SAND2022-2099C

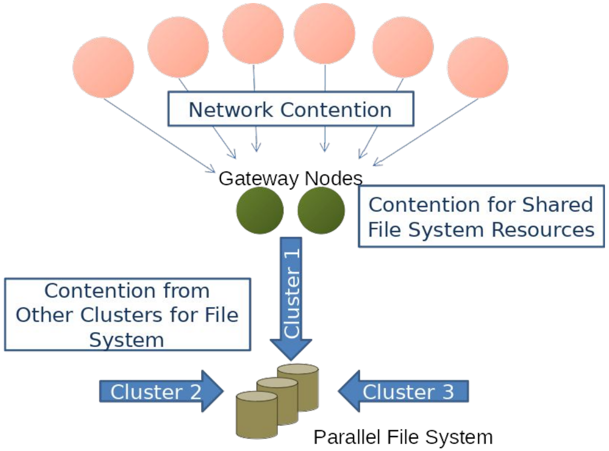
Towards Access Pattern Aware Checkpointing For Kokkos Applications

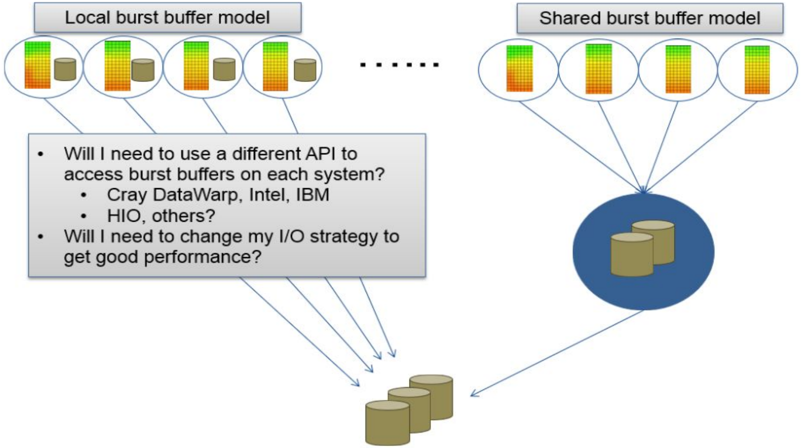
Nigel Tan, Bogdan Nicolae, Nicolas Morales,
Keita Teranishi, Sanjukta Bhowmick,
Franck Cappello, Michela Taufer

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

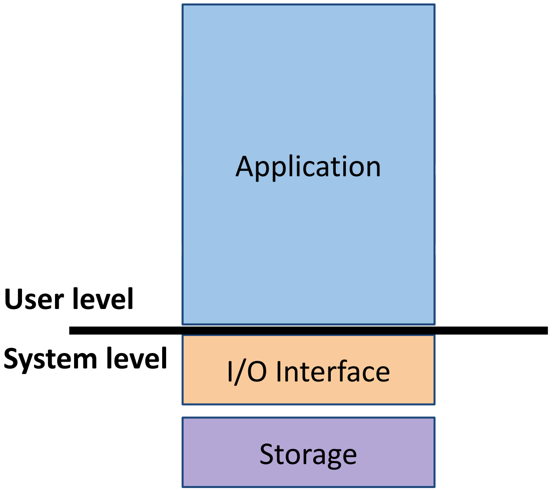
Checkpointing is Difficult at Scale





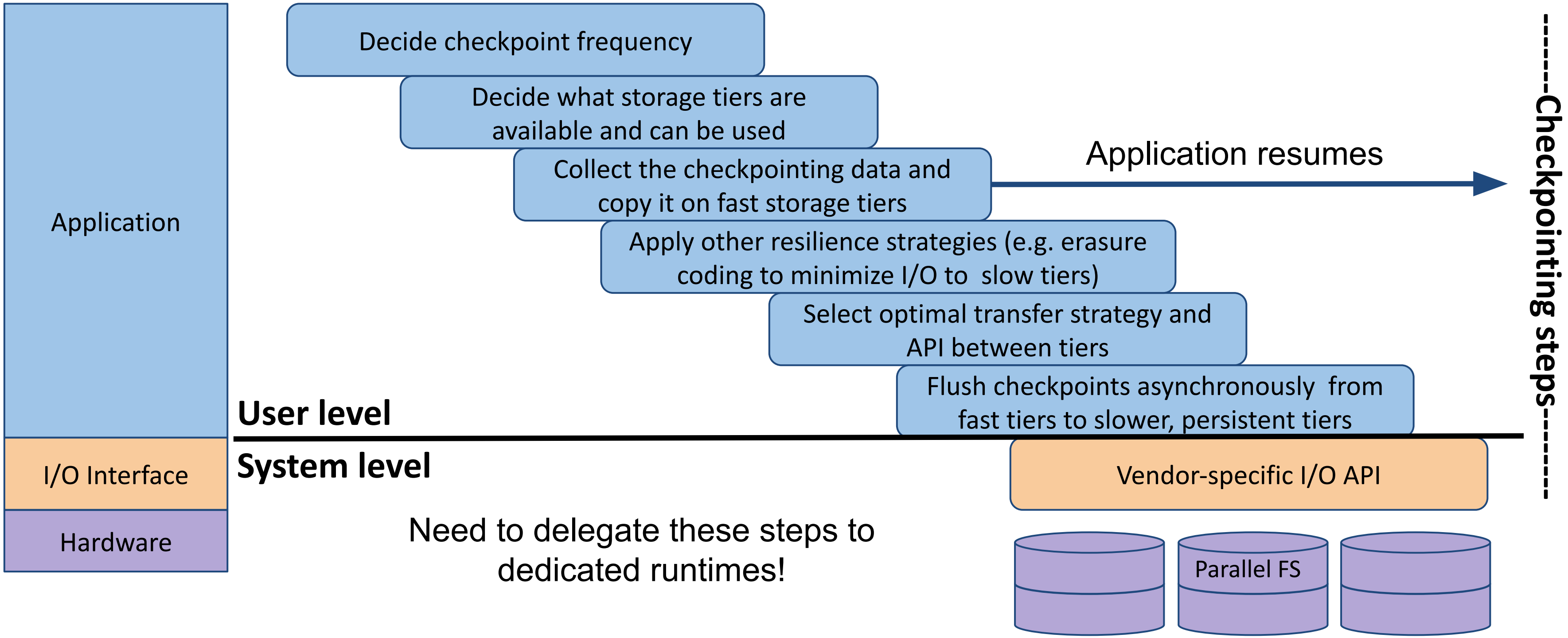
- Performance and scalability
  - Need to checkpoint frequently (failures happen more often, many intermediate datasets, etc.)
  - Many processes, each featuring a large checkpoint size
  - Limited I/O bandwidth available per process due to contention (processes checkpoint simultaneously)
- Heterogeneity and complexity of storage hierarchy
  - Many options in addition to PFS: burst buffers, object stores, caching layers, etc.
  - Many vendors, each with its own API

