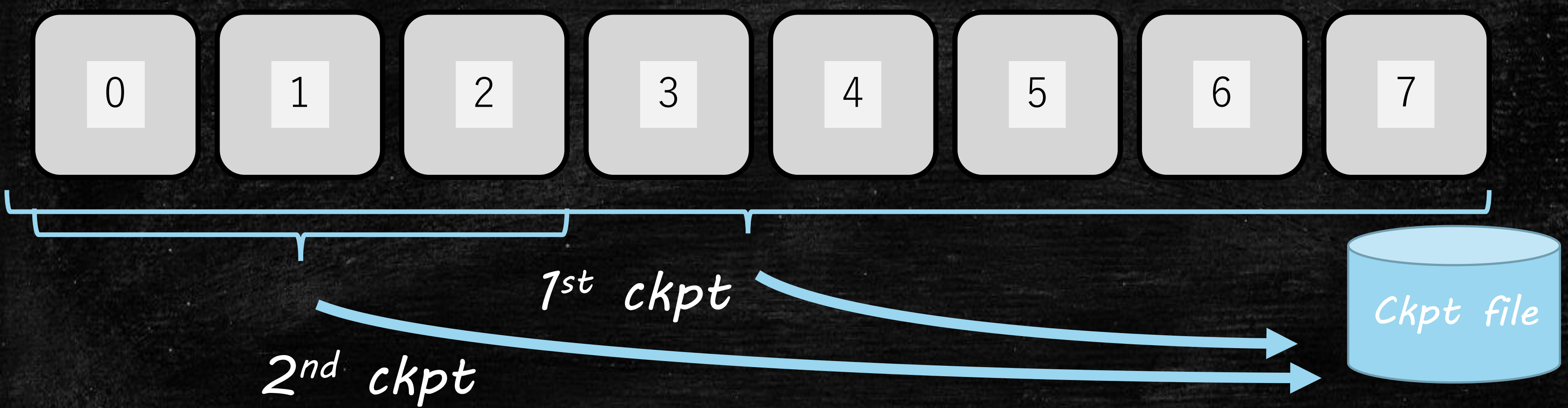
Application-Level Ckpt

- **Only necessary data for restarting the process are periodically saved in a checkpoint file.**
  - *Programmers explicitly write the file I/O operations in their applications.*
  - *Most of practical HPC applications would have a kind of application-level checkpointing capability, which simply write specific data to checkpoint files.*
    - *For example, `printf` in C language, and `WRITE` statement in Fortran are simply used for the file I/O operations of application-level checkpointing.*



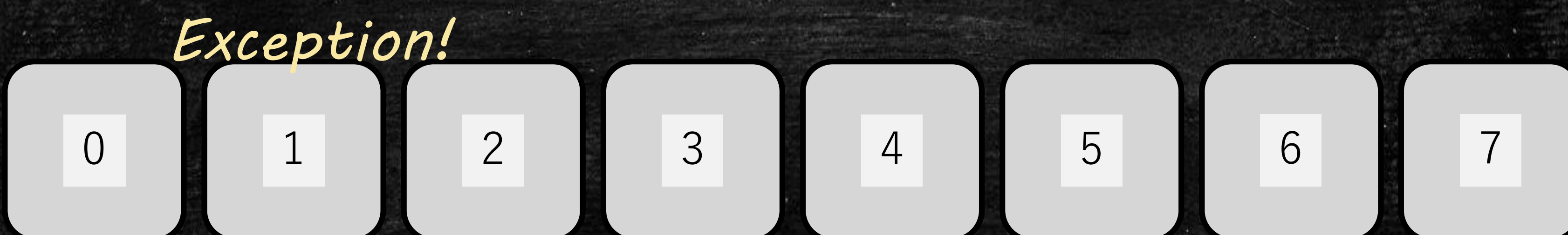
Incremental Ckpt

- **Only updated data** since the last checkpointing are written to a checkpoint file to overwrite the previous data.
  - *Reduce the amount of data written upon checkpointing, and hence the checkpoint overhead.*



Page-Based Incremental Ckpt

- All pages are write-protected after checkpointing.
- An exception occurs when an application tries to update a page.
- The exception handler records the information about updated pages, and disables the protection.
- Finally, the application can update the page.



