Checkpointing with DMTCP and MVAPICH2 for Supercomputing

Kapil Arya

Mesosphere, Inc. & Northeastern University

DMTCP Developer
Apache Mesos Committer

kapil@mesosphere.io

Joint with:
Jiajun Cao, Gene Cooperman, Rohan Garg, Shawn Matott, DK Panda, Hari Subramoni, Jérôme Vienne

Overview

- Quick Introduction to Checkpointing
  - DMTCP Internals
- Petascale Checkpointing
  - InfiniBand, Storage Backend
- Latest Experimental Results
  - Overheads and checkpoint/restart times

Introduction to Checkpointing

What is Checkpointing?

Checkpoint-Restart is the ability to save a set of running processes to a checkpoint-image on disk, and to later restart it from disk.

Checkpoint-restart involves saving and restoring:

- all of user-space memory
- state of all threads
- kernel state
- network state
- …

Use-cases:

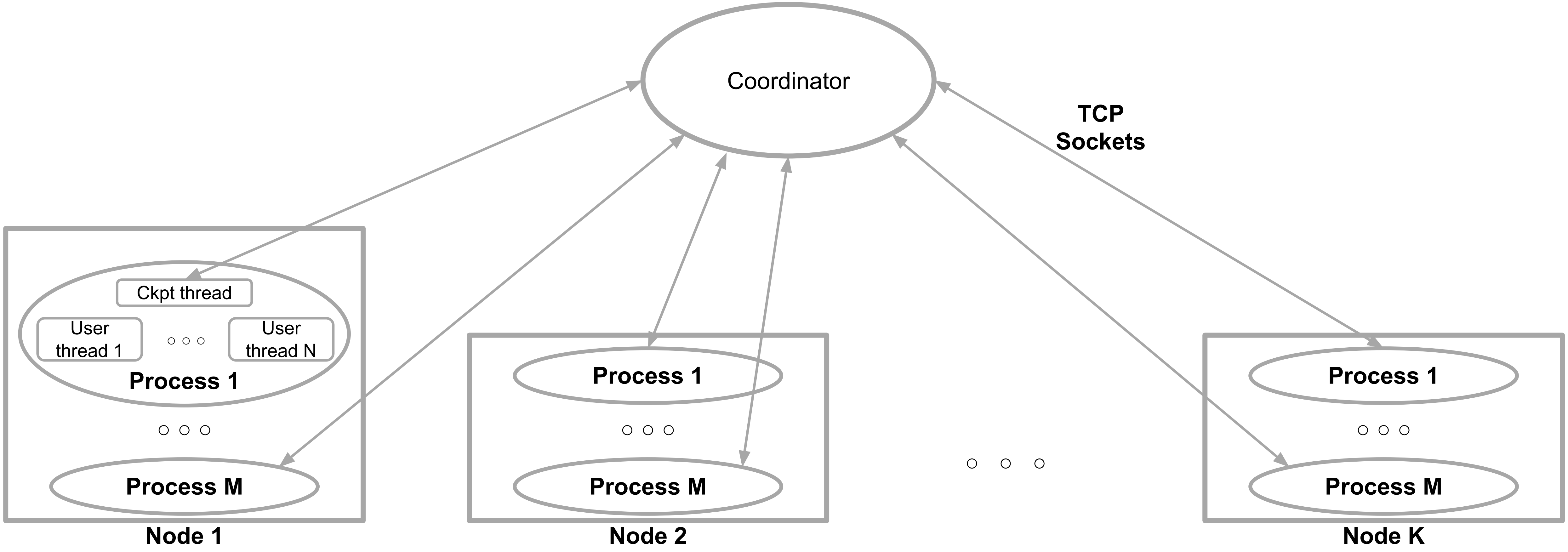
- Fault tolerance
- Scheduling and process migration
- Debugging
- Faster startup times
- Save/restore workspace (for interactive sessions)
- Speculative execution (what-if scenarios)

DMTCP: Distributed MultiThreaded CheckPointing

- Open source system-level checkpointing
- Transparent to the user
  - Works without modifying the source code or binary
- User-space
  - No kernel modules
- Handles distributed applications
  - Centralized coordinator
- Handles MPI libraries, resource managers, process managers, etc.
  - Open MPI, MVAPICH2, Intel MPI, …

dmtcp.sourceforge.net
github.com/dmtcp

DMTCP Architecture



DMTCP Architecture

- Function wrappers
  - `LD_PRELOAD` -- intercept “interesting” library calls (e.g., socket, open, close, etc.)

