Lecture 33: More on MPI I/O

William Gropp
www.cs.illinois.edu/~wgropp

Today's Topics

- High level parallel I/O libraries
  - Options for efficient I/O
- Example of I/O for a distributed array
- Understanding why collective I/O offers better performance
- Optimizing parallel I/O performance

Portable File Formats

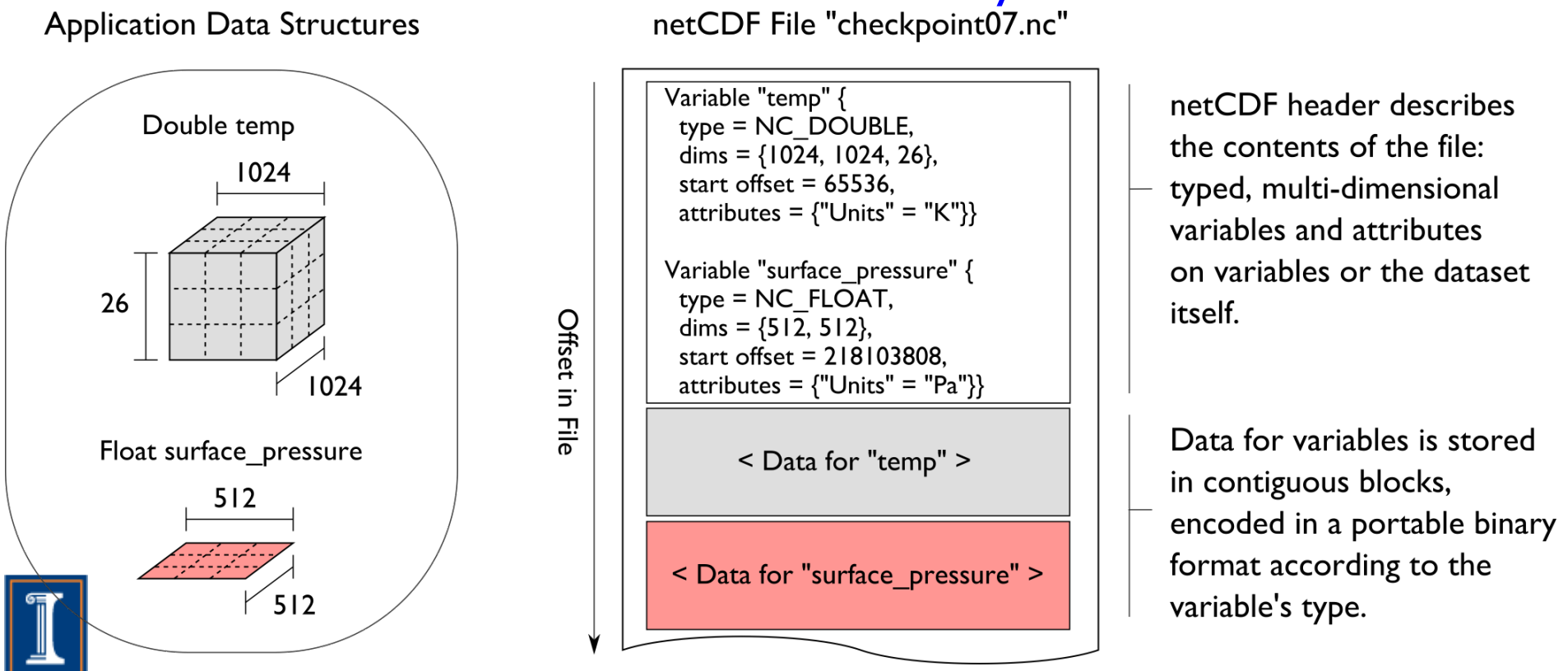
- Ad-hoc file formats
  - ♦ Difficult to collaborate
  - ♦ Cannot leverage post-processing tools
- MPI provides external32 data encoding
  - ♦ No one uses this
- High level I/O libraries
  - ♦ netCDF and HDF5
  - ♦ Better solutions than external32
    - Define a "container" for data
      - Describes contents
      - May be queried (self-describing)
    - Standard format for metadata about the file
    - Wide range of post-processing tools available

Higher Level I/O Libraries

- Scientific applications work with structured data and desire more self-describing file formats
- netCDF and HDF5 are two popular "higher level" I/O libraries
  - ♦ Abstract away details of file layout
  - ♦ Provide standard, portable file formats
  - ♦ Include metadata describing contents
- For parallel machines, these should be built on top of MPI-IO
  - ♦ HDF5 has an MPI-IO option
    - http://www.hdfgroup.org/HDF5/

