HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 14

Fabian Wermelinger
Harvard University
CS205

Thursday, March 9th 2023

LAST TIME

- A few more topics on blocking procedures:
  - Probing messages
  - Message matching
  - Shift operations (with virtual topologies)
- MPI message protocols (eager/rendezvous)
- Non-blocking semantics
- Example for 1D diffusion equation
- Non-blocking collectives

TODAY

Main topic: ***MPI Input/Output (I/O) operations***

Details:

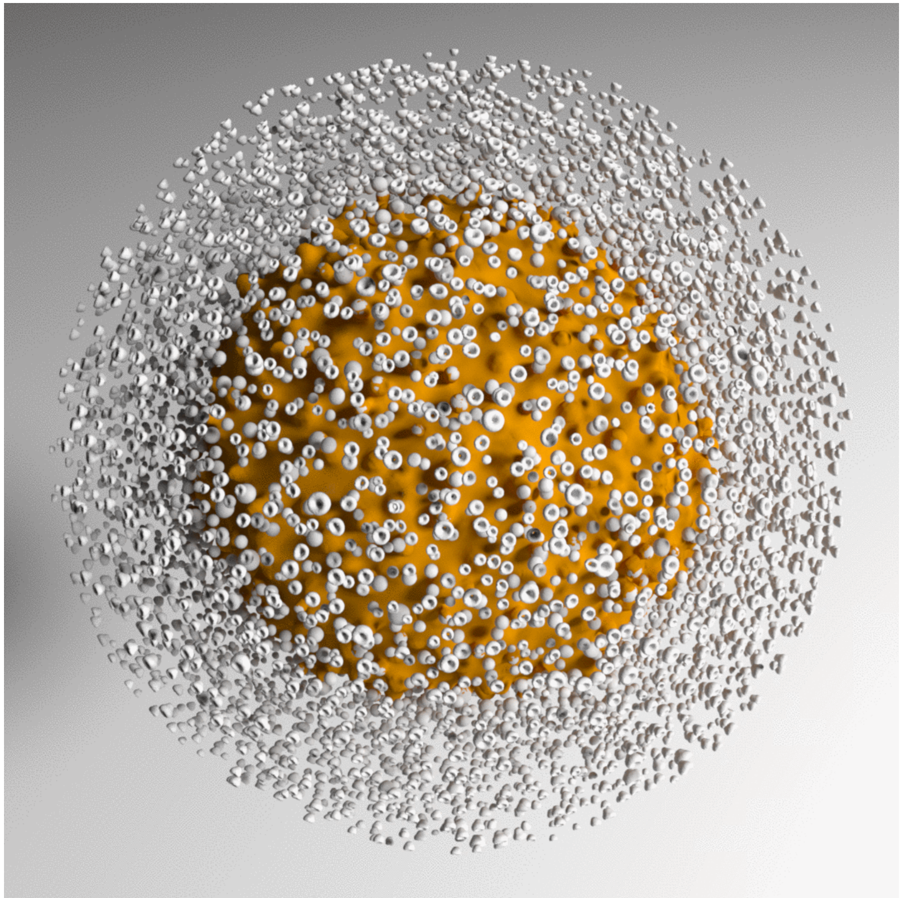
- *Quiz 3*
- MPI I/O terminology
- MPI file views
- MPI file management
- Methods for MPI I/O data access
- File pointer manipulations
- Example for parallel I/O with HDF5
- What is left in the MPI standard?

AGENDA CHECK:

- Project milestone 3 → Presentation of your concrete project proposal to the class. Please see https://harvard-iacs.github.io/2023-CS205/pages/project.html#M3 for the details. Due in the week after spring break. Presentations are during class in increasing team order: team 1-11 on Tuesday, 12-22 on Thursday. The schedule is available at this link: https://code.harvard.edu/CS205/main/tree/master/project/01_proposal_pres_sched.csv

INPUT/OUTPUT OPERATIONS

- When you develop applications, you will often need facilities to deal with input and output of data.
- ***Input:*** you may need to be able to *restart* your program from a previous state that you have saved in a file (checkpoints).
- ***Output:*** you may need to analyse your data with a third-party program for which you must *write* a certain subset of your data in a certain interval.
- Reading or writing data often (especially large amounts of data) will quickly become the bottleneck in your application. *Efficient I/O is important.* → Sometimes it makes sense to perform (parallel) in-situ analysis of your data instead of writing data files.



Visualization of cloud cavitation collapse with 12500 bubbles. Data for the visualization includes pressure (yellow isosurface) and gas volume fraction (bubbles) which are 884GB of data for each field (uncompressed). Still manageable for Peta-scale simulations. Exa-scale will be difficult to treat even larger amounts of data for file I/O.

