

# MPI-IO

## Parallel Storage Systems

2026-05-21



---

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>

## MPI-IO

Review

Introduction

Concepts and Functionality

Performance Considerations

Summary

- Across how many servers would you distribute a 1 GiB file?
  1. One server
  2. 64 servers
  3. 1,024 servers
  4. As many servers as possible

- Across how many servers would you distribute a 1 GiB file?
  1. One server ✓
  2. 64 servers ✓
  3. 1,024 servers
  4. As many servers as possible

- Why is the starting server for data distribution often chosen randomly?
  - Easy implementation
  - Even load distribution
  - Fault tolerance

- Why is the starting server for data distribution often chosen randomly?
  - Easy implementation
  - Even load distribution ✓
  - Fault tolerance

- What is the difference between POSIX and NFS semantics?
  1. POSIX guarantees each write will be synchronized to storage immediately
  2. POSIX guarantees each write will be visible by other processes immediately
  3. NFS guarantees each write will not be visible by other processes until the file is closed
  4. NFS guarantees data will be synchronized to storage when the file is closed

- What is the difference between POSIX and NFS semantics?
  1. POSIX guarantees each write will be synchronized to storage immediately
  2. POSIX guarantees each write will be visible by other processes immediately ✓
  3. NFS guarantees each write will not be visible by other processes until the file is closed
  4. NFS guarantees data will be synchronized to storage when the file is closed

- When would you stripe a directory across multiple servers?
  1. Always
  2. When it contains more than 1,000 files
  3. When it contains more than 1,000,000 files
  4. Never

- When would you stripe a directory across multiple servers?
  1. Always
  2. When it contains more than 1,000 files
  3. When it contains more than 1,000,000 files ✓
  4. Never

## MPI-IO

Review

**Introduction**

Concepts and Functionality

Performance Considerations

Summary

- Parallel applications require efficient parallel I/O
  - Synchronous and serial I/O are bottlenecks
- Synchronous I/O causes all tasks to wait for completion
- Serial I/O requires sending all data to a chosen task
  - One task cannot hold all data, must be done iteratively
- Common scenarios in HPC
  - Reading input data
    - Starting conditions, large data sets
  - Writing output data
    - Result data, checkpoints



[Gorda, 2013]

- MPI-IO denotes the I/O part of the MPI standard
  - MPI-IO was introduced with MPI 2.0 in 1997
  - Parallel applications often use MPI anyway
- The most popular implementation is called ROMIO
  - ROMIO is being developed and distributed as part of MPICH
  - It is also being used by OpenMPI and other MPICH derivatives
  - Uses the Abstract-Device Interface for I/O (ADIO)
- An alternative implementation in OpenMPI is called OMPIO

- MPI-IO provides element-based access
  - POSIX only provides a byte stream
- I/O interface is very similar to the communication interface
  - Reading and writing behave like sending and receiving
  - Collective and non-blocking operations are supported
  - MPI-IO also supports derived datatypes
- MPI-IO is typically not used directly by applications
  - Instead, I/O libraries such as HDF5 use it internally

- MPI-IO provides the basis for many I/O libraries
  - HDF5 and NetCDF use MPI-IO for parallel access to shared files
  - ADIOS also supports MPI-IO for parallel I/O
- POSIX can also be used for parallel I/O
  - HDF5 and NetCDF only do serial I/O via POSIX
  - ADIOS also supports parallel I/O via POSIX, but not to a shared file
- ROMIO contains efficient algorithms and implementations for parallel I/O
  - Frees higher layers from having to implement them as well
  - Libraries can focus on their primary tasks

- MPI-IO abstracts from the underlying file systems
  - MPI-IO provides its own syntax and semantics
- ROMIO supports a wide range of architectures
  - IBM SP, Intel Paragon, HP Exemplar, SGI Origin2000, Cray T3E, NEC SX-4 etc.
- ROMIO also supports many file systems
  - IBM PIOFS, Intel PFS, HP/Convex HFS, SGI XFS, NEC SFS, PVFS, Lustre, NFS, NTFS, Unix File System (UFS) etc.