netCDF Data Model

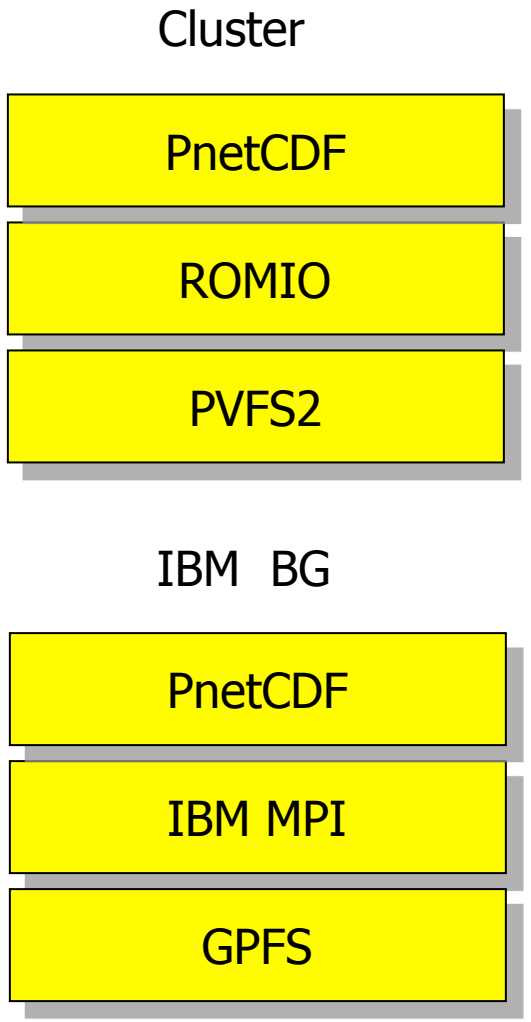
- netCDF provides a means for storing multiple, multi-dimensional arrays in a single file, along with information about the arrays



PARALLEL@ILLINOIS

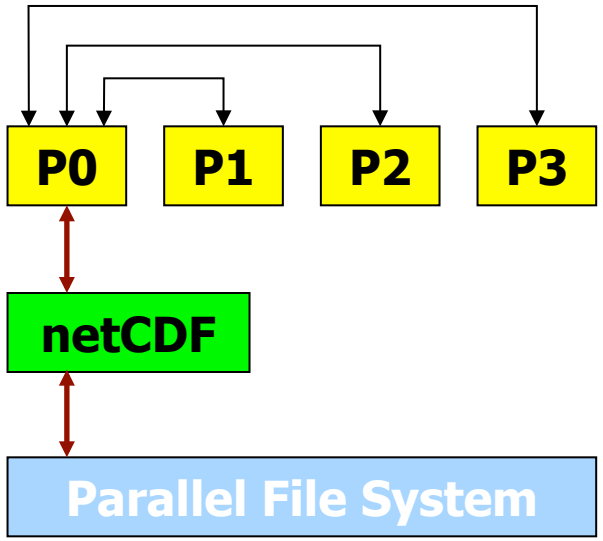
Parallel netCDF (PnetCDF)

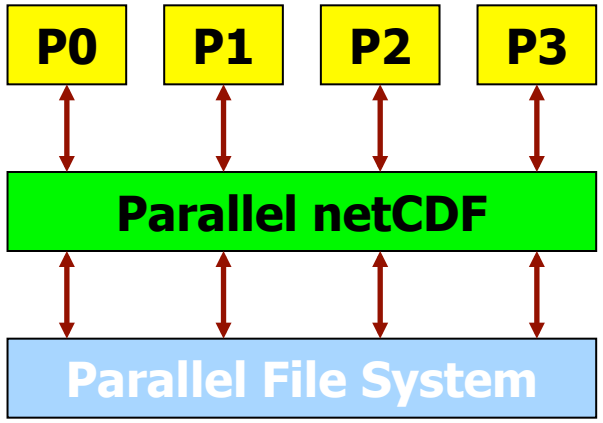
- (Serial) netCDF
  - API for accessing multi-dimensional data sets
  - Portable file format
  - Popular in both fusion and climate communities
- Parallel netCDF
  - Very similar API to netCDF
  - Tuned for better performance in today's computing environments
  - Retains the file format so netCDF and PnetCDF applications can share files
  - PnetCDF builds on top of any MPI-IO implementation



I/O in netCDF and PnetCDF

- (Serial) netCDF
  - Parallel read
    - All processes read the file independently
    - No possibility of collective optimizations
  - Sequential write
    - Parallel writes are carried out by shipping data to a single process
    - Just like our stdout checkpoint code
- PnetCDF
  - Parallel read/write to shared netCDF file
  - Built on top of MPI-IO which utilizes optimal I/O facilities of the parallel file system and MPI-IO implementation
  - Allows for MPI-IO hints and datatypes for further optimization





