Libraries

Parallel Storage Systems

2023-06-12

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 happens if a file opened with `MPI_MODE_SEQUENTIAL` is accessed non-sequentially?
  1. MPI will fall back to regular access without optimization
  2. MPI will print a warning
  3. Undefined behavior

- What is the point of collective I/O operations?
  1. Synchronize all processes in a communicator
  2. Provide additional information for optimizations
  3. Order operations by rank

- Which benefits do non-contiguous I/O operations have?
  1. Many small operations are faster to execute
  2. Multiple operations can always be merged into a contiguous one
  3. Fewer operations cause less overhead

- Which guarantees does MPI-IO's semantics provide?
  1. Operations can be overlapping and concurrent
  2. Operations can be non-overlapping but concurrent
  3. Operations can be overlapping but non-concurrent
  4. Operations have to be non-overlapping and non-concurrent

Outline

Motivation

- POSIX and MPI-IO can both be used for parallel I/O
  - Both interfaces are not very comfortable to use
- Low-level interfaces are problematic for scientific applications
  - Exchangeability of data is important
    - POSIX and MPI-IO only offer limited portability
  - Implementing parallel I/O is complicated
    - Byte- or element-based access is cumbersome
  - Achieving high performance is difficult
    - In-depth knowledge about storage system is necessary

Motivation...

- I/O libraries provide additional functionality
  - Self-describing data allows exchanging data more easily
    - Data can be read and interpreted without prior knowledge
  - Internal structuring increases flexibility
    - Such data formats allow storing multiple variables
  - Abstract I/O definitions make it easier for developers
    - I/O calls do not have to be added to the code manually
- Performance problems can be avoided using optimized libraries
  - Strict semantics and missing optimizations can degrade performance

Motivation...

- Benefits
  - Data is more portable
  - Usability is increased
  - Development is made easier
- Drawbacks
  - Introduces additional software layers
  - Interaction between layers becomes more complex

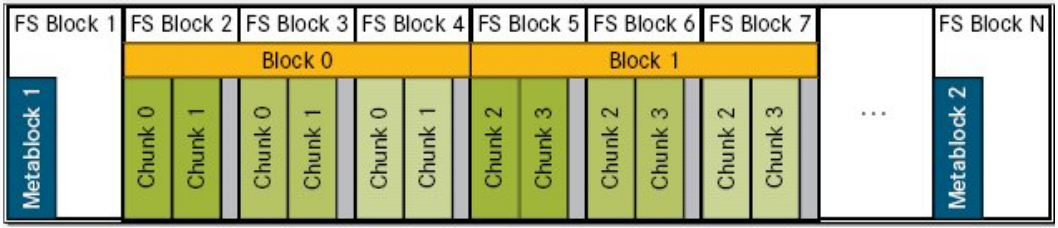
Approaches

- Some libraries are focused on improving performance
  - SIONlib
- There are multiple self-describing data formats
  - NetCDF (Network Common Data Form)
  - HDF (Hierarchical Data Format)
- Abstract I/O definitions are rather special
  - ADIOS (Adaptable IO System)

Outline

- SIONlib offers efficient access to process-local files
- Accesses are mapped to one or a few physical files
  - Number of files depends on what promises better performance
  - Files are aligned to file system's blocks/stripes
- Access should be as backwards compatible as possible
  - Wrappers for `fread` and `fwrite`
  - Opening and closing files requires special functions

- What is the goal of aligning accesses to file system stripes?
  1. Bypassing read-modify-write overhead
  2. Reducing or eliminating locking overhead
  3. Better support for collective I/O operations



[SIONlib, 2021]

- Data of a process is kept within a file system block
  - File system blocks are separated into chunks
  - Multiple file system blocks are grouped into a block
- Metablock 1: Static metadata (written on open)
- Metablock 2: Dynamic metadata (written on close)

Example

- numfiles: Number of physical files (-1 for automatic detection)
- chunksize: Maximum size of a write
- fsblocksize: Size of a file system block (-1 for automatic detection)

```
1  int fd;
2  FILE* fp;
3
4  fd = sion_paropen_mpi(..., &numfiles,
5                        ..., &chunksize,
6                        &fsblocksize,
7                        ..., &fp, ...);
8
9  for (...) {
10     fwrite(..., fp);
11 }
12
13 sion_parclose_mpi(fd);
```