POSIX I/O

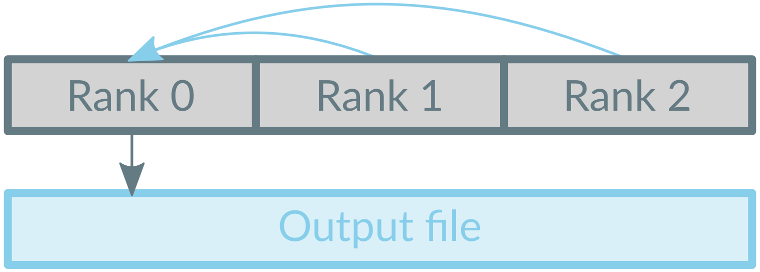
- POSIX provides a model of a widely portable file system, but is not suitable for *parallel* I/O.
- Optimizations for efficient parallel file I/O include: grouping, collective buffering and disk-directed I/O.

1D diffusion problem from last time

```
void dump_posix(const std::vector<double> &v)
{
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    const int N = static_cast<int>(v.size()) - 2;
    std::vector<double> u(size * N);
    MPI_Gather(&v[1], N, MPI_DOUBLE,
        u.data(), N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    if (0 == rank) {
        std::ofstream out("u.dat", std::ios::app);
        out << std::scientific;
        for (size_t i = 0; i < u.size(); ++i) {
            out << ' ' << u[i];
        }
        out << '\n';
    }
}
```

- All processes send their data to the *root* rank (`MPI_Gather`).
- Only the *root* rank writes the data (sequential write).

Pros: one single file
Cons: not scalable!



ALTERNATIVE: EACH RANK WRITES A FILE

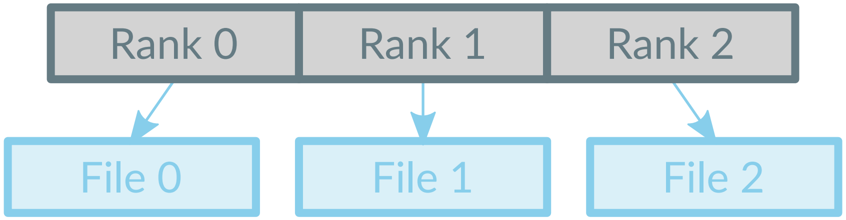
- To avoid the `MPI_Gather`, each rank writes its data to a separate file.
- Post-processing the data from many split files is very cumbersome and does not scale. You must write post-processing routines that can deal with the split file structure.
- Third-party visualization tools like ParaView will require a custom reader.

1D diffusion problem from last time

```
void dump_rank(const std::vector<double> &v)
{
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    const int N = static_cast<int>(v.size()) - 2;
    std::ofstream out("u_rank" + std::to_string(rank)
                      + ".dat", std::ios::app);
    out << std::scientific;
    for (int i = 1; i < N; ++i) {
        out << ' ' << v[i];
    }
    out << '\n';
}
```

- All processes write their data in separate file (parallel).

Pros: *scalable*
Cons: *many (possibly) small files which is bad for HPC filesystems (e.g. Lustre)*



MPI I/O

- MPI provides scalable procedures for efficient parallel I/O. See Chapter 14 in the MPI 4.0 standard.
- If you deal with large I/O you must work with *binary* data.

1D diffusion problem from last time

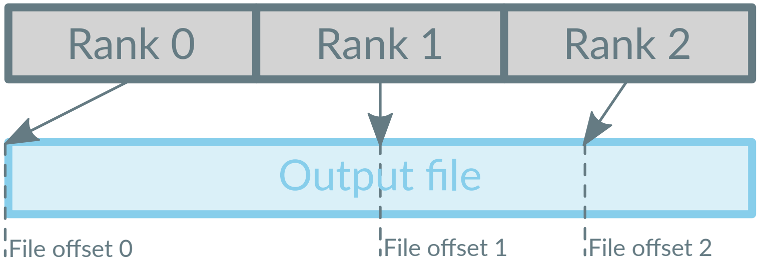
```cpp
void dump_mpi_binary(const std::vector<double> &v)
{
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_File fh; // MPI file handle
    MPI_Offset bytes, file_offset;

    const int N = static_cast<int>(v.size()) - 2;
    bytes = N * sizeof(double);
    file_offset = rank * bytes;

    MPI_File_open(MPI_COMM_WORLD,
                  "u_mpi.bin",
                  MPI_MODE_CREATE | MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &fh);
    MPI_File_get_size(fh, &bytes); // append to file
    file_offset += bytes;
    MPI_File_write_at_all(fh, file_offset, v.data(),
        N, MPI_DOUBLE, MPI_STATUS_IGNORE);
    MPI_File_close(&fh);
}
```

- All processes write their data into *the same* file in parallel.
- Think of a file as a stream of bytes → byte offsets for different ranks are now important.