PARALLEL@ILLINOIS

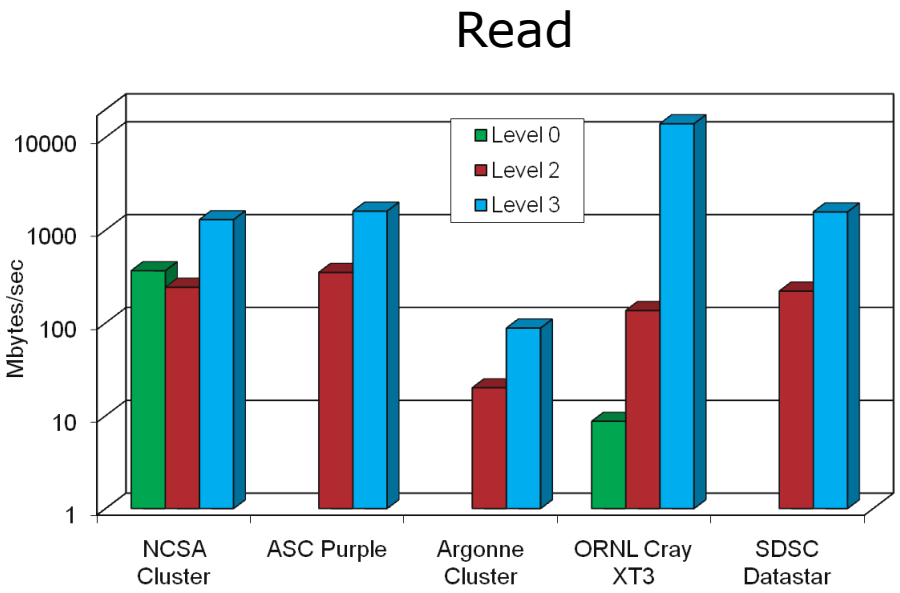
Optimizations

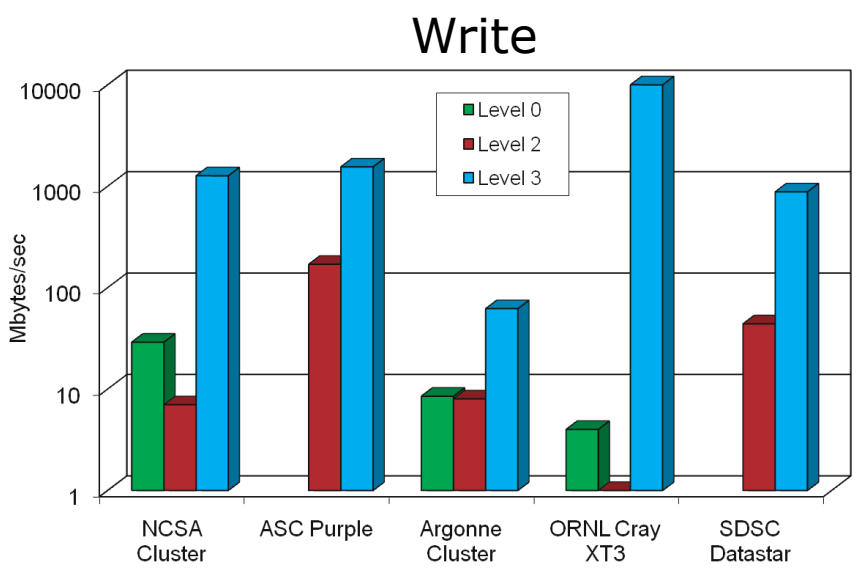
- Given complete access information, an implementation can perform optimizations such as:
  - Data Sieving: Read large chunks and extract what is really needed
  - Collective I/O: Merge requests of different processes into larger requests
  - Improved prefetching and caching

Bandwidth Results

- 3D distributed array access written as levels 0, 2, 3
- Five different machines (dated but still relevant)
  - NCSA Teragrid IA-64 cluster with GPFS and MPICH2
  - ASC Purple at LLNL with GPFS and IBM's MPI
  - Jazz cluster at Argonne with PVFS and MPICH2
  - Cray XT3 at ORNL with Lustre and Cray's MPI
  - SDSC Datastar with GPFS and IBM's MPI
- Since these are all different machines with different amounts of I/O hardware, we compare the performance of the different levels of access on a particular machine, not across machines

