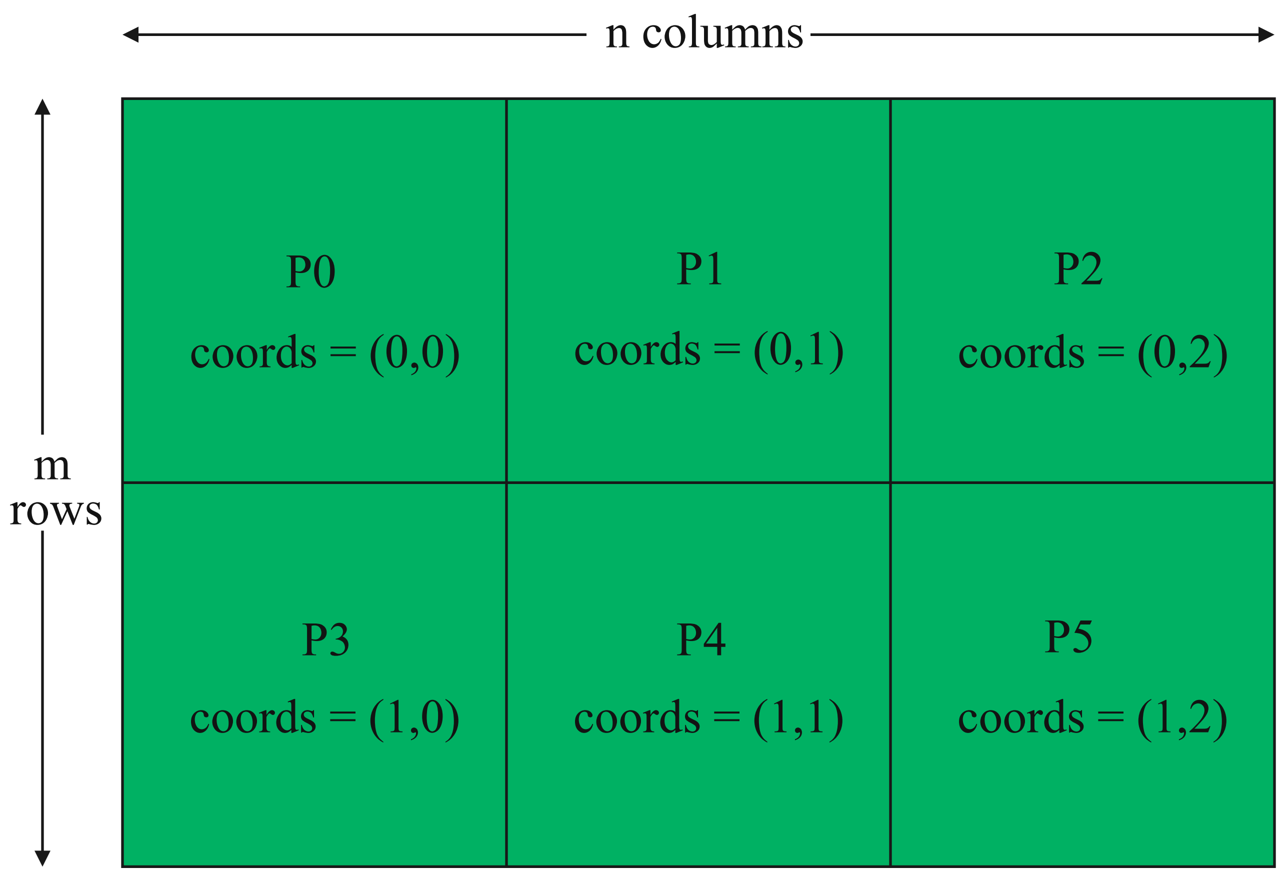
Parallel I/O and MPI-IO contd.

Rajeev Thakur

Outline

- Accessing noncontiguous data with MPI-IO
- Special features in MPI-IO for accessing subarrays and distributed arrays
- I/O performance tuning

Accessing Arrays Stored in Files



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

Using the "Distributed Array" (Darray) Datatype

```
int gsizes[2], distribs[2], dargs[2], psizes[2];

gsizes[0] = m;    /* no. of rows in global array */
gsizes[1] = n;    /* no. of columns in global array*/

distribs[0] = MPI_DISTRIBUTE_BLOCK;
distribs[1] = MPI_DISTRIBUTE_BLOCK;

dargs[0] = MPI_DISTRIBUTE_DFLT_DARG;
dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;

psizes[0] = 2; /* no. of processes in vertical dimension
                  of process grid */
psizes[1] = 3; /* no. of processes in horizontal dimension
                  of process grid */
```

Darray Continued

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Type_create_darray(6, rank, 2, gsizes, distribs, dargs,
               psizes, MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &fh);

MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
                  MPI_INFO_NULL);

local_array_size = num_local_rows * num_local_cols;
MPI_File_write_all(fh, local_array, local_array_size,
                MPI_FLOAT, &status);

