Introduction to I/O at CHPC

Martin Čuma, m.cuma@utah.edu

Center for High Performance Computing

Fall 2020

Outline

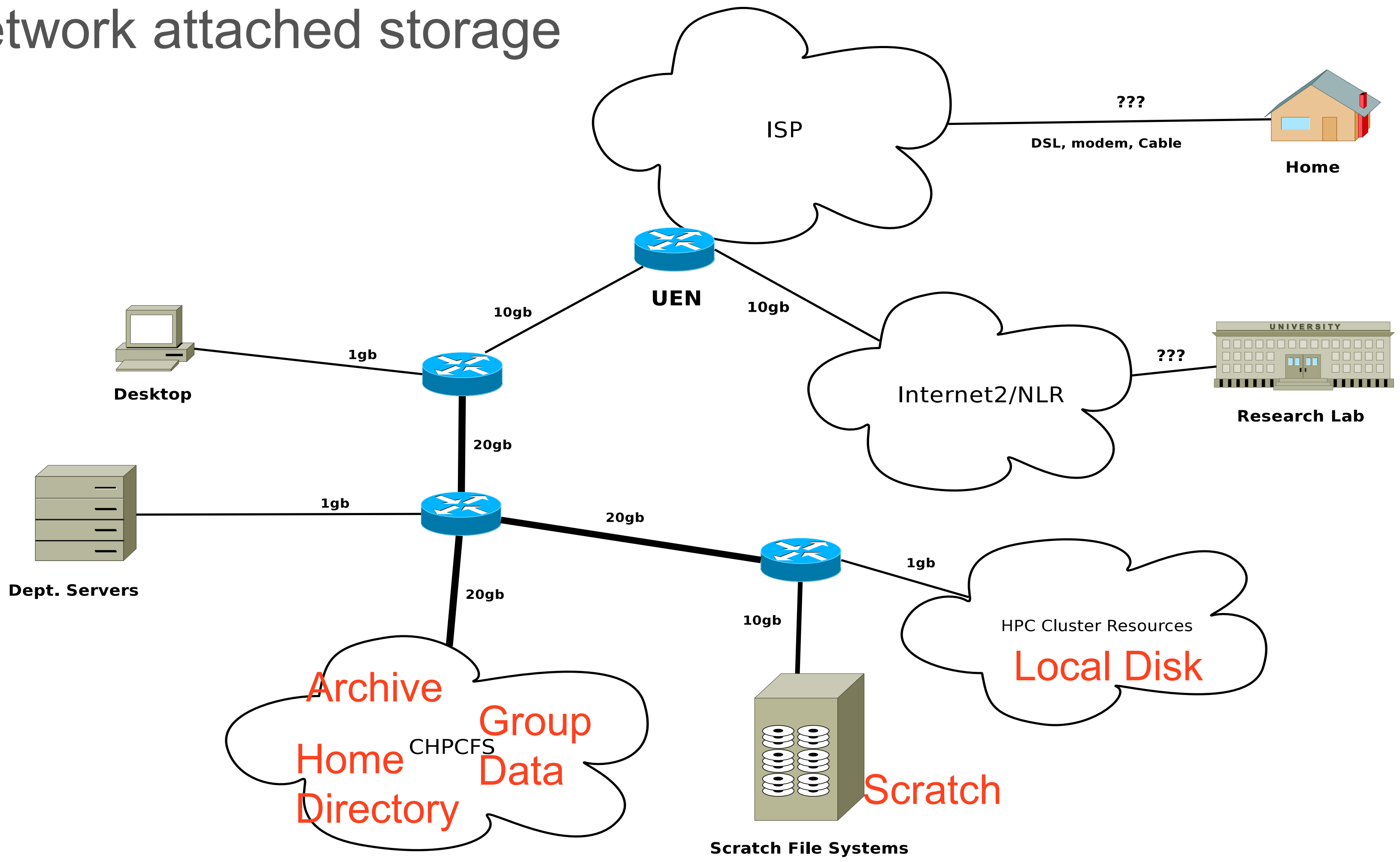
- Types of storage available at CHPC
- Types of file I/O
- Considerations for fast I/O
- Compressed I/O
- Parallel I/O

