

# AI-Ckpt: Leveraging Memory Access Patterns for Adaptive Asynchronous Incremental Checkpointing

Bogdan Nicolae, Franck Cappello

IBM Research  
Ireland

Argonne National Lab  
USA



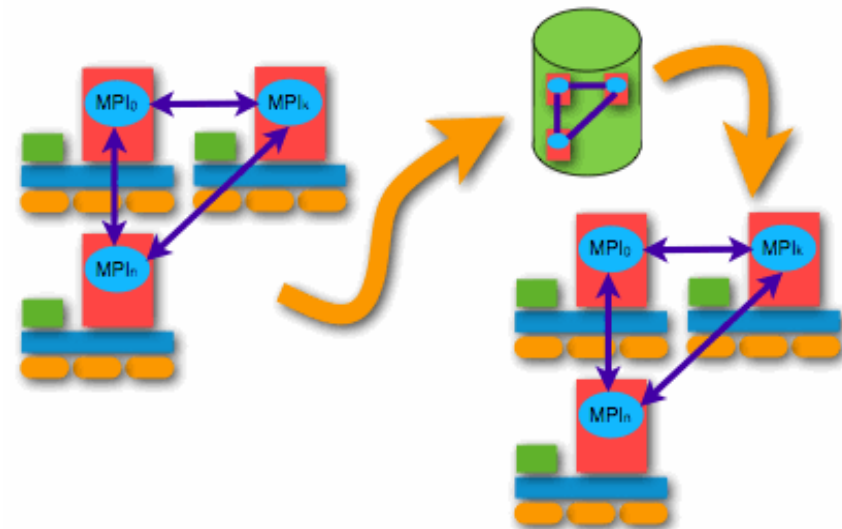
## Outline

- Overview on Checkpoint-Restart
- A case for asynchronous approaches
- Proposal: an asynchronous scheme that learns from past memory access patterns and adapts to current patterns in order to flush memory pages in an optimized order
- Implementation details
- Evaluation
- Conclusions



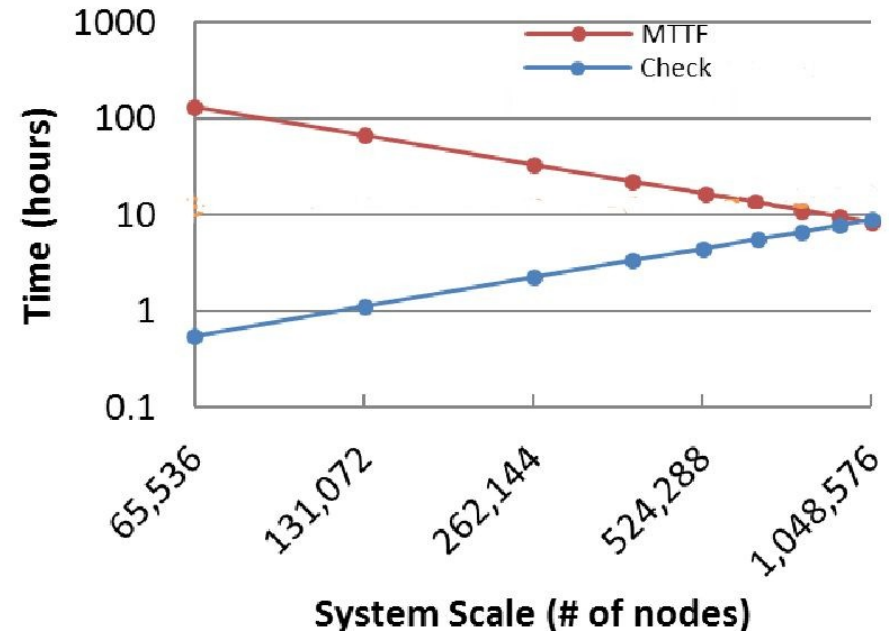
## Checkpoint-Restart: a quick overview

- Focus area: HPC applications
  - Large scale (heading to exascale)
  - Tightly coupled (MPI)
  - Long execution time (days)
- Failures are frequent and hard to overcome
- HPC Checkpoint-Restart
  - Saves a globally consistent state periodically (memory of processes)
  - Enables fault tolerance
  - ...but also
    - Migration
    - Debugging