MPI_File_close(&fh);
```

A Word of Warning about Darray

- The darray datatype assumes a very specific definition of data distribution -- the exact definition as in HPF
- For example, if the array size is not divisible by the number of processes, darray calculates the block size using a *ceiling* division (20 / 6 = 4)
- darray assumes a row-major ordering of processes in the logical grid, as assumed by cartesian process topologies in MPI-1
- If your application uses a different definition for data distribution or logical grid ordering, you cannot use darray. Use subarray instead.

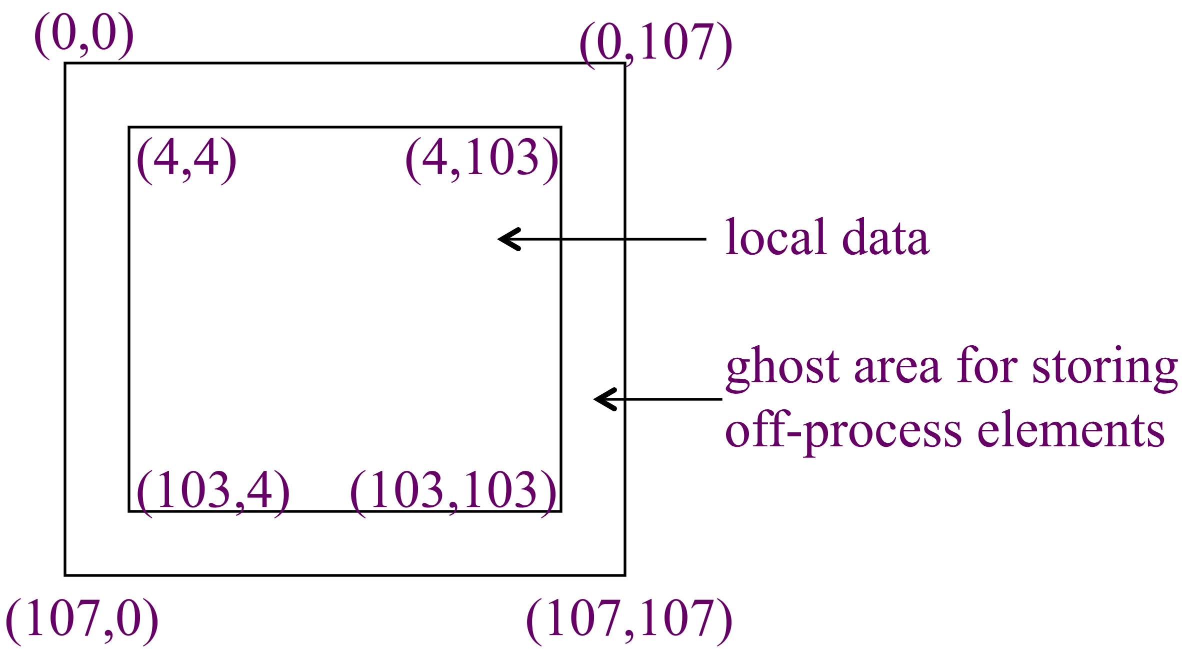
Using the Subarray Datatype

```
gsizes[0] = m;  /* no. of rows in global array */
gsizes[1] = n;  /* no. of columns in global array*/

psizes[0] = 2; /* no. of procs. in vertical dimension */
psizes[1] = 3; /* no. of procs. in horizontal dimension */

lsizes[0] = m/psizes[0]; /* no. of rows in local array */
lsizes[1] = n/psizes[1]; /* no. of columns in local array */

dims[0] = 2; dims[1] = 3;
periods[0] = periods[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
MPI_Comm_rank(comm, &rank);
MPI_Cart_coords(comm, rank, 2, coords);
```

Subarray Datatype contd.

```
/* global indices of first element of local array */
start_indices[0] = coords[0] * lsizes[0];
start_indices[1] = coords[1] * lsizes[1];

MPI_Type_create_subarray(2, gsizes, lsizes, start_indices,
                         MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
               MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
                MPI_INFO_NULL);
local_array_size = lsizes[0] * lsizes[1];
MPI_File_write_all(fh, local_array, local_array_size,
                 MPI_FLOAT, &status);
```

Local Array with Ghost Area in Memory

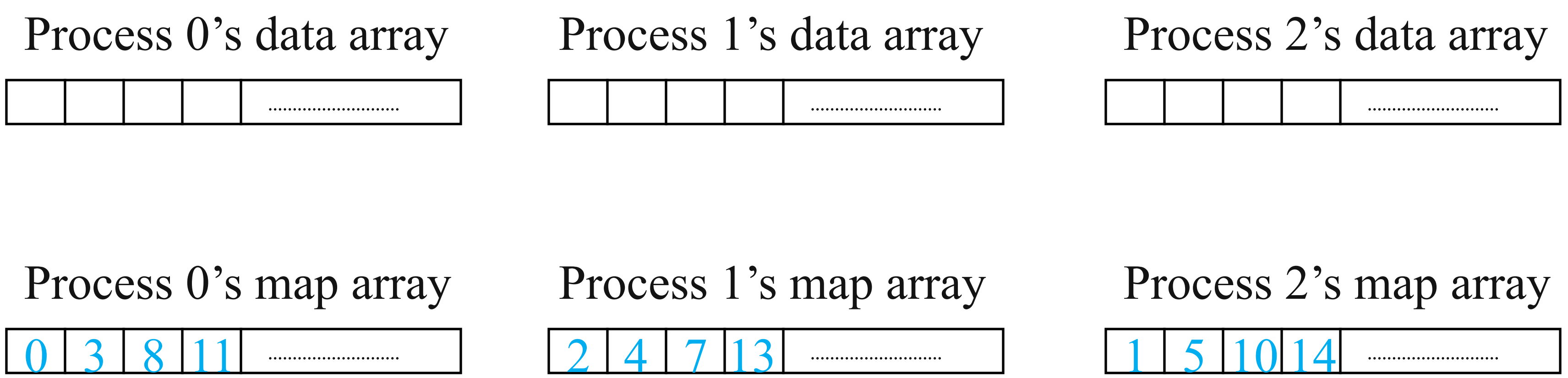


- Use a subarray datatype to describe the noncontiguous layout in memory
- Pass this datatype as argument to **`MPI_File_write_all`**

Local Array with Ghost Area

```
memsizes[0] = lsizes[0] + 8;
   /* no. of rows in allocated array */
memsizes[1] = lsizes[1] + 8;
   /* no. of columns in allocated array */
start_indices[0] = start_indices[1] = 4;
   /* indices of the first element of the local array
      in the allocated array */

MPI_Type_create_subarray(2, memsizes, lsizes,
         start_indices, MPI_ORDER_C, MPI_FLOAT, &memtype);
MPI_Type_commit(&memtype);

/* create filetype and set file view exactly as in the
   subarray example */

MPI_File_write_all(fh, local_array, 1, memtype, &status);
```

Accessing Irregularly Distributed Arrays



The map array describes the location of each element of the data array in the common file

Accessing Irregularly Distributed Arrays