Storage options

- **Local Disk** (i.e. /scratch/local)
  - Most consistent I/O
  - No expectation of data retention
  - Unique per machine
- **Network Mounted Scratch** (i.e. /scratch/kingspeak/serial)
  - No expectation of data retention (It's scratch)
  - Expected to maintain a high level of I/O performance under significant load
- **Home Directory** (i.e. /uufs/chpc.utah.edu/common/home/uNID)
  - Per department backed up (except CHPC_HPC file system)
  - Intended for critical/volatile data
  - Expected to maintain a high level of responsiveness
- **Group Data Space** (i.e. /uufs/chpc.utah.edu/common/home/pi_grp)
  - Optional per department archive
  - Intended for active projects, persistent data, etc.
  - Usage expectations to be set by group
- **Archive storage** (i.e. pando-rgw1.chpc.utah.edu)

CHPC network topology

- Network attached storage



Storage considerations

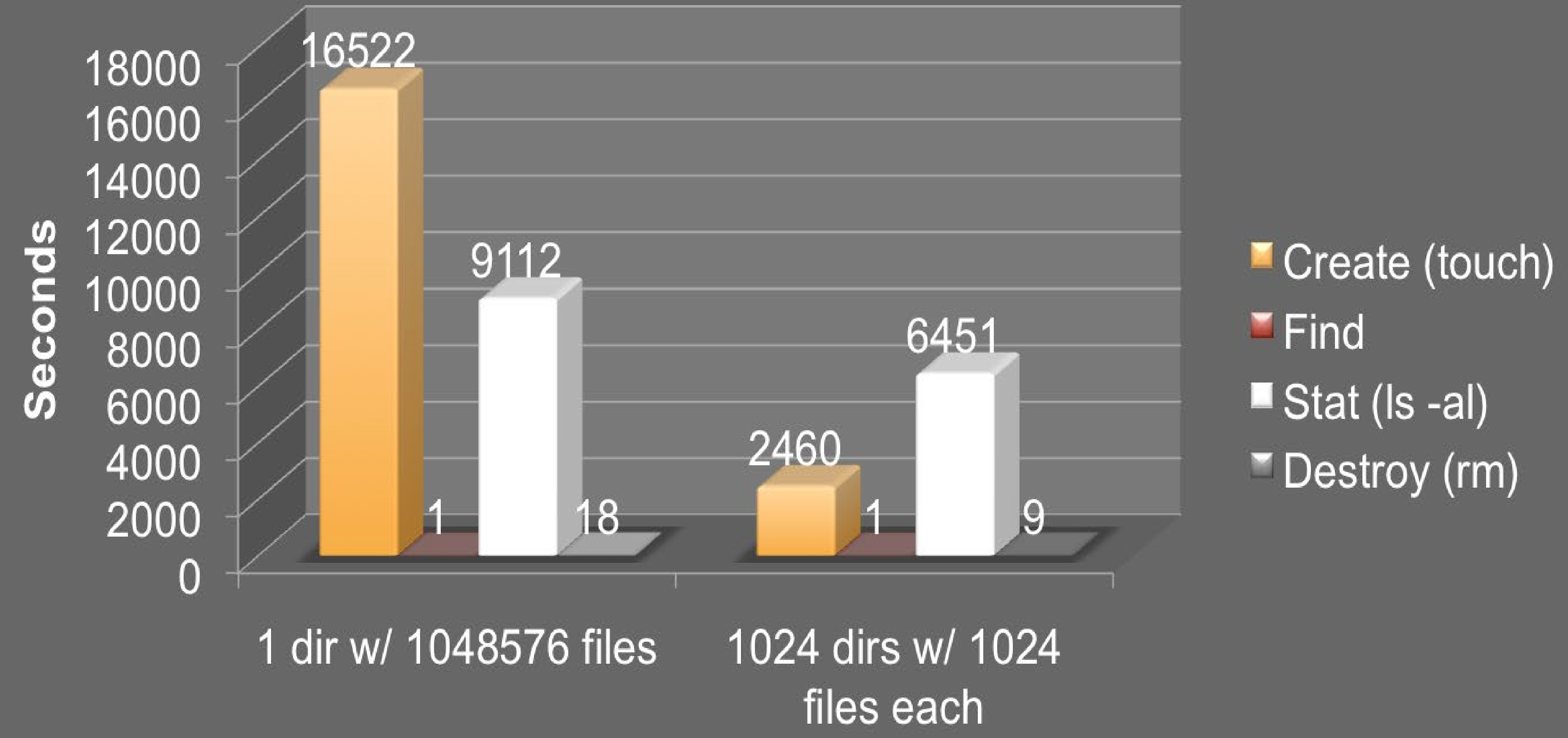
- Shared Environment
  - Many to one relationship (over-subscribed)
  - Consider your usage impact when choosing a storage location
  - Evaluate different I/O methodologies
  - Importance of the data (back up, scratch)
- Choose appropriate space
  - Programs, input + important data – backed up home – daily/weekly
  - Research data – group (if have one) – backed up ~ 4x/year
  - Reproducible, large data – scratch – no backup, guaranteed for 30 days
  - Temporary job data – local scratch – on each compute node, available during duration of the job
  - Archive – pando, Google Drive, Box
- Copying/moving files
  - What transfer performance is expected (network, file servers)
  - What transfer protocol to use (scp, rsync, **Globus**,…)