- SIONlib offers multiple access modes
  - Collective open using `sion_paropen_mpi`
    - Works similar to `MPI_File_open`
  - Individual open using `sion_open_rank`
    - Allows accessing the chunks of a specific process
  - Serial access using `sion_open` and `sion_close`
- Mapping and alignment are handled intelligently
  - Exclusive use of individual file system blocks by processes
    - Performance is improved at the cost of potentially unused space due to padding
  - Mapping to internal file layout is handled transparently
    - Application developers can focus on their problem at hand

- SIONlib is focused on performance and works around deficiencies in file systems
- Too many files can have negative impacts on performance
  - Creating and opening many files requires significant metadata performance
  - File systems are typically not designed for large file and directory counts
- Too few files can also degrade performance due to locking
  - Necessary to minimize locking using appropriate file layouts and access patterns

Outline

- Developed by the Unidata Program Center
  - University Corporation for Atmospheric Research
- Project started in 1989
  - Based on NASA's Common Data Format
- Mainly used in scientific applications
  - Most commonly used in climate science, meteorology and oceanography
- Consists of libraries and data formats

- There are four data formats
  1. Classical format (CDF-1)
  2. Classical format with 64 bit offsets (CDF-2)
  3. Classical format with full 64 bit support (CDF-5)
  4. NetCDF-4 format
- NetCDF's data formats are open standards
  - CDF-1 and CDF-2 are international standards of the Open Geospatial Consortium
- CDF-1, CDF-2 and CDF-5 are independent formats
  - NetCDF-4 uses the underlying HDF5 format

- Initially no support for parallel I/O when using CDF-1 and CDF-2
  - Lead to the development of Parallel-NetCDF with an incompatible interface
- Since NetCDF-4 there is support for parallel I/O via HDF5
  - Parallel I/O was limited to the NetCDF-4 format
- Recent versions support parallel I/O for all formats
  - NetCDF-4 is supported via HDF5
  - CDF-1, CDF-2 and CDF-5 are supported via Parallel-NetCDF

- NetCDF has bindings for a wide range of programming languages
  - C, Fortran, C++, Java, R, Perl, Python, Ruby etc.
- Data formats are independent of the used architecture
  - Automatic conversion according to endianness etc.
- NetCDF supports groups and variables
  - Groups contain variables, variables contain data
  - Flat hierarchy since groups cannot contain groups
- Additional attributes can be attached to variables

- NetCDF supports multi-dimensional arrays
  - `char`, `byte`, `short`, `int`, `float` and `double`
    - Since CDF-5: ubyte, ushort, uint, int64 and uint64
  - NetCDF-4: `ubyte`, `ushort`, `uint`, `int64`, `uint64` and `string`
- Dimensions can be arbitrarily large
  - Only one unlimited dimension in CDF-1, CDF-2 and CDF-5
    - Example: Matrix can only grow in the time dimension
  - Multiple unlimited dimensions in NetCDF-4
    - Requires a more complex data formats (provided by HDF5)

- NetCDF comes with additional features
  - Data can be compressed transparently
- Tools expand the usability of NetCDF further
  - `ncdump` comes with NetCDF and allows printing file contents
  - NetCDF Operators (NCO) can perform a variety of operations on data
- NetCDF is used as the foundation of further standards
  - The Climate Data Interface supports NetCDF and other formats

Example

```
netcdf ... {
dimensions:
    time = UNLIMITED ; // (8760 currently)
variables:
    double time(time) ;
        string time:units = "days" ;
        string time:long_name = "Julian_date" ;

// global attributes:
        string :Conventions = "None" ;
        string :creation_date = "Wed Jul 16 12:52:44 CEST 2014" ;
}
```

- ncdump can be used to inspect the data

Example...

1. Create file with `nc_create("file.nc", ..., &ncid)`
   - Parallel access with `nc_create_par`
   - Backend can be selected with `NC_MPIIO` or `NC_NETCDF4`

Example...

1. Create file with `nc_create("file.nc", ..., &ncid)`
   - Parallel access with `nc_create_par`
   - Backend can be selected with `NC_MPIIO` or `NC_NETCDF4`
2. Define dimension with `nc_def_dim(ncid, "dim", ..., &dimid)`

Example...

1. Create file with `nc_create("file.nc", ..., &ncid)`
   - Parallel access with `nc_create_par`
   - Backend can be selected with `NC_MPIIO` or `NC_NETCDF4`
2. Define dimension with `nc_def_dim(ncid, "dim", ..., &dimid)`
3. Define group with `nc_def_grp(ncid, "group", &grpid)`

Example...