```
integer (kind=MPI_OFFSET_KIND) disp

call MPI_FILE_OPEN(MPI_COMM_WORLD, '/pfs/datafile', &
                   MPI_MODE_CREATE + MPI_MODE_RDWR, &
                   MPI_INFO_NULL, fh, ierr)

call MPI_TYPE_CREATE_INDEXED_BLOCK(bufsize, 1, map, &
                    MPI_DOUBLE_PRECISION, filetype, ierr)
call MPI_TYPE_COMMIT(filetype, ierr)
disp = 0
call MPI_FILE_SET_VIEW(fh, disp, MPI_DOUBLE_PRECISION, &
                 filetype, 'native', MPI_INFO_NULL, ierr)

call MPI_FILE_WRITE_ALL(fh, buf, bufsize, &
                      MPI_DOUBLE_PRECISION, status, ierr)

call MPI_FILE_CLOSE(fh, ierr)
```

Nonblocking I/O

```
MPI_Request request;
MPI_Status status;

MPI_File_iwrite_at(fh, offset, buf, count, datatype,
                   &request);

for (i=0; i<1000; i++) {
    /* perform computation */
}

MPI_Wait(&request, &status);
```

Split Collective I/O

- A restricted form of nonblocking collective I/O
- Only one active nonblocking collective operation allowed at a time on a file handle
- Therefore, no request object necessary

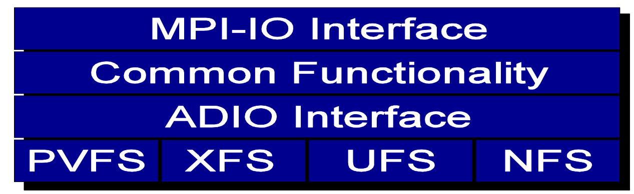
```
MPI_File_write_all_begin(fh, buf, count, datatype);

for (i=0; i<1000; i++) {
    /* perform computation */
}

MPI_File_write_all_end(fh, buf, &status);
```

MPI-IO Implementations

- There are a collection of different MPI-IO implementations
- Each one has its own set of special features
- Three better-known ones are:
  - ROMIO from Argonne National Laboratory
    - *Included in many MPI implementations (MPICH2, Open MPI, vendor MPIs)*
  - MPI-IO/GPFS from IBM
  - MPI/SX and MPI/PC-32 from NEC
    - *originally derived from ROMIO*
- Quick overview of these…

ROMIO MPI-IO Implementation

- ANL implementation
- Leverages MPI communication
- Layered implementation supports many storage types
  - Local file systems (e.g. XFS)
  - Parallel file systems (e.g. PVFS2)
  - NFS, Remote I/O (RFS)
- UFS implementation works for most other file systems
  - e.g. GPFS and Lustre
- Included with many MPI implementations
- Includes data sieving and two-phase optimizations



IBM MPI-IO Implementation

- For GPFS on the AIX platform
- Includes two special optimizations
  - Data shipping -- mechanism for coordinating access to a file to alleviate lock contention (type of aggregation)
  - Controlled prefetching -- using MPI file views and access patterns to predict regions to be accessed in future
- Not available for GPFS on Linux
  - Use ROMIO instead

NEC MPI-IO Implementation

- For NEC SX platform (MPI/SX) and Myrinet-coupled PC clusters (MPI/PC-32)
- Includes *listless I/O* optimization
  - Fast handling of noncontiguous I/O accesses in MPI layer
  - Great for situations where the file system is lock based and/or has only contiguous I/O primitives

Tuning MPI-IO

General Guidelines for Achieving High I/O Performance

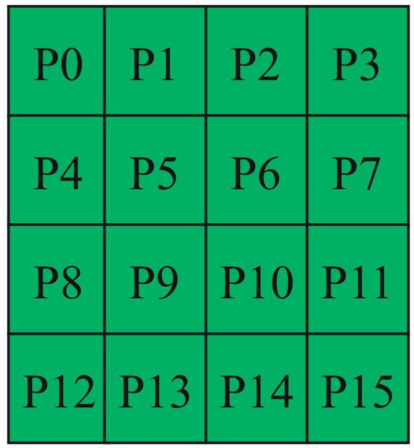
- Buy sufficient I/O hardware for the machine
- Use fast file systems, not NFS-mounted home directories
- Do not perform I/O from one process only
- Make large requests wherever possible
- For noncontiguous requests, use derived datatypes and a single collective I/O call

Using the Right MPI-IO Function

- Any application as a particular “I/O access pattern” based on its I/O needs
- The same access pattern can be presented to the I/O system in different ways depending on what I/O functions are used and how
- We classify the different ways of expressing I/O access patterns in MPI-IO into four *levels*: level 0 -- level 3
- We demonstrate how the user’s choice of *level* affects performance

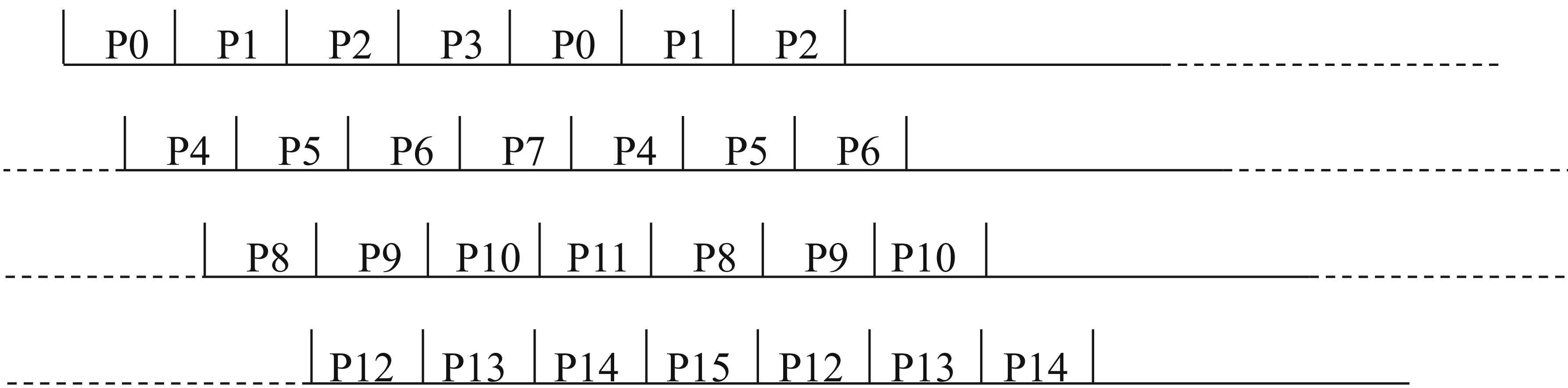
Example: Distributed Array Access

Large array distributed among 16 processes



Each square represents a subarray in the memory of a single process

Access Pattern in the file



Level-0 Access

- Each process makes one independent read request for each row in the local array (as in Unix)

