Fault Tolerance

2014/11/26

14D54033 – Shweta Salaria (Matsuoka Lab)

Today's Paper

Checkpointing Orchestration: Toward a Scalable HPC Fault-Tolerant Environment

Hui Jin – Illinois Institute of Technology

Tao Ke – Illinois Institute of Technology

Yong Chen – Texas Tech University

Xian-He Sun – Illinois Institute of Technology

CCGrid'12

Outline

- Checkpointing/Restart
- Checkpointing in large scale systems
- Checkpointing Orchestration
- Traditional Checkpointing
- Orchestration Design
- Implementation
- Performance Evaluation
- Related Work
- Conclusion
- Future Work
- Thoughts on Paper

Checkpointing/Restart

- De-facto fault tolerant mechanism for parallel applications
- Periodic checkpoints to store a snapshot of application to a stable storage
- Parallel File Systems (PFS) serves as the storage of checkpoint images
  - mitigates I/O wall problem
- Resumes the application from last checkpoint in case of failures
- Saves work loss

Checkpointing in large-scale systems

- Concurrent I/O requests to PFS in a burst
- Data access contention
  - excess data access
- I/O contention
  - no. of compute nodes > no. of I/O servers
- More processes, Higher Contention
- Limits scalability

Checkpointing in large scale systems

Impact of I/O Contention:
A cluster of 32 compute nodes, 4 I/O server nodes, PVFS2, Open MPI, synthetic parallel application, 16 GB checkpoint size

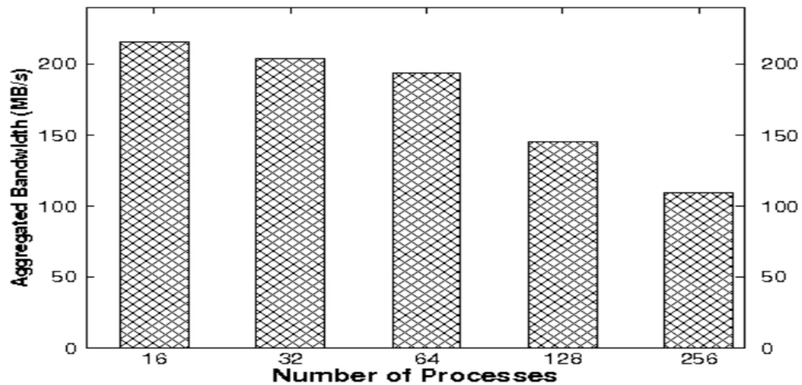


Figure 1. Aggregated Bandwidth under Contentions

Average bandwidth was halved when the number of processes were increased from 16 to 256

Checkpointing in large scale systems

- I/O contention as the dominant performance factor
- Checkpointing scalability
  - limits scalability of applications
- Challenge: optimization of checkpointing under existing hardware and software stack to maintain its feasibility at post-Petascale.
- Proposed Solution: Checkpointing Orchestration

Checkpointing Orchestration

- Objective is to reduce contention caused by burst of checkpoint requests
- Two-fold orchestration

1. Vertical checkpointing
   - rearranges the data layout of checkpoint files on PFS
   - reduces data access contention
2. Staged checkpointing marshaling
   - serializes the concurrent checkpoint on each compute node
   - reduces I/O contention

Traditional Checkpointing

- Type
  - Coordinated, Uncoordinated
- Level
  - Application-level, system-level
- Pattern
  - N-N, N-1

Traditional Checkpointing

- Data striped over multiple I/O servers
  - facilitates fast processing time of single checkpoint
- PFS client on each compute node
  - captures I/O requests to/from I/O server
- A burst of write requests by data intensive application
- Services requests in round-robin fashion
- Overhead
  - context switch
  - contention causes physical disk head movement
- Processing time of one single checkpoint represents overall performance
  - need to reevaluate role of stripping

