# Data models with MPI-IO

ATPESC 2018

Rob Latham

Argonne National Laboratory

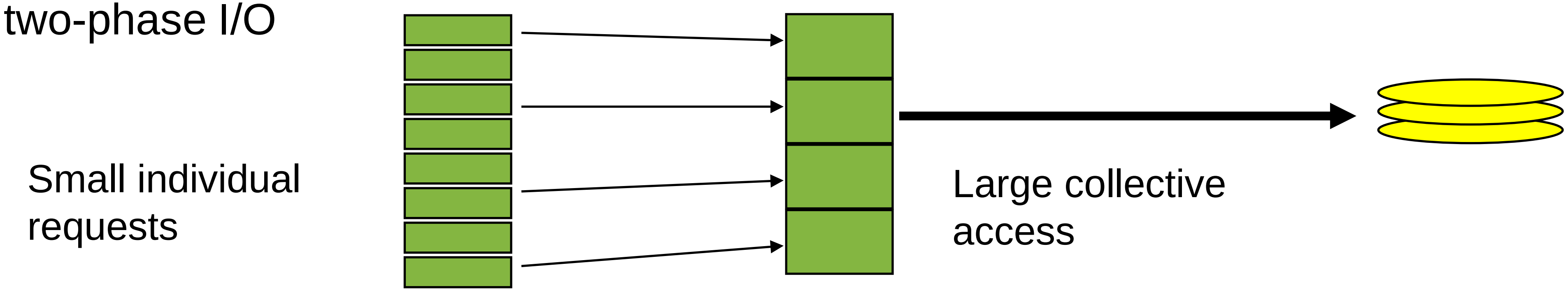
Q Center, St. Charles, IL (USA)
3 August 2018

# Parallel I/O and MPI

- The stdio checkpoint routine works but is not parallel
  - One process is responsible for all I/O
  - Wouldn’t want to use this approach for real
- How can we get the full benefit of a parallel file system?
  - We first look at how parallel I/O works in MPI
  - We then implement a fully parallel checkpoint routine
- 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

# Collective I/O

- 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 transformations/optimizations at the MPI-IO layer
  - Discussed these earlier today
  - e.g., two-phase I/O

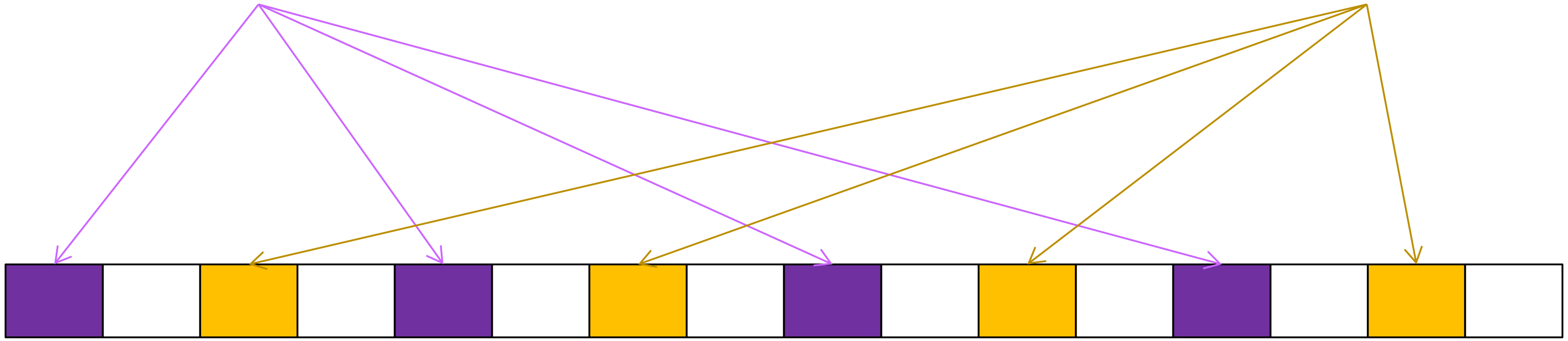


# Simple MPI-IO

- Collective open: all processes in communicator
- File-side data layout with *file views*
- Memory-side data layout with *MPI datatype* passed to write

```
MPI_File_open(COMM, name, mode,
         info, fh);
MPI_File_set_view(fh, disp, etype,
         filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
         datatype, status);
```


```
MPI_File_open(COMM, name, mode,
         info, fh);
MPI_File_set_view(fh, disp, etype,
         filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
         datatype, status);
```