Pros: parallel and scalable
Cons: can lead to bad performance if not carefully implemented



MPI I/O

- The logic behind MPI I/O is the same as in POSIX:
  1. Open or create a *file handle*.
  2. Read or write data through the *file handle*.
  3. *Close* the file when done.
- Many *I/O libraries* make use of MPI I/O capabilities. For example:
  - ***HDF5:*** https://www.hdfgroup.org/solutions/hdf5 *(large community, Python support)*
  - ***VTK:*** https://kitware.github.io/vtk-examples/site/VTKFileFormats/
  - ***NetCDF:*** https://www.unidata.ucar.edu/software/netcdf/
  - ***ADIOS:*** https://www.olcf.ornl.gov/center-projects/adios/

  The file format for these I/O libraries is often compatible with data readers of third-party post-processing software. Alternative XML wrappers exist for use as well, for example the e**X**tensible **D**ata **M**odel and **F**ormat (XDMF).
- Opening and closing files are *collective* operations.
- A *file handle* is associated to a *communicator*.
- Read and write operations can either be *collective* or *non-collective*.

MPI I/O TERMINOLOGY

The following are important terms related to MPI file I/O: (see Section 14.1.1 in the MPI 4.0 standard)

file
: An MPI file is an *ordered* collection of typed data items. MPI supports *random* or *sequential* access to any integral set of these items.

displacement
: A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins.

etype
: An *etype* (*elementary datatype*) is the *unit of data access and positioning*. It can be any MPI predefined datatype or derived datatype (user defined). Data access is performed in etype units, reading or writing whole data items of type *etype*. Offsets are expressed as a count of *etypes*.

filetype
: A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a *single* etype or a derived MPI data type constructed from multiple instances of the same etype.

view
: A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a *displacement*, an *etype* and a *filetype*.

offset
: An *offset* is a position in the file relative to the current view, expressed as a count of etypes.

file size and end of file
: The *size* of an MPI file is measured in bytes from the beginning of the file.

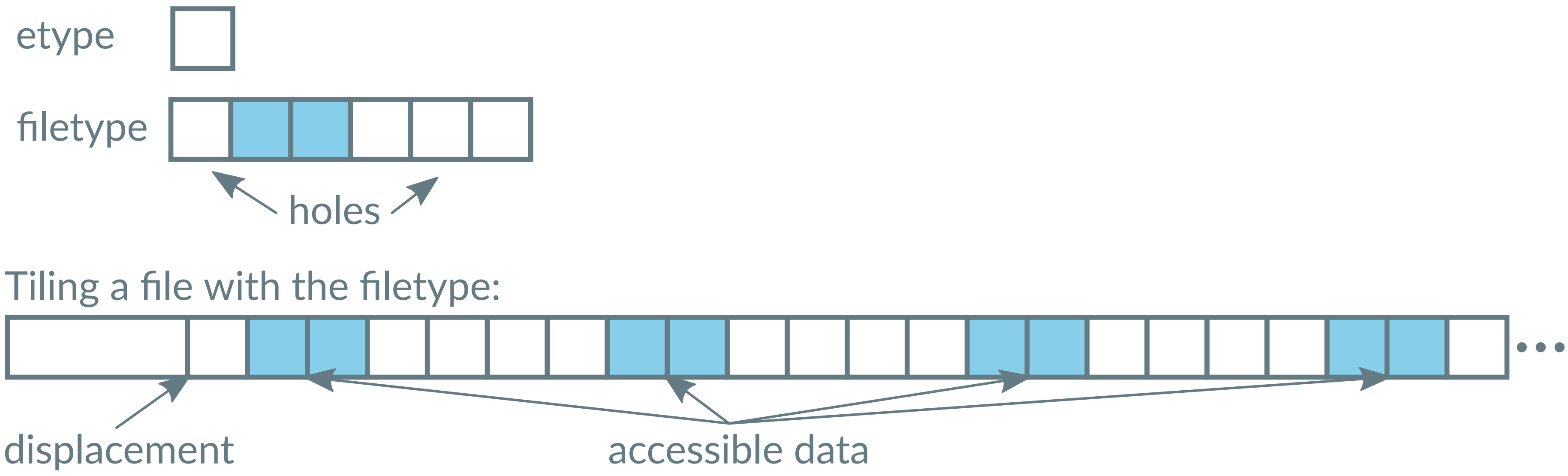
file pointer
: A *file pointer* is an implicit offset maintained by MPI. *Individual* file pointers are local to each process. *Shared* file pointers are shared by the group of processes that opened the file.

file handle
: A *file handle* is an opaque object created by `MPI_File_open` and freed by `MPI_File_close`. All operations on an open file must be through the file handle.

