A 1 PB/s File System to Checkpoint Three Million MPI Tasks

Raghunath Rajachandrasekar * - Adam Moody # - Kathryn Mohror # - DK Panda *
* Network-Based Computing Laboratory | The Ohio State University
# Lawrence Livermore National Laboratory

THE OHIO STATE UNIVERSITY

Lawrence Livermore National Laboratory

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. The project is supported in part by NSF grants CCF-0937842 and OCI-11148371. [LLNL-PRES-639372]

What is checkpointing?
&
Why do we need it?

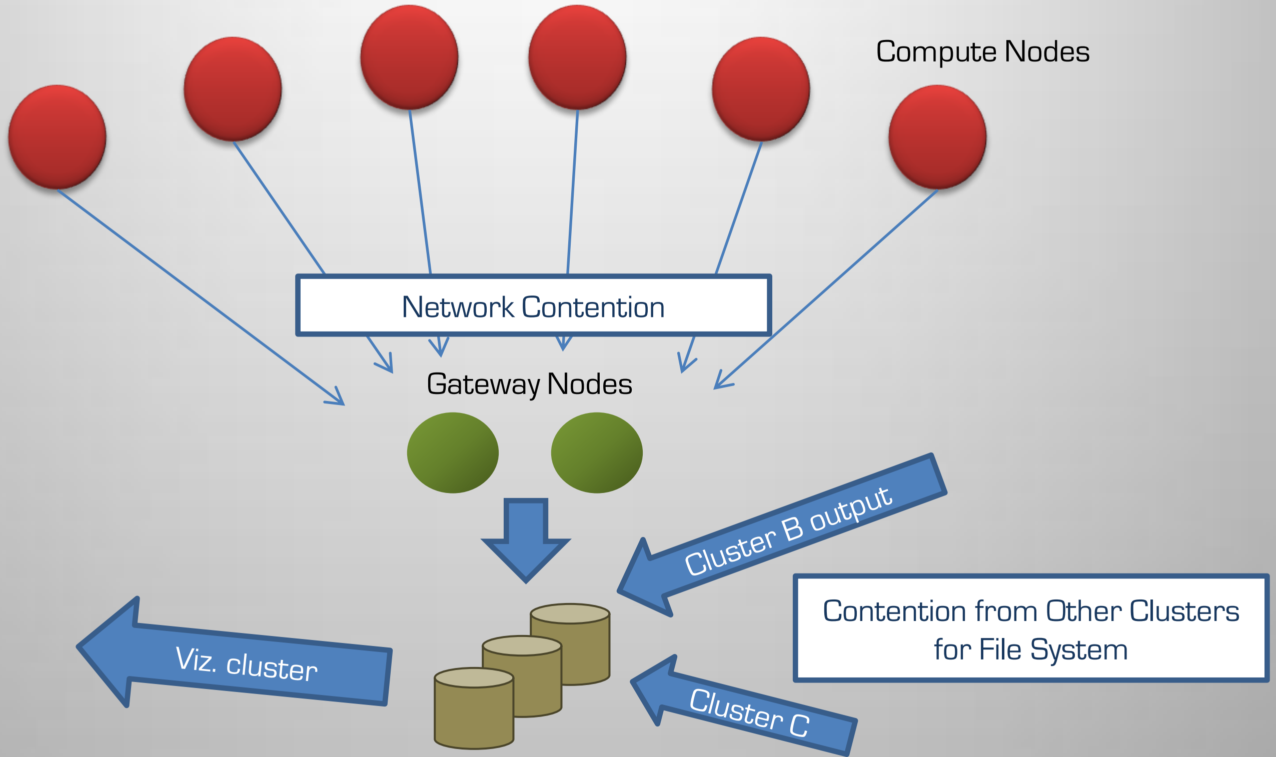
Tolerating failures with Checkpointing

- HPC architectures evolving to accommodate complex apps
  - Multi-core CPUs, GPUs, Xeon Phi, SSDs, Smart NICs…
- Failure is inevitable => Imperative to design fault-resilient systems
- Several tools and techniques are used to tolerate failures
  - Proactive and reactive
- Checkpoint-Restore mechanisms are predominantly used

That's awesome!

But, we have a problem..

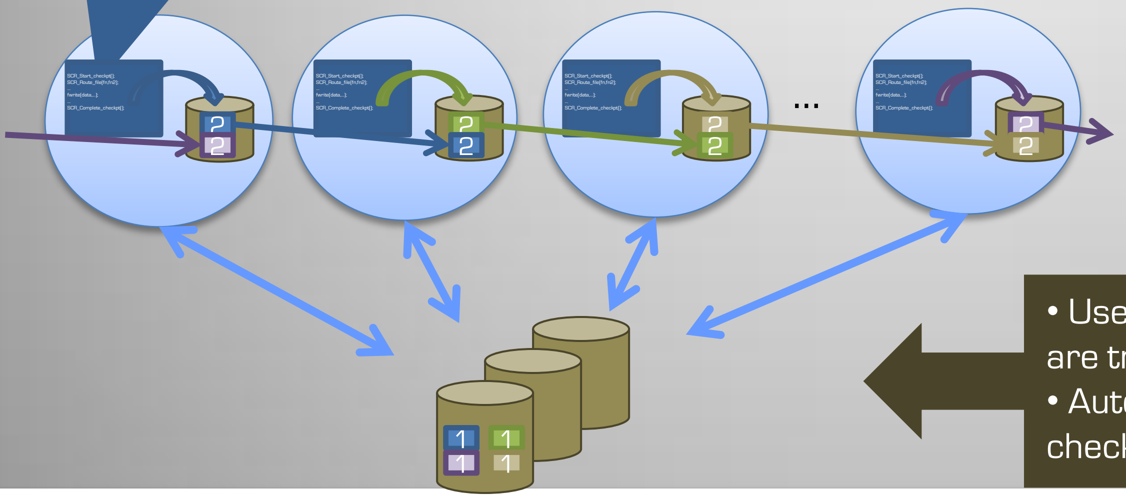
Contention for shared-storage resources



But, we also have a solution..

Scalable Checkpoint/ Restart (SCR)