- MPI-IO's interface can be used by applications and libraries
  - Provides portability across a wide range of file systems and architectures
- File system specifics are contained in ADIO modules
  - Allows providing the best possible performance
    - For instance, data distribution functions are often not portable
  - Also allows hiding different file system syntax and semantics
- It also contains generic optimizations for parallel I/O
  - Optimizations are especially important with growing numbers of processes
  - We will take a look at Data Sieving and Two-Phase I/O later

## MPI-IO

Review

Introduction

**Concepts and Functionality**

Performance Considerations

Summary

- File
  - Files are opened collectively by all processes in a communicator
  - Access can be done sequentially or randomly
  - Files are a collection of typed elements
- File pointer
  - File pointer determines position within a file (like POSIX)
  - Individual or shared file pointers are possible

- Data type
  - Smallest possible unit used for accessing a file
  - Can also be an elementary type or a derived data type
- Displacement
  - Determines the position a file view begins at
  - Expressed as a byte position relative from the start of a file
    - Can be used for headers etc.

- File type
  - A pattern that describes the structure of a file
  - Consists of data types and holes
  - Pattern is repeated within the file

Data Type



Hole



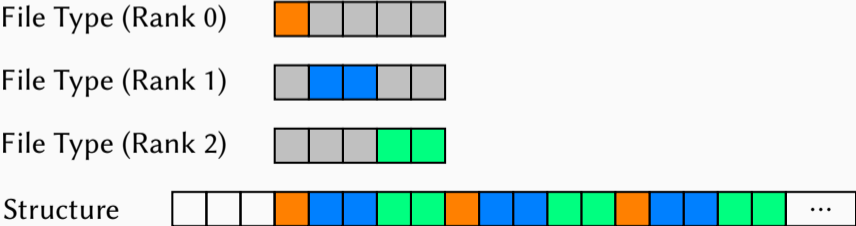
File Type



Structure



- File view
  - Process-specific view of the file
  - Determined by displacement, data type and file type



- Offset
  - Offset describes a position within a file
  - Given as a number of data types
  - Interpreted relative to the current file view
- File size
  - Size of the file in bytes

- File handle
  - Describes an open file, similar to a file descriptor
  - Required for almost all other operations
- Hints
  - Additional information that can be passed to the implementation
  - Typically used to improve performance or reduce overhead

- `MPI_File_open`: Opens a file
  - `comm`: Communicator for collective open
  - `filename`: Path to the file
  - `amode`: Access mode used for opening
  - `info`: Optional hints
  - `fh`: File handle for further operations
- `MPI_File_close`: Closes a file
  - `fh`: File handle

```
1 int MPI_File_open (MPI_Comm comm,  
2                   char* filename,  
3                   int amode,  
4                   MPI_Info info,  
5                   MPI_File* fh)  
6  
7 int MPI_File_close (MPI_File* fh)
```

- `MPI_File_open` is a collective operation
  - All processes have to open the same file (potentially different names)
  - Process-local files are possible by using `MPI_COMM_SELF`
- File name is implementation-specific
  - ROMIO allows specifying an ADIO module explicitly
    - For example: `pvfs2:/pvfs/path/to/file`
- The initial file view is a byte stream (like POSIX)
  - That is, all processes have access to the whole file

- `MPI_File_open` offers multiple access modes
  - `MPI_MODE_RDONLY`: Read-only
  - `MPI_MODE_RDWR`: Read and write
  - `MPI_MODE_WRONLY`: Write-only
  - `MPI_MODE_CREATE`: Create file if it does not exist
  - `MPI_MODE_EXCL`: Return error if file exists already
  - `MPI_MODE_DELETE_ON_CLOSE`: Delete file on close

- `MPI_File_open` offers multiple access modes...
  - `MPI_MODE_UNIQUE_OPEN`: File will not be opened concurrently
  - `MPI_MODE_SEQUENTIAL`: File will be accessed only sequentially
  - `MPI_MODE_APPEND`: File pointers will be set to the end of file
- Access modes can also be combined when it makes sense
- Some modes offer potential for optimizations
  - Caching and locking can be optimized for `MPI_MODE_UNIQUE_OPEN`
  - Read ahead can be used for `MPI_MODE_SEQUENTIAL`