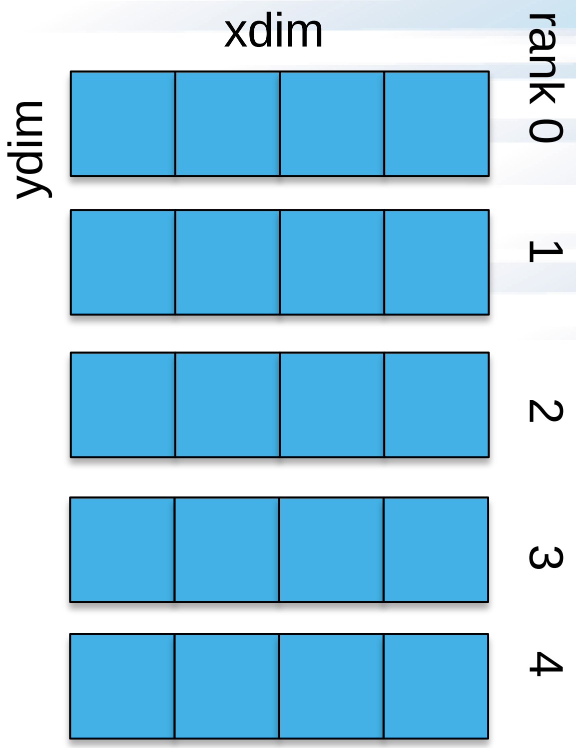



# Collective MPI I/O Functions

- Not going to go through the MPI-IO API in excruciating detail
  - Happy to discuss during exercises, evening
- **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
  - OK to participate with zero data
    - All processes must call a collective
    - Process providing zero data might participate behind the scenes anyway

# HANDS-ON 5: writing with MPI-IO

- Let's take "I/O from master" example and make it parallel
- Use `MPI_File_open` instead of `open`
- Only one process needs to write header
  - Independent `MPI_File_write`
- Every process sets a "file view"
  - Need to skip over header – file view has an "offset" field just for this case
  - The "file view" here is not complicated but we are operating on integers, not bytes:
    - `MPI_File_set_view(fh, sizeof(header), MPI_INT, MPI_INT, "native", info));`
- Each process writes one slice/row of array
  - MPI_File_write_at_all
  - Offset "rank*XDIM*YDIM"
  - "(bufer, count, datatype)" tuple: (`values, XDIM*YDIM, MPI_INT`)



# Solution fragments for Hands-On 5

Header I/O from rank 0:

```
if (rank == 0) {
    MPI_CHECK(MPI_File_write(fh,
        &header, sizeof(header), MPI_BYTE,
        MPI_STATUS_IGNORE) );
}
```

Collective I/O from all ranks

```
MPI_File_write_at_all(fh, rank*XDIM*YDIM,
        values, XDIM*YDIM, MPI_INT,
        MPI_STATUS_IGNORE));
```

# Hands-on 5 continued: Darshan

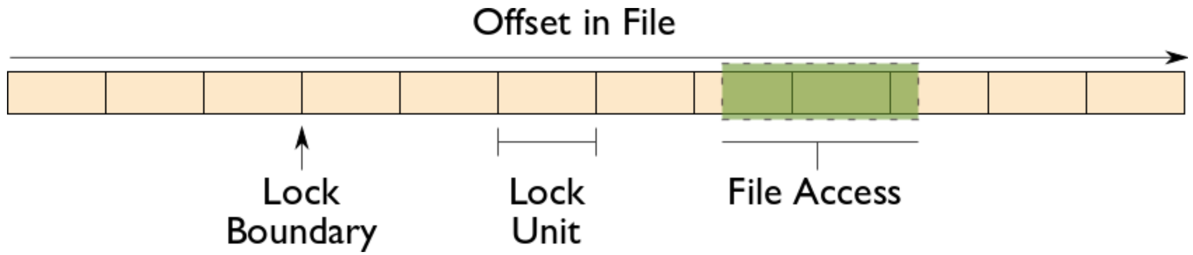
- A lot like #4:  let’s use Darshan
- What do you think the report will say?
- OK, now you generated the report.  Were you surprised?

# Managing Concurrent Access

**Files are treated like global shared memory regions. Locks are used to manage concurrent access**:

- Files are broken up into lock units
  - Unit boundaries are dictated by the storage system, regardless of access pattern
- Clients obtain locks on units that they will access before I/O occurs
- Enables caching on clients as well (as long as client has a lock, it knows its cached data is valid)
- Locks are reclaimed from clients when others desire access

If an access touches any data in a lock unit, the lock for that region must be obtained before access occurs.





