Parallel I/O

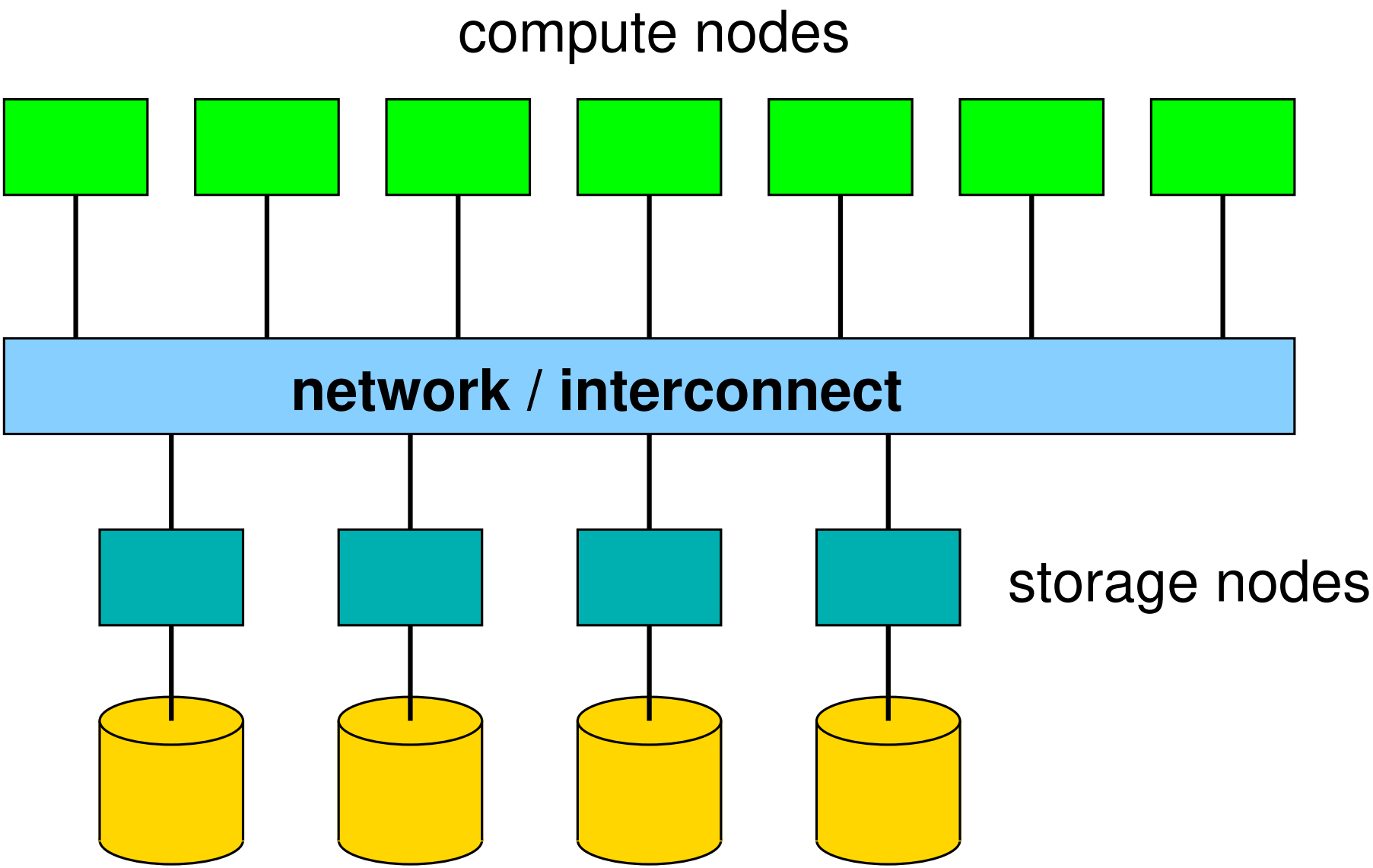
CPS343

Parallel and High Performance Computing

Spring 2020

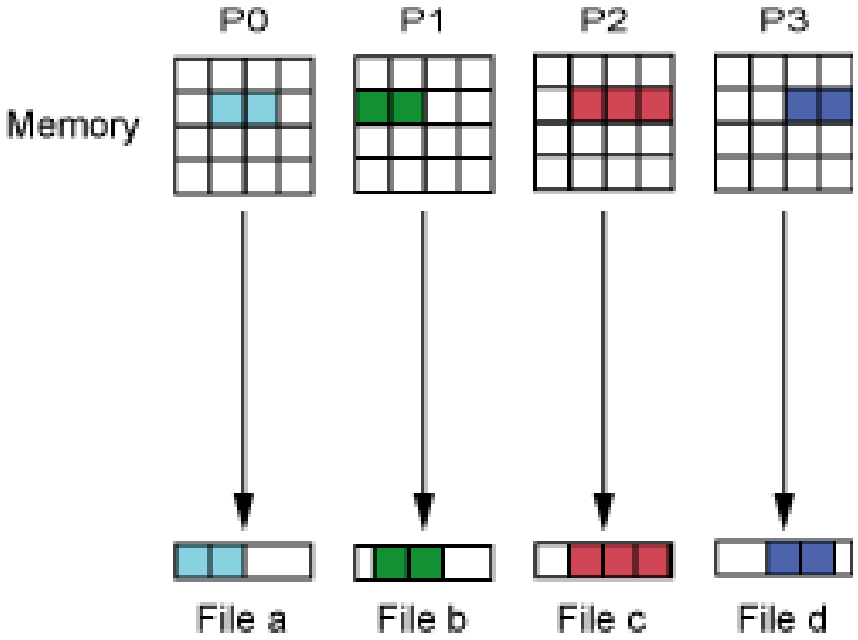
Outline

Parallel cluster structure



Outline

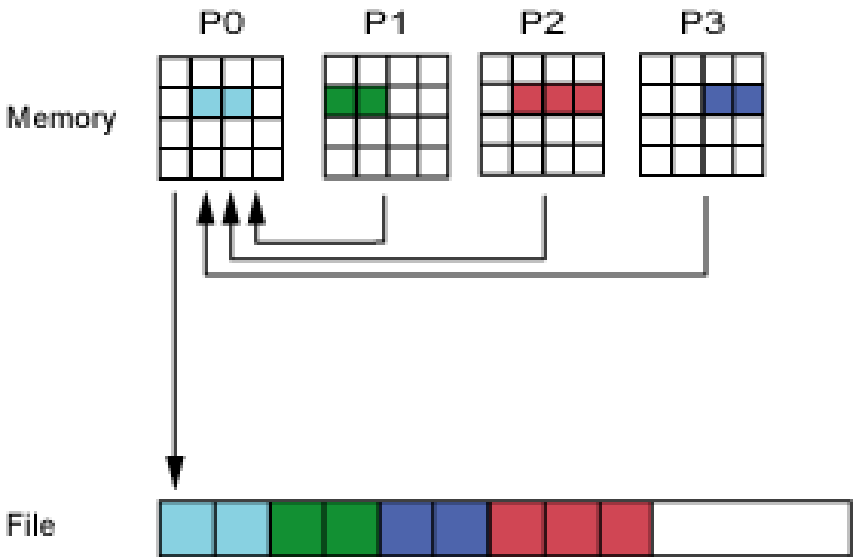
One file per process: all write



- Each process writes to its own file.
- Relatively simple I/O strategy.
- Independent writes can perform well because multiple storage servers can support parallel I/O to many separate files.
- Does not scale well if many nodes will subsequently need to access many different files.

Image source: Section 3 in http://docs.cray.com/cgi-bin/craydoc.cgi?mode=View;id=S-2490-40