```
MPI_File_open(..., file, ..., &fh)
for (i=0; i<n_local_rows; i++) {
    MPI_File_seek(fh, ...);
    MPI_File_read(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

Level-1 Access

- Similar to level 0, but each process uses collective I/O functions

```
MPI_File_open(MPI_COMM_WORLD, file, ..., &fh);
for (i=0; i<n_local_rows; i++) {
    MPI_File_seek(fh, ...);
    MPI_File_read_all(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

Level-2 Access

- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

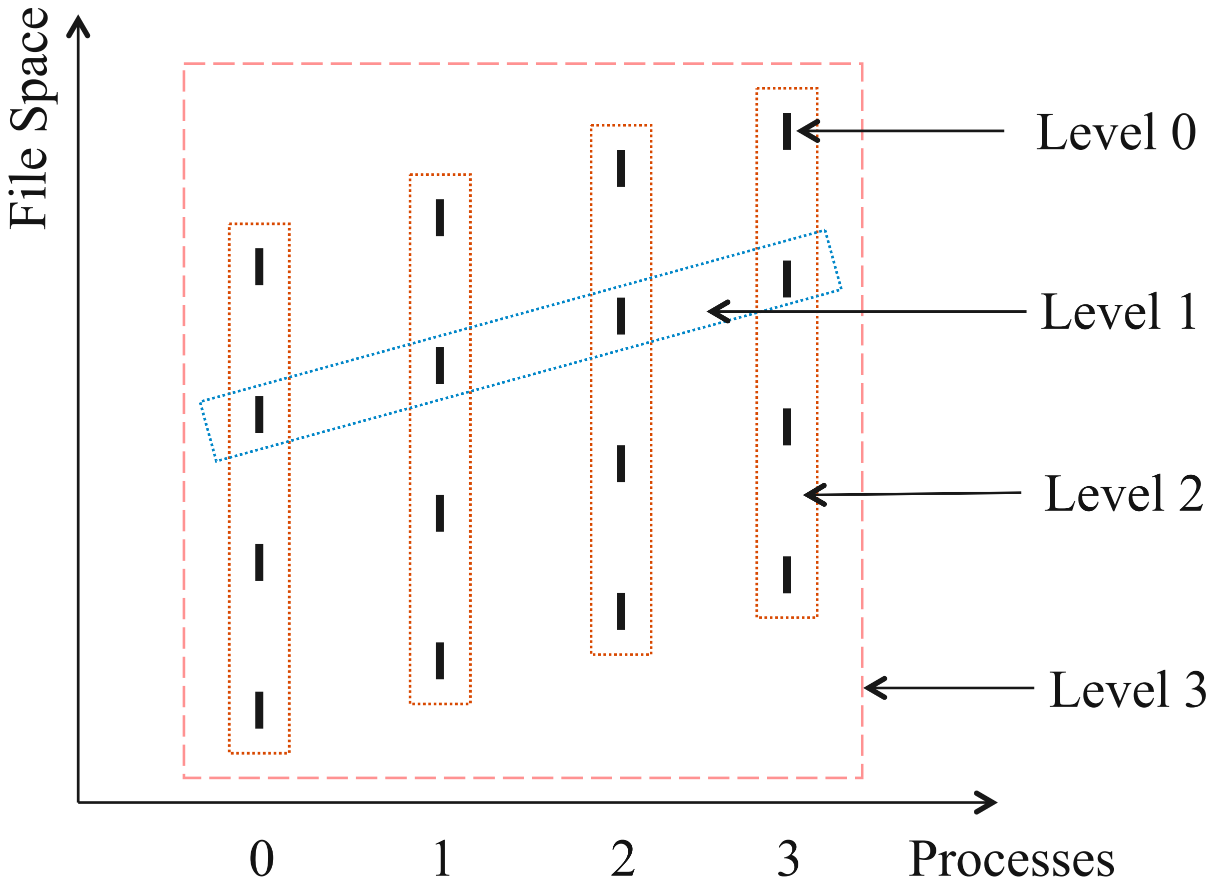
```
MPI_Type_create_subarray(..., &subarray, ...);
MPI_Type_commit(&subarray);
MPI_File_open(..., file, ..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read(fh, A, ...);
MPI_File_close(&fh);
```

Level-3 Access

- Similar to level 2, except that each process uses collective I/O functions

