Techniques to improve the scalability of Checkpoint-Restart

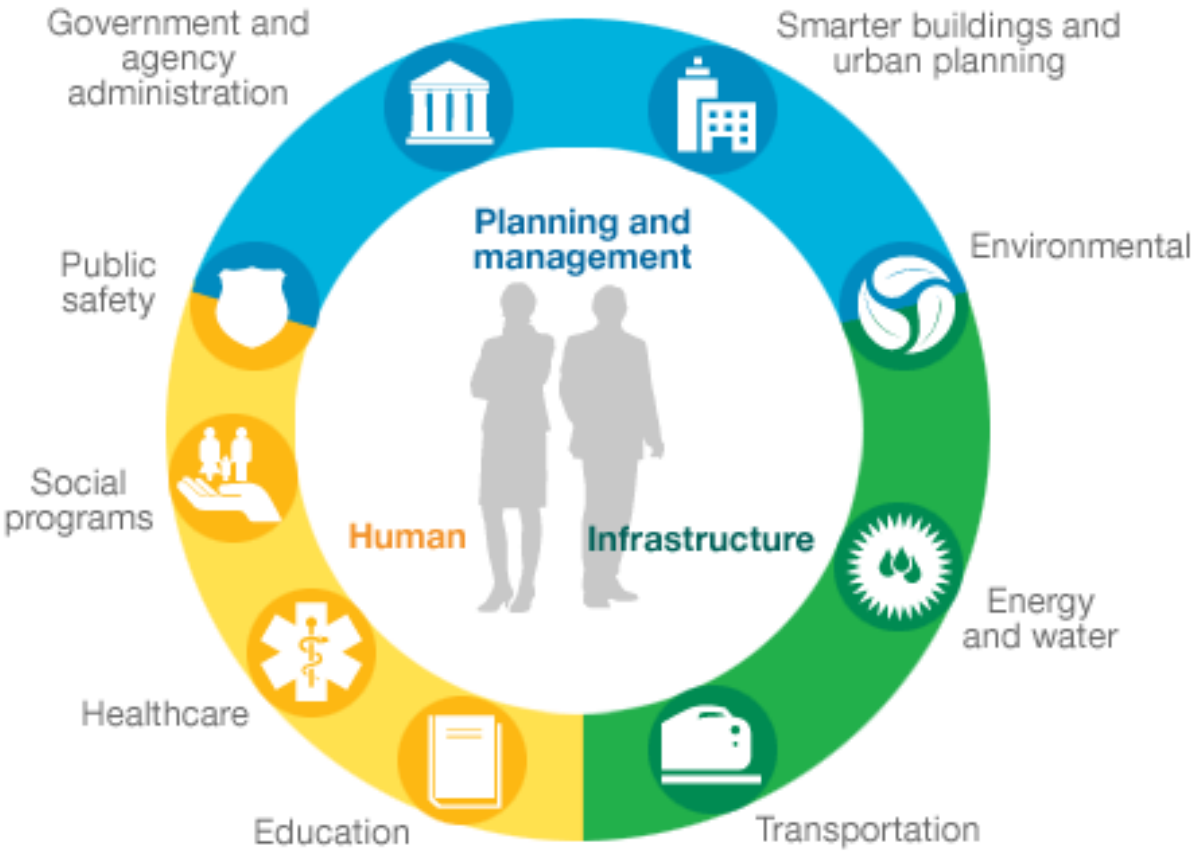
Bogdan Nicolae
Exascale Systems Group
IBM Research Ireland

