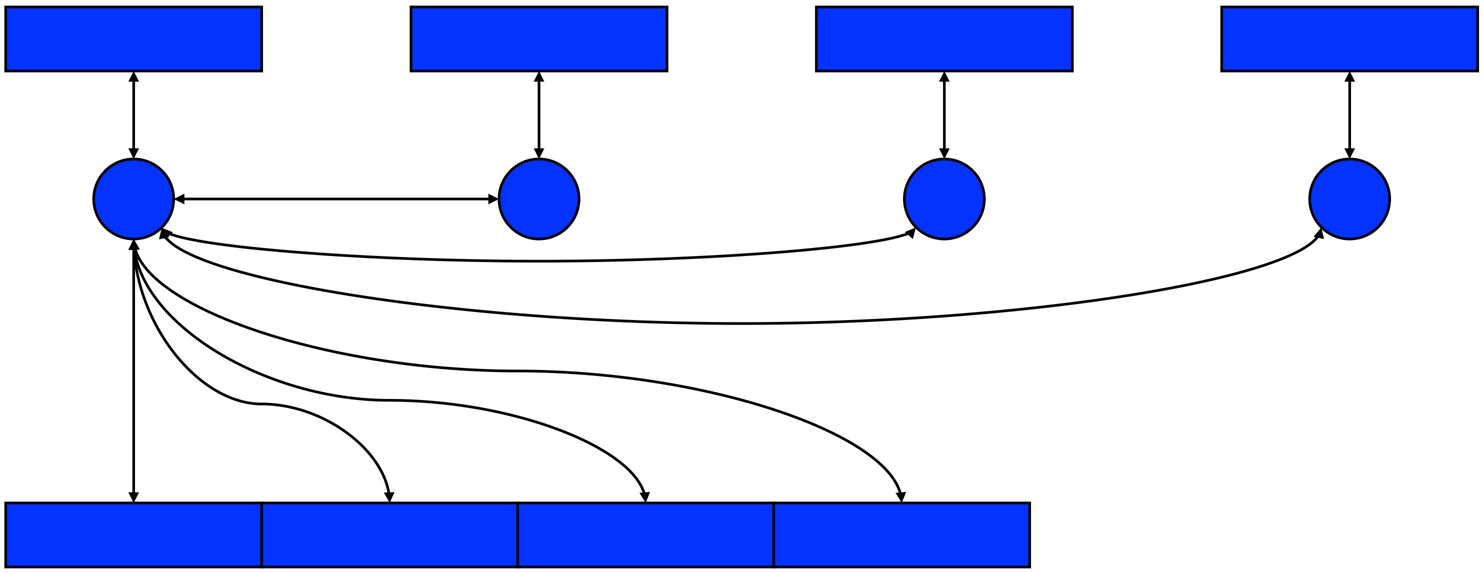
Lecture 32: Introduction to MPI I/O

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

Parallel I/O in MPI

- Why do I/O in MPI?
  - ♦ Why not just POSIX?
    - Parallel performance
    - Single file (instead of one file / process)
- MPI has replacement functions for POSIX I/O
  - ♦ Provides migration path
- Multiple styles of I/O can all be expressed in MPI
  - ♦ Including some that cannot be expressed without MPI