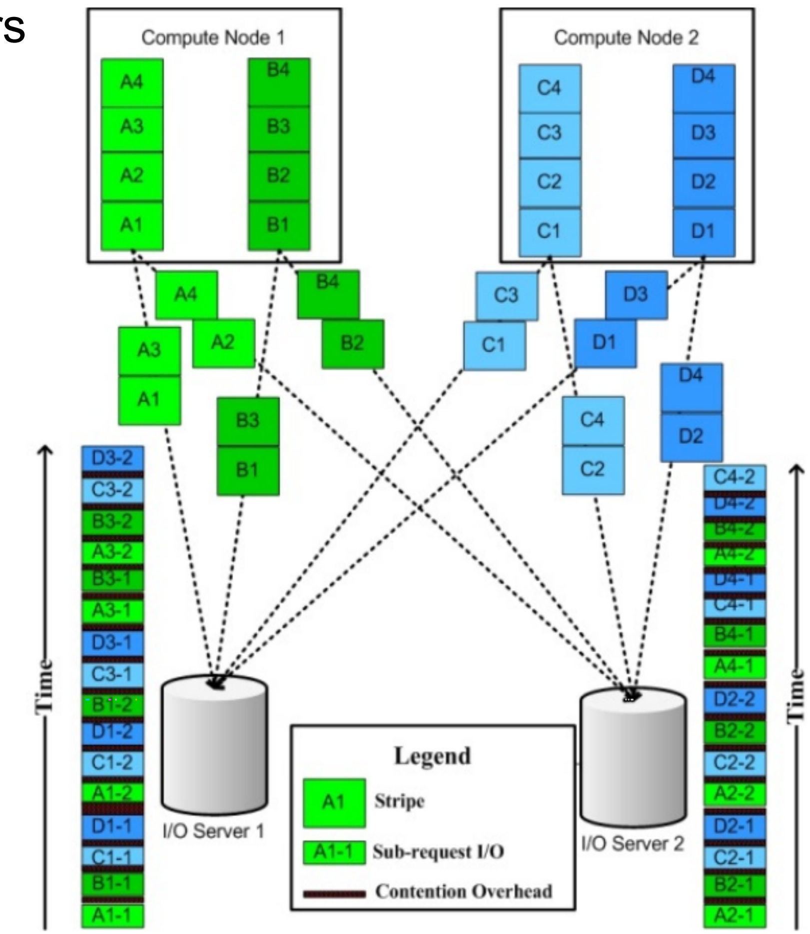


Figure 2. Traditional Checkpointing

Orchestration Design
Vertical Checkpointing

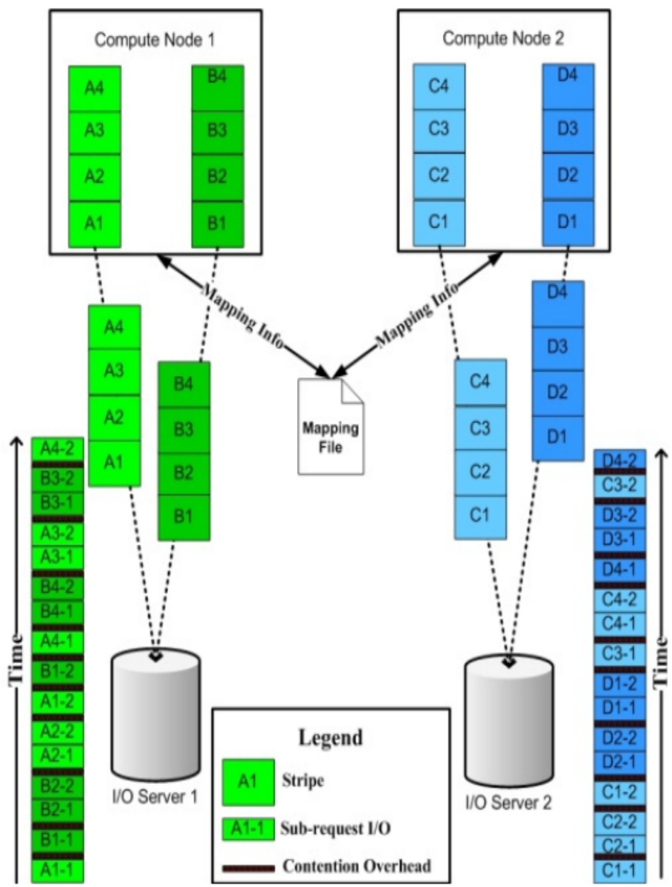


Figure 3. Vertical Checkpointing

- Disables stripping
- One dedicated I/O server for each checkpoint
  - reduces contention
- Mapping File
  - hashes PFS clients to PFS servers
- Works well for well optimized MPI applications
  - each I/O server with same no. of compute nodes & associated checkpoints
- Irregular workload
  - need to work on mapping file
- Reduces no. of checkpoint requests served by each I/O server
- Lessens cost of coordination among I/O servers
- Problem: I/O interleaving of checkpointing requests