- Checkpoint thread
  - Quiesce user-threads during checkpoint
  - Drain network
  - Save/restore process state

- Centralized coordinator
  - Key-value datastore via publish-subscribe
  - One IPv4 socket per process
  - Node-local shared-memory arenas
    - Leader-election for shared resources (e.g., fds, files, etc.)

Checkpointing at Petascale

Petascale Checkpointing Concerns

- InfiniBand
  - Hybrid RC/UD Mode
- Storage backend
  - Checkpoint cost
- A surprise!

InfiniBand: Hybrid RC/UD Mode

- RC: Reliable Connection mode
  - Point-to-point connection during initialization
  - Upto $\boldsymbol{n^2}$ connections for **n** MPI processes
  - Performance penalty

- UD: Unreliable Datagram
  - “on-demand” lazy establishment
  - Default for computation with >64 processes
  - Checkpointing requires virtualization of remote-ids
    - Address changes dynamically!

Storage Backend: Lustre

- Full memory dumps are expensive
  - On the order of minutes
- It (mostly) works!
  - Backend was reasonably fast
- A (very) small number of processes contribute to the biggest slowdown
  - More on that later in evaluation

A Surprise: Scaling Issues with TCP Sockets!

- Symptoms:
    - Excessively slow launching with 8K cores
    - Failure to launch at 16K!
        - Some processes getting killed randomly
        - Few unable to connect to coordinator

- Observations:
    - Single 10G ethernet cable per rack (meant for sysadmins)
    - File-descriptor limits set to 16K

Debugging Attempt 1: IP-over-IB

- Assumption:
  - Ethernet is congested
- Suggested workaround:
  - Use IP-over-IB
- Result:
  - Didn’t help!
- Observations:
  - IPoIB uses slower ports on the InfiniBand adapter
  - IPoIB **slower** than IP

Debugging Attempt 2: Staggered Sleep

- Assumption:
  - System can’t handle that many simultaneous socket connections
- Workarounds:
  - Force processes to randomly sleep up to 60 seconds during launch
- Result:
  - It works!

Debugging Attempt 3: Tree-of-coordinators

- Assumption:
  - System can’t handle that many simultaneous socket connections
- Workarounds:
  - Node-local coordinators communicates with all local processes
  - Central coordinator communicates with node-local coordinators
- Result:
  - It works!

Is DMTCP Coordinator the Bottleneck?

- Simulated 20K process launch with coordinator
  - No issues!
  - Registered all clients in < 4 seconds.

Further Investigation Needed!

- The issue seems to be very specific to Stampede
- Occurs only at > 8K cores
- Large-scale reservations aren’t easily available!