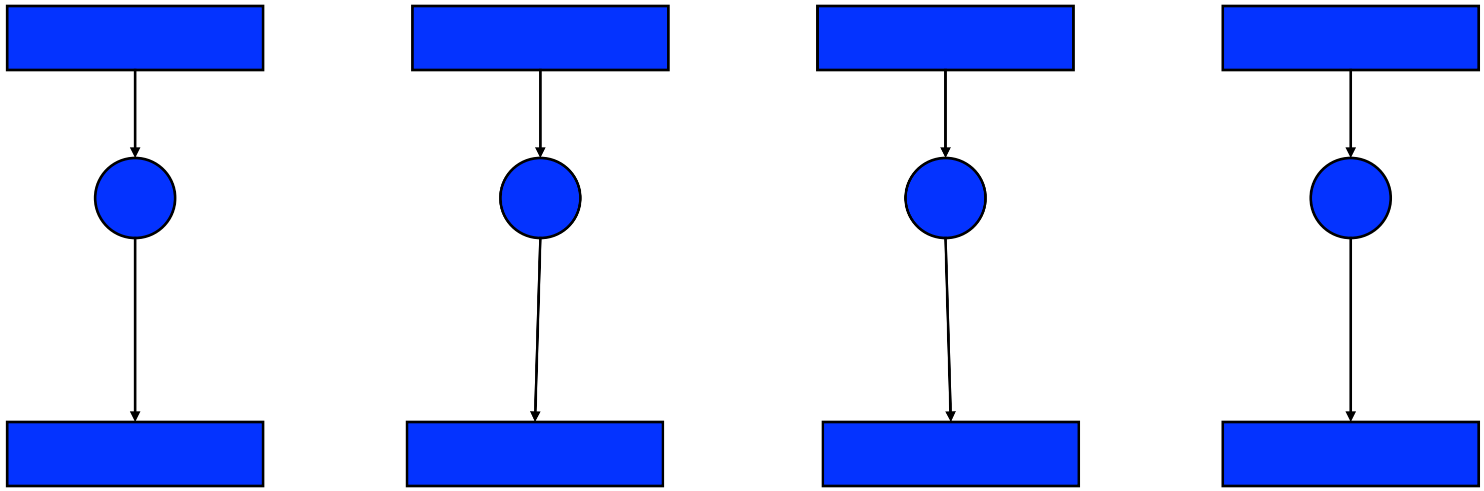
Non-Parallel I/O



- Non-parallel
- Performance worse than sequential
- Legacy from before application was parallelized
- Either MPI or not

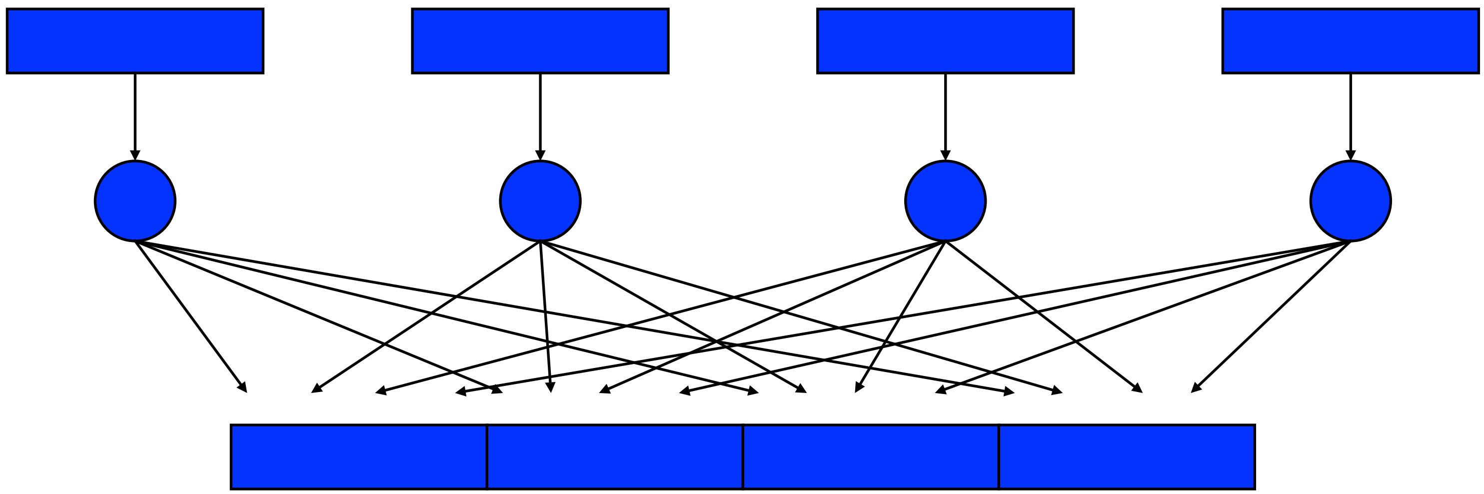
Independent Parallel I/O

- Each process writes to a separate file



- Pro: parallelism
- Con: lots of small files to manage
- Legacy from before MPI
- MPI or not

Cooperative Parallel I/O



- Parallelism
- Can only be expressed in MPI
- Natural once you get used to it

Why MPI is a Good Setting for Parallel I/O

- Writing is like sending and reading is like receiving.
- Any parallel I/O system will need:
  - collective operations
  - user-defined datatypes to describe both memory and file layout
  - communicators to separate application-level message passing from I/O-related message passing
  - non-blocking operations
- I.e., lots of MPI-like machinery

What does Parallel I/O Mean?