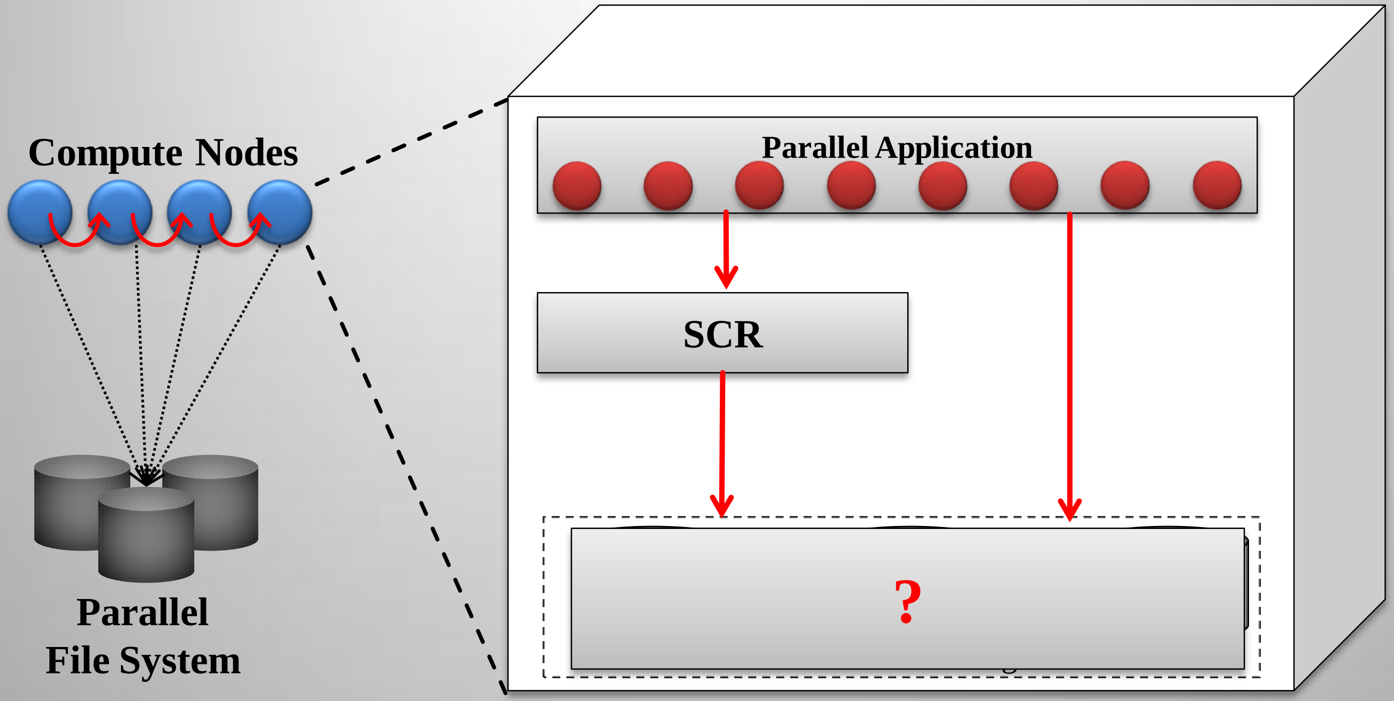
```
SCR_Start_checkpt();
SCR_Route_file(fn,fn2);
…
fwrite(data,…);
…
SCR_Complete_checkpt();
```



- First write checkpoints to node-local storage
- When checkpoint is complete, apply redundancy schemes

- Users select which checkpoints are transferred to global storage
- Automatically drain last checkpoint of the job

Node-local storage with SCR



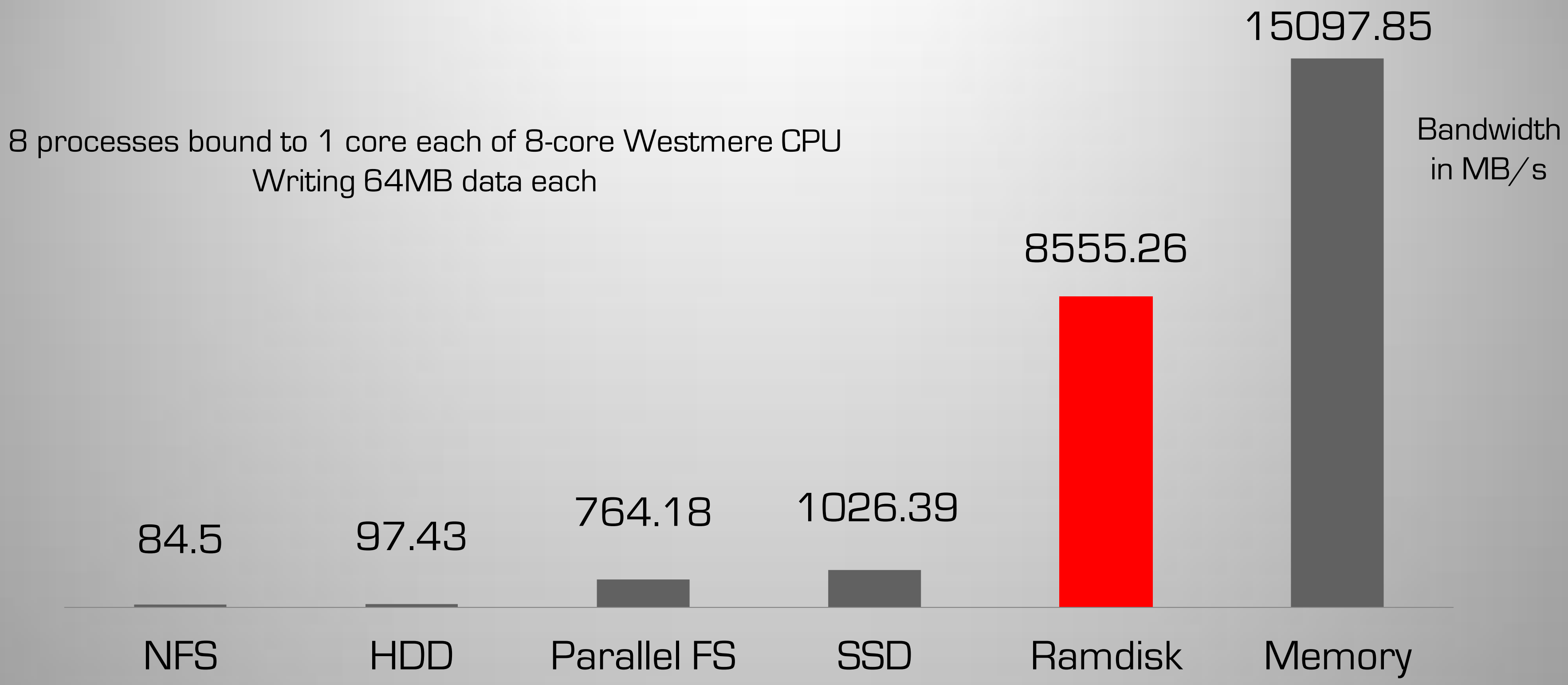
Design Goals

- Large checkpoints -- small memory
- Leverage the node-local storage hierarchy
- Asynchronous copy-out capability
- Portability
- Future-proof
- Improve checkpointing throughput

