Optimizations

Parallel Storage Systems

2023-06-19

Jun.-Prof. Dr. Michael Kuhn

michael.kuhn@ovgu.de

Parallel Computing and I/O
Institute for Intelligent Cooperating Systems
Faculty of Computer Science
Otto von Guericke University Magdeburg
https://parcio.ovgu.de

Outline

- What is the main feature of SIONlib?
    1. Self-describing data format
    2. Optimized mapping and alignment
    3. Convenient I/O interface

Libraries

- What is chunking in HDF5 used for?
    1. Aligning accesses to file system stripes
    2. Allow features such as compression
    3. Enable multiple unlimited dimensions

Libraries

- Why is alignment important for performance?
  1. Prevent unnecessary communication with servers
  2. Prevent access and locking conflicts
  3. Prevent read-modify-write operations

Outline

Introduction

- Parallel I/O is much more complex than serial I/O
  - Parallel distributed file systems introduce additional complexity
  - Access is often done via layered libraries
  - Communicating via the network causes additional latency
- Complexity often has an impact on performance
  - Parallel distributed file systems are necessary for high performance
  - Libraries are necessary for convenient use by applications
    - MPI-IO, HDF, NetCDF etc.
- Complex interactions and optimizations on all layers

Introduction…

- There are several ways to improve performance
  - Some are controlled by the storage system, some by the user
  - Hybrid approaches require information from the user
- Advantages and disadvantages
  - System optimizations are independent of user knowledge
    - No additional complexity for users
    - Missing information also limits achievable performance
  - Additional information is often necessary for significant improvements
    - For example, stripe size in parallel distributed file systems

Outline

Caching

- Caching forms the basis for other optimizations
  - For example, aggregation and scheduling require caching of some form
- Server-side caching is relatively unproblematic
  - Cache exists at a central location, no consistency problems
  - Data can be lost when the server crashes
- Client-side caches are more problematic but also more promising
  - Data is first collected in RAM and then sent to the servers
    - Allows merging multiple network messages into one
  - Potentially allows reducing the amount of data to send
    - Data might be overwritten and only the final state has to be sent
- Client-side caching is often prevented by the environment
  - POSIX specifies that changes have to be visible globally

Caching...

- Read operations should be satisfied from the cache
  - Especially interesting when combined with read ahead
  - Allows hiding latency introduced by the network and storage devices
- Write operations can be handled by the local cache
  1. Data is first written to the cache and later flushed to device ("write-behind")
     - Can be done for access patterns without conflicts
     - Example: Non-overlapping write-only access patterns
  2. Data is written to the cache and the device at the same time ("write-through")
- Caching might also require multi-threading
  - One thread is often not enough to achieve maximum performance

Quiz

- Which caching mode would you use when data safety is important?
  1. Write-behind
  2. Write-through
  3. No caching

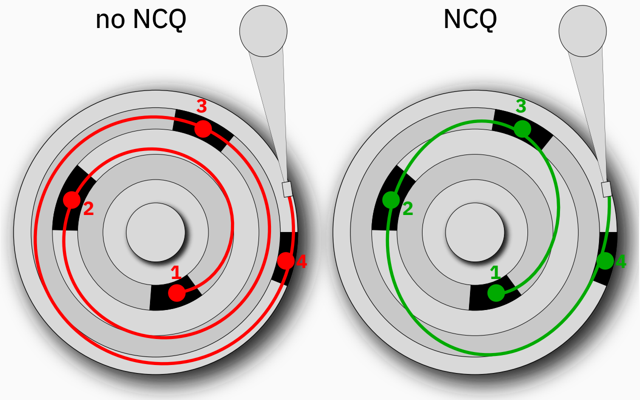
Caching…

- Caching increases the chances for conflicts
  - Concurrent access by multiple clients can overlap
  - Outdated data leads to coherence and consistency problems
- Still useful for a number of scenarios
  - Server-side caching almost always makes sense
  - Whenever no or only a few conflicts can occur
    - Home directories are only accessed by the owning user
    - Process-local files are only accessed by the owning process
- We will take a look at burst buffers later
  - Additional cache level to accelerate the file system