- At the program level:
  - Concurrent reads or writes from multiple processes to a common file
- At the system level:
  - A parallel file system and hardware that support such concurrent access

Independent I/O with MPI-IO

The Basics: An Example

- Just like POSIX I/O, you need to
  - Open the file
  - Read or Write data to the file
  - Close the file
- In MPI, these steps are almost the same:
  - Open the file: MPI_File_open
  - Write to the file: MPI_File_write
  - Close the file: MPI_File_close

A Complete Example

```
#include <stdio.h>
#include "mpi.h”
int main(int argc, char *argv[])
{
    MPI_File fh;
    int buf[1000], rank;
    MPI_Init(0,0);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_File_open(MPI_COMM_WORLD, "test.out",
                  MPI_MODE_CREATE|MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &fh);
    if (rank == 0)
       MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
    MPI_File_close(&fh);
    MPI_Finalize();
    return 0;
}
```

Comments on Example

- File Open is collective over the communicator
  - Will be used to support collective I/O, which we will see is important for performance
  - Modes similar to Unix open
  - MPI_Info provides additional hints for performance
- File Write is independent (hence the test on rank)
  - Many important variations covered in later slides
- File close is collective; similar in style to MPI_Comm_free

Writing to a File

- Use **MPI_File_write** or **MPI_File_write_at**
- Use **MPI_MODE_WRONLY** or **MPI_MODE_RDWR** as the flags to **MPI_File_open**
- If the file doesn't exist previously, the flag **MPI_MODE_CREATE** must also be passed to **MPI_File_open**
- We can pass multiple flags by using bitwise-or '|' in C, or addition '+" in Fortran

Ways to Access a Shared File

- **MPI_File_seek**
- **MPI_File_read**
- **MPI_File_write**

  like Unix I/O

- **MPI_File_read_at**
- **MPI_File_write_at**

  combine seek and I/O for thread safety

- **MPI_File_read_shared**
- **MPI_File_write_shared**

  use shared file pointer

Using Explicit Offsets

```
#include "mpi.h"
MPI_Status status;
MPI_File fh;
MPI_Offset offset;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
            MPI_MODE_RDONLY, MPI_INFO_NULL, &fh)
nints = FILESIZE / (nprocs*INTSIZE);
offset = rank * nints * INTSIZE;
MPI_File_read_at(fh, offset, buf, nints, MPI_INT,
                 &status);
MPI_Get_count(&status, MPI_INT, &count);
printf("process %d read %d ints\n", rank, count);

MPI_File_close(&fh);
```

Why Use Independent I/O?

- Sometimes the synchronization of collective calls is not natural
- Sometimes the overhead of collective calls outweighs their benefits
  - Example: very small I/O during header reads

Noncontiguous I/O in File

- Each process describes the part of the file for which it is responsible
  - This is the "file view"
  - Described in MPI with an offset (useful for headers) and an MPI_Datatype
- Only the part of the file described by the file view is visible to the process; reads and writes access these locations
- This provides an efficient way to perform *noncontiguous accesses*