Orchestration Design
Staged Checkpointing Marshaling

- Serializes checkpoints on each compute node
- Staging phase
  - stages checkpoint to local memory
  - mitigates the impact of small VFS writes
  - operates in memory and thus faster
- Flushing phase
  - flushes checkpoint from local memory to PFS server
  - mutex to govern multiple checkpoint requests in a stream
  - reduces contention

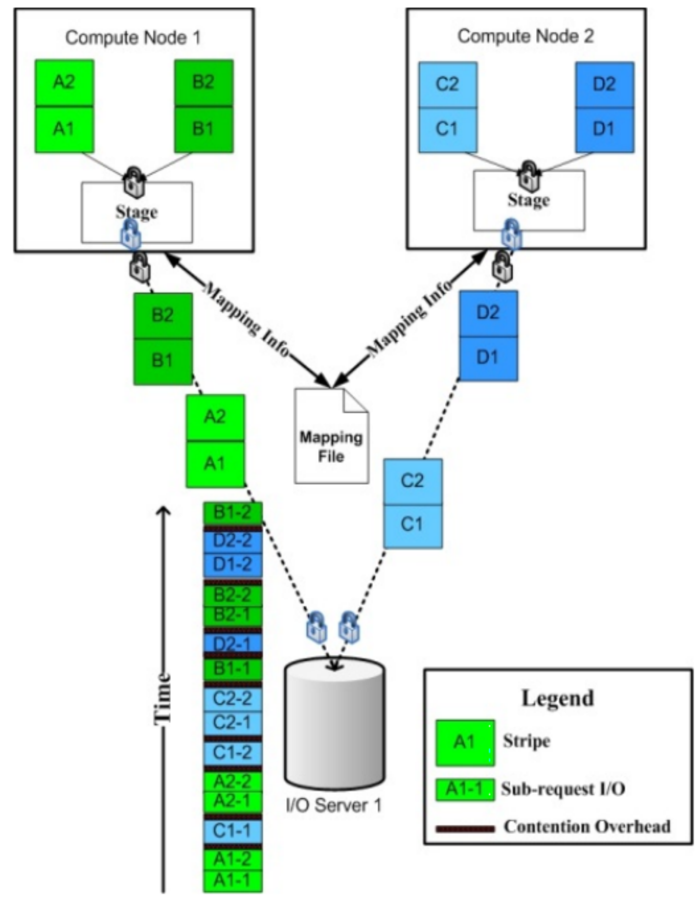


Figure 4. Staged Checkpointing Marshaling

Orchestration Design

Staged Checkpointing Marshaling

Wait(StageMutex)

Stage Checkpoint onto local memory

Wait(PFSMutex)

Signal(StageMutex)

Flush checkpoint to PFS server

Signal(PFSMutex)

Algorithm 1. Pseudo Code for Staged Marshaling

- StageMutex limits one checkpoint process for staging at one time
- PFSMutex marshals the concurrent checkpoints in a serialized manner
- Interleaved mutexes
  - avoids excessive memory usage
- Different compute nodes processes competes concurrently for shared I/O server
- Marshalling checkpoint from all compute nodes that share one I/O server slows down the performance
  - the lag of current checkpoint delays all other checkpoints

Implementation

- Vertical checkpointing implementation
  - PVFS2
  - directory attributes reset to enforce single I/O server access
  - each checkpoint processes the mapping file and piggybacks hashed I/O server information in the hint field of PFS client
  - services both regular I/O request and checkpoint I/O request
- Staged checkpointing marshaling
  - Open MPI
  - fcntl system call for mutex locks
  - ram-based file as the mutex lock file
  - lock file is shared by a limited no. of processes inside one compute node

Performance Evaluation

Test Environment

- A cluster of 32 Sun Fire Linux-based compute nodes
- Dual 2.7 GHz Opteron quad-core processors
- 8 GB memory, 250 GB SATA hard drive
- 1 Gigabit NIC, fat tree topology
- Open MPI v1.4 as the MPI
- NAS Parallel Benchmark (NPB) as parallel application
- PVFS2 (4 I/O server nodes)
- 64 KB stripe size
- Each I/O server is also a metadata server