## Checkpoint-Restart: how to keep it scalable?

- What are the current limitations?
  - Blocking writes
  - Too much checkpointing data
  - Too much coordination
  - I/O bottlenecks due to concurrency
- Directions
  - **Asynchronous techniques**
  - Reduction of checkpointing data
  - Protocols with less coordination
  - Leverage local storage (and make it resilient)



Source: D Zhao et al, Exploring Reliability of Exascale Systems through Simulations, HPC'13

## How to enable asynchronous checkpointing

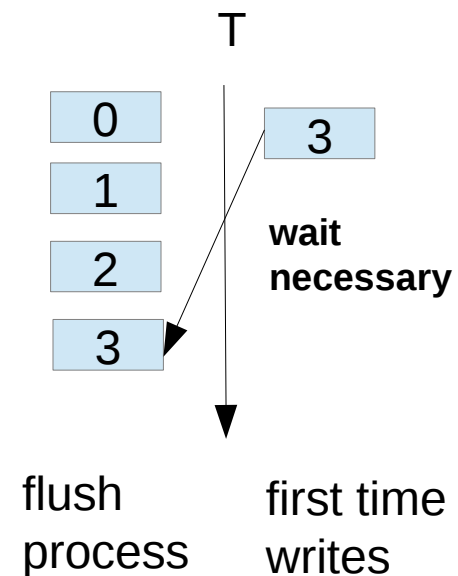
- State of art:
  - Full copy, then flush in the background
    - High copy overhead
    - High extra memory utilization
    - No synchronization overhead
  - Copy-on-write (using page tracking)
    - Less copy overhead
    - High extra memory utilization
    - No synchronization overhead but monitoring overhead
  - Zero-copy
    - No copy overhead or extra memory utilization
    - High synchronization overhead
- What we ideally want: minimize memory utilization without paying too much for synchronization/copy overhead



## Zoom on zero-copy: why synchronization matters

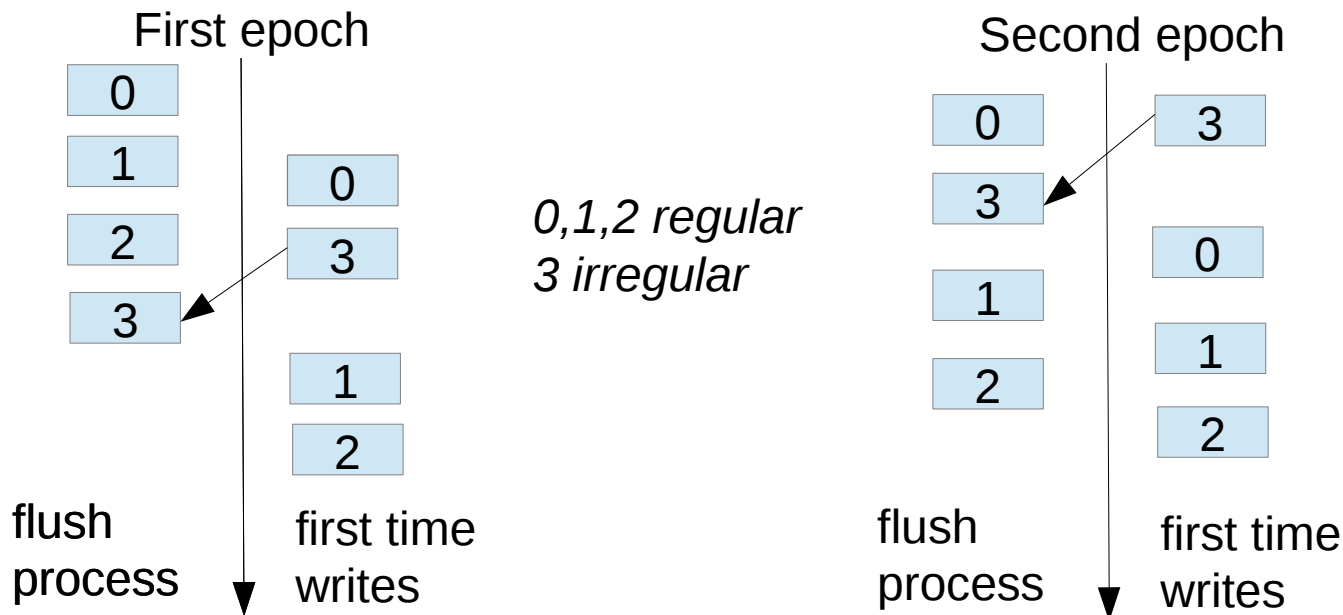
- Consistency requirement: after checkpoint request, flush memory page before first time write
- Consequence: order of flushing matters
- Obvious solution: if a write needs to wait, change order to flush as soon as possible
  - Can we do better?
- Key idea of this paper: HPC apps are iterative in nature, thus we expect first time writes to be predictable
  - Assumption: first time writes are almost periodic (we call the period “epoch”)
  - For simplicity, we assume the epoch corresponds to the checkpoint interval

