File Aggregation for Asynchronous Multi-Level Checkpointing

Author: Mikaila J. Gossman
Advisor: Dr. Bogdan Nicolae
Advisor: Dr. Jon C. Calhoun
Email: mikailg@g.clemson.edu

MOTIVATION

» **Checkpointing serves** numerous **defensive, administrative, and productive** tasks making it a **fundamental I/O pattern** of HPC applications
» Checkpointing involves **large quantities of data being written to stable storage** such as a parallel file system (PFS) ***concurrently***
» Historically, checkpointing is performed **synchronously** – e.g. blocking the application from progress until the checkpoint is persisted to the PFS
  ✓ Eliminates competition for finite resources (CPUs, network bandwidth, memory)
  × Suffers from increasing synchronization overhead at large scale
» **Asynchronous checkpointing** is becoming the default method for checkpointing – e.g. use multi-level storage tiers to capture checkpoints locally (DRAM, NVMe, etc.), then persist them (PFS) in the background concurrently with the application
  ✓ Reduces time the application is ***directly*** blocked
  × Introduces competition for finite resources which may ***indirectly*** block the application

VELOC

» **VELOC** [1] is a state-of-the-art **asynchronous multi-level checkpointing library**
» Natively adopts one of the following flushing strategies:
  » **N-to-N (file-per-process):**
    ✓ Low complexity
    × Oversubscribes I/O resources (metadata servers, storage devices) as N grows
    × Difficult for users to manage
  » **N-to-1 :**
    ✓ Easier for users to manage
    × Contention for storage devices as N grows
    × Data layout is vital

COMPARISON OF FLUSHING STRATEGIES

» **N-to-M chooses a subset of processes (M) as *I/O leaders*** to interact with I/O infrastructure on behalf of all processes
  ✓ Reduces contention for storage devices
  ✓ Reduces metadata concurrency
  × May require communication and/or synchronization
» **N-to-M alleviates serialization bottlenecks of N-to-1** and **over-utilization of resources characteristic of N-to-N**

Strategy	No I/O Locks?	No Communication? (Synchronization)	Scalable?	User Manageability
N-to-N	✓	✓	x	x
N-to-1	x	✓	x	✓
N-to-M	✓	✓	✓	✓

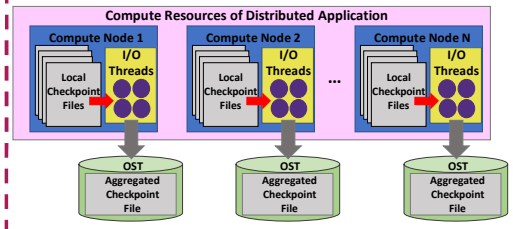
✓ = good, ✓ = depends on implementation, x = bad

CHALLENGES

» Current **aggregation libraries (MPI-IO) are not designed for asynchronous workflows** [5]:
  × High, frequent synchronization costs for collective operations
  × Unbounded resource competition (CPU cores, network bandwidth, memory) with the application
» Multi-threaded implementations are needed to overcome serialization but require synchronization
» How to:
  1. **Avoid I/O locks and contention for storage targets?**
  2. **Reduce synchronization overhead between threads?**
  3. **Minimize impact on concurrently running workloads?**
  4. **Maximize the I/O throughput of the checkpoint flush?**

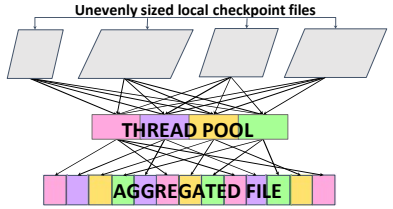
CONTRIBUTIONS

» **Prototype an optimized, tunable flushing aggregation implementation** within VELOC
  » Selects M compute nodes (tunable by user) as I/O leaders
  » Leaders flush checkpoints to the PFS in fine-grained chunks avoiding over-utilization of resources
» Remove communication overhead between compute nodes by letting **M = # compute nodes** and combine local checkpoints of co-located processes on the same compute node into 1 file
» Analyze performance of our aggregation strategy compared to expected peak performance
» Compare our aggregation to state-of-the-art aggregation libraries ADIOS2 [3] and an optimized implementation of MPI-IO (GenericIO [2])



FLUSH STRATEGY

Spawn a number of I/O threads equal to the number of local files which process each file sequentially in small chunks (64 MB) until all local data has been transferred



✓ Eliminates busy-waiting or idleness on fast threads
✓ Low memory footprint avoids interference with concurrent workloads
× Incurs OS-level and thread-level synchronization costs
× Contention for OSTs (mitigated if stripe size = chunk size)

TESTING METHODOLOGY

» To compare our aggregation strategy to other aggregation libraries/strategies we **develop a checkpointing microbenchmark** using 64 processes across 8 nodes (8 processes per node); each process checkpoints 1 GB (+/- 20%)
» **Find expected maximum aggregated throughput when aggregating eight 1 GB files of data** (= local data per compute node in our microbenchmark) to 1 file with 8 threads via:
  » Linux system utility call `dd` (a wrapper for sys. call `write`)
  » OpenMP microbenchmark [4]
» We calculate the maximum aggregate throughput via:

$$\frac{\text{total checkpoint size } [GB]}{\text{slowest thread } [s]}$$

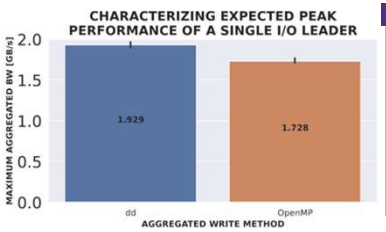
» Run each experiment 3 times and average
» We configure ADIOS2 to flush files under the same constraints as our aggregation strategy (e.g. 8 I/O threads and 64 MB buffers)

