Parallel I/O

Advanced Topics Spring 2009

Prof. Robert van Engelen

Overview

- The parallel I/O bottleneck
- Why is traditional UNIX-style file access insufficient?
- High-performance parallel file access
- The MPI-IO library

High-Performance I/O Needs

- Parallel scientific applications typically run a set of processes that need to coordinate …
  - Reading data from a single file
  - Write output to a single file
  - Checkpoint during computations
  - Perform out of core computations on data passed via files

Need for Parallel I/O

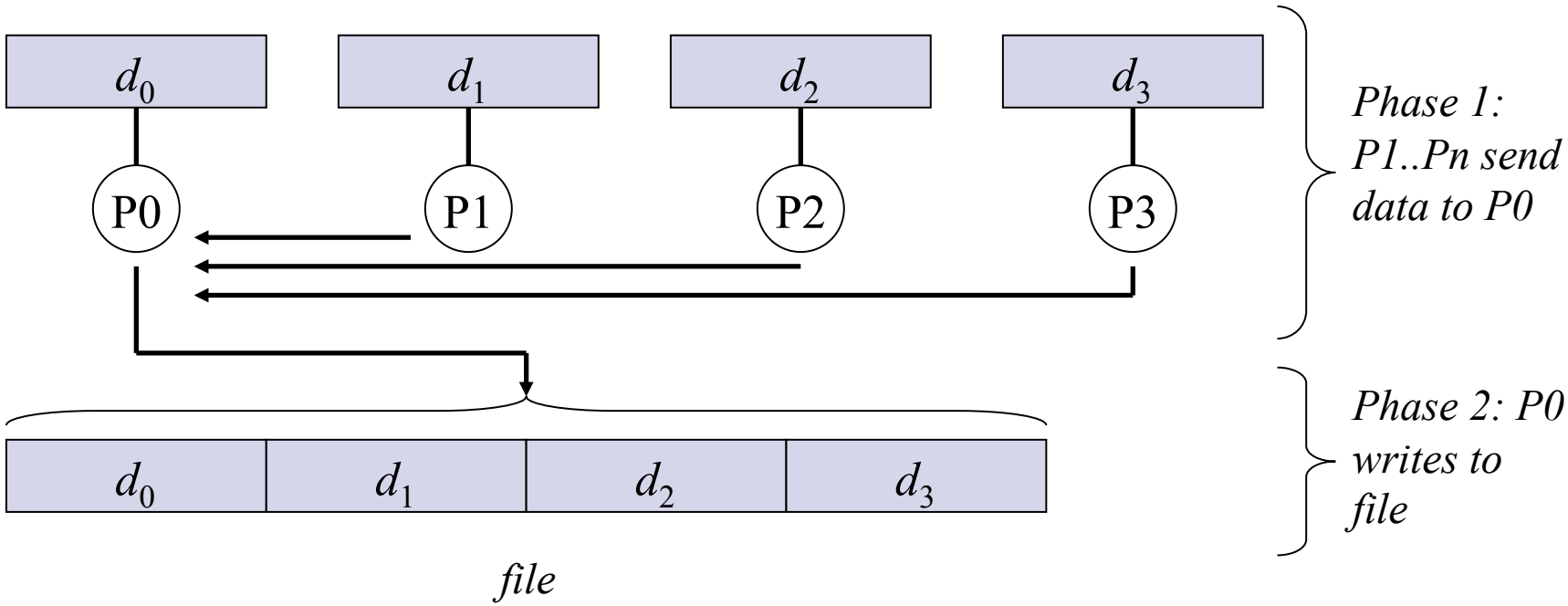
- Single processor performs I/O:
  - To delegate all I/O to one processor is cumbersome and slow
- Multiple processors perform I/O on separate files:
  - Managing a local file by each process requires pre- and post processing to merge and split global files
- Standard UNIX I/O is not optimal for parallel I/O
- True parallel I/O requires concurrent multi-process access to a file system

Parallel I/O Bottleneck

- I/O is much slower than CPU
  - 10-1000 MB/s versus 10-1000 Gflop/s
- Many I/O subsystems designed for high performance
  - Good: optimized for contiguous large data transfers of >GB
  - Bad: small I/O requests, non-contiguous accesses
  - Bad: UNIX-like interface (single contiguous data access)
- Out of necessity, parallel applications usually perform many small I/O requests of 1kB or less
  - Realistic transfer rate <10% of peak I/O bandwidth

"Parallel" I/O: Simple Solution

- Processes send data to P0
- P0 performs the file I/O

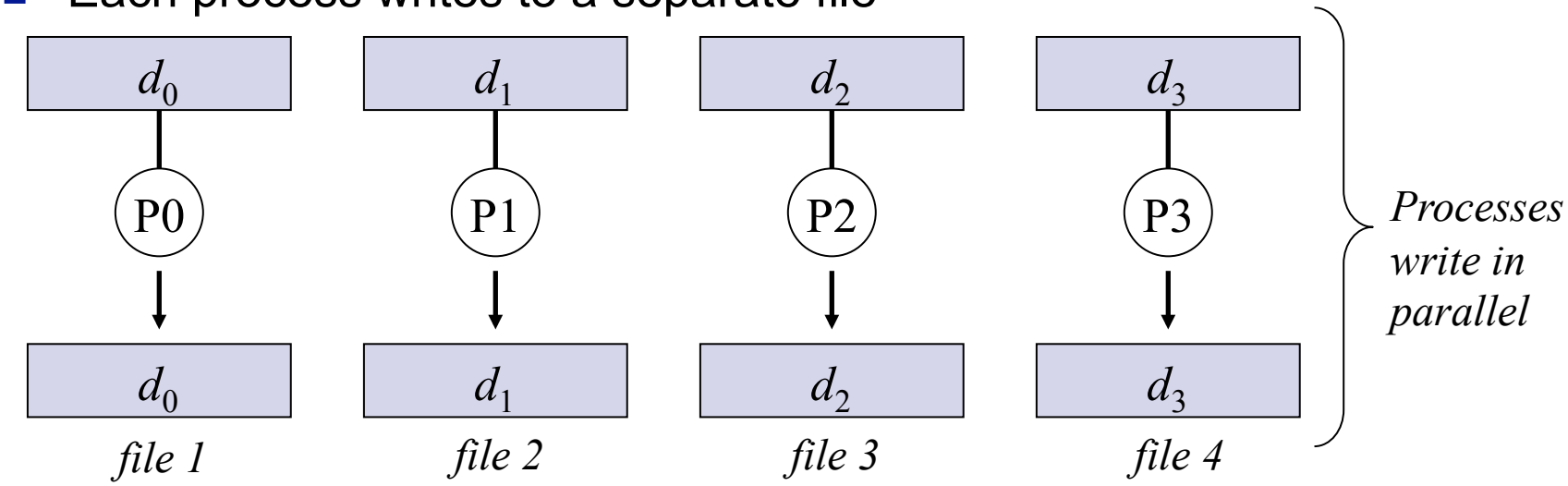


“Parallel” I/O: Simple Solution

- The good
  - I/O from only one process
  - No specialized I/O library needed
  - Results in single file is easy to manage
  - Parallel code is easy to derive from original sequential code (when parallelizing an application)
  - Bigger block size leads to better performance
- The bad
  - Single node bottleneck
  - Poor performance when data size is significant
  - Poor scalability
  - Single point of failure