Distributed Array Access

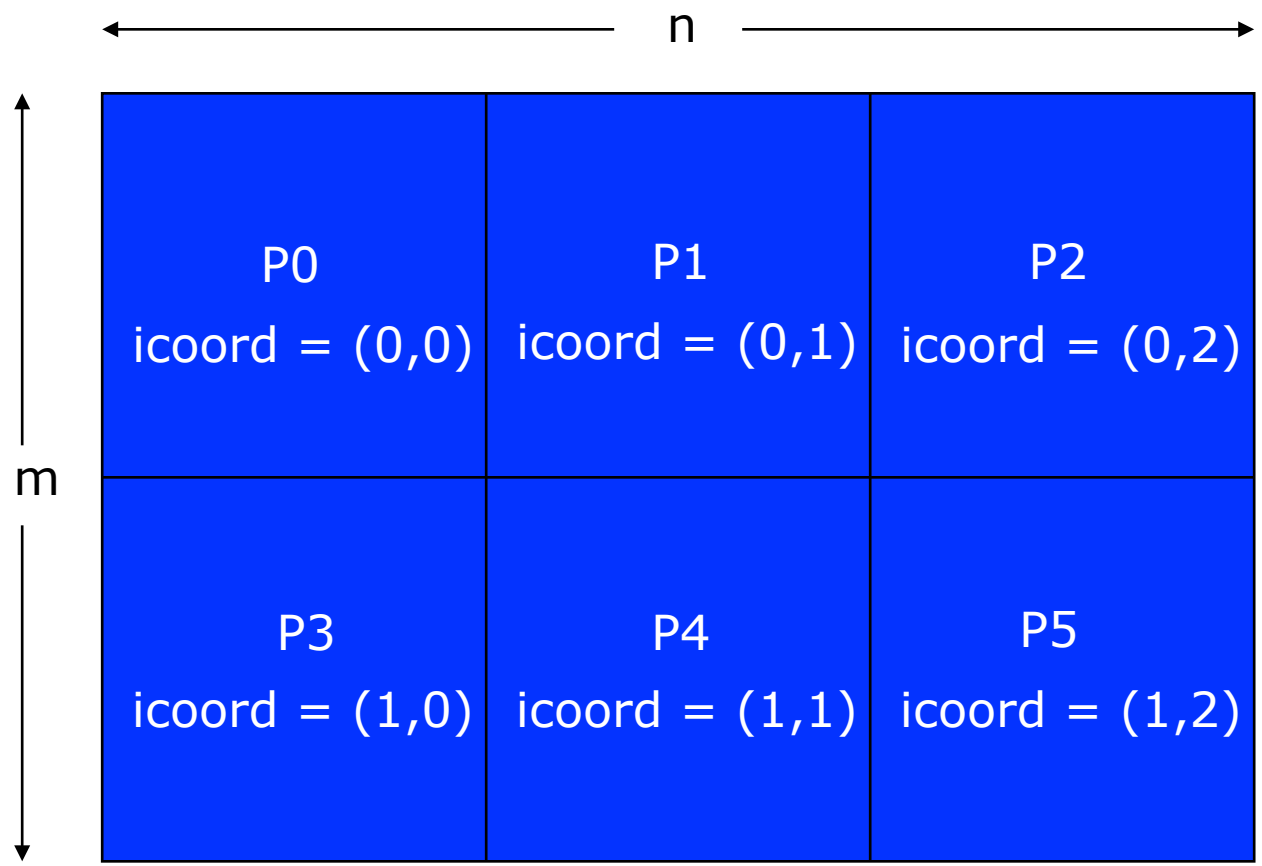




- Array size: 512 x 512 x 512
- Note log scaling!
- Thanks to Weikuan Yu, Wei-keng Liao, Bill Loewe, and Anthony Chan for these results.

Detailed Example: Distributed Array Access

- Global array size: *m* x *n*
- Array is distributed on a 2D grid of processes: *nproc(1) x nproc(2)*
- Coordinates of a process in the grid: *icoord(1), icoord(2)* (0-based indices)
- Local array stored in file in a layout corresponding to global array in column-major (Fortran) order
- Two cases
  - ♦ local array is stored contiguously in memory
  - ♦ local array has "ghost area" around it in memory

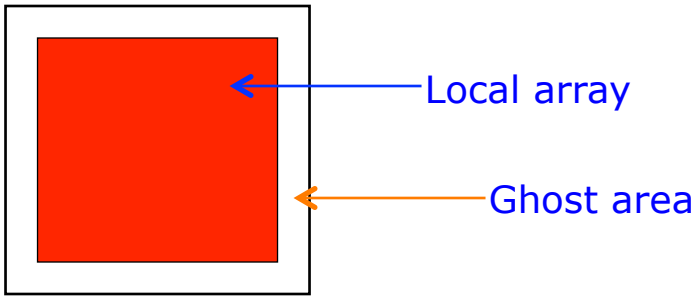


nproc(1) = 2, nproc(2) = 3

CASE I: LOCAL ARRAY CONTIGUOUS IN MEMORY

```
iarray_of_sizes(1) = m
iarray_of_sizes(2) = n
iarray_of_subsizes(1) = m/nproc(1)
iarray_of_subsizes(2) = n/nproc(2)
iarray_of_starts(1) = icoord(1) * iarray_of_subsizes(1)      ! 0-based, even in Fortran
iarray_of_starts(2) = icoord(2) * iarray_of_subsizes(2)      ! 0-based, even in Fortran
ndims = 2
zeroOffset = 0

call MPI_TYPE_CREATE_SUBARRAY(ndims, iarray_of_sizes, iarray_of_subsizes,
          iarray_of_starts, MPI_ORDER_FORTRAN, MPI_REAL, ifiletype, ierr)
call MPI_TYPE_COMMIT(ifiletype, ierr)

call MPI_FILE_OPEN(MPI_COMM_WORLD, '/home/me/test',
          MPI_MODE_CREATE + MPI_MODE_RDWR, MPI_INFO_NULL, ifh, ierr)

call MPI_FILE_SET_VIEW(ifh, zeroOffset, MPI_REAL, ifiletype, 'native', MPI_INFO_NULL, ierr)

ilocal_size = iarray_of_subsizes(1) * iarray_of_subsizes(2)
call MPI_FILE_WRITE_ALL(ifh, local_array, ilocal_size, MPI_REAL, istatus, ierr)

call MPI_FILE_CLOSE(ifh, ierr)
```

CASE II: LOCAL ARRAY HAS GHOST AREA OF SIZE 2 ON EACH SIDE




```
!  Create derived datatype describing layout of local array in memory

iarray_of_sizes(1) = m/nproc(1) + 4
iarray_of_sizes(2) = n/nproc(2) + 4
iarray_of_subsizes(1) = m/nproc(1)
iarray_of_subsizes(2) = n/nproc(2)
iarray_of_starts(1) = 2                    ! 0-based, even in Fortran
iarray_of_starts(2) = 2                    ! 0-based, even in Fortran
ndims = 2

call MPI_TYPE_CREATE_SUBARRAY(ndims, iarray_of_sizes, iarray_of_subsizes,
          iarray_of_starts, MPI_ORDER_FORTRAN, MPI_REAL, imemtype, ierr)
call MPI_TYPE_COMMIT(imemtype, ierr)
```

Continued...