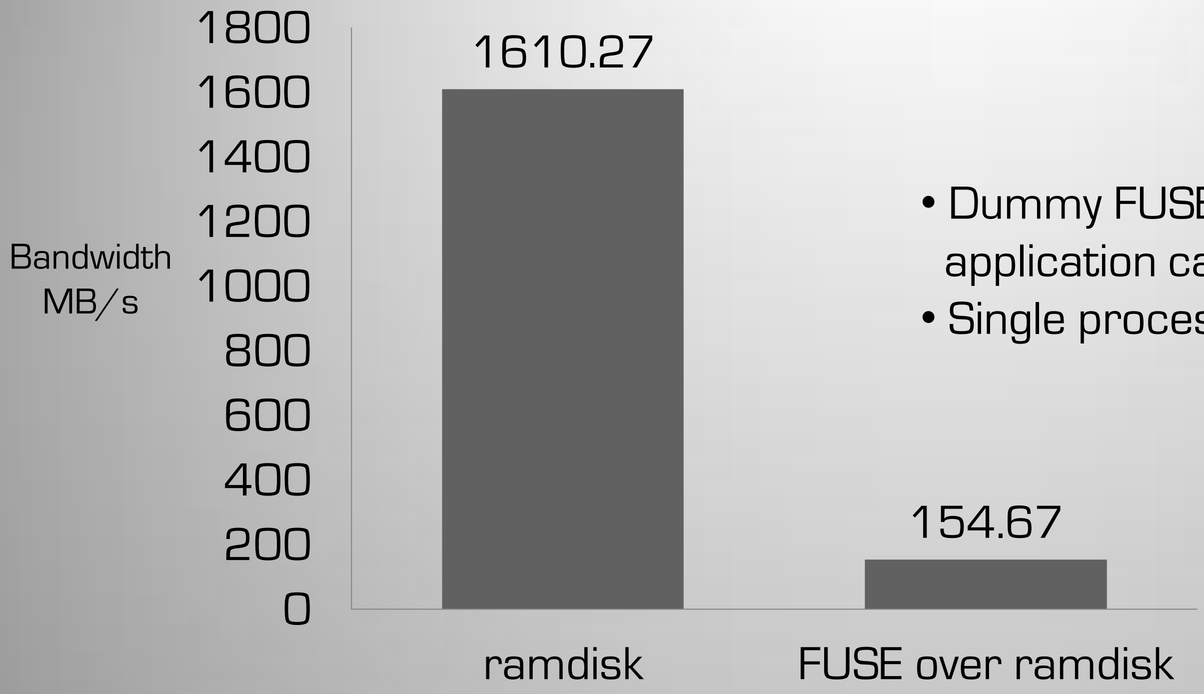
Possible Solutions

- Storage medium for the checkpoint data
  - Use kernel-provided "ramdisk"
  - Write an "in-memory" filesystem backed by an mmap'ed file
  - Just use the kernel buffer-cache
  - Manage System V IPC / Persistent memory segments
- Intercept application I/O
  - Write a FUSE-based file system
  - Trap I/O calls from the application using linker support

Can we not just use the RAM disk?



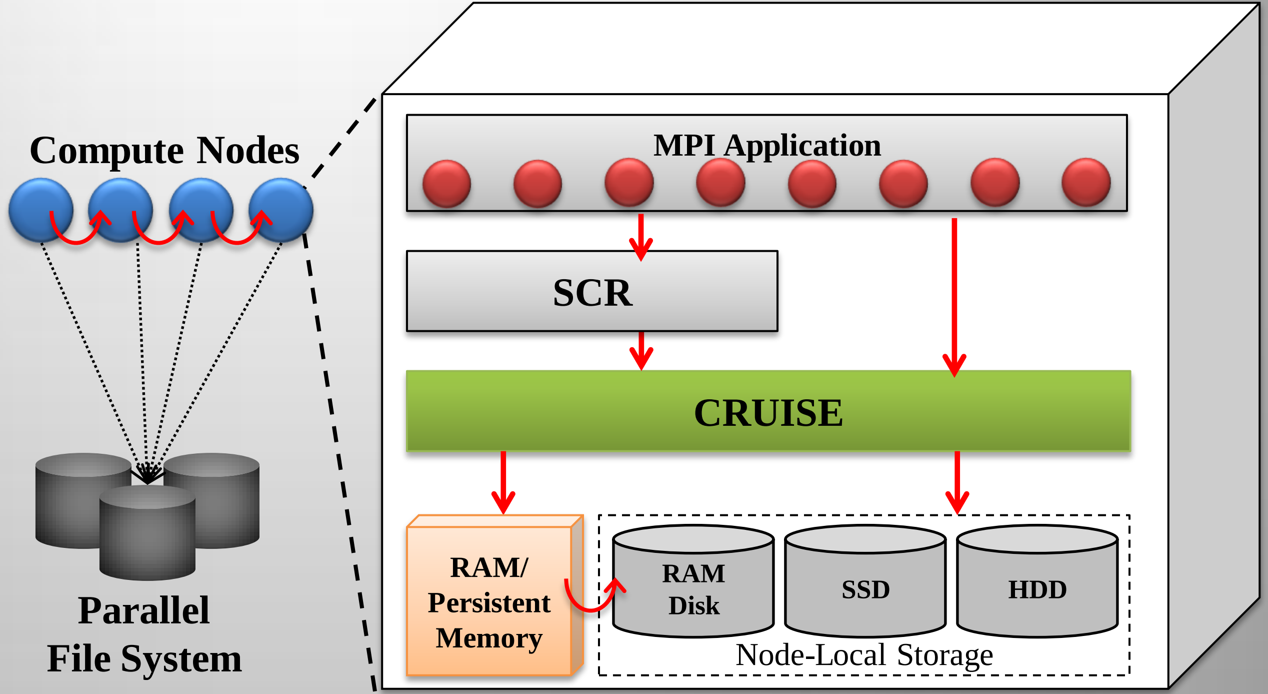
Can we not just write a FUSE file system?



- Dummy FUSE file system forwards all application calls to actual system call
- Single process writing 50MB data

CRUISE: Checkpoint-Restart in User-Space

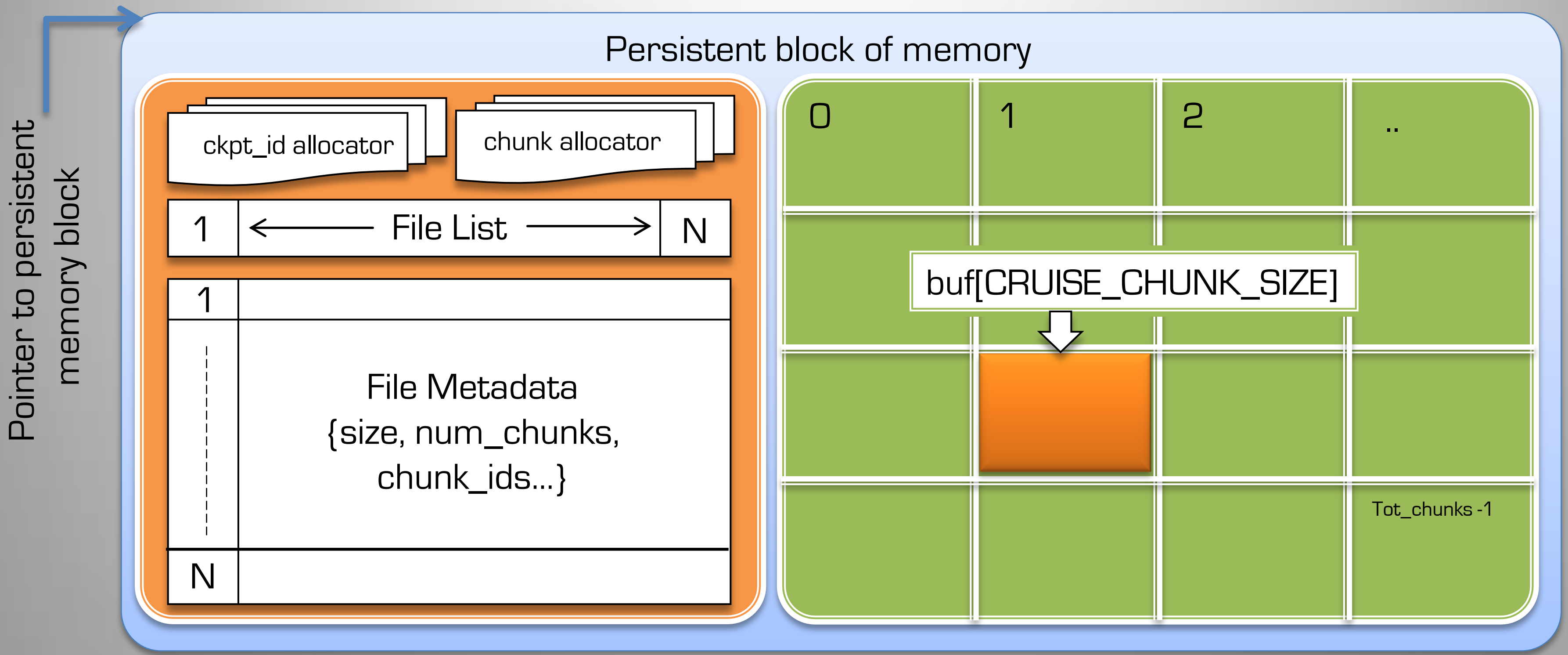
- Manages data in a byte-addressable persistent-memory region
- Library to intercept I/O operations from an application that links to it
  - Can statically / dynamically intercept I/O calls (LD_PRELOAD or -wrap,function)
  - Also implements the different calls such as open, write, read, ftruncate, lseek, etc….



