MPI-IO

Argonne National Laboratory is a
U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC.

MPI-IO

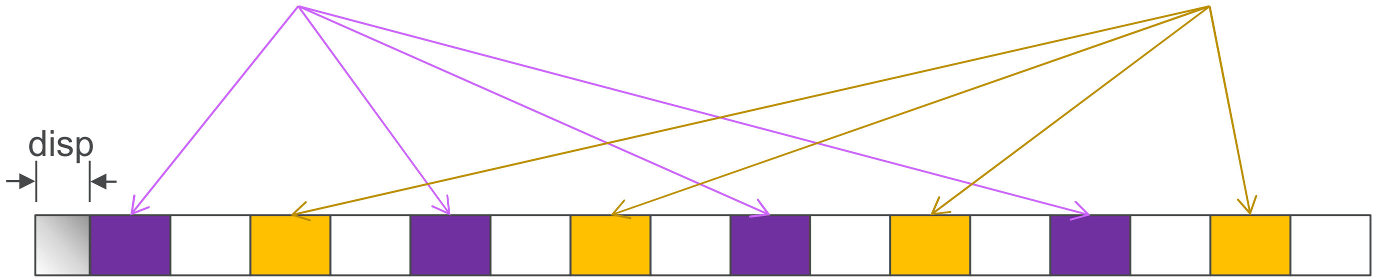
- I/O interface specification for use in MPI apps
- Data model is same as POSIX: stream of bytes in a file
- Features many improvements over POSIX:
  - Collective I/O
  - Noncontiguous I/O with MPI datatypes and file views
  - Nonblocking I/O
  - Fortran bindings (and additional languages)
  - System for encoding files in a portable format (external32)
    - Not self-describing – just a well-defined encoding of types
- Implementations available on most platforms (more later)

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

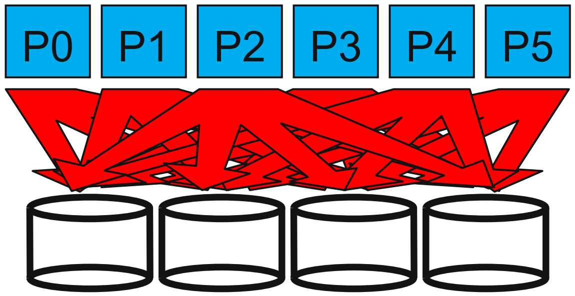
```
MPI_File_open(COMM, name, mode,
          info, fh);
MPI_File_set_view(fh, disp, etype,
          filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
          datatype, status);
```

```
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);
```

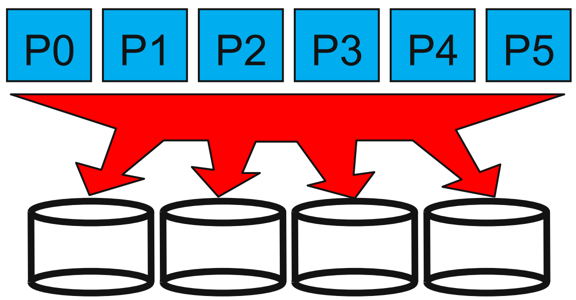


INDEPENDENT AND COLLECTIVE I/O

- Independent I/O operations
  - Specify only what a single process will do
  - Do not pass on relationships between I/O on other processes
- Many applications have alternating phases of computation and I/O
  - During I/O phases, all processes read/write data
  - We can say they are collectively accessing storage
- Collective I/O is coordinated access to storage by a group of processes
  - Collective I/O functions are called by all processes participating in I/O
  - Allows I/O layers to know more about access as a whole, more opportunities for optimization in lower software layers, better performance



