Introduction to I/O at CHPC

Martin Čuma, m.cuma@utah.edu

Center for High Performance Computing

Fall 2016

Outline

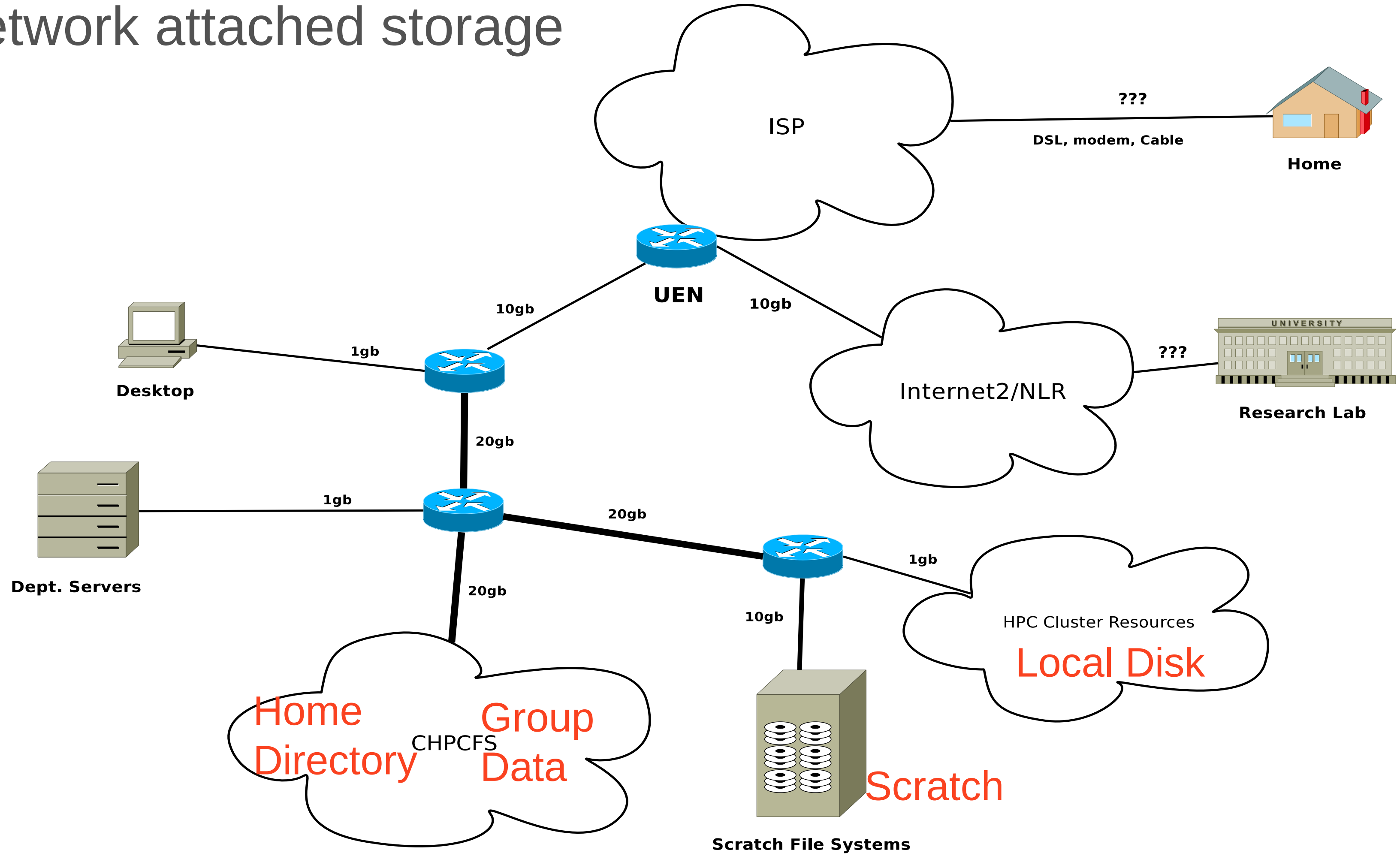
- Types of storage available at CHPC
- Types of file I/O
- Considerations for fast I/O
- Compressed I/O
- Parallel I/O

Storage options

- **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
- **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
- **Local Disk** (i.e. /scratch/local)
  - Most consistent I/O
  - No expectation of data retention
  - Unique per machine

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
- Copying/moving files
  - What transfer performance is expected (network, file servers)
  - What transfer protocol to use (scp, rsync, Globus,…)