Worst case scenario



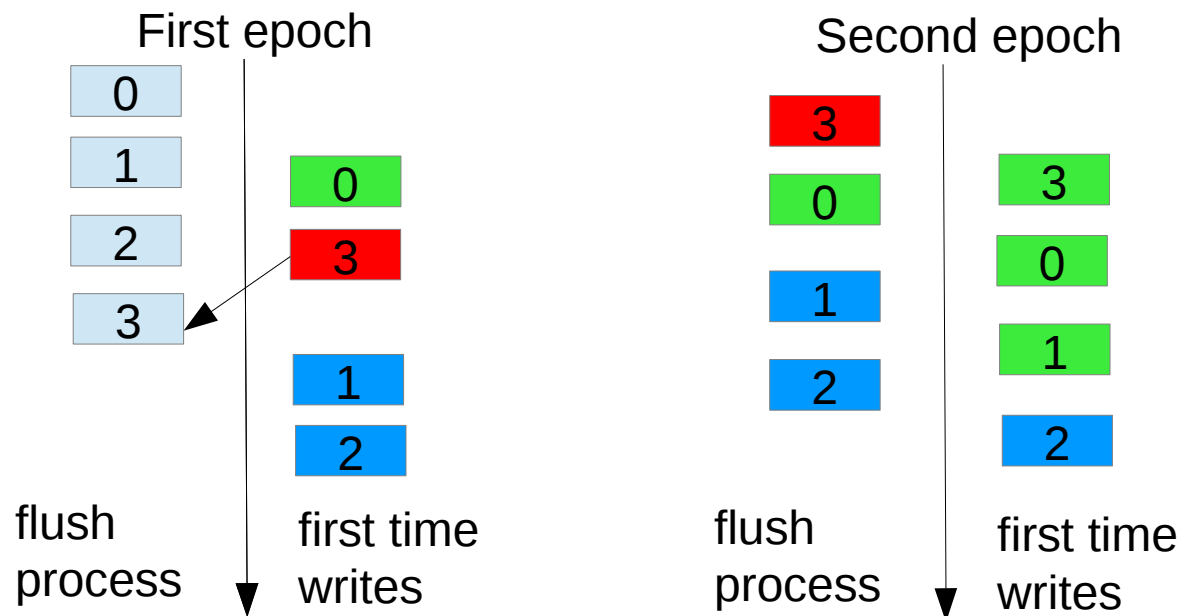
## Let's apply this idea to zero-copy

- Proposal:
  - Monitor first time writes during each epoch
  - Flush during each epoch according to ordering from previous epoch
- What if we have irregularities (i.e. order inside epoch unknown) ?