Performance considerations

- Directory Structure
  - Poor performance when too many files are in the same directory
  - Organizing files in a tree avoids this issue
  - Directory block count significance
- Network vs. Local
  - IOPS vs. Bandwidth
  - Network I/O
    - Overhead
    - Limited by network pipe
    - More efficient for bandwidth vs. IOPS
  - Local I/O
    - Limited size
    - Not globally accessible
    - Depending on hardware offers a fair balance between bandwidth and IOPS

Performance example

- Single Directory vs. Hierarchical Directory Structure



Troubleshooting performance

- Diagnosing Slowness
  - Open a ticket (issues@chpc.utah.edu)
  - File system
  - System load
  - Network load
- Monitoring
  - Ganglia
  - http://www.chpc.utah.edu graphs under Usage menu
  - Home and group, HPC cluster scratch
  - Use df -h to find what file server is your home, e.g.

```
df -h | grep u0101881
drycreek-vg5-0-lv1.chpc.utah.edu:/uufs/drycreek/common/drycreek-vg5-0-
lv1/chpc/u0101881
                      4.0T  2.9T  1.2T  72%
/uufs/chpc.utah.edu/common/home/u0101881
```

Data transfer options

- Smaller files to/from personal computers
  - scp (-C) – WinSCP (Windows), Cyberduck (Mac) – not persistent
  - rsync – Linux, persistent (`rsync -azv from to`)
- Large files
  - Use data transfer nodes (bypass campus firewall), https://www.chpc.utah.edu/documentation/data_services.php#Data_Transfer_Nodes
    `airplane0[1-4]-dmz.chpc.utah.edu`
    `dtn0[1,4]-dmz.chpc.utah.edu`
  - Linux/mac can use rsync
  - Set up Globus end point on your personal machine, https://www.chpc.utah.edu/documentation/software/globus.php
  - Globus is highly recommended due to its performance and resilience
    - E.g. transfer of 1 GB file from CHPC home directory to laptop on UConnect took 2 minutes using Globus and ½ hour using WinSCP
    - Can also share data with other users via a web link

rclone example

- Back up home directory files to Google Drive
  - https://gist.github.com/mcuma/dd7755736c7cca57d524c4831794238d
- Set up rclone

  -
    ```
    ml rclone
    rclone config
    ```
- Create an exclude file
  - Exclude hidden, options files
  - Be careful about backing up many small files – it'll take a very long time
- Create a new directory on Gdrive and do initial sync

  -
    ```
    rclone mkdir gdrive:myhome/myuser
    rclone -v sync $HOME gdrive:myhome/myuser --exclude-
    from $HOME/bin/scripts/exclude.txt --skip-links >&
    $HOME/bin/scripts/backup.log
    ```
- Run a cron job that automatically backs up e.g. daily
  - `crontab -e`
  - `0 2 * * * $HOME/bin/scripts/backup_to_gdrive.sh`

Google drive gotchas

- Google limits transferred amount to ~750 GB/day
  - Use rclone option `--bwlimit 70M` (70 Mbit/s)
- rclone currently does not support symbolic links, which is why we use --skip-links option. Links support is planned for the future.
- use exclude file on dot directories and other potential files as copying a lot of small files is very slow. It took me 3 days to copy 50 GB in 60000 files
- by default the UofU Google Drive permissions are set to be viewable by everyone at the U with a link. It is not easy to mass change this to all files being private. In Google Drive web based GUI one can only mass select files in current directory and the permission change does not propagate into subdirectories. www.whohasaccess.com can scan for files and turn off access but it seems like it can only go 2 levels deep

Programming I/O options

- **Plain file I/O**
  - ASCII text or binary
  - Plain text, formatted text (XML), binary
  - Largest disk space use, slowest
- **Compressed I/O**
  - Compress data before writing them to disk
  - Reduced disk usage, faster I/O (less data to read/write)
- **I/O libraries**
  - NetCDF, HDF
  - Flexible for structured data (e.g. different physical properties on a grid)
  - Can use compression OR parallel I/O
  - Data portability
- **Parallel I/O**
  - Faster performance if have parallel file system
  - Easier I/O from parallel (MPI) processes
  - Maintain a good ratio between number of compute tasks and file system servers