Outline

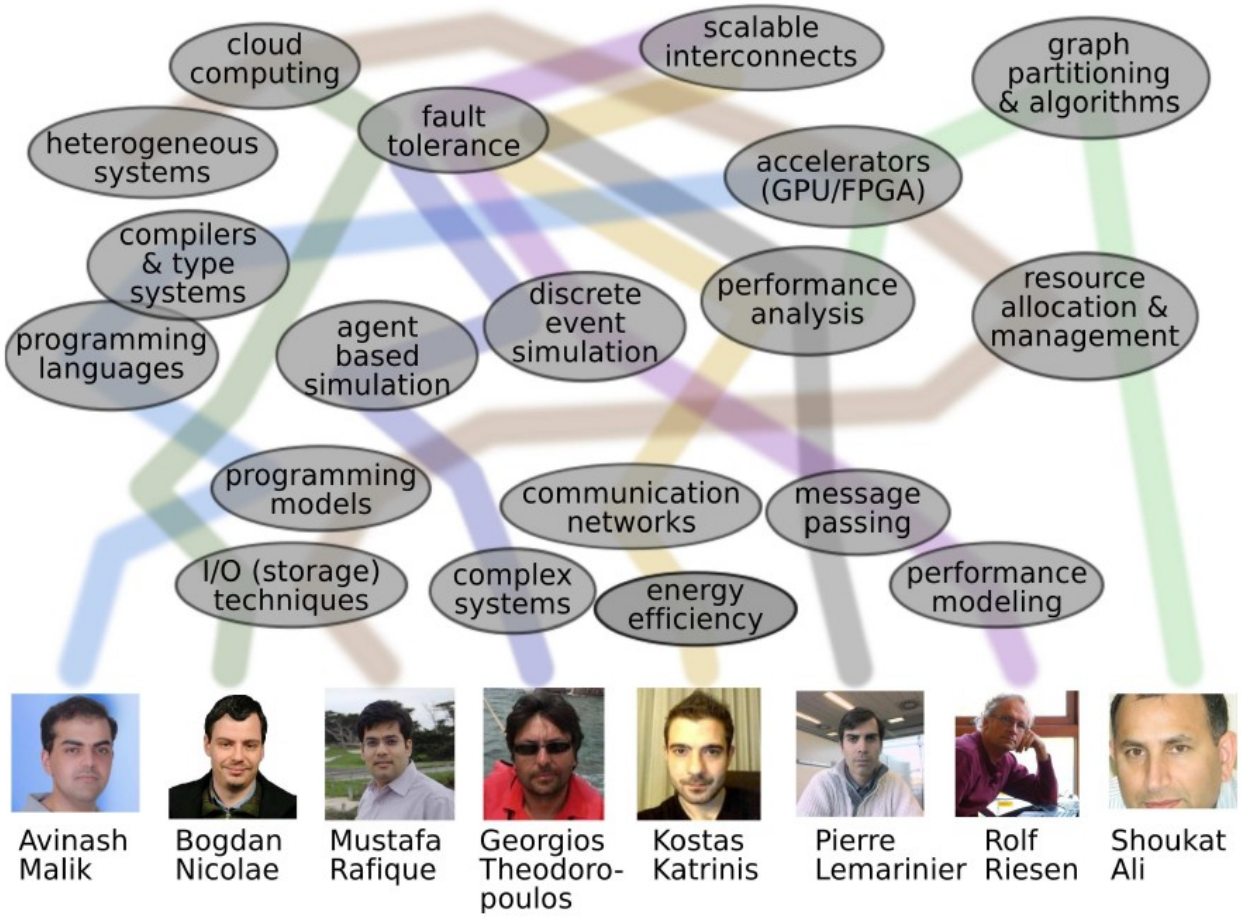
- A few words about the lab and team
- Challenges of Exascale
- A case for Checkpoint-Restart
- Two techniques to improve the scalability of Checkpoint-Restart
  - Leveraging memory access patterns and bounded COW buffers to optimize asynchronous incremental checkpointing
  - Collective inline memory contents deduplication
- Conclusions
- Future work / Collaboration opportunities

The smarter cities technology centre

- Classic HPC (1)
  - Simulations
- Data centric computing (2)
  - Data-intensive
  - Streaming
  - Data warehousing
- Specifics
  - Convergence of (1) and (2)
  - Big data