MPI VIEW

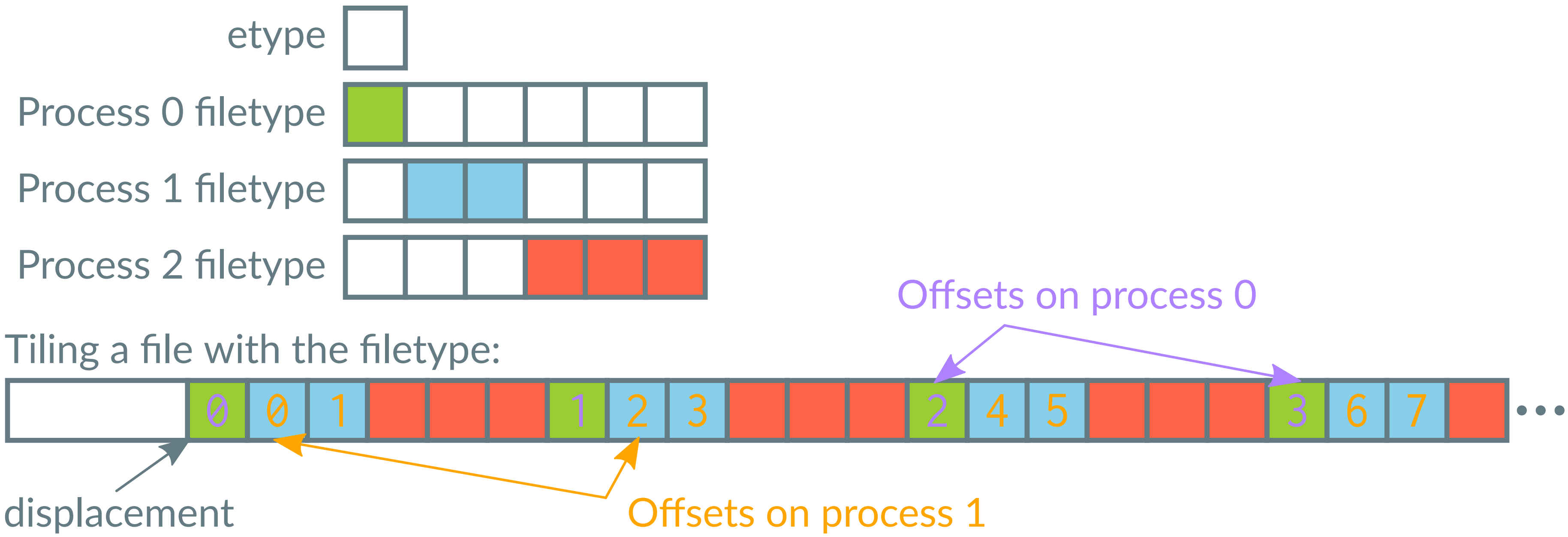
- A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes.
- Each process has its *own view*, defined by three quantities: a *displacement*, an *etype* and a *filetype*.

Etypes and filetypes:



MPI VIEW WITH MULTIPLE PROCESSES

Partitioning a file among parallel processes:



- A file is simply a *contiguous* chunk of bytes.
- The *displacement* is an *absolute* byte position relative to the beginning of the file. The displacement defines where a view begins.
- An *offset* is a position in the file *relative to the current view*, expressed as a *count of etypes*. Each process has its own view.
- MPI views can be changed by the user during program execution.
- The default view is a linear byte stream (displacement is *zero*, etype and filetype are equal to `MPI_BYTE`).
- → for HW3 you can work with a default view. See `man MPI_File_set_view` to change a view.

OPEN AND CLOSE AN MPI FILE

- A file for parallel MPI I/O can be opened with `MPI_File_open`.
- To close a file use `MPI_File_close`.
- Both of these procedures are *collective* and must be called by all ranks in the group.
- You must close all open files with `MPI_File_close` *before* calling `MPI_Finalize`.
- The communicator for MPI file I/O must be an *intra-communicator*.