1. Create file with `nc_create("file.nc", ..., &ncid)`
   - Parallel access with `nc_create_par`
   - Backend can be selected with `NC_MPIIO` or `NC_NETCDF4`
2. Define dimension with `nc_def_dim(ncid, "dim", ..., &dimid)`
3. Define group with `nc_def_grp(ncid, "group", &grpid)`
4. Define variable with `nc_def_var(grpid, "data", ..., &varid)`

Example...

1. Create file with `nc_create("file.nc", ..., &ncid)`
   - Parallel access with `nc_create_par`
   - Backend can be selected with `NC_MPIIO` or `NC_NETCDF4`
2. Define dimension with `nc_def_dim(ncid, "dim", ..., &dimid)`
3. Define group with `nc_def_grp(ncid, "group", &grpid)`
4. Define variable with `nc_def_var(grpid, "data", ..., &varid)`
5. Write attribute with `nc_put_att_*(grpid, varid, "attr", ...)`

Example...

1. Create file with `nc_create("file.nc", ..., &ncid)`
   - Parallel access with `nc_create_par`
   - Backend can be selected with `NC_MPIIO` or `NC_NETCDF4`
2. Define dimension with `nc_def_dim(ncid, "dim", ..., &dimid)`
3. Define group with `nc_def_grp(ncid, "group", &grpid)`
4. Define variable with `nc_def_var(grpid, "data", ..., &varid)`
5. Write attribute with `nc_put_att_*(grpid, varid, "attr", ...)`
6. Leave define mode with `nc_enddef(ncid)`
   - Performed implicitly with NetCDF-4 format
   - Compression, endianness, fill values etc. can only be set on first definition

Example...

1. Create file with `nc_create("file.nc", ..., &ncid)`
   - Parallel access with `nc_create_par`
   - Backend can be selected with `NC_MPIIO` or `NC_NETCDF4`
2. Define dimension with `nc_def_dim(ncid, "dim", ..., &dimid)`
3. Define group with `nc_def_grp(ncid, "group", &grpid)`
4. Define variable with `nc_def_var(grpid, "data", ..., &varid)`
5. Write attribute with `nc_put_att_*(grpid, varid, "attr", ...)`
6. Leave define mode with `nc_enddef(ncid)`
   - Performed implicitly with NetCDF-4 format
   - Compression, endianness, fill values etc. can only be set on first definition
7. Write variable with `nc_put_var_*(grpid, varid, ...)`

Example...

1. Create file with `nc_create("file.nc", ..., &ncid)`
   - Parallel access with `nc_create_par`
   - Backend can be selected with `NC_MPIIO` or `NC_NETCDF4`
2. Define dimension with `nc_def_dim(ncid, "dim", ..., &dimid)`
3. Define group with `nc_def_grp(ncid, "group", &grpid)`
4. Define variable with `nc_def_var(grpid, "data", ..., &varid)`
5. Write attribute with `nc_put_att_*(grpid, varid, "attr", ...)`
6. Leave define mode with `nc_enddef(ncid)`
   - Performed implicitly with NetCDF-4 format
   - Compression, endianness, fill values etc. can only be set on first definition
7. Write variable with `nc_put_var_*(grpid, varid, ...)`
8. Close file with `nc_close(ncid)`

- There are two major cases when reading data
  1. File structure is unknown (different application etc.)
     - Available groups and variables have to be determined
  2. File structure is known (same application, documentation)
     - Groups and variables can be accessed directly using their names

1. File is opened using `nc_open`
   - Parallel access with `nc_open_par`
2. Group IDs can be queried via `nc_inq_ncid`
3. Variable IDs can be queried using `nc_inq_varid`
4. Variables are read with `nc_get_var`
5. File is closed via `nc_close`

- Self-describing nature of NetCDF requires two modes
  - Define mode: Automatically active after creating a file
  - Data mode: Active after opening an existing file
- Define mode allows changing the file's structure
  - For example, adding dimensions, attributes and variables
  - Some settings can only be changed directly after definition
    - Including compression, endianness, error correction and fill values
- Data mode allows storing data
- NetCDF-4 changes the mode automatically and on demand
  - Can be done manually via `nc_redef` and `nc_enddef`

- Parallel-NetCDF is an alternative implementation for parallel I/O
  - Supports CDF-1, CDF-2 and CDF-5
- Developed by Northwestern University and Argonne National Laboratory
  - Developer overlap with MPI-IO and OrangeFS
- Interface itself is not compatible with NetCDF
  - NetCDF-4 can make use of Parallel-NetCDF

Outline

- HDF consists of data formats and libraries
  - Allows managing self-describing data, similar to NetCDF
- Current version is HDF5
  - HDF4 is still actively supported
- Earlier versions had problems with their interface
  - Application programming interface was complicated to use
  - Limitations like 32 bit offsets etc.