Design Assumptions

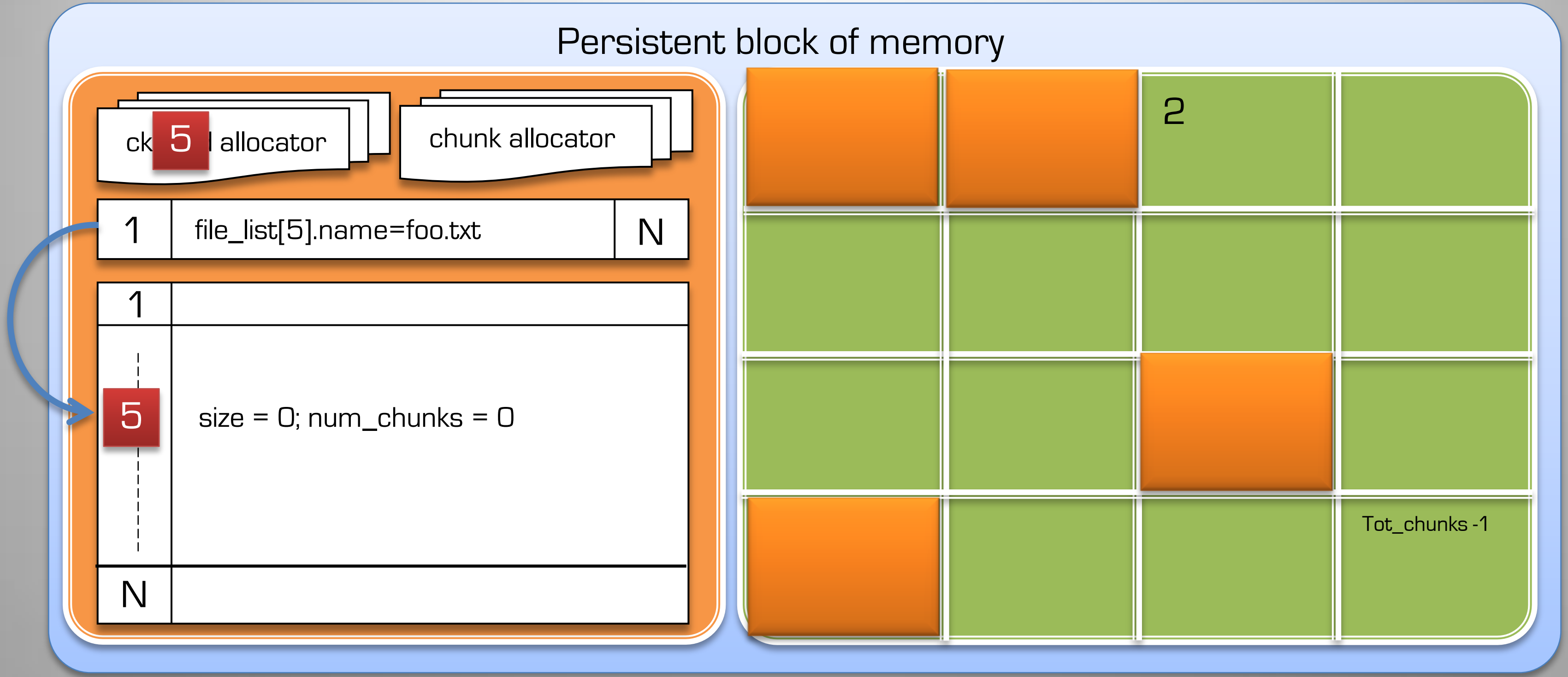
- No-shared files
  - Eliminates inter-process consistency and need for file locking
- Dense files
  - No need to optimize for potential holes
- Write-once-read-rarely model
  - Asynchronous RDMA without ensuring consistency
- Temporal nature of checkpoint data
  - No need to track POSIX timestamps, SCR handles versioning
- Globally coordinated operation
  - Can clear internal locks after a failure