Naive Checkpointing is Unfeasible



- Writing checkpoints directly to the parallel file system (naive checkpointing) incurs unacceptable overheads
- Why do users still do it?
  - Not aware of the complexity of the storage stack (are other storage tiers are available?)
  - Not aware how to leverage storage tiers efficiently (e.g. async flushing from fast to slow tiers)
- Even if users are knowledgeable, the development effort is overwhelming
  - Too many combinations of storage tiers and strategies, each with their own performance model and/or API

Efficient Checkpointing is Complex



State of Art

Very Low Overhead Checkpointing (VeloC)

Performance portability framework

- Implement powerful execution and memory abstractions for developing fast portable applications across modern platforms
- Provide useful containers such as multidimensional arrays (Views) that support CPUs and GPUs

https://kokkos.org

Asynchronous checkpoint-restart runtime

- Delivers efficient and scalable asynchronous checkpointing using complex heterogeneous storage hierarchies
- Flexible modular architecture to support a large variety of strategies and vendor APIs

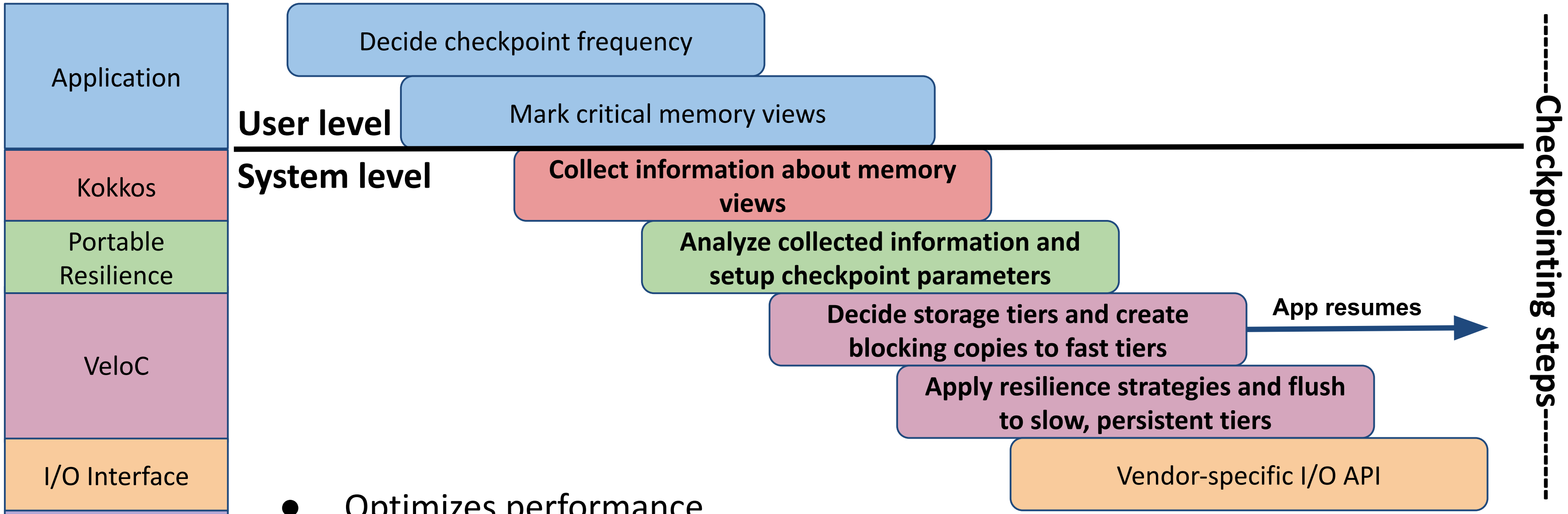
https://veloc.readthedocs.io

Portable Resilience: combines Kokkos abstractions (memory views) with VELOC abstractions (protected memory regions) to enable an efficient performance-portable resilience runtime

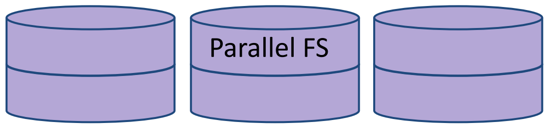
Morales, N., Teranishi, K., Nicolae, B., Trott, C. and Cappello, F. 2021.Towards High Performance Resilience Using Performance Portable Abstractions. *EuroPar'21: 27th International European Conference on Parallel and Distributed Systems* (Lisbon, Portugal, 2021).

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations -- the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. It is also supported by the National Science Foundation (NSF) under Grants CCF-1617488, CCF-1619253, OAC-2003709, OAC-1948447/2034169, and OAC-2003624/2042084.