## Proposal: besides order, take into account circumstances

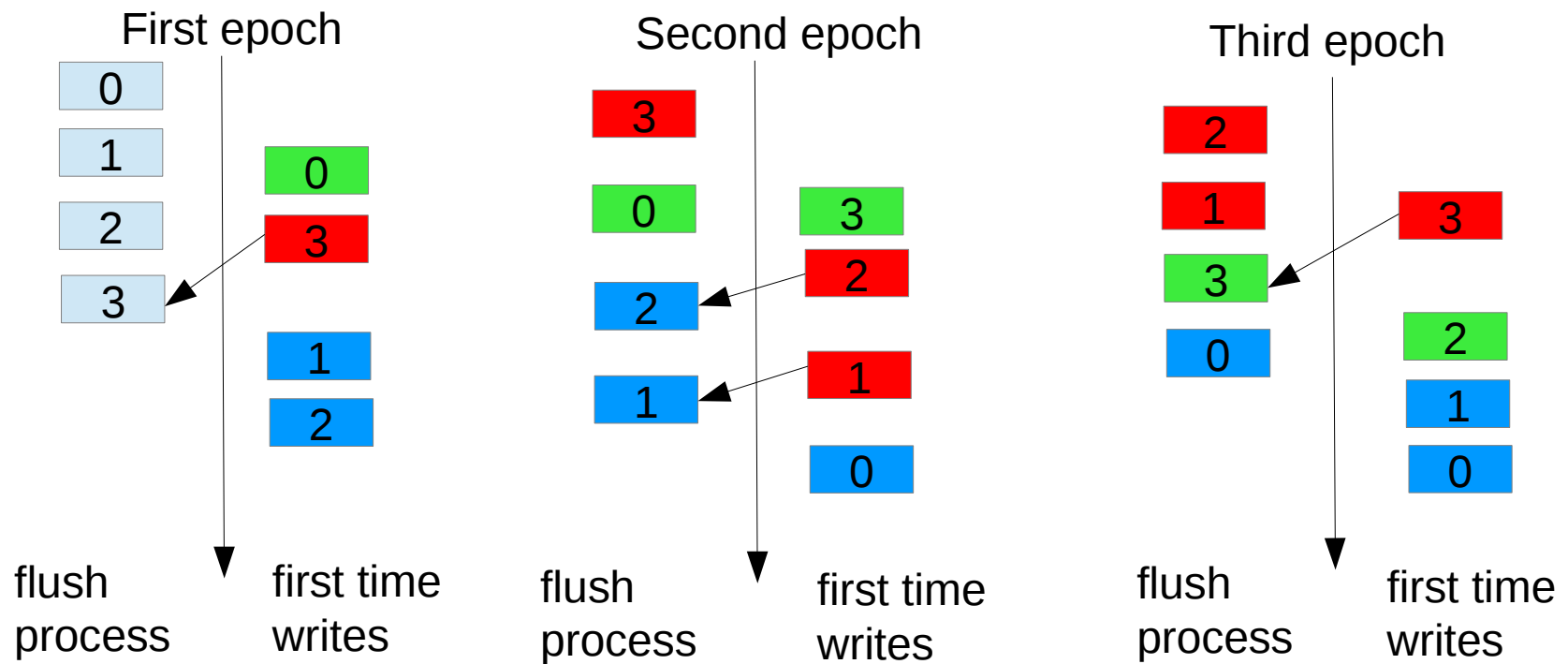
- Assign colors to pages according to their behavior
  - **RED**: app had to wait for page to be flushed
  - **GREEN**: page was flushed before first time write
  - **BLUE**: checkpoint already completed before first time write
- Flush “dangerous” pages first (i.e. red > green > blue)



## What about larger deviations?

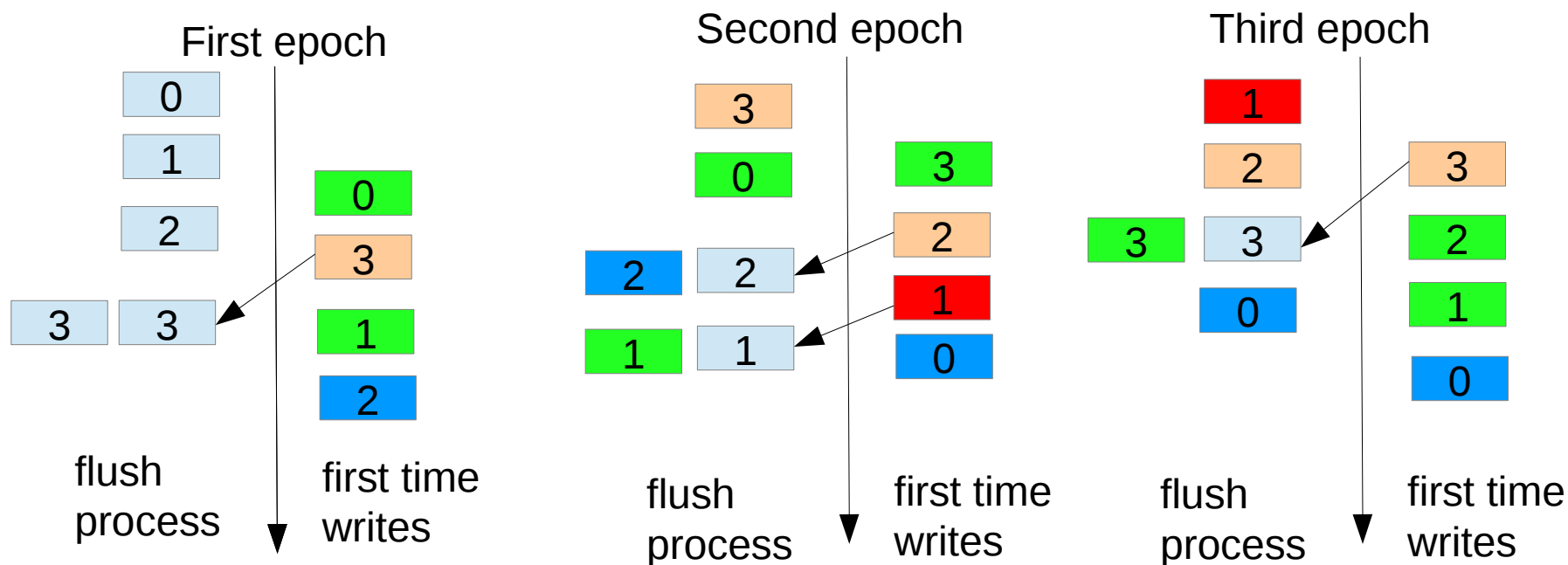
- Large deviations are more problematic
- Can we do better?