Experimental Evaluation

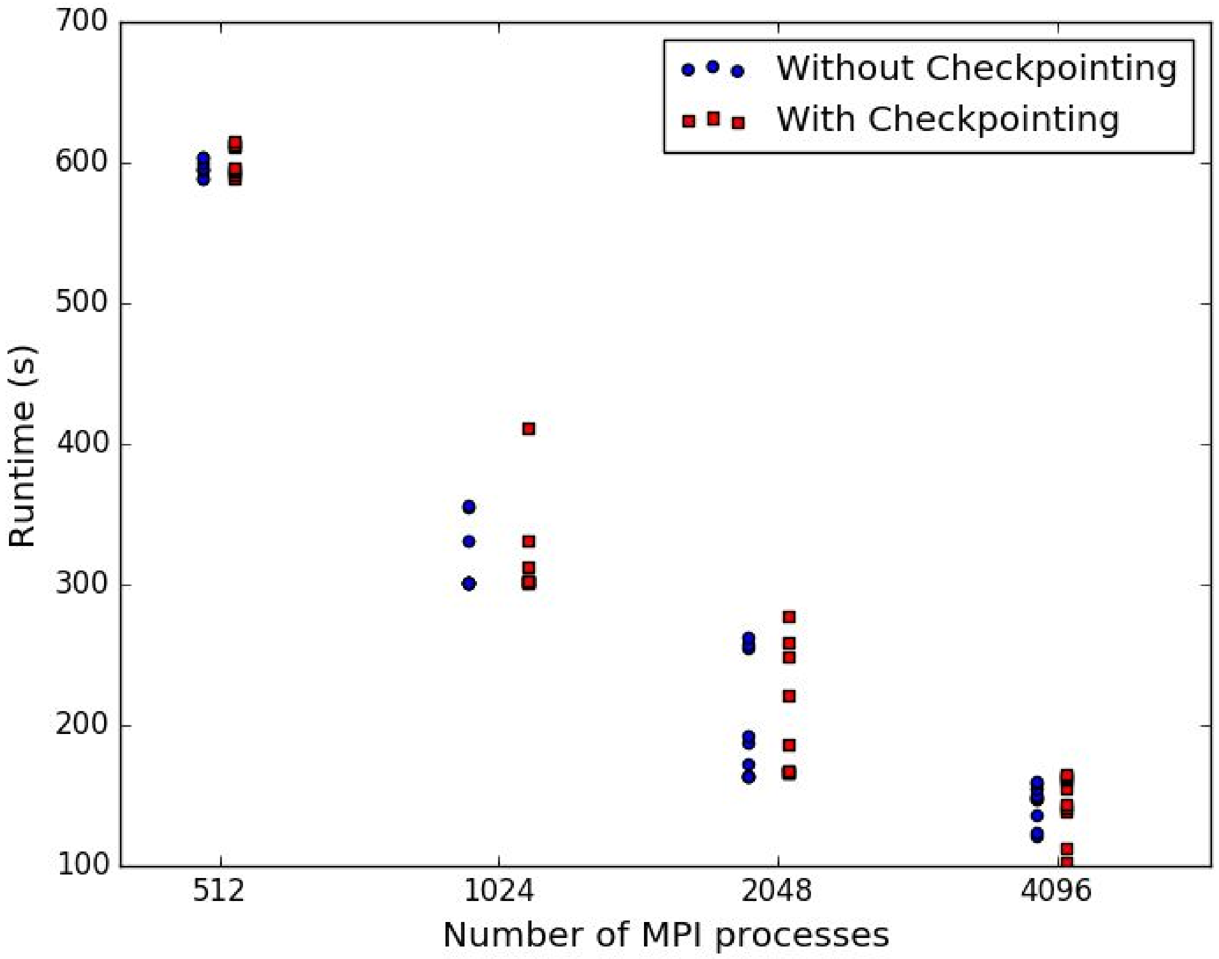
Setup

- Stampede supercomputer at TACC
  - 16-cores, 32GB RAM per node
  - Largest experiment with 24,000 cores (1,500 Nodes)
- Benchmarks:
  - HPCG and NAS LU.E benchmarks
- Metrices
  - Launch overhead
  - Runtime overhead
  - Checkpoint/restart times

Launch Overhead

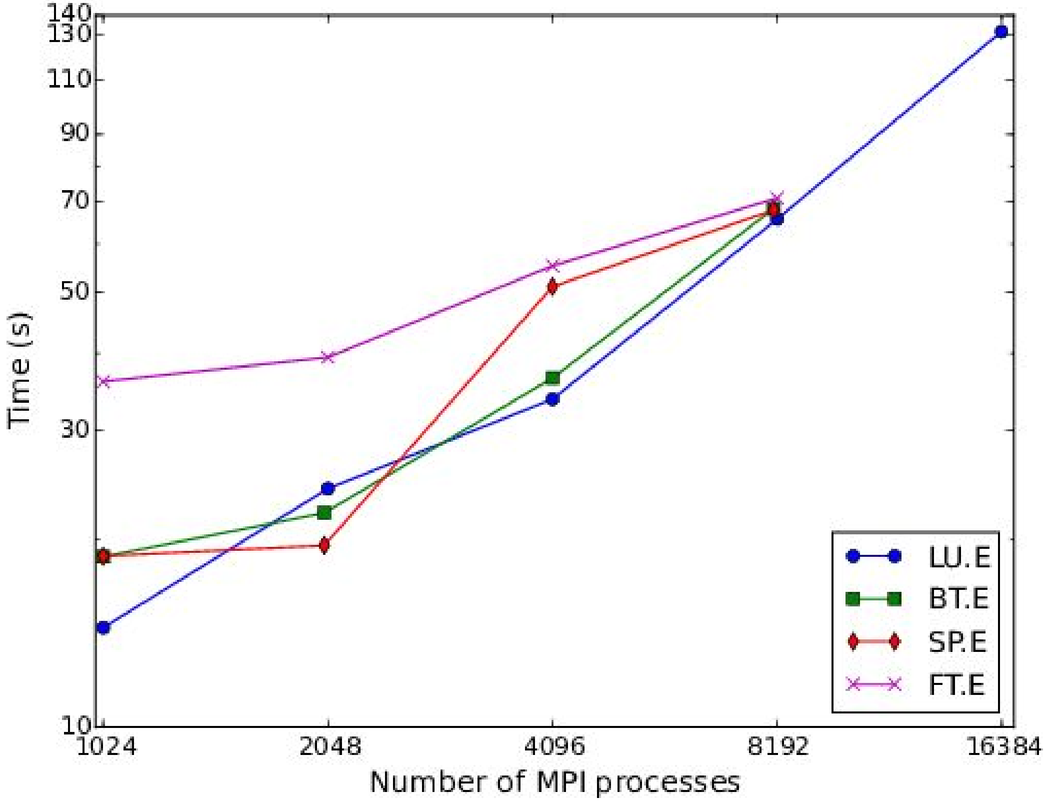
Number of Processes	**Launch Time (seconds)**
1K	0.3 - 7.5
2K	0.8 - 10.5
4K	3.2 - 86.7
8K	29.2 - 87.9
16K	99.3 - 120.8
16K (with tree-of-coordinators)	**15.2 - 21.6**