```
MPI_Type_create_subarray(..., &subarray, ...);
MPI_Type_commit(&subarray);
MPI_File_open(MPI_COMM_WORLD, file, ..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read_all(fh, A, ...);
MPI_File_close(&fh);
```

The Four Levels of Access



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

Two Key Optimizations in ROMIO

- Data sieving
  - For independent noncontiguous requests
  - ROMIO makes large I/O requests to the file system and, in memory, extracts the data requested
  - For writing, a read-modify-write is required
- Two-phase collective I/O
  - Communication phase to merge data into large chunks
  - I/O phase to write large chunks in parallel

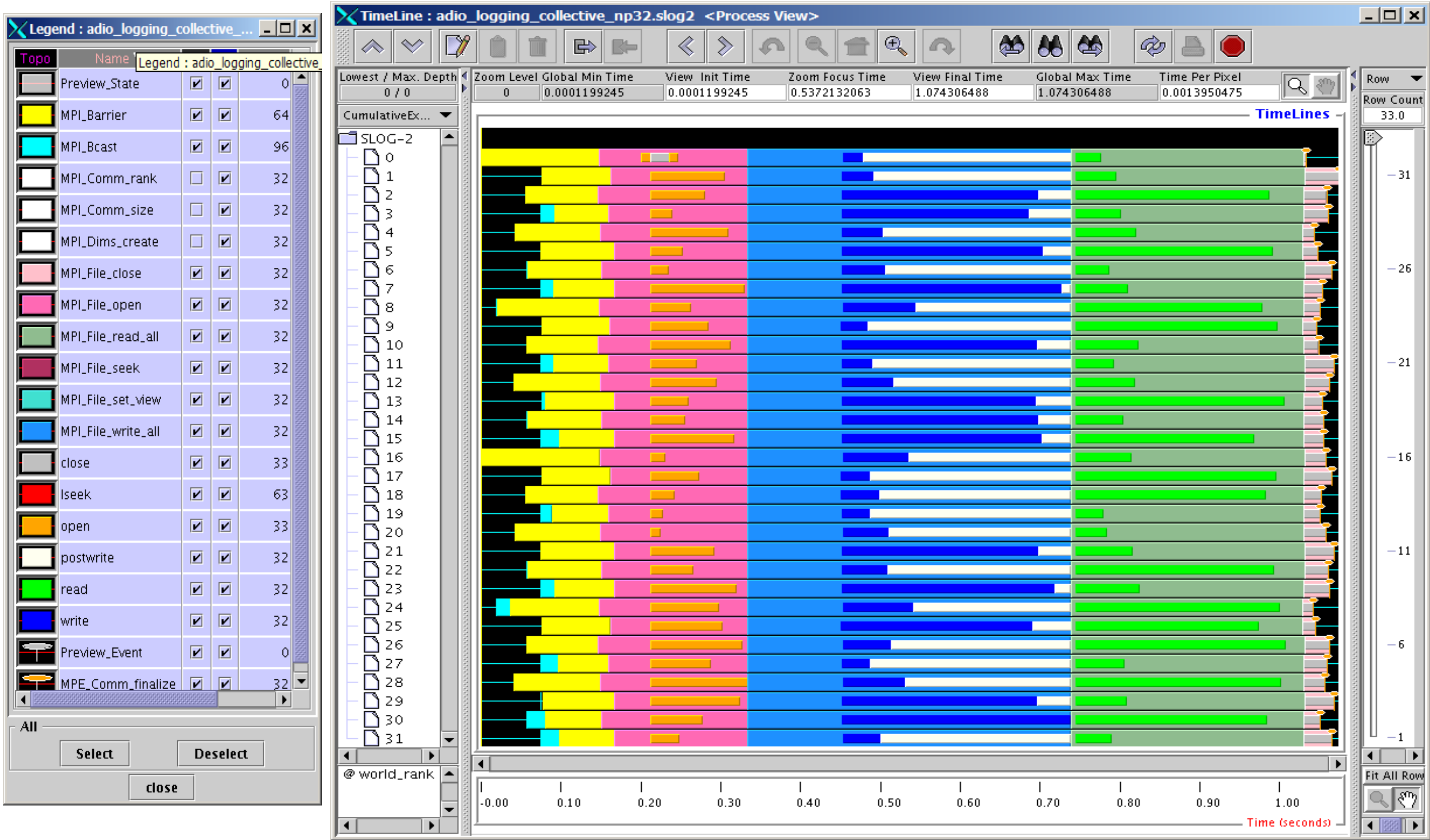
Performance Instrumentation

- We instrumented the source code of our MPI-IO implementation (ROMIO) to log various events (using the MPE toolkit from MPICH2)
- We ran a simple 3D distributed array access code written in three ways:
  - posix (level 0)
  - data sieving (level 2)
