File System Semantics Requirements of HPC Applications

Chen Wang

University of Illinois at Urbana-Champaign

Kathryn Mohror

Lawrence Livermore National Laboratory

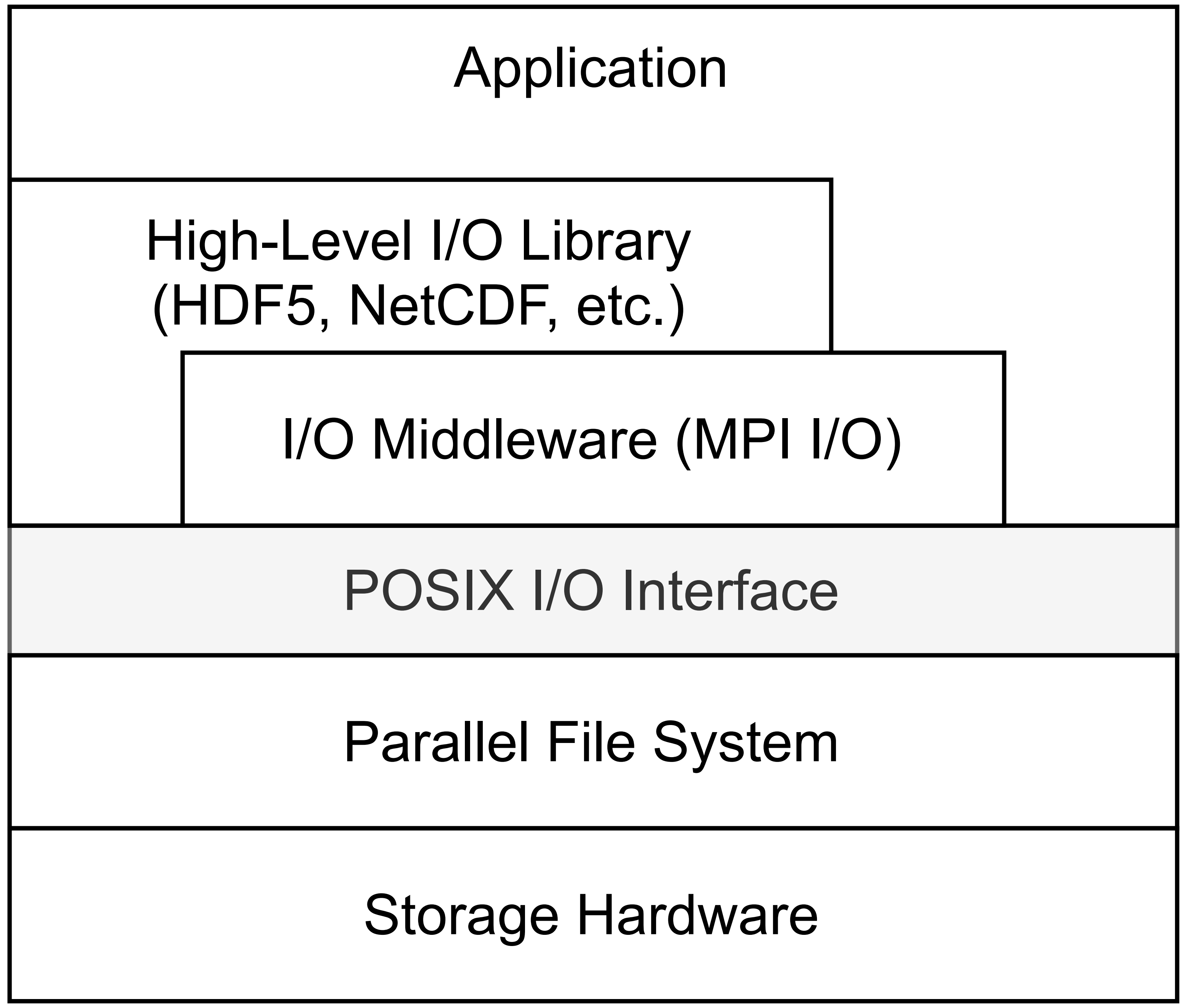
Marc Snir

University of Illinois at Urbana-Champaign

Parallel I/O in Modern Systems

Key components

- I/O Stack
  - POSIX interface
  - I/O libraries: HDF5, ADIOS, or MPI-IO
  - Parallel File Systems
    - Burst buffer file systems
    - Non-volatile memory file systems
- I/O performance depends on I/O access patterns, PFS configurations, etc.