Runtime Overhead for LU.E

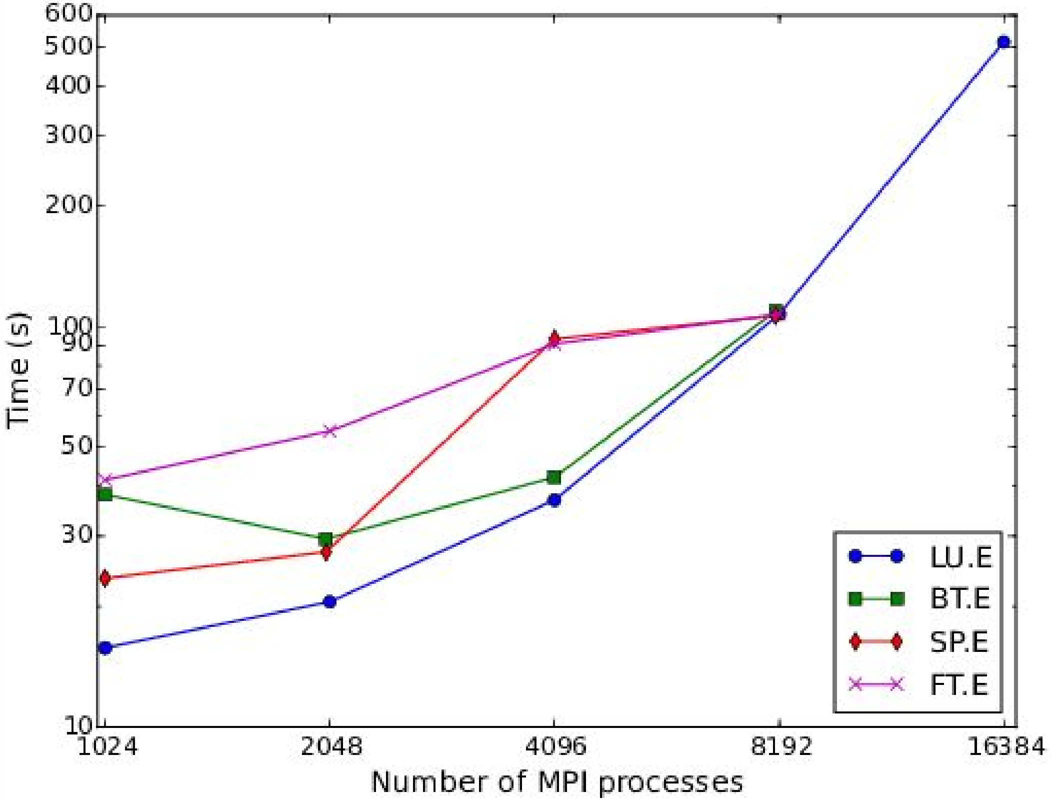


# Processes	Native Runtime (s)	Runtime w/ DMTCP (s)	Overhead %
512	596.6	601.4	0.8
1024	316.2	313.8	0.5
2048	197.6	201.9	2.2
4096	144.0	144.1	0.1