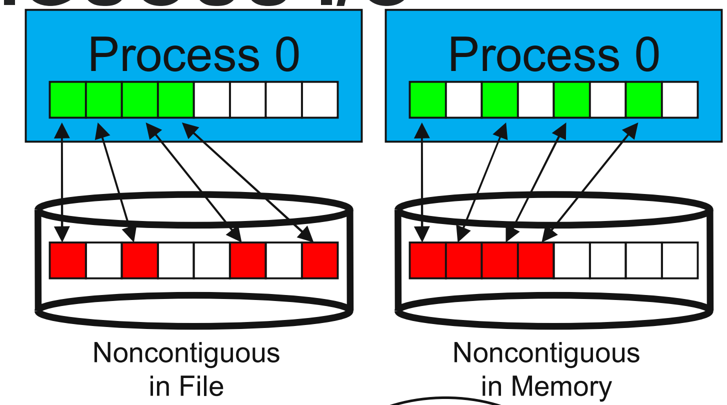
Independent I/O

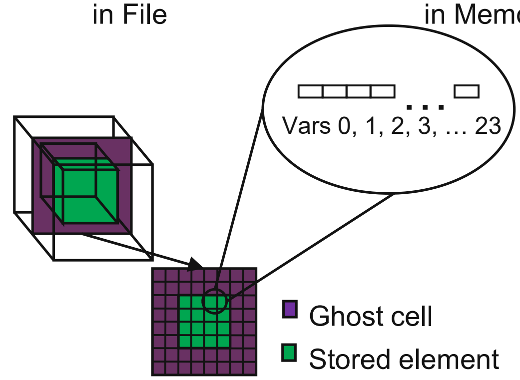


Collective I/O

CONTIGUOUS AND NONCONTIGUOUS I/O

- Contiguous I/O moves data from a single memory block into a single file region
- Noncontiguous I/O has three forms:
  – Noncontiguous in memory
  – Noncontiguous in file
  – Noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g., block decomposition)
- Describing noncontiguous accesses with a single operation passes more knowledge to I/O system



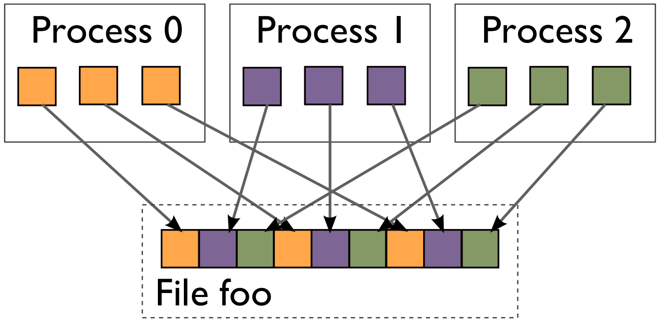


Extracting variables from a block and skipping ghost cells will result in noncontiguous I/O

I/O TRANSFORMATIONS

Software between the application and the PFS performs transformations, primarily to improve performance

- Goals of transformations:
  - Reduce number of I/O operations to PFS (avoid latency, improve bandwidth)
  - Avoid lock contention (eliminate serialization)
  - Hide huge number of clients from PFS servers
- "Transparent" transformations don't change the final file layout
  - File system is still aware of the actual data organization
  - File can be later manipulated using serial POSIX I/O



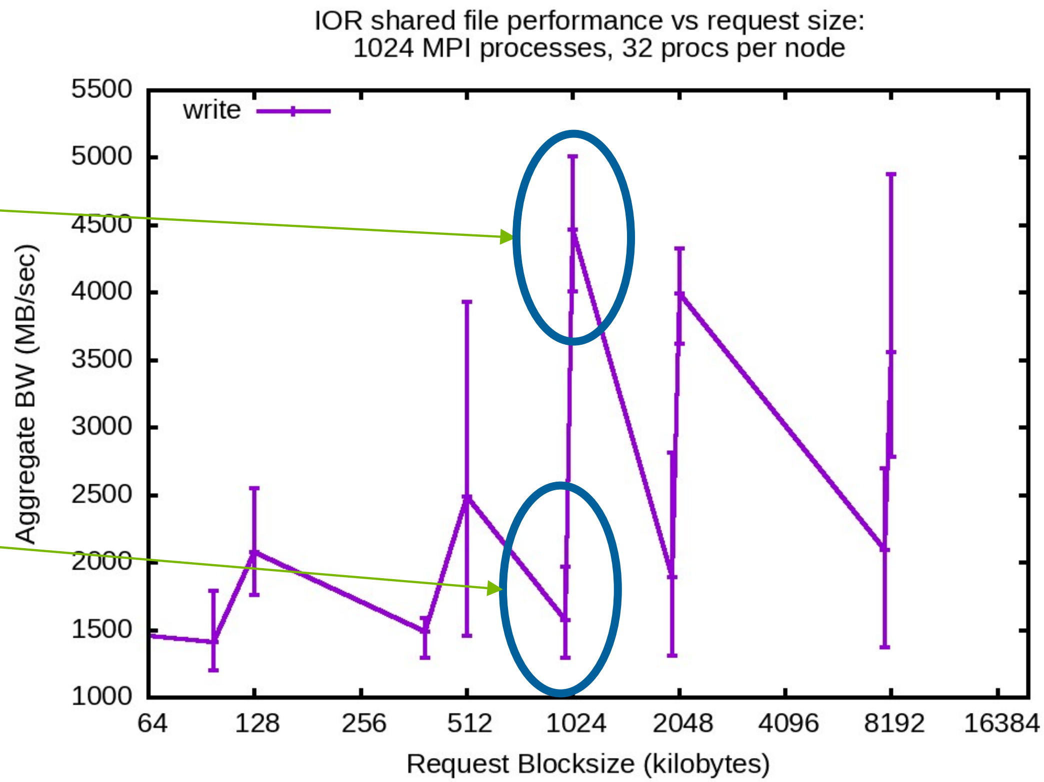
When we think about I/O transformations, we consider the mapping of data between application processes and locations in file

