Advanced Message-Passing Programming

Miscellaneous MPI-IO topics

epcc

THE UNIVERSITY OF EDINBURGH

ARCHER Training Courses

Sponsors

Reusing this material

MPI-IO Errors

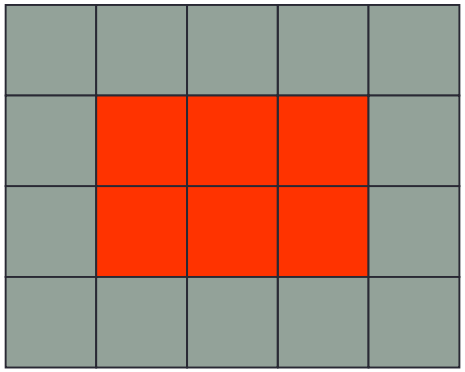
- Unlike the rest of MPI, MPI-IO errors are not fatal
  - probably don't want your program to crash if a file open fails
  - always need to check the error code!
- Many different error codes can be reported
  - I would suggest simply quitting if **`ierr != MPI_SUCCESS`**
- Can change this behaviour for file operations
  - same functionality as **`MPI_Errhandler_create`** etc.
  - called **`MPI_File_create_errhandler`**, ...
  - error handlers are attached to file handles rather than communicators
  - can set handler to be **`MPI_ERRORS_ARE_FATAL`**

Size of File on Disk

- Useful to check length of output file
  - `ls -l <filename>`
  - check that size (in bytes) is what you expect
- Can be confusing if file already exists
  - length will be increased if new file is longer than existing file
  - but may not be decreased if new file is shorter!
- Delete old files before running your test programs

Datatype for `MPI_File_read` / `write`

- Usually pass the basic type of the array being processed
  - eg **`MPI_FLOAT`**, **`MPI_REAL`**
- Can pass derived types
  - useful for receiving the core of an array when local arrays have halos



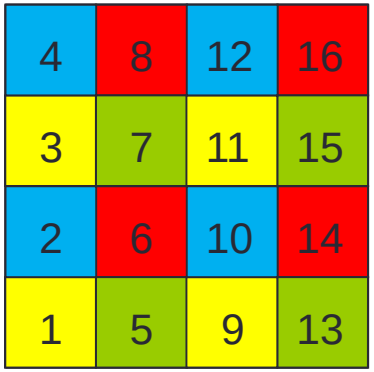
```
MPI_File_read_all(fh, &x[1][1], 1, vector3x2, ...);

MPI_FILE_READ_ALL(fh, x(2,2)  , 1, vector3x2, ...)
```

  – or could use a 3x2 subarray and pass **`&x[0][0]`** or **`x(1,1)`**

General Decompositions

- We have just considered block decompositions
  - where local array size is an exact multiple of global array size
- If the sizes don't match
  - define different sized subarrays on each process
  - eg processes at the edge of the grid have smaller subsections
- This does not generalize to block-cyclic decompositions
  - how do we specify discontinuous subarrays?



Distributed Arrays

```
int MPI_Type_create_darray(int size, int rank,
  int ndims, int array_of_gsizes[],
  int array_of_distribs[], int array_of_dargs[],
  int array_of_psizes[], int order,
  MPI_Datatype oldtype, MPI_Datatype *newtype);

MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS
  ARRAY_OF_GSIZES, ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS,
  ARRAY_OF_PSIZES, ORDER, OLDTYPE, NEWTYPE, IERR)

INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*),
  ARRAY_OF_DISTRIBS(*), ARRAY_OF_DARGS(*),
  ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERR
```

- See the man page for full details!
  - uses HPF conventions for block-cyclic distributions

Unstructured Data

- Imagine a particle simulation
  - each particle is a compound object with a type and position (x,y,z)
    - eg a C struct or Fortran type
  - each particle has unique global identifier 1, 2, 3, ..., N-1, N
- Particles move around
  - at the end of a simulation, each process will have:
    - a different number of particles
    - with a random mixture of global identifiers
- Two choices
  - write to file in the order they appear in the processes
  - write to file with position based on global identifier