Plain file I/O

- **Simplest but the least efficient**
  - Text representation of a number takes a lot of space
  - Binary files require specific formatting
  - Plain text, formatted text (XML, Json), binary
- **Uses**
  - Simple input/output files (e.g. simulation parameters)
  - Larger data better in binary form

```
double a[100];
myfile = fopen("testfile", "w");
fwrite(a,sizeof(double),100,myfile);
fclose(myfile);
```

Compressed file I/O

- **Write routines to compress/uncompress data**
  - We have some that we're happy to share
  - C++ can use Boost Iostreams library
  - Interpreted languages have higher level libraries for this

- **Uses**
  - Large amount of single type data
  - For different arrays use different files

```
#include <zlib.h>
int bzwrite(char * filename, double *a, int n)
{ … }

double a[100];
bzwrite("testfile",a,100);
```

I/O libraries

- **Lots of functions**
  - File open/close
  - Combine multiple data into single data structures
  - Define data views
  - Define data hierarchies
  - Split data between files/processes
  - File compression
  - Parallel I/O
- **Specific formats**
  - NetCDF – Network Common Data Form – from UCAR
  - HDF – Hierarchical Data Format – from NCSA
- **Use**
  - Call routines from the library
  - Include headers in the source files
  - Link libraries during executable build

Compressed HDF5 write

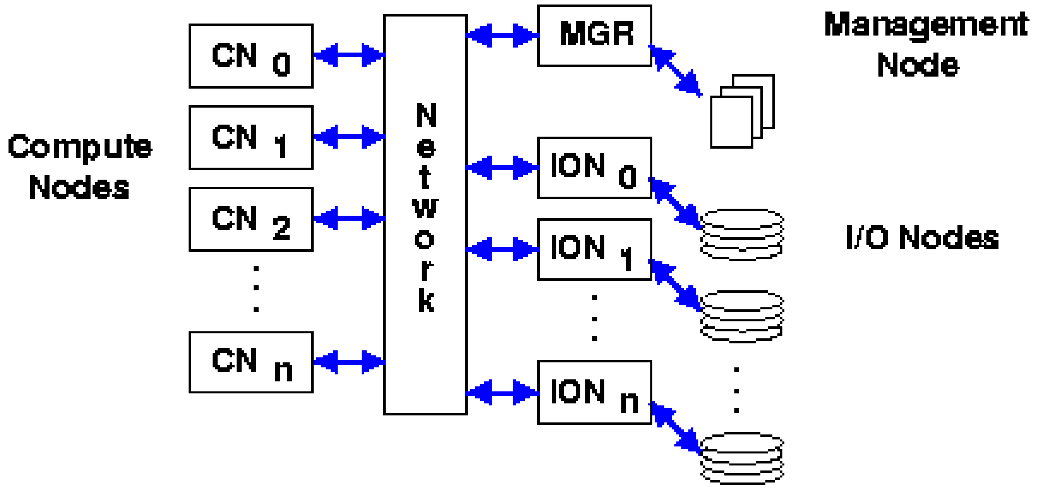
```
// test if SZIP compression is available
avail = H5Zfilter_avail(H5Z_FILTER_SZIP);
// Create a new file using the default properties.
file = H5Fcreate (filename, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
// Create dataspace of size "dims".
space = H5Screate_simple (1, &dims, NULL);
// Create the dataset creation property list, add the szip compression filter and set the chunk size.
dcpl = H5Pcreate (H5P_DATASET_CREATE);
status = H5Pset_szip (dcpl, H5_SZIP_NN_OPTION_MASK, 8);
status = H5Pset_chunk (dcpl, 1, &chunk);
// Create the dataset.
dset = H5Dcreate (file, dataset, H5T_NATIVE_DOUBLE, space, H5P_DEFAULT, dcpl,
    H5P_DEFAULT);
// Write the data to the dataset.
status = H5Dwrite (dset, H5T_NATIVE_DOUBLE, H5S_ALL, H5S_ALL, H5P_DEFAULT,
             out);
// Close and release resources.
status = H5Pclose (dcpl);
status = H5Dclose (dset);
status = H5Sclose (space);
status = H5Fclose (file);
```

I/O in interpreted languages

- **Matlab**
  - Default mat file uses compressed HDF5
  - `save('myfile.mat',variables)`
- **Python**
  - Pickle files – serializes objects to file – faster than text I/O
    - E.g. Pandas `mydataframe.to_pickle('myfile.pkl')`
  - HDF5 – `h5py` module (including parallel with `mpi4py`)
- **R**
  - Native R format, `.Rds`, `.RData`
  - Several HDF5 libraries, most common is `rhdf5` from Bioconductor