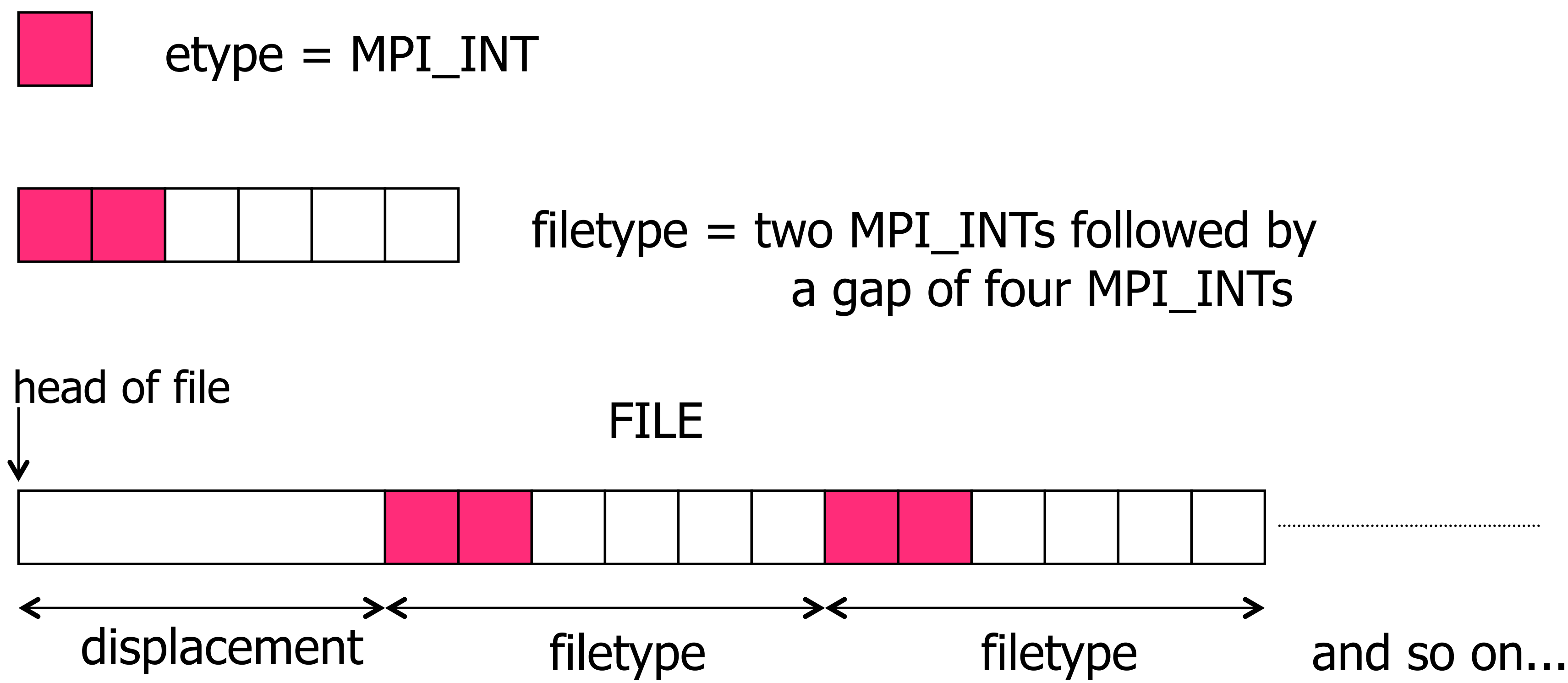
Noncontiguous Accesses

- Common in parallel applications
- Example: distributed arrays stored in files
- A big advantage of MPI I/O over Unix I/O is the ability to specify noncontiguous accesses in memory **and** file within a single function call by using derived datatypes
  - POSIX only supports non-contiguous in file, and only with IOVs
- Allows implementation to optimize the access
- Collective I/O combined with noncontiguous accesses yields the highest performance

File Views

- Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to **`MPI_File_set_view`**
- *displacement* = number of bytes to be skipped from the start of the file
  - e.g., to skip a file header
- *etype* = basic unit of data access (can be any basic or derived datatype)
- *filetype* = specifies which portion of the file is visible to the process