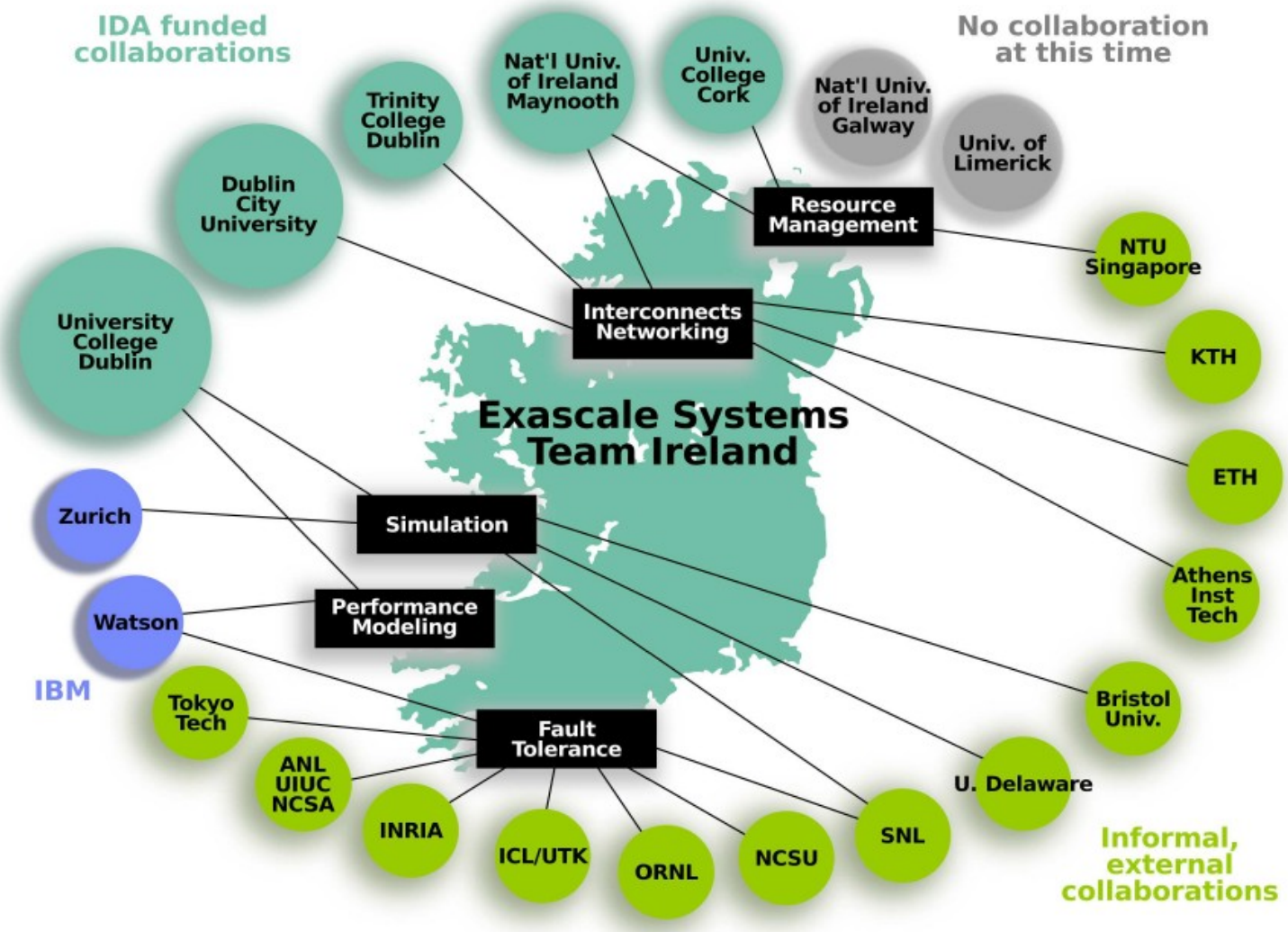
The Exascale Team



- Works at system and runtime level
- Gets workloads from application teams
- Leverages new prototypes from hardware teams

SmarterCities Technology Centre

Collaborations



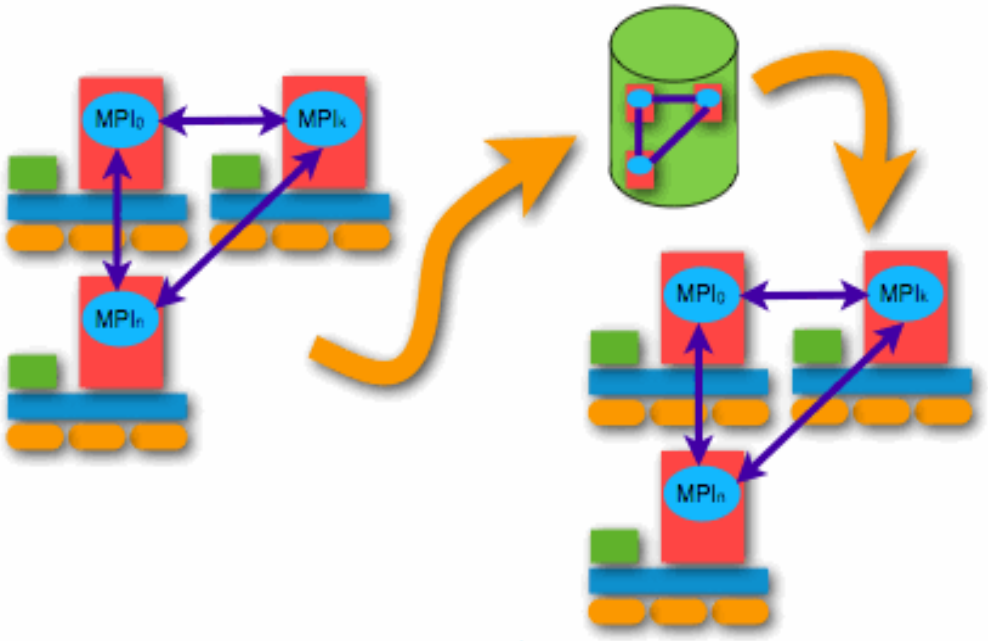
SmarterCities Technology Centre

Challenges of Exascale

- Massive 4000x increase in concurrency
  - Mostly within the same compute node, hybrid systems likely
- Imbalance between compute power and memory
  - 500x compute, only 30x memory despite increasingly data-intensive
  - Need to make better use memory and drive its costs down
- Huge energy consumption
  - Power-awareness in applications and runtime
- Poor scalability of I/O and data management
  - Need new I/O mechanisms and storage systems
  - Leverage local storage (SSDs, NVMs, etc.)
- High component failure
  - Must improve system resilience and manage component failure
  - Checkpoint/Restart not scalable in its current form

Checkpoint-Restart: is it really dead?

- What are the current limitations?
  - Blocking writes
  - Too much data
  - Coordinated protocols
  - I/O bottlenecks due to parallel FS
- Directions
  - Asynchronous techniques
  - Reduction of checkpointing data
  - Uncoordinated protocols
  - Leverage local storage (make it resilient)



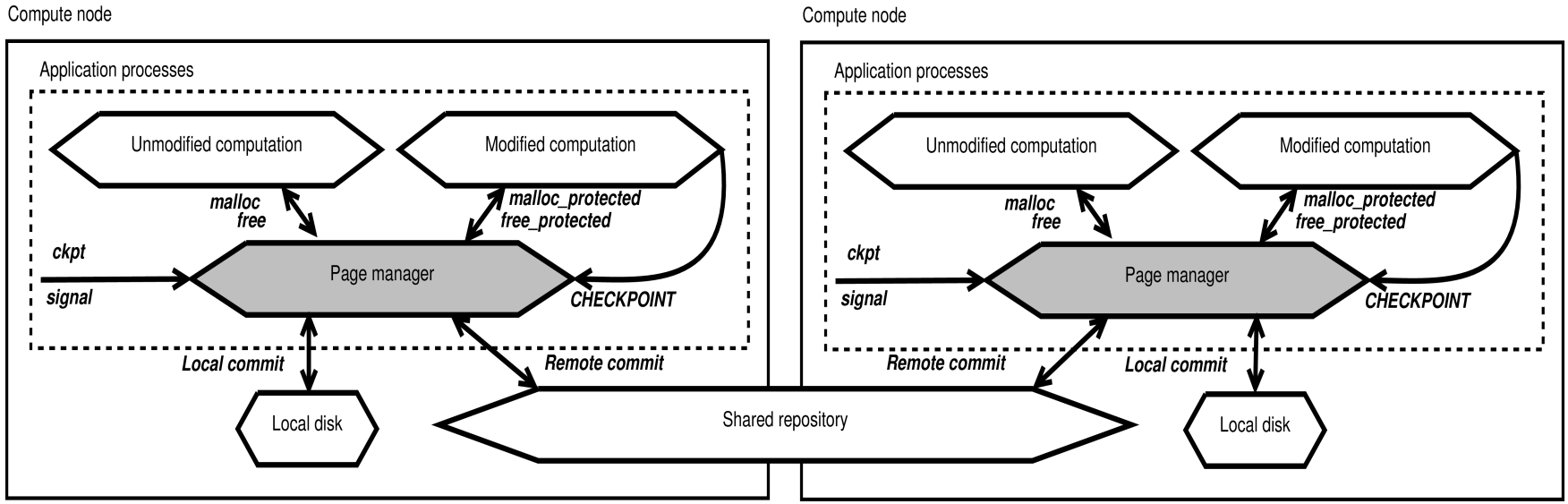
Contribution #1: Leveraging Memory Access Patterns for Adaptive Asynchronous Incremental Checkpointing