Parallel I/O

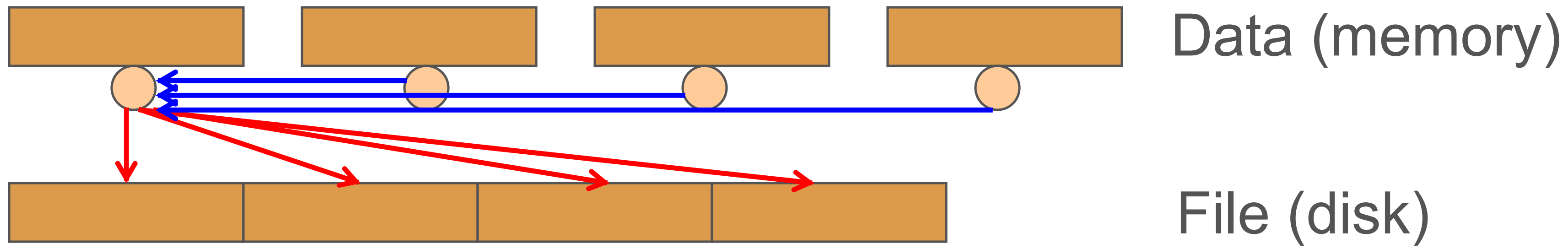
- **Best with parallel file system**
  - Many paths between the compute nodes and I/O nodes
  - **Lustre**, GPFS, BeeGFS
- **Also works with other network file systems**
  - Single path to the file server but then may spread the files to many disks to speed up the I/O
  - **NFS**
- **MPI-IO**
  - Part of MPI standard
  - Individual and collective I/O functions
  - Allows parallel I/O from multiple nodes to single file
  - Regular or irregular array file access (derived data types)



THE UNIVERSITY OF UTAH™ TOGETHER WE REACH

Non-parallel I/O

- **Non-parallel I/O from an MPI program**



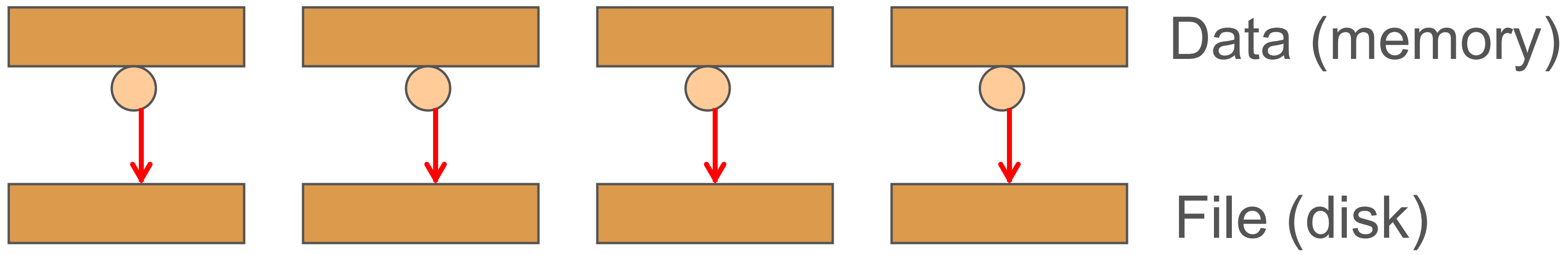
```
MPI_Status status;  FILE *myfile;
for (i=0;i<BUFSIZE;i++) buf[i]=myrank*BUFSIZE+i;
if (myrank != 0)
    MPI_Send(buf,BUFSIZE,MPI_INT,0,99,MPI_COMM_WORLD);
else {
    myfile = fopen("testfile", "w");
    fwrite(buf,sizeof(int),BUFSIZE,myfile);
    for (i=1;i<numprocs;i++) {
       MPI_Recv(buf,BUFSIZE,MPI_INT,i,99,MPI_COMM_WORLD,&status);
       fwrite(buf,sizeof(int),BUFSIZE,myfile);
    }
    fclose(myfile);
}
```

Pros:
- close to serial code
- big blocks – better perf.
- single file

Cons:
- no parallelism limits scalability, perf.