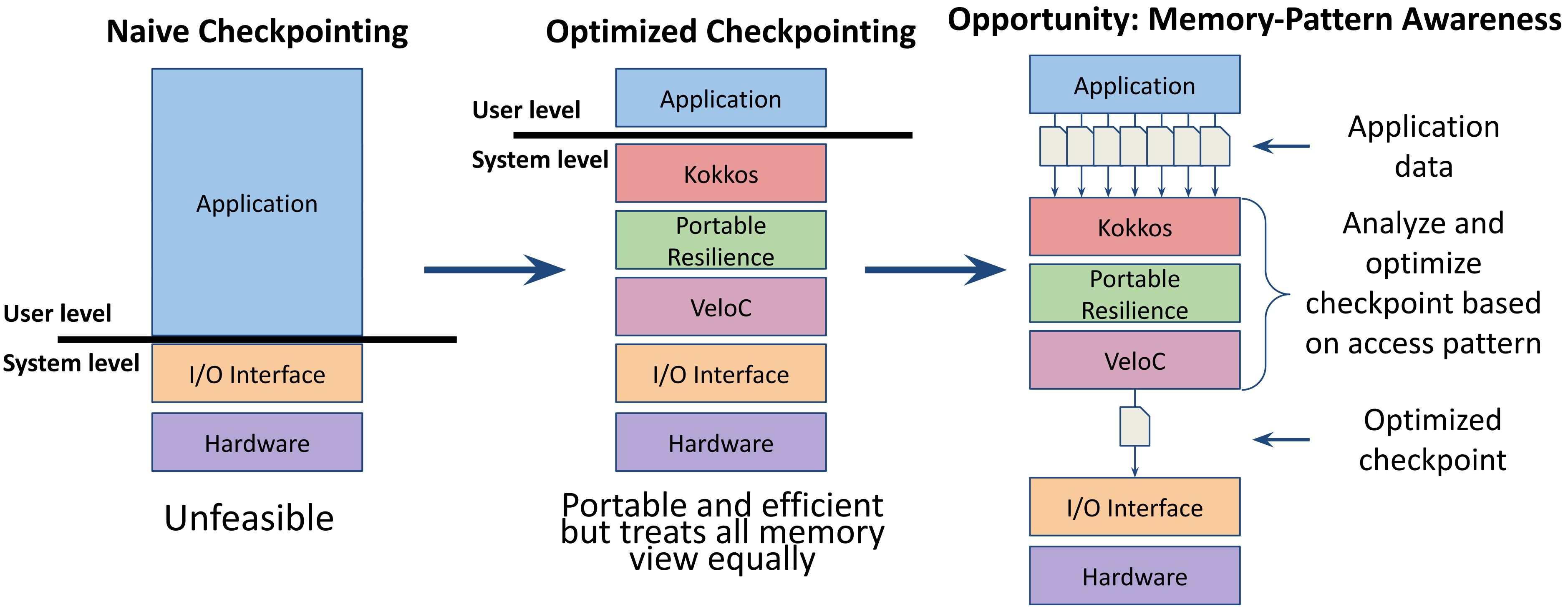
Kokkos+VeloC: Portable+Efficient Ckpt



- Optimizes performance
- Minimizes user effort at application-level



Goal: Memory-Pattern Aware Ckpt



Memory Pattern Aware Checkpointing

We look at memory access patterns for opportunities to improve checkpointing

System level

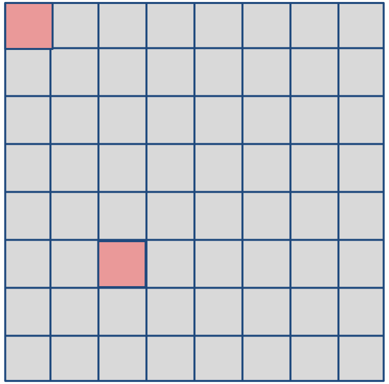
Kokkos — Collect memory pattern information

Kokkos Resilience — Analyze patterns and generate hints

VeloC — Optimize checkpointing based hints

Tightly couple the software layers to enable more efficient checkpointing based on runtime properties

Example: Sparse Update Patterns

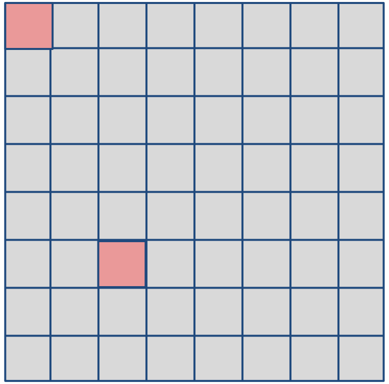


Step i

Updated cell

Unchanged cell

Example: Sparse Update Patterns II



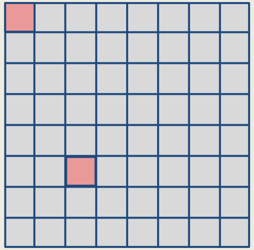
Step i

Updated cell

Unchanged cell

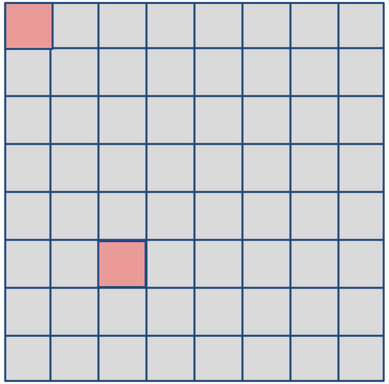
Full Checkpoint

Checkpoints everything!



Total Checkpoint
64 cells

Example: Sparse Update Patterns III



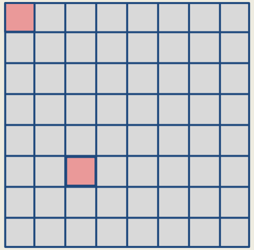
Step i

Updated cell

Unchanged cell

Full Checkpoint

Checkpoints everything!



Total Checkpoint
64 cells

Incremental Checkpoint

Checkpoint ONLY the changes

Row	0	5
Col	0	2

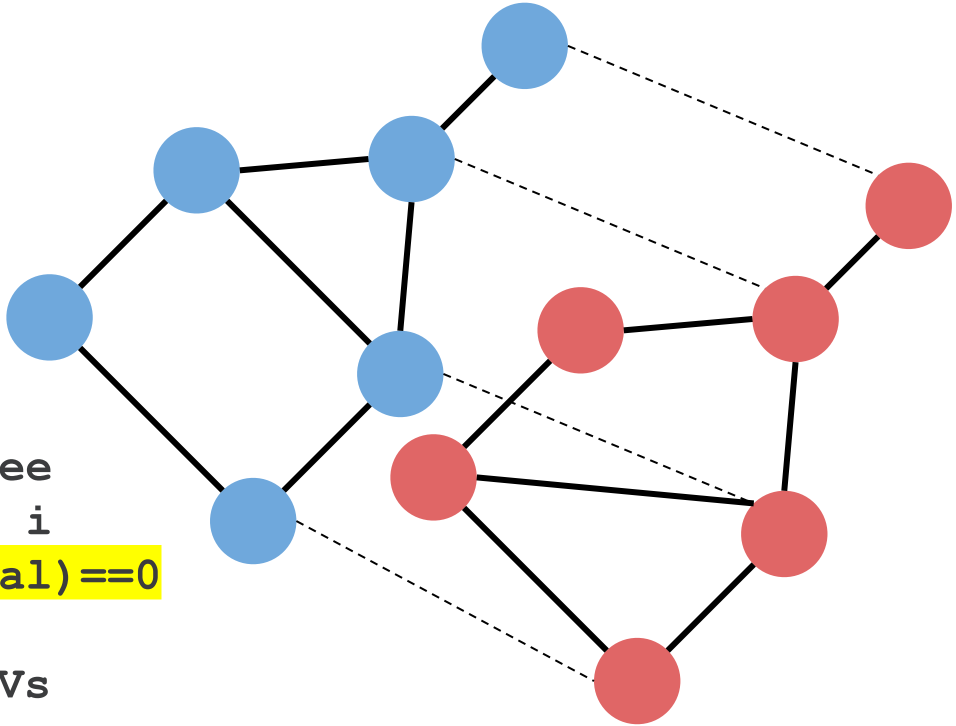
Checkpoint Size: 6

Example App: Fido (Graph Alignment)

- Compare two graphs based on common substructures called graphlets

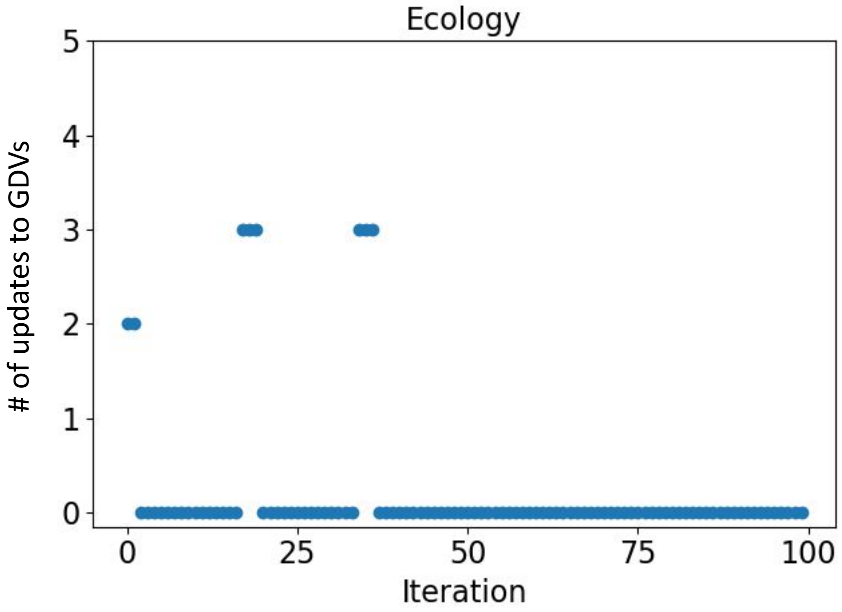
Algorithm:

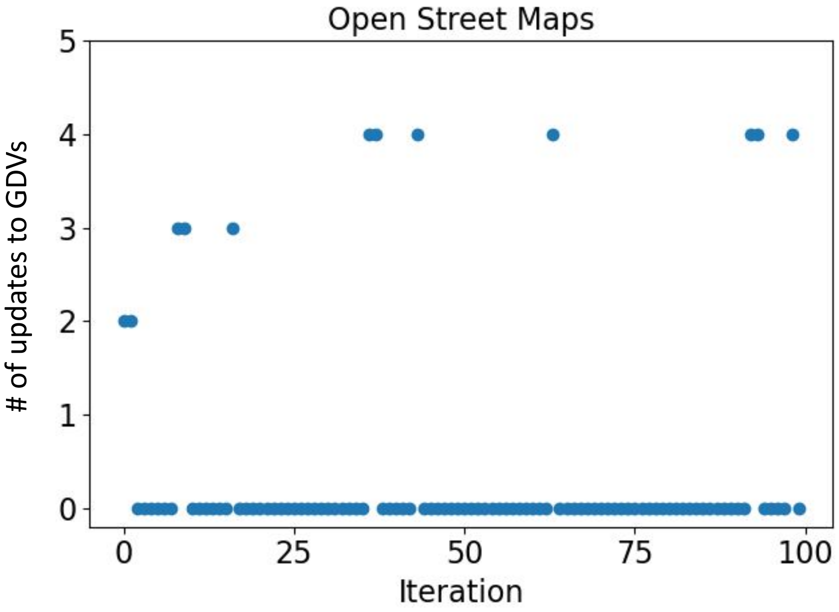
```
For each graph
    For i in [0,num_vertices]
        Calculate graphlet degree
        vector (GDV) for vertex i
        If (i%checkpoint_interval)==0
            Checkpoint GDVs
Match vertices with similar GDVs
```



Sparse Update Pattern in Fido

We track the memory access pattern for two sets of graphs (i.e., Ecology and Open Street Maps)

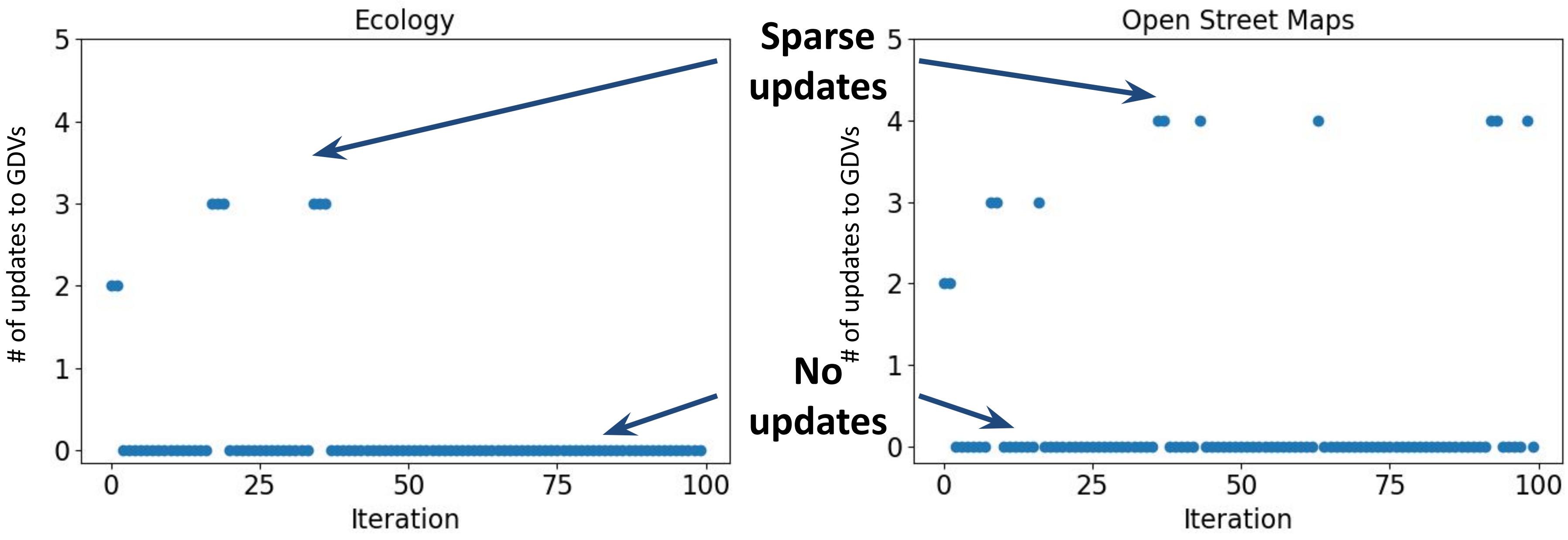




Graphs from SuiteSparse collection: sparse.tamu.edu

Sparse Update Pattern in Fido

We track the memory access pattern for two sets of graphs (i.e., Ecology and Open Street Maps)



Graphs from SuiteSparse collection: sparse.tamu.edu

Incremental Checkpointing for Sparse Updates

Goal: Instead of full checkpoints, write only diffs to previous checkpoints

Challenge: Faster I/O due to smaller ckpt size, but additional overheads

System level

System level	**Collect application information at runtime**	Provide hooks into the applications memory access pattern
Additional overheads	**Analyze collected information and generate checkpointing hints**	Detect whether data is sparsely updated
	Adjust memory layout	Restructure data for checkpointing
	Select granularity and scope of diffs	Block size, reference checkpoints
	Deduplicate	Eliminate redundant data from checkpoint
	Apply checkpointing strategies	

Incremental Checkpoints for Sparse Updates

Goal: Collect pattern information from the application

Challenge: Must not require any code changes by the user

System level	**Collect application information at runtime**	Provide hooks into the applications memory access pattern
Additional overheads	**Analyze collected information and generate checkpointing hints**	Detect whether data is sparsely updated
	Adjust memory layout	Restructure data for checkpointing
	Select granularity and scope of diffs	Block size, reference checkpoints
	Deduplicate	Eliminate redundant data from checkpoint
	Apply checkpointing strategies	