Parallel I/O on Multiple Files

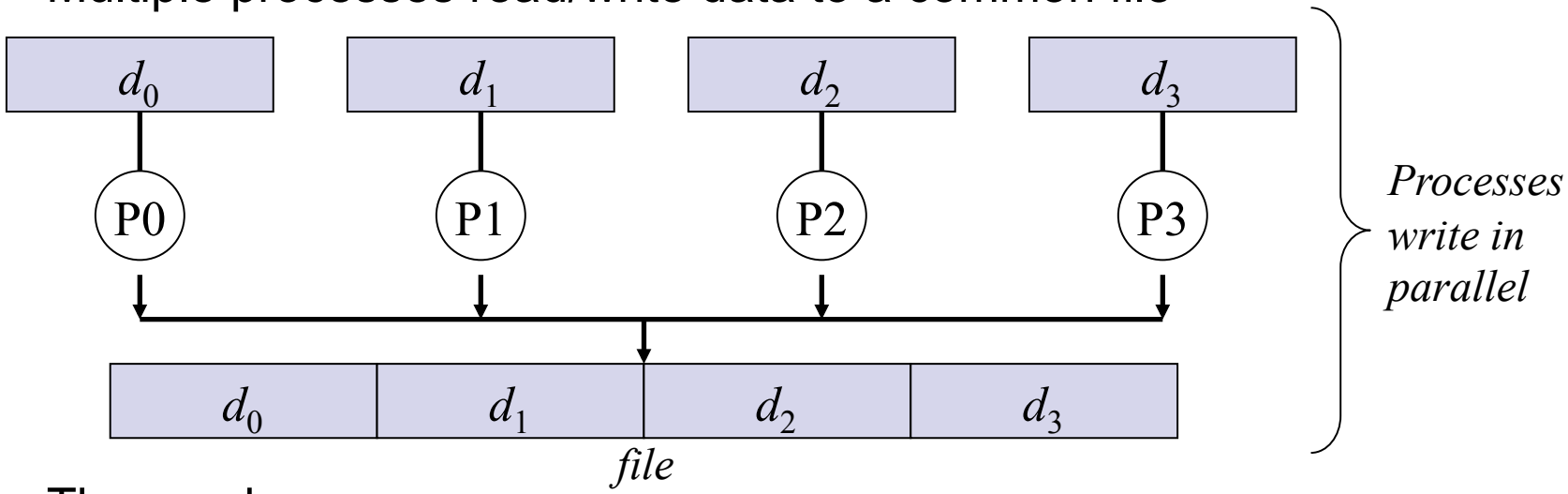
- Each process writes to a separate file



- The good:
  - Parallel and better performance
- The bad:
  - Must manage many small files
  - How to read data back when the number of procs changes?

Parallel I/O Library Support

- Multiple processes read/write data to a common file



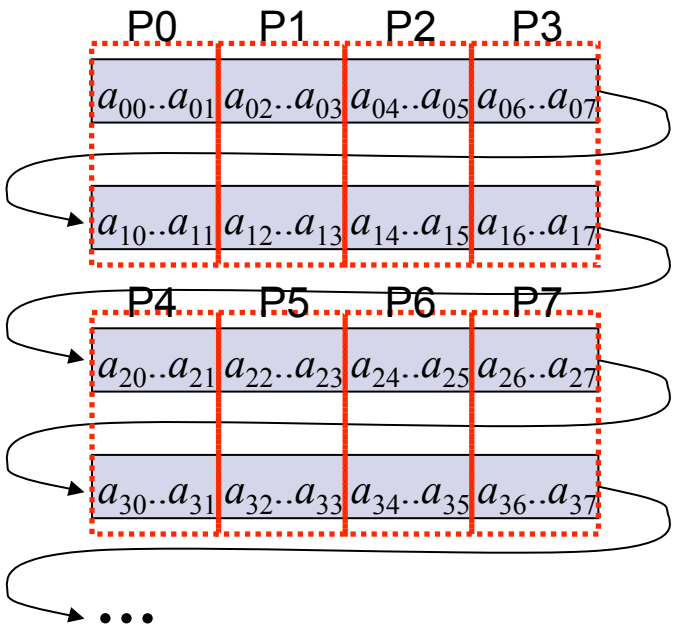
- The good:
  - Simultaneous I/O, ensures performance and scalability by avoiding fragmented seek-read/write access patterns to the file
  - Maps to collective communications (scatter/gather)
  - Single file is easy to manage
- The bad:
  - Requires more complicated I/O library support

Parallel I/O Example

8×8 array (block,block)-distributed over 16 processors

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Row-major array layout in file



Each process has a different "view" of the file content and starts reading at a different displacement and skipping "holes"

For example: P0 reads bytes 0..7, 64..71, P1 reads 8..15, 72..79, …

Parallel I/O Library Design

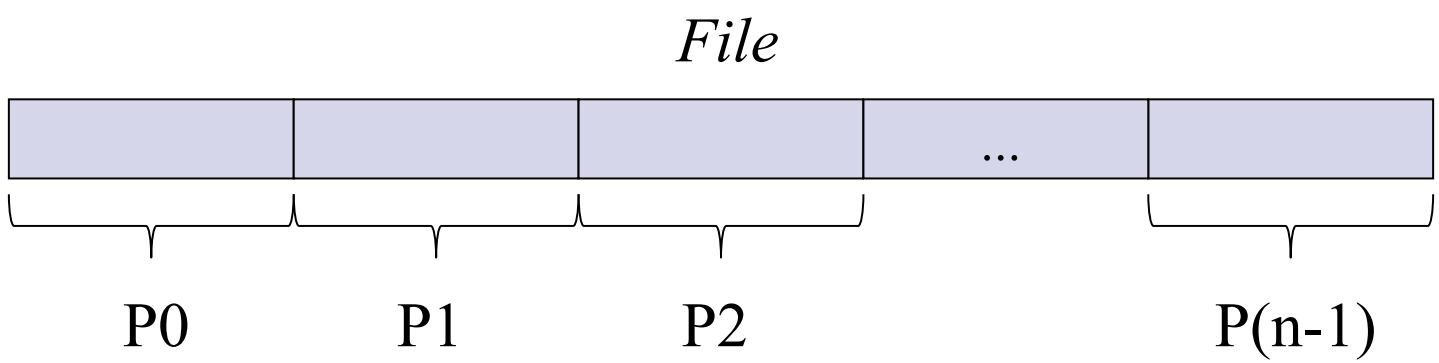
- Simple UNIX-like API is not sufficiently powerful
- Parallel I/O API problem
  - Should provide simultaneous single file access
  - Should provide contiguous and non-contiguous file access
  - Should provide collective I/O operations
  - Should provide synchronization and atomicity (locks)
  - Should use standard basic and derived data types (interoperability)
- Similar to message passing API…
  - Writing ≈ sending, reading ≈ receiving
  - Collective I/O on one file ≈ gather/scatter on one data set
  - MPI provides features for synchronization and atomicity
  - MPI provides interoperable data types

MPI-IO

- MPI-IO is part of the MPI-2 standards for parallel I/O
- MPI-IO defines the following concepts:
  - **Etype** (elementary data type)
    - A basic data type or derived data type
  - **File**
    - Is an ordered collection of etypes
    - Opened and manipulated collectively by a group of processes
  - **Fileview**
    - Each process may have a different view of the file content
    - A view provides non-contiguous file access, where a specific mutually disjoint subset of the file is identified for each process
  - **File type**
    - The etype used to create a view
  - **Displacement** (offset)
    - Defines the location where a view begins with respect to a process