CASE II continued…

```
! Create derived datatype to describe layout of local array in the file

iarray_of_sizes(1) = m
iarray_of_sizes(2) = n
iarray_of_subsizes(1) = m/nproc(1)
iarray_of_subsizes(2) = n/nproc(2)
iarray_of_starts(1) = icoord(1) * iarray_of_subsizes(1)      ! 0-based, even in Fortran
iarray_of_starts(2) = icoord(2) * iarray_of_subsizes(2)      ! 0-based, even in Fortran
ndims = 2
zeroOffset = 0

call MPI_TYPE_CREATE_SUBARRAY(ndims, iarray_of_sizes, iarray_of_subsizes,
          iarray_of_starts, MPI_ORDER_FORTRAN, MPI_REAL, ifiletype, ierr)
call MPI_TYPE_COMMIT(ifiletype, ierr)

! Open the file, set the view, and write

call MPI_FILE_OPEN(MPI_COMM_WORLD, '/home/thakur/test',
          MPI_MODE_CREATE + MPI_MODE_RDWR, MPI_INFO_NULL, ifh, ierr)
call MPI_FILE_SET_VIEW(ifh, zeroOffset, MPI_REAL, ifiletype, 'native', MPI_INFO_NULL, ierr)
call MPI_FILE_WRITE_ALL(ifh, local_array, 1, imemtype, istatus, ierr)
call MPI_FILE_CLOSE(ifh, ierr)
```

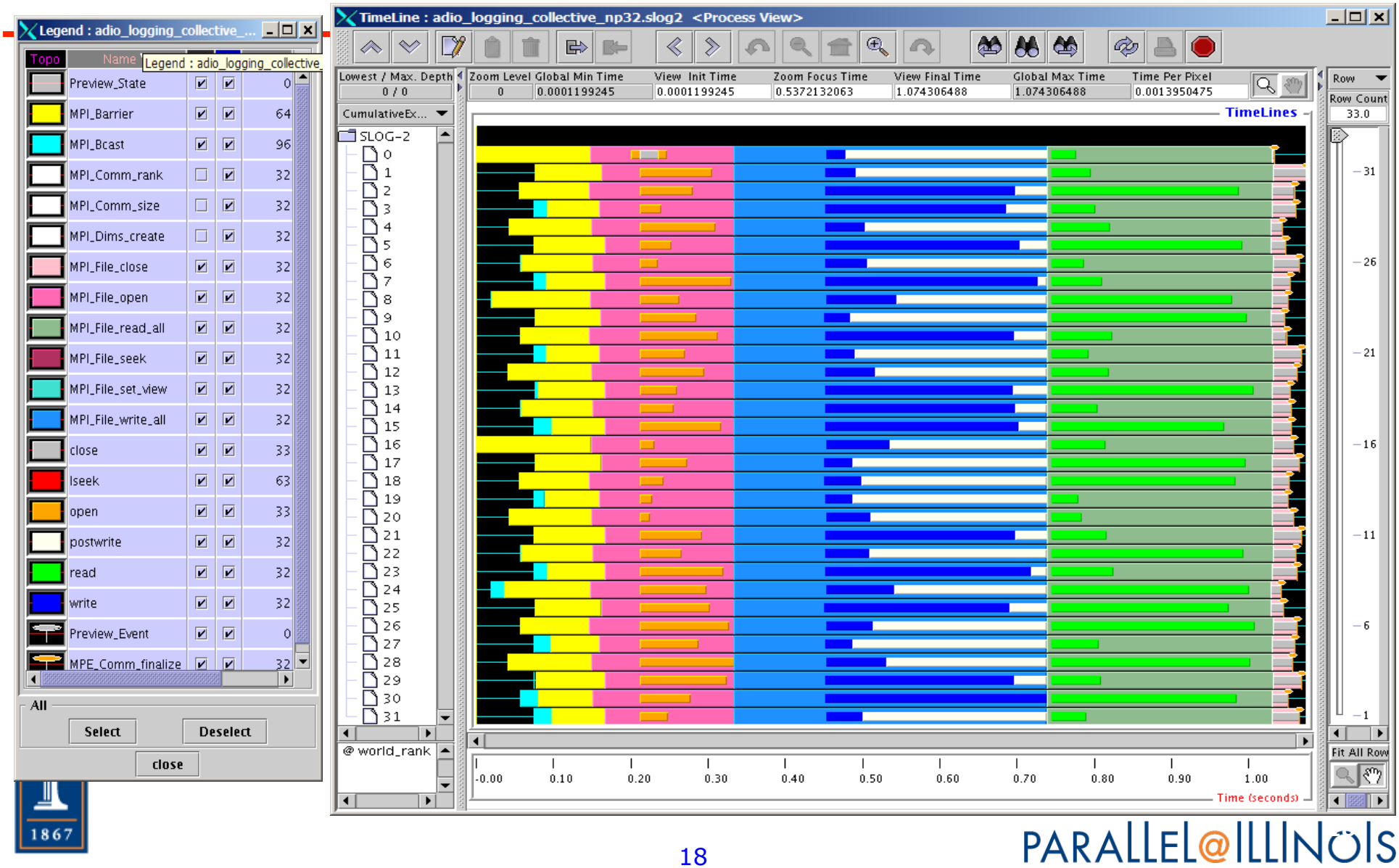
Comments On MPI I/O

- Understand why collective I/O is better, and how it enables efficient implementation of I/O
- More on tuning MPI IO performance
- Common errors in using MPI IO

Collective I/O

- The next slide shows the trace for the collective I/O case
- Note that the entire program runs for a little more than 1 sec
- Each process does its entire I/O with a single write or read operation
- Data is exchanged with other processes so that everyone gets what they need
- Very efficient!