*0,1,2 inverted to 2,1,0  
3 irregular*



## Proposal: small copy-on-write buffer to avoid waiting

- New colors (COW buffer has 1 slot in example)
  - **RED**: COW buffer full, app had to wait for page to be flushed
  - **ORANGE**: COW buffer not full, wait avoided by performing COW
  - **GREEN**: page was flushed before first time write
  - **BLUE**: checkpoint already completed before first time write



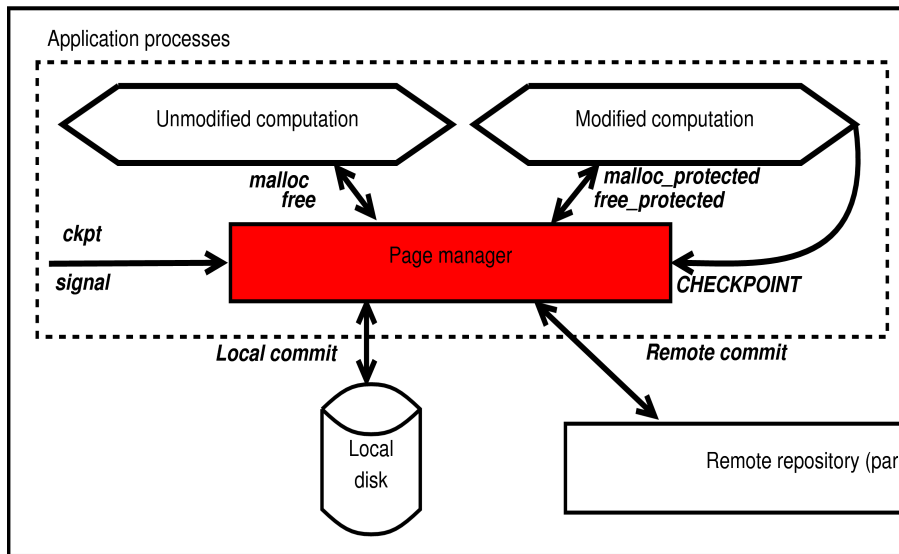
## Final algorithm

- Flush pages in the following order
  - Any page that was colored red in the current epoch
  - Any page that was colored orange in the current epoch
  - Any remaining page according to color from previous epoch
    - Color the page green in the current epoch
- If two pages have the same color, respect order from prev epoch
- After flushing is done, color all first time writes as blue
- Pages that didn't get any color are not flushed in the next epoch
  - This enables incremental support
  - If we do not want incremental support (e.g. because most pages change), we can simply deactivate monitoring after flushing is done (to reduce overhead) and implicitly color all remaining pages blue