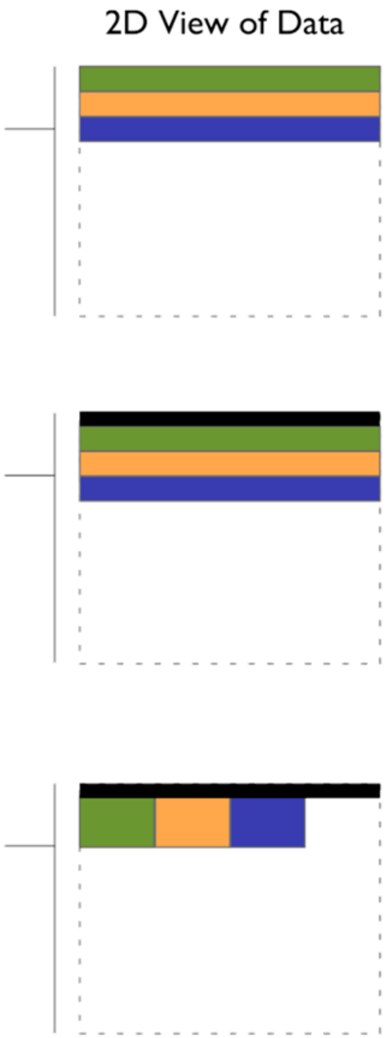

U.S. DEPARTMENT OF ENERGY | Office of Science
ECP EXASCALE COMPUTING PROJECT

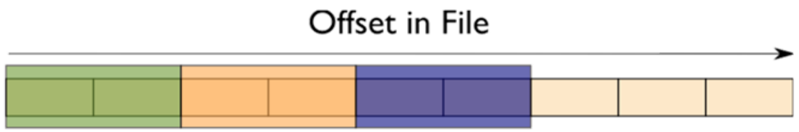
# Implications of Locking in Concurrent Access

The left diagram shows a row-block distribution of data for three processes. On the right we see how these accesses map onto locking units in the file.

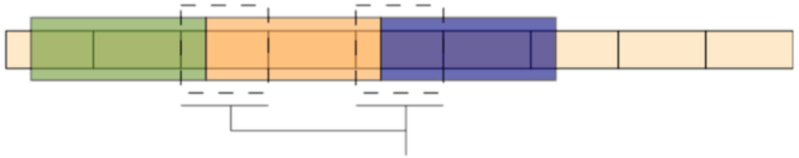
In this example a header (black) has been prepended to the data. If the header is not aligned with lock boundaries, false sharing will occur.

In this example, processes exhibit a block-block access pattern (e.g. accessing a subarray). This results in many interleaved accesses in the file.

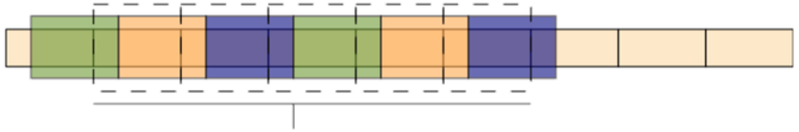



When accesses are to large contiguous regions, and aligned with lock boundaries, locking overhead is minimal.



These two regions exhibit *false sharing*: no bytes are accessed by both processes, but because each block is accessed by more than one process, there is contention for locks.

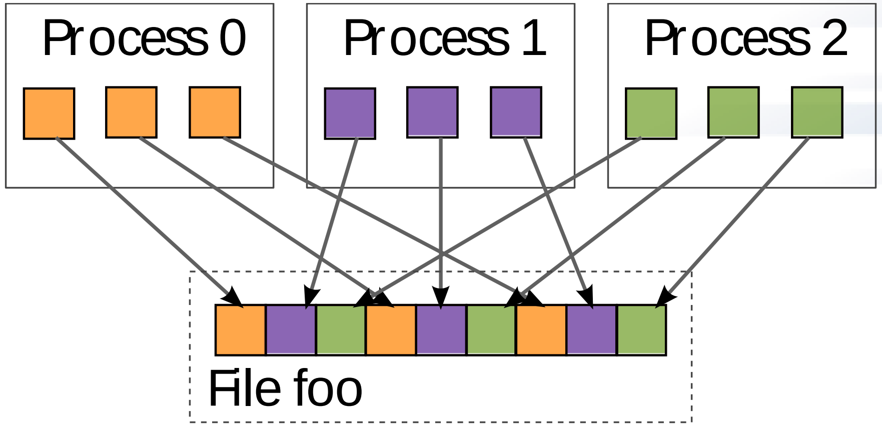


When a block distribution is used, sub-rows cause a higher degree of false sharing, especially if data is not aligned with lock boundaries.