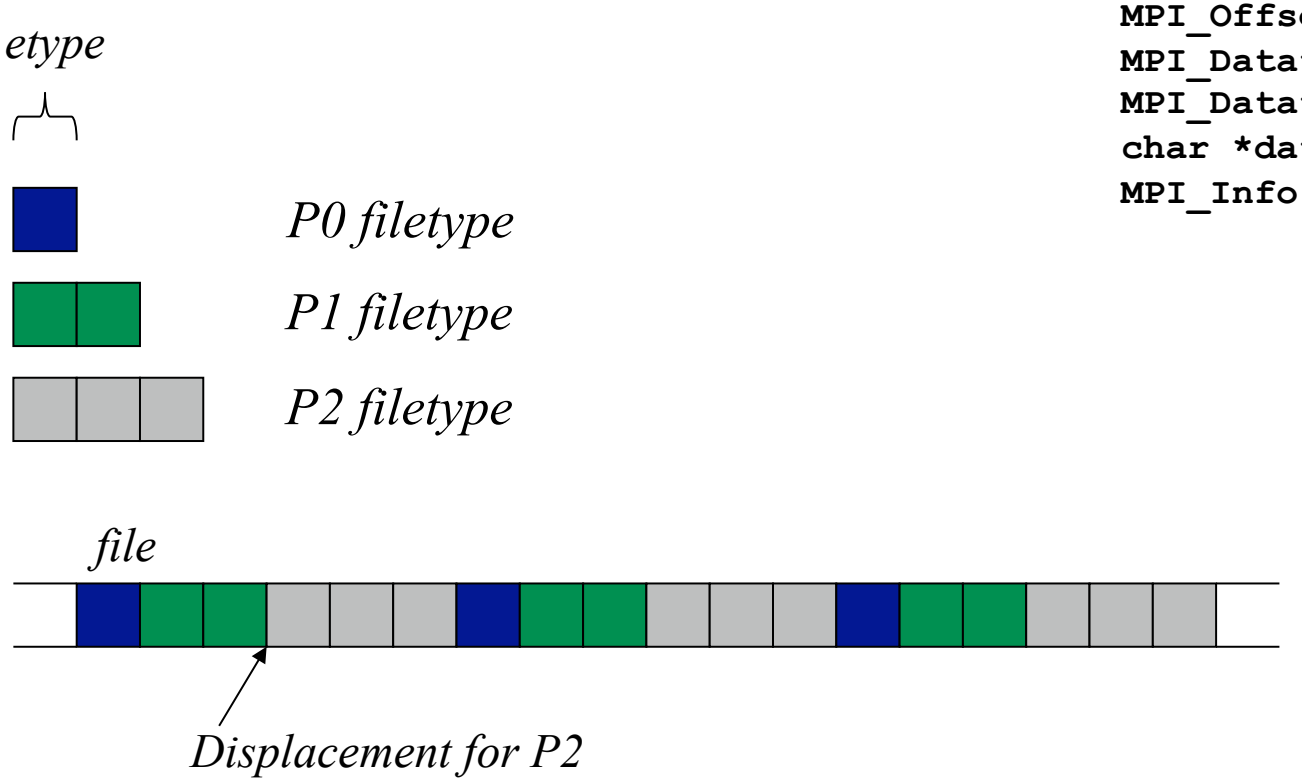
MPI-IO File Views

- The displacement, etype, and filetype creates a fileview by invoking `MPI_File_set_view`
- A fileview allows simultaneous writing/reading of noncontiguous interleaved data by multiple processes
- Each process has a different fileview of a single file



Interleaving with MPI-IO File Views

```
int MPI_File_set_view(MPI_File fh,
                      MPI_Offset disp,
                      MPI_Datatype etype,
                      MPI_Datatype filetype,
                      char *datarep,
                      MPI_Info info)
```

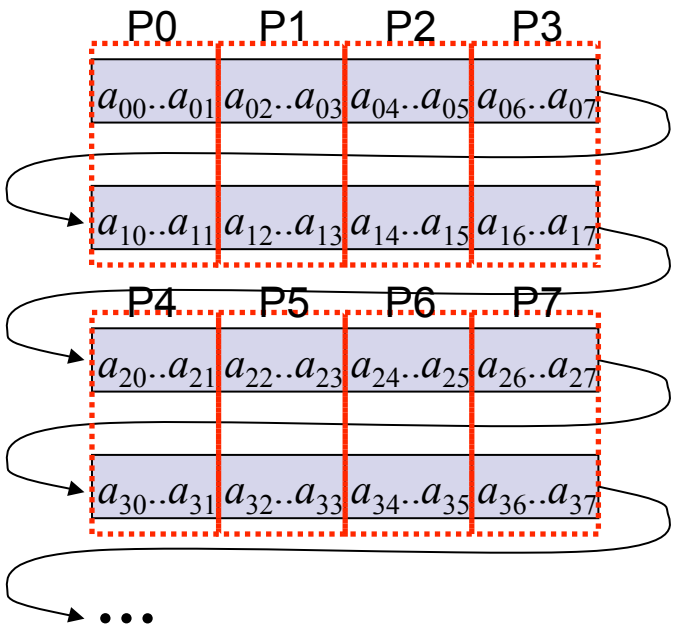


Example Revisited

8×8 array (block,block)-distributed over 16 processors

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Row-major array layout in file

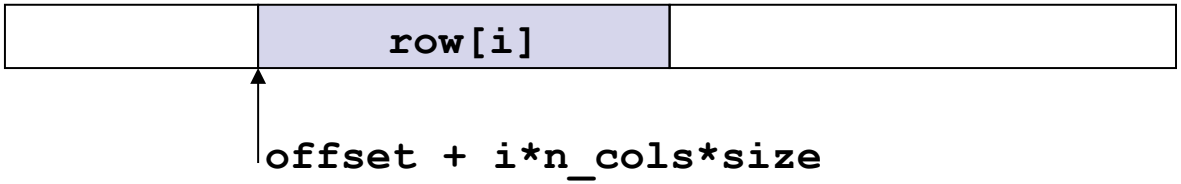



```
n_rows = 8
n_cols = 8
n_local_rows = 2
n_local_cols = 2
size = sizeof(int)
offset = (rank / 4) * n_cols * n_local_rows * size
       + (rank % 4) * n_local_cols * size
```

MPI-IO Access Level 0

- Many independent, contiguous requests
  - UNIX-style file access pattern
  - Individual file pointers per process per file handle
  - Not collective, not blocking on other threads

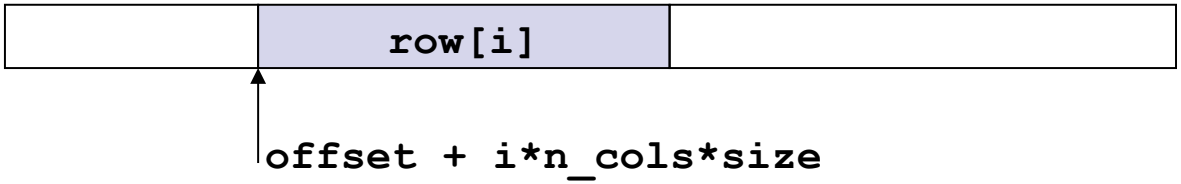
```
MPI_File_open(MPI_COMM_WORLD, "filename",
              MPI_MODE_RDONLY, …, &fh);
for (i = 0; i < n_local_rows; i++) {
  MPI_File_seek(fh, offset + i*n_cols*size, MPI_SEEK_SET);
  MPI_File_read(fh, row[i], n_local_cols, MPI_INT, &stat);
}
MPI_File_close(&fh);
```



MPI-IO Access Level 1

- Many collective, contiguous requests
  - Processes access file at the same time
  - Collective, blocks on other threads
  - Improved efficiency through collectively orchestrated access

```
MPI_File_open(MPI_COMM_WORLD, "filename",
              MPI_MODE_RDONLY, …, &fh);
for (i = 0; i < n_local_rows; i++) {
  MPI_File_seek(fh, offset + i*n_cols*size, MPI_SEEK_SET);
  MPI_File_read_all(fh, row[i], n_local_cols, MPI_INT, &stat);
}
MPI_File_close(&fh);
```



MPI-IO Access Level 2

- Single independent, non-contiguous request
  - Each process creates a file type describing a subarray with **`MPI_Type_create_subarray()`**
  - Each process creates a view using **`MPI_File_set_view()`** (a collective call) to access the non-contiguous data in the subarray
  - Reads are independent