Performance Evaluation

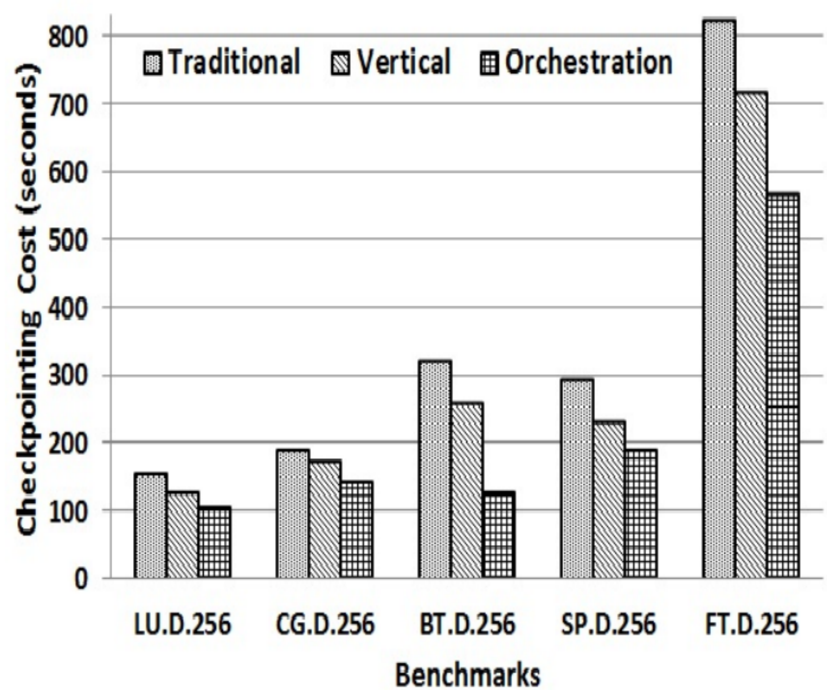
Performance with Different Benchmarks

Problem Size	Class=C	Class=D			
Benchmarks/# of Procs	256	32/36	64	128/196	256
LU	2.5GB	12GB	12GB	14GB	16GB
CG	2.1GB	20GB	20GB	21GB	22GB
BT	4.2GB	26GB	28GB	31GB	32GB
SP	3.7GB	22GB	24GB	27GB	28GB
FT	9.3GB	N/A	81GB	81GB	82GB

Table I: Benchmarks and the Overall Image Size (GB)

- Checkpointing orchestration saved 254 seconds for benchmark FT

Performance with Different Benchmarks



- 157.41 -> 105.99 seconds for LU
  - speedup close to 30%

Performance Evaluation

Task Scaling Performance

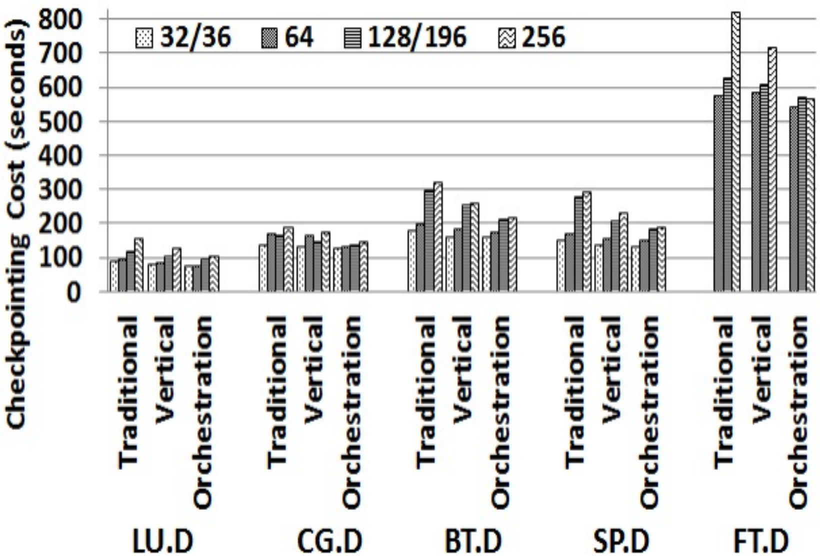


Figure 6. Task Scaling Performance (class=D)

- Overhead increases was less than 15% for LU and CG when the no. of processes are doubled
- Gap b/w traditional & orchestration is enlarged as the no. of processes increase