REQUEST SIZE AND I/O RATE

Request matches Lustre “stripe size”: good performance with low variability

Small deviations from “power of two” (e.g. 1024k vs 10^6) can tank performance

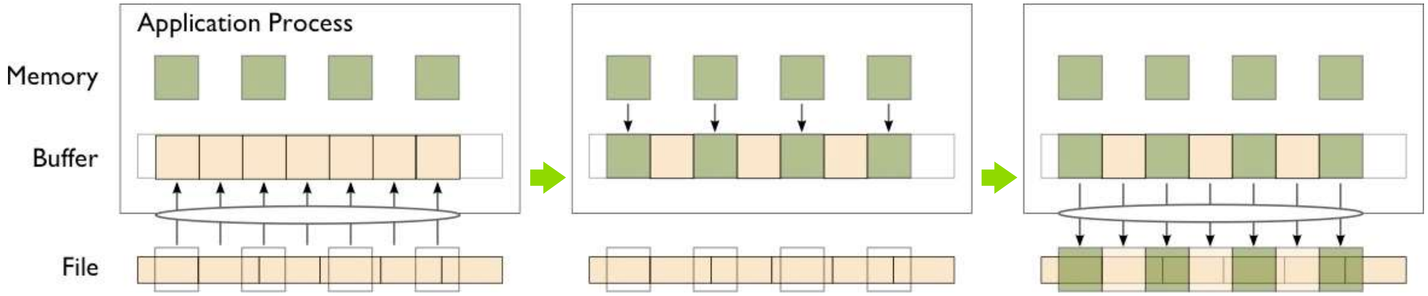
In general, larger requests better.



Tests run on 1K processes of HPE/Cray Theta at Argonne

REDUCING NUMBER, INCREASING SIZE OF OPERATIONS

- **Because most operations go over the network, 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)



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

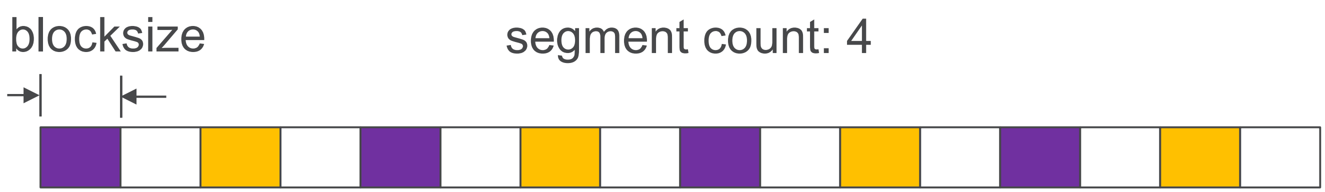
Step 2: Elements to be written to file are replaced in intermediate buffer.