Scheduling

- Scheduling allows reordering I/O operations to improve performance
  - Requires caching to work in a reasonable way
  - Often performed as a preliminary stage for aggregation
- Reordering I/O requests can help devices
  - HDDs have different access latencies depending on the head position
    - Seek time (4–15 ms) and rotational latency (2–7 ms) are relevant
  - Scheduling can also make sense for SSDs
    - For instance, allowing parallel access to multiple flash cells
  - Seeking is an expensive operation for many storage devices
- Linux supports several low-level I/O schedulers
  - Among others, `cfq`, `deadline` and noop

Scheduling... [helix84, 2007]



- Native Command Queueing (NCQ) is a popular example for scheduling
  - Changing the order of operations allows improving operation throughput

Aggregation

- Aggregation merges multiple I/O operations to improve performance
  - Can also form the basis for more advanced optimizations
  - Requires caching to able to access operations to merge
- Individual operations cannot be optimized meaningfully
  - “Write 100 bytes at offset 2342”
- Additional context enables further optimizations
  - “Write 100 bytes each at offets 2342, 2442 and 2542”
  - Operation order can be problematic from a performance point of view

Aggregation...

- Aggregation is especially useful for small operations
  - Large operations are usually faster
    - Reduces seek times and read-modify-write overhead
  - Can be combined with reordering done by scheduling
- Merging can provide benefits by its own
  - Fewer I/O operations correspond to fewer system calls
    - Mode/context switches have constant overhead
    - Aggregation must be performed in user space
- Aggregation is widely used, like scheduling
  - Almost all of Linux's I/O schedulers aggregate operations
    - Even noop performs aggregation

Replication

- Replication stores data redundantly at several locations
  - Also allows storing data closer to the user (for example, for clouds/grids)
- Can be used to implement load balancing
  - Large numbers of accesses can be distributed across multiple replicas
- Problematic when data has to be modified
  - Data must be updated at all locations and could lead to inconsistencies
  - Degrades write performance if users have to wait for updates to finish
- Most useful if data is accessed mostly for reading
  - If files are read-only, there are no disadvantages (except for storage overhead)
  - Most often used in big data and cloud contexts, increasingly also in HPC

Metadata

- Metadata operations are critical for overall performance
  - Data can only be accessed when metadata has been found
- Example: POSIX time stamp for last access (`atime`)
  - Executing `file *` in a directory with millions of files
    - Updates the time stamp for all files
    - Moreover, first few bytes of each file have to be read
- Problem can be worked around
  - `no[dir]atime`, `relatime`, `strictatime` und `lazytime`
  - Alternatively, specify `O_NOATIME` when using open

"It's also perhaps the most stupid Unix design idea of all times. […] 'For every file that is read from the disk, let's do a ... write to the disk! And, for every file that is already cached and which we read from the cache ... do a write to the disk!'"

– Ingo Molnar

Metadata...

- Metadata operations often depend on each other
  - Makes concurrent execution problematic
  - Examples: Path resolution, creating a file etc.
- There is a multitude of approaches to improve metadata performance
  - Aggregating metadata operations
    - Compound operations
  - Reducing the amount of metadata operations
    - Relaxed semantics
  - Intelligently distribute metadata load
    - Dynamic metadata management

Outline

Example: Cooperative Caching [Liao et al., 2005]

- Global cache that is available on all nodes
  - Potentially huge capacity of several terabytes or even petabytes
  - Improves latency and throughput when accessing files
- Data is read from the main memory of a specific client
  - Typically faster than reading data from the file system
  - In the best case, data is available in the local main memory
- Data is also written to main memory
  - Data is then flushed to the file system in the background
  - Safety measures to ensure that data cannot be lost