- Motivation: state-of-art asynchronous techniques have drawbacks
  - Full copy, then flush in the background
    - High copy overhead
    - High memory utilization
    - No synchronization overhead
  - Copy-on-write
    - Less copy overhead
    - High memory utilization
    - Monitoring overhead
  - Zero-copy
    - No copy overhead or memory utilization
    - High synchronization overhead
- What we ideally want: minimize memory utilization without paying too much for synchronization/copy overhead

Contribution #1: Our approach

- Bounded copy-on-write buffer
  - Enables the app to specify how much mem to reserve for COW
- Monitor memory writes between ckpt requests (i.e. “epoch”)
  - Classify pages:
    - **RED:** COW buffer full, app had to wait for page to be flushed
    - **YELLOW**: COW not full, app could do COW and then continue
    - **GREEN**: page was flushed before app wrote to it
    - **BLUE**: checkpoint already completed before app wrote to page
- Leverage monitoring info to prioritize flushes
  - Flush pages of in current COW buffer
  - Otherwise use page color from previous epoch: from “hot” to “cold”
- Implicit incremental support due to monitoring info

Contribution #1: Architecture and implementation



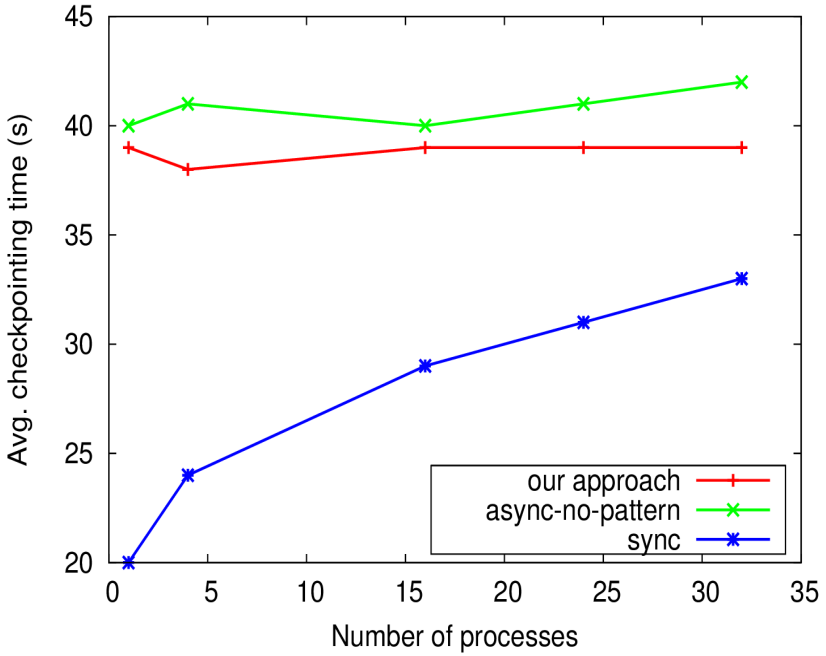
- Page manager does monitoring and flushing
  - CHECKPOINT primitive initiates checkpoint
  - Writes trapped using SEGFAULT
- Explicit protection of memory contents using allocation primitives
- Implicit protection of memory contents using a modified jemalloc