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

NONCONTIG WITH IOR

- IOR can describe access with an MPI datatype
  - `--mpiio.useStridedDatatype –b … -s …`

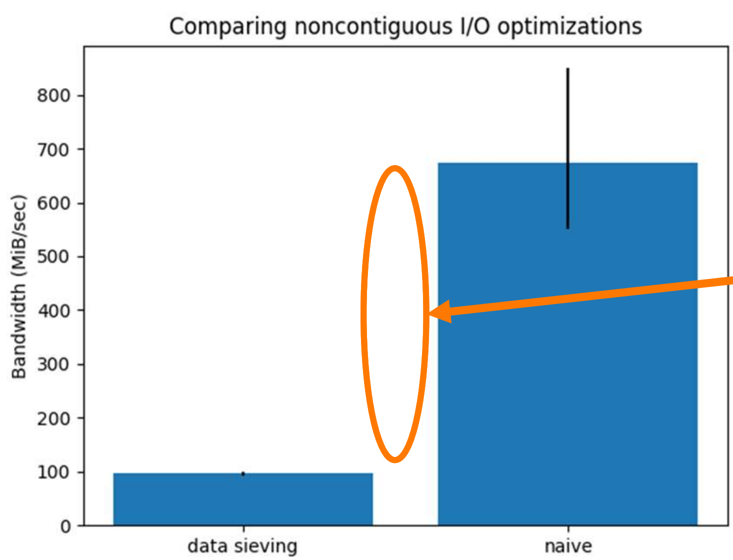
- (buggy in recent versions: use 4.0rc1 or newer)



DATA SIEVING IN PRACTICE

Not always a win, particularly for writing:

- Enabling data sieving instead made writes slower: why?
  - Locking to prevent false sharing (not needed for reads)
  - Multiple processes per node writing simultaneously
  - Internal ROMIO buffer too small, resulting in write amplification [1]



	Naiive	Data Sieving
MPI-IO writes	192	192
MPI-IO Reads	0	0
Posix Writes	192000	192000
Posix Reads	0	192015
MPI-IO bytes written	1 920 000 000	1 920 000 000
MPI-IO bytes read	0	0
Posix bytes read	0	100 039 006 128
[1] Posix bytes written	1 920 000 000	100 564 552 704