Approach

- Define a derived type to match the particle object
  - eg **`MPI_PARTICLE`**
  - use this as the etype
- Writing in process order
  - need to know where to start in the file
  - calculate the sum of the number of particles on previous ranks
    - using **`MPI_Scan`**
- Writing in global order
  - call **`MPI_Type_indexed`** (or **`create_indexed_block`**)
  - use this as the filetype
  - write multiple instances of **`MPI_PARTICLE`**

Unstructured Meshes

- Similar to global ordering of particles
  - each element has both a local and global identifier
  - want the file to be ordered by the global id

- Define an **`MPI_ELEMENT`**
  - use this as the etype
  - create an indexed filetype based on global id

Blocking IO

- This code spends a lot of time waiting while saving to disk

```
define big arrays: old and new
loop many times
  ! do a computationally expensive operation
  new = expensive_function(old)
  old = new
  every 10 iterations:
    save_to_disk(old)
 end loop
```

Non-blocking IO

- This code overlaps computation and IO

```
define big arrays: old and new
loop many times
  ! do a computationally expensive operation
  new = expensive_function(old)
  if (saving to disk):
    finish: isave_to_disk(old)
  old = new
  every 10 iterations:
    start: isave_to_disk(old)
 end loop
```

Non-blocking IO in MPI-IO

- Two forms
- General non-blocking
  - `MPI_File_iwrite(fh, buf, count, datatype, request)`
  - finish by waiting on request
  - but no collective version

- Split collective
  - `MPI_File_write_all_begin(fh, buf, count, datatype)`
  - `MPI_File_write_all_end(fh, buf, status)`

  - only a single outstanding IO operation at any one time
  - allows for collective version

Serial IO

- How can I read MPI-IO files in a serial program?
- Using native format
  - data is raw bytes
  - use **`fread`** in C or direct access unformatted IO in Fortran
  - see **`ioread.c`** and **`ioread.f90`** for examples
  - Fortran approach is quite old-fashioned (direct access IO)
    - new `access="stream"` functionality makes this a bit simpler
- Other MPI-IO formats will require more work!
- Note that you can do single process IO in MPI-IO
  - pass **`MPI_COMM_SELF`** to **`MPI_File_open`**