Design of CRUISE

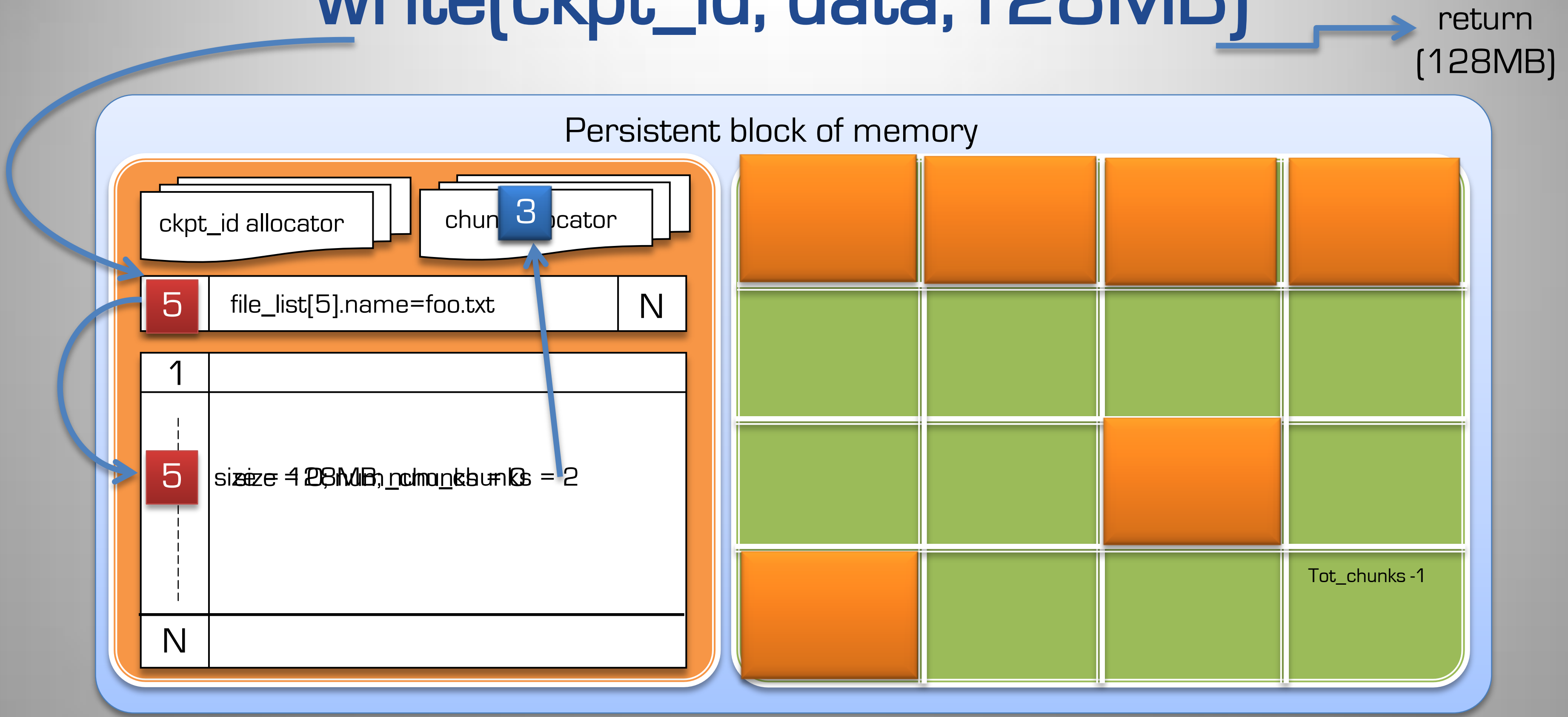


open($CRUISE/foo.txt)

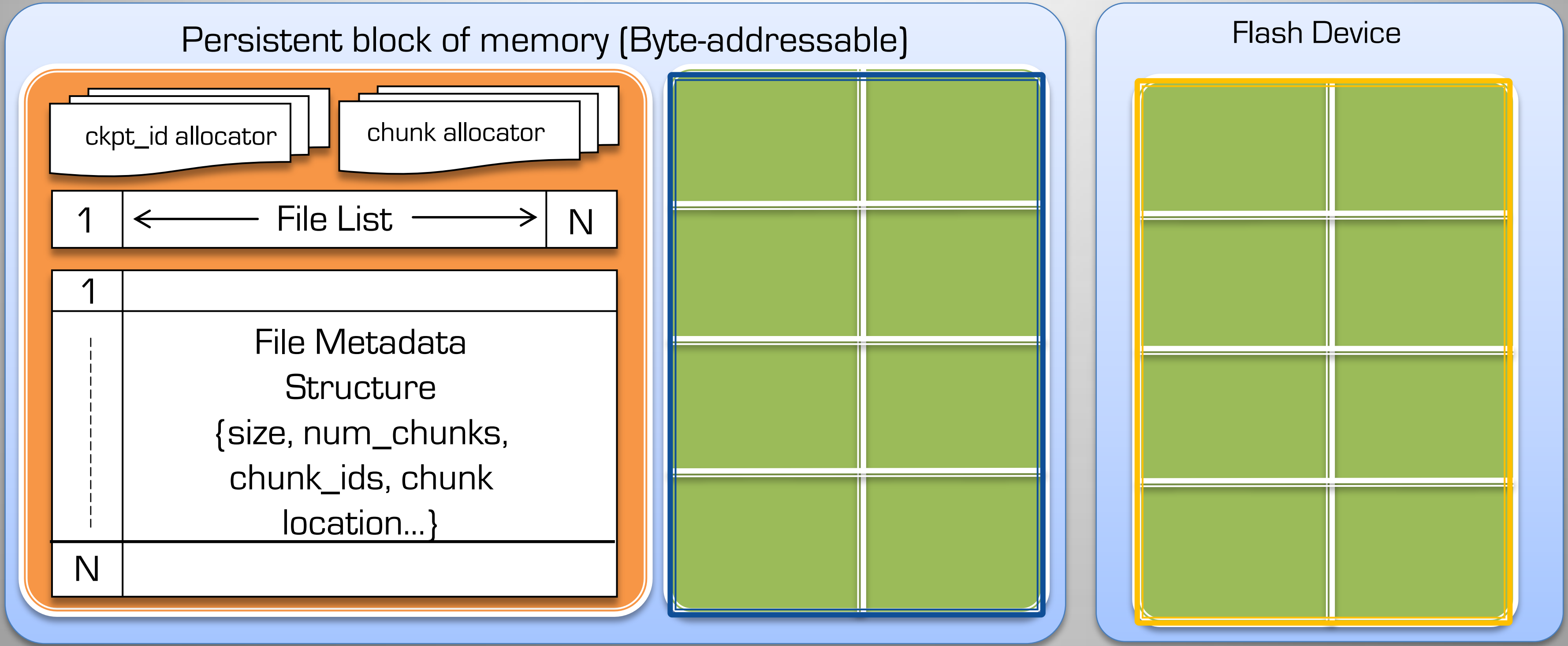
return
(ckpt_id)



write(ckpt_id, data,128MB)



Spill-over to SSD



Spill-over to SSD

$T_{spillover}$ – throughput of cruise with spillover
T_{MEM} – throughput of memory
T_{SSD} – throughput of SSD
$size_{tot}$ – total size of checkpoint
$size_{MEM}$ – size of checkpoint in memory
$size_{SSD}$ – size of checkpoint in SSD

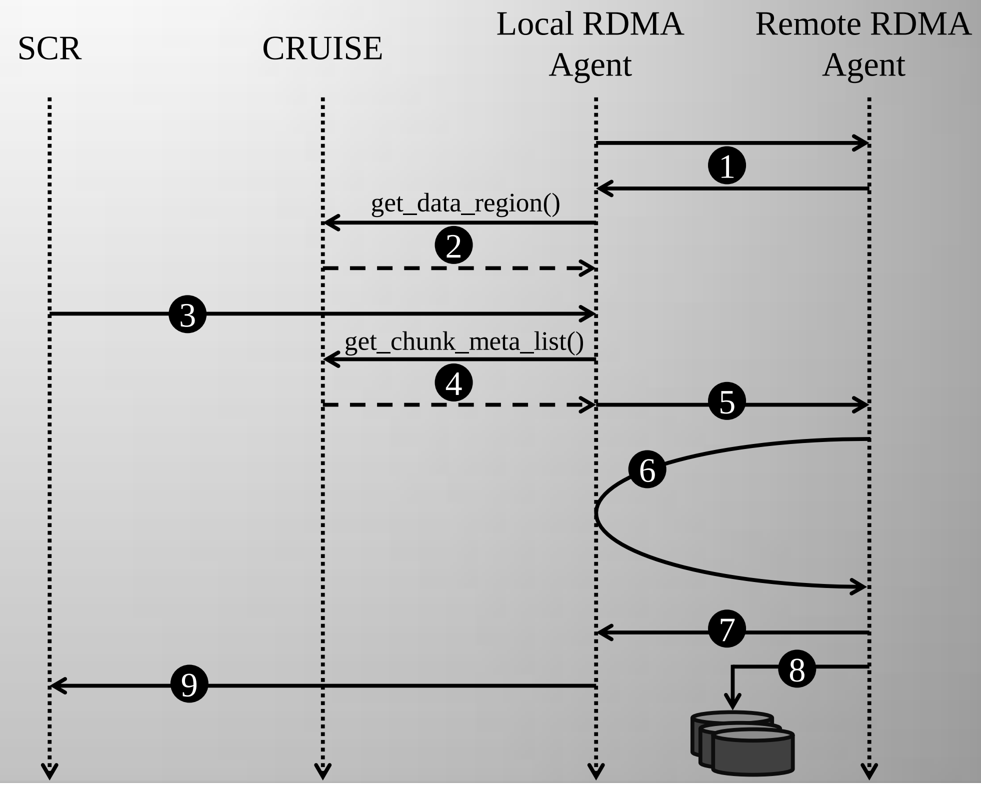
$$T_{spillover} = \frac{size_{tot}}{\frac{size_{MEM}}{T_{MEM}} + \frac{size_{SSD}}{T_{SSD}}}$$