Selected Darshan statistics

https://github.com/radix-io/io-sleuthing/tree/main/examples/noncontig

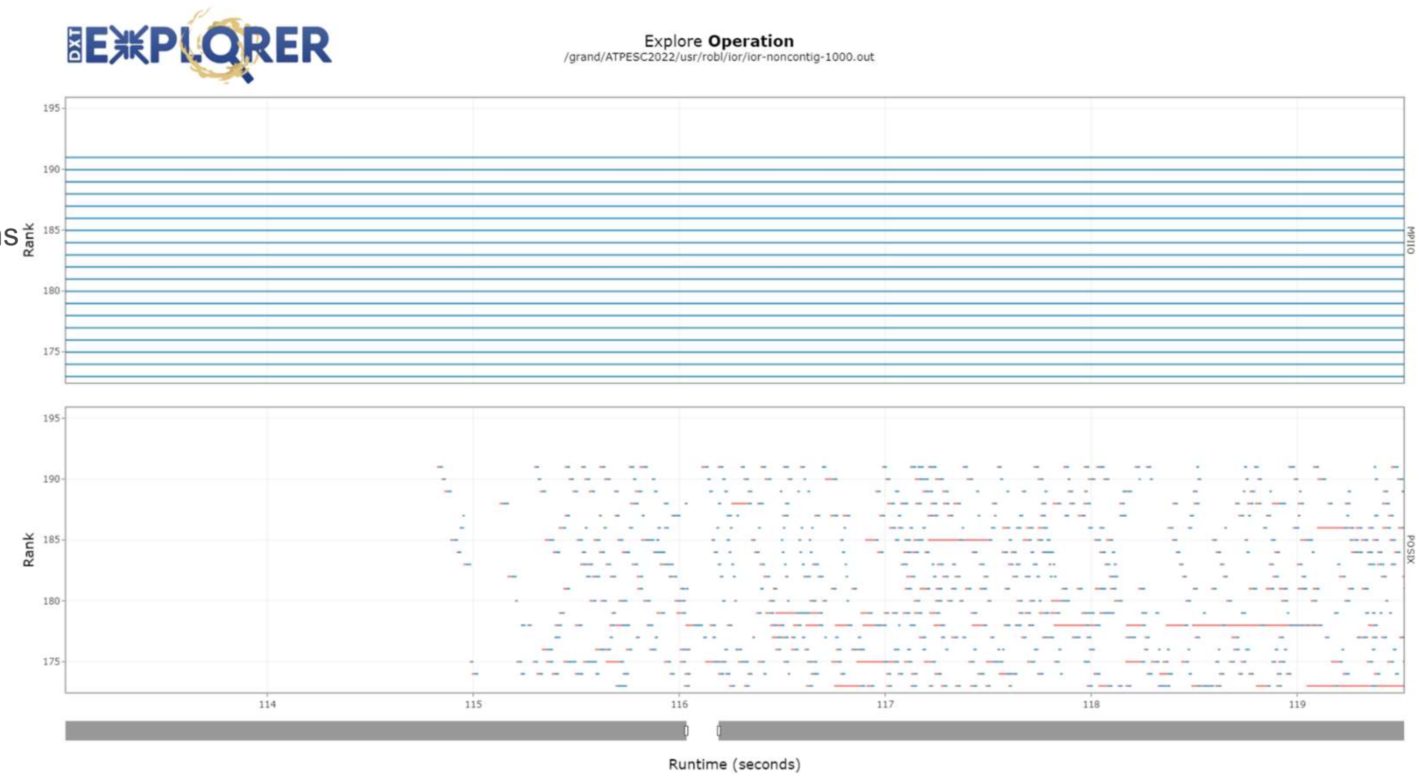
DATA SIEVING: TIME LINE

Top: MPI I/O call describing noncontiguous regions

One MPI I/O call (top) turns into many POSIX operations (below)

Independent: no coordination possible. Each process does its own data sieving.

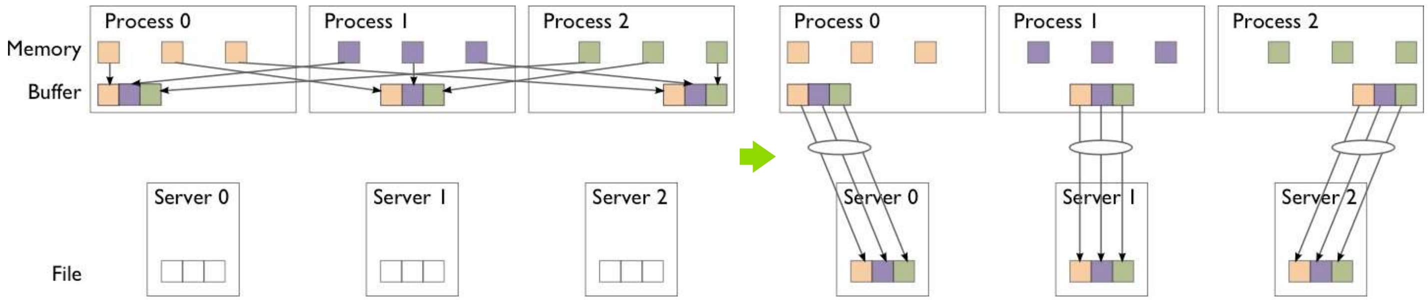
Gaps between operations show lock acquisition.



https://github.com/hpc-io/dxt-explorer Interactive log analysis tool by Jean Luca Bez

AVOIDING LOCK CONTENTION

- **To avoid lock contention when writing to a shared file, we can reorganize data between processes**
- *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.