The amode describes the *access mode:*

`MPI_MODE_RDONLY`	Read only access
`MPI_MODE_RDWR`	Read/write access
`MPI_MODE_WRONLY`	Write only access
`MPI_MODE_CREATE`	Create the file if it does not exist
`MPI_MODE_EXCL`	Error if creating file that already exists
`MPI_MODE_DELETE_ON_CLOSE`	Delete file on close
`MPI_MODE_UNIQUE_OPEN`	File will not be concurrently opened elsewhere
`MPI_MODE_SEQUENTIAL`	File will only be accessed sequentially
`MPI_MODE_APPEND`	Set initial position of all file pointers to the end of file

```c
int MPI_File_open(MPI_Comm comm,
                  const char *filename,
                  int amode,
                  MPI_Info info,
                  MPI_File *fh)
```

```c
int MPI_File_close(MPI_File *fh)
```

Multiple access modes can be combined with a bitwise OR. *For example, to create a write only file:* `MPI_MODE_CREATE | MPI_MODE_WRONLY`.

MPI FILE MANAGEMENT

- ***Get the size of a file:***

```
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
```

This returns the `size` of the file associated with the handle `fh` in *bytes*.

- ***Set the size of a file:***

```
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
```

Resize the file associated with handle `fh` to `size` bytes. This is a *collective* procedure and all ranks in the groups must pass an *equal* `size`. *Implementation dependent whether file space for a larger file is allocated or not.*

- ***Preallocate the size of a file:***

```
int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
```

The same as `MPI_File_set_size` but ensures the storage space is allocated for the first `size` bytes. This is a *collective* procedure and all ranks in the groups must pass an *equal* `size`. *(**Note:** Preallocation of file storage may be a time-consuming operation).*

- ***Delete a file:***

```
int MPI_File_delete(const char *filename, MPI_Info info)
```

Delete the file identified by the filename `filename`.

MPI INFO OBJECT

- Many procedures in MPI take an `info` argument of type `MPI_Info` (not just file I/O procedures). See Chapter 10 in the MPI 4.0 standard.
- It stores an unordered set of (`key`,`value`) pairs (both *strings*) and a `key` can only have *one* value.
- Some `info` hints allow the MPI library to restrict its support for certain operations in favor of performance improvements and resource utilization.
- Hints specified via `info` allow a user to provide information such as file access patterns and file system specifics to direct optimization.
- An MPI implementation is free to ignore all hints.
- If you do not need the `info` object, you can pass the `MPI_INFO_NULL` handle.
- See Section 14.2.8 in the MPI 4.0 standard for some reserved file hints (not further discussed here).

METHODS OF DATA ACCESS IN MPI

- Data is moved between files and processes by issuing *read* and *write* calls.
- There are ***three*** *orthogonal* aspects to data access:
  1. ***positioning:*** *explicit offset* vs. MPI implicit *file pointers*
  2. ***synchronism:*** blocking vs. non-blocking (and split collective)
  3. ***coordination:*** non-collective vs. collective
- Collective file operations may perform much better than their non-collective counterparts because *global* data accesses have significant potential for automatic optimization.
- MPI maintains *implicit* file pointers inside the `MPI_File` handle. ***Individual file pointers*** are per process and there is *one* ***shared file pointer*** for the rank collective.
- MPI file procedures with *explicit offsets* are of the form `MPI_File_xxx_at`, procedures that use *shared file pointers* are of the form `MPI_File_xxx_shared` and procedures that use *individual file pointers* have no special ending.
- Every non-collective procedure `MPI_File_xxx` has a *collective* counterpart with `_all` appended to their name. The *split-collective* variants have a `_begin` and `_end` appended, respectively. The collective counter part of `_shared` is `_ordered` (no additional `_all`).

OVERVIEW OF MPI DATA ACCESS PROCEDURES

positioning	*synchronism*	*coordination*: *non-collective*	*coordination*: *collective*
explicit offsets	*blocking*	`MPI_File_read_at` `MPI_File_write_at`	`MPI_File_read_at_all` `MPI_File_write_at_all`
	non-blocking	`MPI_File_iread_at` `MPI_File_iwrite_at`	`MPI_File_iread_at_all` `MPI_File_iwrite_at_all`

	split collective	N/A	`MPI_File_read_at_all_begin`
			`MPI_File_read_at_all_end`
			`MPI_File_write_at_all_begin`
			`MPI_File_write_at_all_end`

individual file pointers	*blocking*	`MPI_File_read` `MPI_File_write`	`MPI_File_read_all` `MPI_File_write_all`
	non-blocking	`MPI_File_iread` `MPI_File_iwrite`	`MPI_File_iread_all` `MPI_File_iwrite_all`

	split collective	N/A	`MPI_File_read_all_begin`
			`MPI_File_read_all_end`
			`MPI_File_write_all_begin`
			`MPI_File_write_all_end`

shared file pointers	*blocking*	`MPI_File_read_shared` `MPI_File_write_shared`	`MPI_File_read_ordered` `MPI_File_write_ordered`
	non-blocking	`MPI_File_iread_shared` `MPI_File_iwrite_shared`	N/A

	split collective	N/A	`MPI_File_read_ordered_begin`
			`MPI_File_read_ordered_end`
			`MPI_File_write_ordered_begin`
			`MPI_File_write_ordered_end`

