Dynamically Adaptable I/O Semantics for High Performance Computing

Michael Kuhn

Scientific Computing
Department of Informatics
University of Hamburg

2015-07-14

- More complex applications often produce more data
- Parallel distributed file systems with sizes of up to 60 PB and throughputs up to several TB/s
- One or more I/O interfaces offer access to data
  - Standardized access interfaces provide portability (POSIX)
  - Proprietary interfaces might offer improved performance

- High-level I/O interfaces are provided by I/O libraries
  - Offer additional features usually not found in file systems
  - Popular interfaces include MPI-IO, HDF and NetCDF
- Syntax defines operations, semantics defines behavior
- No knowledge about the applications' I/O requirements
  - Optimizations are often based on heuristic assumptions
  - Semantical information can provide needed knowledge

- POSIX features very strict consistency requirements
  - Changes have to be visible to other clients immediately
  - I/O is intended to be atomic
  - Easy to support in local file systems but effectively prohibits client-side caching in parallel distributed file systems
- MPI-IO's consistency requirements are less strict
  - Changes are immediately visible only to the process itself
  - Requires sync-barrier-sync construct to handle concurrency
  - Correctly handles non-overlapping or non-concurrent writes

- I/O semantics can only be changed in a limited fashion
  - `strictatime`, `relatime` and `noatime` change the file system's behavior regarding the last access timestamp
  - `posix_fadvise` allows announcing the access pattern
  - MPI-IO's atomic mode for stricter consistency semantics
- Provided facilities are often restricted
  - Usually only possible at file open or mount time
  - Mount options restricted to administrators
  - Often apply to the whole file

- JULEA features dynamically adaptable semantics
  - Applications developers can specify the I/O requirements at runtime on a per-operation basis
  - File system adapts itself according to applications' demands

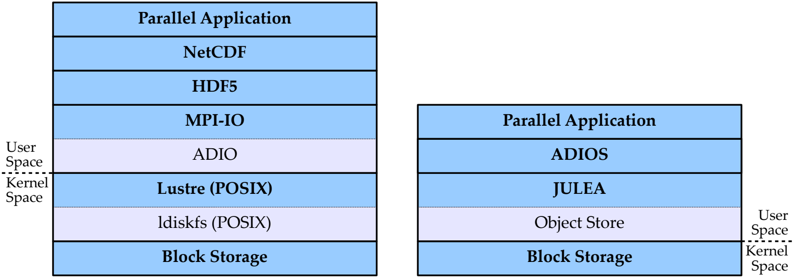


Figure: HPC and JULEA I/O stacks

- HPC: complex, loss of information, data transformations
- JULEA: easier to analyze, concentration into a single layer

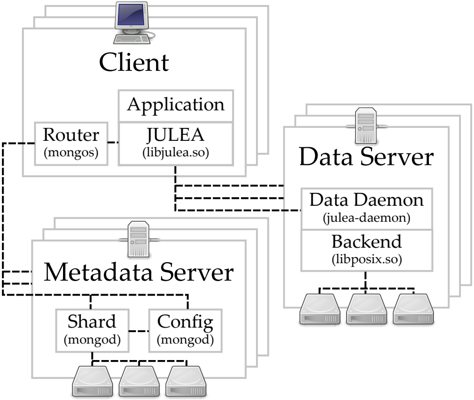


Figure: JULEA's architecture

- Designed a new I/O interface and file system prototype
- Architecture follows that of established file systems

- Semantics are dynamically adaptable according to the applications' I/O requirements
  - Developers can specify coarse-grained ("checkpoint") or fine-grained requirements ("strict consistency semantics")
  - File system can tune operations for specific applications
- All accesses to the file systems are performed via batches
  - Each batch can consist of multiple operations
  - Combine different kinds of operations within one batch

```
batch = new Batch(POSIX_SEMANTICS);

store = julea.create("test store", batch);
collection = store.create("test collection", batch);
item = collection.create("test item", batch);
item.write(..., batch);

batch.execute();
```

Listing 1: Executing multiple operations in one batch

- Namespace is split into stores, collections and items
- Provide a defined point for the operations' execution
  - Traditional approaches can only guess

- All important aspects of the semantics can be changed
  - Performance-related: atomicity, concurrency, consistency, ordering, persistency and safety
  - Further ideas: redundancy, security, transformation
- Templates for easy use and established semantics
  - Default: Concurrent non-overlapping operations
  - POSIX: Provided for backwards compatibility
  - Temporary (local): Allow transparent use of advanced technologies such as burst buffers

Atomicity

- Whether accesses should be executed atomically
  - Large operations usually involve several servers
  - Atomicity requires locking
- Levels of atomicity
  - None: Accesses are not executed atomically
  - Operation: Single operations are executed atomically
  - Batch: Complete batches are executed atomically
- Avoid unnecessary locking overhead
  - Many POSIX-compliant file systems perform unnecessary atomic write operations for shared access