Example: Cooperative Caching… [Liao et al., 2005]

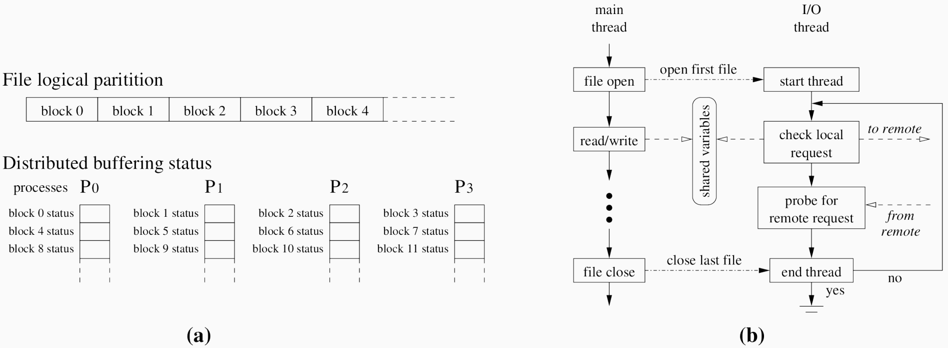


Fig. 1. (a) The buffering status is statically distributed among processes in a round-robin fashion. (b) Design of the I/O thread and its interactions with the main thread and remote requests.

- Data is lost if a client node crashes
    - Can be prevented using redundancy or frequently writing data back to storage

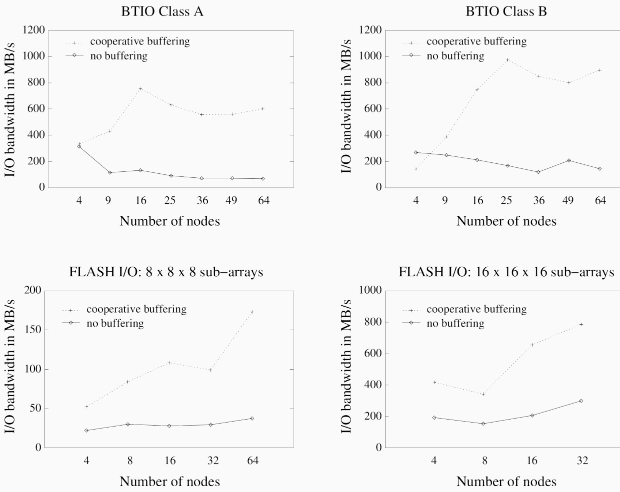


Fig. 2. I/O bandwidth results for BTIO and FLASH I/O benchmarks.

Example: Cooperative Caching… [Liao et al., 2005]

- Moves load from the file system to the application
  - Additional cache level for data
- Advantages
  - File system is eliminated as the bottleneck
  - Mapping is static and does not require further coordination
  - Communication throughput is typically higher than I/O throughput
- Disadvantages
  - Main memory capacity is decreased due to caching
  - Data throughput is limited by responsible client
  - Can have negative influence on application performance

Example: ZFS Scheduling [Chad Mynhier, 2006]

- ZFS assigns a priority and a deadline to each I/O operation
  - A higher priority implies a shorter deadline
- Read operations generally receive a higher priority than write operations
  - Reads are more important for the (perceived) latency
  - Write operations can be buffered in a cache
  - Read operations usually have to access the storage device
    - Large data sets cannot be cached in their entirety
- Linux's `deadline` scheduler works similarly

Example: ZFS Scheduling… [Chad Mynhier, 2006]

- Reads are faster on ZFS with load
  - No difference without load
  - Important for system's interactivity
- Write operations take longer
  - Writes can be cached more easily

File System	Without Load	With Load
ZFS	0:09	0:10
ext3	0:09	5:27
reiserfs	0:09	3:50

512 MB file with moderate load

File System	Without Load	With Load
ZFS	0:32	0:36
UFS	0:50	5:50
ext3	0:36	54:21
reiserfs	0:33	69:45

2 GB file with high load

Example: hashFS [Lensing et al., 2013]

- Reminder: Path resolution is sequential and causes significant overhead
  - Many small metadata accesses for all path components
- Hashing allows direct access to metadata and data
  - Use full path to determine hash
  - Reduces amount of accesses to one read operation per file
  - Permissions of parent components have to be taken into account
- Problem: Renaming a parent changes hashes of all children

Quiz

- How would you handle renames?

