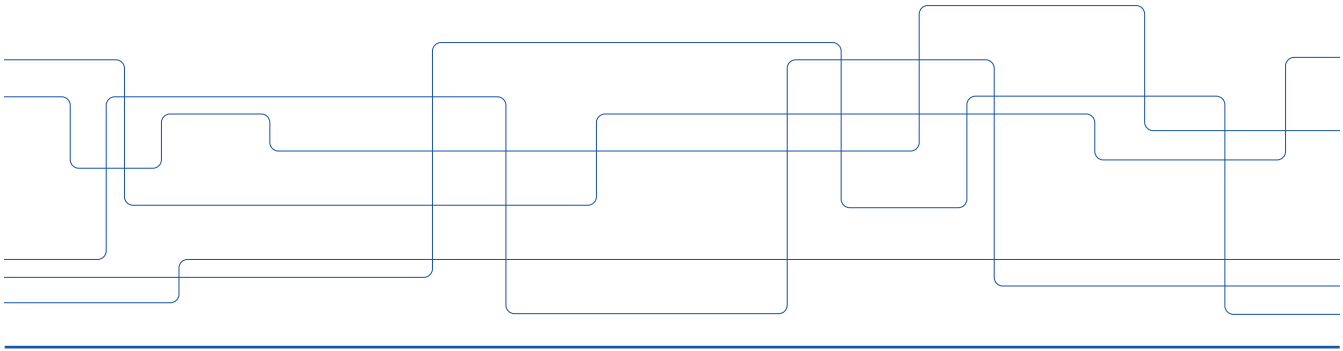
KTH ROYAL INSTITUTE
OF TECHNOLOGY

Introduction to MPI I/O

Niclas Jansson
PDC Center for High Performance Computing



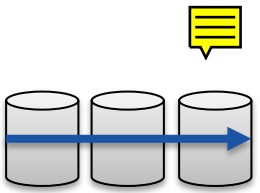
“A supercomputer is a device for converting a CPU-bound problem into an I/O bound problem”

Ken Batcher

Introduction

- Performance of magnetic disks
  - Latency 2-10 ms (time it take the disk to spin under read/write/head)
  - 1,000x slower than internode communication
  - 10,000,000x slower than processors
  - Bandwidth over 100 MB/s
    - > *But only for large transfers!*
- Performance of SSD?
  - Single I/O controller!
- Improve performance with striping
  - Distribute work across controllers
  - File system still limited by network bandwidth (**single node file system**)

Single disk

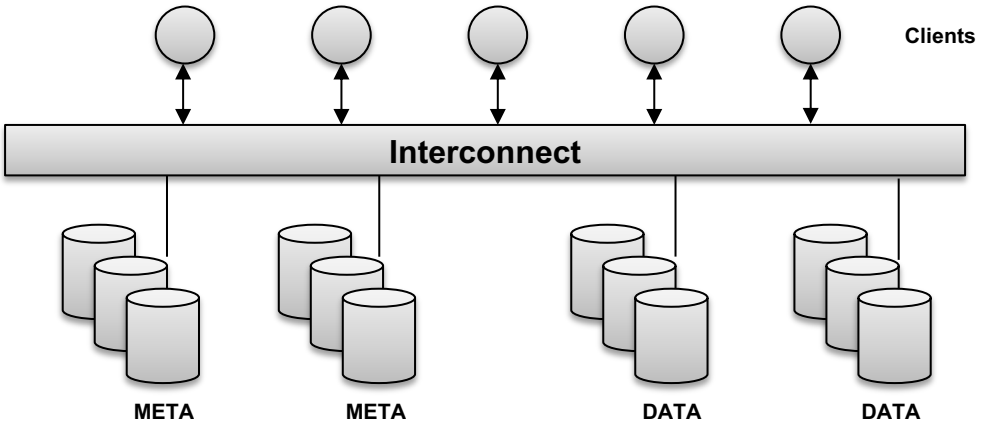


RAID

Parallel File Systems

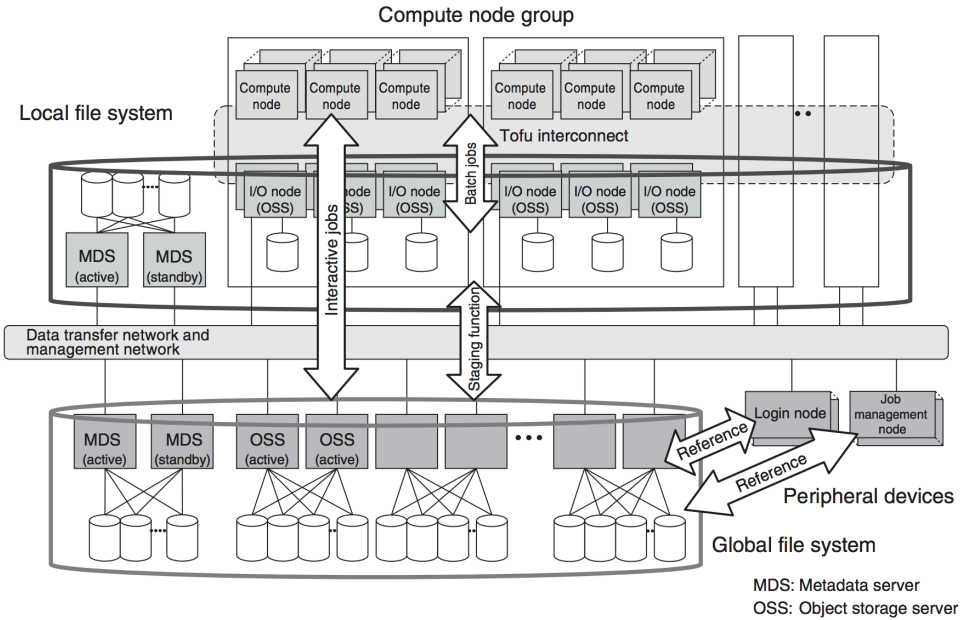
A parallel file system breaks up a dataset and distributes (or stripes) it across **multiple storage devices**, which can be located in **local and/or remote servers**