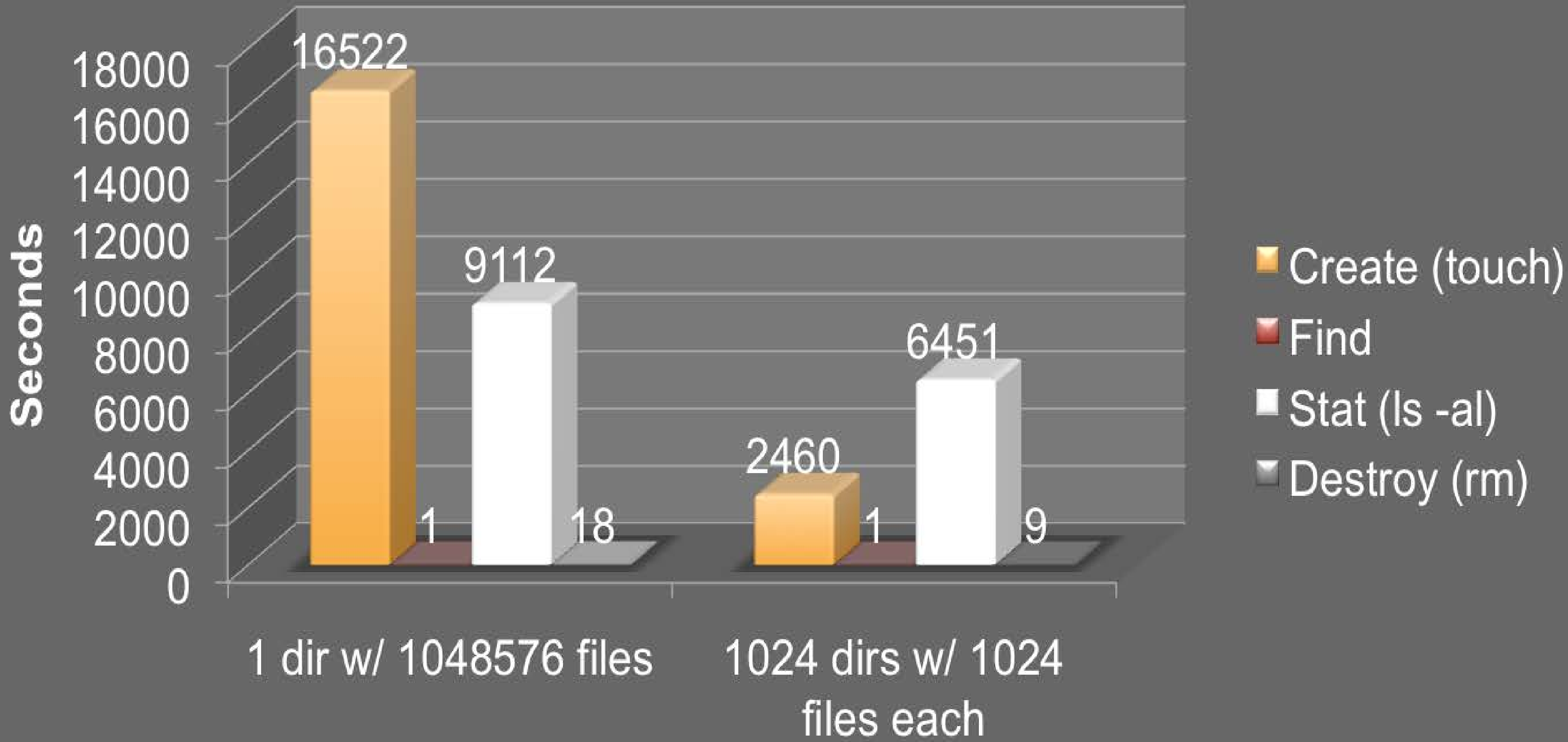
Performance considerations

- 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

THE UNIVERSITY OF UTAH™

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
```

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

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), 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

- **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

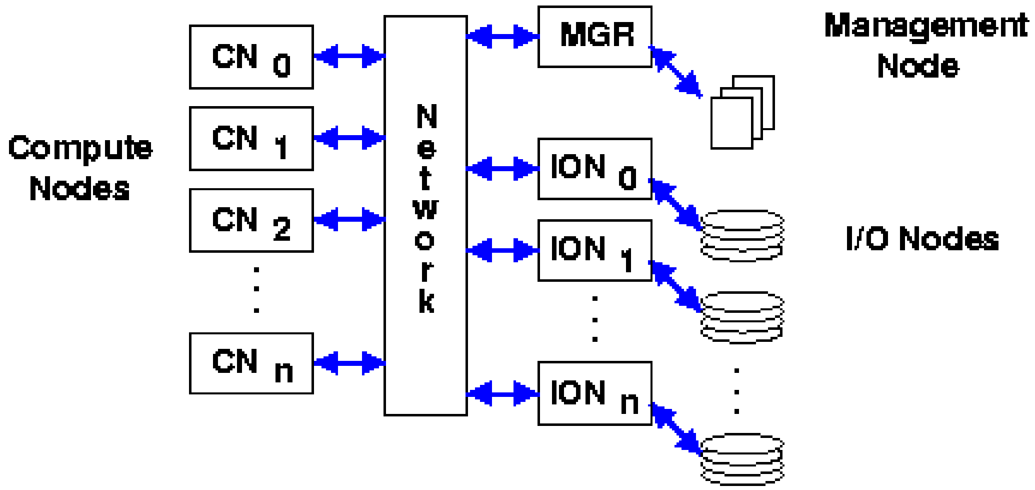
```
// test if SZIP compression is available
avail = H5Zfilter_avail(H5Z_FILTER_SZIP);
// Create a new file using the default properties.
file = H5Fcreate (filename, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
// Create dataspace. rent size.
space = H5Screate_simple (1, &dims, NULL);
// Create the dataset 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);
```

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;
count[0] = dims; count[1] = 1;