Other MPI-IO read / write calls

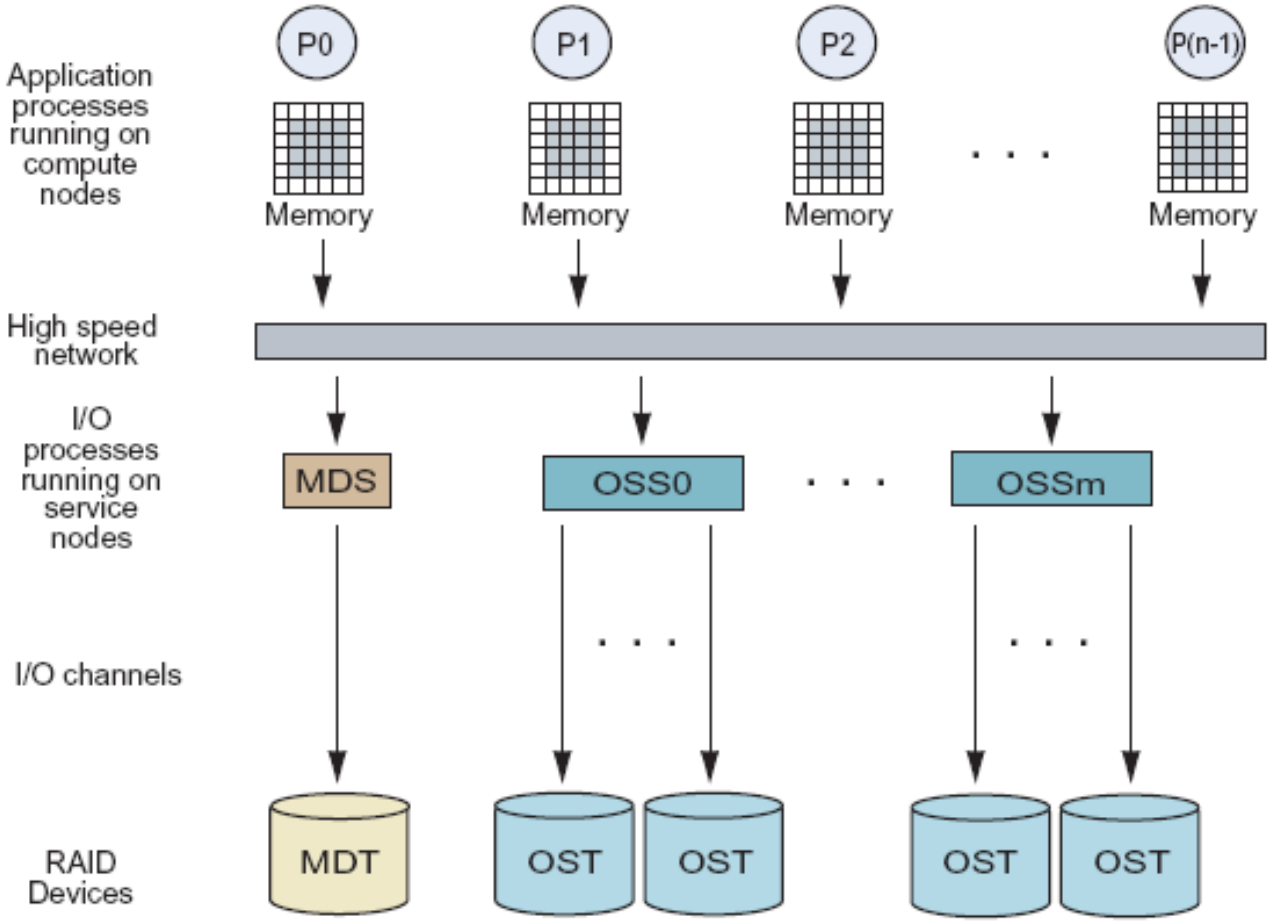
- I have advised
  - define a datatype to represents mapping from local to global data
  - use this in **`MPI_File_set_view()`**
  - then do linear reads / writes; gaps are automatically skipped

- Alternative approach
  - let everyone see the whole file (i.e. do not set a view)
  - manually seek to correct location using, e.g., **`MPI_File_write_at()`**
  - displacement is in units of the extent of the **`etype`**

- Disadvantages
  - a very low-level, manual approach less amenable to IO optimisation
  - danger that each request is handled individually with no aggregation
  - can use **`MPI_File_write_at_all()`** but might still be slow