- What happens if a file opened with `MPI_MODE_UNIQUE_OPEN` is opened elsewhere?
  1. MPI will fall back to regular open
  2. MPI will abort the application
  3. MPI will crash
  4. Undefined behavior

- What happens if a file opened with `MPI_MODE_UNIQUE_OPEN` is opened elsewhere?
  1. MPI will fall back to regular open
  2. MPI will abort the application
  3. MPI will crash
  4. Undefined behavior ✓

- `MPI_File_seek`: Sets the file pointer
  - `fh`: File handle
  - `offset`: Offset within file
  - `whence`: Positioning mode
  - Behaves like POSIX's `lseek`

```
1  int MPI_File_seek (MPI_File fh,  
2                          MPI_Offset offset,  
3                          int whence)
```

- MPI supports three modes for positioning

1. Individual file pointers
2. Shared file pointers
3. Explicit offsets

### 1. Individual file pointers

- File pointer is process-local and updated with each operation
  - Behaves like POSIX's read and write
- Accesses by different processes can lead to conflicts

## 2. Shared file pointers

- File pointer is global and updated with each operation
  - Syntax: MPI\_ . . . \_shared and MPI\_ . . . \_ordered
- Can be used to ensure different processes do not access the same data
- Support can be limited depending on the used file system

## 3. Explicit offsets

- Offset is specified with each operation
  - Behaves like POSIX's pread and pwrite
  - Syntax: MPI\_ . . . \_at
- Concurrent access by multiple processes can be done safely
  - Requires calculating the offset manually

- `MPI_File_seek` and `MPI_File_seek_shared` set the file pointer
  - Both functions support three positioning modes
    1. `MPI_SEEK_SET`: File pointer is set to specified offset
    2. `MPI_SEEK_CUR`: File pointer is incremented by specified offset
    3. `MPI_SEEK_END`: File pointer is set to end of file plus offset
  - The offset can also be negative (especially useful for `MPI_SEEK_END`)

- Shared file pointers can be used for coordinated access
  - All processes use the same global file pointer
    - Access conflicts can be avoided like this
  - Accesses update the file pointer for all processes
- Can be problematic to implement efficiently
  - Requires some form of (distributed) locking
    - File pointer can only be updated by one process at a time
  - Complicated to scale when using large number of processes
    - Changes have to be announced to all other processes
  - Shared file pointers are not supported by all file systems
    - OrangeFS does not support locks and therefore no shared file pointers

- MPI\_...\_shared can be used for individual operations
  - Operations can be performed in an arbitrary order
- MPI\_...\_ordered can be used for collective operations
  - Operations are performed according to the processes' ranks
- Useful for several use cases
  - Shared log file, where all processes append new entries
  - Output data to be written to a file in the processes' order

- `MPI_File_read`: Reads from file
  - `fh`: File handle
  - `buf`: Buffer to read into
  - `count`: Number of elements
  - `type`: Element type
  - `status`: Read status
  - Behaves like `read`
- `MPI_File_write`: Writes to file
  - `fh`: File handle
  - `buf`: Buffer to write from
  - `count`: Number of elements
  - `type`: Element type
  - `status`: Write status
  - Behaves like `write`

```
1 int MPI_File_read (MPI_File fh,  
2                   void* buf,  
3                   int count,  
4                   MPI_Datatype type,  
5                   MPI_Status* status)  
6  
7 int MPI_File_write (MPI_File fh,  
8                   void* buf,  
9                   int count,  
10                  MPI_Datatype type,  
11                  MPI_Status* status)
```

```
1 MPI_File fh;
2 MPI_Offset size;
3 MPI_Status status;
4 int nbytes;
5
6 MPI_File_open(MPI_COMM_WORLD, "/tmp/mpi-io",
7               MPI_MODE_RDWR | MPI_MODE_CREATE | MPI_MODE_DELETE_ON_CLOSE,
8               MPI_INFO_NULL, &fh);
9 MPI_File_write(fh, data, sizeof(data), MPI_BYTE, &status);
10 MPI_Get_count(&status, MPI_BYTE, &nbytes);
11 MPI_File_get_size(fh, &size);
12 MPI_File_close(&fh);
```