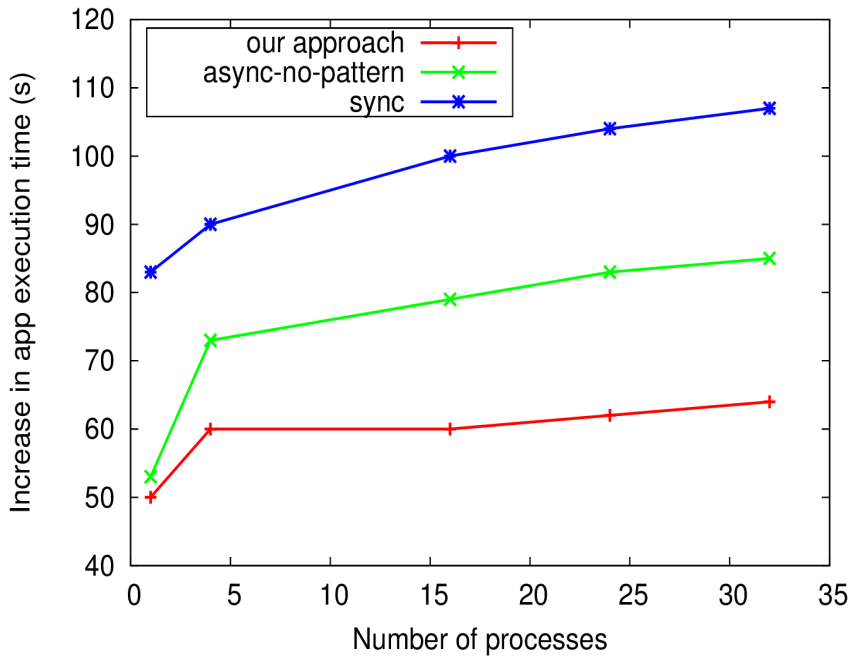
Results: preliminaries

- Experimetal setup
  - Platform: G5K
  - App: Benchmarks + CM1 (only CM1 presented in this talk)
  - Checkpoints at regular intervals (total of 3)

- Experiments
  - Compare three approaches:
    - Synchronous checkpointing
    - Asynchronous checkpointing without leveraging access pattern
    - Our approach
  - We are interested in:
    - Performance results: duration of checkpointing and impact on app
    - How the benefits of our approach depend on COW buffer size

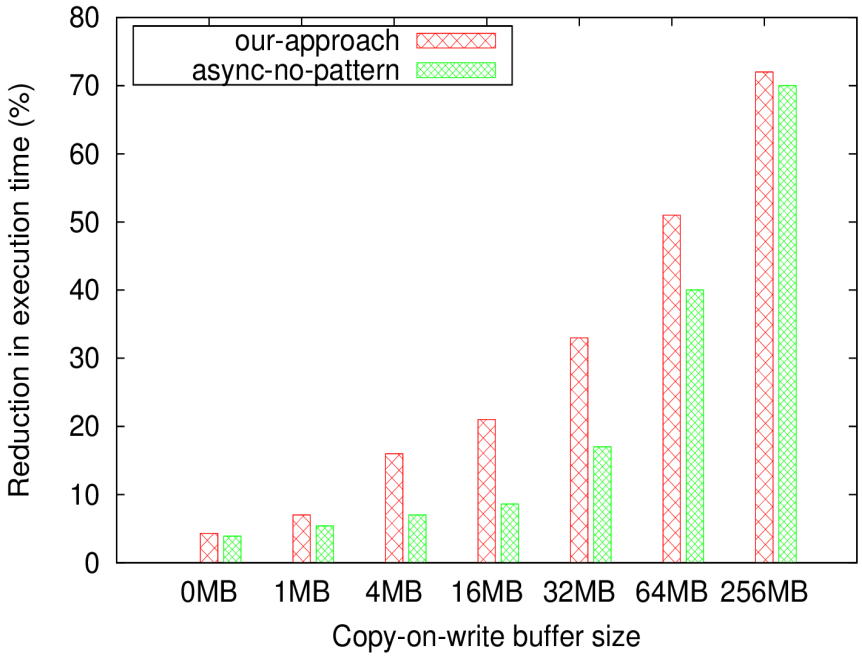
Results: performance evaluation (16 MB COW buffer, 400MB/process)

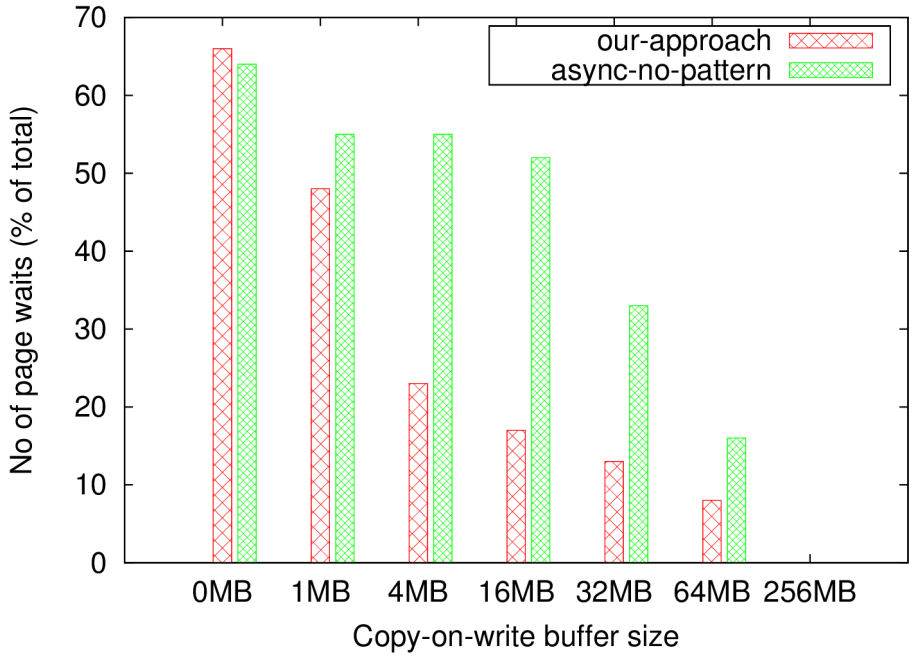




- Conclusions
  - Sync ckpt is slow has poor scalability
  - Naïve async ckpt takes longer but overlapping reduces overhead
  - Adaptation to access pattern further reduces overhead by almost 50% compared to the naive async approach