Motivation

POSIX I/O interface and semantics

- Most widely-deployed PFSs (e.g., Lustre and GPFS) support POSIX semantics.
- POSIX was designed decades ago for use by a single machine with a single storage device.
  - Not for highly-concurrent operations to PFS.
- The primary challenge arises from the strict adherence to POSIX semantics. The POSIX standard states:
  - *Any successful **read** from each byte position in the file that was modified by the **last write** shall return the data specified by the **write** for that position until such byte positions are again modified*.
  - *Any subsequent successful **write** to the same byte position in the file shall **overwrite** that file data*.

Motivation

File systems with relaxed semantics

- Ways to ensure POSIX semantics:
  - Disable page cache
  - Use some locking mechanism (e.g., Lustre, GPFS)
- POSIX semantics is usually an overkill.
  - Many new file systems with relaxed semantics are developed.
  - Different jobs normally do not access the same file
  - It is uncommon that multiple processes perform write-after-write or read-after-write.
    - We need experimental data to find out!

Key Questions

- What are the common I/O characteristics of HPC applications?
  - How are files accessed? Sequentially? Strided? Any write-after-write?
  - Which metadata operations are used? Is it possible to relax some metadata operations too?
- Given an application, will it run correctly on a file system with weaker consistency semantics?
  - It is challenging to determine the semantics needed by an application since I/O patterns depend on execution flow and on the behavior of high-level I/O libraries.
  - PFSs relax POSIX semantics in different ways and are often poorly documented.
  - There are no accepted categorizations or definitions of the relaxed semantics implemented by PFSs.

Our Goal

- Provide tools and data to…
    - study the I/O characteristics of applications.
    - understand the semantics requirements of the applications, so users can choose the best PFS according to their need.
- For PFS and I/O library developers
    - Make better optimization decisions
    - Know to what extend consistency semantics can be relaxed without breaking the targeted applications.

How?

- Provide terminology for the categorization of the consistency semantics of PFSs.
  - We list several example PFSs for each category.
- Develop a method for detecting I/O accesses that can cause conflicts under weaker consistency models.
- We present the I/O characteristics of 17 HPC applications using POSIX or I/O libraries.
  - All codes and traces are publicly available.

Categorization of PFS consistency semantics

This categorization considers only data consistency semantics. We define four commonly used consistency models based on **when a write made by one process becomes visible to other processes**.

1. Strong consistency semantics
2. Commit consistency semantics
3. Session consistency semantics
4. Eventual consistency semantics

Consistency Semantics	File Systems
Strong Consistency	GPFS, Lustre, GekoFS, BeeGFS, BatchFS, OrangeFS
Commit Consistency	BCFS, UnifyFS, SymphonyFS, BurstfS
Session Consistency	NFS, AFS, DDN IME, Gfarm/BB
Eventual Consistency	PLFS, echofs, MarFS

Categorization of PFS consistency semantics

Strong consistency semantics

- Most HPC systems do not have global clocks, we employ happens-before order (denoted by $\rightarrow$) defined by the program order and communication.
- Strong consistency semantics is define as follows:
  - A read $\boldsymbol{r}$ from a byte returns the value written by a write $\boldsymbol{w}$ to the byte if $\boldsymbol{w} \rightarrow \boldsymbol{r}$, and for any other write $\boldsymbol{w'}$ to the same byte either $\boldsymbol{w'} \rightarrow \boldsymbol{w}$ or $\boldsymbol{r} \rightarrow \boldsymbol{w'}$.
- Most general-purpose PFSs support strong consistency semantics.
  - Lustre, GPFS, BeeGFS, etc.