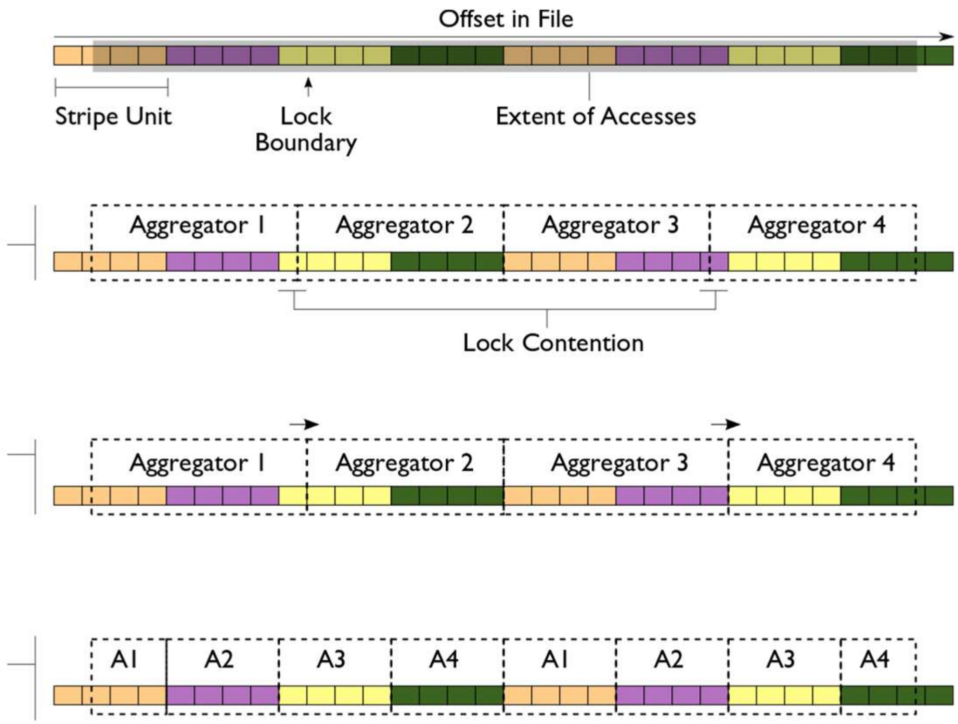
TWO-PHASE I/O ALGORITHMS

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).

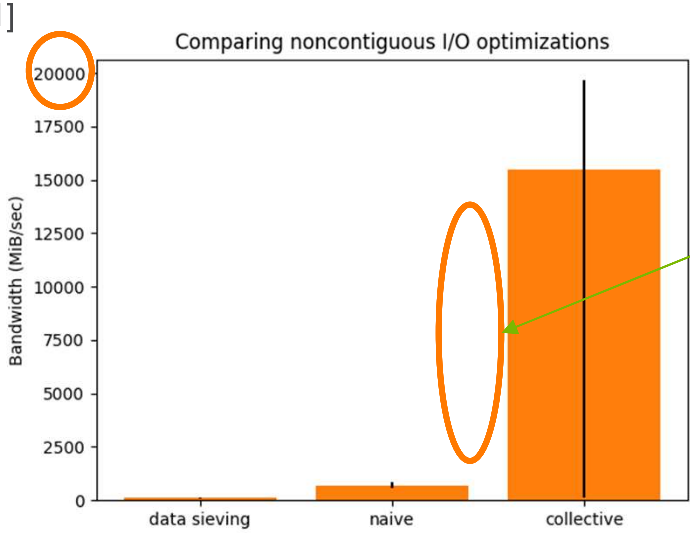


For more information, see W.K. Liao and A. Choudhary, “Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols,” SC2008, November 2008.

TWO-PHASE I/O IN PRACTICE

- Consistent performance independent of access pattern
  - Note re-scaled y axis [1]
- No write amplification, no read-modify-write
- Some network communication but networks are fast
- Requires “temporal locality” -- not great if writes “skewed”, imbalanced, or some process enter collective late.
- (Yes, those are some “impressive” error bars…)

[1]



[2]

	Naiive	Data Sieving	Two-phase
MPI-IO writes	192	192	**192**
MPI-IO Reads	0	0	**0**
Posix Writes	192000	192000	**1832**
Posix Reads	0	192015	**0**
MPI-IO bytes written	1 920 000 000	1 920 000 000	**1 920 000 000**
MPI-IO bytes read	0	0	**0**
Posix bytes read	0	100 039 006 128	**0**
Posix bytes written	1 920 000 000	100 564 552 704	**1 920 000 000**