Performance

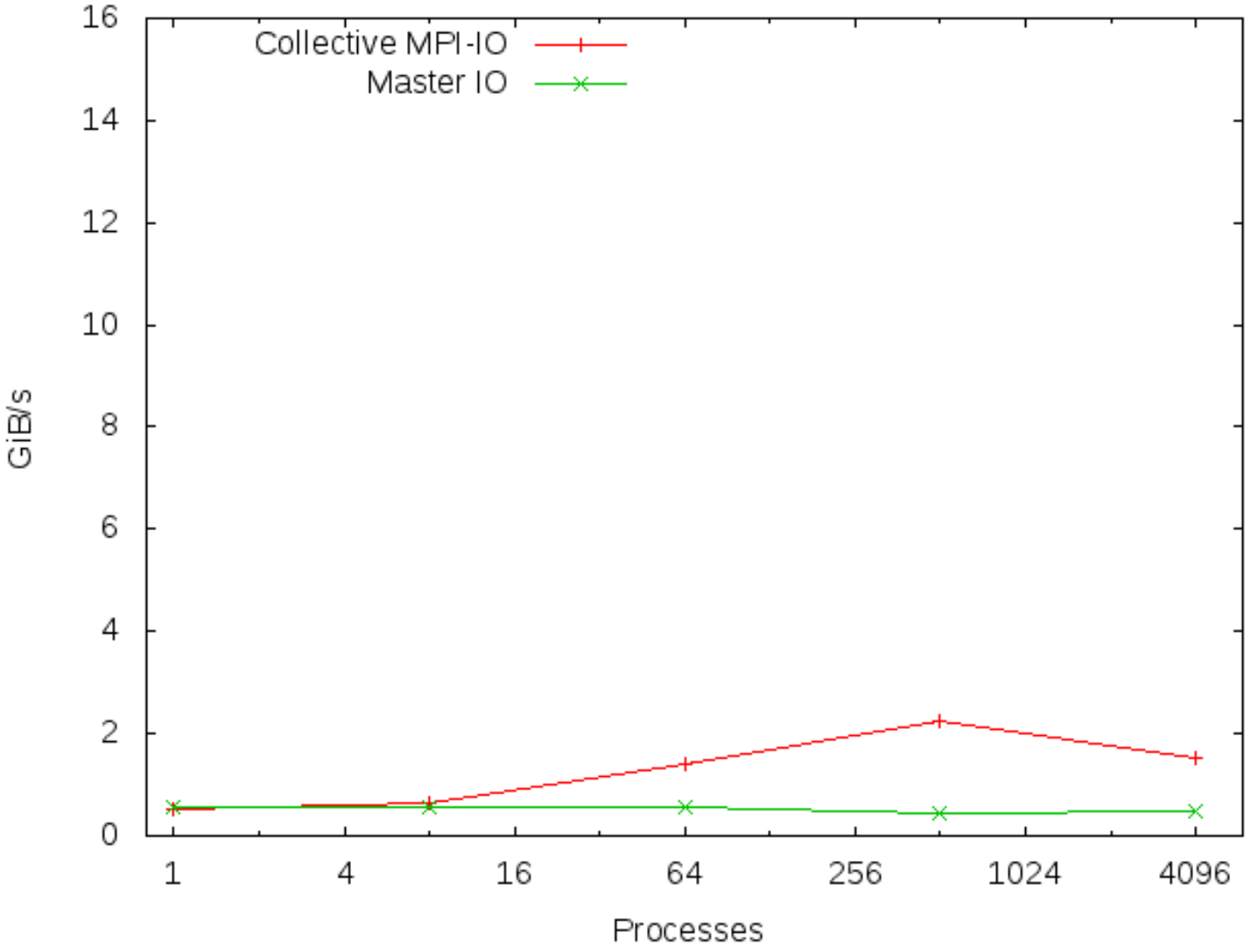
- Recall schematic overview of parallel file system Lustre



Application-side parallel IO

- Implementing MPI-IO has achieved
  - all data going to a single file
  - minimal stress on Meta Data Server (MDS) – a serial bottleneck
  - potential for many processes to write simultaneously
- But …
  - performance requires multiple parallel writes to disk
  - in Lustre, requires multiple Object Storage Servers (OSS) writing to multiple Object Storage Targets (OST)
  - an OSS is like an IO server, an OST is like a physical disk
- User has control over assignment of files to OSTs
  - but default is only a single OST (previously 4 OSTs)
  - MPI-IO performance not much better than naïve master IO

Parallel vs serial IO, default Lustre (4 stripes)