- Users don't know the physical location of the data blocks
- Parallel file systems often use a **metadata server** to store information about the data, such as filename, location owner etc



Parallel File Systems

Rather complex at scale, e.g. the 380 GB/s file system on K



K. Sakai et al.: *High Performance and Highly Reliable File System for the K computer*, FUJITSU Sci. Tech. J., Vol 48, No. 3, 2012

Great Parallel I/O Performance

IO-500

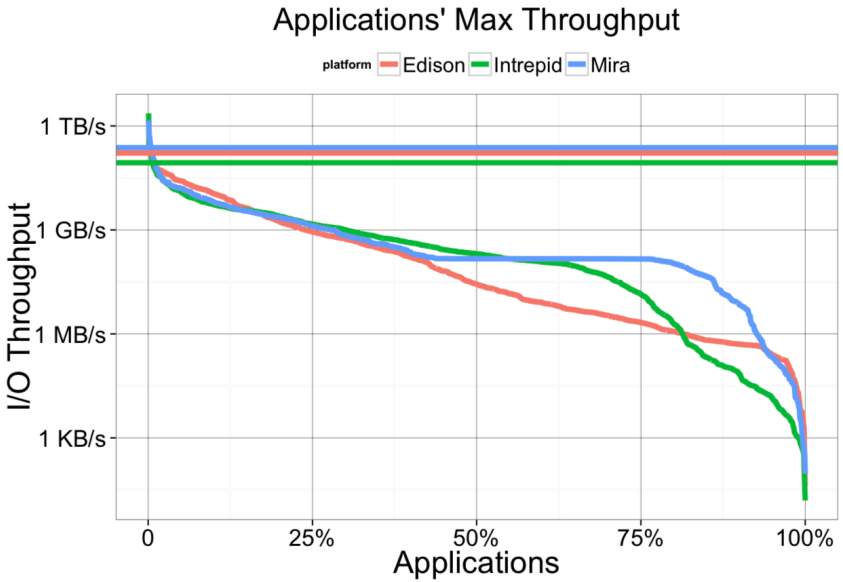
This is the official **ranked list**[1)] from ISC HPC 2019.

Please see also the 10 node challenge ranked list.

The list shows the best result for a given combination of system/institution/filesystem.

#	information							io500		
	institution	system	storage vendor	filesystem type	client nodes	client total procs	data	score	bw	md
									GiB/s	kIOP/s
1	University of Cambridge	Data Accelerator	Dell EMC	Lustre	512	8192	zip	620.69	162.05	2377.44
2	Oak Ridge National Laboratory	Summit	IBM	Spectrum Scale	504	1008	zip	330.56	88.20	1238.93
3	JCAHPC	Oakforest-PACS	DDN	IME	2048	2048	zip	275.65	492.06	154.41
4	Korea Institute of Science and Technology Information (KISTI)	NURION	DDN	IME	2048	4096	zip	156.91	554.23	44.43
5	CSIRO	bracewell	Dell/ThinkParQ	BeeGFS	26	260	zip	140.58	69.29	285.21
6	DDN	IME140	DDN	IME	17	272	zip	112.67	90.34	140.52
7	DDN Colorado	DDN IME140	DDN	IME	10	160	zip	109.42	75.79	157.96
8	DDN	AI400	DDN	Lustre	10	160	zip	104.34	19.65	553.98
9	KAUST	ShaheenII	Cray	DataWarp	1024	8192	zip	77.37	496.81	12.05
10	University of Cambridge	Data Accelerator	Dell EMC	BeeGFS	184	5888	zip	74.58	58.81	94.57
11	DDN Japan	AI400	DDN	Lustre	10	160	zip	74.10	12.22	449.28
12	HHMI Janelia Research Campus	Weka	WekaIO		10	3200	zip	66.43	27.74	159.12