Selected Darshan statistics

https://github.com/radix-io/io-sleuthing/tree/main/examples/noncontig

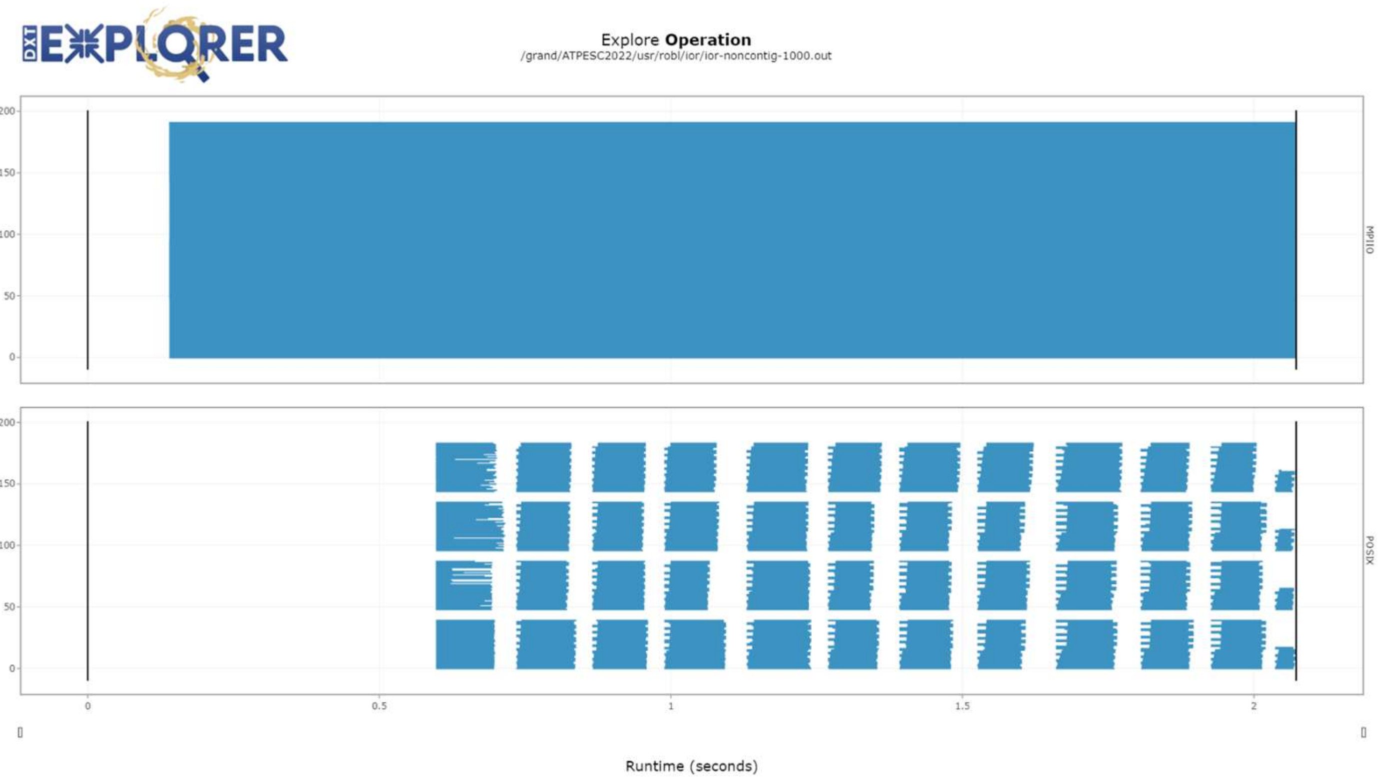
TWO-PHASE I/O: TIME LINE

Top: collective MPI I/O call describing noncontiguous regions

One collective MPI I/O call per process: library transforms request.

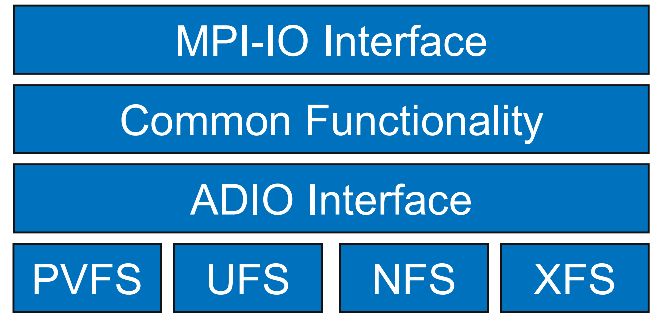
Lustre-specific optimization: select processes and request sizes based on file stripe size, stripe count.

Gaps between operations show data exchange over network



MPI-IO IMPLEMENTATIONS

- Different MPI-IO implementations exist
- Today two better-known ones are:
  – ROMIO from Argonne National Laboratory
    • Leverages MPI-1 communication
    • Supports local file systems, network file systems, parallel file systems
      – UFS module works on GPFS, Lustre, and others
    • Includes data sieving and two-phase optimizations
    • Basis for many vendor MPI implementations
    • https://wordpress.cels.anl.gov/romio/
  – OMPIO from OpenMPI
    • Emphasis on modularity and tighter integration into MPI implementation
    • https://docs.open-mpi.org/en/v5.0.x/faq/ompio.html



ROMIO's layered architecture.