# I/O Transformations

**Software between the application and the file system performs transformations, primarily to improve performance.**

- Goals of transformations:
  - Reduce number of operations to PFS (avoiding latency)
  - Avoid lock contention (increasing level of concurrency)
  - Hide number of clients (more on this later)
- With "transparent" transformations, data ends up in the same locations in the file as it would have been normally
  - i.e., the file system is still aware of the actual data organization

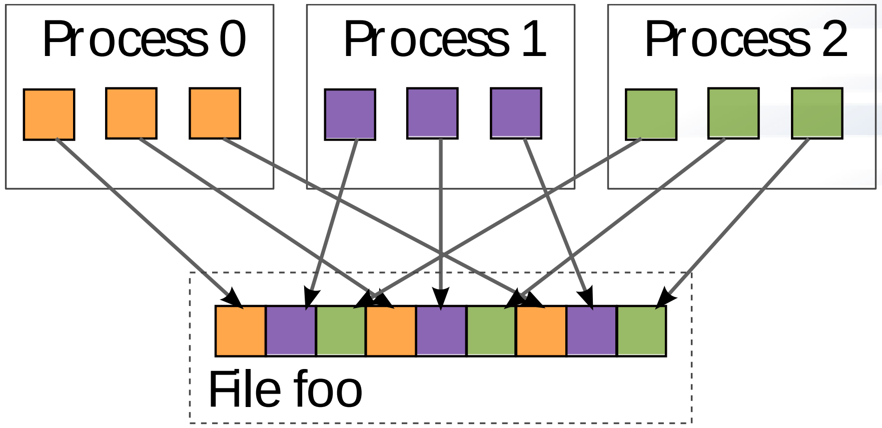



When we think about I/O transformations, we consider the mapping of data between application processes and locations in file.

# I/O Transformations

**Software between the application and the file system performs transformations, primarily to improve performance.**

- We will tour through a few examples of data transformations in the following slides
- The important thing to remember is that software already exists to do these things for you in HDF5, PnetCDF, ADIOS, and MPI-IO
- If you find yourself replicating these optimizations by hand, look around to see if you can find an off-the-shelf solution

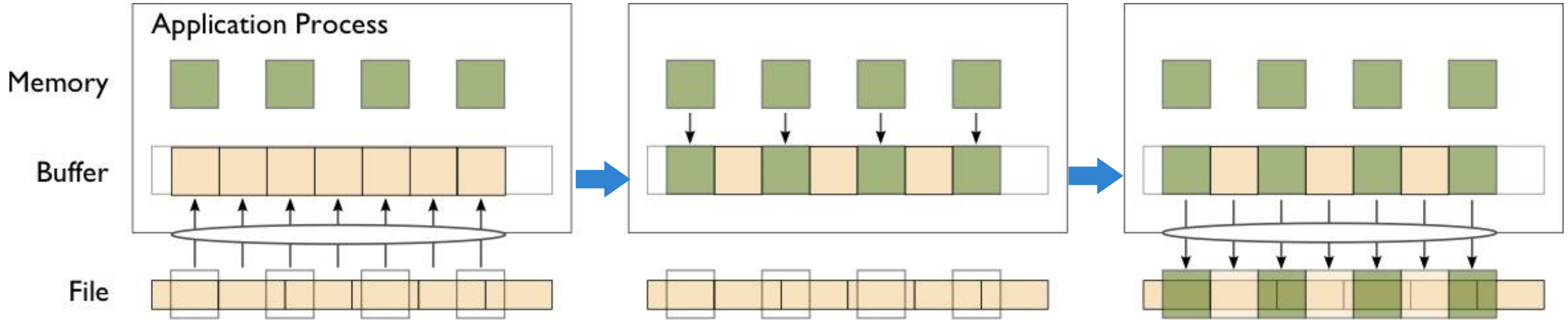



When we think about I/O transformations, we consider the mapping of data between application processes and locations in file.

# Reducing Number of Operations

**Because most operations go over multiple networks, I/O to a PFS incurs more latency than with a local FS.** *Data sieving* is a technique to address I/O latency by combining operations:

- When reading, application process reads a large region holding all needed data and pulls out what is needed
- When writing, three steps required (below)
- Somewhat counter-intuitive: do extra I/O to avoid contention