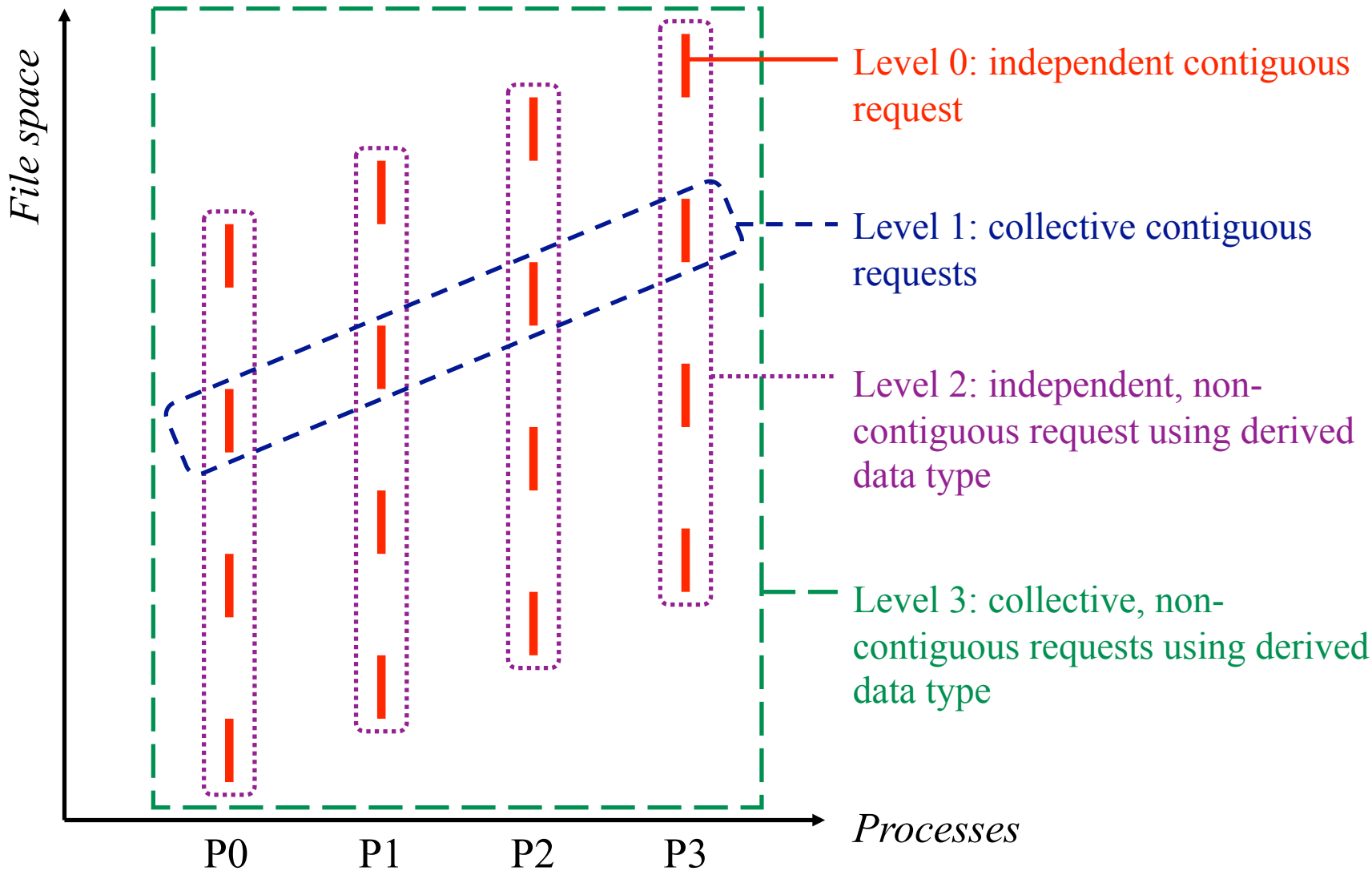
```
MPI_Type_create_subarray(…, &subarray, …);
MPI_Type_commit(&subarray);
MPI_File_open(MPI_COMM_WORLD, "filename",
              MPI_MODE_RDONLY, …, &fh);
MPI_File_set_view(fh, …, subarray, …);
MPI_File_read(fh, local_array, n_local_cols*n_local_rows,
              MPI_INT, &stat);
MPI_File_close(&fh);
```

MPI-IO Access Level 3

- Single collective, non-contiguous request
  - Each process creates a file type describing a subarray with **`MPI_Type_create_subarray()`**
  - Each process creates a view using **`MPI_File_set_view()`** (a collective call) to access the non-contiguous data in the subarray
  - Reads are collective

```
MPI_Type_create_subarray(…, &subarray, …);
MPI_Type_commit(&subarray);
MPI_File_open(MPI_COMM_WORLD, "filename",
              MPI_MODE_RDONLY, …, &fh);
MPI_File_set_view(fh, …, subarray, …);
MPI_File_read_all(fh, local_array, n_local_cols*n_local_rows,
                  MPI_INT, &stat);
MPI_File_close(&fh);
```

MPI-IO Levels Overview



MPI-IO Performance

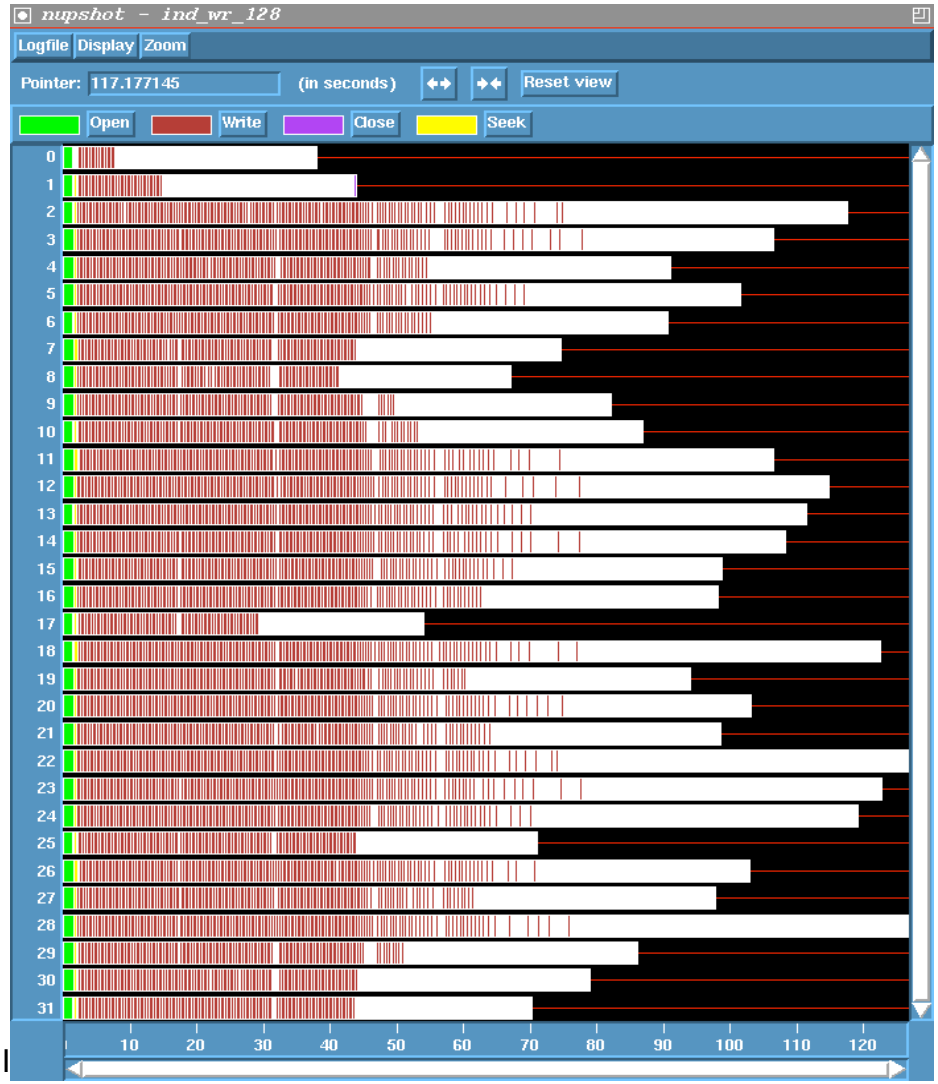
- Performance of ROMIO in MB/s bandwidth for an 512×512×512 integer array

Machine	#procs	Level 0 / 1	Level 2	Level 3
HP Exemplar	64	5.42	14.2	68.2
IBM SP	64	2.13	11.9	90.2
Intel Paragon	256	3.01	9.50	132
NEC SX-4	8	0.71	322	563
SGI Origin2000	32	14.0	118	175