SYSTEM OVERVIEW

» Experiments run on Oak Ridge National Lab's **Frontier**:
  » Lustre parallel file system with maximum aggregated bandwidth of 14 TB/s
  » 1350 OSTs : 450 OSS'
  » 9,408 computes nodes each with a 64-core AMD EPYC CPU and 2 hardware threads per core
» VeloC version 1.7
» GenericIO git tag 20190417
» ADIOS2 version 2.8.3

RESULTS

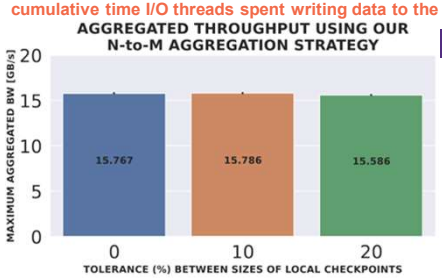
This experiment profiles the expected peak performance of an I/O leader using `dd` and OpenMP



Profiling OSTs with dd and OpenMP

» **dd shows an expected maximum aggregated bandwidth ~2 GB/s** per I/O leader when the block size = 64 MB
» **OpenMP benchmarks show a maximum aggregated BW of ~1.7 GB/s**
» In our 8 node scenario, **we expect that our aggregation should be able to obtain a maximum aggregated BW of ~16 GB/s** (2 GB/s * 8 nodes)

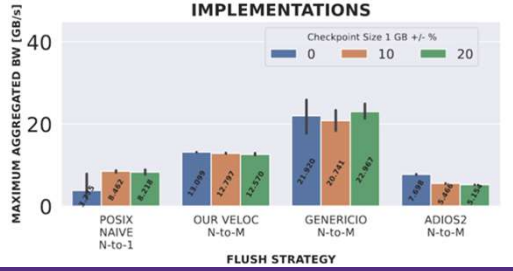
This experiment uses fine-grained timers to calculate cumulative time I/O threads spent writing data to the PFS



N-to-M Fine-Grained Flush Analysis

» **Our aggregation method performs better than expected, obtaining close to 16 GB/s**
» Likely due to the fact **our aggregation strategy is aware of timing imbalance among threads**
» In dd and OpenMP each thread flushes exactly 1 GB of data, however, our strategy allows **faster threads to process more work, ultimately finishing faster**

This test we compare our aggregation strategy against the naive N-to-1 aggregation currently supported by VELOC and two state-of-the-art aggregation libraries, GenericIO and ADIOS2, and calculate throughput via the slowest application process to persist the checkpoint



THROUGHPUT COMPARISON OF AGGREGATION STRATEGIES

» Because the throughput is now measured from the application side, which includes other function calls, we see a small drop in throughput for our aggregation strategy
» **Our aggregation strategy outperforms N-to-1 aggregation by ~1.5x**
» **Because GenericIO is synchronous** (no regard for concurrent workloads), **they get ~2x higher throughput** by mapping the full file into memory and reducing their collective calls, but this is unsuitable for asynchronous workflows like VELOC checkpointing
» **We obtain ~2.5x higher throughput than ADIOS2**

CONCLUSIONS

» **We implement a novel aggregation strategy within VELOC** and profile it's performance on Frontier
» **Our aggregation strategy obtains slightly better throughput than expected**, likely due to our thread management which anticipates timing imbalance among threads
» **GenericIO implements synchronous C/R, blocking the application 30x longer compared to VELOC**, however, because they can fully utilize memory, they achieve 2.5x higher throughput
» **Compared to ADIOS2 we obtain 2.5x greater throughput under the same I/O thread constraints** (8 threads and 64 MB buffers)

FUTURE WORKS

» In modern HPC systems compute nodes outnumber OSTs (on frontier this number is ~7:1), thus, N-to-M is not an ideal aggregation pattern for larger applications
» In the future, we experiment with our aggregation strategy when M << # of compute nodes utilized by application, which will introduce greater synchronization costs and communication overhead
» Evaluate the effects our and other aggregation strategies has on different resource-bound (e.g. memory, compute, or communication) concurrently running workloads

REFERENCES

[1] VELOC – https://veloc.readthedocs.io/en/latest/
[2] GenericIO – https://git.cels.anl.gov/hacc/genericio
[3] ADIOS2 – Adios2.readthedocs.io/
[4] OpenMP Benchmark – https://github.com/FTHPC/HPC_IO_PROFI LING
[5] https://arxiv.org/pdf/2112.02289.pdf

Get in contact:

Argonne and Oak Ridge National Labs are acknowledged for generous allotment of compute time on the Theta and Frontier Supercomputers. This material is based upon work supported by the National Science Foundation under Grant No. SHF-1910197 and SHF-1943114. The material was supported by U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11.357