- File access at ***explicit offsets*** measured in *bytes* from the beginning of the file.
- MPI updates the ***individual file pointers*** implicitly *per rank*.
- MPI updates the ***shared file pointers*** implicitly for the *collective*. Multiple calls to the same procedure are *serialized*.

OVERVIEW OF MPI DATA ACCESS SYNCHRONISM

positioning	*synchronism*
explicit offsets	*blocking*
	non-blocking
	split collective
individual file pointers	*blocking*
	non-blocking
	split collective
shared file pointers	*blocking*
	non-blocking
	split collective

- A blocking MPI file I/O procedure will not return until the procedure is complete.
- Non-blocking MPI file I/O is the same as non-blocking communication calls. Requires a `MPI_Request` object to *test* or *wait* for completion using `MPI_Test` or `MPI_Wait`, respectively (or one of their variations).
- Split collective calls are provided to allow the user and MPI implementation to optimize the collective operation. The `_begin` procedures are ***incomplete*** and ***non-local*** because they may or may not return before they are called in all MPI processes of the process group. They are an exception in which incomplete procedures are non-local. Split collectives may be implemented with their corresponding blocking collective routine when either the `_begin` or `_end` call is issued.
- *Non-collective* file I/O procedures can be used to access the data separately and possibly involve fewer calls than there are ranks in the communicator. These calls may perform better for small file sizes and are convenient to to deal with file headers.
- Collective calls must be issued by *all* ranks and may result in better I/O performance for general (large) files.

BLOCKING, NON-COLLECTIVE MPI FILE I/O

- The POSIX `read()/fread()` and `write()/fwrite()` are blocking, non-collective operations and use individual file pointers. The MPI *equivalents* are `MPI_File_read` and `MPI_File_write`.
- Implementations of data access routines may *buffer* data to improve performance (this does not affect reads). For writes, *the only guarantee* that data has been transferred to the storage device provides `MPI_File_sync` (*collective* call).

Individual file pointer:

```
int MPI_File_read(MPI_File fh,
                  void *buf,
                  int count,
                  MPI_Datatype datatype,
                  MPI_Status *status)
```

```
int MPI_File_write(MPI_File fh,
                   const void *buf,
                   int count,
                   MPI_Datatype datatype,
                   MPI_Status *status)
```

The *offset* is defined to be the current value of the individual file pointer (implicitly managed by MPI).

Explicit offset:

```
int MPI_File_read_at(MPI_File fh,
                     MPI_Offset offset,
                     void *buf,
                     int count,
                     MPI_Datatype datatype,
                     MPI_Status *status)
```

```
int MPI_File_write_at(MPI_File fh,
                      MPI_Offset offset,
                      const void *buf,
                      int count,
                      MPI_Datatype datatype,
                      MPI_Status *status)
```

The *offset* (in etype units) must be specified explicitly.

BLOCKING, COLLECTIVE MPI FILE I/O

- These are the same procedures as in the previous slide but *collective*.
- *All* ranks in the *intra-communicator* that called `MPI_File_open` must call these procedures.

Individual file pointer:

```
int MPI_File_read_all(MPI_File fh,
                      void *buf,
                      int count,
                      MPI_Datatype datatype,
                      MPI_Status *status)
```

```
int MPI_File_write_all(MPI_File fh,
                       const void *buf,
                       int count,
                       MPI_Datatype datatype,
                       MPI_Status *status)
```

The *offset* is defined to be the current value of the individual file pointer (implicitly managed by MPI).

Explicit offset:

```
int MPI_File_read_at_all(MPI_File fh,
                         MPI_Offset offset,
                         void *buf,
                         int count,
                         MPI_Datatype datatype,
                         MPI_Status *status)
```

```
int MPI_File_write_at_all(MPI_File fh,
                          MPI_Offset offset,
                          const void *buf,
                          int count,
                          MPI_Datatype datatype,
                          MPI_Status *status)
```

The *offset* (in etype units) must be specified explicitly.

FILE POINTER MANIPULATION

- A `MPI_File` handle has both *individual file pointers* and *shared file pointers*.
- MPI implicitly manages these file pointers. For example, when you call `MPI_File_write` the *individual file pointers* will be advanced automatically depending on the current view and how many elements you write.