Source: Thakur and Gropp, Argonne National Labs

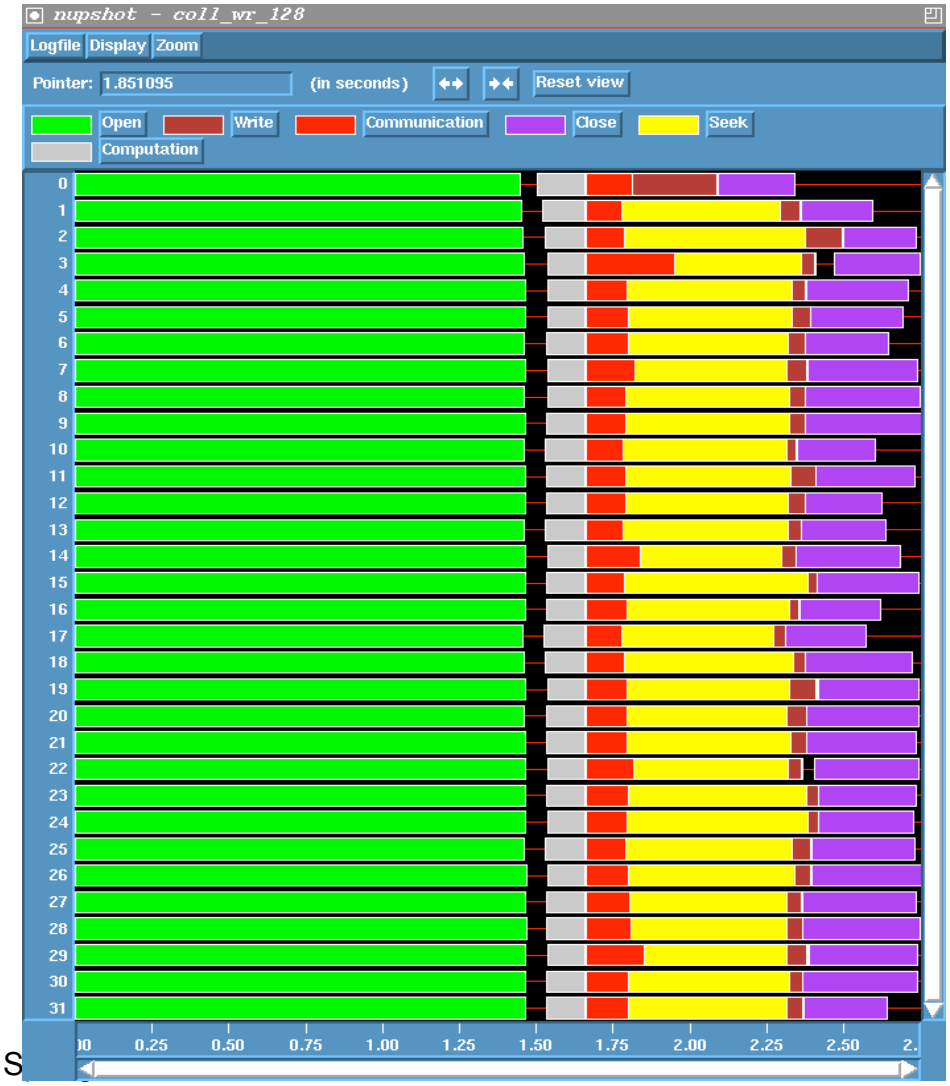
Level 0 Independent Writes

- Lots of seeks and small writes
- 130 seconds



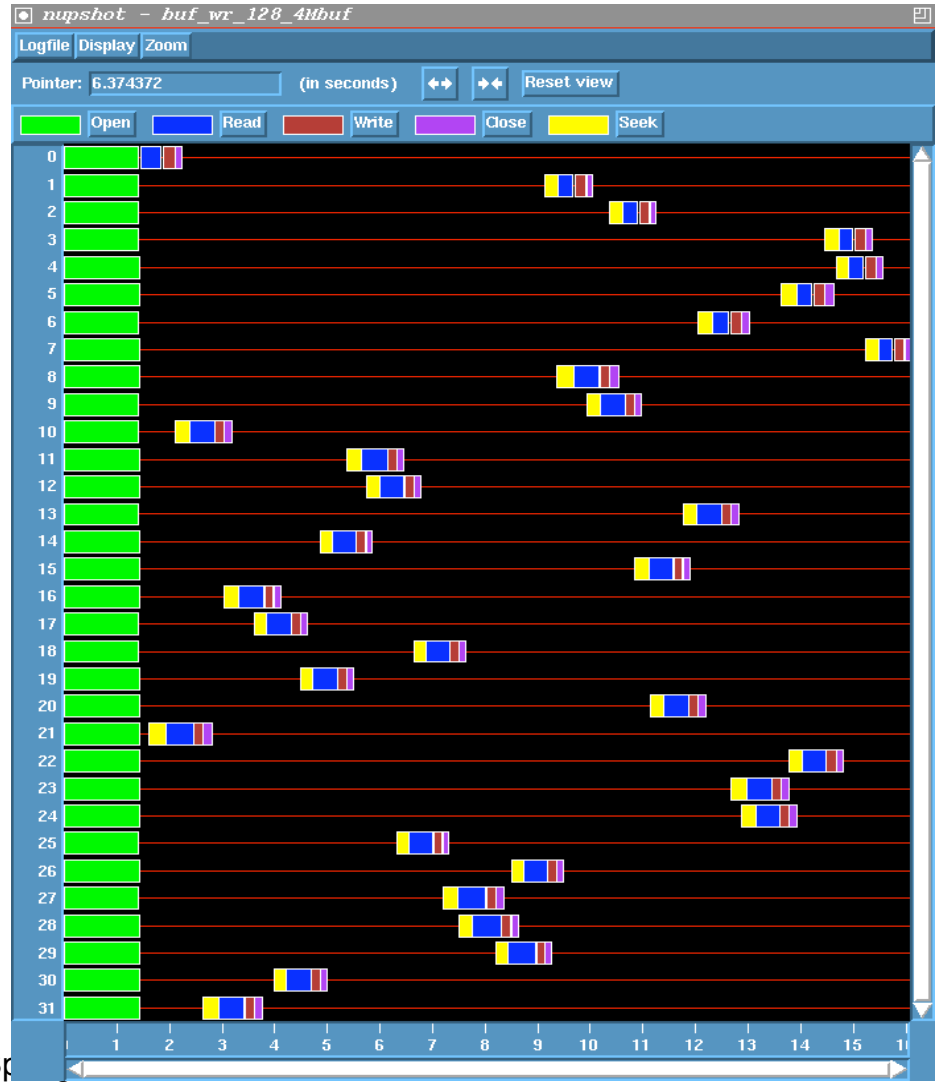
Level 3 Collective Write

- Computation and communication precede seek and write
- 2.75 seconds



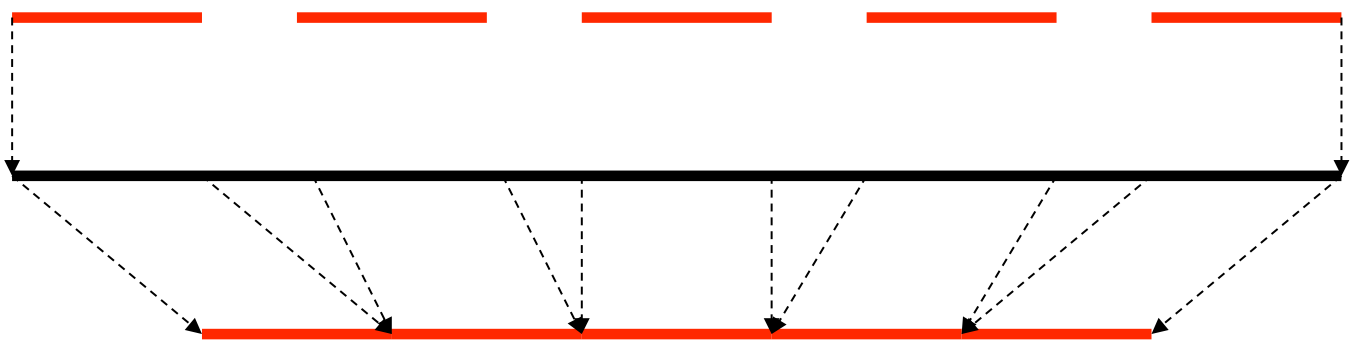
Level 2 Independent Writes with Data Sieving