Quiz

- How would you handle renames?
  1. Hash are recomputed immediately
     - Depending on number of files, high overhead

- How would you handle renames?
  1. Hash are recomputed immediately
     - Depending on number of files, high overhead
  2. Renames are stored in a mapping table
     - Table accesses cause additional overhead

Example: Dynamic Metadata Mgmt [Weil et al., 2004]

- Metadata is typically distributed statically based on a hash
  - Dynamic metadata management uses responsibility for subtrees
- Metadata management is distributed dynamically based on load
  - Metadata servers are responsible for one or more file system subtrees
  - Responsibilities can be changed at runtime
- Clients do not have a-priori knowledge about responsible servers
  - Clients ask a random server for metadata
  - Servers forward requests if necessary

Example: Dynamic Metadata Mgmt… [Weil et al., 2004]

- Trees are split up and distributed at runtime
  - Allows adapting metadata management to current load situation
- Metadata can also be replicated when necessary
  - Replication is triggered when metadata is accessed often
  - Replicas are stored on different servers
- Advantages
  - Can be used to distribute load more evenly
- Disadvantages
  - Requires more communication and adds communication between servers
  - Increases latency for first file access

- Static distribution can cause single server to become overwhelmed
  - For instance, many clients creating files in a shared directory
- Static distribution stays unbalanced
  - Clients would have to adapt
- Dynamic distribution adapts to load

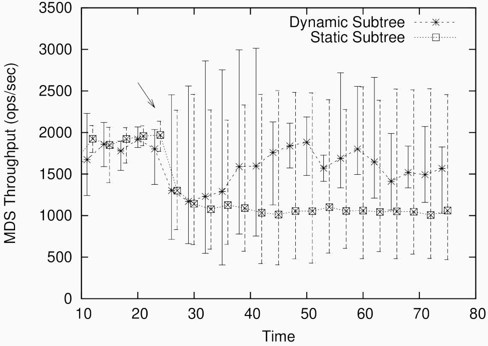


Figure 5: The range and average throughput of MDSs is shown under a dynamic workload. When clients migrate and create files in new portions of the hierarchy, a static subtree distribution remains unbalanced, while the dynamic partition re-balances load and achieves higher average performance by migrating newly popular portions of the hierarchy to non-busy nodes.

Example: Dynamic Metadata Mgmt. . . [Weil et al., 2004]

- Responsibility is moved due to load
  - Leads to more forwarded requests
- Static distribution has less overhead
  - Performance is still lower

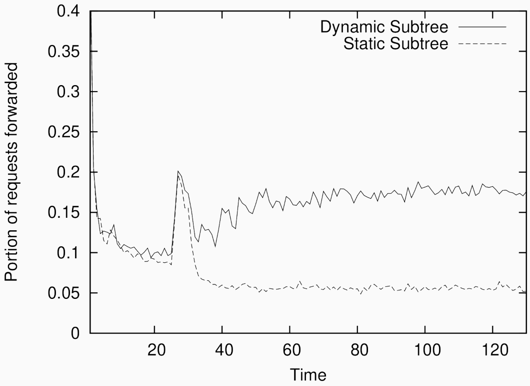


Figure 6: Forwarded requests for static and dynamic partitioning under a dynamic workload. The spike represents a shift in workload, while the difference after that point highlights overhead due to client ignorance of metadata movement from dynamic load balancing.