Results: impact of leveraging access pattern for variable COW buffer size





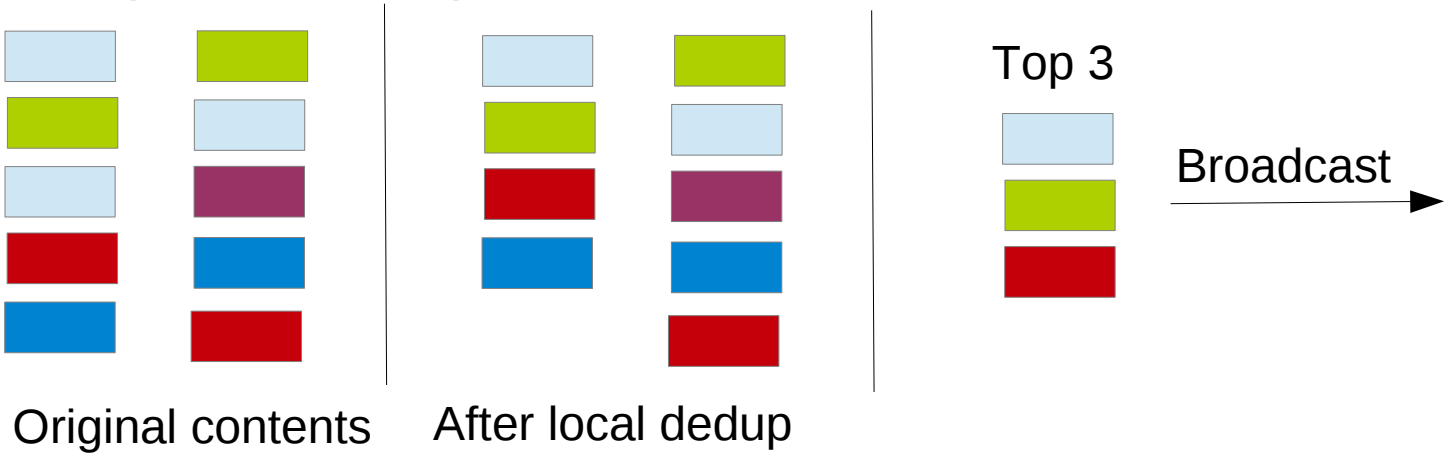
- Conclusions
  - If COW buffer is large, access pattern makes little difference
  - Huge reduction (>50%) in RED pages for small COW buffer size
  - When memory is scarce, our approach has substantial benefits

Contribution #2: Collective inline memory deduplication of checkpointing data

- Motivation: lots of checkpointing data generated by HPC aplications contains duplicates (e.g. up to 70% according to the HPC data deduplication study by D. Meister et al., SC'12)
- State of art in de-duplication for HPC
  - Wast majority are offline approaches (i.e. applied after checkpoints were written to parallel FS)
  - Inline approaches exist, but they operate locally on the memory space of individual processes
- Our idea: look at the collective memory space of all processes and try to identify also duplicates belonging to different processes. Why is this challenging?
  - Huge space to look for duplicates => high performance overhead
  - Not easy to parallelize efficiently
  - Metadata about identified duplicates is huge itself!

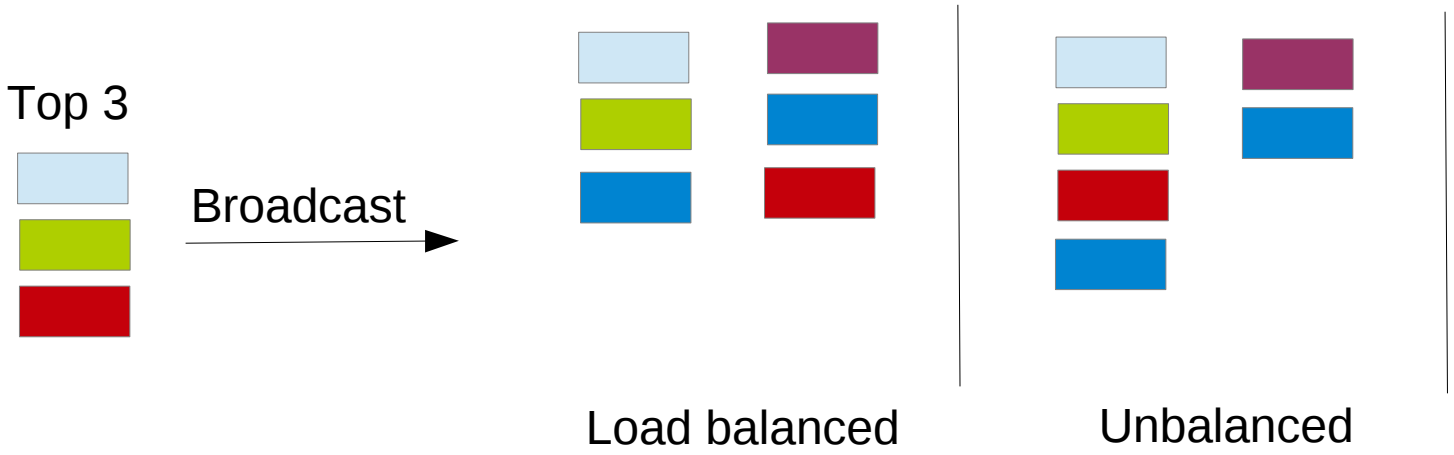
Our approach(1)