- Access data in large blocks
- Requires lock, read, modify, write, unlock for each operation
- 4 MB blocks
- 16 seconds



Data Sieving: Read

- Data sieving for reads



User's request for non-contiguous data from a file

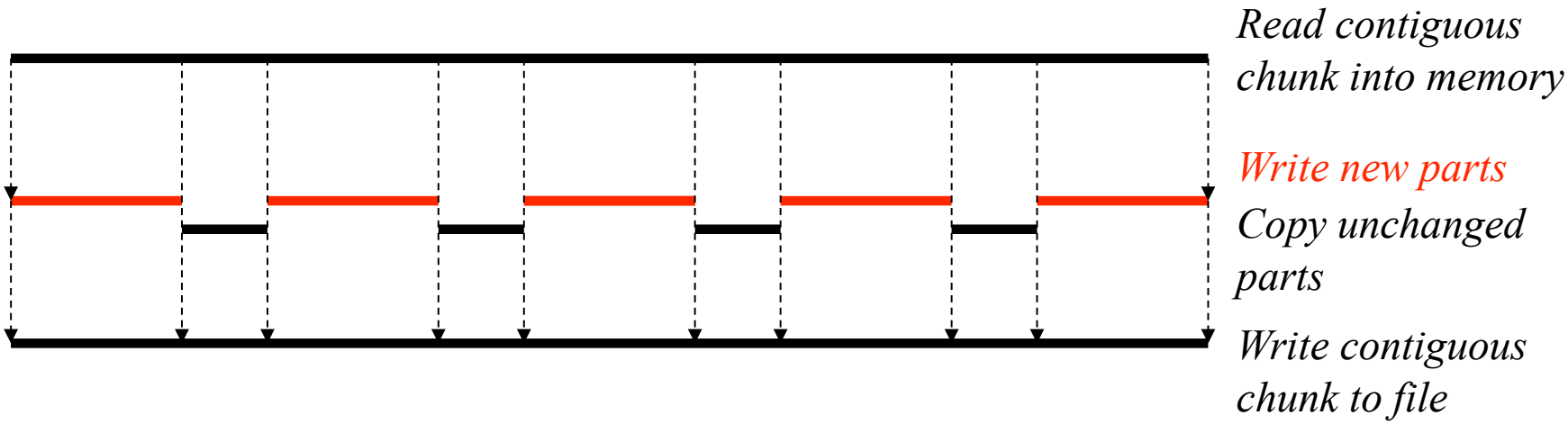
Read contiguous chunk into memory

Copy requested portions into user's destination (single or multiple buffers)

- The good
  - High performance non-contiguous reads with lower system I/O
- The bad
  - Memory requirements to store the potentially useless "holes"

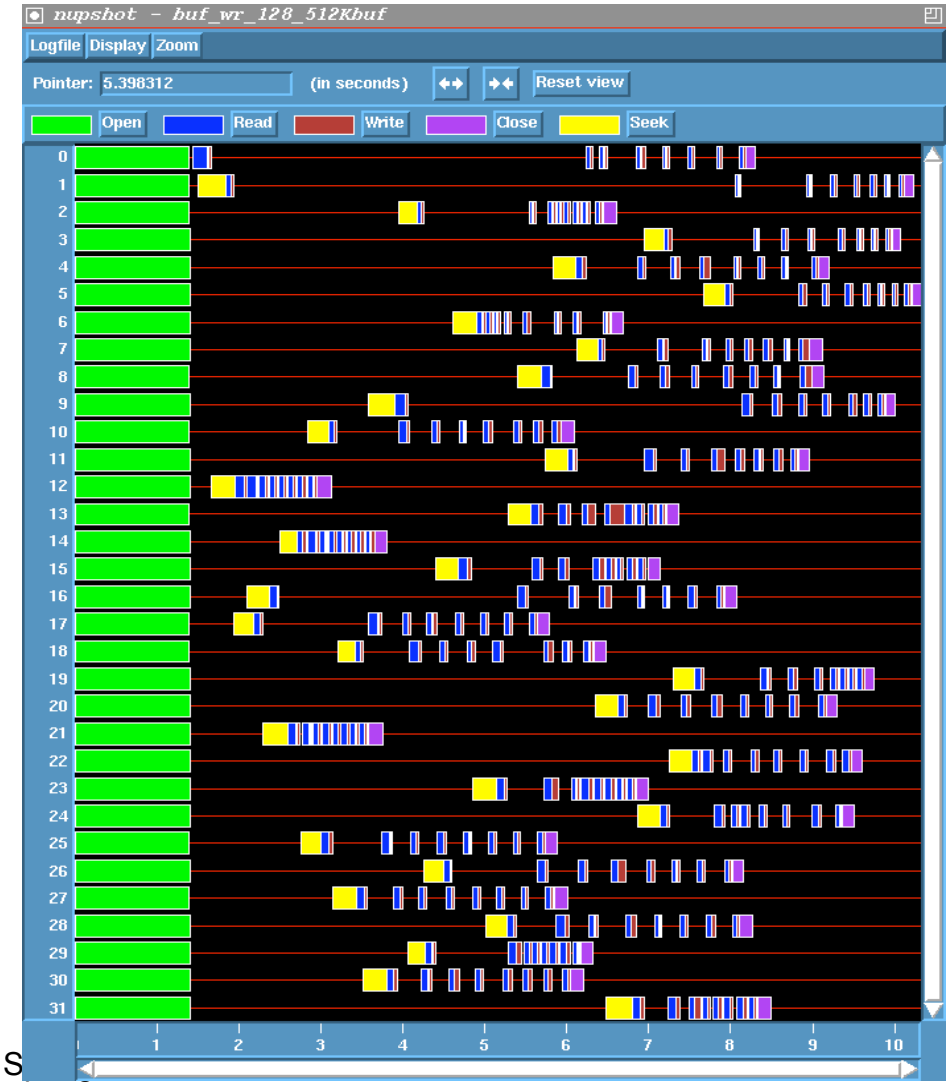
Data Sieving: Read-Modify-Write

- Data sieving for writes is more complicated
  - Read a block, modify parts, and write back



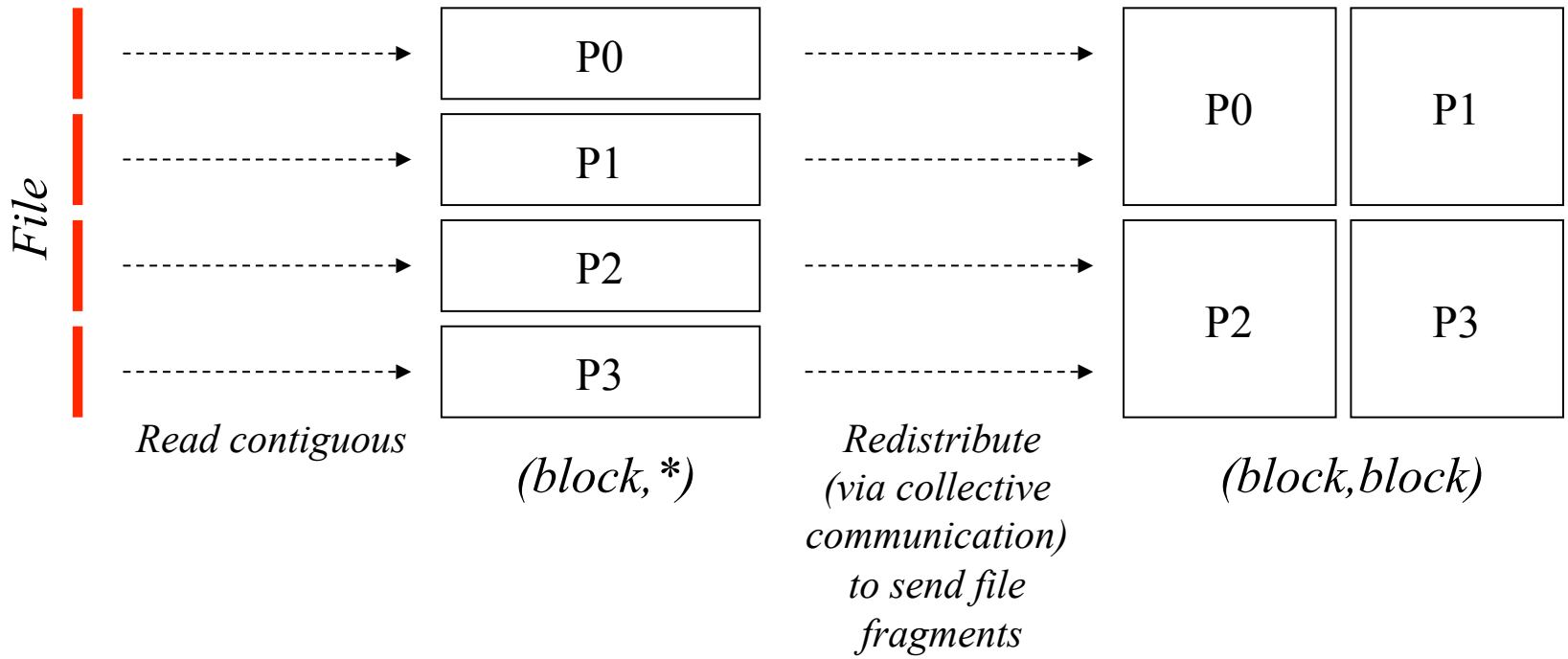
Level 2 Independent Writes with Data Sieving: Small Blocks