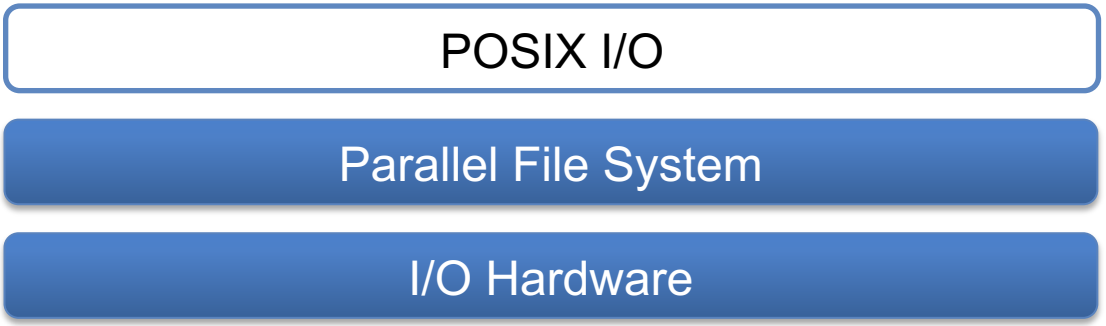
The Reality…



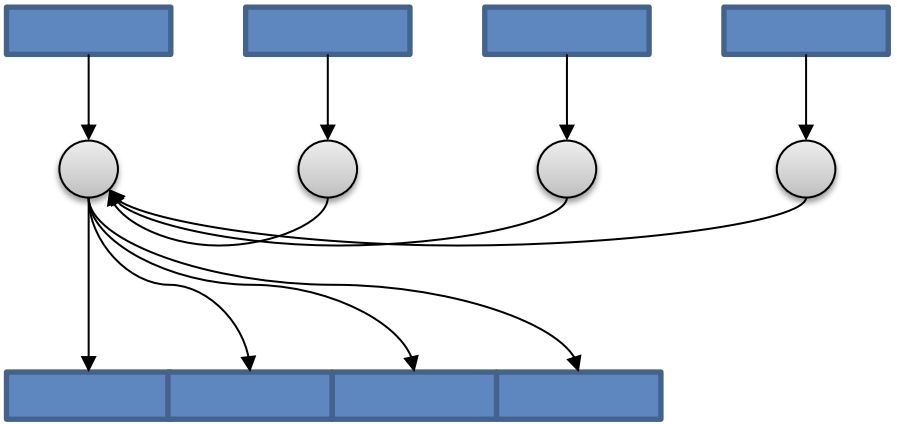
A Multiplatform Study of I/O Behavior on Petascale Supercomputers, Luu, Winslett, Gropp, Ross, Carns, Harms, Prabhat, Byna, Yao. HPDC'15

The Parallel I/O Software Stack

- POSIX defines a standard way for an application to obtain basic services from the operating system
- Used by almost all serial applications to perform I/O (created in the dark ages when a single computer owned its own file system)
  - No ability to describe collective I/O accesses
- Rather expensive schematics to guarantee on large clusters
  - After a write, **any read, from any other process, must see that write**

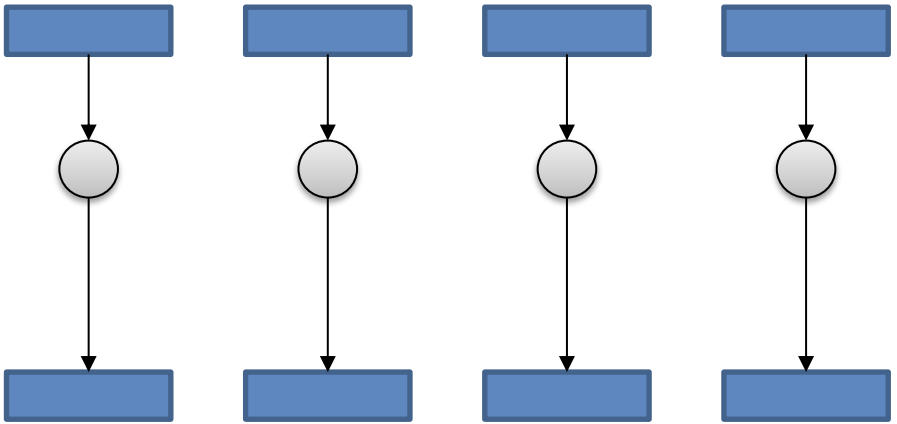


Serial I/O



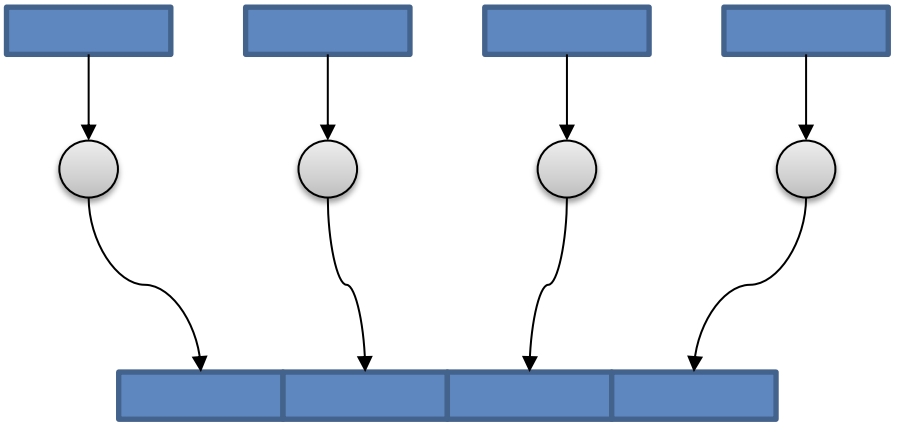
- One process performs I/O (serial)
  - Data aggregation (e.g. using MPI)
  - Single I/O process, single file
- Simple to implement, but **does not scale!**