A Simple Noncontiguous File View Example



Noncontiguous File View Code

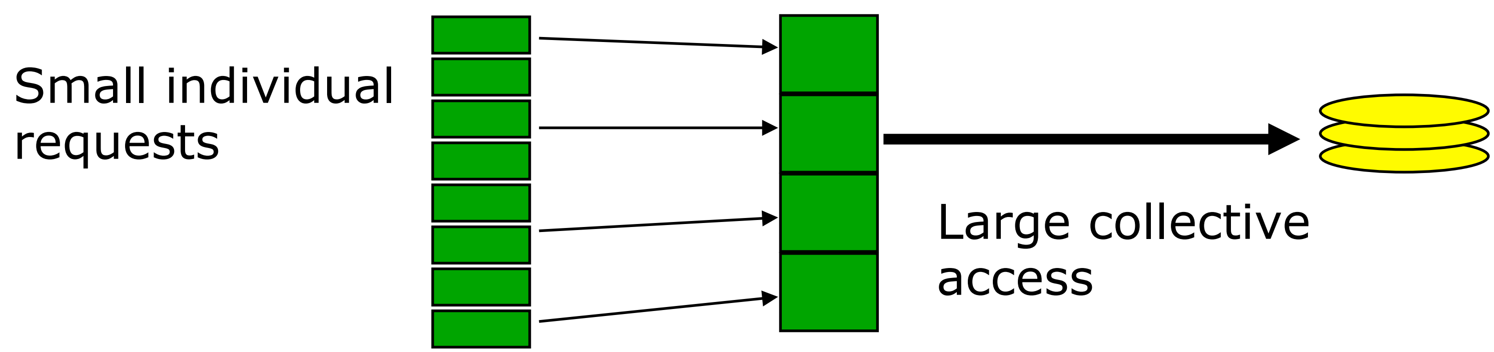
```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

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, "/pfs/datafile",
     MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Collective I/O and MPI

- A critical optimization in parallel I/O
- All processes (in the communicator) must call the collective I/O function
- Allows communication of "big picture" to file system
  - Framework for I/O optimizations at the MPI-IO layer
- Basic idea: build large blocks, so that reads/writes in I/O system will be large
  - Requests from different processes may be merged together
  - Particularly effective when the accesses of different processes are noncontiguous and interleaved



PARALLEL@ILLINOIS

Collective I/O Functions

- **`MPI_File_write_at_all`**, etc.
  - ♦ **`_all`** indicates that all processes in the group specified by the communicator passed to **`MPI_File_open`** will call this function
  - ♦ **`_at`** indicates that the position in the file is specified as part of the call; this provides thread-safety and clearer code than using a separate "seek" call