	0	1	2	3	4	5	6	7
protect	yes	no	yes	yes	yes	yes	yes	yes
update	no	yes	no	no	no	no	no	no

Implementation Issues of Incremental Ckpt

- **Implementation needs system programming**
  - *Write protection of pages*
  - *Exception handling*

→ **Application programmers might be unfamiliar with them.**
  - *In general, incremental checkpointing has been implemented* **as system-level checkponting, not application-level.**

- **The cost of exception handling is not negligible.**
  - *The first write access to each page since the last checkpointing invokes exception handler. Exception handling might be invoked frequently, resulting in degrading the memory access performance.*

Outline

- Introduction
- Checkpointing methods
- Application-level incremental checkpointing with auto-tuning
- Evaluation and discussions
- Conclusions

Application-Level Incremental Ckpt with Auto-Tuning

- **Appicpr: a simple API for programmers to make application-level ckpt incremental.**
  - The API is designed by considering legacy HPC applications in mind, and can be called from Fortran programs.

- **Multiple pages are merged into one management region for the update information management.**
  - How many pages should be merged into one management region? The optimal management region size depends on the memory update patterns of the application.

→ Automatic tuning for each application.

Code example

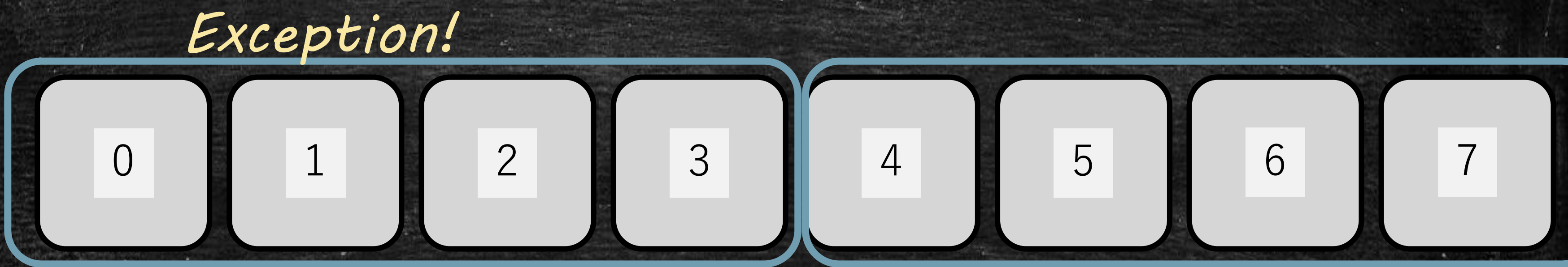
- **Conventional application-level ckpt**

```
real, dimension(asize,asize) :: array

open(newunit=u,file='test.dat',form='unformatted')
write(u) array
close(u)
```

- **Application-level ckpt with Appicpr**

```
real, dimension(asize,asize) :: array

!open(newunit=u,file='test.dat',form='unformatted')
call appic_open('test.dat')
call appic_register(array,sizeof(array))
!write(u) array
call appic_write(array)
!close(u)
call appic_close()
```

Effects of Merging Pages



	0	1	2	3	4	5	6	7
protect	no	no	no	no	yes	yes	yes	yes
update	?	yes	?	?	no	no	no	no

- **Merit**
  - *Reducing the number of exceptions handler invocations, and thus **reduce the exception handling overhead**.*
- **Demerit**
  - *The whole of each management region is written to a checkpoint file even though **it may contain unchanged pages**.*
    = After disabling the write protection, an exception does not occur and thus the update information about the other pages is unknown.