Parallel I/O – One file per process



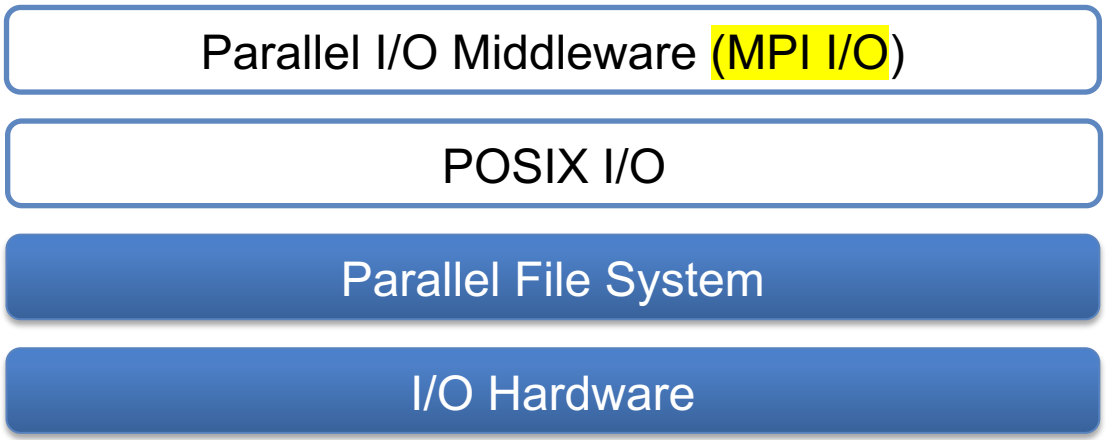
- All processes performs I/O
  - One file per process, how does it scale?
  - File system limitations
    - > *Large amount of concurrent file operations,* ***does not scale!***
    - > *Number of files per directory,* ***does not scale!***

Parallel I/O – Shared file



- All processes performs I/O
  - Single shared file (file system is happy)
  - Not easy to implement (**not supported directly by POSIX!**)
  - Performance highly depends on the implementation

The Parallel I/O Software Stack



MPI I/O

- Defined in the MPI standard since 2.0
  - Uses MPI datatypes to describe files
  - Uses send/receive like operations to read/write data
  - Common interface for all platform/languages
- Provides high-performance (parallel) I/O operations
  - **POSIX-like operations:** processes can open, close, seek, read and write files, as usual
  - **Non-Contiguous access:** Selective access to different parts of a file
  - **Collective I/O operations:** Optimised read/write by multiple processes into shared files
  - **Non-Blocking I/O:** Similar to non-blocking MPI send/receive, but for I/O operations
- **Large** and **complex** API, will only scratch the surface

Refer to the I/O chapter in the MPI standard:

https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node304.htm

The Parallel I/O Software Stack

Applications(CFD, Combustion, ...)

High-Level I/O Libraries (HDF5, NetCDF,…)

Parallel I/O Middleware (MPI I/O)

POSIX I/O

Parallel File System

I/O Hardware

High-Level I/O Libraries

- **HDF5 (Hierarchical Data Format)**
  - Hierarchical data organisation in a single file
  - Typed, multidimensional array storage
  - C,C++ and Fortran interfaces
  - Portable data format
- **Parallel NetCDF (Network Common Data Format)**
  - Collection of variables in a single file
  - Typed, multidimensional array variables
  - C and Fortran interfaces
  - Portable data format
- Most of these High-level libraries are built on top of MPI I/O
- These libraries are perfect **if an application's data layout is similar** to their data model

The Parallel I/O Software Stack

Applications(CFD, Combustion, ...)

Parallel I/O Middleware (MPI I/O)

POSIX I/O

Parallel File System

I/O Hardware

MPI I/O – File Manipulation

Opening/Closing files very similar to POSIX I/O

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

file handle

```
subroutine MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
CHARACTER*(*) FILENAME
INTEGER COMM, AMODE, INFO, FH, IERROR
subroutine MPI_FILE_CLOSE(FH, IERROR)
INTEGER FH, IERROR
```

- `COMM`: MPI communicator, File is open on all processes in the communicator
- `FILENAME`: Path to the file
- `AMODE`: File access mode (e.g. create, read-only, read/write etc)
- `INFO`: Hints for the MPI implementation
- `FH`: File handle, similar to a POSIX file pointer
- `IERROR`: Return value (Fortran specific)

MPI I/O – File Manipulation

File access mode:

- `MPI_MODE_RDONLY` – read only
- `MPI_MODE_RDWR` – reading and writing
- `MPI_MODE_WRONLY` – write only
- `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 end of file.

MPI Info Hints:

Complex hints for the implementation on how to e.g. organize the data. However, it's a *hint,* thus some implementations simply ignores them.

In most cases using `MPI_INFO_NULL` is sufficient.

For more information see:

https://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/node182.htm#Node182

MPI I/O

The MPI interface supports two types of I/O modes

Independent I/O

- Basic interface
- Similar to POSIX I/O, but with support for derived data types

Collective I/O

- I/O by all processes
- High-Performance API
- Vendor optimised strategies

MPI I/O

The MPI interface supports two types of I/O modes

Independent I/O

- Basic interface
- Similar to POSIX I/O, but with support for derived data types

Collective I/O

- I/O by all processes
- High-Performance API
- Vendor optimised strategies

Independent I/O

- **Read/Write/Seek** operations very similar to normal Point-to-point communication in MPI

```
int MPI_File_read(MPI_File fh, void *buf, int count,
                  MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write(MPI_File fh, void *buf, int count,
                   MPI_Datatype datatype, MPI_Status *status)
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
```

- `fh`: File handle
- `buf`: Buffer to read/write
- `count`: Number of elements of type `datatype` (MPI type)
- `status`: Status object, same as in `MPI_Recv`, e.g. for counting
- `offset`: File offset (bytes), from current position or from `whence`

- All **Read/Write** operations are blocking
- All operations **advances** the file pointer's position

Independent I/O

Assume you want to open a file on a specific rank, and write some information at a specific offset.