Checkpoint-Restart Times for Various NAS Benchmarks



Checkpoint



Restart

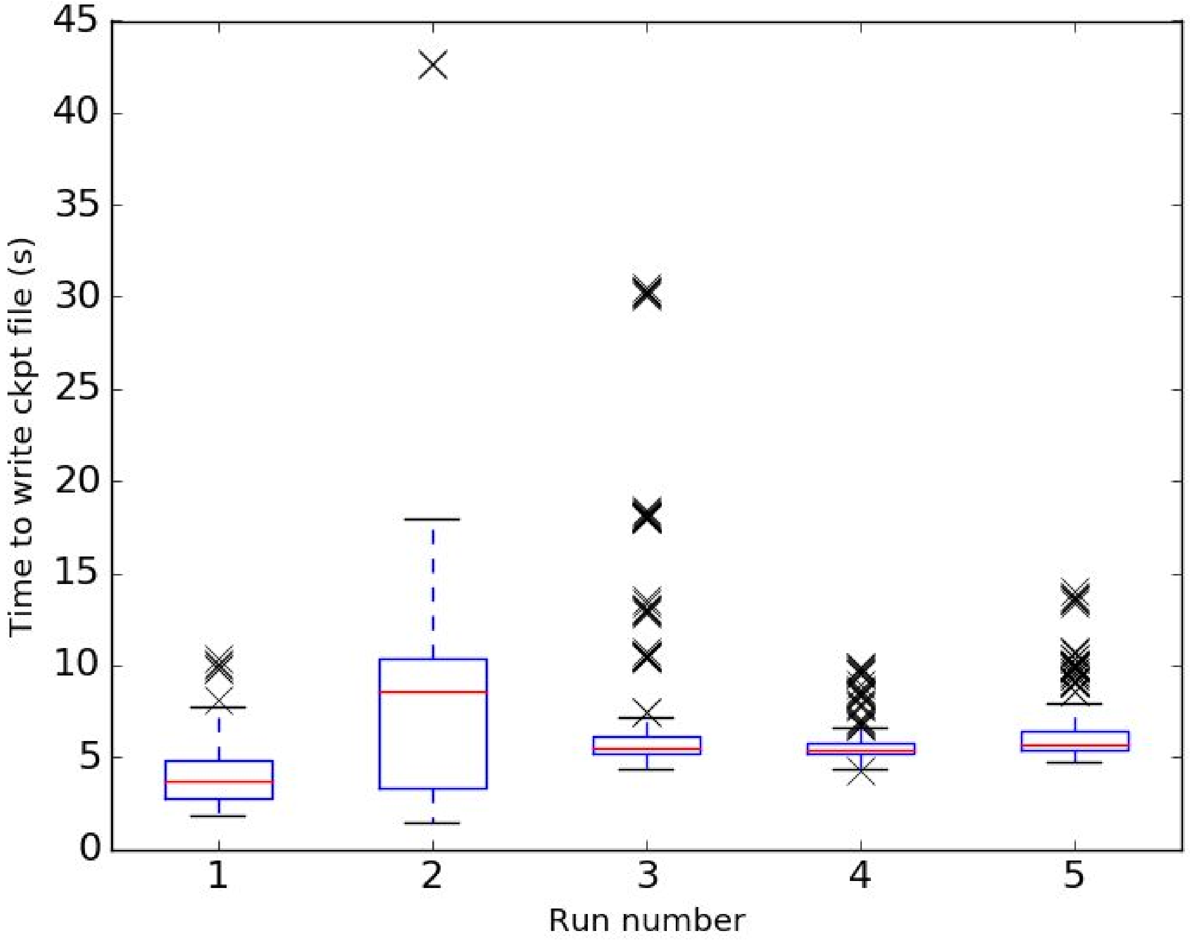
LU.E ckpt image size
(per process)