Incremental Checkpoints for Sparse Updates

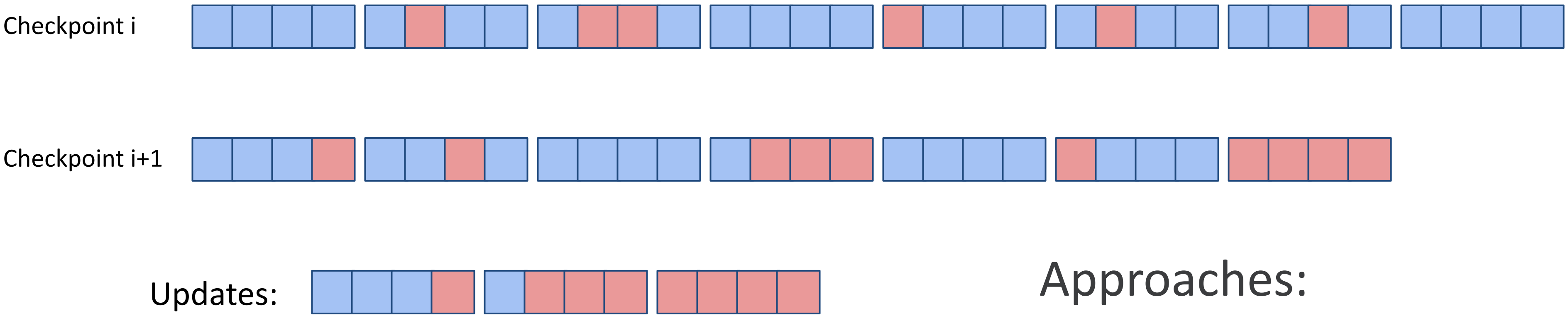
Goal: Find differences between checkpoints

Challenge: Analysis overhead

System level

System level	**Collect application information at runtime**	Provide hooks into the applications memory access pattern
	Analyze collected information and generate checkpointing hints	Detect whether data is sparsely updated
Additional overheads	**Adjust memory layout**	Restructure data for checkpointing
	Select granularity and scope of diffs	Block size, reference checkpoints
	Deduplicate	Eliminate redundant data from checkpoint
	Apply checkpointing strategies	

Detecting Updates

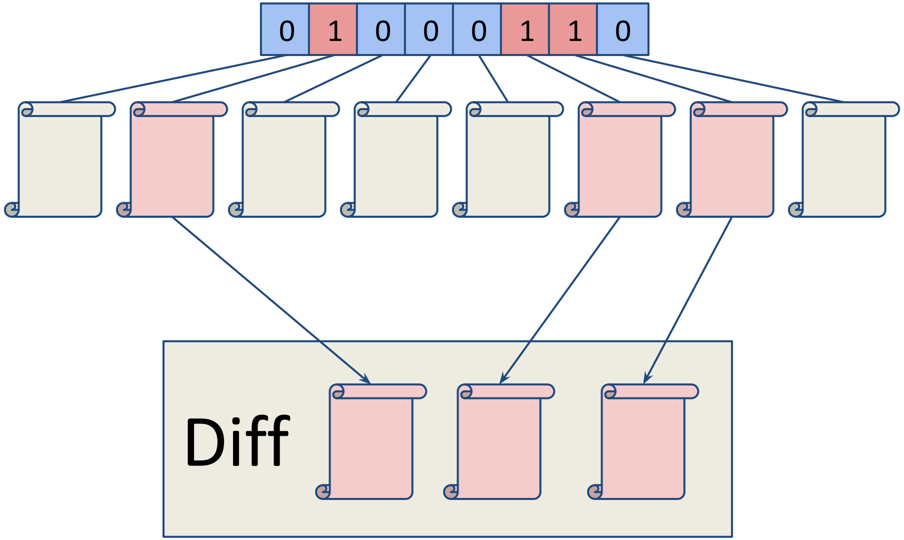


How do we efficiently identify the differences between the current and all previous checkpoints?

Approaches:

- **Dirty page tracking**
- **Naive scanning**
- **Hash based**

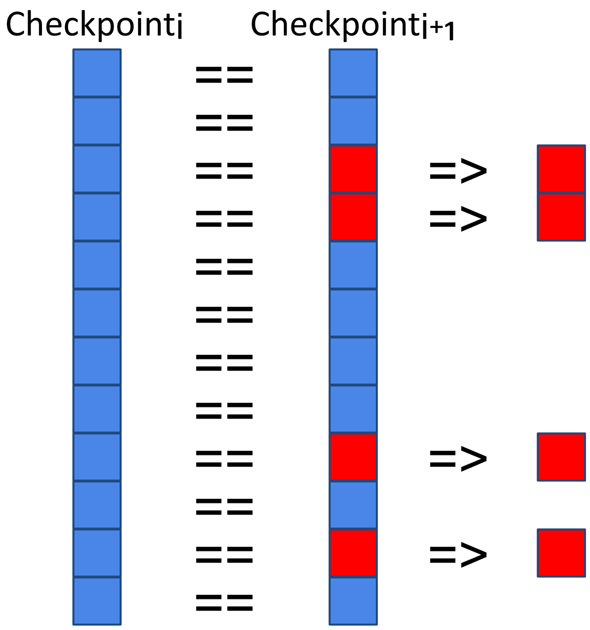
Dirty Page Tracking



OS tracks which pages have been written to.
Check pagemap for dirty pages.

- Fast, no computation or comparisons
- Tracking is automatically done by the OS
- Requires up-to-date kernel
- Coarse grain (4-64KB)
- Does not work for GPU based applications

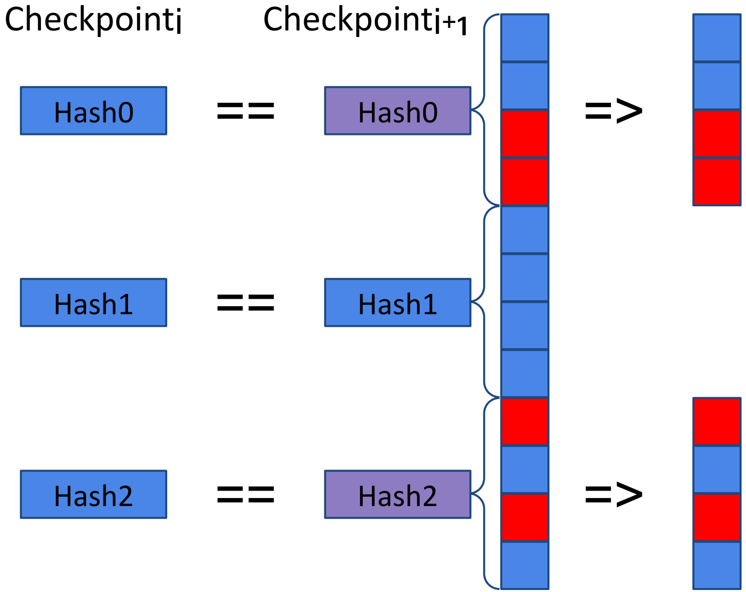
Scanning For Updates



Naive scanning: Scan data at checkpoint i and i+1 for differences

- High overhead due to scanning through multiple checkpoints
  - Potentially many checkpoints
- Checkpoints are large
  - Large overhead from many comparisons

Hash Based Methods



Hash based: Divide checkpoint into blocks, compute hashes for each block and compare hashes between checkpoint i and i+1

- Extra computation cost for hashes
- Tradeoff based on block size
  - Smaller blocks => smaller checkpoint
  - Larger blocks => faster to find differences
- Tradeoff based on hash function
  - Strong hash => less collisions
  - Weak hash => faster computation