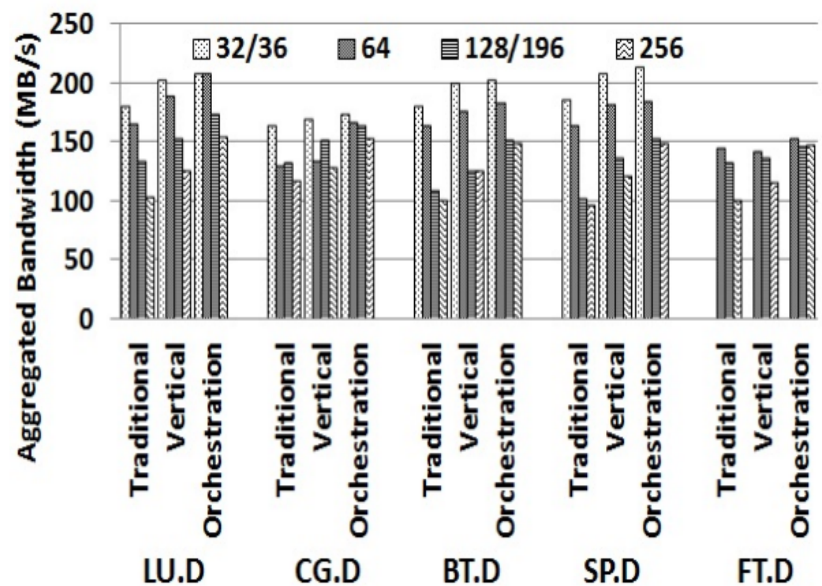


Figure 7. Task Scaling Bandwidth (class=D)

- Both traditional & vertical checkpointing exhibit bandwidth degradation
- Orchestration shows relatively stable bandwidth for CG & FT
- Traditional : 50% bandwidth reduction
- Orchestration : less than 25%

Performance Evaluation

Problem Size Scaling Performance

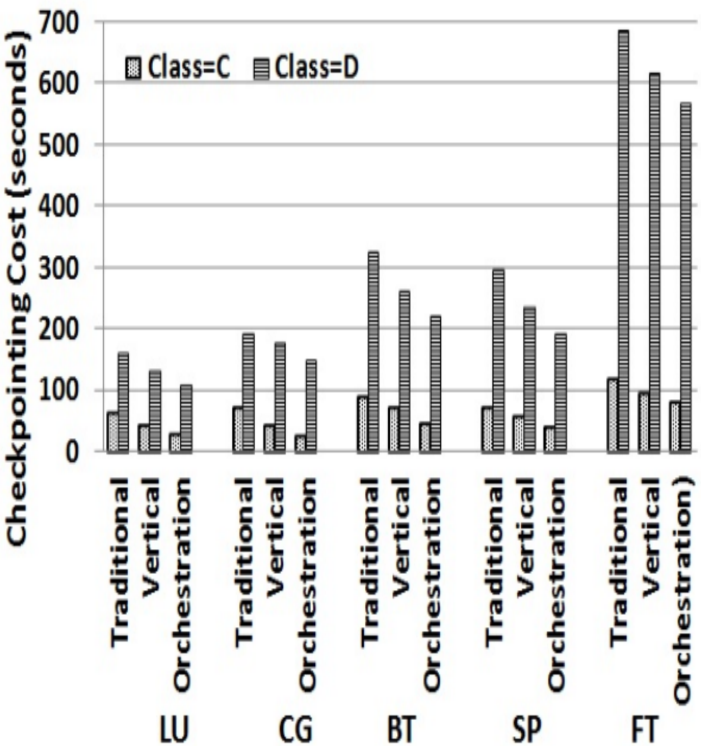


Figure 8. Problem Size Scaling Performance

- As problem size increases
  - checkpointing cost increases
  - advantage of orchestration drops
  - I/O overhead increases
  - contention overhead doesn't increase at the same pace
- Low performance improvement for class D problems

Related Work

- File system optimization for checkpointing
  - Lightweight File System (LWFS), Parallel Log-Structured File System (PLFS)
  - No consideration for I/O contention
  - Collective I/O, data sieving – implemented in MPI-I/O
  - Most checkpointing utilities adopts POSIX API
- Checkpointing System Optimization
  - Modifying coordination protocols
  - aggregating the write requests
  - No consideration for concurrent parallel checkpoints

Conclusion

- Controlled management of both PFS and checkpointing system
- PFS – customize data distribution to reduce data access contention
- Checkpointing system – reorganize checkpointing order to avoid I/O contention
- Considers mixed workloads of the system
- ORCHECK software

Future Work

- Checkpointing orchestration for large-scale computing environment
- PFS on emerging storage media such as SSD
- Build a coordinated framework that facilitates both checkpointing and parallel file systems

Thoughts on Paper

- Checkpointing orchestration over traditional checkpointing
  - increases aggregated bandwidth
  - reduces contention
  - scalable to some extent
- Low performance improvement as no. of processes increase
- Problem size increases
  - I/O increases
  - checkpointing cost increases
  - overhead on I/O server increases (verticalization)