- Each process specifies only its own access information — the argument list is the same as for the non-collective functions

The Other Collective I/O Calls

- **`MPI_File_seek`**
- **`MPI_File_read_all`**
- **`MPI_File_write_all`**

like Unix I/O

- **`MPI_File_read_at_all`**
- **`MPI_File_write_at_all`**

combine seek and I/O for thread safety

- **`MPI_File_read_ordered`**
- **`MPI_File_write_ordered`**

use shared file pointer

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

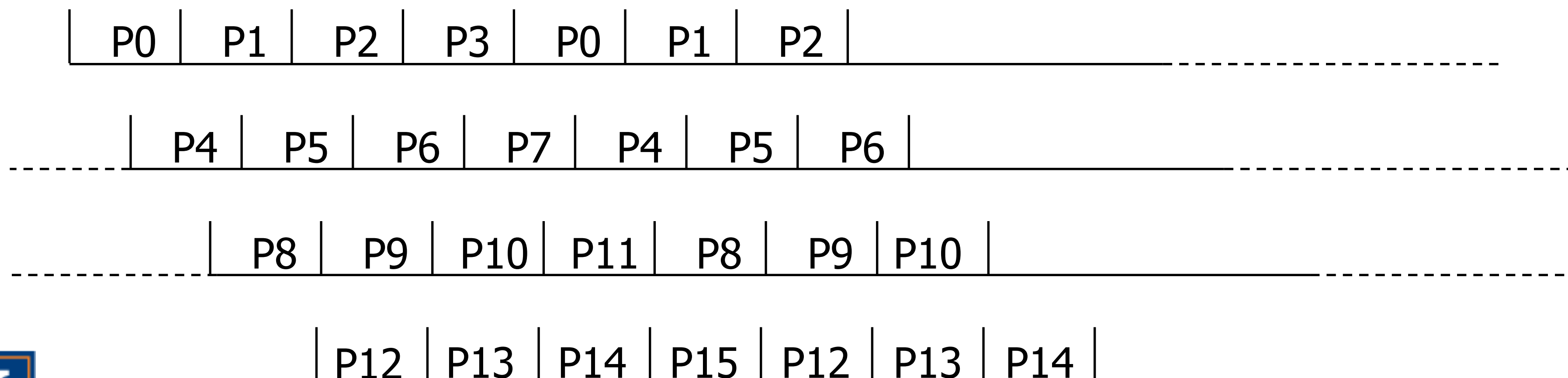
Example: Distributed Array Access

Large array distributed among 16 processes

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

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

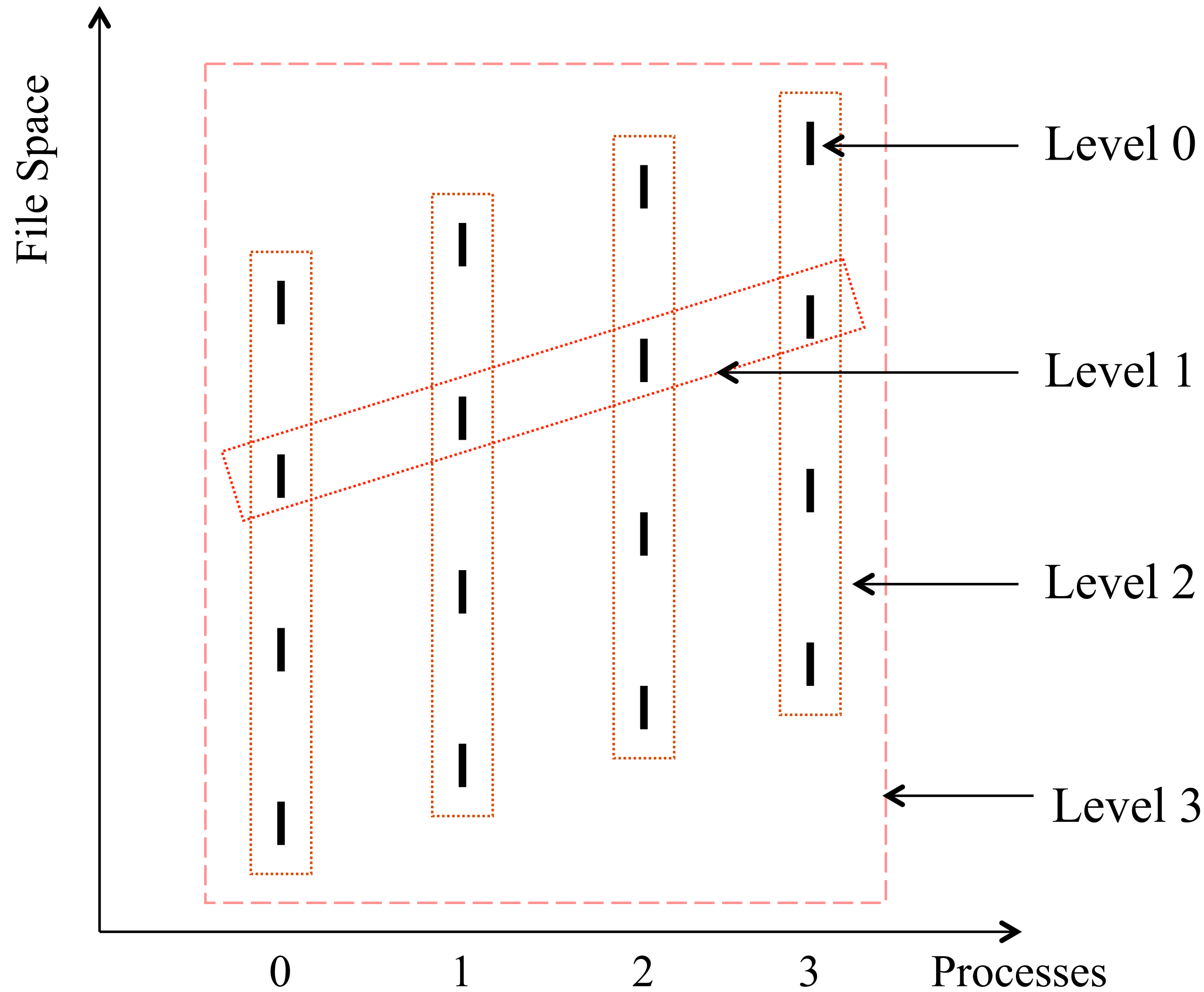
```
MPI_Type_create_subarray(...,
                         &subarray, ...);
MPI_Type_commit(&subarray);
MPI_File_open(..., file, ..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read(fh, A, ...);
MPI_File_close(&fh);
```

Level-3 Access

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

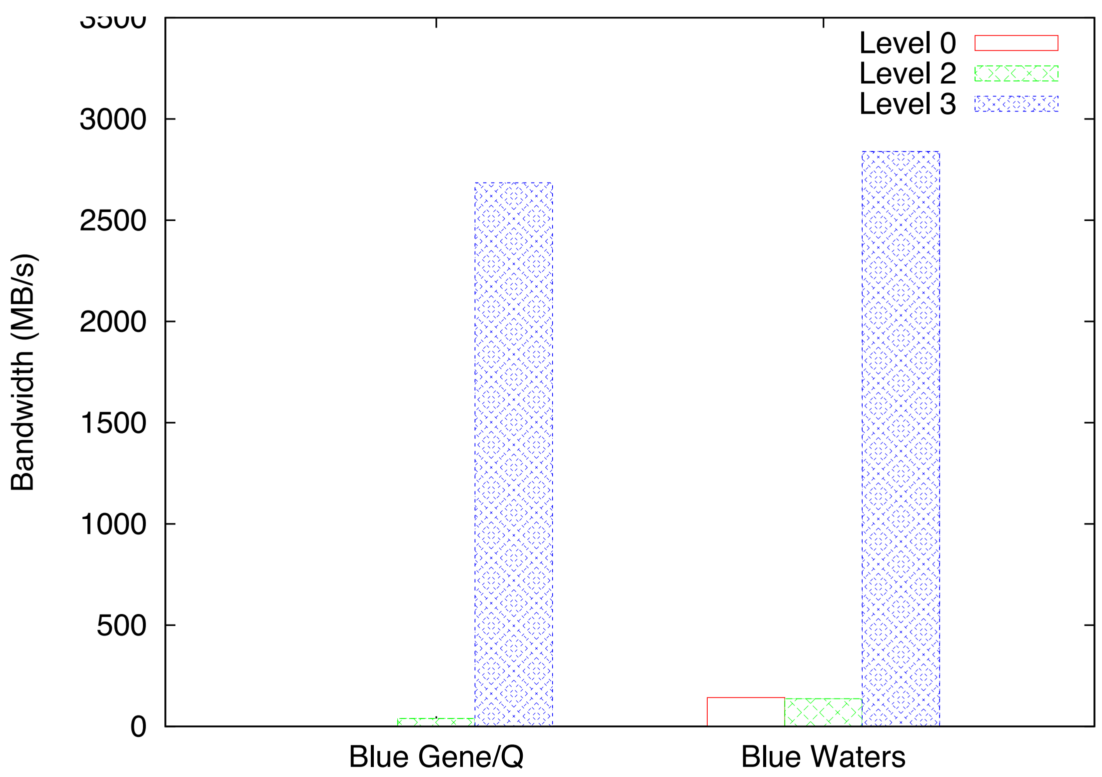
```
MPI_Type_create_subarray(...,
                   &subarray,  ...);
MPI_Type_commit(&subarray);
MPI_File_open(MPI_COMM_WORLD, file,...,
                    &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read_all(fh, A, ...);
MPI_File_close(&fh);
```

The Four Levels of Access



Collective I/O Can Provide Far Higher Performance

- Write performance for a 3D array output in canonical order on 2 supercomputers, using 256 processes (1 process / core)
- Level 0 (independent I/O from each process for each contiguous block of memory) too slow on BG/Q
- Total BW is still low because relatively few nodes in use (16 for Blue Waters = ~180MB/sec/node)



PARALLEL@ILLINOIS

Summary

- Key issues that I/O must address
  - ♦ High latency of devices
    - Nonblocking I/O; cooperative I/O
  - ♦ I/O inefficient if transfers are not both large and aligned with device blocks
    - Collective I/O; datatypes and file views
  - ♦ Data consistency to other users
    - POSIX is far too strong (primary reason parallel file systems have reliability problems)
    - "Big Data" file systems are weak (eventual consistency; tolerate differences)
    - MPI is precise and provides high performance; consistency points guided by users