Primary method we use for this work

Incremental Checkpoints for Sparse Updates

Goal: Create an incremental checkpoint from application data

Challenge: Additional overhead must not exceed savings from smaller checkpoint

System level

- **Collect application information at runtime** — Provide hooks into the applications memory access pattern
- **Analyze collected information and generate checkpointing hints** — Detect whether data is sparsely updated
- Additional overheads:
  - **Adjust memory layout** — Restructure data for checkpointing
  - **Select granularity and scope of diffs** — Block size, reference checkpoints
  - **Deduplicate** — Eliminate redundant data from checkpoint
- **Apply checkpointing strategies**

Incremental Checkpoints for Sparse Updates

Challenge: Memory layout may not be optimal for checkpointing

Tradeoff: Cost to adjust memory layout vs cost to create incremental checkpoints

System level

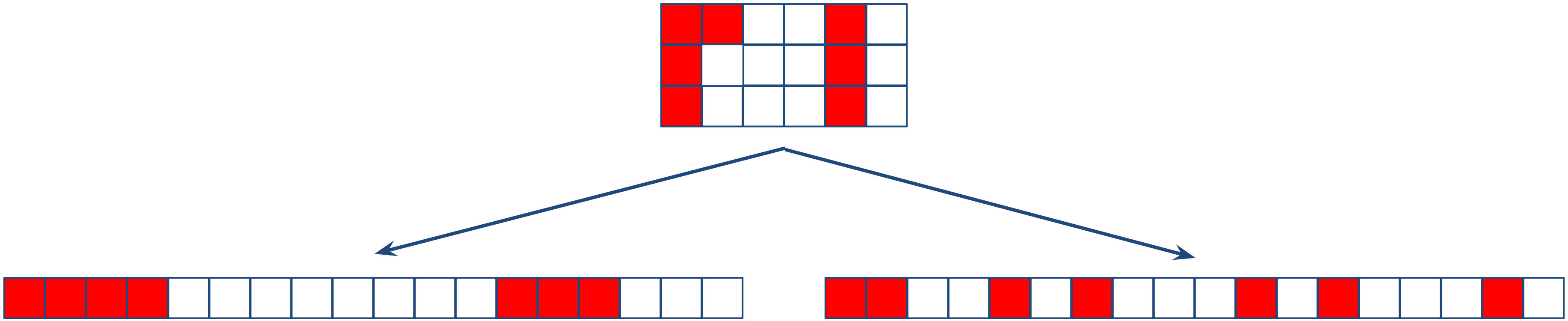
- **Collect application information at runtime** — Provide hooks into the applications memory access pattern
- **Analyze collected information and generate checkpointing hints** — Detect whether data is sparsely updated

Additional overheads:

- **Adjust memory layout** — Restructure data for checkpointing
- **Select granularity and scope of diffs** — Block size, reference checkpoints
- **Deduplicate** — Eliminate redundant data from checkpoint
- **Apply checkpointing strategies**

Memory Layout

We want to reorganize memory such that updates are contiguous for better checkpoint performance



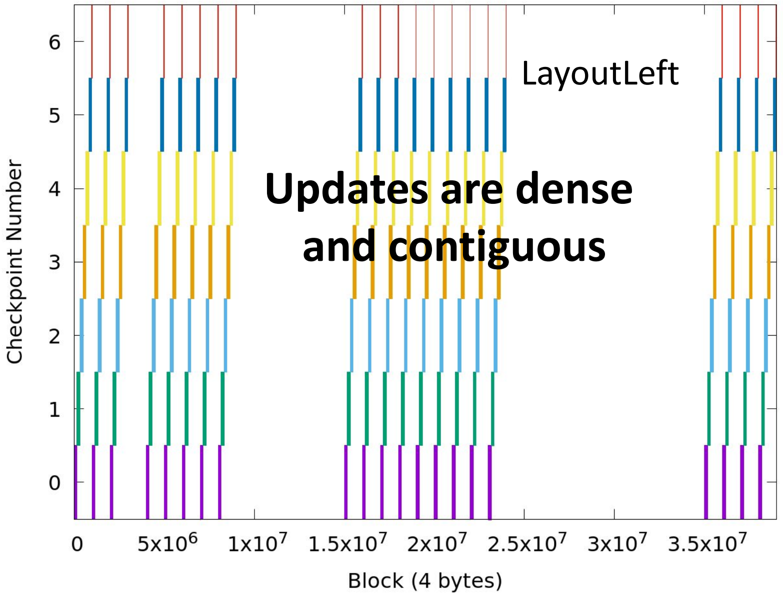
LayoutLeft (Column major order)

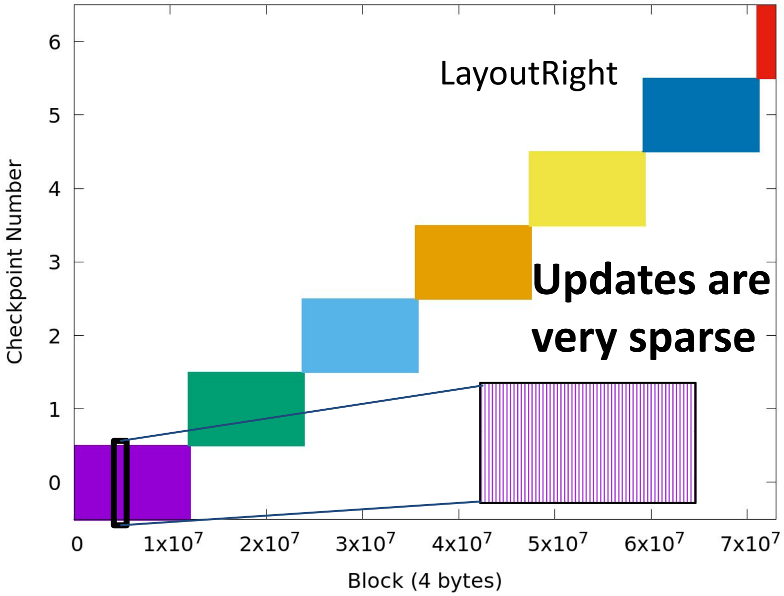
LayoutRight (Row major order)

LayoutLeft is preferred in this example (updates are dense and contiguous)

Kokkos
Kokkos Resilience
VeloC

LayoutLeft vs LayoutRight: Ecology Graphs





For the Ecology Graphs, LayoutLeft leads to updates being grouped closer together (dense and contiguous). Other graphs may exhibit different behavior.