**Step 1**: Data in region to be modified are read into intermediate buffer (1 read).

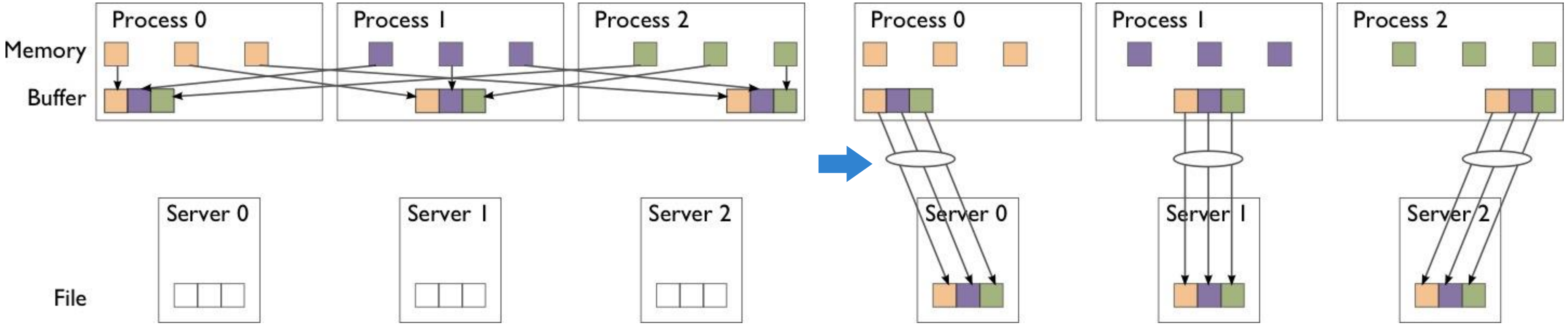
**Step 2**: Elements to be written to file are replaced in intermediate buffer.

**Step 3**: Entire region is written back to storage with a single write operation.

# Avoiding Lock Contention

**We can reorder data among processes to avoid lock contention.** *Two-phase I/O* splits I/O into a data reorganization phase and an interaction with the storage system (two-phase write depicted):

- Data exchanged between processes to match file layout
- $0^{th}$ phase determines exchange schedule (not shown)




**Phase 1**: Data are exchanged between processes based on organization of data in file.

**Phase 2**: Data are written to file (storage servers) with large writes, no contention.

# Two-Phase I/O Algorithms
## (or, You don't want to do this yourself…)

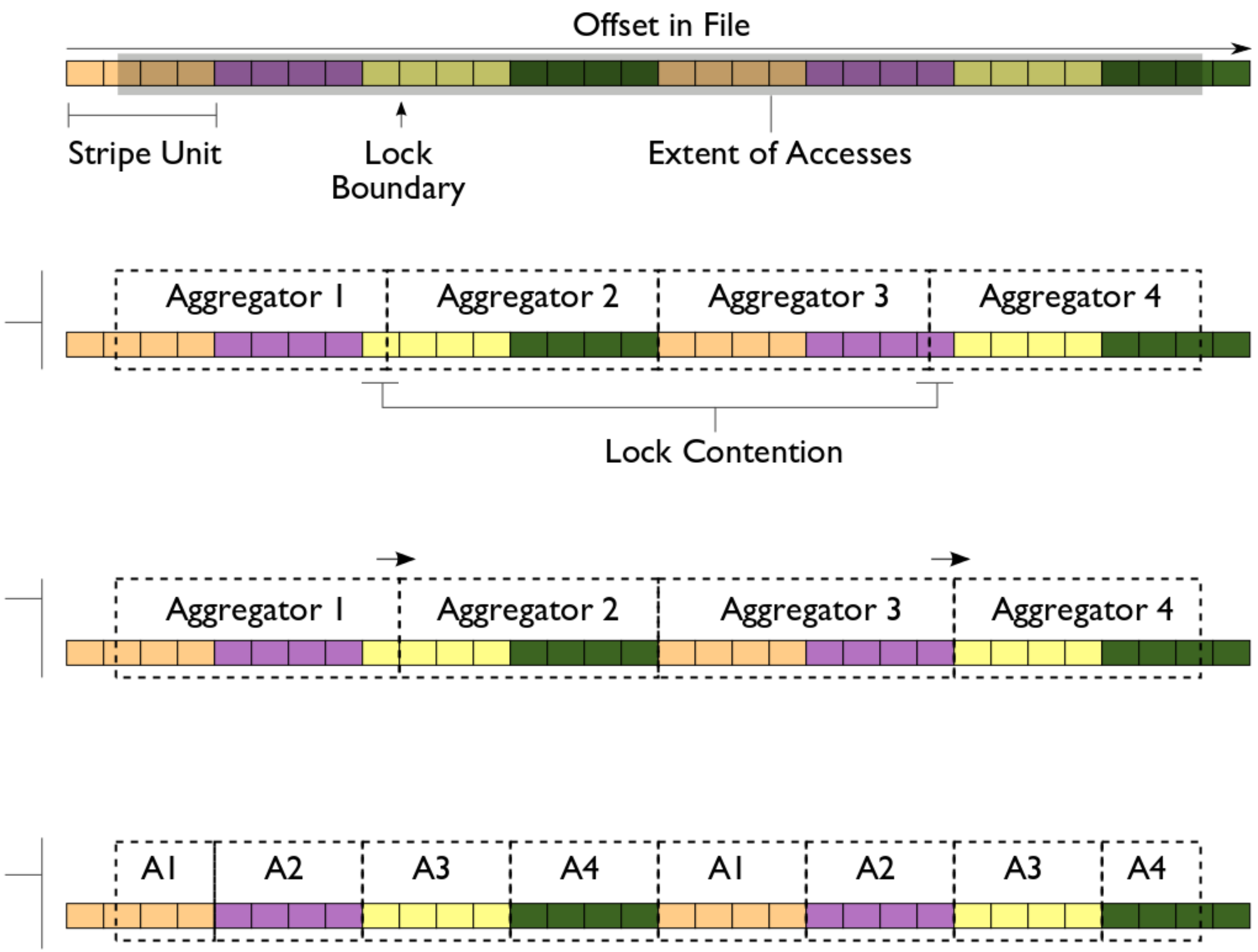
For more information, see W.K. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," SC2008, November, 2008.

Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):

One approach is to evenly divide the region accessed across aggregators.

Aligning regions with lock boundaries eliminates lock contention.

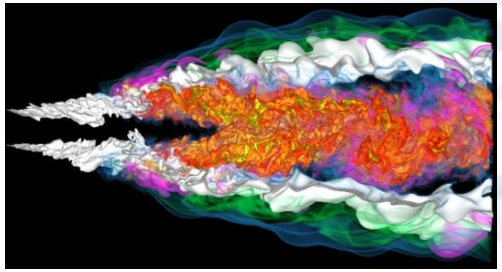
Mapping aggregators to servers reduces the number of concurrent operations on a single server and can be helpful when locks are handed out on a per-server basis (e.g., Lustre).

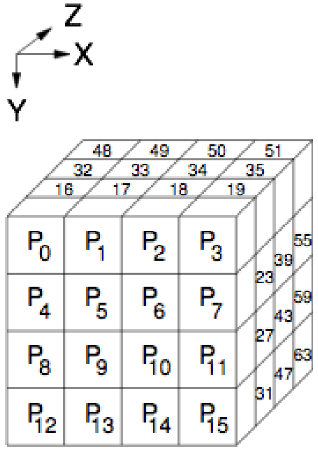


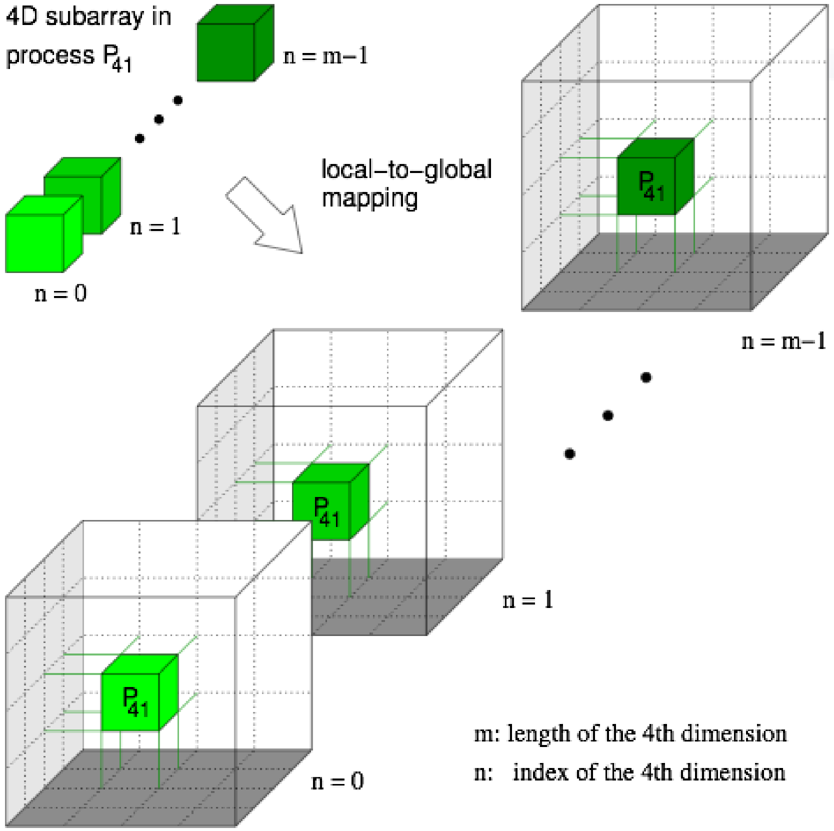

Today's systems also choose aggregators that are "closest" to storage

# S3D Turbulent Combustion Code

- S3D is a turbulent combustion application using a direct numerical simulation solver from Sandia National Laboratory
- Checkpoints consist of four global arrays
  - 2 3-dimensional
  - 2 4-dimensional
  - 50x50x50 fixed subarrays