  - collective I/O (level 3)
- The code was run on 32 nodes of the Jazz cluster at Argonne with PVFS-1 as the file system
- We collected the trace files and visualized them with Jumpshot

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!

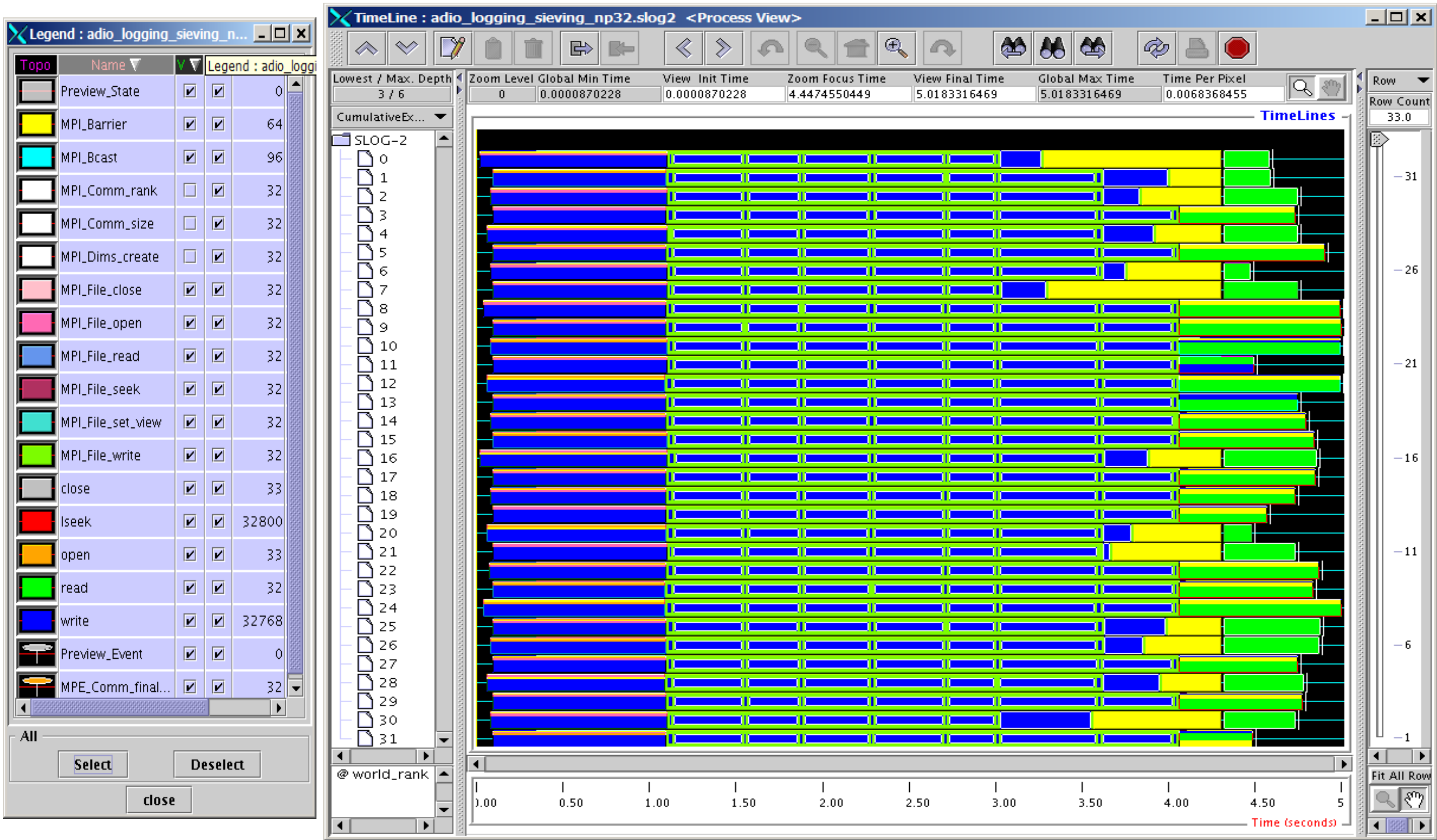
Collective I/O



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)

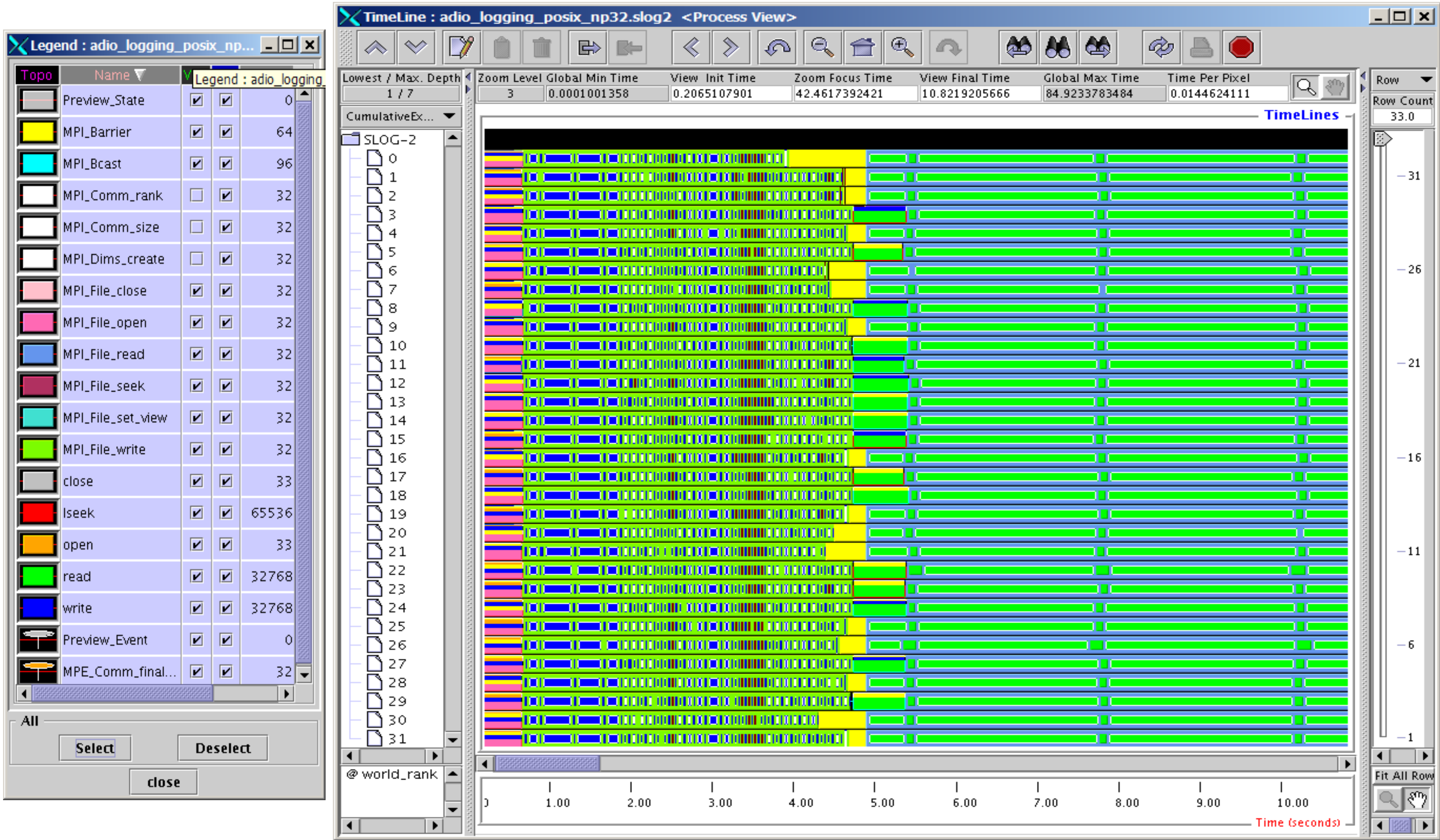
Data Sieving



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



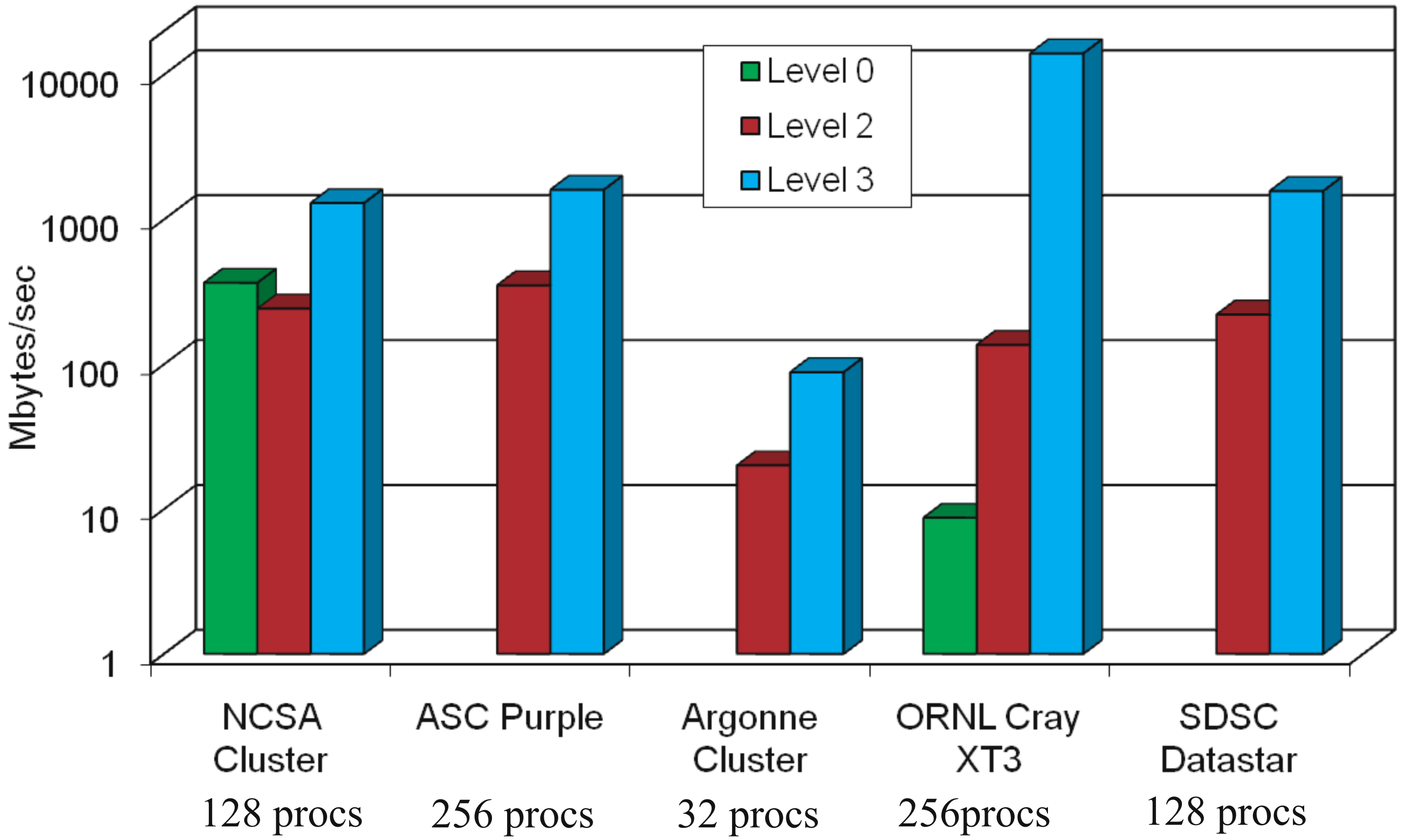
Legend : adio_logging_posix_np...
Topo
Name
Preview_State 0
MPI_Barrier 64
MPI_Bcast 96
MPI_Comm_rank 32
MPI_Comm_size 32
MPI_Dims_create 32
MPI_File_close 32
MPI_File_open 32
MPI_File_read 32
MPI_File_seek 32
MPI_File_set_view 32
MPI_File_write 32
close 33
lseek 65536
open 33
read 32768
write 32768
Preview_Event 0
MPE_Comm_final... 32
All
Select
Deselect
close
TimeLine : adio_logging_posix_np32.slog2 <Process View>
Lowest / Max. Depth
1 / 7
Zoom Level
3
Global Min Time
0.0001001358
View Init Time