Parallel I/O, no MPI

- **Non-MPI parallel I/O**

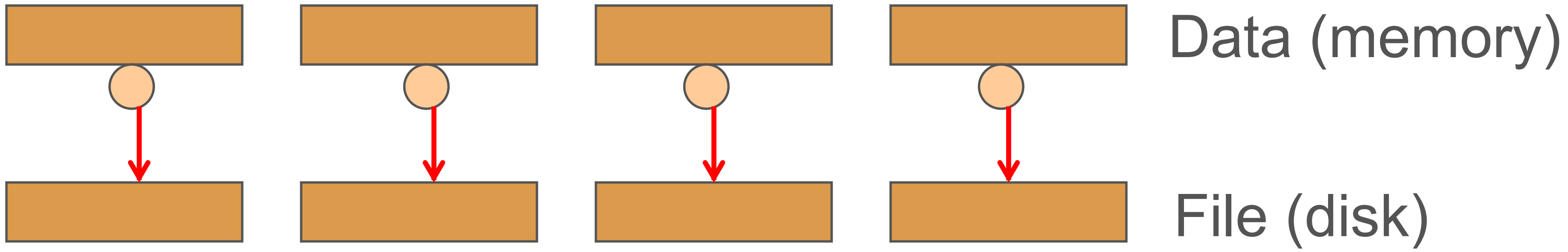


```
char filename[128];
FILE *myfile;

for (i=0;i<BUFSIZE;i++) buf[i]=myrank*BUFSIZE+i;
sprintf(filename, "testfile.%d", myrank);

myfile = fopen(filename, "w");
fwrite(buf,sizeof(int),BUFSIZE,myfile);
fclose(myfile);
```

Pros:
- parallelism
- possibility to compress
Cons:
- lots of files (potentially small)
- harder to combine data from distributed files

MPI-I/O to separate files

- **MPI-I/O to separate files**

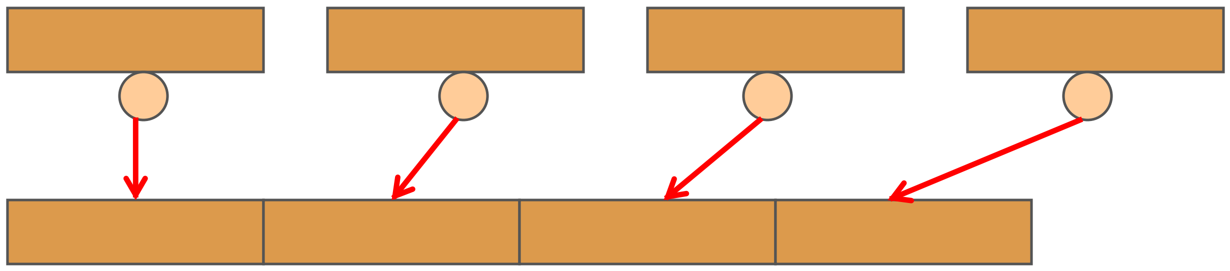


```
char filename[128];
MPI_File myfile; MPI_Status status;

for (i=0;i<BUFSIZE;i++) buf[i]=myrank*BUFSIZE+i;
sprintf(filename, "/scratch/ibrix/chpc_gen/username/testfile.%d", myrank);

MPI_File_open(MPI_COMM_SELF, filename, MPI_MODE_WRONLY|MPI_MODE_CREATE,
              MPI_INFO_NULL, &myfile);
MPI_File_write(myfile, buf, BUFSIZE, MPI_INT, &status);
MPI_File_close(&myfile);
```

- same as Example 2
- easy way to start with MPI-I/O
- individual communication

MPI-I/O to one file

- **MPI-I/O to a single file**



Pros:
- single file
- parallel I/O
- can use MPI data structures

Cons:
- no compression

```
MPI_File myfile; MPI_status status; MPI_Offset offset;

for (i=0;i<BUFSIZE;i++) buf[i]=myrank*BUFSIZE+i;

MPI_File_open(MPI_COMM_WORLD, "/scratch/general/lustre/username/testfile",
              MPI_MODE_WRONLY|MPI_MODE_CREATE, MPI_INFO_NULL, &myfile);
offset = myrank*BUFSIZE*sizeof(int);
MPI_File_set_view(myfile, offset, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_write(myfile, buf, BUFSIZE, MPI_INT, &status);
/* MPI_File_write_at(myfile, offset, buf, BUFSIZE, MPI_INT, &status); */
MPI_File_close(&myfile);
```

MPI-I/O file open/close

- **MPI_File_open** to open file

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

- flags – MPI_MODE_RDONLY, MPI_MODE_WRONLY, MPI_MODE_RDWR, MPI_MODE_CREATE, MPI_MODE_APPEND, …
- combine with bitwise-or '|' in C or with addition '+' in Fortran

```
MPI_File_open(MPI_COMM_WORLD, "/scratch/general/lustre/testfile”,
  MPI_MODE_WRONLY|MPI_MODE_CREATE, MPI_INFO_NULL, &myfile);
```