- Basic structure is very similar to POSIX
  - Separate call of `MPI_Get_count` necessary (return value is a 32 bit integer)
  - Metadata access is limited to `MPI_File_get_size`

- MPI-IO only supports a few operations, especially for metadata
  - Its interface is more akin to an object store than a file system
- There is no support for directory operations at all
  - Full path must be known to open a file
- File management is also very limited
  - Files can only be create via `MPI_File_open`
- Files can be grown and shrunk
  - `MPI_File_set_size` and `MPI_File_preallocate`
- No support for rich metadata like in POSIX
  - No equivalent to `stat`, files size via `MPI_File_get_size`

- MPI-IO supports non-contiguous data types
  - Enables access to complex structures using a single I/O call
  - Provides convenience for developers but also potential for optimizations
- Accesses are also possible to do manually
  - Would introduce developer and performance overhead
  - Similar to `readv`, `writew`, `aio_read`, `aio_write` and `lio_listio`
    - `readv` and `writew` can only access contiguous areas and are thus not as powerful

- `MPI_Type_vector`: Create vector type
  - `count`: Number of blocks
  - `blocklength`: Number of elements in each block
  - `stride`: Distance between blocks
  - `old`: Old data type
  - `new`: New data type
- Example: 3×3 matrix diagonal


```
1  int MPI_Type_vector (int count,  
2                        int blocklength,  
3                        int stride,  
4                        MPI_Datatype old,  
5                        MPI_Datatype* new)  
6  
7  MPI_Type_vector(3, 1, 4,  
8                  MPI_DOUBLE, &newtype);  
9  MPI_Type_commit(&newtype);  
10 MPI_File_write(fh, buffer,  
11                1, newtype, &status);
```


```
1 MPI_Type_vector(3, 1, 4, MPI_DOUBLE, &newtype);
```


|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

- Assumption: Matrix is stored row- or column-wise in memory
  - A 3×3 matrix has three diagonal elements
  - Each diagonal element is a double value
  - Diagonal elements are separated by three elements (have a distance of four)

- MPI-IO supports collective I/O for improved performance
  - All processes perform their access at the same time in a coordinated fashion
    - Syntax: `MPI_..._all`
  - Collective operations provide additional information for potential optimizations
    - Individual operations can result in random access patterns
- Example: Small non-contiguous accesses
  - Each processes accesses several small areas within the file
  - All processes together access the whole file

File Type (Rank 0) 

File Type (Rank 1) 

File Type (Rank 2) 

Structure 

- MPI-IO also supports non-blocking I/O operations
  - Work similar to non-blocking communication operations
    - Syntax: MPI\_...\_i...
  - Allows overlapping I/O and computation (and more)
    - Enables applications to be productive while performing I/O
    - Speedup is limited to 2 with only I/O and computation
- Status can be checked using standard MPI functions
  - For example, MPI\_Wait and MPI\_Test

- Non-blocking collective I/O operations are called split collectives
  - Separated due to optimization and implementation reasons
    - Syntax: MPI\_ . . . \_begin and MPI\_ . . . \_end
- Split collectives have several limitations
  - Per process and file only one split collective is allowed at a time
  - Cannot be combined with regular collective operations
  - No collective I/O operations are allowed while a split collective is in progress
  - Implementations are allowed to perform blocking operations internally

- What happens if the buffer of a non-blocking operation is reused before the operation is finished?
  1. MPI will finish the non-blocking operation first
  2. MPI will abort the application
  3. MPI will crash
  4. Undefined behavior

- What happens if the buffer of a non-blocking operation is reused before the operation is finished?
  1. MPI will finish the non-blocking operation first
  2. MPI will abort the application
  3. MPI will crash
  4. Undefined behavior ✓

- Hints can be used to provide implementations with additional information
  - Typically used for optimizations
  - Can be specified for `MPI_File_open` and others
- Examples:
  - Number of devices a file should be distributed across
  - Size used for distributing blocks
  - Information about the data layout
- Hints are optional and do not have to be specified
  - Implementations are also free to ignore hints as they see fit
  - Some implementations allow specifying hints via environment variables