- Popular file can overwhelm server
  - All requests forwarded to one server, which responds slowly
- Replication distributes load
  - Requests forwarded to all of them, higher performance

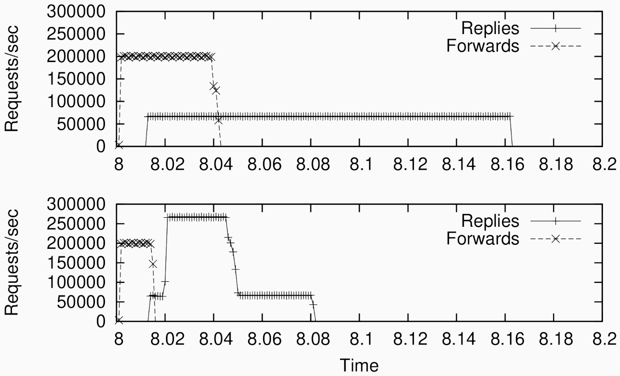
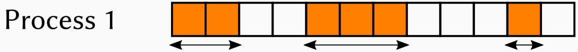


Figure 7: No traffic control (top): nodes forward all requests to the authoritative MDS who slowly responds to them in sequence. Traffic control (bottom): the authoritative node quickly replicates the popular item and all nodes respond to requests.

Outline

Non-Contiguous I/O

- Traditionally, only contiguous regions can be read/written
  - Native support for non-contiguous I/O in MPI-IO
  - POSIX does not offer native support for this
- Can be imagined as I/O operations with holes
  - Similar to sparse files, which also contain holes
  - For instance, users can read/write a matrix diagonal with one operation
- Offers the foundation for a number of high-level optimizations
  - In combination with collective I/O, further optimizations are possible

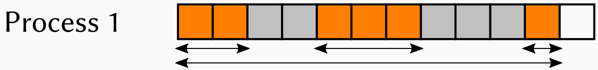


Non-Contiguous I/O…

- Individual contiguous parts still have to be accessed separately
  - Storage devices only offer block-based access
  - Many small accesses can have a negative impact on performance
    - Goal: Aggregate accesses so they become contiguous
- Two main approaches in MPI-IO
  1. Read or write contiguous blocks
     - Might potentially contain more data than required
     - This optimization is called data sieving
  2. Combine multiple non-contiguous I/O operations
     - The aggregation might result in a large contiguous access
     - This is especially interesting in combination with collective I/O

Data Sieving

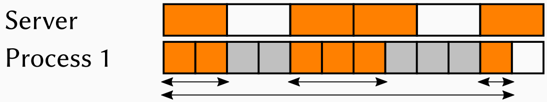
- Data sieving is an optimization for non-contiguous I/O
  - Implemented and used by default in ROMIO
- Turn non-contiguous accesses into contiguous ones for the storage devices
  - Often faster than performing many small accesses and skipping the holes
    - This also applies to non-rotational storage devices such as SSDs
- Unnecessary data is discarded
  - Not always worth it, therefore necessary to estimate costs
  - Estimation especially complex in parallel distributed file systems



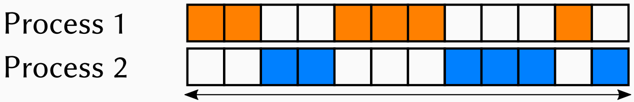
Quiz

- Which additional problems are present in parallel distributed file systems?
  1. Clients could communicate with more servers than necessary
  2. File systems do not support non-contiguous I/O, which is necessary
  3. Data sieving requires read-modify-write in parallel distributed file systems

- Data sieving can lead to access conflicts
  - Reading is relatively unproblematic
- Writing can cause more problems
  - Old data has to be read first to fill the holes
  - Read-modify-write causes overhead
- Both reading and writing can negatively affect performance
  - Logically contiguous ≠ physically contiguous
    - File system allocation, sector remapping, distribution etc.
  - Might lead to more communication with servers than necessary