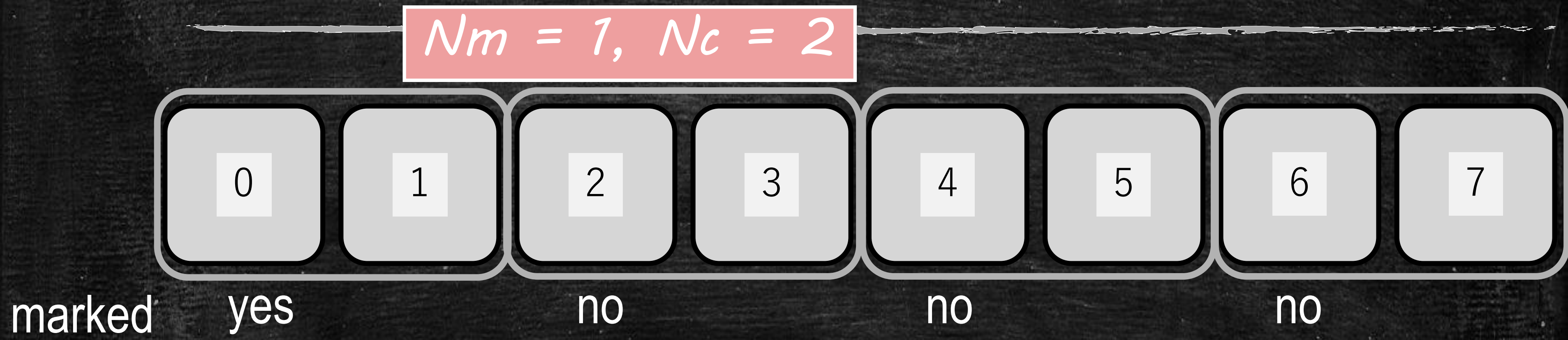
Management Granularity Auto-Tuning

Nm = 2, Nc = 3

	0	1	2	3	4	5	6	7
marked	yes	yes	no	no	no	no	no	no

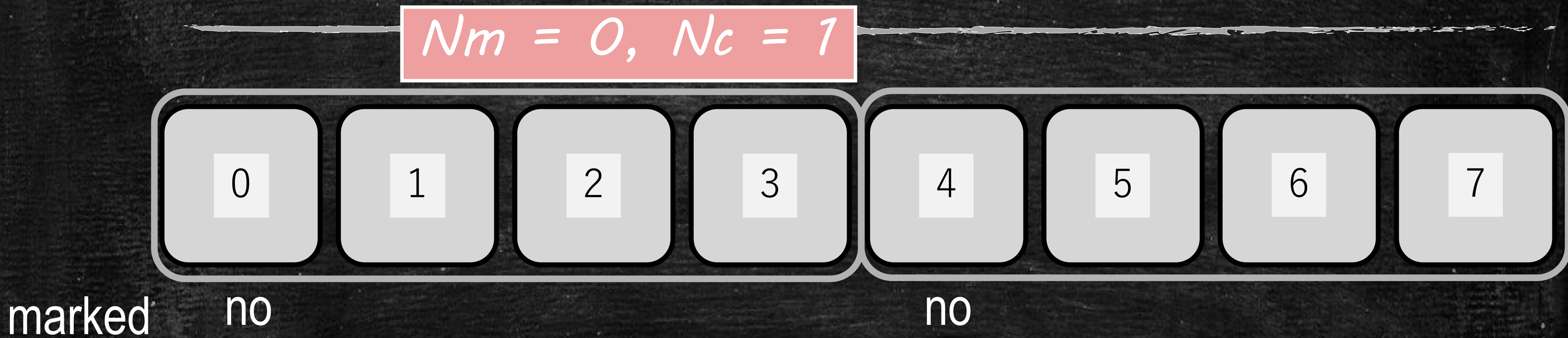
- **Procedure of management granularity auto-tuning**
  - *Initially, each page is a management region.*
  - *At checkpointing, only updated regions are written to a checkpoint file.*
  - *If a region and its next region are both updated, the region is "marked"*
  - *Number of marked regions >= Number of updated regions / 2.*
    = Marked regions are in majority of updated regions.
  - *The management granularity (the number of pages in a management region) is doubled.*

Management Granularity Auto-Tuning



- **Procedure of management granularity auto-tuning**
  - *Initially, each page is a management region.*
  - *At checkpointing, only updated regions are written to a checkpoint file.*
  - *If a region and its next region are both updated, the region is "marked"*
  - *Number of marked regions >= Number of updated regions / 2.*
    = Marked regions are in majority of updated regions.
  - *The management granularity (the number of pages in a management region) is doubled.*

Management Granularity Auto-Tuning



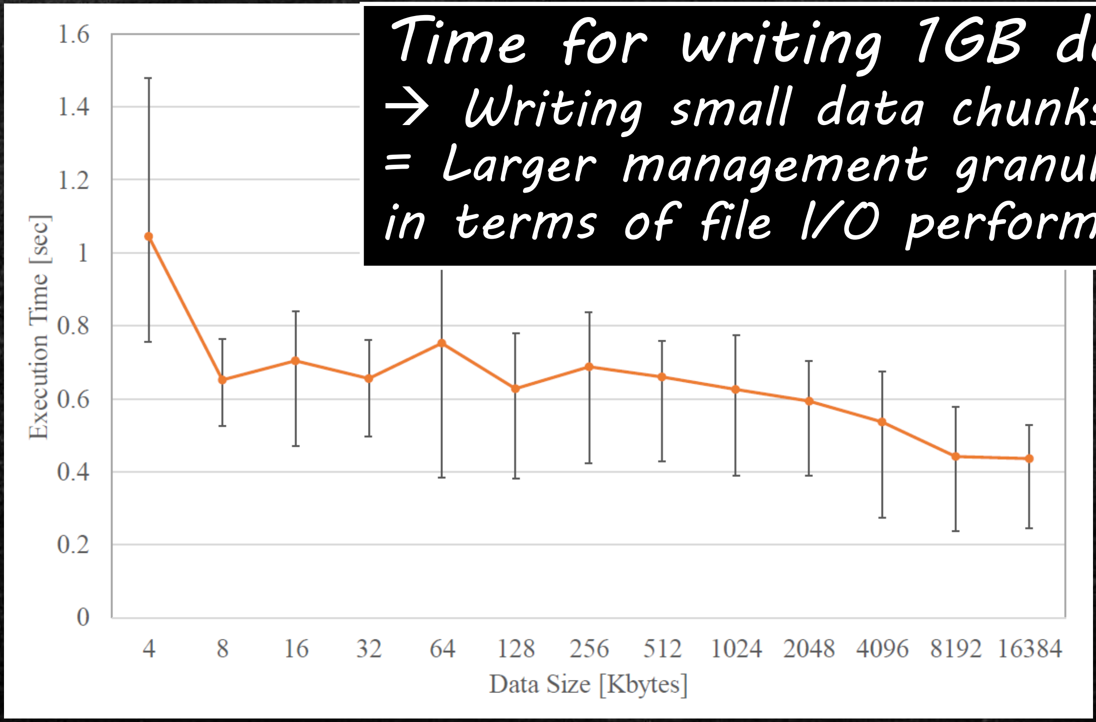
- **Procedure of management granularity auto-tuning**
  - Initially, each page is a management region.
  - At checkpointing, only updated regions are written to a checkpoint file.
  - If a region and its next region are both updated, the region is "marked"
  - Number of marked regions >= Number of updated regions / 2.
    = Marked regions are in majority of updated regions.
  - The management granularity (the number of pages in a management region) is doubled.