Test #	% in SSD	Spill size (MB)	Expected Throughput	Achieved Throughput
1	0	0	15074.17	15074.17
2	3.125	16	10349.12	10586.61
3	6.25	32	7879.33	8134.46
4	12.5	64	5333.61	5312.26
5	25	128	3240.00	3110.58
6	50	256	1815.06	2163.93
7	100	512	965.67	965.67

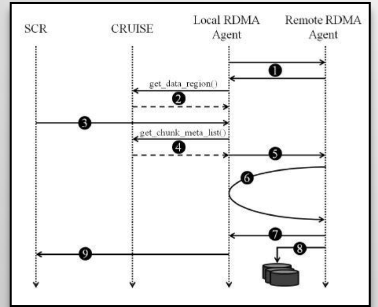
- 300GB OCZ VeloDrive PCIe SSD
- 8 processes writing 512 MB each
- #1,#7 – Native throughput of Memory and SSD respectively

RDMA-based Checkpoint Draining

- Asynchronous I/O without competing with application for CPU resources
- Data-staging architectures
- Job-/Process-migration frameworks



Implementation specifics




```
open(const char *path, int flags, ...)
if path matches CRUISE mount prefix then
    lookup corresponding FileID
    if path not in File List then
        pop new FileID from free_fid_stack
        if out of FileIDs then
            return EMFILE
        end if
        insert path in File List at FileID
        initialize File Metadata for FileID
    end if
    return FileID + RLIMIT_NOFILE
else
    return __real_open(path, flags, ...)
end if
```

Figure 4: Pseudo-code for open() function wrapper

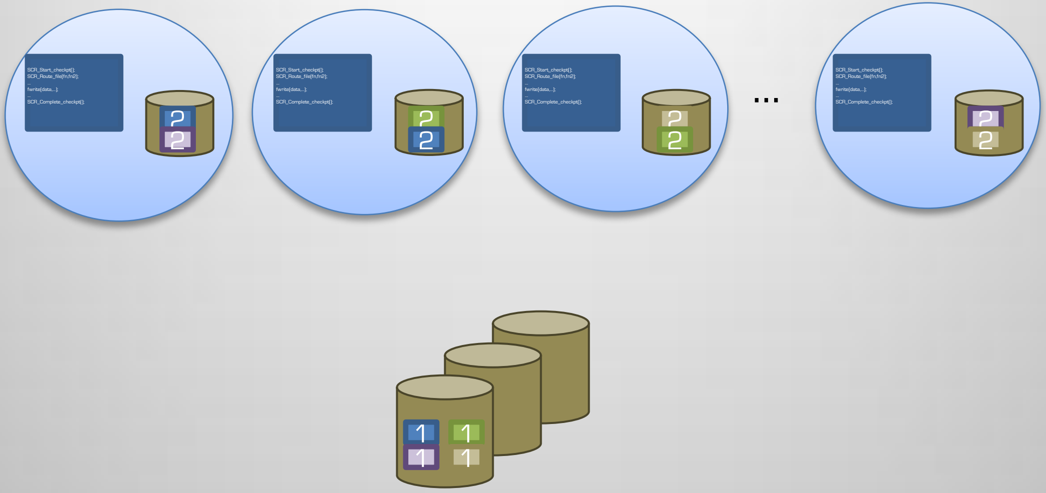
```
write(int fd, const void *buf, size_t count)
if fd more than RLIMIT_NOFILE then
    FileID = fd - RLIMIT_NOFILE
    get File Metadata for FileID
    compute number of additional data-chunks
    required to accommodate the write
    if additional data-chunks needed then
        pop data-chunks from free_chunk_stack
        if out of memory data-chunks then
            pop data-chunks from
            the free_spillover_stack
        end if
        store new ChunkIDs in File Metadata
    end if
    copy data to chunks
    update file size in File Metadata
    return number bytes written
else
    return __real_write(fd, buf, count)
end if
```

Figure 5: Pseudo-code for write() function wrapper

Please refer to the paper for more details on the implementation!

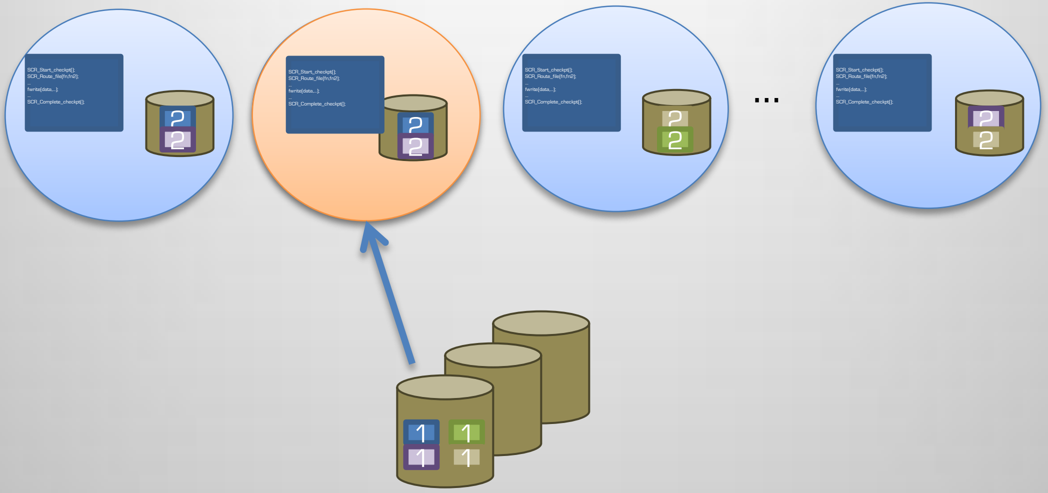
What happens when things fail?