- Clients perform I/O operations in a coordinated fashion
  - Individual accesses are uncoordinated and therefore random
- Operations can be scheduled and aggregated more effectively
  - Non-contiguous accesses by multiple clients can be merged



- Non-collective operations could lead to accesses of process 2 executing first
  - Looks like random accesses to the file system
  - Causes non-contiguous accesses not to be aggregated

Two Phase [Thakur et al., 1999]

- Two Phase is an optimization for collective I/O
  - An implementation of the general idea is included in ROMIO
- Idea: Clients coordinate independently of the file system
  - Clients are responsible for contiguous blocks
  - Blocks are disjoint and contain all requested data
- Leads to a 1-to-1 communication in the best case
  - Usually, one client has to contact multiple servers
  - Helps reduce the network and storage device overhead
- Additional communication overhead is introduced and can be detrimental
  - Worst case: All data is being sent a second time

Two Phase… [Thakur et al., 1999]

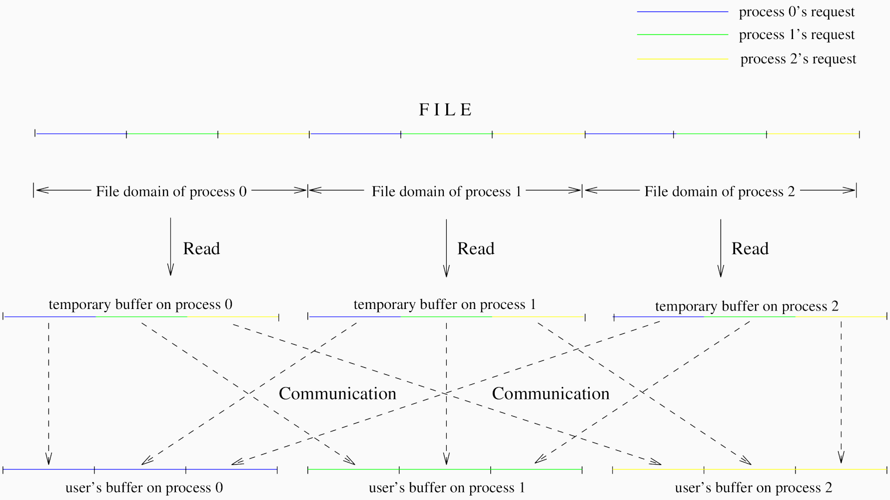


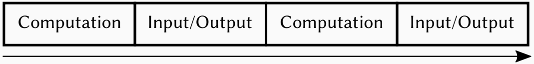
Figure 3. A simple example illustrating how ROMIO performs a collective read

Asynchronous I/O

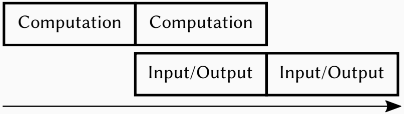
- Asynchronous I/O allows overlapping I/O with computation, communication etc.
  - Only works if there is enough concurrent work to do
  - Buffer cannot be accessed while the asynchronous is pending
- Removes implicit synchronization from parallel applications
  - Requires special asynchronous I/O functions
    - For instance, `MPI_File_iwrite` and `aio_write`
  - Progress can be checked with separate functions
    - For example, `MPI_test` and `aio_return`
- Has the potential to introduce race conditions
  - Data can only be changed when I/O is finished
  - Buffering the I/O can help work around the problem

Asynchronous I/O...