Outline

- Introduction
- Checkpointing methods
- Application-level incremental checkpointing with auto-tuning
- Evaluation and discussions
- Conclusions

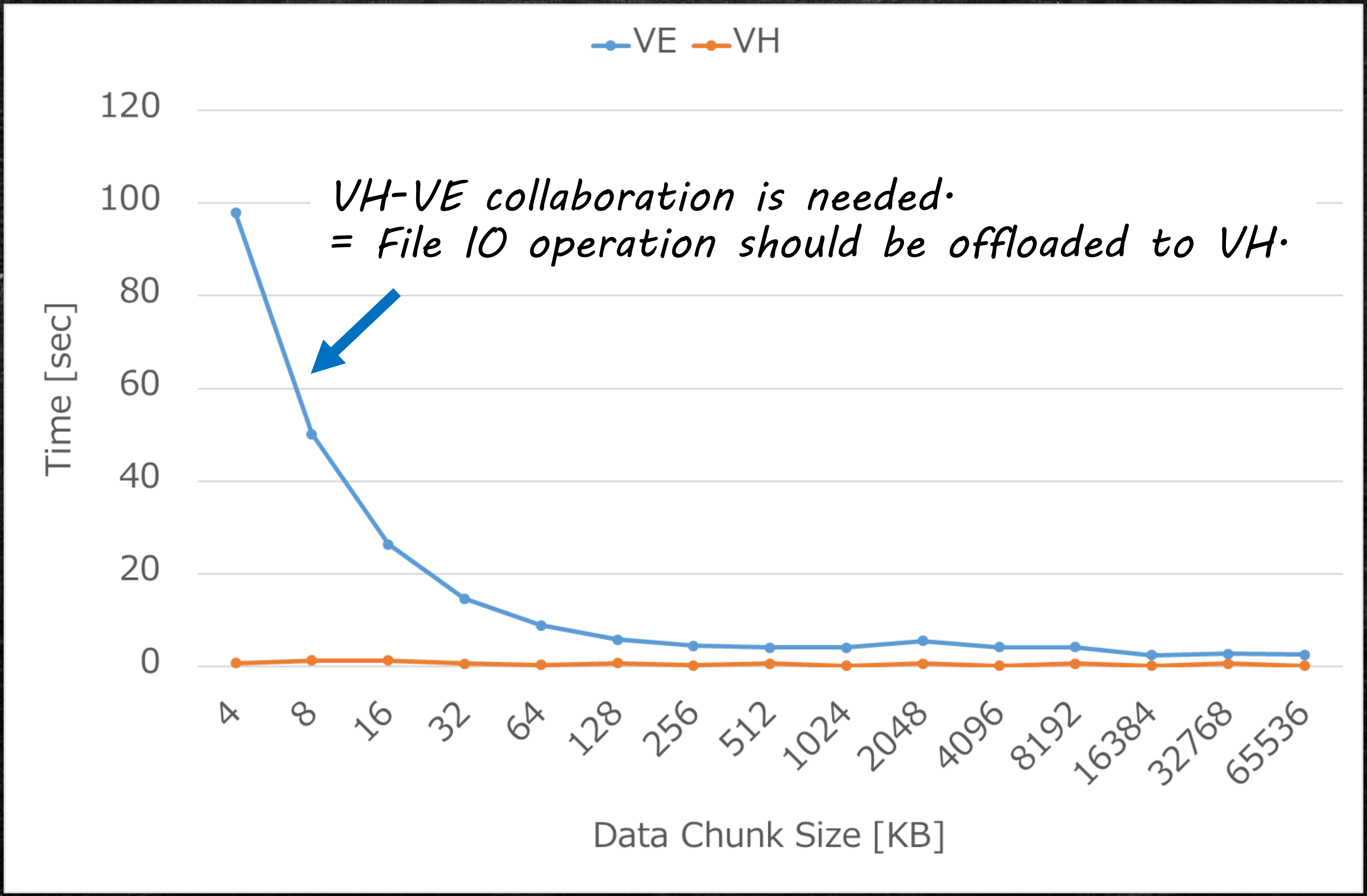
Experimental Setup

- **NEC LX406Re-2 System**
  - Cyberscience Center, Tohoku University
  - Intel Xeon E5-2695v2
  - NEC ScaTeFS file system

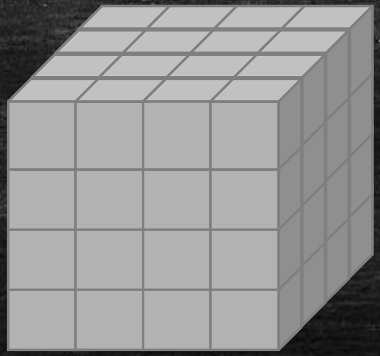


Time for writing 1GB data to a file.
→ Writing small data chunks is slow.
= Larger management granularity is beneficial in terms of file I/O performance.