#Processes	1K	2K	4K	8K	16K
Ckpt size (MB)	428	342	300	280	285

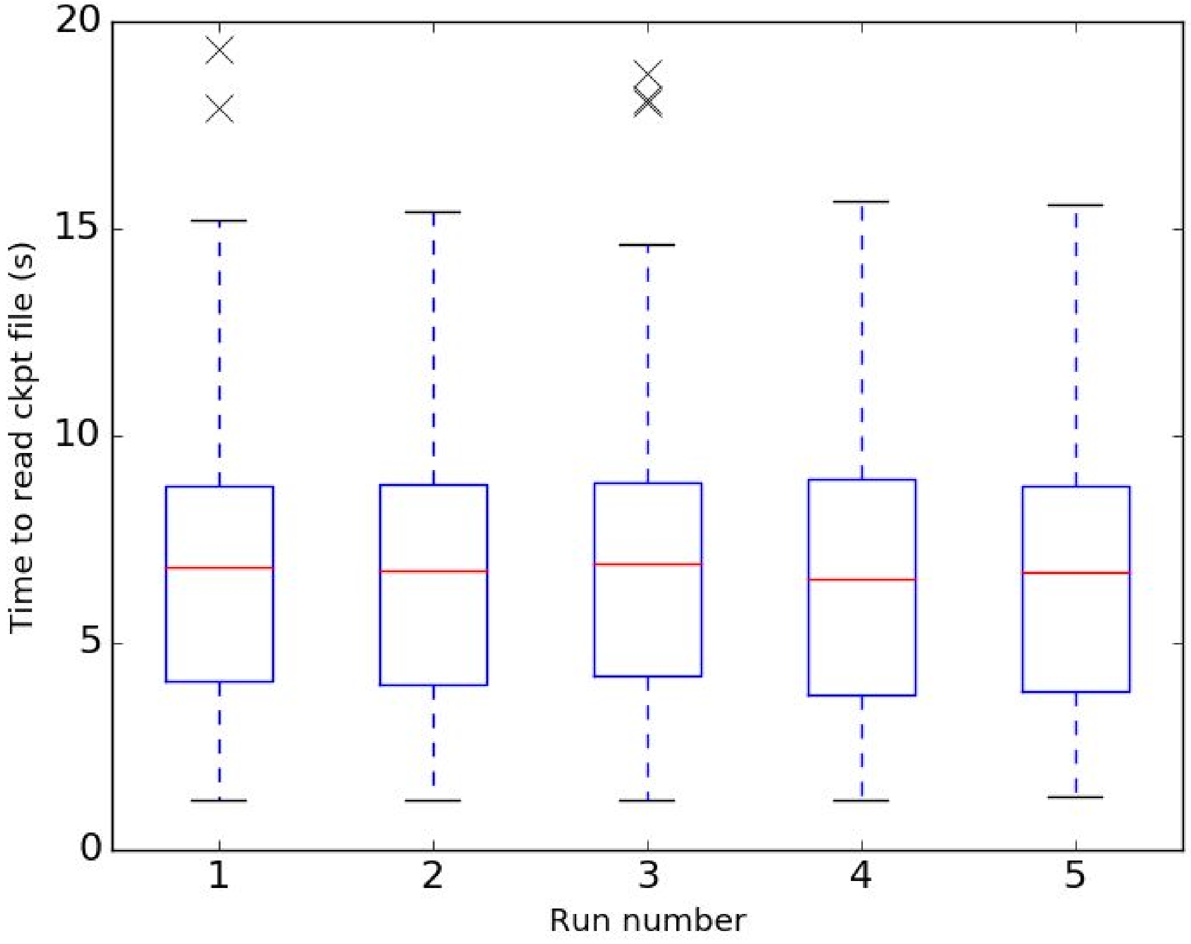
Checkpoint-Restart Times for HPCG

#Processes	Checkpoint time (s)	Restart time (s)	Total checkpoint size (TB)	Write bandwidth (GB/s)
8,192	136.1	215.3	9.4	69
16,368	367.4	706.6	19	52
24,000	634.8	1183.8	29	46

Variation in Checkpoint Image Read/Write Times



Time to write checkpoint image for LU.E.256



Time to read checkpoint image for LU.E.1024

Potential Solutions

- “Don’t try to be smart with Lustre!” -- Luster devs

- Distribute checkpoint creation over time
    - Identify local cliques/groups and checkpoint asynchronously

- Future node-local SSDs
    - Save checkpoints locally (and/or with neighbors) with delayed synchronization

THANK YOU!