Categorization of PFS consistency semantics

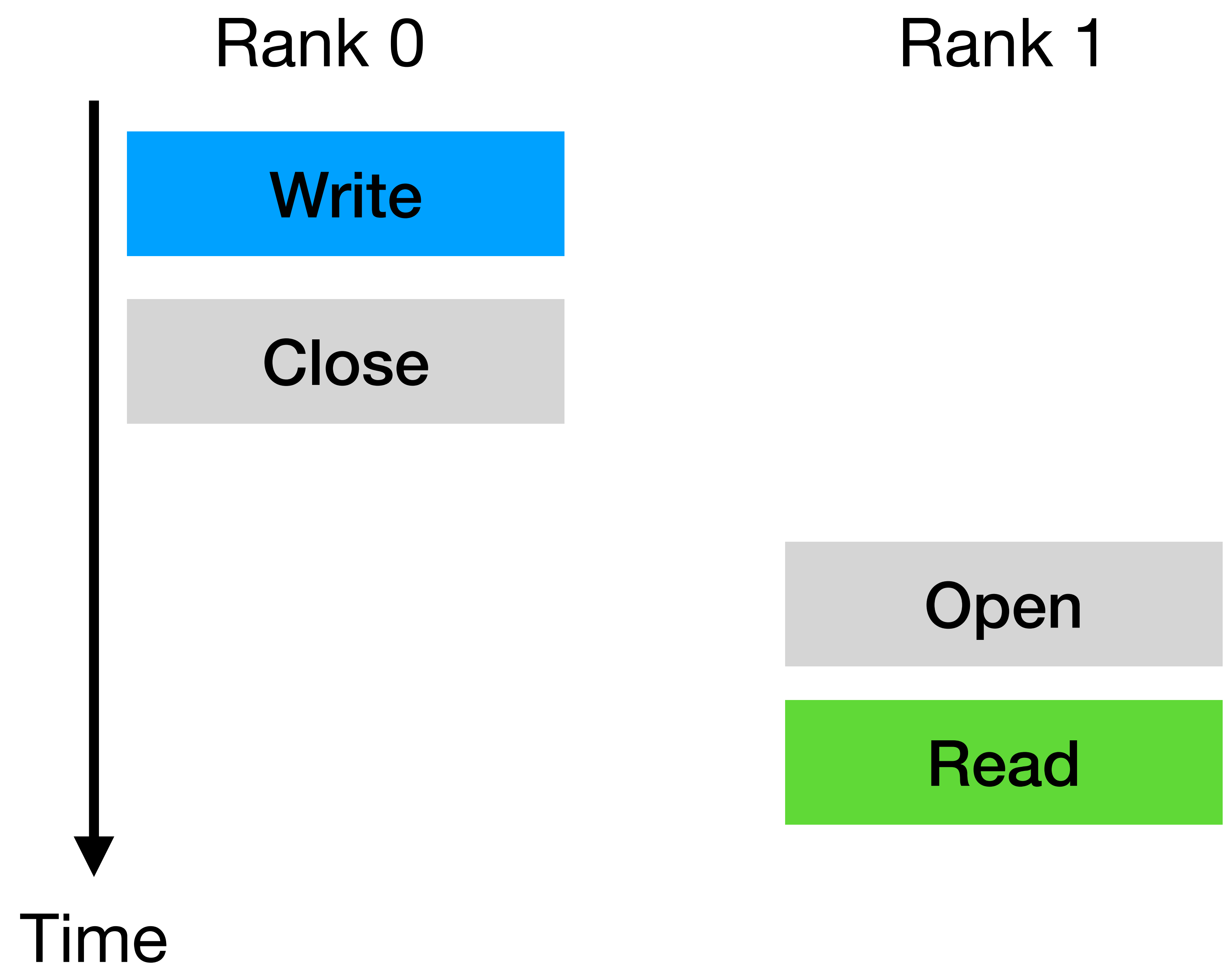
Commit consistency semantics

- We define commit consistency semantics as a less strict consistency model, where “commit” operations are explicitly executed by processes.
- I/O updates performed by a process become globally visible no later than the return of the commit operation.
- Many user-level and BB PFSs belong to this category.
  - UnifyFS, BurstFS, SymphonyFS, etc.