- Search for top-K most popular pages only
  - Perform a local deduplication step
  - With all remaining pages from all processes perform a global reduction to obtain top-K
  - Broadcast top-K most popular pages to all processes for further deduplication
  - Logarithmic approches (in proc #) can be used (e.g. MPI all_reduce)
- Example with two processes, K=3:



Our approach (2)

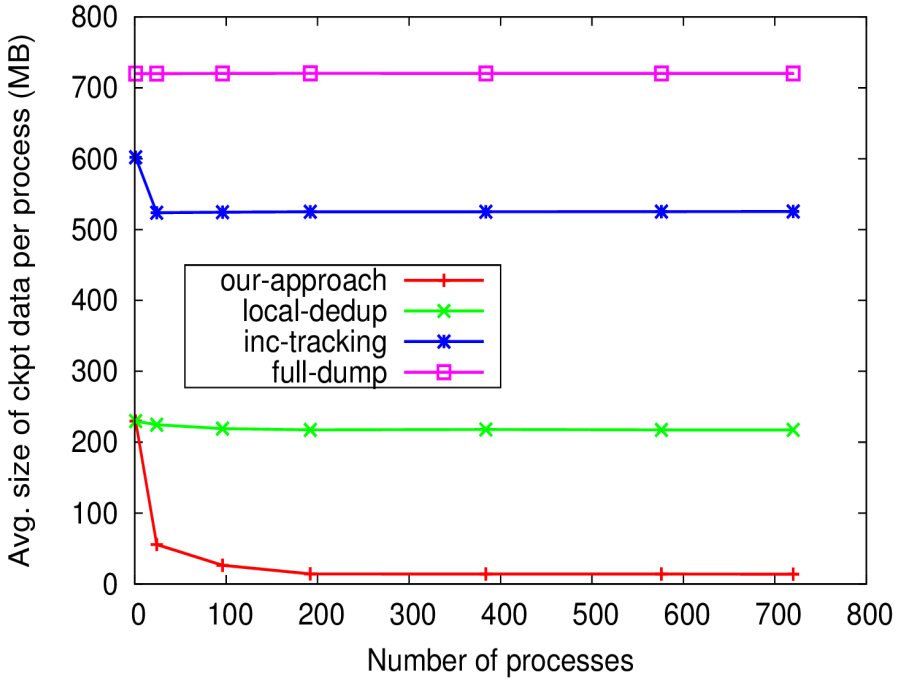
- What to do once we know top-K?
  - Basically store only one copy
  - ...But who stores what?
  - Load balancing is crucial in order to avoid waiting for slow checkpointers that are responsible for more pages than others
  - We propose an algorithm to do that during reduction
- Example with two processes, K=3:

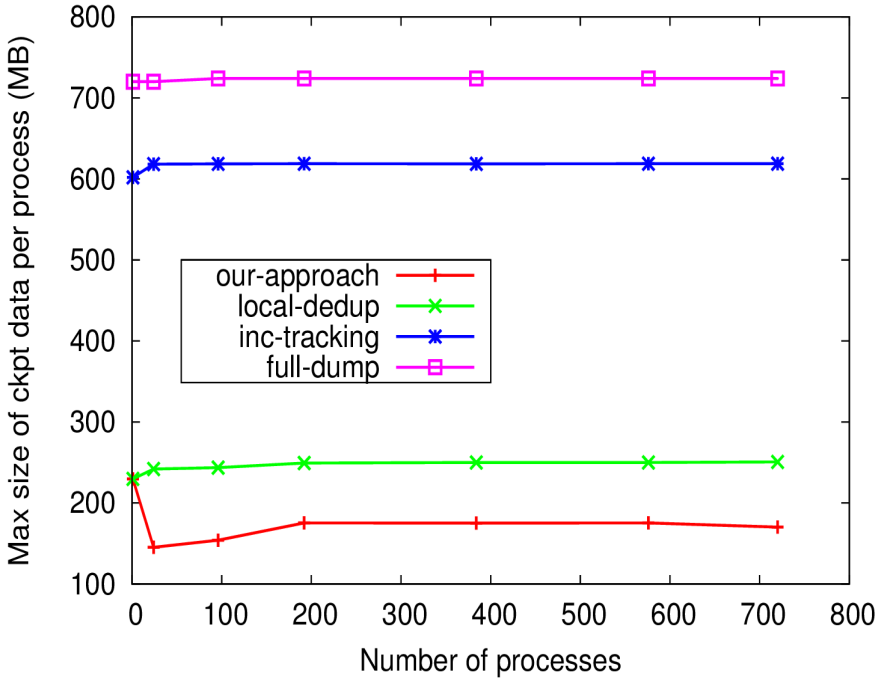


Results: preliminaries

- Experimetal setup
  - Platform: G5K (INRIA), Shamrock (IBM Ireland)
  - App: Benchmarks + CM1 (only CM1 presented in this talk)
  - Checkpoints at regular intervals (total of 3)
- Experiments
  - Compare four approaches:
    - No dedup
    - Incremental
    - Local dedup
    - Collective dedup using our approach
  - We are interested in:
    - Performance overhead: impact on app performance
    - How much storage space can be saved
    - How well our load balancing strategy performs