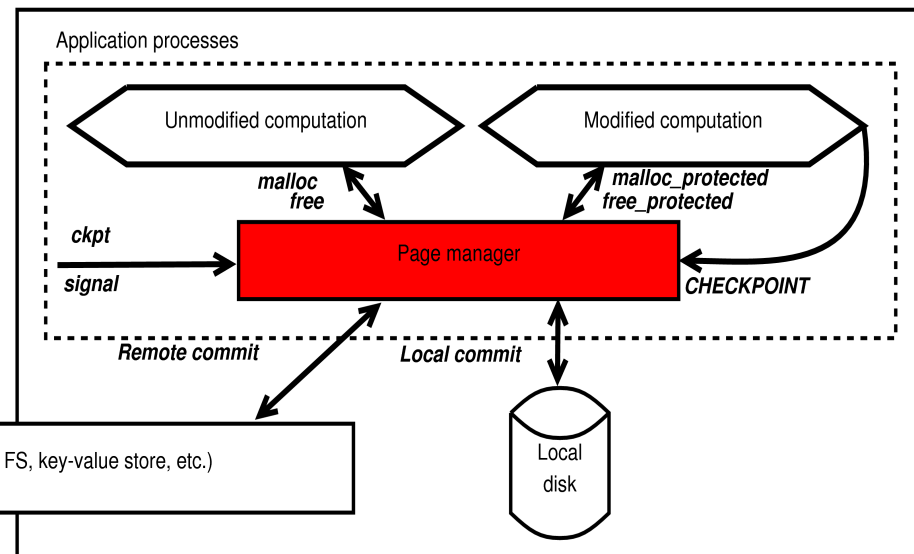


## Architecture and implementation

Compute node



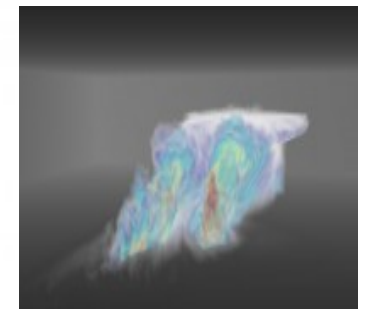
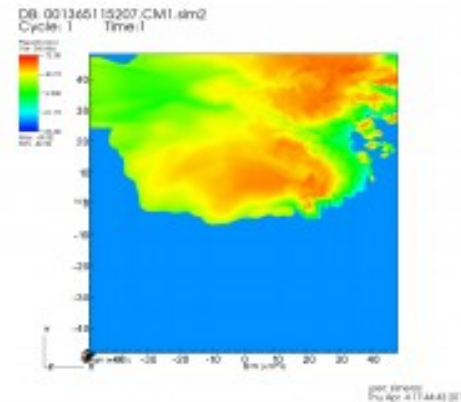
Compute node



- Page manager responsible for monitoring and flushing
  - SEGFAULTs used to trap first time writes
- Application links with page manager
  - Explicit (custom API) or implicit (jemalloc) protection of memory regions
  - CHECKPOINT primitive explicitly called or triggered externally
  - Epoch assumed to match checkpoint interval (needs improvement)

## Results: Experimental setup

- Platform:
  - Shamrock
  - Grid'5000
- Configuration:
  - Debian Sid, MPICH2 1.4.1
- Applications
  - Synthetic benchmark specifically written to evaluate proposal
  - Real life applications:
    - CM1: numerical model for studying atmospheric phenomena
    - MILC: MIMD Lattice Computation (quantum chromodynamics)