Categorization of PFS consistency semantics

Session consistency semantics

- Also known as close-to-open semantics.
- A write becomes visible to others no later than when the modified file is closed by the writing process and subsequently opened by the reading process.
- Example PFSs: NFS, DDN IME, Gfarm/BB, etc.



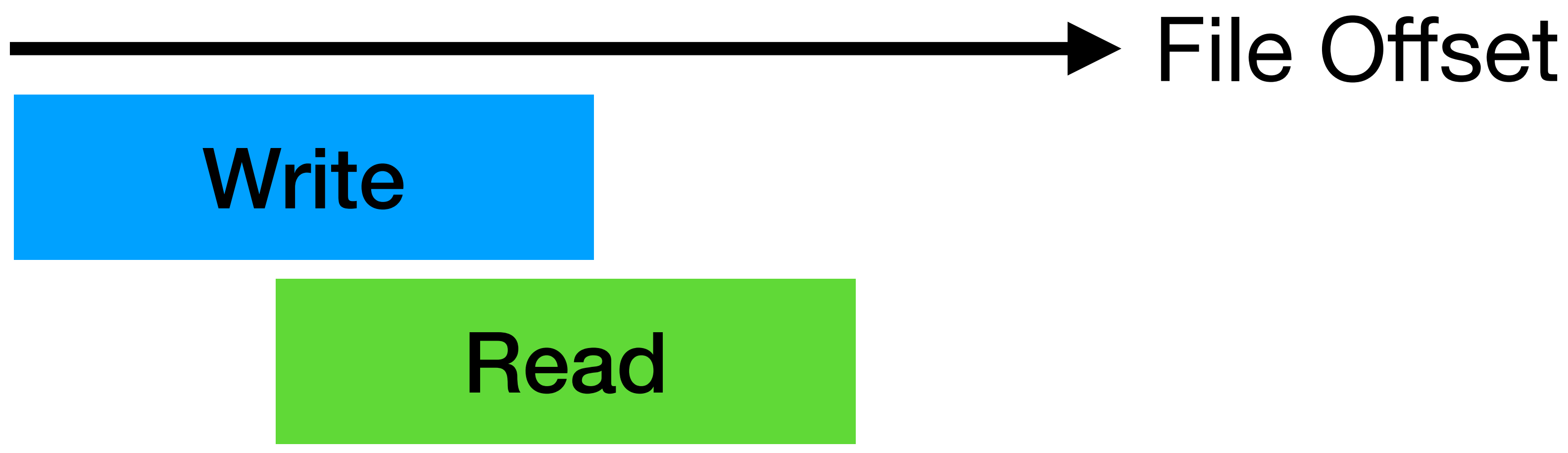
Overlaps and Conflicts

- Overlap: two I/O operations access an overlapping area of the same file.
- Conflict: overlap and can potentially lead to data hazards if POSIX semantics is weakened.
- Four potential conflicts:
    - **RAW-[S|D]**: read-after-write by the same process (S) or by different processes.
    - **WAW-[S|D]**: write-after-write by the same process (S) or by different processes.