- HDF supports groups and data sets
  - Data sets store data as multi-dimensional arrays
  - Groups can be used to structure the namespace
  - Groups and data sets are similar to directories and files
- Groups can contain both data sets and further groups
  - Leads to a hierarchical namespace, similar to POSIX
- Attributes can be attached to data sets and groups
  - For instance: Physical units, minimum, maximum etc.

- Objects can be accessed using POSIX-like paths
  - Example: `/path/to/dataset`
  - Path can be used to describe data
- HDF files are self-describing
  - Can be opened without a-priori knowledge about the structure and content
  - Attributes allow interpreting the actual data

- Data sets are multi-dimensional arrays of base data types
  - Integer, float, character, bit field, opaque, enumeration, reference, array, variable-length and compound
- Data sets have certain properties
  - Size, precision, endianness etc.
- Arrays can have multiple unlimited dimensions

Example

```
HDF5 "..." {
GROUP "/" {
   ATTRIBUTE "creation_date" {
      DATATYPE  H5T_STRING {
         STRSIZE H5T_VARIABLE;
         STRPAD H5T_STR_NULLTERM;
         CSET H5T_CSET_ASCII;
         CTYPE H5T_C_S1;
      }
      DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
   }
}
}
```

- h5dump can be used to inspect the data (like ncdump)

- HDF stores data differently depending on the used programming language
  - Data is stored row-major (according to the C conventions)
  - Fortran data (column-major) is converted automatically

1	2	3
4	5	6
7	8	9

3×3 array

1	2	3	4	5	6	7	8	9

Row-major storage (C)

1	4	7	2	5	8	3	6	9

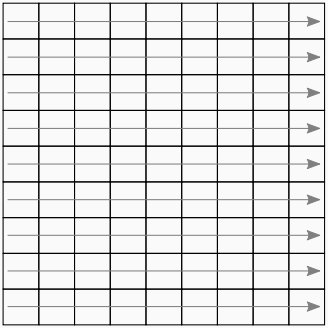
Column-major storage (Fortran)

- How would you read Fortran data into a C application?
  1. Use non-contiguous data types
  2. Read sequentially and transpose in memory
  3. Read individual elements in row-major order

- Chunking splits up data sets into smaller portions
  - Required for features such as compression and other filters
- Chunking allows extending data in all dimensions
  - Not easily possible when using contiguous storage
- Enables optimizations not possible when stored contiguously
  - Data can be aligned to stripes or allow efficient column-major access
- Chunking also causes some overhead
  - Chunk index has to be managed efficiently
  - Often results in lower performance than contiguous storage

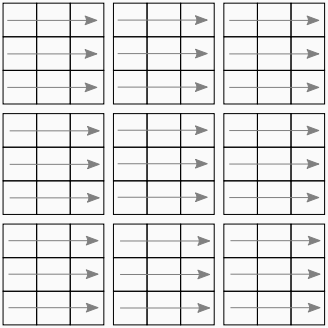
Chunking...

- Data sets are stored contiguously
  - Simple layout with little overhead
- Data set can be extended in one direction
  - Other directions can be mapped to a static size



Chunking...

- Data sets are split up into chunks
  - Can be extended in multiple directions
  - More complex management
- Chunks are read or written completely
  - HDF5 uses a chunk cache internally



- Which chunk size would you choose when compressing data?
  1. 64 KiB
  2. 1 MiB
  3. 64 MiB
  4. 1 GiB

- HDF supports multiple backends (Virtual File Layer)
  - Used to realize access using POSIX and MPI-IO (and more)
    - MPI-IO allows efficient parallel access to shared HDF files
- HDF contains several tools to work with data
  - `h5dump` can be used to print data
- Additional functionality similar to NetCDF
  - Transparent compression and user-defined filters
- HDF is under active development
  - Virtual Object Layer (VOL) allows alternative storage approaches
    - Higher level of abstraction in comparison to the VFL
  - Additional features for exascale and cloud

Outline

- ADIOS provides an abstract I/O interface
  - No byte- oder element-based access to data
  - ADIOS directly supports application data structures
- ADIOS has been designed for high performance
  - It is used in a wide range of scientific applications
  - Performance is increased using caching, merging of operations etc.