Thanks to Jackie Chen (SNL), Ray Grout (SNL), and Wei-Keng Liao (NWU) for providing the S3D I/O benchmark, Wei-Keng Liao for providing this diagram, C. Wang, H. Yu, and K.-L. Ma of UC Davis for image.

# Impact of Transformations on S3D I/O

- Testing with PnetCDF output to single file, three configurations, 16 processes
  - All MPI-IO optimizations (collective buffering and data sieving) disabled
  - Independent I/O optimization (data sieving) enabled
  - Collective I/O optimization (collective buffering, a.k.a. two-phase I/O) enabled

Application did the same thing in every case

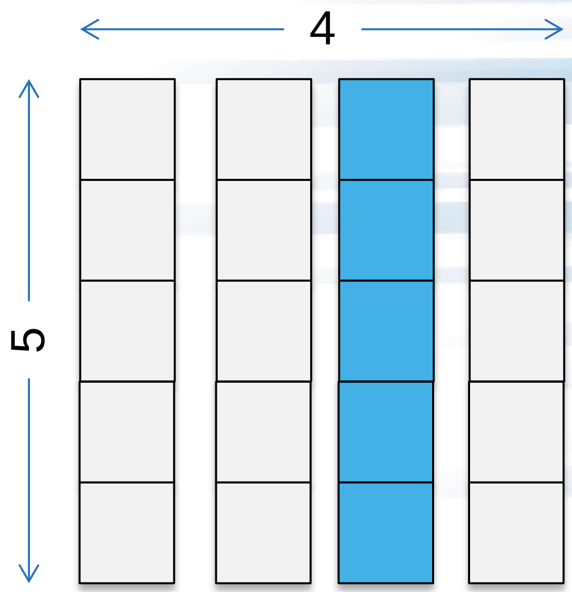
| | Coll. Buffering and Data Sieving Disabled | Data Sieving Enabled | Coll. Buffering Enabled (including Aggregation) |
|---|---|---|---|
| POSIX writes | 102,401 | 81 | **5** |
| POSIX reads | 0 | 80 | 0 |
| MPI-IO writes | 64 | 64 | 64 |
| Unaligned in file | 102,399 | 80 | 4 |
| Total written (MB) | 6.25 | **87.11** | 6.25 |
| Runtime (sec) | 1443 | 11 | 6.0 |
| Avg. MPI-IO time per proc (sec) | **1426.47** | 4.82 | 0.60 |



U.S. DEPARTMENT OF ENERGY | Office of Science | ECP EXASCALE COMPUTING PROJECT

# HANDS-ON 6: reading with MPI-IO

- Slightly different: all processes read one row
  - For simplicity, same row
- File view will be more complicated, use MPI "Subarray" datatype
- In C, array access is described in "row-major"
  - `array_size[0] = 5; array_size[1] = 4;`
- File view uses derived 'subarray', not built-in MPI_INT
- Location in file given with subarray type; no offset in `MPI_File_read_all`
  - Still provide a "buffer, count, datatype" tuple for memory layout






U.S. DEPARTMENT OF ENERGY | Office of Science

EXASCALE COMPUTING PROJECT

# Solution fragments

## Type creation

```c
/* In C-order the arrays are row-major:
 *
 * |-----|
 * |-----|
 * |-----|
 *
 * The 'sizes' of the above array would be 3,5
 * The last column would be a "subsize" of 3,1
 * And a "start" of 0,5 */

sizes[0] = nprocs; sizes[1] = XDIM;
sub[0] = nprocs;   sub[1] = 1;
starts[0] = 0;     starts[1] = XDIM/2;

MPI_Type_create_subarray(NDIMS,
   sizes, sub, starts,
   MPI_ORDER_C, MPI_INT, &subarray);
MPI_Type_commit(&subarray);
```