Overlaps and Conflicts

Detecting Overlaps

- Define an I/O operation as a tuple *(t, r, os, oe, type)*
    - *t: timestamp*
    - *r: rank*
    - *os: start offset*
    - *oe: end offset*
    - *type: write or read*



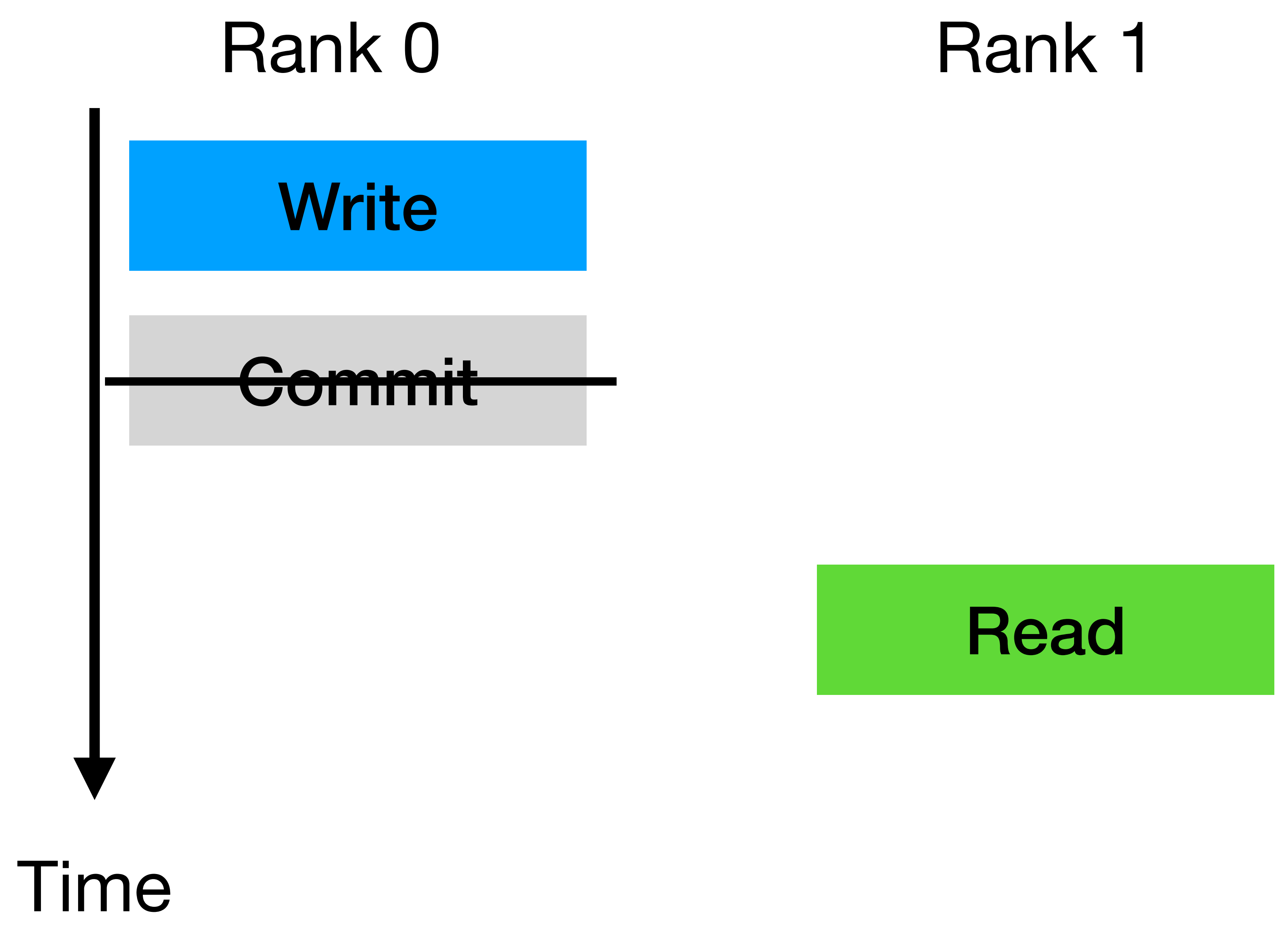
Algorithm 1 Detecting overlaps

Sort tuples by os
for each tuple T_i **do**
    for each tuple T_j, $j > i$ **do**
        if $os_j > oe_i$ **then**
            break ▷ subsequent tuples will not overlap with T_i
        else
            $P[r_i, r_j] \leftarrow 1$ ▷ T_i and T_j overlap