- Setting the MPI communicator to `MPI_COMM_SELF`, opens the file the file on the calling process. (How about `MPI_COMM_WORLD`?)

```
1  program foo
2    use mpi
3    integer :: ierr, fh, rank
4
5    call MPI_Init(ierr)
6    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
7
8    if (rank .eq. 0) then
9       call MPI_File_open(MPI_COMM_SELF, '/tmp/out.bin', &
10           MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, fh, ierr)
11
12      call MPI_File_close(fh, ierr)
13   end if
14
15   call MPI_Finalize(ierr)
16
17 end program foo
```

- File access/synchronisation is the programmer's responsibility!

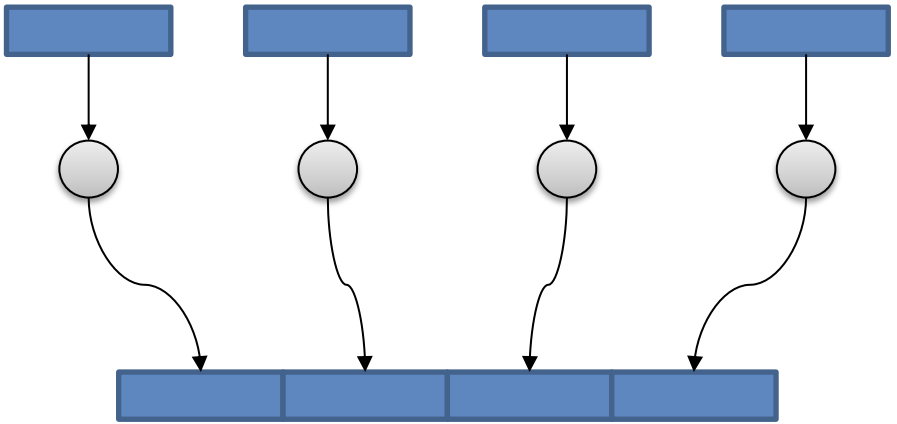
Independent I/O

```
1  program foo
2    use mpi
3    integer :: ierr, fh, rank
4    integer :: data(42)
5    integer (kind=MPI_OFFSET_KIND) :: offset
6    integer :: status(MPI_STATUS_SIZE)
7
8    call MPI_Init(ierr)
9    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
10
11   if (rank .eq. 0) then
12      call MPI_File_open(MPI_COMM_SELF, '/tmp/out.bin', &
13           MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, fh, ierr)
14
15      offset = 4711
16      call MPI_File_seek(fh, offset, MPI_SEEK_SET, ierr)
17
18      call MPI_File_write(fh, data, 42, MPI_INTEGER, status, ierr)
19
20      call MPI_File_close(fh, ierr)
21   end if
22
23   call MPI_Finalize(ierr)
24
25 end program foo
```

- Still serial I/O, how to parallelize it?

Independent I/O

Assume we want to write each data chunk into a shared file



- Possible in independent I/O mode (with proper data access)
  - Open the file in `MPI_COMM_WORLD`
  - Give each rank a different offset/section to write at

Independent I/O

- Explicit offsets **Read/Write**, low level routines with `_at` suffix, read/write data starting at a given 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, void *buf,
                      int count, MPI_Datatype datatype, MPI_Status *status)
```

- `fh`: File handle
- `buf`: Buffer to read/write
- `count`: Number of elements of type `datatype` (MPI type)
- `status`: Status object, same as in `MPI_Recv`, e.g. for counting
- `offset`: File offset (bytes), from the beginning of the file

- Seek+I/O in one call, **does not advance** the file pointer's position
- Straightforward implementation, but tedious (and possibly error prone) bookkeeping of file offsets
- Difficult to handle non-contiguous data
  - Requires multiple read/writes with different offsets

Independent I/O

Example: output four times the MPI rank into a shared file, ordered and in parallel:

MPI_COMM_WORLD

MPI_File_write_at

Which offset?

```
program foo
   use mpi
   integer :: ierr, fh, rank
   integer :: data(42)
   integer (kind=MPI_OFFSET_KIND) :: offset
   integer :: status(MPI_STATUS_SIZE)

   call MPI_Init(ierr)
   call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

   if (rank .eq. 0) then
      call MPI_File_open(MPI_COMM_SELF, '/tmp/out.bin', &
           MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, fh, ierr)

      offset = 4711
      call MPI_File_seek(fh, offset, MPI_SEEK_SET, ierr)

      call MPI_File_write(fh, data, 42, MPI_INTEGER, status, ierr)

      call MPI_File_close(fh, ierr)
   end if

   call MPI_Finalize(ierr)

end program foo
```

Independent I/O

Example: output four times the MPI rank into a shared file, ordered and in parallel:

```
[>od -D out.bin
0000000                0              0              0              0
0000020                1              1              1              1
0000040                2              2              2              2
0000060                3              3              3              3
0000100
```

```
1  program foo
2    use mpi
3    integer :: ierr, fh, rank, isize
4    integer :: data(4)
5    integer (kind=MPI_OFFSET_KIND) :: offset
6    integer :: status(MPI_STATUS_SIZE)
7
8    call MPI_Init(ierr)
9    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
10   call MPI_Type_size(MPI_INTEGER, isize, ierr)
11
12   call MPI_File_open(MPI_COMM_WORLD, '/tmp/out.bin', &
13        MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, fh, ierr)
14
15   data = rank
16   offset = rank * 4 * isize
17   call MPI_File_write_at(fh, offset, data, 4, MPI_INTEGER, status, ierr)
18
19   call MPI_File_close(fh, ierr)
20
21   call MPI_Finalize(ierr)
22
23 end program foo
```

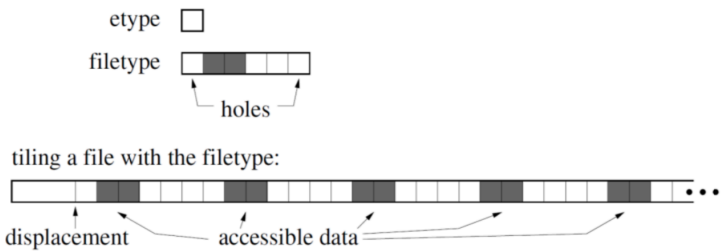
Independent I/O

- Use MPI File Views to tell each process **which** part of the file to **Read/Write** into

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
                      MPI_Datatype filetype, char *datarep, MPI_Info info)
```

- `fh`: File handle
- `disp`: Displacement (in bytes) from the start of the file
- `etype`: MPI datatyxpe
- `filetype`: Specifies which portion of the file is visible to the process
- `datarep`: Data representation
- `info`: Hints for the implementationMPI

- Creative `filetype`'s can be used to write non-contiguous data



Independent I/O

```
1  program foo
2    use mpi
3    integer :: ierr, fh, rank, isize
4    integer :: data(4)
5    integer (kind=MPI_OFFSET_KIND) :: disp
6    integer :: status(MPI_STATUS_SIZE)
7
8    call MPI_Init(ierr)
9    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
10   call MPI_Type_size(MPI_INTEGER, isize, ierr)
11
12   call MPI_File_open(MPI_COMM_WORLD, '/tmp/out.bin', &
13        MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, fh, ierr)
14
15   data = rank
16   disp = rank * 4 * isize
17   call MPI_File_set_view(fh, disp, MPI_INTEGER, MPI_INTEGER, &
18        'native', MPI_INFO_NULL, ierr)
19   call MPI_File_write(fh, data, 4, MPI_INTEGER, status, ierr)
20
21   call MPI_File_close(fh, ierr)
22
23   call MPI_Finalize(ierr)
24
25 end program foo
```

- A naïve implementation is very similar to using explicit offsets
- Best use of file views is when one defines (creative) derived MPI datatypes (for `filetype`)

Derived datatypes

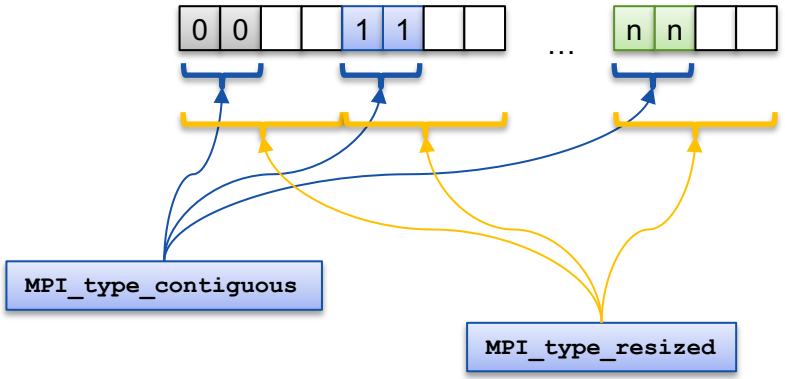
MPI provides methods for creating derived datatypes, that can be based on primitive types (e.g. `MPI_INTEGER`) or custom structures combining several different types. Some examples:

- **Contiguous:** Simple constructor that replicates a datatype into contiguous locations
- **Indexed:** Non-contiguous data layout with non equal displacements (user provided) between blocks
- **Vector:** Constructor that replicates a datatype into equally spaced blocks
- **Struct:** General type, equivalent to a structure in C
- **Resized:** Padded custom or primitive types

Note: once created these types can be used for any kind of MPI operations not only I/O, for example **send/recv** of complex data structure in a single MPI call (very handy indeed!)