Failure Model – Process Failures



- MPI runtime is required to clean the environment
- Data persists across process death => can be restarted directly

Failure Model – Node Failures

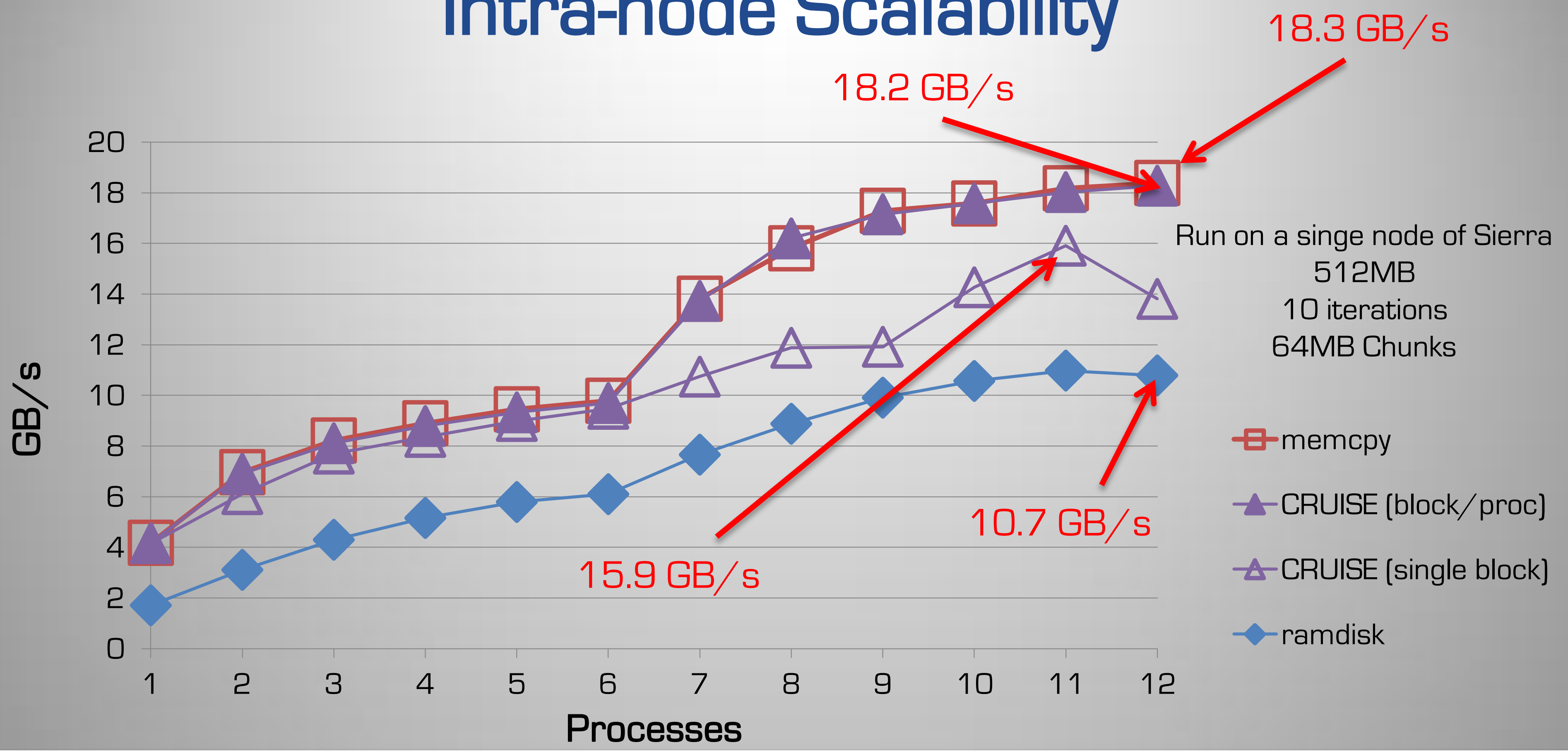


- Redundancy schemes applied by SCR rebuilds lost CRUISE files
- If unable to rebuild, fetches latest copy from parallel file system

Performance Evaluation

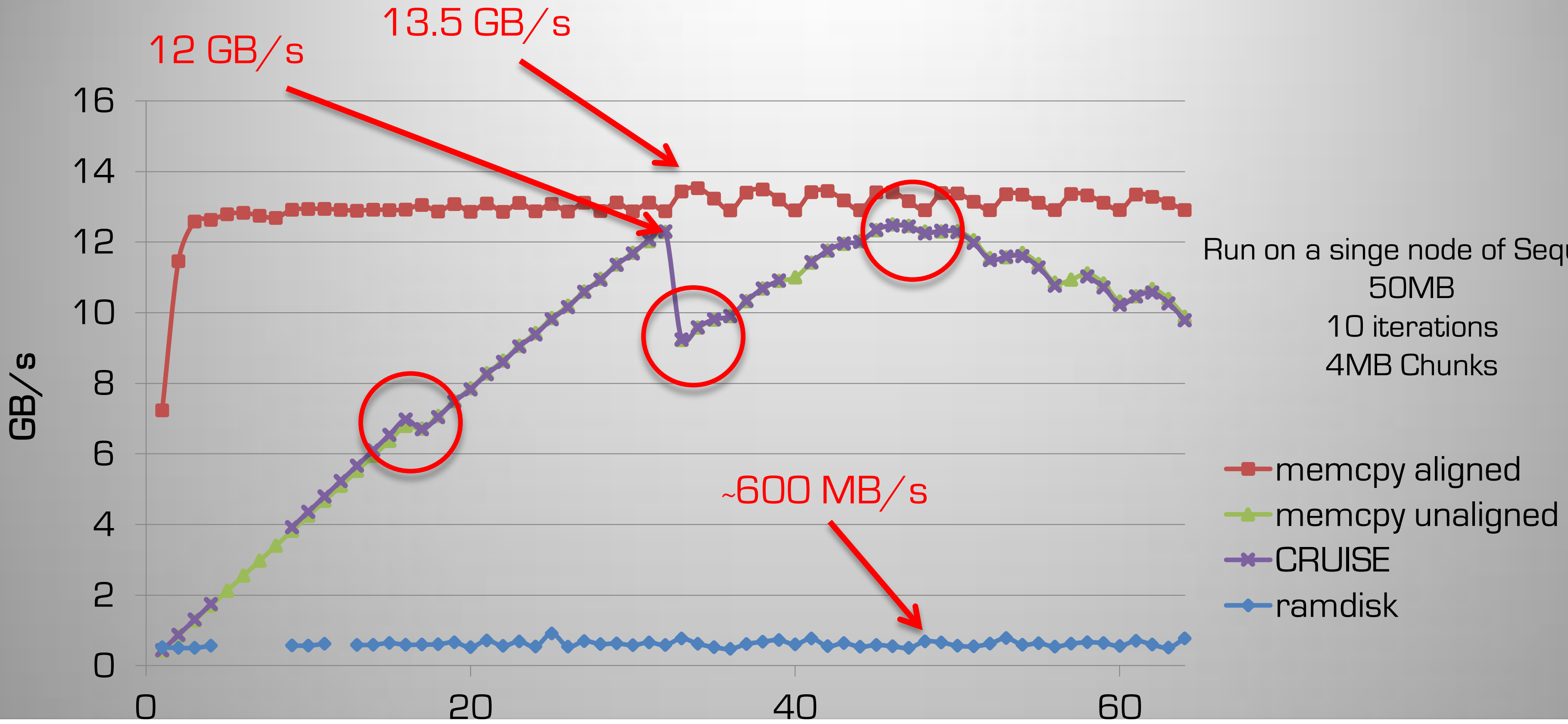
- Sierra (1,944 nodes) - #150 in Top500 (Nov'12)
  - TOSS 2.0 | 6-core Intel Xeon | 24GB RAM/node | InfiniBand QDR | ICC v11.1
- Zin (2,916 nodes) - #29 in Top500 (Nov'12)
  - TOSS 2.0 | 8-core Intel Xeon | 32GB RAM/node | InfiniBand QDR | ICC v11.1
- Sequoia (98,304 nodes) - #3 in Top500 (#1 last year)
  - IBM BG/Q – CNK | 16 cores/node | 16GB RAM/node | IBM BG -Torus | IBM compiler v12.1