Overlaps and Conflicts

Detecting Conflicts

- Two tuples $(t_1, r_1, os_1, oe_1, type_1)$ and $(t_2, r_2, os_2, oe_2, type_2)$ are a conflict pair if the following conditions are satisfied:
  - They overlap
  - The first operation is a write: $type_1 = write$
  - For commit semantics: r_1 does not execute any commit operation after t_1 and before t_2.
  - For session semantics: there is no close operation on r_1 with t_c and open operation on r_2 with t_o so that $t_1 < t_c < t_o < t_2$.



Experiments

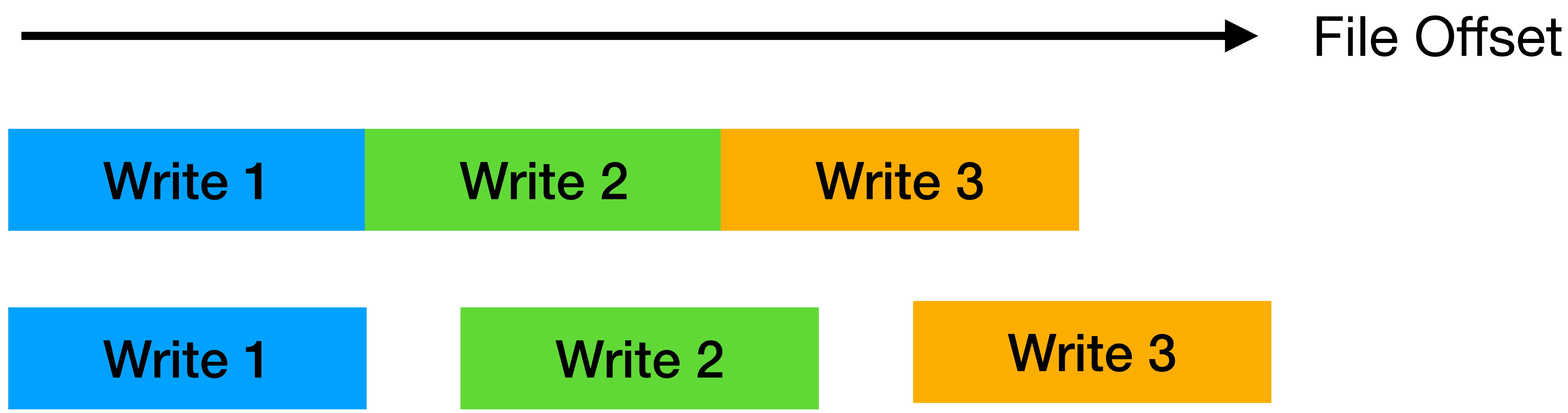
- Quartz at LLNL
  - Intel Xeon E5-2695, 36 cores, 128GB memory.
  - PFS: Lustre.
- 17 HPC applications using different I/O libraries.
- Two scales:
  - 32 nodes * 32 ranks/node.
  - 8 nodes * 8 ranks / node.
- Traces are generated using Recorder
  - The codes are available at:
    - https://github.com/uiuc-hpc/Recorder
    - https://pypi.org/project/recorder-viz
  - Dataset: https://doi.org/10.6075/J0Z899X4

App	I/O library
FLASH	HDF5
Nek5000	POSIX
QMCPACK	HDF5
VASP	POSIX
LBANN	POSIX
LAMMPS	ADIOS, NetCDF, HDF5, MPI-IO, POSIX
ENZO	HDF5
NWChem	POSIX
ParaDiS	HDF5, POSIX
Chombo	HDF5
GTC	POSIX
GAMESS	POSIX
MILC_QCD	POSIX
MACSio	Silo
pF3D-IO	POSIX
HACC-IO	MPI-IO, POSIX
VPIC-IO	HDF5