Derived datatypes

Example: output the MPI rank twice, followed by two blanks into a shared file, ordered and in parallel:



Derived datatypes

To create a **contiguous** derived type, we need to define it, and commit it before it can be used by MPI

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
int MPI_Type_commit(MPI_Datatype *datatype)
```

- `count`: Number of instances of old datatype
- `oldtype`: Base datatype
- `newtype`: New, derived datatype

Derived datatypes

To create a **contiguous** derived type, we need to define it, and commit it before it can be used by MPI

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
int MPI_Type_commit(MPI_Datatype *datatype)
```

- `count`: Number of instances of old datatype
- `oldtype`: Base datatype
- `newtype`: New, derived datatype

To pad a derived type we need to create a new, **resized** type

```
int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,
                            MPI_Aint extent, MPI_Datatype *newtype)
```

- `oldtype`: Base datatype
- `lb`: lower bound of new type
- `extent`: Extent of new datatype
- `newtype`: New datatype with an upper bound of `lb + extent`

Independent I/O

- Recall the file pattern:

Contiguous type, hold two int

Pad it with two blanks

Only write the contiguous type

```
program foo
  use mpi
  integer :: ierr, fh, rank, isize
  integer :: data(2), CONT_TYPE, MY_TYPE
  integer (kind=MPI_OFFSET_KIND) :: disp
  integer (kind=MPI_ADDRESS_KIND) :: extent, lb
  integer :: status(MPI_STATUS_SIZE)

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

  call MPI_Type_size(MPI_INTEGER, isize, ierr)
  call MPI_Type_contiguous(2, MPI_INTEGER, CONT_TYPE, ierr)
  lb = 0
  extent = 4 * isize
  call MPI_Type_create_resized(CONT_TYPE, lb, extent, MY_TYPE, ierr)
  call MPI_Type_commit(CONT_TYPE, ierr)
  call MPI_Type_commit(MY_TYPE, ierr)

  call MPI_File_open(MPI_COMM_WORLD, '/tmp/out.bin', &
       MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, fh, ierr)

  data = rank + 1
  disp = rank * 4 * isize
  call MPI_File_set_view(fh, disp, MPI_INTEGER, MY_TYPE, &
       'native', MPI_INFO_NULL, ierr)
  call MPI_File_write(fh, data, 1, CONT_TYPE, status, ierr)

  call MPI_File_close(fh, ierr)

  call MPI_Finalize(ierr)

end program foo
```

MPI I/O

The MPI interface supports two types of I/O modes

Independent I/O

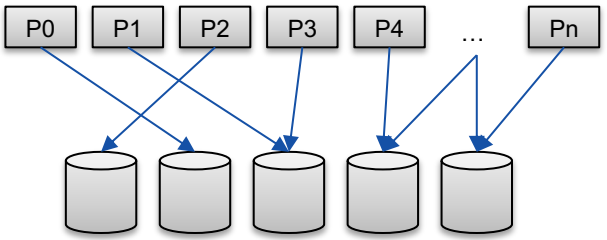
- Basic interface
- Similar to POSIX I/O, but with support for derived data types

Collective I/O

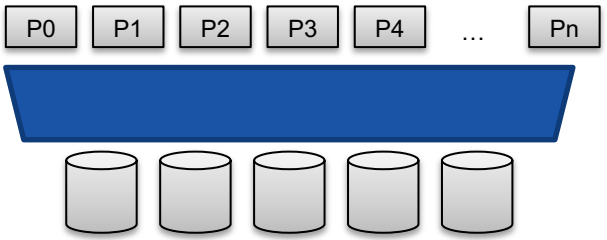
- I/O by all processes
- High-Performance API
- Vendor optimised strategies

Collective I/O

- **Independent I/O**

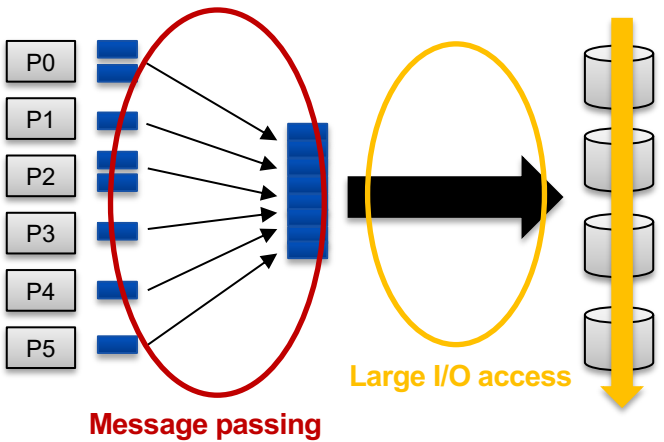


- **Collective I/O**



Collective I/O

- A critical optimisation in parallel I/O
- All ranks, associated with a file handle (communicator) must call the collective I/O routine
  - One I/O call instead of many smaller ones
  - Allows for internal optimisation within the MPI runtime
  - **I/O requests from different ranks may be merged together**



Collective I/O

- **Read/Write** operations more or less similar to independent I/O, except that all ranks associated with a file must call them

```
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, void *buf, int count,
                       MPI_Datatype datatype, MPI_Status *status)
```

- All collective functions ends with `_all`, otherwise same arguments as before
- Similarly for the collective **explicit offsets Read/Write**

```
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, void *buf,
                          int count, MPI_Datatype datatype, MPI_Status *status)
```

- Note `offset` can be **different** on each rank, despite being a collective call
- **File views** works exactly the same as before

Additional topics

MPI I/O topics we haven't covered in this lecture

- Non-blocking read/write, `MPI_File_iread/MPI_File_iwrite`
  - Works similar to non-blocking send/receive
  - Synchronised with `MPI_Wait`
- Shared file pointers
  - MPI maintains exactly **one shared** file pointer per collective file handle (shared in the communicator group)
  - MPI I/O calls ends with `_shared`
  - File views/offsets must be the **same** on all ranks
  - Allows for ordered (per rank) reads/writes using functions ending with `_ordered`
- Various house keeping routines e.g. **delete/query/allocate** etc.