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);
```

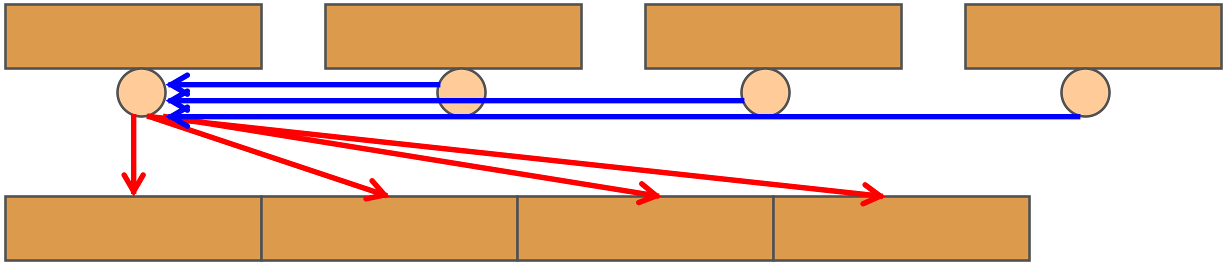
Parallel I/O

- **Best with parallel file system**
  - Many paths between the compute nodes and I/O nodes
  - **Lustre**, GPFS, **IBRIX**, PVFS2
- **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)



Example 1

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



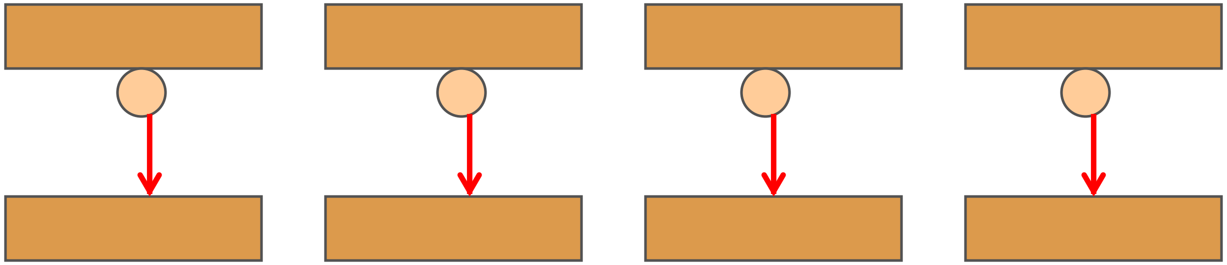
```
MPI_Status status;  FILE *myfile;
for (i=0;i<BUFSIZE;i++) buf[i]=myrank*BUFSIZE+i;
if (myrank != 0)
    MPI_Send(buf,BUFSIZE,MPI_INT,0,99,MPI_COMM_WORLD);
else {
    myfile = fopen("testfile", "w");
    fwrite(buf,sizeof(int),BUFSIZE,myfile);
    for (i=1;i<numprocs;i++) {
       MPI_Recv(buf,BUFSIZE,MPI_INT,i,99,MPI_COMM_WORLD,&status);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

Example 2

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



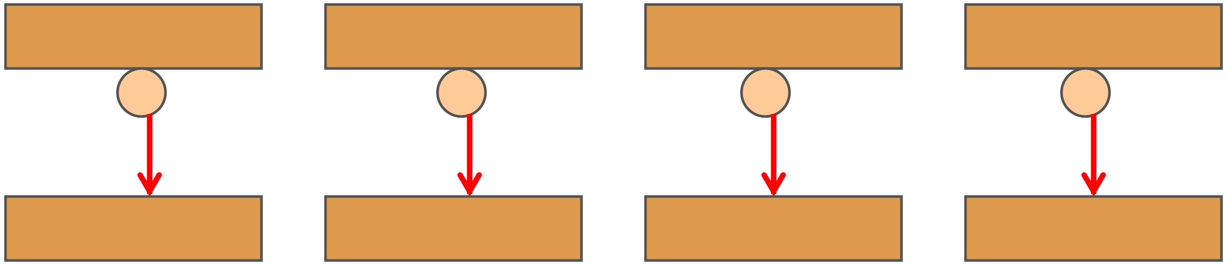
```
char filename[128];
FILE *myfile;