Collective I/O

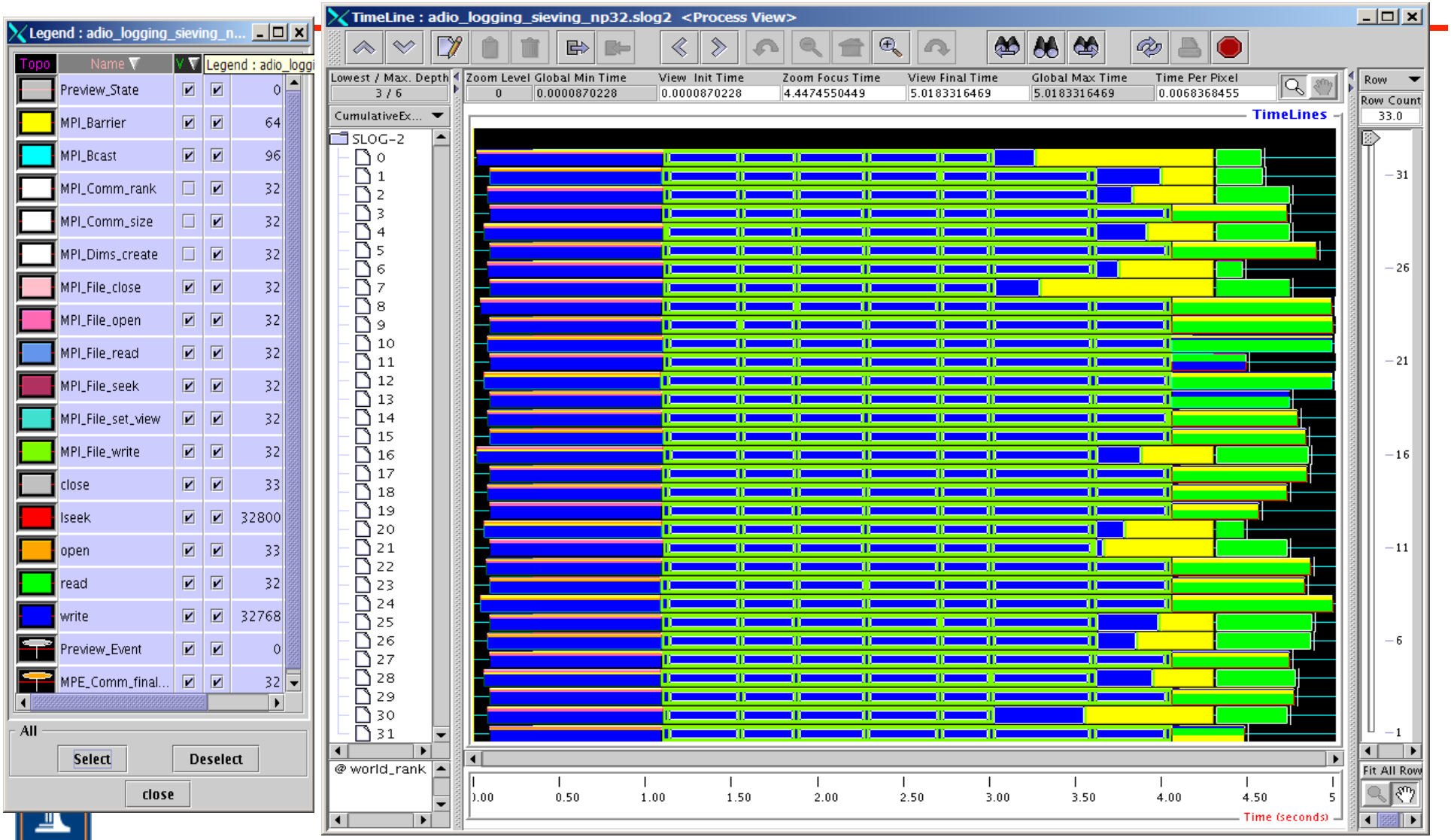


PARALLEL@ILLINOIS

Data Sieving

- The next slide shows the trace for the data sieving case
- Note that the program runs for about 5 sec now
- Since the default data sieving buffer size happens to be large enough, each process can read with a single read operation, although more data is read than actually needed (because of holes)
- Since PVFS doesn't support file locking, data sieving cannot be used for writes, resulting in many small writes (1K per process)