- **MPI_File_close** to close file

```
MPI_File_close(fileh, ierr)
int MPI_File_close(MPI_File *fh)
```

```
MPI_File_close(&myfile);
```

MPI-I/O file read/write

- **MPI_File_write** or **MPI_File_write_at** to write into file

```
MPI_File_write(fh, buf, count, datatype, status, ierr)
MPI_File_write_at(fh, offset, buf, count, datatype, status, ierr)
int MPI_File_write(MPI_File fh, void *buf, int count,
MPI_Datatype datatypetype, MPI_Status status)
int MPI_File_write(MPI_File fh, MPI_Offset offset, void *buf, int
count, MPI_Datatype datatypetype, MPI_Status status)
MPI_File_write(myfile, buf, BUFSIZE, MPI_INT, &status);
```

- **MPI_File_read** or **MPI_File_read_at** to read from a file

```
MPI_File_read(fh, buf, count, datatype, status, ierr)
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierr)
int MPI_File_read(MPI_File *fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status status)
int MPI_File_read_at(MPI_File *fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Status status)
call MPI_File_read(fh, buf, nints, MPI_INTEGER, status, ierr)
```

MPI-I/O file views

- **MPI_File_set_view** – assigns regions of the file to separate processors

```
MPI_File_set_view(fh, offset, etype, filetype, datarep, info,
  ierr)
int MPI_File_set_view(MPI_File fh, MP_offset offset, MPI_Datatype
  etype, MPI_datatype filetype, char *datarep, MPI_Info info)
MPI_File_set_view(myfile, myrank*BUFSIZE*sizeof(int), MPI_INT,
  MPI_INT, "native", MPI_INFO_NULL);
```

- View specified by triplet
  - offset – no. bytes skipped from the start of the file
  - etype –unit of data in the memory
  - filetype – unit of data in the file
  - MPI_File_read or MPI_File_read_at to read from a file

MPI-I/O file read

```
include 'mpif.h'
integer status(MPI_STATUS_SIZE); integer (kind=MPI_OFFSET_KIND) offset
integer fh; integer (kind=MPI_OFFSET_KIND) FILESIZE

call MPI_File_open(MPI_COMM_WORLD, "/scratch/global/username/testfile", &
                   MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
call MPI_File_get_size(fh, FILESIZE)
nints = FILESIZE / (nprocs*INTSIZE)
offset = rank * nints * INTSIZE
call MPI_File_seek(fh, offset, MPI_SEEK_SET, ierr)
call MPI_File_read(fh, buf, nints, MPI_INTEGER, status, ierr)
call MPI_Get_count(status, MPI_INTEGER, count, ierr)
print *, 'process ', rank, 'read ', count, 'integers '
call MPI_File_close(fh, ierr)
```

- MPI_File_seek – set offset

```
MPI_File_seek(fileh, offset, whence, ierr)
int MPI_File_seek(MPI_File *fh, MPI_Offset offset, int whence)
```

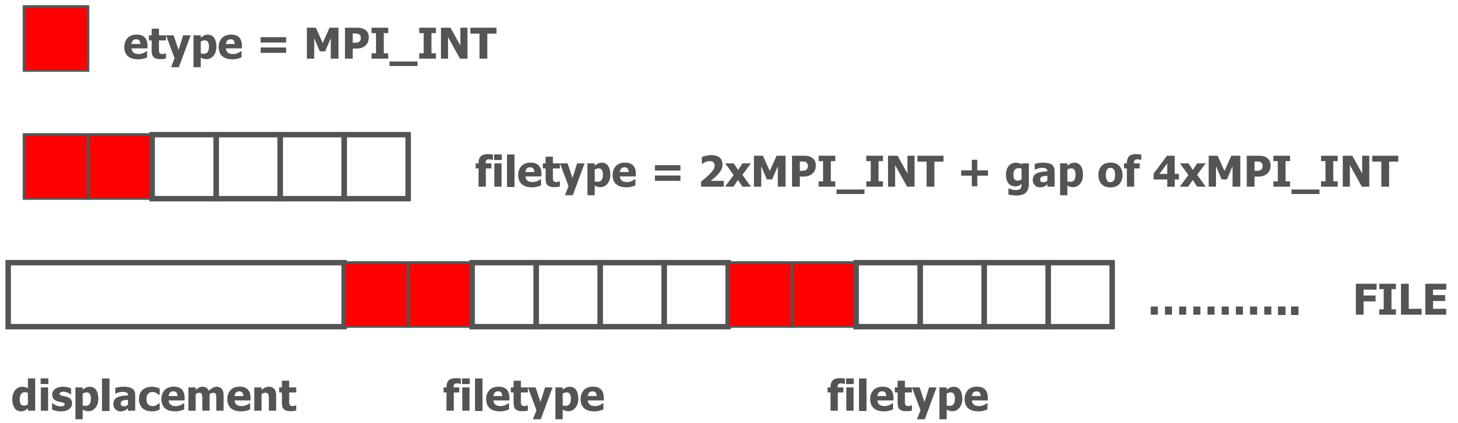
  - whence – MPI_SEEK_SET, MPI_SEEK_CUR, MPI_SEEK_END