More significant on SX-AT



Himeno Benchmark

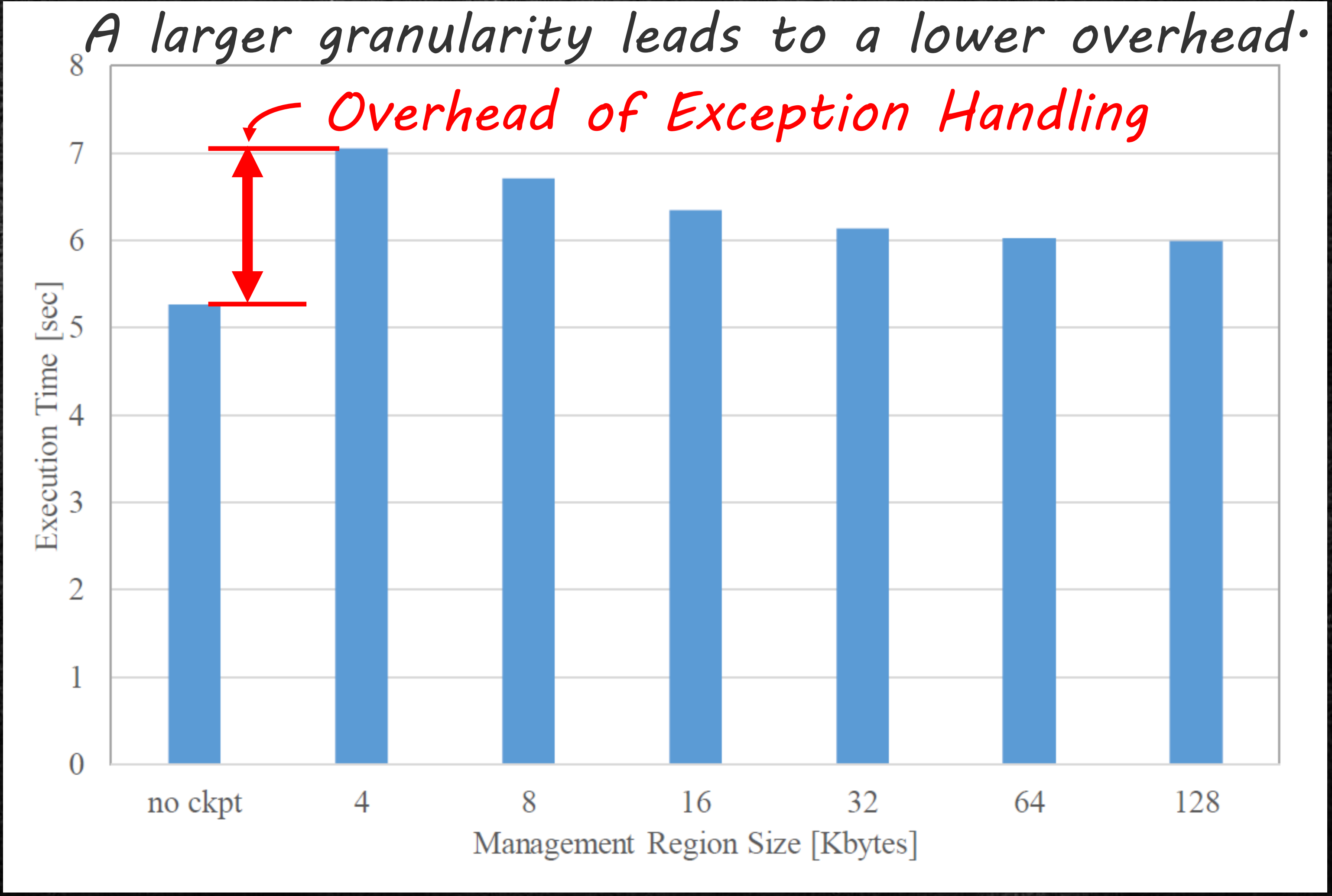


- **3-dimensional Jacobi kernel**
  - *Every element is sequentially updated one by one.*
  - *Checkpoint is taken when a whole slice is updated.*

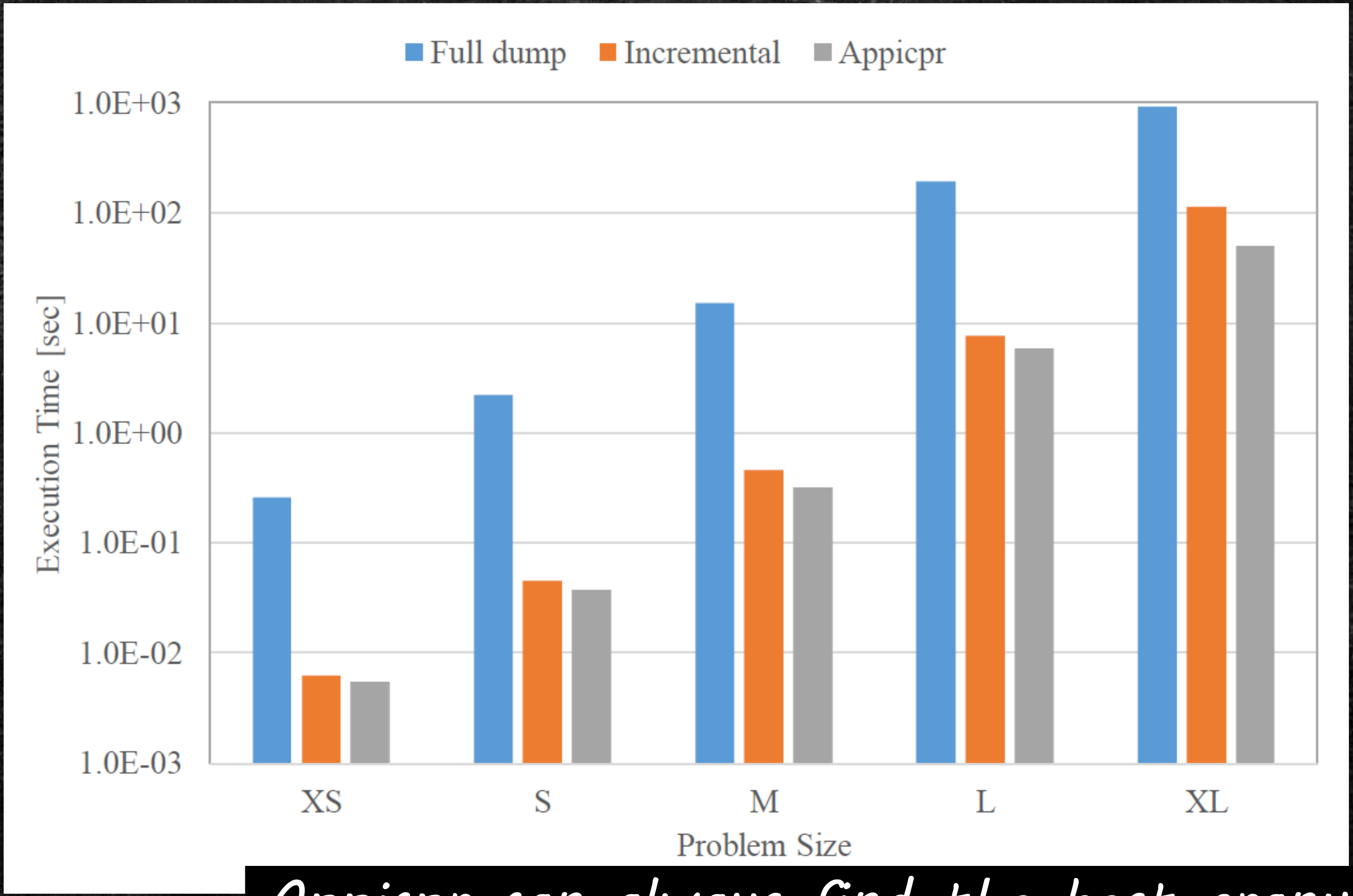
→**Each slice should be one management region.**

= **We have to judge if each slice is updated or not.**

Management Granularity



Performance Evaluation Results



Appicpr can always find the best granularity, and reduce the checkpointing overheads.