- ADIOS1 allows specifying an I/O configuration via an XML file
  - XML contains a description of relevant data structures
  - Code is automatically generated from this abstract description
- Allows developers to work on a high level of abstraction
  - No need for dealing with the middleware or the file system
  - Some changes to the I/O configuration do not require a recompilation

Example

```
<adios-config host-language="C">
  <adios-group name="checkpoint">
    <var name="rows" type="integer"/>
    <var name="columns" type="integer"/>
    <var name="matrix" type="double" dimensions="rows,columns"/>
  </adios-group>
  <method group="checkpoint" method="MPI"/>
  <buffer size-MB="100" allocate-time="now"/>
</adios-config>
```

- Variables are combined into groups
  - Each group can have a separate I/O method
- Allows optionally setting buffer sizes etc.

Example...

```
1 adios_open(&adios_fd, "checkpoint", "checkpoint.bp", "w", MPI_COMM_WORLD);
2 #include "gwrite_checkpoint.ch"
3 adios_close(adios_fd);
```

- Code is generated by the gpp.py script
  - gread_checkpoint.ch and gwrite_checkpoint.ch
- Developers have to include the appropriate header
  - Variable names etc. have to match ADIOS's generated code

Example...

```
1 adios_groupsize = 4 \
2                 + 4 \
3                 + 8 * (rows) * (columns);
4 adios_group_size (adios_handle, adios_groupsize, &adios_totalsize);
5 adios_write (adios_handle, "rows", &rows);
6 adios_write (adios_handle, "columns", &columns);
7 adios_write (adios_handle, "matrix", matrix);
```

- Group size determined automatically
- adios_write calls write the actual data
  - Write operations are cached as much as possible

Example...

```
1 s = adios_selection_writeblock (rank);
2 adios_schedule_read (fp, s, "matrix", 1, 1, matrix);
3 adios_perform_reads (fp, 1);
4 adios_selection_delete (s);
```

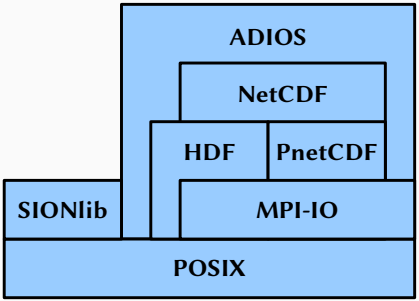
- Reading is more complex than writing
  - Offers additional functionality
- Parts of the data can be selected for reading
  - ADIOS then determines the best reading strategy
- Multiple read operations can be scheduled
  - Scheduled operations are only queued and later executed in a batch

- ADIOS uses its own file format (Binary Packed)
  - BP can be converted to HDF5, NetCDF and ASCII
- ADIOS supports data transformations
  - Compression is a common data transformation
- Read operations are scheduled for efficient execution
  - Can be used to stage data to different tiers
- Annotations can be used to improve performance further
  - `adios_{start,stop}_calculation`: Marks the calculation phases to allow performing I/O in parallel to computation
  - `adios_end_iteration`: Provides timing information for flushing data

Outline

Interaction

- Libraries use underlying interfaces
  - Interfaces can be provided by other I/O libraries
  - Sometimes depend on multiple interfaces
- Debugging might require knowledge of dependencies and their optimizations
  - Unclear which dependency causes problems



Disjoint [Bartz et al., 2015]

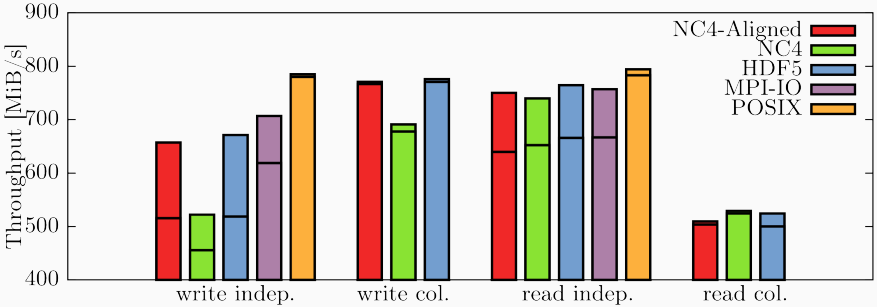


Fig. 5. Disjoint Pattern

- Clients are responsible for contiguous blocks of data
  - Regions are disjoint to avoid conflicts and locking
- Each client potentially communicates with all servers