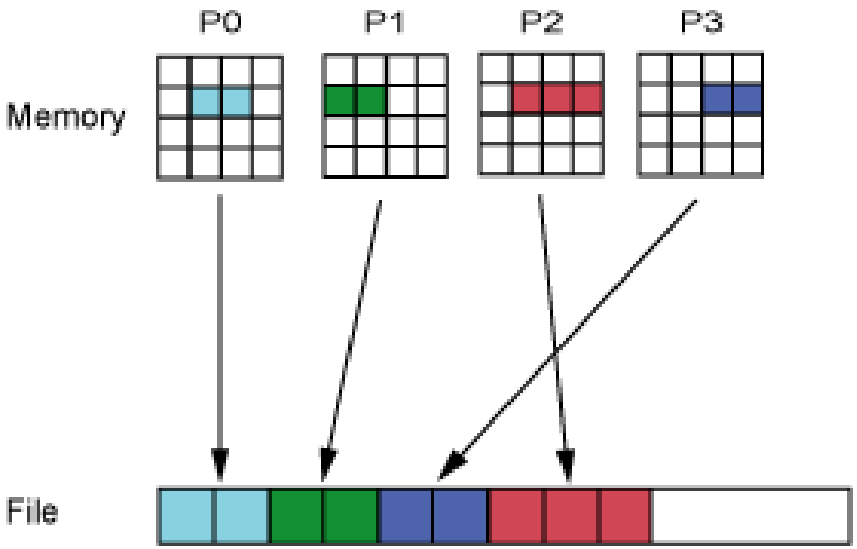
Single shared file: one writes



- All processes involved in a write operation send data to one process, which then writes the data to a single shared file.
- Relatively simple; consolidates data which may be an advantage.
- Uses data aggregation.
- Writes are sequential and performance is limited by bandwidth of single process.

Image source: Section 3 in http://docs.cray.com/cgi-bin/craydoc.cgi?mode=View;id=S-2490-40

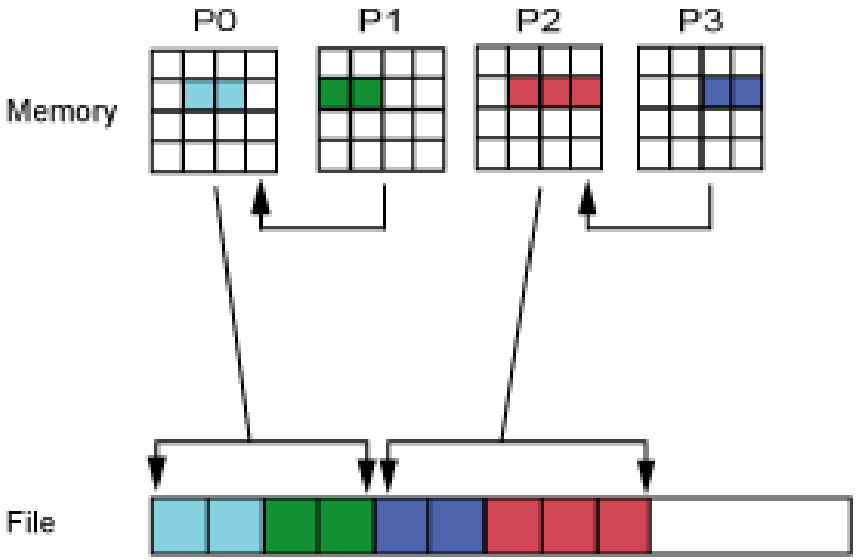
Single shared file: all write



- All processes involved in a write operation send data to a single shared file.
- Requires extra work by the application to maintain the separate file offsets for each process.
- Parallel I/O can be achieved.

Image source: Section 3 in http://docs.cray.com/cgi-bin/craydoc.cgi?mode=View;id=S-2490-40

Single shared file: some write



- A subset of all processes involved in a write operation sends data to a single shared file.
- Uses data aggregation.
- Sometimes used to strike a balance between many processes writing at the same time (with resulting frequent extent lock contention) and only one process writing (with the resulting loss of parallel I/O).

Image source: Section 3 in http://docs.cray.com/cgi-bin/craydoc.cgi?mode=View;id=S-2490-40

MPI I/O

- Developed by IBM in 1994 and subsequently appeared in MPI-2 standard
- Uses underlying MPI send/receive routines to move data
- Supports MPI derived datatypes
- Both blocking and non-blocking send/receive modes are supported; allows I/O in parallel with computation
- read/write operations can be *independent* or *collective*

Independent vs Collective I/O

Independent I/O

- The "one process per file" mode we've already seen is an example of independent I/O
- I/O operations can occur in parallel
- Only one process is reading to and/or writing from the file

Collective I/O

- Like other MPI collective operations, *all* processes sharing a communicator must participate
- I/O operations can occur in parallel
- Typically each process reads or writes a portion of the file corresponding to its own memory space