Safety

- Specify how safely data and metadata should be handled
  - Provides guarantees about the state of the data and metadata after execution
- Levels of safety
  - None: No safety guarantees are made
  - Network: It is guaranteed that changes have been transferred to the servers as soon as the batch finishes
  - Storage: It is guaranteed that changes have been stored on the storage devices as soon as the batch finishes
- Allows adjusting the overhead of data safety measures
  - Eliminate one of two network messages by not requesting the server's acknowledgment for unimportant data

- Concurrency: Specify whether concurrent accesses will take place and how the access pattern will look like
- Consistency: Specify if and when clients will see modifications performed by other clients
- Ordering: Specify whether operations within a batch are allowed to be reordered
- Persistency: Specify if and when data and metadata must be written to persistent storage

1 Introduction and Motivation

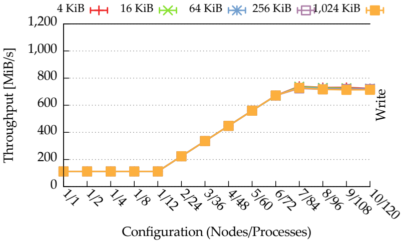
2 JULEA Approach

3 Evaluation Results

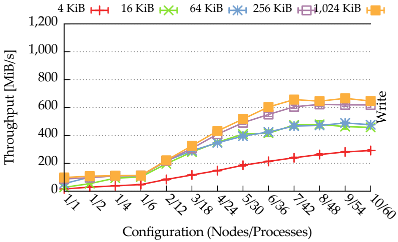
4 Conclusion and Outlook

- Evaluate potential of dynamically adaptable semantics
  - Using synthetic benchmarks and real applications
  - Large number of concurrently accessing clients
- Clients first write data and then read it back again
  - Write and read phases are completely separated
  - Individual and shared files, non-overlapping accesses
- Represents a very simple and common I/O pattern

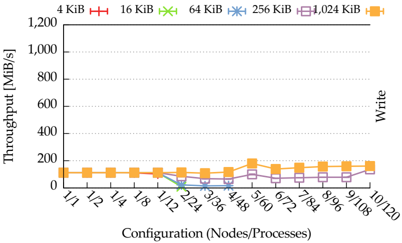
Data Performance



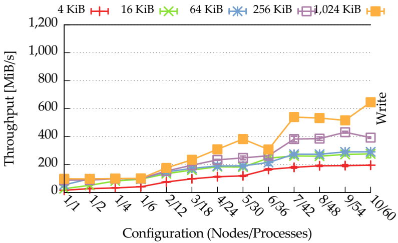
(a) Lustre: individual files, POSIX



(b) JULEA: individual items

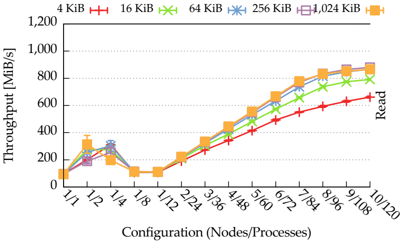


(c) Lustre: shared file, POSIX

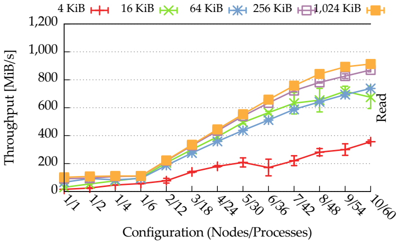


(d) JULEA: shared item

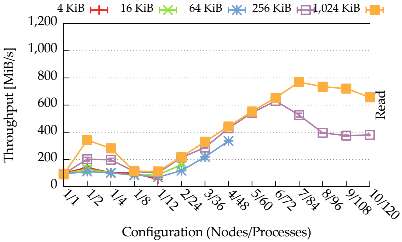
Data Performance



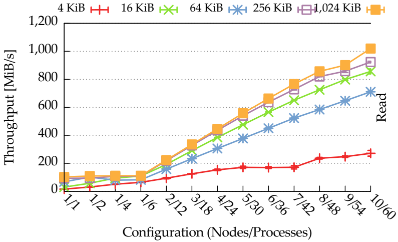
(e) Lustre: individual files, POSIX



(f) JULEA: individual items

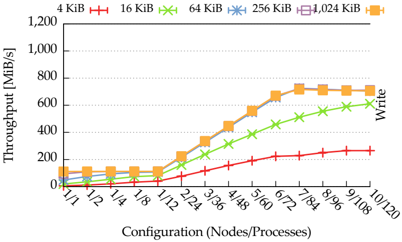


(g) Lustre: shared file, POSIX

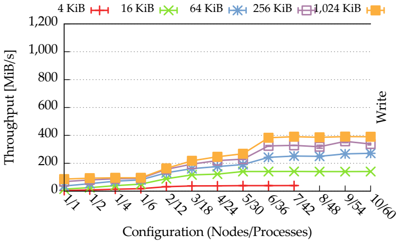


(h) JULEA: shared item

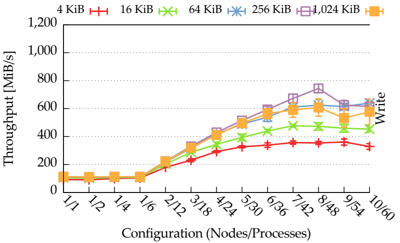
Data Performance



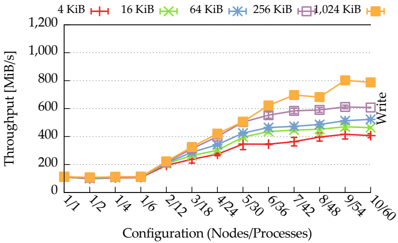
(i) Lustre: ind. files, MPI-IO (atomic)