MPI-I/O collectives

- All processors in group/communicator used to open file will take part in I/O
- Generally more efficient than individual I/O
- Each process specifies its own info
- Implementation can optimize I/O
- Argument list the same as in non-collective functions
- Function names – add `_all`
- e.g. `MPI_File_read_all`

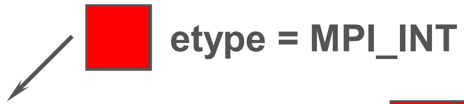
Noncontiguous access

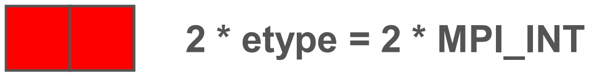
- E.g. distributed arrays stored in files
- Specify noncontiguous access in memory and file within a single function using derived data types
- Simple file view example:

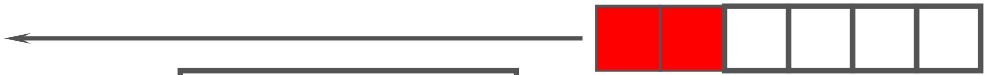


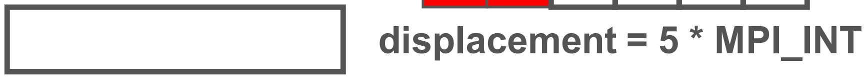
File view example

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp; MPI_Status status;
MPI_File myfile

MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0; extent = 6*sizeof(int);
MPI_Type_create_resized(contig, lb, extent, filetype);
MPI_Type_commit(&filetype);
disp = 5*sizeof(int); etype= MPI_INT;

MPI_File_open(MPI_COMM_WORLD, "/scratch/general/lustre/username/datafile",
              MPI_MODE_RDWR|MPI_MODE_CREATE, MPI_INFO_NULL, &myfile);
MPI_File_set_view(myfile, disp, etype, filetype, "native", MPI_INFO_NULL);
MPI_File_write_all(myfile, buf, 1000, MPI_INT, &status);
MPI_File_close(&myfile);
MPI_Type_free(&filetype);
```









Can use any MPI data types, including distributed data types (Darray, Subarray)

Parallel HDF5 write

```
// Set up file access property list with parallel I/O access
plist = H5Pcreate(H5P_FILE_ACCESS);
status = H5Pset_fapl_mpio(plist, MPI_COMM_WORLD, info);
// Create a new file using the default properties.
file = H5Fcreate (filename, H5F_ACC_TRUNC, H5P_DEFAULT, plist);
status = H5Pclose(plist);
// Create dataspace.  Setting maximum size to NULL sets the maximum size to be the current size.
filespace = H5Screate_simple (1, &gdims, NULL);
//Create the dataset creation property list,
dset = H5Dcreate (file, dataset, H5T_NATIVE_DOUBLE, filespace, H5P_DEFAULT, H5P_DEFAULT,
    H5P_DEFAULT);
// Define and select the hyperslab selection.
offset[0] = poffset; offset[1] = 0; // global offset of the local data (1 dimensional array)
count[0] = dims; count[1] = 1; // amount of local data to write
memspace = H5Screate_simple(1, count, NULL);
status = H5Sselect_hyperslab (filespace, H5S_SELECT_SET, offset, NULL, count, NULL);
// Create property list for collective dataset write.
plist = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist, H5FD_MPIO_COLLECTIVE);
// Write the data to the dataset.
status = H5Dwrite (dset, H5T_NATIVE_DOUBLE, memspace, filespace, plist, out);
// Close and release resources.
status = H5Dclose (dset);
status = H5Sclose (filespace);
status = H5Sclose (memspace);
status = H5Pclose(plist);
status = H5Fclose (file);
```

Summary

- Types of data storage at CHPC
- Data transfer options
- Kinds of I/O
  - Plain, compressed, libraries, MPI-I/O
- Examples of each I/O kind
- Parallel I/O with examples