## MPI-IO

Review

Introduction

Concepts and Functionality

**Performance Considerations**

Summary

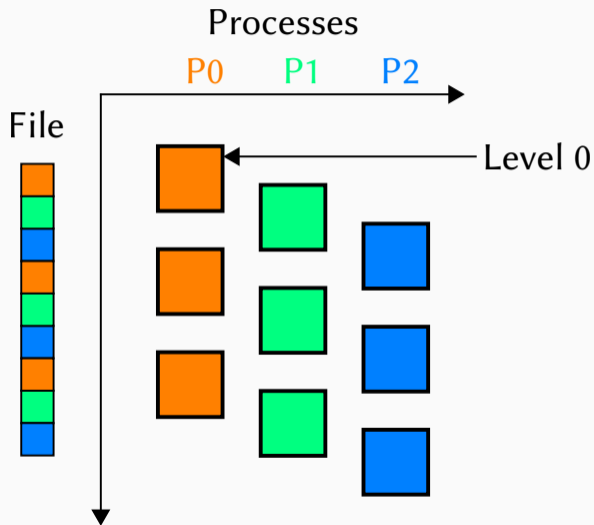
- Achievable performance depends on the used operations
  - Large accesses are typically more efficient than small ones
  - Contiguous accesses are usually better than non-contiguous ones
- MPI-IO offers several possibilities of performing I/O
  - Contiguous vs. non-contiguous, individual vs. collective
- Example with a  $3 \times 3$  matrix:
  - Matrix is stored row-wise in memory
  - Matrix should be read by three processes
  - Each process is responsible for one column

- Each process performs individual accesses
- Contiguous region is read in each iteration

```
1 for (i = 0; i < 3; i++)  
2 {  
3     MPI_File_seek(fh, ...);  
4     MPI_File_read(fh, ...,  
5         1, MPI_DOUBLE, ...);  
6 }
```

- Each process performs individual accesses
  - One row is read per iteration
- Contiguous region is read in each iteration
  - Individual accesses lead to random pattern

```
1 for (i = 0; i < 3; i++)
2 {
3     MPI_File_seek(fh, ...);
4     MPI_File_read(fh, ...,
5                   1, MPI_DOUBLE, ...);
6 }
```

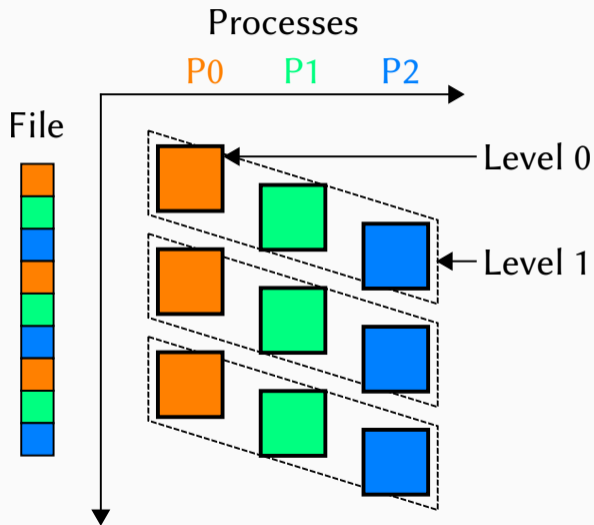


- Processes perform coordinated collective access
- Contiguous region is read in each iteration

```
1 for (i = 0; i < 3; i++)
2 {
3     MPI_File_seek(fh, ...);
4     MPI_File_read_all(fh, ...,
5                       1, MPI_DOUBLE, ...);
6 }
```

- Processes perform coordinated collective access
  - One row is read per iteration
- Contiguous region is read in each iteration
  - Collective access leads to contiguous pattern

```
1 for (i = 0; i < 3; i++)
2 {
3     MPI_File_seek(fh, ...);
4     MPI_File_read_all(fh, ...,
5                       1, MPI_DOUBLE, ...);
6 }
```