Seek a file position:

You can update *individual file pointers* with

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
```

This is a *non-collective* procedure. The `offset` is in etype units and *can be negative* to search backwards. It is *erroneous* to seek to a *negative* file position. The `whence` argument can be:

- `MPI_SEEK_SET`: the pointer is set to `offset`.
- `MPI_SEEK_CUR`: the pointer is set to the current pointer position *plus* `offset`.
- `MPI_SEEK_END`: the pointer is set to the end of the file *plus* `offset`.

Similarly to update *shared file pointers*

```
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset,
                         int whence)
```

This is a *collective* procedure.

Get a file position:

You can get the current offset in an *individual file pointer* with

```
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
```

The `offset` is in etype units relative to the current view.

Similarly for *shared file pointers*

```
int MPI_File_get_position_shared(MPI_File fh,
                                 MPI_Offset *offset)
```

You can convert an *offset* to an *absolute byte position* with

```
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
                             MPI_Offset *disp)
```

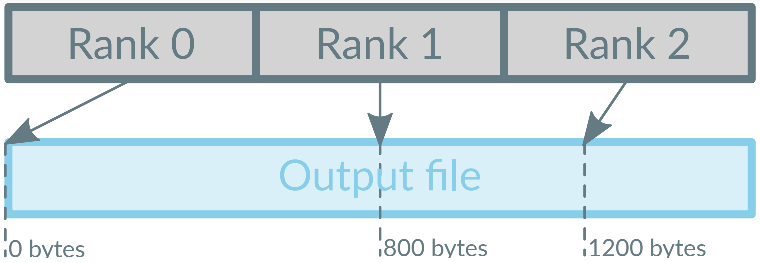
The absolute byte offset relative to the beginning of the file is returned in `disp`.

ARBITRARY BYTE CHUNKS ON DIFFERENT RANKS

- An *exclusive scan* (`MPI_Exscan`) is often required to determine the correct offset in a file.
- It is a common pattern required when writing arbitrary length outputs. Another example where exclusive scans are needed is for *compressed data* I/O.

With explicit offset:

```
void dump_chunk_at(const std::vector<double> &v)
{
    MPI_File fh; // MPI file handle
    MPI_Offset bytes, file_offset = 0;
    bytes = v.size() * sizeof(double);
    MPI_Exscan(&bytes, &file_offset, 1, MPI_OFFSET, MPI_SUM,
               MPI_COMM_WORLD);
    MPI_File_open(MPI_COMM_WORLD, "out.bin",
                  MPI_MODE_CREATE | MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &fh);
    MPI_File_write_at_all(fh, file_offset, v.data(), v.size(),
                          MPI_DOUBLE, MPI_STATUS_IGNORE);
    MPI_File_close(&fh);
}
```



With individual file pointers:

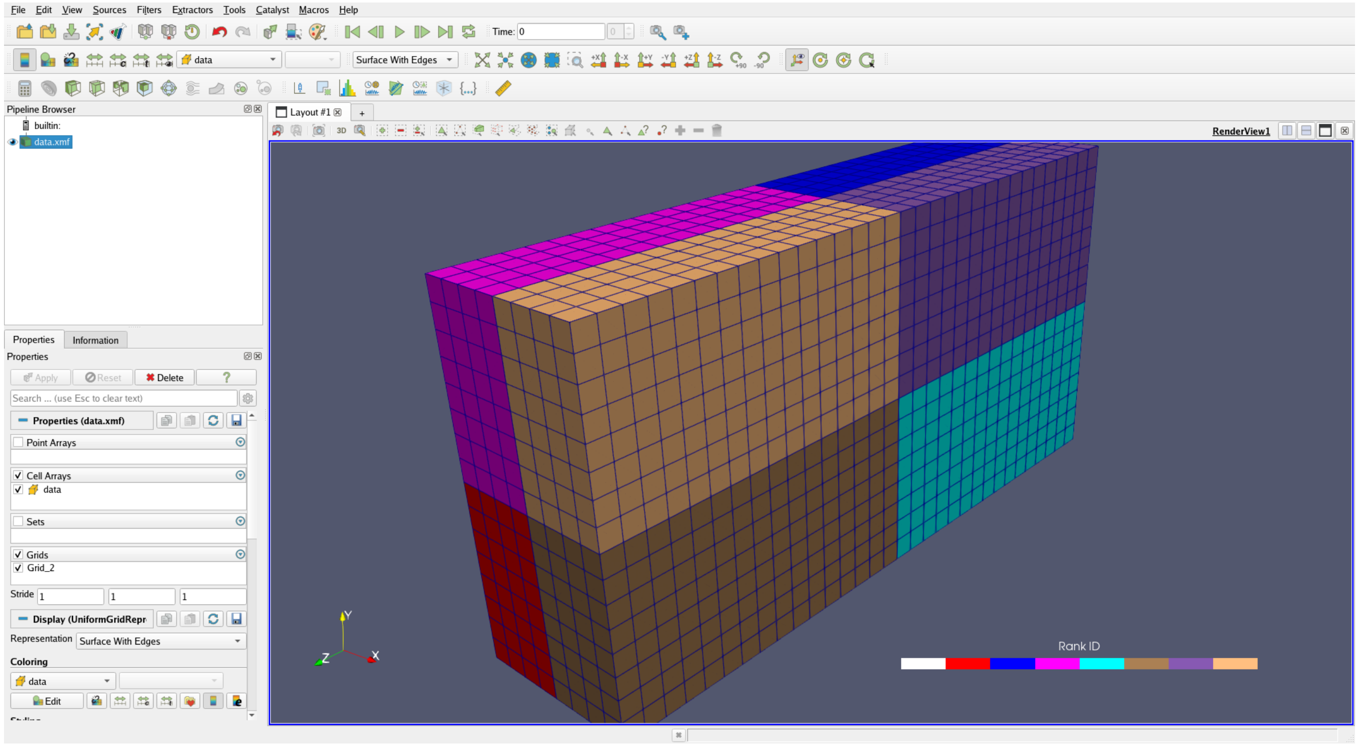
```
void dump_chunk(const std::vector<double> &v)
{
    MPI_File fh; // MPI file handle
    MPI_Offset bytes, file_offset = 0;
    bytes = v.size() * sizeof(double);
    MPI_Exscan(&bytes, &file_offset, 1, MPI_OFFSET, MPI_SUM,
               MPI_COMM_WORLD);
    MPI_File_open(MPI_COMM_WORLD, "out.bin",
                  MPI_MODE_CREATE | MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &fh);
    MPI_File_seek(fh, file_offset, MPI_SEEK_SET);
    MPI_File_write_all(fh, v.data(), v.size(), MPI_DOUBLE,
                       MPI_STATUS_IGNORE);
    MPI_File_close(&fh);
}
```

Rank	**0**	**1**	**2**
`v.size()` (type `double`)	100	50	40
`file_offset (MPI_BYTE)`	0	800	1200

Exclusive scan example for 3 ranks. See figure on the left.

PARALLEL I/O WITH HDF5 AND MPI

- The Hierarchical Data Format (HDF) is a widely used file format designed for large amounts of data.
- The parallel HDF5 library uses MPI and you can therefore integrate it easily in your applications.
- A python interface exists as well: https://www.h5py.org/ (example in lecture code)
- HDF5 data can easily be visualized with Open Source tools such as ParaView:



- See the `parallel_hdf5` code in the class `git` repository to generate the data.
- You can compile and run the code on a compute node or submit the job with `sbatch submit.sh`.
- You can load the `data.xmf` file with ParaView.

WHAT ELSE IS THERE IN MPI?

- We did not cover the full MPI spectrum. MPI topics that are relevant as well include:
  - MPI data types (see Chapter 5 in the MPI 4.0 standard)
  - MPI communicator management (see Chapter 7 in the MPI 4.0 standard)
  - MPI process topologies (see Chapter 8 in the MPI 4.0 standard)
  - Process creation and management (see Chapter 11 in the MPI 4.0 standard)
  - One-sided MPI communication (see Chapter 12 in the MPI 4.0 standard)
- User defined MPI data types are useful to simplify the selection of message data, e.g., subdomain boundary cells in structured grids.
- MPI process topologies are very useful to simplify the determination of source/destination relationships when sending messages on a lattice.

RECAP

- MPI I/O allows for scalable and high performance file operations.
- MPI file views provide a very flexible mechanism for complex data patterns.
- Collective MPI I/O procedures offer more room for optimizations and generally perform better than the non-collective counterparts.
- You can overlap non-blocking I/O calls with computation as well.
- *High I/O volume can quickly become the main bottleneck in your application. Consideration of data compression techniques are often considered in this situation!*

Further reading:

- Chapter 14 in *MPI 4.0 Standard*, MPI Forum, 2021 (pdf in `git` repo as well)
- Section 10 (MPI I/O) in Eijkhout, "*Parallel Programming for Science and Engineering*", free pdf
- Sections 1, 2, 3, 4, 5, 7, 15 (additional MPI reading) in Eijkhout, "*Parallel Programming for Science and Engineering*", free pdf