- Smaller blocks mean less contention
- 512 KB blocks
- 10.2 seconds

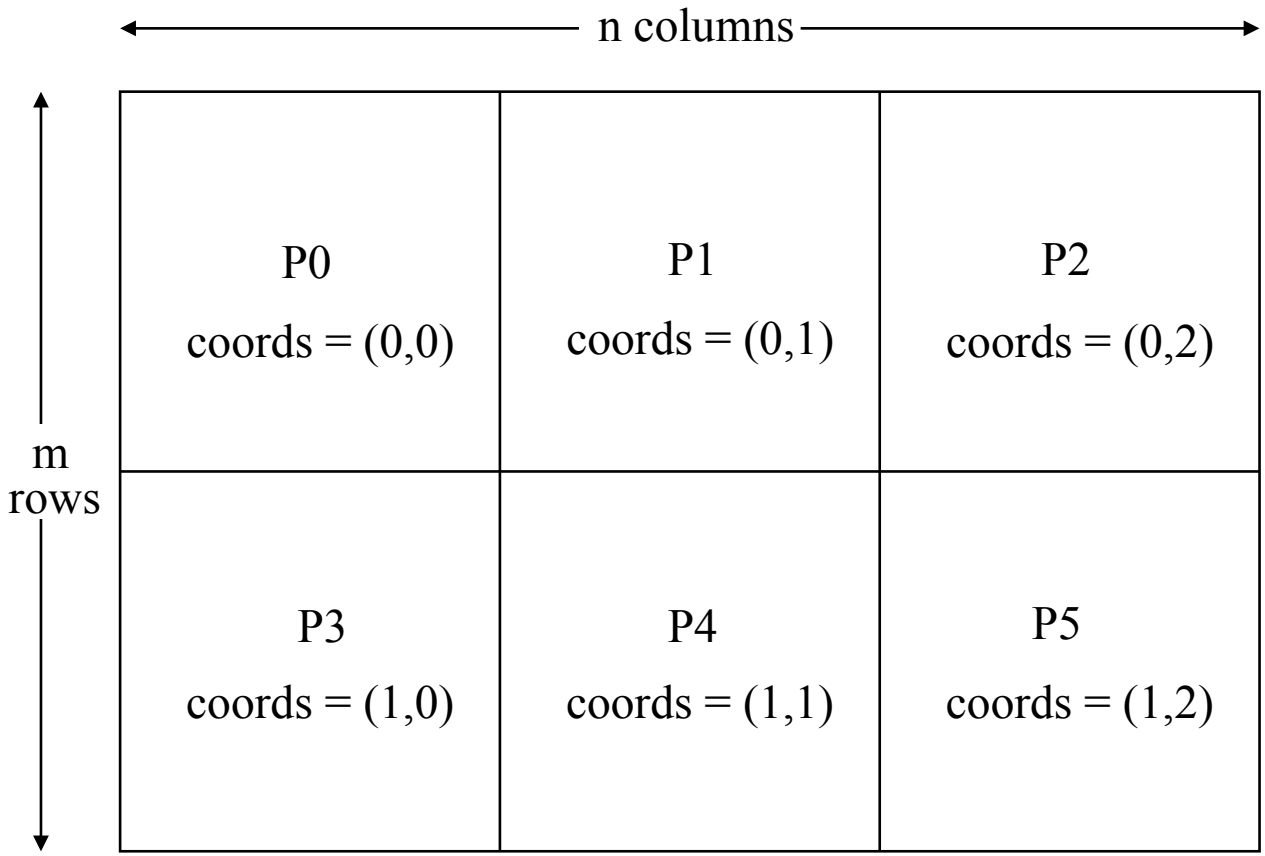


Two-Phase Collective I/O

- Two-phase collective I/O
- When array layout on file differs from array layout on processors, for example (block,*) → (block,block)



Example: Distributed Array



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

Example: Distributed Array (cont'd)

```
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 */
```

Example: Distributed Array (cont'd)

```
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);
```

Distributed Arrays

- The darray datatype assumes a very specific definition of data distribution
  - Same definition as in HPF
  - If the array size is not divisible by the number of processes, darray calculates the block size using a ceiling division
  - Assumes a row-major ordering of processes in the logical grid, as assumed by cartesian process topologies in MPI-1
- If an application uses a different definition for data distribution or logical grid ordering, do not use darray
  - Use subarray instead

Example: Subarray

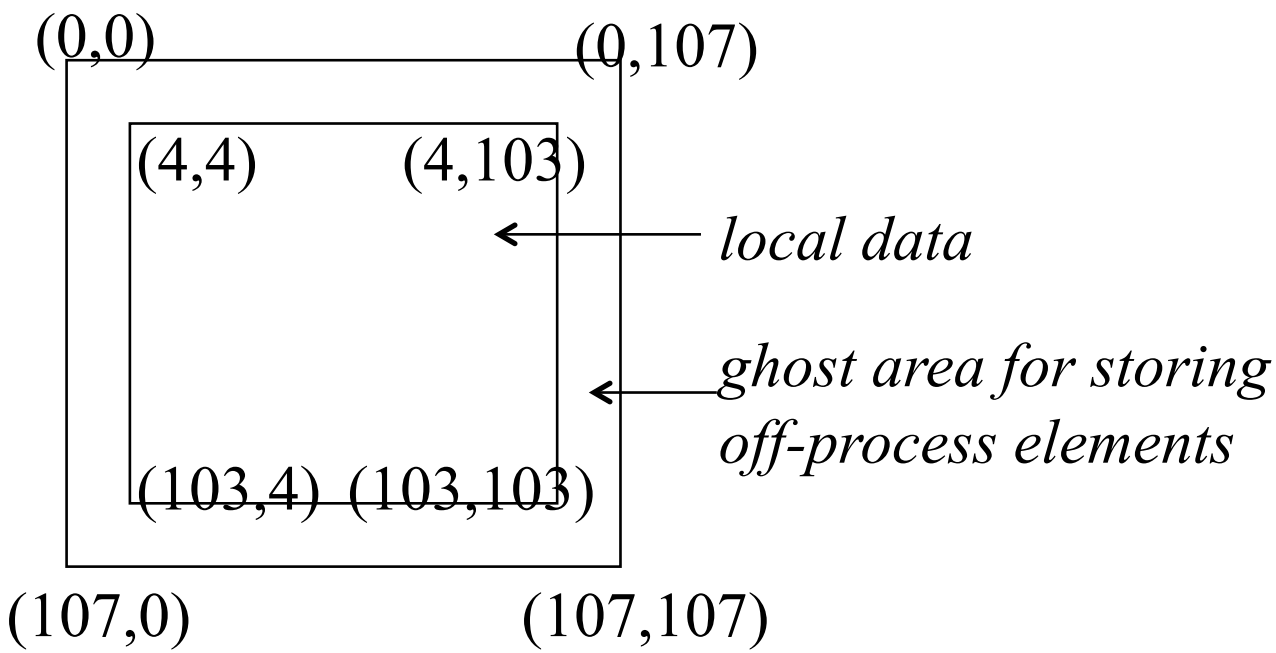
```
gsizes[0] = m;  /* no. of rows in global array */
gsizes[1] = n;  /* no. of columns in global array*/

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);
```