Intra-node Scalability



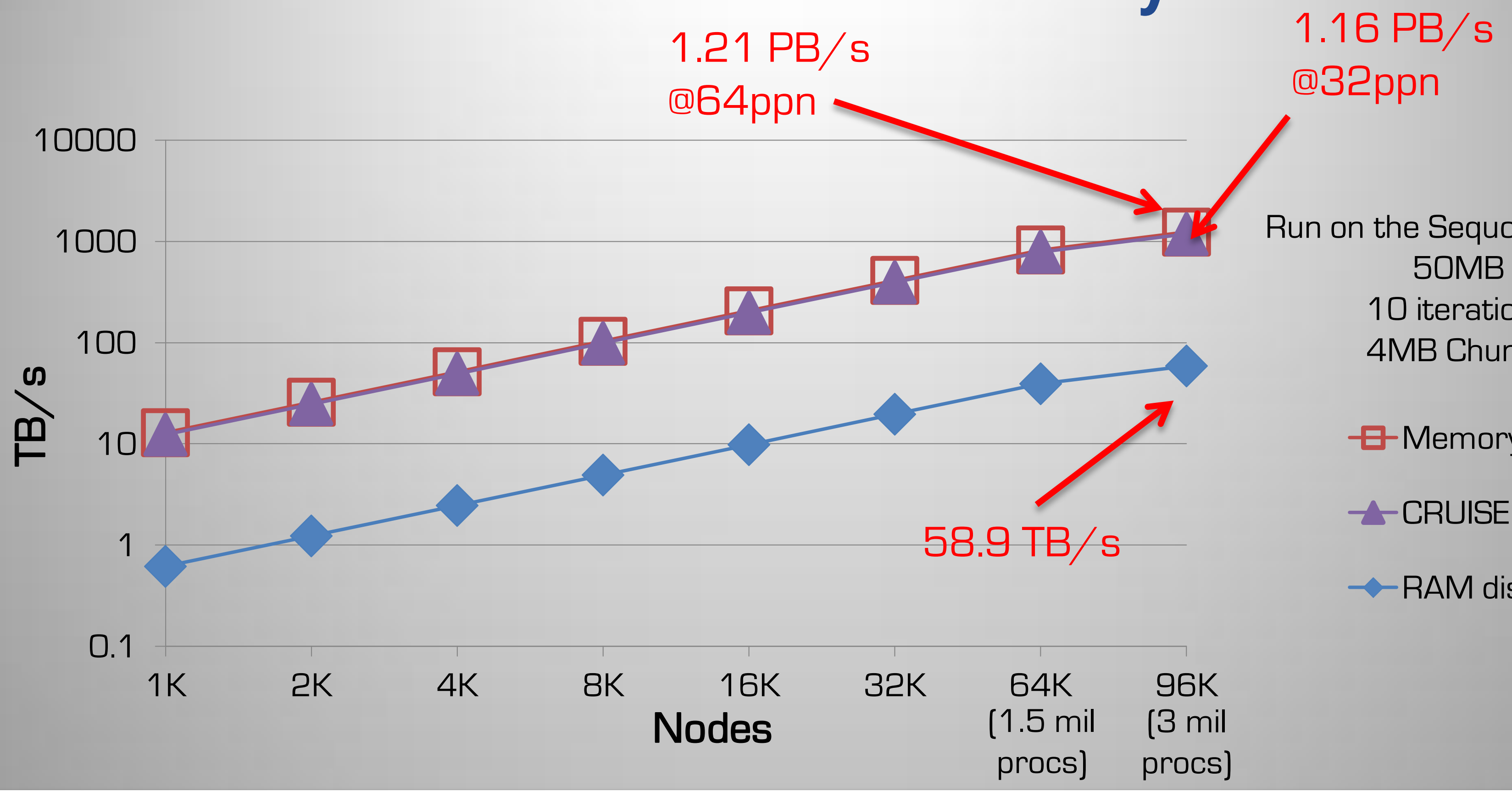
Run on a singe node of Sierra
512MB
10 iterations
64MB Chunks

Intra-node Scalability



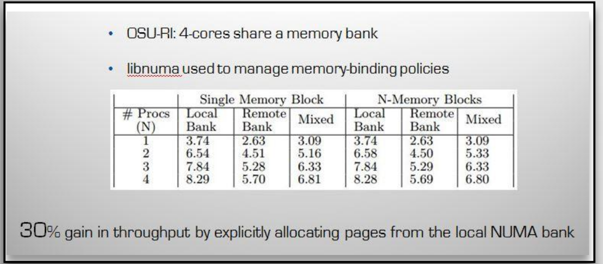
Run on a singe node of Sequoia
50MB
10 iterations
4MB Chunks

Inter-node Scalability

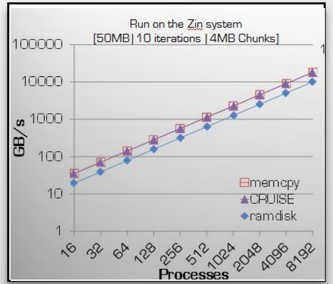


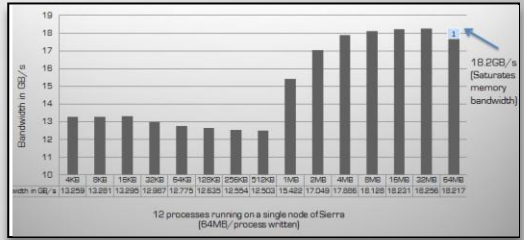
Run on the Sequoia system
50MB
10 iterations
4MB Chunks

Performance Evaluation



# Procs (N)	Single Memory Block: Local Bank	Single Memory Block: Remote Bank	Single Memory Block: Mixed	N-Memory Blocks: Local Bank	N-Memory Blocks: Remote Bank	N-Memory Blocks: Mixed
1	3.74	2.63	3.09	3.74	2.63	3.09
2	6.54	4.51	5.16	6.58	4.50	5.33
3	7.84	5.28	6.33	7.84	5.29	6.33
4	8.29	5.70	6.81	8.28	5.69	6.80





Please refer to the paper for more results!

Summary

- CRUISE: a file system to extend capabilities of multi-level checkpointing systems
- Allows asynchronous draining of checkpoints using RDMA techniques
- Checkpoint data cascades down the storage hierarchy
- **20x faster than RAM disk**, can run on systems without RAM disk
- **Scales linearly** with node-count
- **1.16PB/s** when **three million** processes checkpoint simultaneously

The path forward

- Evaluation with real-word applications (pF3D laser-plasma)
- Enhanced caching policies when data spills-over
- Impact of CRUISE on non-CR application file I/O
- Releasing the CRUISE for use by the community

Thank you!

rajachan@cse.ohio-state.edu

nowlab.cse.ohio-state.edu

THE OHIO STATE UNIVERSITY

Lawrence Livermore National Laboratory