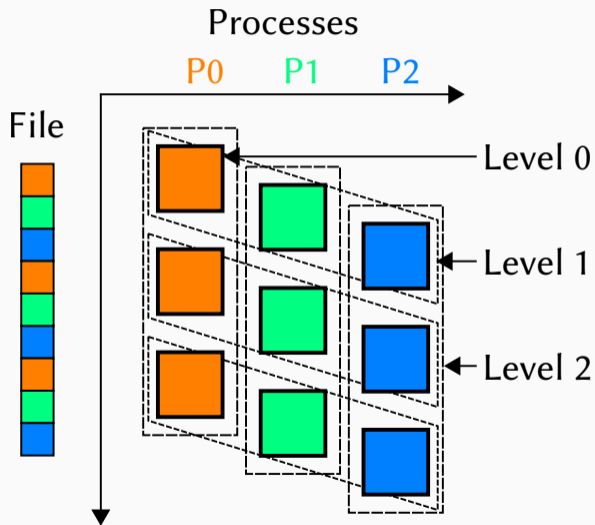


- Each process performs individual accesses
- Non-contiguous column is read by each process

```
1 MPI_Type_vector(3, 1, 3,  
2     MPI_DOUBLE, &newtype);  
3 MPI_Type_commit(&newtype);  
4  
5 MPI_File_seek(fh, ...);  
6 MPI_File_read(fh, ...,  
7     1, newtype, ...);
```

- Each process performs individual accesses
  - All columns are read
- Non-contiguous column is read by each process
  - Individual accesses lead to random pattern

```
1 MPI_Type_vector(3, 1, 3,  
2     MPI_DOUBLE, &newtype);  
3 MPI_Type_commit(&newtype);  
4  
5 MPI_File_seek(fh, ...);  
6 MPI_File_read(fh, ...,  
7     1, newtype, ...);
```

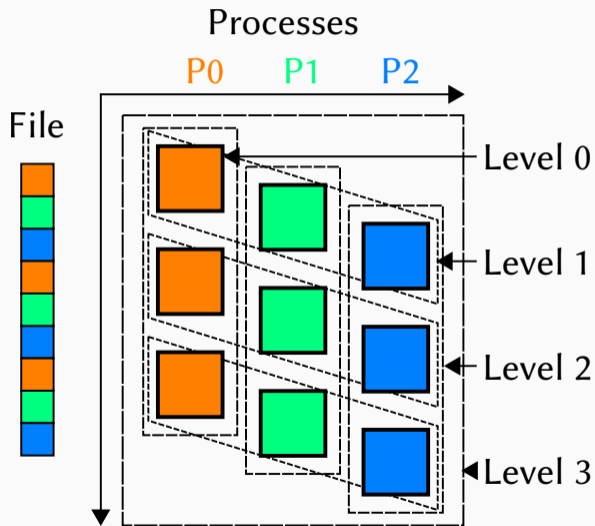


- Processes perform coordinated collective access
- Non-contiguous column is read by each process

```
1 MPI_Type_vector(3, 1, 3,  
2     MPI_DOUBLE, &newtype);  
3 MPI_Type_commit(&newtype);  
4  
5 MPI_File_seek(fh, ...);  
6 MPI_File_read_all(fh, ...,  
7     1, newtype, ...);
```

- Processes perform coordinated collective access
  - All columns are read
- Non-contiguous column is read by each process
  - Collective access leads to contiguous pattern

```
1 MPI_Type_vector(3, 1, 3,  
2     MPI_DOUBLE, &newtype);  
3 MPI_Type_commit(&newtype);  
4  
5 MPI_File_seek(fh, ...);  
6 MPI_File_read_all(fh, ...,  
7     1, newtype, ...);
```



- Reminder: POSIX has strict consistency and coherence requirements
  - Changes have to be visible globally after a write
  - I/O should be performed atomically
  - Requirements are relatively easy to support locally
- Efficient parallel I/O is complicated by POSIX semantics
  - Data cannot be cached as effectively due to synchronization
  - Atomicity might require distributed locks

- MPI-IO has less strict requirements than POSIX
  - Changes only have to be visible to the current process
  - Non-overlapping or non-concurrent operations are handled correctly
- Changes do not have to be visible globally immediately after an operation
  - Allows reducing locking overhead and increasing scalability
- MPI-IO semantics is enough for most scientific applications
  - For example, non-overlapping accesses to computed data are common