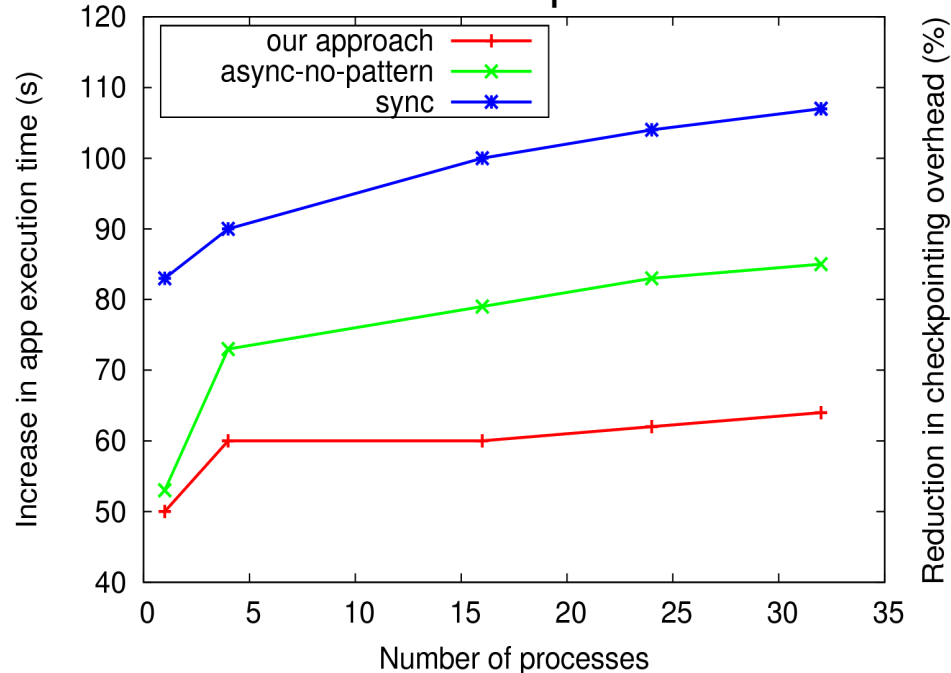
## Methodology

- Three approaches are compared:
  - Synchronous checkpointing
  - Asynchronous checkpointing without leveraging access pattern
  - Our approach
- Settings:
  - CM1 on G5K: 32 nodes, 1 process/node, flush to PVFS (10 nodes)
  - MILC on Shamrock: 28 nodes, 10 processes/node, flush to local
  - Page size is OS default (4KB)
  - Three checkpoints evenly spaced throughout runtime
- We are interested in:
  - Performance results: duration of checkpointing and impact on app
  - How the benefits of our approach depend on COW buffer size