Example: Subarray (cont'd)

```
/* 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);
```

Example: Local Area With Ghost Cells

- Use a subarray datatype to describe the noncontiguous layout in memory

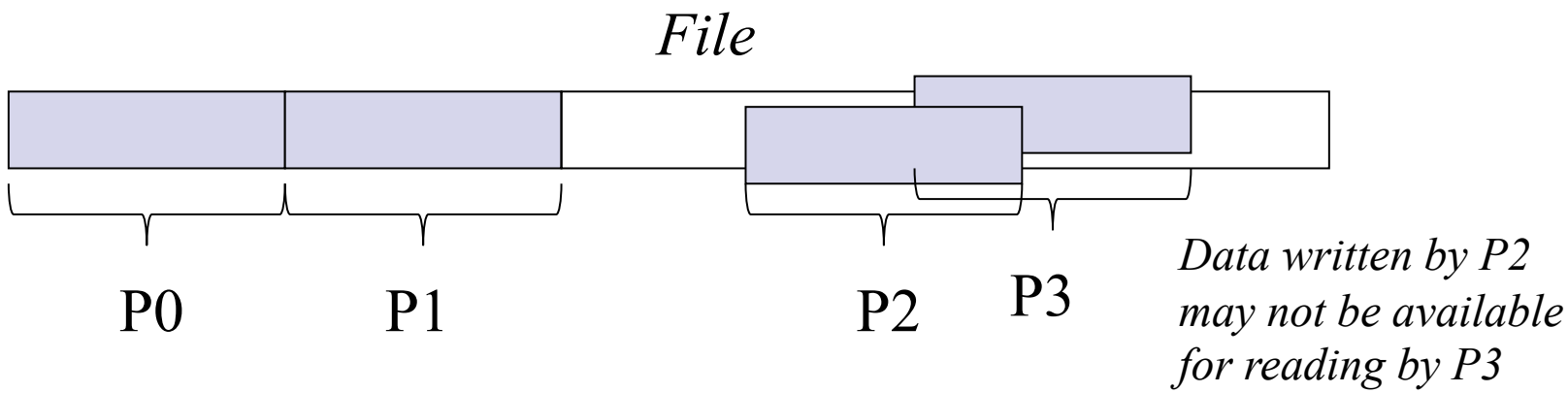


Example: Local Area With Ghost Cells (cont'd)

```
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);
```

Consistency Semantics of Parallel I/O

- Consistency semantics define the I/O results when multiple processes access a common file and one or more processes write to that file
  - When the regions of the file that different processes read and write to do not overlap, constancy is guaranteed
  - When the regions overlap, consistency cannot be achieved unless the user takes extra steps



Example 1

- Each process writes to a separate region of the file and reads back only what it wrote
  - MPI-IO guarantees that the data will be read correctly

P0

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=0,cnt=100)
MPI_File_read_at(off=0,cnt=100)
```

P1

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=100,cnt=100)
MPI_File_read_at(off=100,cnt=100)
```

Example 2

- Each process wants to read what the other wrote
  - In this case MPI-IO does not guarantee that the data will be automatically read correctly

P2

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=0,cnt=100)
MPI_Barrier
MPI_File_read_at(off=100,cnt=100)
```

P3

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=100,cnt=100)
MPI_Barrier
MPI_File_read_at(off=0,cnt=100)
```

Example 2 (cont'd)

- The user must take extra steps to ensure correctness
- There are three choices:
  - Set atomicity to true
  - Close the file and reopen it
  - Ensure that no write sequence on any process is concurrent with any sequence (read or write) on another process

Example 2: Atomicity

- Each process wants to read what the other wrote
  - Option 1 is to set atomicity to true

P2

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_set_atomicity(fh1,1)
MPI_File_write_at(off=0,cnt=100)
MPI_Barrier
MPI_File_read_at(off=100,cnt=100)
```

P3

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_set_atomicity(fh2,1)
MPI_File_write_at(off=100,cnt=100)
MPI_Barrier
MPI_File_read_at(off=0,cnt=100)
```

Example 2: Close and Reopen

- Each process wants to read what the other wrote
  - Option 2 is to close and reopen the file

P2

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=0,cnt=100)
MPI_File_close
MPI_Barrier
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_read_at(off=100,cnt=100)
```

P3

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=100,cnt=100)
MPI_File_close
MPI_Barrier
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_read_at(off=0,cnt=100)
```

Example 2: Use File_sync to Separate Access Sequences

- Each process wants to read what the other wrote
  - Option 3 is to ensure that no write sequence on any process is concurrent with any sequence (read or write) on another process
  - Separate sequences using MPI_File_sync

P2

```
MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=0,cnt=100)
MPI_File_sync
MPI_Barrier
MPI_File_sync /*collective*/

MPI_File_sync /*collective*/
MPI_Barrier
MPI_File_sync
MPI_File_read_at(off=100,cnt=100)
MPI_File_close
```

P3

```
MPI_File_open(MPI_COMM_WORLD,…)

MPI_File_sync /*collective*/
MPI_Barrier
MPI_File_sync
MPI_File_write_at(off=100,cnt=100)
MPI_File_sync
MPI_Barrier
MPI_File_sync /*collective*/
MPI_File_read_at(off=0,cnt=100)
MPI_File_close
```

What About Parallel File Systems (PFS)?

- A parallel file system (or distributed parallel file system)
- Organize I/O devices into a single logical space
  - Files scattered over hosts, each with a local file system
  - Striping of files across distributed hosts for performance
    - Each host may also have multiple disks (e.g. RAID) with striped files
  - Access data in contiguous regions of bytes
  - Export a well-defined API, usually POSIX
- PFS is very general
  - Example implementations: PVFS, Lustre
- Network overhead is bad for parallel I/O
- Striping is good for parallel I/O

What About the POSIX I/O Interface?

- Standard I/O application programming interface (API) across many platforms
- API is best suited for serial applications to perform I/O
  - No way of describing collective access
- Warning: semantics differ between file systems!
  - NFS is the worst of these, supporting API but not semantics
  - Determining FS type is nontrivial
- Hence the need for MPI-IO standard API for parallel I/O

General Guidelines

- Buy sufficient I/O hardware for the machine
  - RAID and special-purpose high-performance disks
- Use fast file systems
  - Avoid NFS
- Do not perform I/O from one process only
  - Many parallel applications use one process to do all the I/O and other processes send/recv data to it, resulting in slow downs
- Make large requests whenever possible
  - Write application to fetch/store large amounts of data at once
- Use MPI-IO and use it the right way

MPI-IO Summary

- The higher the MPI-IO level of the access pattern, the better the performance
  - Collectives are faster than individual accesses
  - A non-contiguous read (write) is faster than a series of contiguous reads (writes)
- Choose an access level pattern based on the application's I/O characteristics
  - Few I/O operations, for example at initialization and completion
    - Does not matter much
  - Frequent I/O operations, for example each simulation timestep
    - Perform I/O from multiple processes
    - Combine data to perform large non-contiguous collective I/O requests
    - For *n*-dim arrays (*n*>1), choose an effective array distribution
    - Use fast I/O hardware (more disks), fast file systems, avoid NFS, …

Further Reading

- [SRC] Chapter 11
- MPI-2 manuals

Acknowledgements

- Rajeev Thakur
  Mathematics and Computer Science Division
  Argonne National Laboratory