Refer to the I/O chapter in the MPI standard:

https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node304.htm

Exercises

- Read a file containing *n* entries each with a double and an integer

- MPI derived type **struct**

```
int MPI_Type_create_struct(int count, int array_of_blocklengths[],
                           MPI_Aint array_of_displacements[],
                           MPI_Datatype array_of_types[], MPI_Datatype *newtype)
```

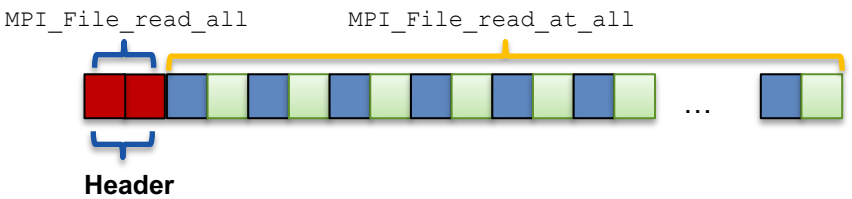
- `count`: number of struct entries
- `array_of_blocklengths`: number of elements for each struct entry
- `array_of_displacements`: Byte displacement of each struct entry (start of)
- `array_of_types`: MPI type of each struct entry (can be derived ones!)
- `newtype`: New datatype describing the struct

Exercises

```c
#include <mpi.h>

typedef struct {
  double d;
  int i;
} foo_t;

int main(int argc, char *argv[]) {
  int i;
  const int len[2] = {1, 1};
  const MPI_Datatype type[2] = { MPI_DOUBLE, MPI_INT };

  MPI_Datatype STRUCT_TYPE;
  MPI_Aint base, disp[2];
  foo_t foo;

  MPI_Init(&argc, &argv);

  MPI_Get_address(&foo.d, disp);
  MPI_Get_address(&foo.i, disp+1);
  base = disp[0];
  for (i = 0; i < 2; i++) MPI_Aint_diff(disp[i], base);

  MPI_Type_create_struct(2, len, disp, type, &STRUCT_TYPE);
  MPI_Type_commit(&STRUCT_TYPE);

  MPI_Finalize();
}
```

Exercises

- For performance reasons we want each of the MPI rank to read a large contiguous chunk of data
- We can for example use a simple load balanced linear distribution to compute *n_local = (n + size – rank – 1) / size*
- Given *n_local*, we need to determine at which offset a rank should start reading entries
  - Perform a partial reduction to compute the offset
    - > `MPI_Scan` *or even better* `MPI_Exscan` *(w/o local contrib.)*
- And finally compute a byte offset for explicit offsets (or file views),
  - Followed by a collective `MPI_File_read_at_all`
- What if the file contains a header?



Exercises

- Certain compilers has the ”bad” habit of padding structures

```
typedef struct {
  double d;
  int i;
} foo_t;
```

```
sizeof(foo_t) = 16
```

```
typedef struct {
  double d;
  short i;
} bar_t;
```

```
sizeof(bar_t) = 16
```

- Compiler specific attributes can fix this

```
typedef struct {
  double d;
  short i;
}__attribute__((packed)) babar_t;
```

```
sizeof(babar_t) = 10
```

- However, even if a derived type is created based on a packed struct the MPI struct might (**most likely will**!) be padded
- To solve this, resize the derived type using `MPI_Type_create_resized`

Derived datatypes

To create a **contiguous** derived type, we need to define it, and commit it before it can be used by MPI

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
int MPI_Type_commit(MPI_Datatype *datatype)
```

- `count`: Number of instances of old datatype
- `oldtype`: Base datatype
- `newtype`: New, derived datatype

To pad a derived type we need to create a new, **resized** type

```
int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,
                            MPI_Aint extent, MPI_Datatype *newtype)
```

- `oldtype`: Base datatype
- `lb`: lower bound of new type
- `extent`: Extent of new datatype
- `newtype`: New datatype with an upper bound of `lb + extent`

Exercises

```c
#include <mpi.h>

typedef struct {
  double d;
  short i;
} __attribute__((packed)) foo_t;

int main(int argc, char *argv[]) {
  int i;
  const int len[2] = {1, 1};
  const MPI_Datatype type[2] = { MPI_DOUBLE, MPI_INT };

  MPI_Datatype STRUCT_TYPE, PACKED_TYPE;
  MPI_Aint base, disp[2], sizeofstruct;
  foo_t foo;

  MPI_Init(&argc, &argv);

  MPI_Get_address(&foo.d, disp);
  MPI_Get_address(&foo.i, disp+1);
  base = disp[0];
  for (i = 0; i < 2; i++) MPI_Aint_diff(disp[i], base);

  MPI_Type_create_struct(2, len, disp, type, &STRUCT_TYPE);
  MPI_Get_address(&foo+1, &sizeofstruct);
  sizeofstruct = MPI_Aint_diff(sizeofstruct, base);
  MPI_Type_create_resized(STRUCT_TYPE, 0, sizeofstruct, &PACKED_TYPE);
  MPI_Type_commit(&PACKED_TYPE);

  MPI_Finalize();
}
```