Data Sieving

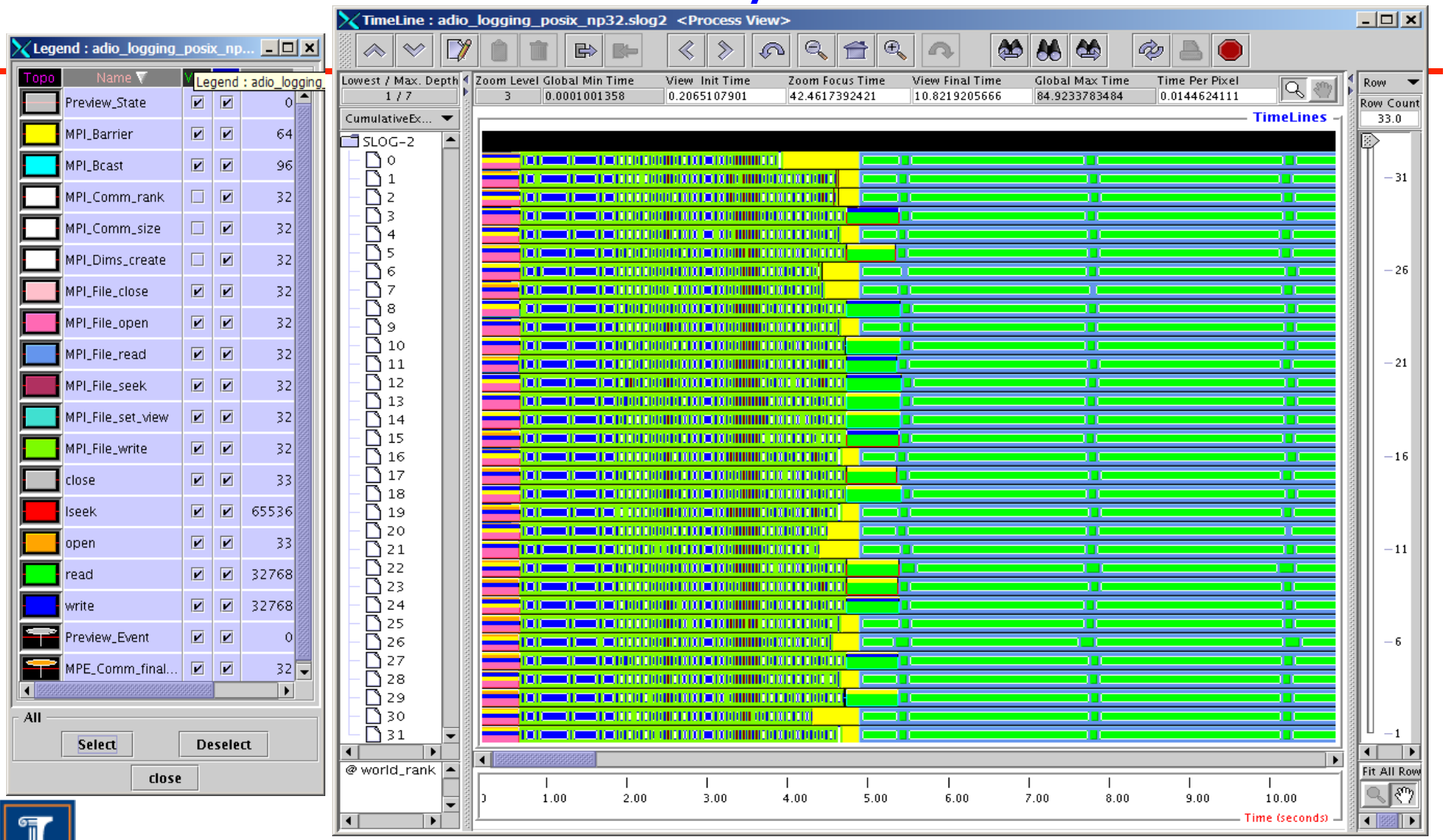


PARALLEL@ILLINOIS

Posix I/O

- The next slide shows the trace for Posix I/O
- Lots of small reads and writes (1K each per process)
- The reads take much longer than the writes in this case because of a TCP-incast problem happening in the switch
- Total program takes about 80 sec
- Very inefficient!

Posix I/O



PARALLEL@ILLINOIS

Passing Hints

- MPI defines MPI_Info
- Provides an extensible list of key=value pairs
- Used to package variable, optional types of arguments that may not be standard
  - Used in IO, Dynamic, and RMA, as well as with communicators

Passing Hints to MPI-IO

```
MPI_Info info;

MPI_Info_create(&info);

/* no. of I/O devices to be used for file striping */
MPI_Info_set(info, "striping_factor", "4");

/* the striping unit in bytes */
MPI_Info_set(info, "striping_unit", "65536");

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_RDWR, info, &fh);

MPI_Info_free(&info);
```

Passing Hints to MPI-IO (Fortran)

```
integer info

call MPI_Info_create(info, ierr)

! no. of I/O devices to be used for file striping
call MPI_Info_set(info, "striping_factor", "4", ierr )

! the striping unit in bytes
call MPI_Info_set(info, "striping_unit", "65536", ierr )

call MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", &
            MPI_MODE_CREATE + MPI_MODE_RDWR, info, &
               fh, ierr )

call MPI_Info_free( info, ierr )
```

Examples of Hints (used in ROMIO)