Outline

Parallel I/O stack

The parallel I/O stack on a Cray XT system is diagrammed as

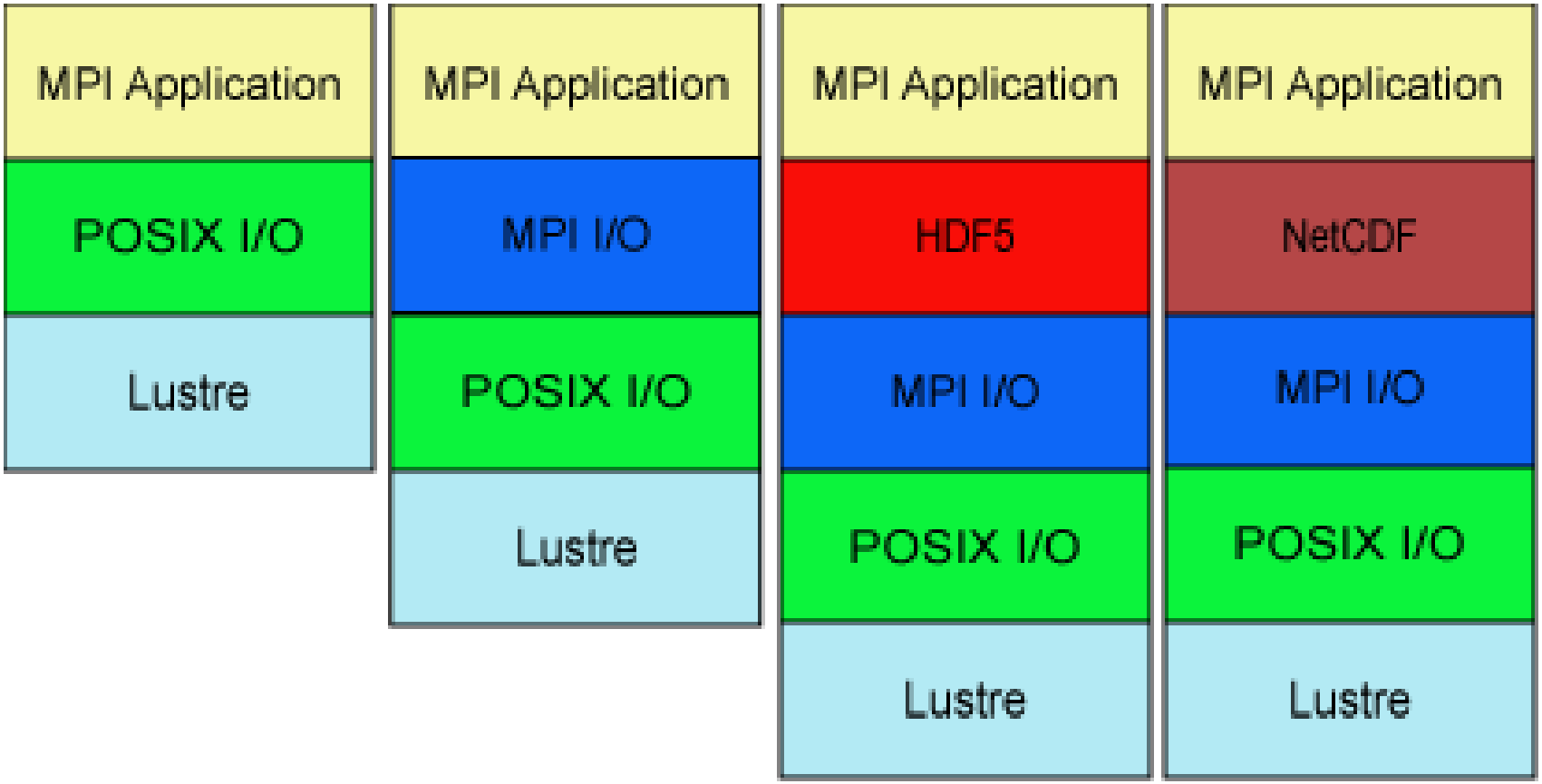


Image source: Section 2 in http://docs.cray.com/cgi-bin/craydoc.cgi?mode=View;id=S-2490-40

Outline

Hyperslabs

In HDF5 a *hyperslab* is a section of data. It is specified by four arrays, each having the same dimension as the dataspace the hyperslab belongs to. For dimension i:

offset (also called *start*) The offset to the start of the hyperslab in dimension i

stride The increment between one element in the hyperslab to the next element in dimension i

count The number of blocks to read or write from dimension i

block The number of elements in a block along dimension i

See "Writing and Reading Hyperslabs" at the bottom of https://portal.hdfgroup.org/display/HDF5/Introduction+to+Parallel+HDF5.