Cellular Automaton Model

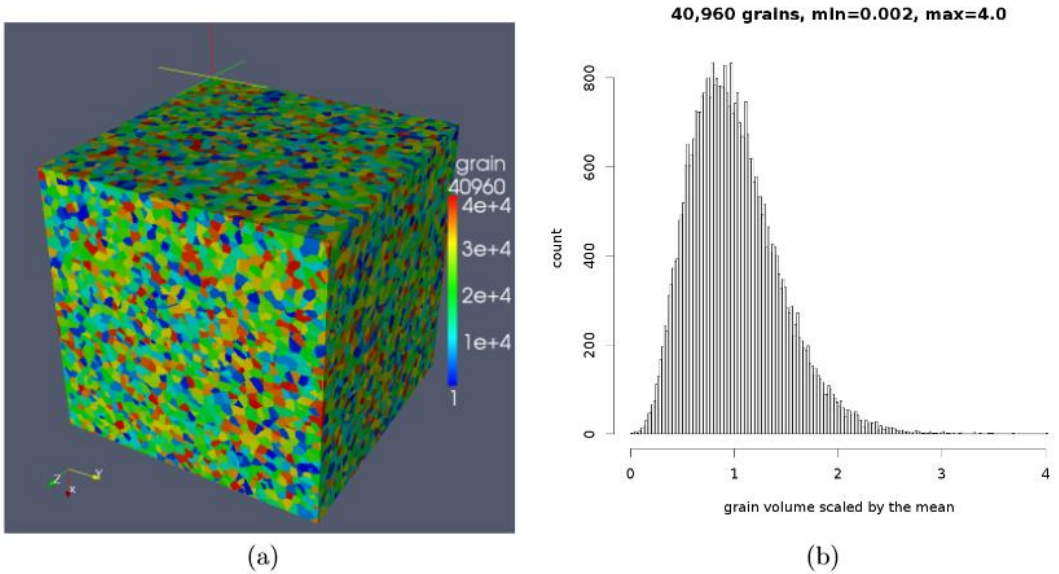


Figure 1: A 4.1×10^9 cell, 40,960 grain equiaxed microstructure model, showing (a) grain arrangement with colour denoting orientation; (b) grain size size (volume) histogram.

- *Fortran coarray library for 3D cellular automata microstructure simulation*, Anton Shterenlikht, proceedings of 7th International Conference on PGAS Programming Models, 3-4 October 2013, Edinburgh, UK.

epcc

THE UNIVERSITY OF EDINBURGH

Benchmark

- Distributed regular 3D dataset across 3D process grid
  - local data has halos of depth 1; set up for weak scaling
  - implemented in Fortran and MPI-IO

```
! Define datatype describing global location of local data
call MPI_Type_create_subarray(ndim, arraygsize, arraysubsize, arraystart,
       MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION, filetype, ierr)

! Define datatype describing where local data sits in local array
call MPI_Type_create_subarray(ndim, arraysize, arraysubsize, arraystart,
       MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION, mpi_subarray, ierr)

! After opening file fh, define what portions of file this process owns
call MPI_File_set_view(fh, disp, MPI_DOUBLE_PRECISION, filetype,
                       'native', MPI_INFO_NULL, ierr)
! Write data collectively
call MPI_File_write_all(fh, iodata, 1, mpi_subarray, status, ierr)
```