## 1. Sync transfers changes to the file system

```
1 MPI_File_sync(fh);  
2 MPI_Barrier(MPI_COMM_WORLD);  
3 MPI_File_sync(fh);
```

1. Sync transfers changes to the file system
2. Barrier synchronizes all processes

```
1 MPI_File_sync(fh);  
2 MPI_Barrier(MPI_COMM_WORLD);  
3 MPI_File_sync(fh);
```

1. Sync transfers changes to the file system
2. Barrier synchronizes all processes
  
3. Sync makes changes visible to all processes

```
1 MPI_File_sync(fh);  
2 MPI_Barrier(MPI_COMM_WORLD);  
3 MPI_File_sync(fh);
```

1. Sync transfers changes to the file system
2. Barrier synchronizes all processes
  - Necessary for correct synchronization
  - First sync has to be finished for all processes before second sync is called
3. Sync makes changes visible to all processes

```
1 MPI_File_sync(fh);  
2 MPI_Barrier(MPI_COMM_WORLD);  
3 MPI_File_sync(fh);
```

- MPI-IO's atomic mode guarantees sequential consistency
  - Has to be enabled explicitly using `MPI_File_set_atomicity`
- Allows MPI-IO to handle overlapping and concurrent accesses correctly
  - Similar to stricter POSIX semantics
- Support depends on file system and is limited
  - ROMIO supports atomic mode, OMPIO does not
  - Typically requires distributed locks
    - Not all file systems implement locks, limiting availability
    - Reminder: OrangeFS does not support locks

## MPI-IO

Review

Introduction

Concepts and Functionality

Performance Considerations

Summary

| Positioning              | Blocking                          | Individual                  | Collective                               |
|--------------------------|-----------------------------------|-----------------------------|--|
| Explicit Offset          | Blocking                          | read_at<br>write_at         | read_at_all<br>write_at_all              |
|                          | Non-Blocking and Split Collective | iread_at                    | read_at_all_begin<br>read_at_all_end     |
|                          |                                   | iwrite_at                   | write_at_all_begin<br>write_at_all_end   |
| Individual File Pointers | Blocking                          | read<br>write               | read_all<br>write_all                    |
|                          | Non-Blocking and Split Collective | iread                       | read_all_begin<br>read_all_end           |
|                          |                                   | iwrite                      | write_all_begin<br>write_all_end         |
| Shared File Pointer      | Blocking                          | read_shared<br>write_shared | read_ordered<br>write_ordered            |
|                          | Non-Blocking and Split Collective | iread_shared                | read_ordered_begin<br>read_ordered_end   |
|                          |                                   | iwrite_shared               | write_ordered_begin<br>write_ordered_end |

- MPI-IO is defined similar to MPI's communication operations
  - Supports collectives, derived data types etc.
- Files are a collection of typed elements
  - Each process has its own file view
  - Multiple data representations enable portability
- Positioning can be performed using different modes
  - Explicitly, with individual file pointers or a shared file pointer
- Different access modes allow optimizing parallel I/O
  - Non-contiguous, collective and non-blocking operations can improve performance

## References

[Gorda, 2013] Gorda, B. (2013). **HPC Technologies for Big Data.**

[http://www.hpcadvisorycouncil.com/events/2013/Switzerland-Workshop/Presentations/Day\\_2/3\\_Intel.pdf](http://www.hpcadvisorycouncil.com/events/2013/Switzerland-Workshop/Presentations/Day_2/3_Intel.pdf).

[Thakur et al., 2002] Thakur, R., Gropp, W., and Lusk, E. L. (2002). **Optimizing noncontiguous accesses in MPI-IO.** *Parallel Comput.*, 28(1):83–105.

- MPI-IO supports multiple data representations
  - Data portability is an important aspect of MPI-IO
  - Often also handled by I/O libraries based upon MPI-IO
- Three possible representations
  - native: Data is not converted in any way and are stored as in memory
  - internal: Portable data representation across all platforms supported by used implementation
  - external32: Portable data representation across all platforms and implementations, potential loss of precision and performance
- Users can also define their own data representations