## File view and read

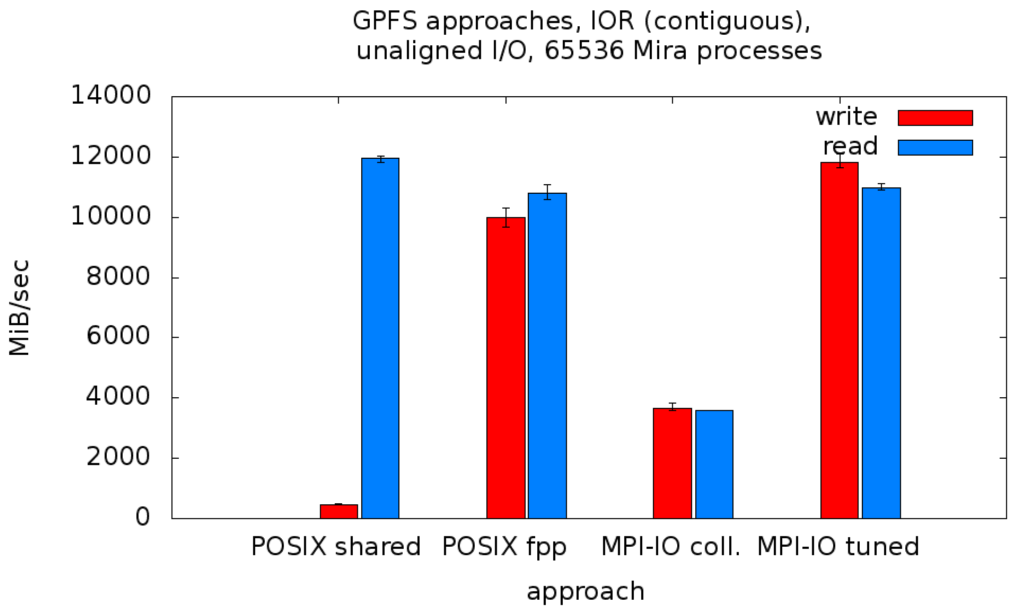
```c
MPI_CHECK(MPI_File_set_view(fh, sizeof(header),
  MPI_INT, subarray, "native", info));
MPI_Type_free(&subarray);
MPI_CHECK(MPI_File_read_all(fh,
  read_buf, nprocs, MPI_INT, MPI_STATUS_IGNORE);
```

# Hands on 6 continued: Darshan

- How does this workload differ from the write?
- Change the ‘read_all’ to an independent ‘read’
  - What do you think the Darshan output will say?  Find out.

# GPFS Access three ways

- POSIX shared vs MPI-IO collective
  - Locking overhead for unaligned writes hits POSIX hard
- Default MPI-IO parameters not ideal
  - Reported to IBM; simple tuning brings MPI-IO back to parity
  - “Vendor Defaults” might give you bad first impression
- File per process (fpp) extremely seductive, but entirely untenable on current generation.

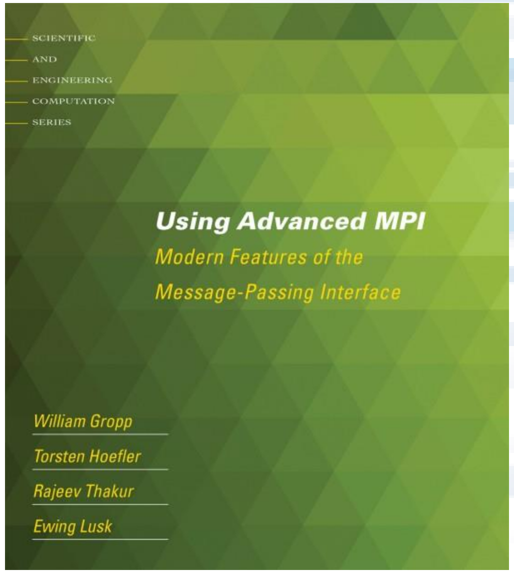


# MPI-IO Takeaway

- Sometimes it makes sense to build a custom library that uses MPI-IO (or maybe even MPI + POSIX) to write a custom format
  - e.g., a data format for your domain already exists, need parallel API
- We've only touched on the API here
  - There is support for data that is noncontiguous in file and memory
  - There are independent calls that allow processes to operate without coordination
- In general we suggest using data model libraries
  - They do more for you
  - Performance can be competitive

# MPI-IO References

- On Cray systems, “man intro_mpi” for 3,000 lines of tuning parameters, debug configuration
- *Using Advanced MPI*, Gropp, Hoeffler, Thakur, Lusk
  - Chapter on MPI I/O routines covers entire API as well as consistency semantics



# Up next: Parallel-NetCDF – hiding MPI-IO details