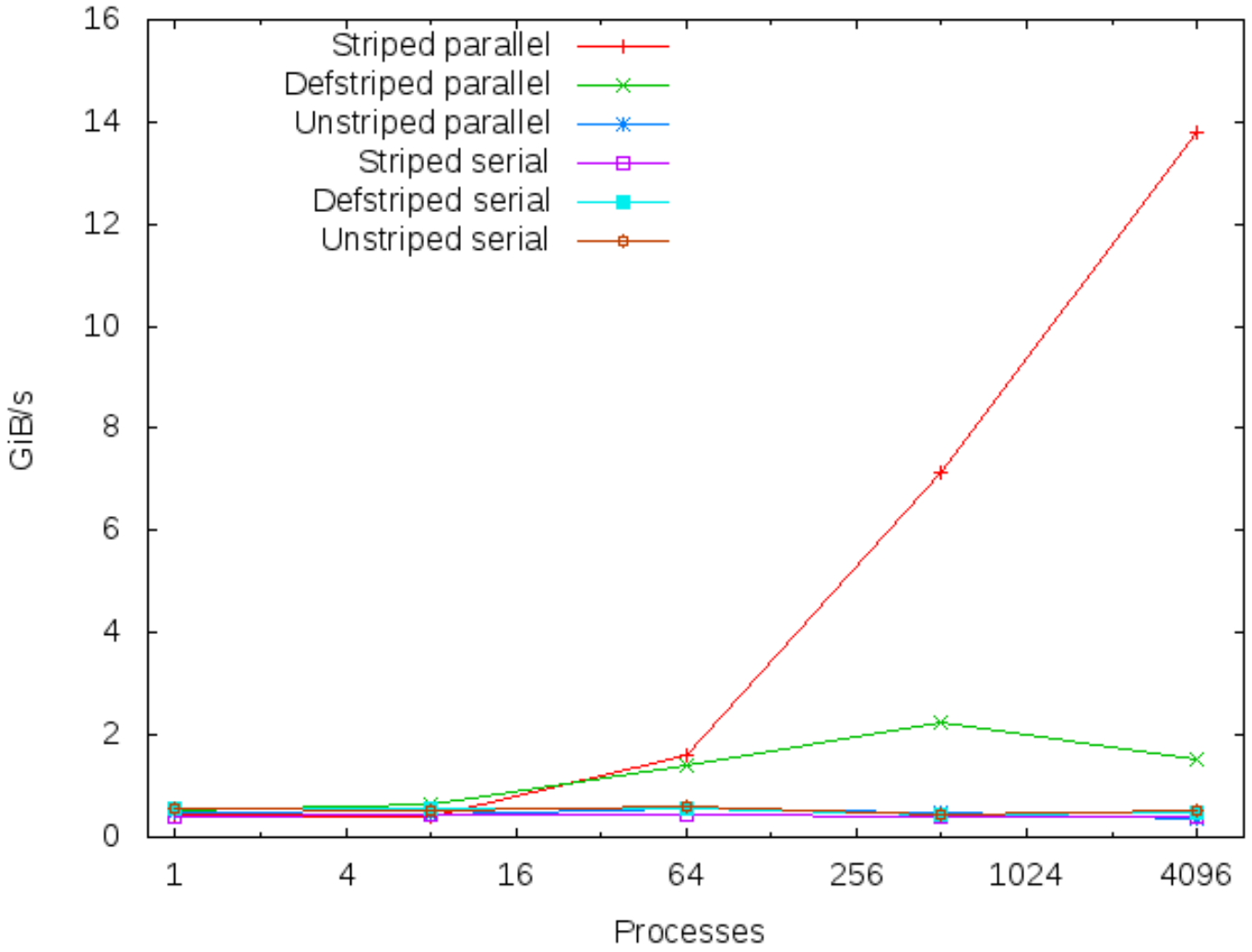
epcc

THE UNIVERSITY OF EDINBURGH

Lustre Striping

- Can split a file across multiple OSTs
  - each block is called a “stripe”; default striping is across 4 OSTs
- **`lfs setstripe -c 8 <directory>`**
  - stripes across 8 OSTs for all files in the directory
  - has substantial benefits for performance
  - stripe count of “-1” means use *all* OSTs
- Test case
  - 128 x 128 x 128 array of doubles on each process in 3D grid
  - scaled up to 4096 processes = 64 GiB
  - identical IO approach as used in exercise
    - generalised to 3D
    - local halos automatically stripped off with derived type in MPI-IO write call

Results on ARCHER



Performance Summary

- Serial IO never gets more than about 500 MiB/s
  - peak for a single OST
- With default striping, never exceed 2 GiB/s
  - 4 stripes = 4 OSTs = 4 x 500 MiB/s
- With full striping, IO bandwith increases with process count
  - can achieve in excess of 10 GiB/s

- Collective IO is essential
  - replacing **`MPI_File_Write_all()`** by **`MPI_File_write()`** disastrous!
  - identical functionality but each IO request now processed separately with file locking

Processes	Bandwidth
1	49.5 MiB/s
8	5.9 MiB/s
64	2.4 MiB/s

Documentation

- MPI web pages
- Short ARCHER report:
  - **`http://www.archer.ac.uk/documentation/white-papers/`**
- Another tutorial
  - **`https://www.lrde.epita.fr/~ricou/mpi-io.ppt`**
- Advanced MPI book
  - "Using Advanced MPI: Modern Features of the Message-Passing Interface"