Outline

Example: Writing in parallel to an HDF5 file

- We assume that unique data is distributed among multiple processes and each process wants to write its portion of the data into a single shared file at the appropriate location.
- That is, the data in the output file should be organized in a specific way and the various processes need to write their data into the file at the appropriate place to honor the desired organization.
- In this example we assume we have an $N_x \times N_y$ 2-D Cartesian grid where each process holds an $n_x \times n_y$ portion of the grid and we want to write the data into a single file as if one process wrote the entire grid.

Example: Writing in parallel to an HDF5 file

The following are the typical steps used to create and write an HDF5 file in parallel.

1. Create a property list from the MPI communicator.
2. Create the output file.
3. Create a global dataspace and dataset descriptor for the output file that will hold data from all processes.
4. Create the local dataspace corresponding to portion of the global dataspace belonging to a single process.
5. Define the hyperslab in the global dataspace for the data.
6. Define the hyperslab in the local dataspace for the data.
7. Set transfer mode to be collective (not independent).
8. Write the data.
9. Release all open dataspaces, datasets descriptors, etc.

Create a property list and file

- First we create the property list `plist_id` and initialize it with data from the MPI communicator `comm`.

```
hid_t plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, MPI_INFO_NULL);
```

- Then we can create an empty file with the properties contained in the property list.

```
hid_t file_id = H5Fcreate(fname, H5F_ACC_TRUNC,
                          H5P_DEFAULT, plist_id);
H5Pclose(plist_id);
```

Note that we can also close (i.e. free) the property list since we're finished with it.

Create dataspaces and file dataset descriptor

- The global dataspace, representing what will be written to the file, should hold the entire grid so it must be dimensioned $N_x \times N_y$.

```
hsize_t dimsf[2], dimsm[2];
dimsf[0] = NX;
dimsf[1] = NY;
dataspace_id = H5Screate_simple(2, dimsf, NULL);
```

- We also create the dataset named `"/grid"` in the file.

```
dataset_id = H5Dcreate(file_id, "/grid", H5T_IEEE_F64LE,
                       dataspace_id,
                       H5P_DEFAULT, H5P_DEFAULT,
                       H5P_DEFAULT);
```

- Each process contains an $n_x \times n_y$ subgrid of data (perhaps including halo/ghost data) and here we create a dataspace to represent this.

```
dimsm[0] = halo_grid->nx;
dimsm[1] = halo_grid->ny;
memspace_id = H5Screate_simple(2, dimsm, NULL);
```

Define the hyperslab in the global dataspace

We need to define two hyperslabs which have the same dimensions. The first locates the local data in the global dataspace while the second locates the same data in the local dataspace.

- Create hyperslab for $n_x \times n_y$ subgrid starting at (x_0, y_0).

```
hsize_t offset[t], count[2];
offset[0] = orig_grid->x0;
offset[1] = orig_grid->y0;
count[0]  = orig_grid->nx;
count[1]  = orig_grid->ny;
H5Sselect_hyperslab(dataspace_id, H5S_SELECT_SET,
                    offset, NULL, count, NULL);
```

- Create hyperslab for $n_x \times n_y$ subgrid start at either $(0, 0)$ (in the case of no halo) or at some offset location to skip over halo values.

```
offset[0] = orig_grid->x0 - halo_grid->x0;
offset[1] = orig_grid->y0 - halo_grid->y0;
H5Sselect_hyperslab(memspace_id, H5S_SELECT_SET,
                    offset, NULL, count, NULL);
```

Set transfer mode and write the data

We're now almost ready to write. First we must set the data transfer mode to *collective* rather than the default mode *independent*.

- Get the current dataset transfer property list and modify it to be *collective*.

```
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
```

- Finally we can write the data. The address of first element of the local data is `&u[0][0]`; any adjustment do to halo data (which should not be written) was accounted for when the local hyperslab was defined.

```
H5Dwrite(dataset_id, H5T_NATIVE_DOUBLE, memspace_id,
         dataspace_id, plist_id, &u[0][0]);
```

Release all open dataspaces, datasets descriptors, etc

- We are all done, we just need to free up the resources we allocated in order to write the file and close the file.

```
H5Pclose(plist_id);
H5Sclose(memspace_id);
H5Dclose(dataset_id);
H5Sclose(dataspace_id);
H5Fclose(file_id);
```

- The complete example program can be found at: http://www.cs.gordon.edu/courses/cps343/code/cart-hdf5.cc