Graphs from SuiteSparse collection: sparse.tamu.edu

Incremental Checkpoints for Sparse Updates

Challenge: Small block size leads to smaller checkpoints but increases overhead compared to larger block sizes

Tradeoff: Checkpoint size vs cost to find differences

System level

- **Collect application information at runtime** — Provide hooks into the applications memory access pattern

Additional overheads:

- **Analyze collected information and generate checkpointing hints** — Detect whether data is sparsely updated
- **Adjust memory layout** — Restructure data for checkpointing
- **Select granularity and scope of diffs** — Block size, reference checkpoints
- **Deduplicate** — Eliminate redundant data from checkpoint
- **Apply checkpointing strategies**

Kokkos

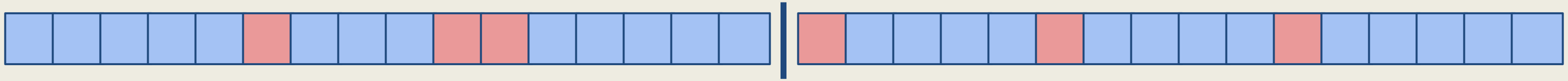
Kokkos Resilience

VeloC

Update Detection Granularity

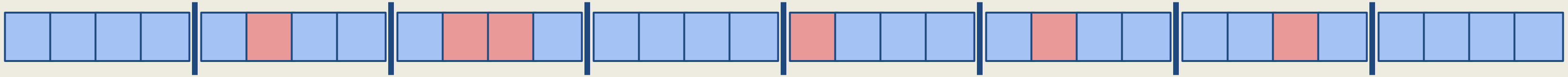
Checkpoint data

Coarse granularity (2 chunks)



Lots of unnecessary data but fewer comparisons to detect updates

Fine granularity (8 chunks)



Less unnecessary data but more comparisons to detect updates

Tradeoff: Checkpoint size vs update detection cost

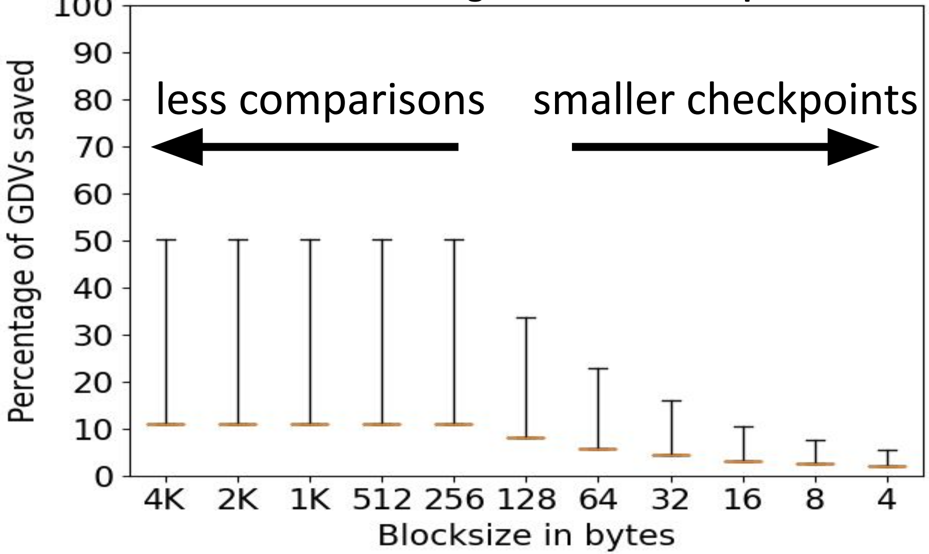
Kokkos

Kokkos Resilience

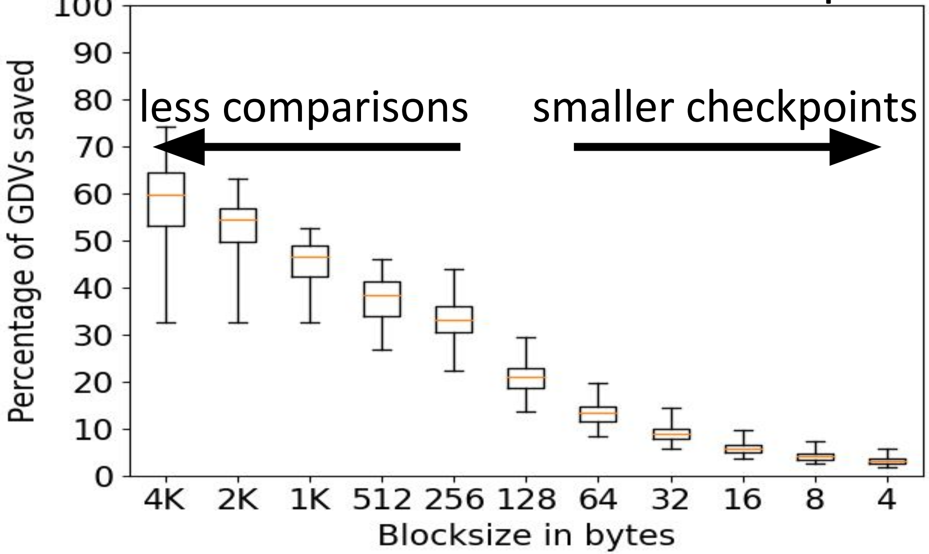
VeloC

Checkpoint Size vs Update Detection Cost

Problem 1: Message Race Event Graph



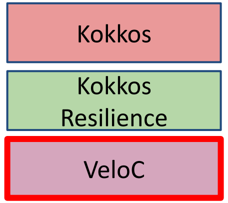
Problem 2: Unstructured Mesh Event Graph