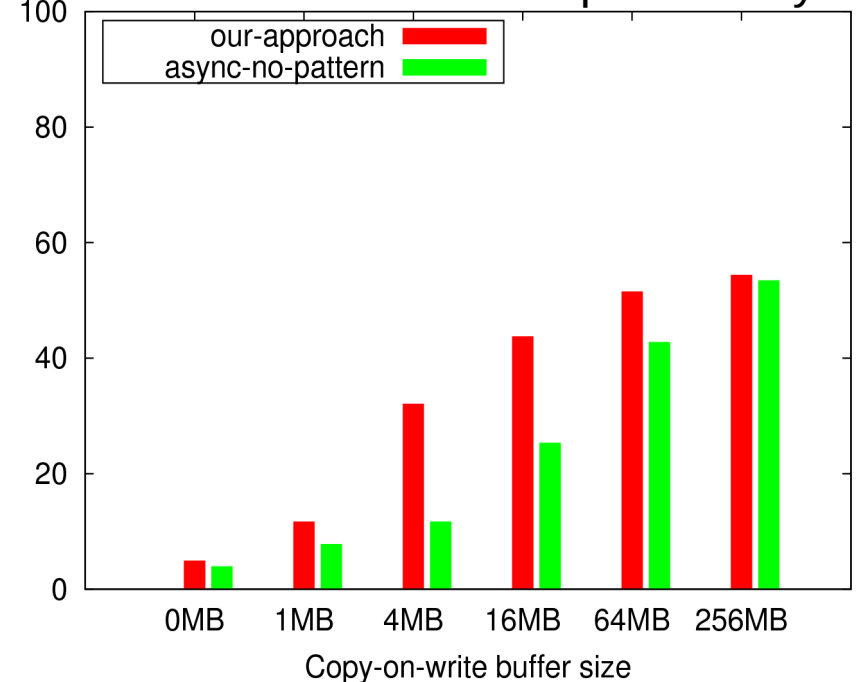


## Results: CM1 (incremental changes: 400MB/728MB per process)

Increase in runtime compared to baseline



Overhead reduction compared to sync



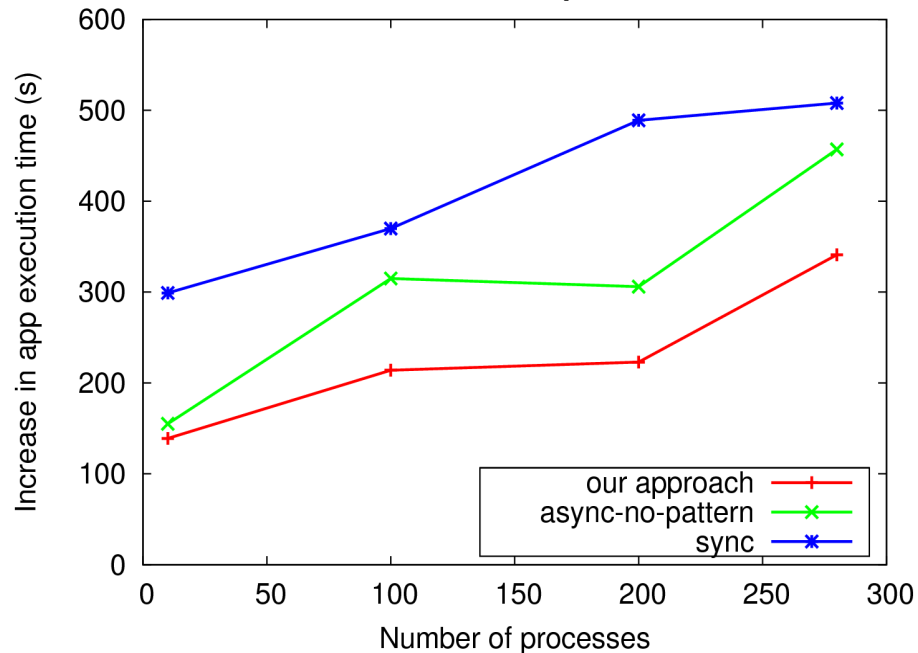
## • Conclusions

- Overlapping significantly reduces overhead
- Small COW buffer needed to survive deviations
- Adaptation to access pattern further reduces overhead for small COW buffer sizes

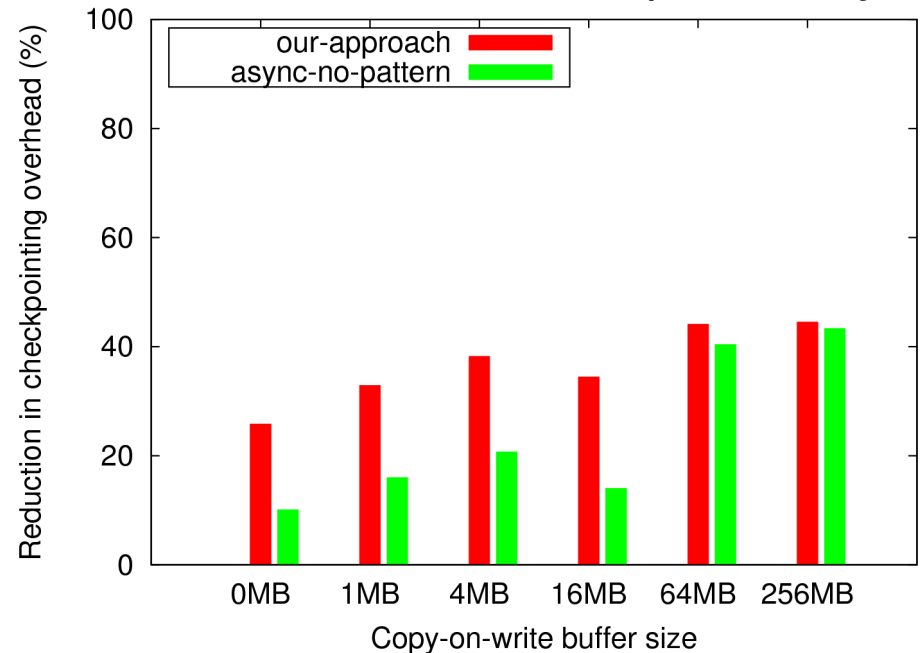
*Baseline: 700s**Sync: 100s*

## Results: MILC (incremental changes: 830MB/866MB per process)

### Increase in runtime compared to baseline



### Overhead reduction compared to sync



*Baseline: 2100s*

*Sync: 2600s*

## Conclusions

- Naïve async and sync much closer, clear distance for our approach
- More regular access pattern, thus better reduction even without COW
- Adaptation to access pattern keeps clear advantage for small COW

## Conclusions

- Asynchronous techniques overlap checkpointing with application execution, lowering downtime and thus performance overhead
- HPC applications exhibit predictable first-time writes that can be exploited to optimize flushing order of memory pages
- Results show:
  - Reduction of checkpointing overhead of up to 60% compared to sync
  - Performs up to 2x better than naïve async
  - All this for less than 5% extra memory dedicated to copy-on-write
- This technique is ideal for situations where little extra memory is available or the extra memory is needed for post-processing (e.g. compression, de-duplication, message logging, etc.)

Contact: [bogdan.nicolae@ie.ibm.com](mailto:bogdan.nicolae@ie.ibm.com)

Web: <http://researcher.ibm.com/person/ie-bogdan.nicolae>