0.2065107901
Zoom Focus Time
42.4617392421
View Final Time
10.8219205666
Global Max Time
84.9233783484
Time Per Pixel
0.0144624111
Row
Row Count
33.0
CumulativeEx...
SLOG-2
TimeLines
@ world_rank
Time (seconds)
Fit All Row

Bandwidth Results

- 3D distributed array access written as levels 0, 2, 3
- Five different machines
  - 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*

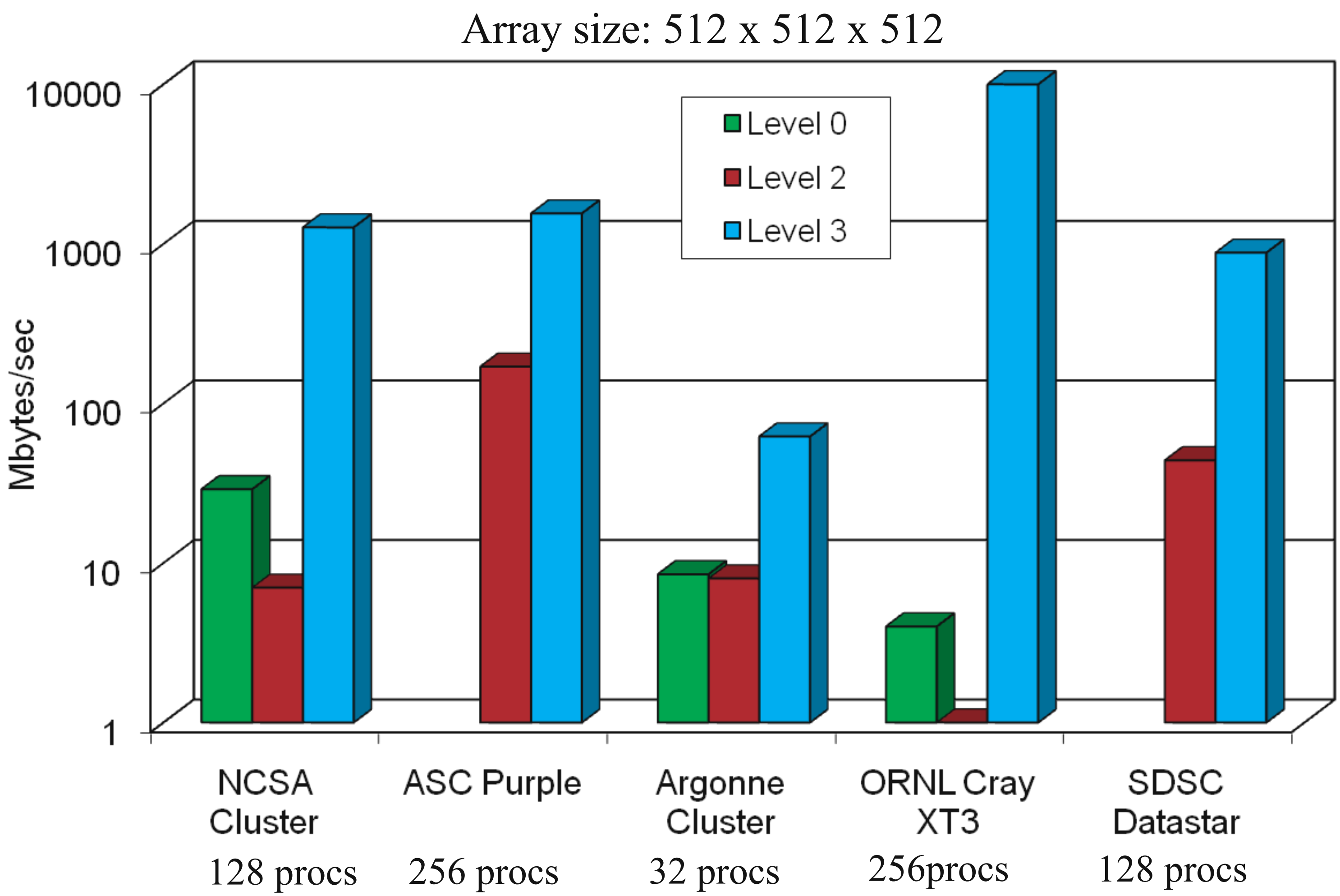
Distributed Array Access: Read Bandwidth

Array size: 512 x 512 x 512



Thanks to Weikuan Yu, Wei-keng Liao, Bill Loewe, and Anthony Chan for these results.

Distributed Array Access: Write Bandwidth



Thanks to Weikuan Yu, Wei-keng Liao, Bill Loewe, and Anthony Chan for these results.

Passing Hints

- MPI-2 defines a new object, **`MPI_Info`**
- Provides an extensible list of key=value pairs
- Used in I/O, One-sided, and Dynamic to package variable, optional types of arguments that may not be standard

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);
```