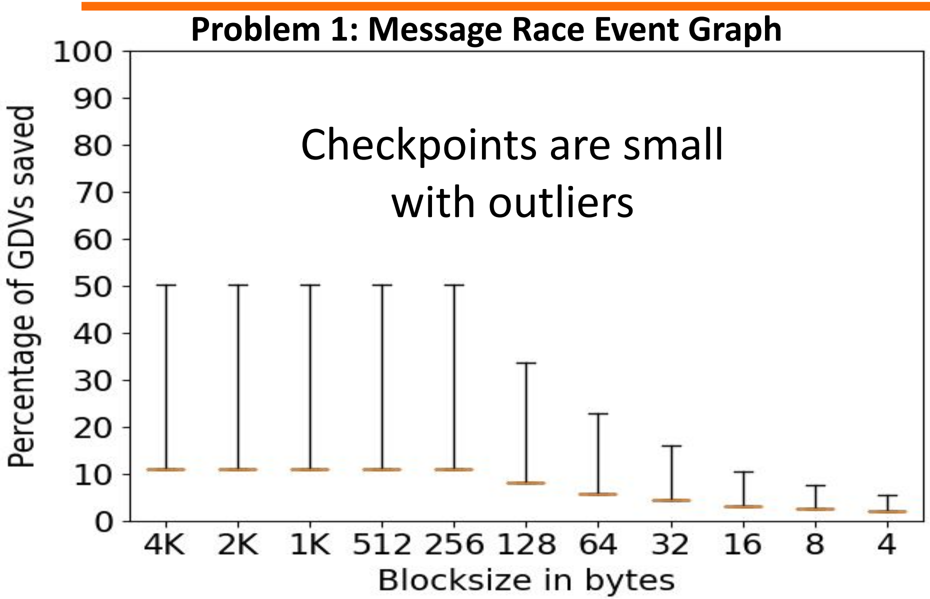


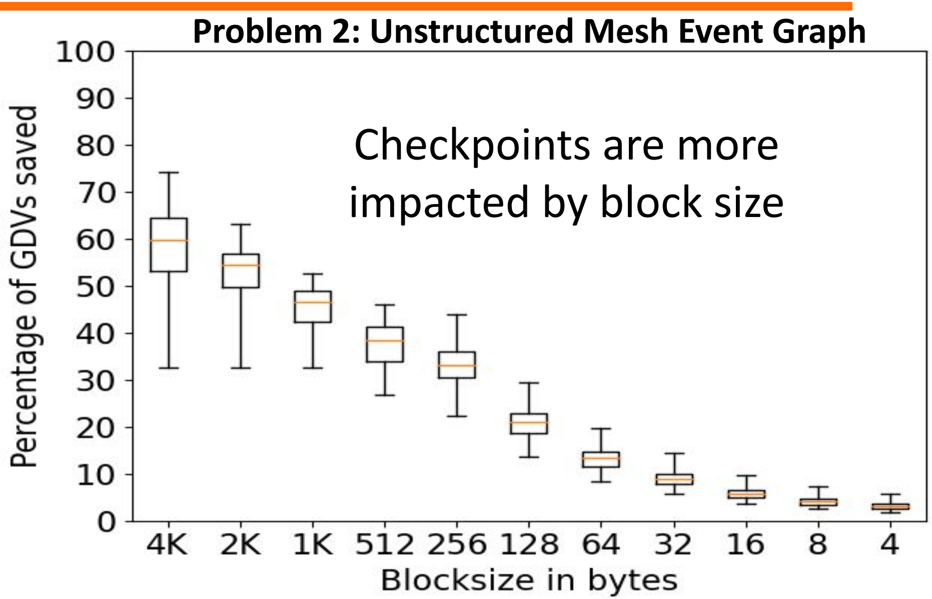
Smaller block sizes lead to smaller checkpoints but need more comparisons to find differences

Message race and unstructured mesh are graphs modeling point-to-point communication patterns

Checkpoint Size vs Update Detection Cost







Different problems have different checkpoint similarities

Message race and unstructured mesh are graphs modeling point-to-point communication patterns

Incremental Checkpoints for Sparse Updates

Challenge: Minimize incremental checkpointing overhead

Tradeoff: CPU vs GPU deduplication for minimizing application stalls

System level	**Collect application information at runtime**	Provide hooks into the applications memory access pattern
Additional overheads	**Analyze collected information and generate checkpointing hints**	Detect whether data is sparsely updated
	Adjust memory layout	Restructure data for checkpointing
	Select granularity and scope of diffs	Block size, reference checkpoints
	Deduplicate	Eliminate redundant data from checkpoint
	Apply checkpointing strategies	

Checkpoint Deduplication

Kokkos

Kokkos Resilience

VeloC

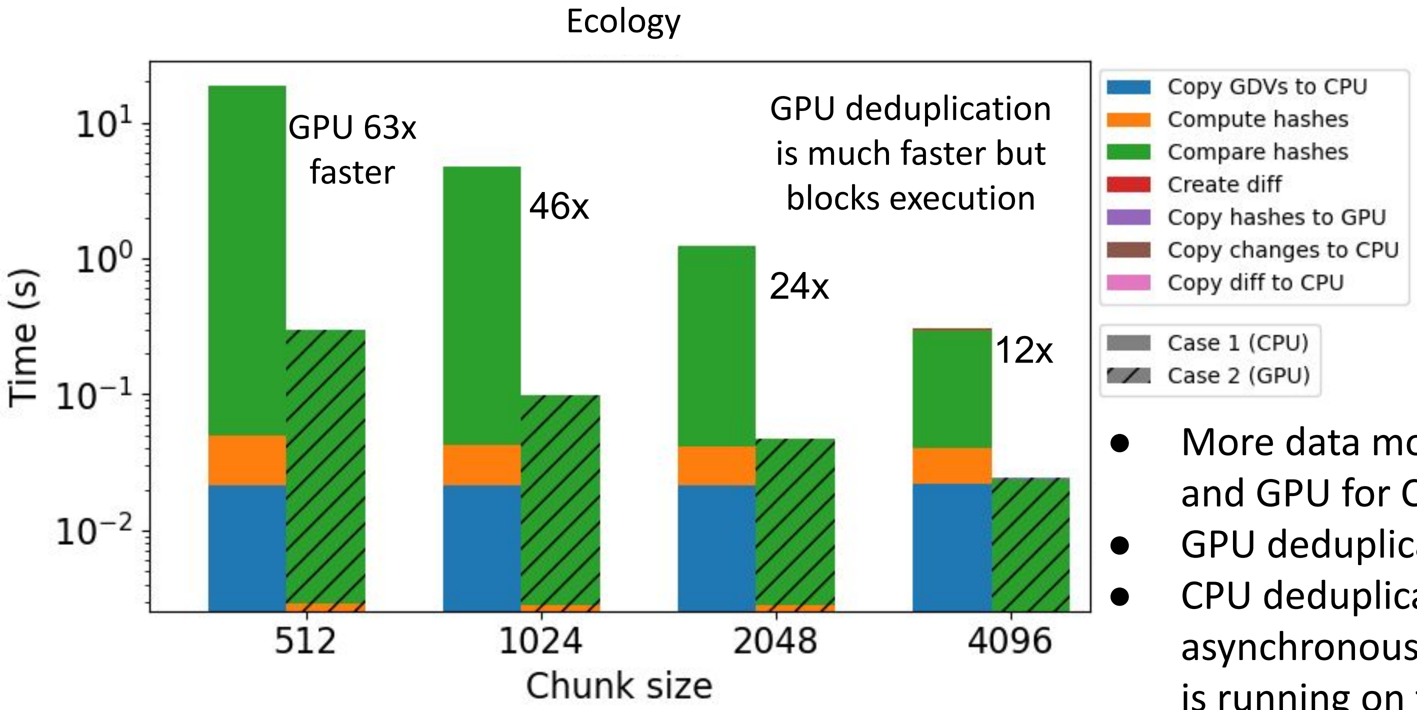
How to eliminate redundant data while minimizing application stalls?

Compute hashes	Divide data into chunks and calculate hashes
Compare hashes	Compare hashes to find updated chunks
Create diff	Collect chunks into a contiguous buffer for checkpointing

Two cases for mixed CPU-GPU applications

- Perform deduplication on the CPU
- Perform deduplication on the GPU

Deduplication Performance on CPU and GPU



Deduplication strategy must minimize the overall checkpoint overhead by minimizing stalls to the application

- More data movement between CPU and GPU for CPU deduplication
- GPU deduplication is blocking
- CPU deduplication can be done asynchronously while the application is running on the GPU

Summary and Future Work

Summary

- Combining Kokkos and VeloC with the Kokkos Resilience layer enables access pattern aware checkpoints for improving checkpoint performance
- Incremental checkpoints can drastically reduce the size of checkpoints for sparsely updated data
- There are trade-offs between checkpoint size and complexity that depend on the update pattern

Future work

- Tightly couple the software layers (Kokkos, Kokkos Resilience, VeloC) for greater performance
- Investigate other access patterns

Acknowledgements

This work was supported in part by the National Science Foundation under Grant #1900888 and Grant #1900765. The authors acknowledge IBM through a Shared University Research Award.

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations -- the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. It is also supported by the National Science Foundation (NSF) under Grants CCF-1617488, CCF-1619253, OAC-2003709, OAC-1948447/2034169, and OAC-2003624/2042084.

kokkos VeloC ECP EXASCALE COMPUTING PROJECT

https://kokkos.org https://veloc.readthedocs.io