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

Example 3

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

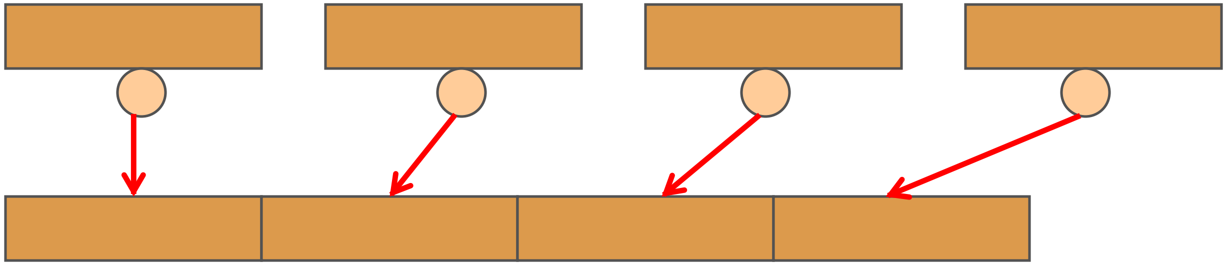


```
char filename[128];
MPI_File myfile; MPI_Status status;

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

Example 4

- **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);
```

THE UNIVERSITY OF UTAH™

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 – basic unit of data access
  - filetype – which portion of the file is visible to the process
  - 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)
```

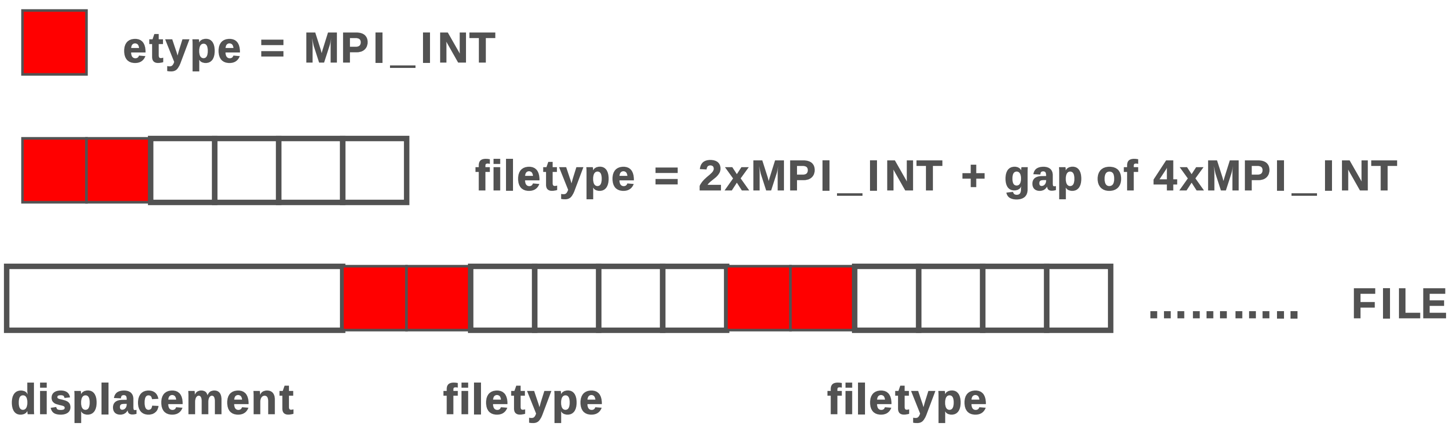
  - whence – MPI_SEEK_SET, MPI_SEEK_CUR, MPI_SEEK_END

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`

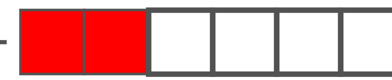
Noncontiguous access

- 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)

Summary

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