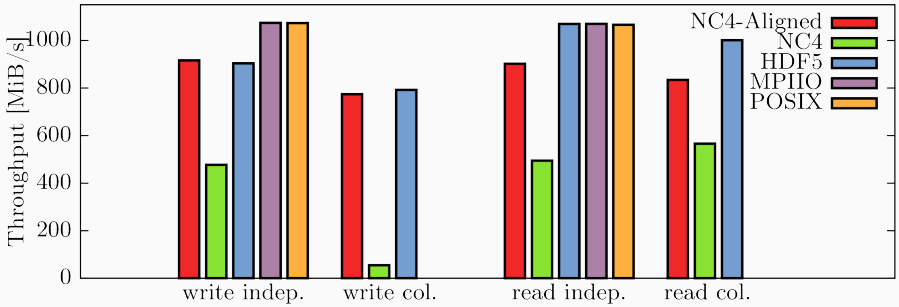


Fig. 6. 1-OST Pattern

- Each client communicates with exactly one server
  - Lower communication overhead and fewer potential conflicts

Access and Chunk Sizes [Bartz et al., 2015]

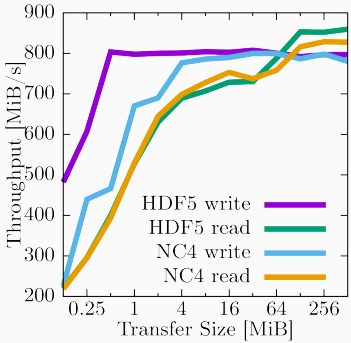


Fig. 7. Varying Transfer Size

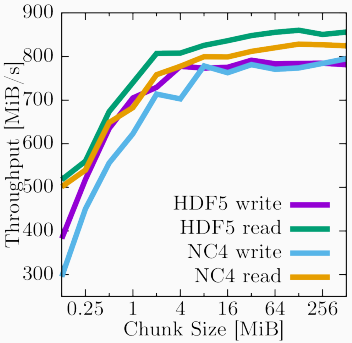


Fig. 8. Chunked Layout

- Performance heavily depends on access and chunk sizes

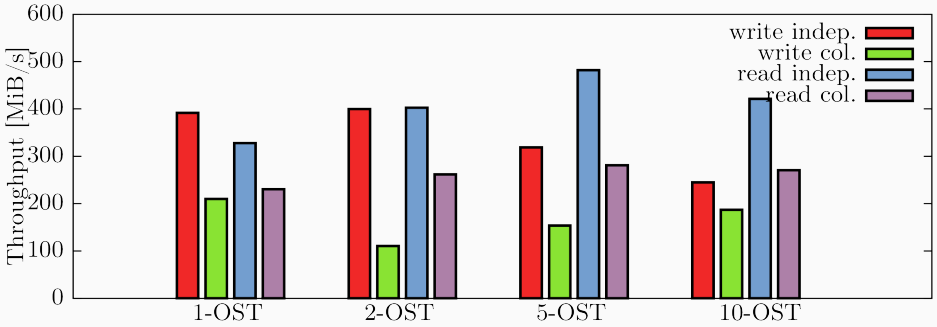


Fig. 9. 1- to 10-OST Pattern, HDF5 with Chunked I/O

- Performance varies depending on the number of servers to contact

Conclusion

- Interaction between I/O libraries is complex
  - Achievable performance cannot be predicted easily
  - Potential performance problems and optimizations on all layers
  - Analysis is complicated by having to capture all layers
- Optimizations are necessary for almost all storage systems
  - HDF5 and SIONlib allow adapting to the file system boundaries
    - NetCDF only has limited support for alignment

Outline

- Low-level I/O interfaces are often not convenient to use
  - I/O libraries provide high-level interfaces for structured access
  - Annotations and metadata enable exchanging of data
- Zoo of libraries available for different use cases
  - Analysis of errors and performance problems is complicated
  - SIONlib works around performance problems of current file systems
  - NetCDF and HDF provide similar functionality for self-describing data

References

[Bartz et al., 2015] Bartz, C., Chasapis, K., Kuhn, M., Nerge, P., and Ludwig, T. (2015). **A Best Practice Analysis of HDF5 and NetCDF-4 Using Lustre.** In Kunkel, J. M. and Ludwig, T., editors, *High Performance Computing - 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings*, volume 9137 of *Lecture Notes in Computer Science*, pages 274–281. Springer.

[SIONlib, 2021] SIONlib (2021). **File Format.** `https://apps.fz-juelich.de/jsc/sionlib/docu/fileformat_page.html`.