Outline

- Introduction
- Checkpointing methods
- Application-level incremental checkpointing with auto-tuning
- Evaluation and discussions
- Conclusions

Conclusions

- **Appicpr**
  - A combination of application-level ckpt, incremental ckpt, and auto-tuning.
  - The reliability can be improved by reducing the ckeckpointing overheads and hence taking checkpoints more frequently.
  - Evaluation results show that the checkpointing overhead can significantly be reduced if only a part of a large array is updated during the checkpointing interval.
- **Future work**
  - In this work, management granularity is monotonically increased, and never decreased, because it is difficult to decide if it should be decreased. This will be further discussed in our future work.
  - Checkpointing interval tuning is considered as well as management granularity.

Energy Cost and Resiliency

- 3-level checkpointing using CheCL:
  - **level-1: RAM ckpt** → can tolerate failures that do not require reboot (**level-1 failures**)
  - **level-2: local disk ckpt** → can tolerate failures that require reboot (**level-2 failures**)
  - **level-3: PFS ckpt** → can tolerate more severe failures (**level-3 failures**)

level-1: RAM ckpt

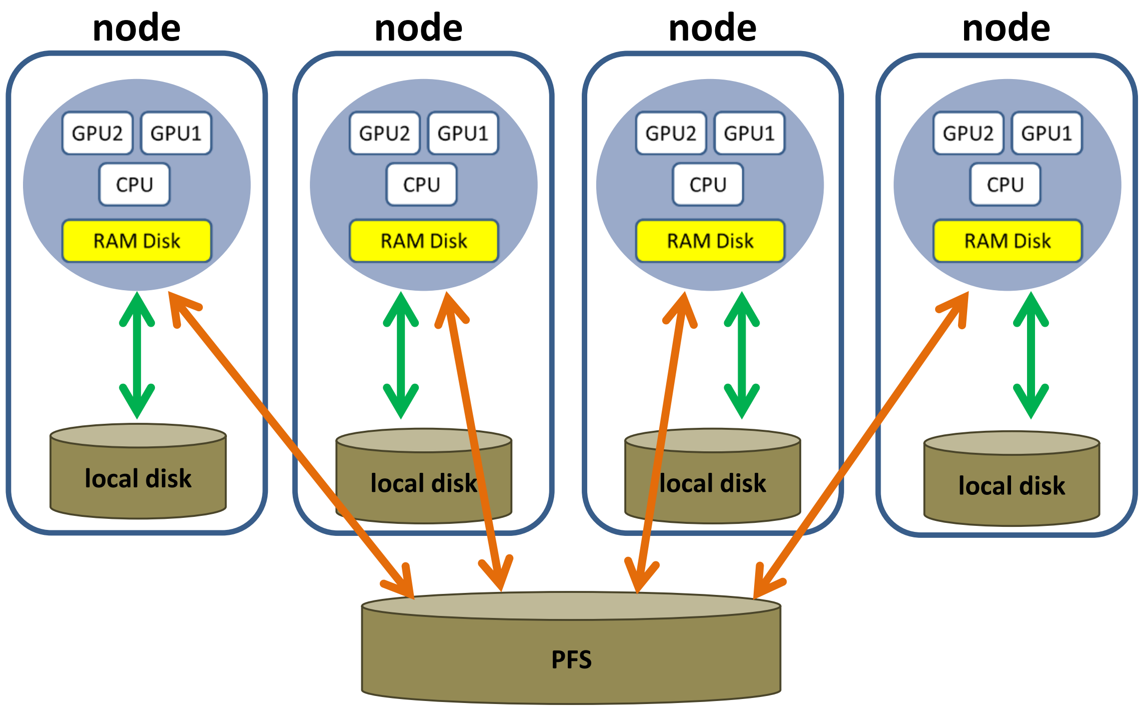
Time →0.35sec/GB
Power →18watt
Energy→6.3J/GB
Recoverable failures: **46%**