MPI-IO Hints

- MPI-IO hints may be passed via:
  - **`MPI_File_open`**
  - **`MPI_File_set_info`**
  - **`MPI_File_set_view`**
- Hints are optional - implementations are guaranteed to ignore ones they do not understand
  - Different implementations, even different underlying file systems, support different hints
- **`MPI_File_get_info`** used to get list of hints
- Next few slides cover only some hints

Examples of Hints (used in ROMIO)

- **`striping_unit`**
- **`striping_factor`**
- **`cb_buffer_size`**
- **`cb_nodes`**
- **`ind_rd_buffer_size`**
- **`ind_wr_buffer_size`**
- **`start_iodevice`**
- **`pfs_svr_buf`**
- **`direct_read`**
- **`direct_write`**

MPI-2 predefined hints

New Algorithm Parameters

Platform-specific hints

MPI-IO Hints: FS-Related

- **`striping_factor`** -- Controls the number of I/O devices to stripe across
- **`striping_unit`** -- Controls the striping unit (in bytes)
- **`start_iodevice`** -- Determines what I/O device data will first be written to
- **`direct_read`** -- Controls direct I/O for reads
- **`direct_write`** -- Controls direct I/O for writes

MPI-IO Hints: Data Sieving

- **`ind_rd_buffer_size`** -- Controls the size (in bytes) of the intermediate buffer used by ROMIO when performing data sieving reads
- **`ind_wr_buffer_size`** -- Controls the size (in bytes) of the intermediate buffer used by ROMIO when performing data sieving writes
- **`romio_ds_read`** -- Determines when ROMIO will choose to perform data sieving for reads (enable, disable, auto)
- **`romio_ds_write`** -- Determines when ROMIO will choose to perform data sieving for writes

MPI-IO Hints: Collective I/O

- **`cb_buffer_size`** -- Controls the size (in bytes) of the intermediate buffer used in two-phase collective I/O
- **`cb_nodes`** -- Controls the maximum number of aggregators to be used
- **`romio_cb_read`** -- Controls when collective buffering is applied to collective read operations
- **`romio_cb_write`** -- Controls when collective buffering is applied to collective write operations
- **`cb_config_list`** -- Provides explicit control over aggregators (see ROMIO User's Guide)

ROMIO Hints and PVFS

- Controlling PVFS
  - **`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 optimizations
  - **`romio_cb_read, romio_cb_write`** - aggregation on/off
  - **`romio_ds_read, romio_ds_write`** - data sieving on/off

File Interoperability

- Users can optionally create files with a portable binary data representation
- "datarep" parameter to **`MPI_File_set_view`**
- **`native`** - default, same as in memory, not portable
- **`internal`** - implementation defined representation providing an implementation defined level of portability
- **`external32`** - a specific representation defined in MPI, (basically 32-bit big-endian IEEE format), portable across machines and MPI implementations

Common Errors

- 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. (happens in irregular applications)

Summary

- MPI-IO 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

Hands-on Exercise

- Write a program to write data from multiple processes to different parts of a shared file. Read it back to verify it is correct.
  - Using independent I/O
  - Using collective I/O
- Do the same using file views