- `striping_unit`
- `striping_factor`
- `cb_buffer_size`

MPI predefined hints

- `cb_nodes`
- `ind_rd_buffer_size`
- `ind_wr_buffer_size`
- `start_iodevice`

New Algorithm Parameters

- `pfs_svr_buf`
- `direct_read`
- `direct_write`

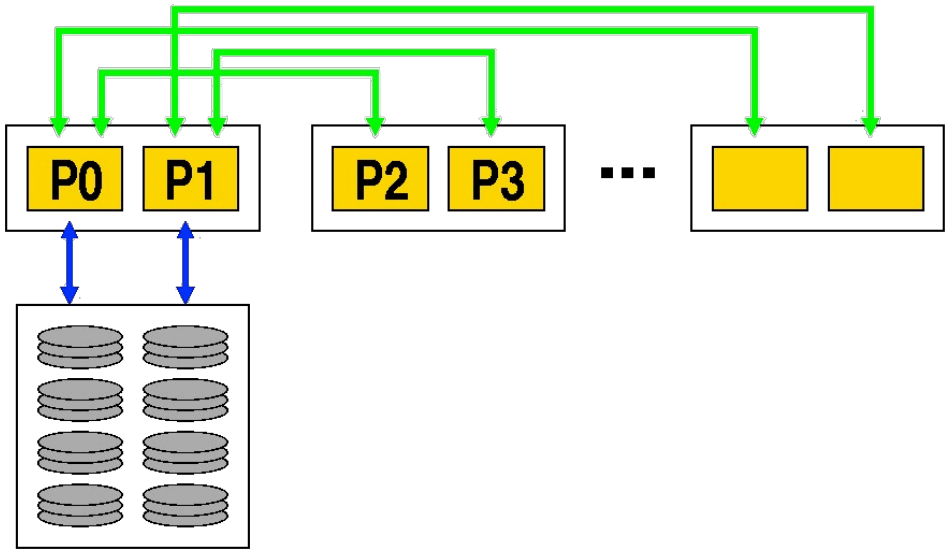
Platform-specific hints

Common MPI I/O Hints

- Controlling parallel file system
  - **`striping_factor`** - size of "strips" on I/O servers
  - **`striping_unit`** - number of I/O servers to stripe across
  - **`start_iodevice`** - which I/O server to start with
- Controlling aggregation
  - **`cb_config_list`** - list of aggregators
  - **`cb_nodes`** - number of aggregators (upper bound)
- Tuning ROMIO (most common MPI-IO implementation) optimizations
  - **`romio_cb_read`**, **`romio_cb_write`** - aggregation on/off
  - **`romio_ds_read`**, **`romio_ds_write`** - data sieving on/off

Aggregation Example

- Cluster of SMPs
- One SMP box has fast connection to disks
- Data is aggregated to processes on single box
- Processes on that box perform I/O on behalf of the others



Ensuring Parallel I/O is Parallel

- On Blue Waters, by default, files are not striped
  - That is, there is no parallelism in distributing the file across multiple disks
- You can set the striping factor on a directory; inherited by all files created in that directory
  - lfs setstripe -c 4 <directory-name>
- You can set the striping factor on a file when it is created with MPI_File_open, using the hints, e.g., set
  - MPI_Info_set(info, "striping_factor", "4");
  - MPI_Info_set(info, "cb_nodes", "4");

Finding Hints on Blue Waters

- Setting the environment variable MPICH_MPIIO_HINTS_DISPLAY=1 causes the program to print out the available I/O hints and their values.

Example of Hints Display

```
PE 0: MPICH/MPIIO environment settings:
PE 0:   MPICH_MPIIO_HINTS_DISPLAY  = 1
PE 0:   MPICH_MPIIO_HINTS          = NULL
PE 0:
MPICH_MPIIO_ABORT_ON_RW_ERROR =
disable
PE 0:   MPICH_MPIIO_CB_ALIGN       = 2
PE 0:   MPIIO hints for ioperf.out.tfaRGQ:
        cb_buffer_size          = 16777216
        romio_cb_read           = automatic
        romio_cb_write          = automatic
        cb_nodes                = 1
        cb_align                = 2
        romio_no_indep_rw       = false
        romio_cb_pfr            = disable
        romio_cb_fr_types       = aar
        romio_cb_fr_alignment   = 1
        romio_cb_ds_threshold   = 0
        romio_cb_alltoall       = automatic
        ind_rd_buffer_size      = 4194304
        ind_wr_buffer_size      = 524288
        romio_ds_read           = disable
        romio_ds_write          = disable
        striping_factor         = 1
        striping_unit           = 1048576
        romio_lustre_start_iodevice = 0
        direct_io               = false
        aggregator_placement_stride = -1
        abort_on_rw_error       = disable
        cb_config_list          = *:*
```

Summary of I/O Tuning

- MPI I/O has many features that can help users achieve high performance
- The most important of these features are the ability to specify noncontiguous accesses, the collective I/O functions, and the ability to pass hints to the implementation
- Users must use the above features!
- In particular, when accesses are noncontiguous, users must create derived datatypes, define file views, and use the collective I/O functions

Common Errors in Using MPI-IO

- Not defining file offsets as MPI_Offset in C and integer (kind=MPI_OFFSET_KIND) in Fortran (or perhaps integer*8 in Fortran 77)
- In Fortran, passing the offset or displacement directly as a constant (e.g., 0) in the absence of function prototypes (F90 mpi module)
- Using darray datatype for a block distribution other than the one defined in darray (e.g., floor division)
- filetype defined using offsets that are not monotonically nondecreasing, e.g., 0, 3, 8, 4, 6. (can occur in irregular applications)