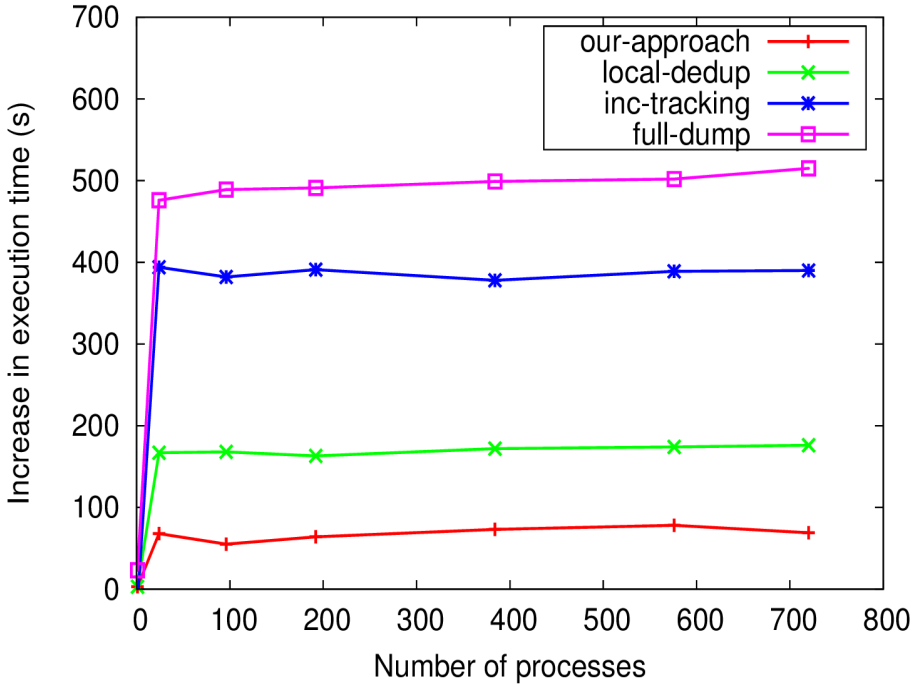
Results: storage space reduction (728MB/process, 24 cores/node)

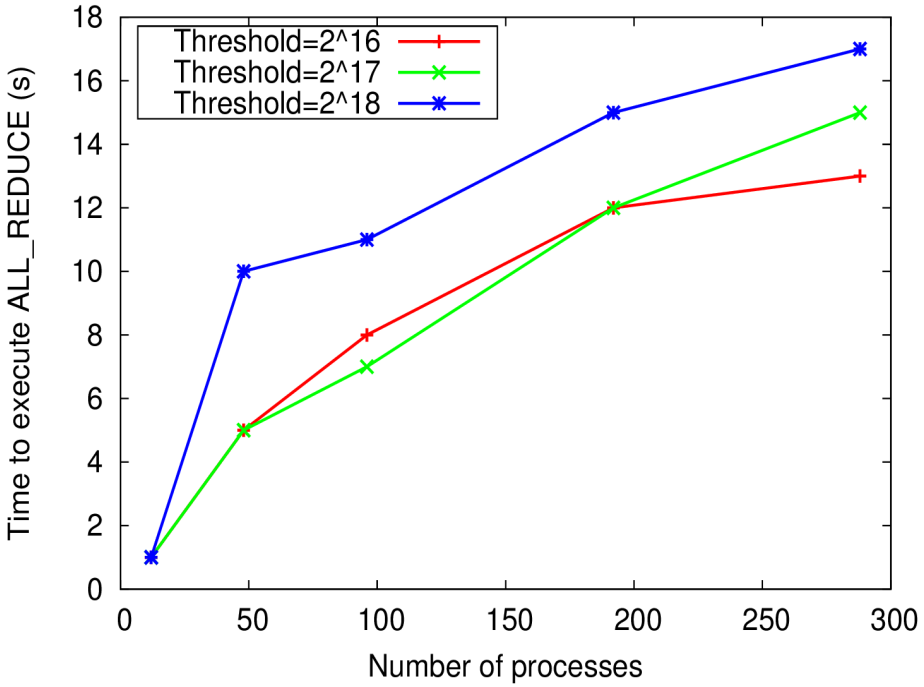




- Conclusions
  - Local dedup is better than incremental => overwrites with same data
  - Massive amounts of duplicate data across different processes
  - Thanks to load balancing, even the most loaded process needs to save half of checkpointing data compared to local deduplication

Results: performance overhead (728MB/process, 24 cores/node)





- Conclusions
  - ALL_REDUCE timings confirm logarithmic cost of top-K calculation
  - Performance overhead is reduced by more than 50% compared to local dedup, and by much more compared to the others

Conclusions

- Checkpoint-Restart is not yet dead :)
- Two directions have high potential for improvement
  - Leveraging access pattern history to improve asynchronous checkpointing with bounded COW buffer
  - Collective inline memory deduplication
- Results show
  - Adaptation to access pattern reduces increase in execution time due to checkpointing for real life apps by almost 50% compared to the naive async approach
  - Large reductions in storage space (up to 90%) and performance overhead (at least 50%) for real apps that exhibit high duplication of memory contents

Future work

- Related to results presented so far
  - How to best combine the two techniques for additional overall reduction of performance overhead
  - Extend the collective deduplication scheme to leverage natural replication due to duplication to provide resilience
  - Study more HPC apps and correlate duplication to semantics of memory contents to understand its nature
- Unrelated to current results
  - Lightweight storage systems that leverage local storage (e.g. memory allocators that combine RAM with local storage)
  - HPC on the Cloud: how to make virtualization an ally instead of enemy

Looking for collaborations in these areas

- *Contact: bogdan.nicolae@ie.ibm.com*
- *Web: http://researcher.ibm.com/person/ie-bogdan.nicolae*