- Use case: Results are written out after computation has finished
    - Traditionally, operation blocks until data has been written



- Asynchronous I/O allows progressing I/O concurrently
    - Only possible if computation does not change the data buffer



- Limitation: The maximum speedup of this approach is 2

Hints

- Users should provide as much information as possible for optimizations
  - Allows the file system and libraries to optimize accesses
- Hints are typically optional
  - That is, users do not have to specify them for correct operation
  - However, the system is also free to ignore them
- Hints can be used to tune a wide range of optimizations
  - Information about access modes: read-only, read-mostly, append-only, non-contiguous access, unique, sequential etc.
  - Adapting buffer sizes
  - Modifying the number of processes involved in I/O (such as Two Phase)

Relaxed Semantics

- Adapting the semantics to application requirements
  - Data: Do not make modifications visible immediately
  - Metadata: Do not store all metadata (for instance, timestamps)
- Users need a way to be able to specify requirements
  - Users typically know best how their applications behave
  - File systems and libraries usually do not have support for this
- There is typically only support for one static semantics
  - Static semantics is suitable for some use cases but never for all

Relaxed Semantics…

- Research topic: Give users ability to control semantics
  - For example, two modes for safety and performance
- Use different locking mechanisms depending on the use case
  - That is, no or very limited locking in performance mode
- Data safety can also be tuned for performance
  - That is, no redundancy and synchronization in performance mode
- Coherence and consistency requirements also differ
  - That is, allow extensive caching in performance mode
- Performance mode could be used for process-local temporary files

Outline

Best Practices

- Access data sequentially if possible (not serially!)
  - More efficient than small accesses here and there
  - Still relevant even with non-rotational storage devices
- Avoid seek operations as much as possible
  - Head movements in an HDD are very slow
  - Communication with different servers causes overhead
- Prevent many small accesses whenever possible
  - Few large accesses, like with message passing
  - I/O suffers from network and storage device latencies

Best Practices…

- Check behavior of I/O functions that are used
  - For instance, which functions are synchronous and which are collective
- Access patterns are an important aspect for overall performance
  - File systems and libraries can compensate in some cases
  - Inefficient applications will still not perform optimally

Summary

- There is a wide range of different I/O optimizations
  - Optimizations are typically performed on all layers of the stack
  - Different workarounds and optimizations can conflict
  - Basic optimizations like caching, scheduling etc. provide the basis
- Achievable performance heavily depends on the application and user
  - Provide as much information as possible, including access patterns, modes etc.
  - I/O interfaces often provide facilities to do so and can optimize more effectively
- User should also perform optimizations manually if possible
  - Improve access patterns, make use of asynchronous I/O etc.

References

[Chad Mynhier, 2006] Chad Mynhier (2006). **ZFS I/O reordering benchmark.** http://cmynhier.blogspot.com/2006/05/zfs-io-reordering-benchmark.html.

[helix84, 2007] helix84 (2007). **Native Command Queuing.** https://en.wikipedia.org/wiki/File:NCQ.svg. License: CC BY-SA 3.0.

[Lensing et al., 2013] Lensing, P. H., Cortes, T., and Brinkmann, A. (2013). **Direct lookup and hash-based metadata placement for local file systems.** In Kat, R. I., Baker, M., and Toledo, S., editors, *6th Annual International Systems and Storage Conference, SYSTOR '13, Haifa, Israel - June 30 - July 02, 2013*, pages 5:1–5:11. ACM.

[Liao et al., 2005] Liao, W., Coloma, K., Choudhary, A. N., and Ward, L. (2005). **Cooperative Write-Behind Data Buffering for MPI I/O.** In Martino, B. D., Kranzlmüller, D., and Dongarra, J. J., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy, September 18-21, 2005, Proceedings*, volume 3666 of *Lecture Notes in Computer Science*, pages 102–109. Springer.

[Thakur et al., 1999] Thakur, R., Gropp, W., and Lusk, E. (1999). **Data Sieving and Collective I/O in ROMIO.** In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, Washington, DC, USA. IEEE Computer Society.

[Weil et al., 2004] Weil, S. A., Pollack, K. T., Brandt, S. A., and Miller, E. L. (2004). **Dynamic Metadata Management for Petabyte-Scale File Systems.** In *Proceedings of the ACM/IEEE SC2004 Conference on High Performance Networking and Computing, 6-12 November 2004, Pittsburgh, PA, USA, CD-Rom*, page 4. IEEE Computer Society.