level-2: local disk ckpt

Time →9.09sec/GB
Power →8watt
Energy→72.7J/GB
Recoverable failures: **79%**

level-3: PFS ckpt

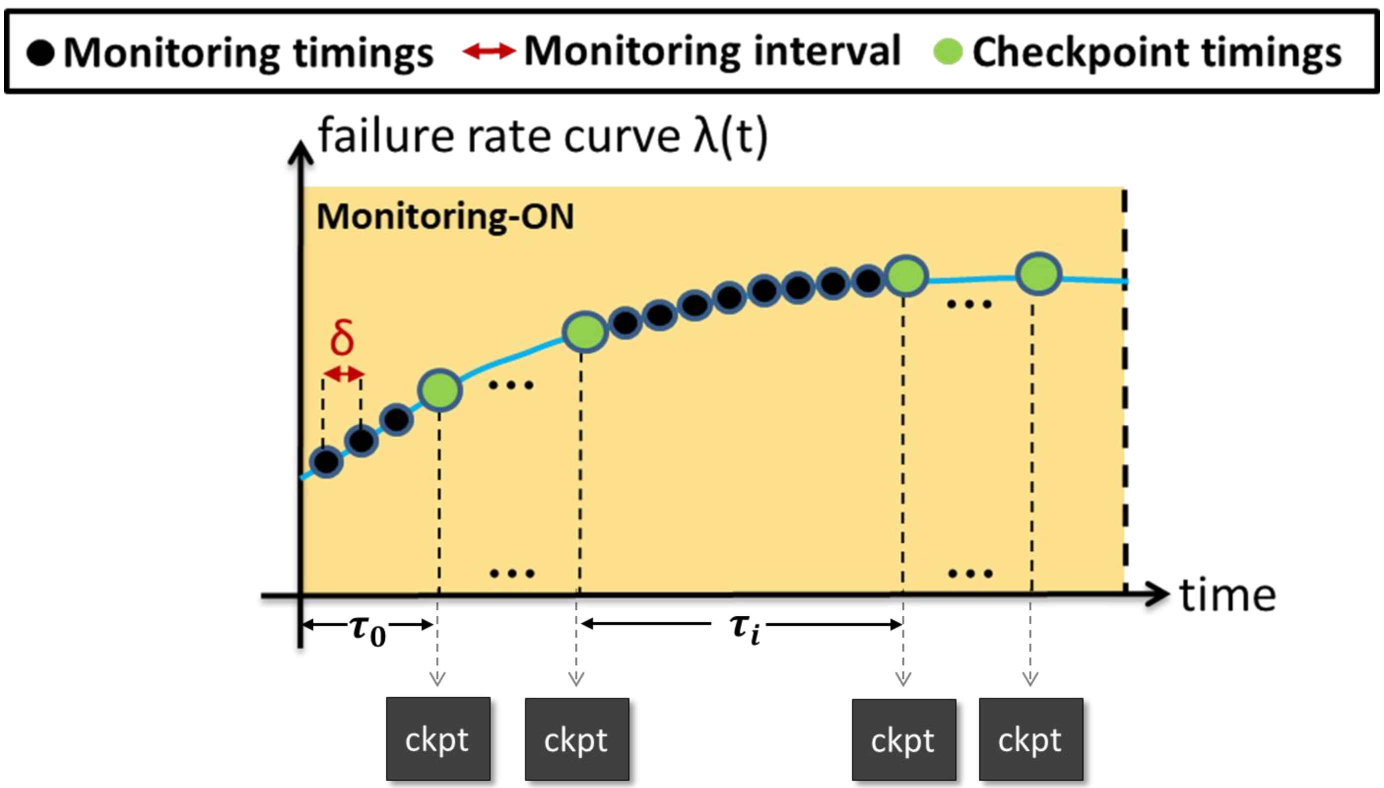
Time →13.79sec/GB
Power →8watt
Energy→110.32J/GB
Recoverable failures: **100%**



Better resiliency comes at the cost of higher energy overhead

Adaptive Checkpointing with Temperature Monitoring

- Temperature monitoring is required for adaptive ckpt
  - Monitor the temperature constantly at an interval $\boldsymbol{\delta}$
  - Translate the temperature data into failure rate $\lambda(t)$
  - Perform runtime analysis to decide optimal checkpoint interval



Problem Statement

- Monitoring overheads could be problematic at large scale
  - I/O overhead, context switches, and interrupts
  - Disk writing overhead for visualization and analysis
  - In Ganglia[6], such overheads are observed to be significant even at smaller scale (42 nodes)

- Trade-off between monitoring overhead and checkpoint interval's optimality
  - Intensive monitoring → ☺ **optimal interval** **BUT** ☹ **large overhead**
  - Less monitoring → ☺ **small overhead** **BUT** ☹ **sub-optimal interval**

- **Aim of this work**: Reduce the reliance on monitoring activities while still maintaining the optimality of checkpoint interval

[6] M. L. Massie et al., The Ganglia Distributed Monitoring System: Design, Implementation, and Experience, Parallel Computing 30(7), 817-840, 2004.

Acknowledgments

- This work was partially supported by JST CREST "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems," DFG SPPEXA ExaFSA project, and Grant-in-Aid for Scientific Research(B) 16H02822.

- The performance evaluation results were obtained using supercomputing resources of the **Cyberscience Center**, Tohoku University.