(j) JULEA: individual items (atomic)

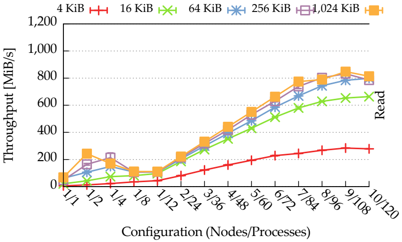


(k) JULEA: individual items (batch)

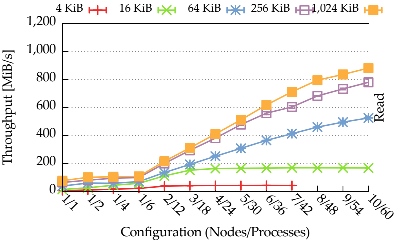


(l) JULEA: individual items (unsafe)

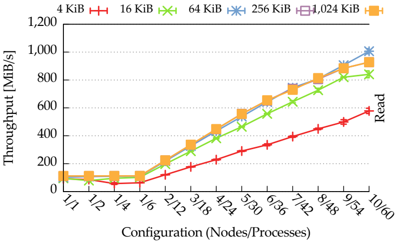
Data Performance



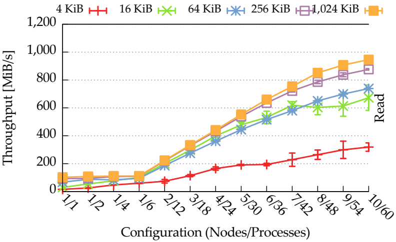
(m) Lustre: ind. files, MPI-IO (atomic)



(n) JULEA: individual items (atomic)



(o) JULEA: individual items (batch)



(p) JULEA: individual items (unsafe)

partdiff

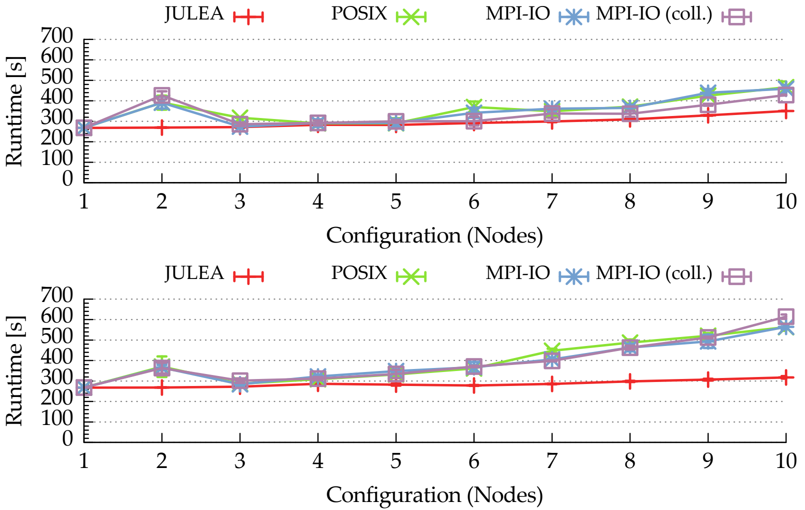


Figure: partdiff checkpointing using one and six processes per node

- Lustre suffers from problems due to POSIX
  - Performance is abysmal for shared files
  - Even with simple access patterns and few clients
- JULEA's performance is limited by underlying file system
  - Batches improve throughput for small block sizes
  - Safety semantics reduce network overhead
  - Atomic operations can be employed only when necessary
- Metadata results are also promising

- POSIX is portable but inflexible
  - No way to relax semantics
  - Effectively forces POSIX semantics upon other layers
- Static approaches are only suitable for a subset of use cases
  - Other file systems are also limited to their semantics
- JULEA offers solutions for the prevailing problems
  - Supports dynamically adaptable I/O semantics
  - Adapt according to the application requirements

- Detached activities to improve I/O interfaces
  - Focused on high-level I/O libraries
- JULEA presents a first uniform approach
  - Allows semantical information to be used across the complete I/O stack

- ADIOS's design is close to JULEA
  - Increase application coverage using a JULEA backend
- Provide dynamically adaptable semantics for established I/O interfaces and parallel distributed file systems
  - Interfaces have to be standardized and supported
  - Agree on semantics suited for modern HPC applications
  - Common set of configurable parameters

ISC 2015 Student Cluster Competition

Visit us at booth 418 and vote for us! ⌣̈

Twitter: @UHH_ISC_SCC