Experiments

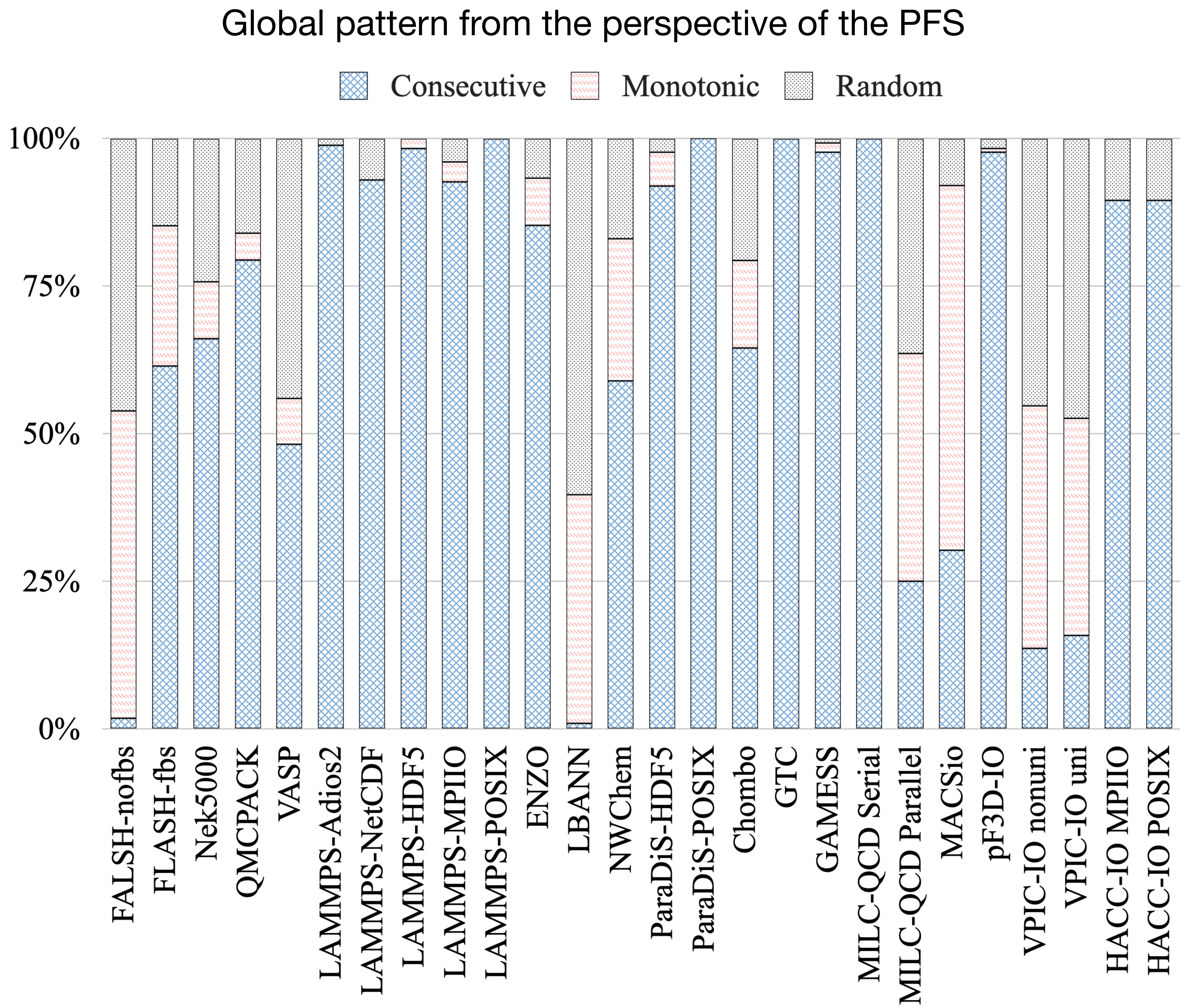
Access patterns

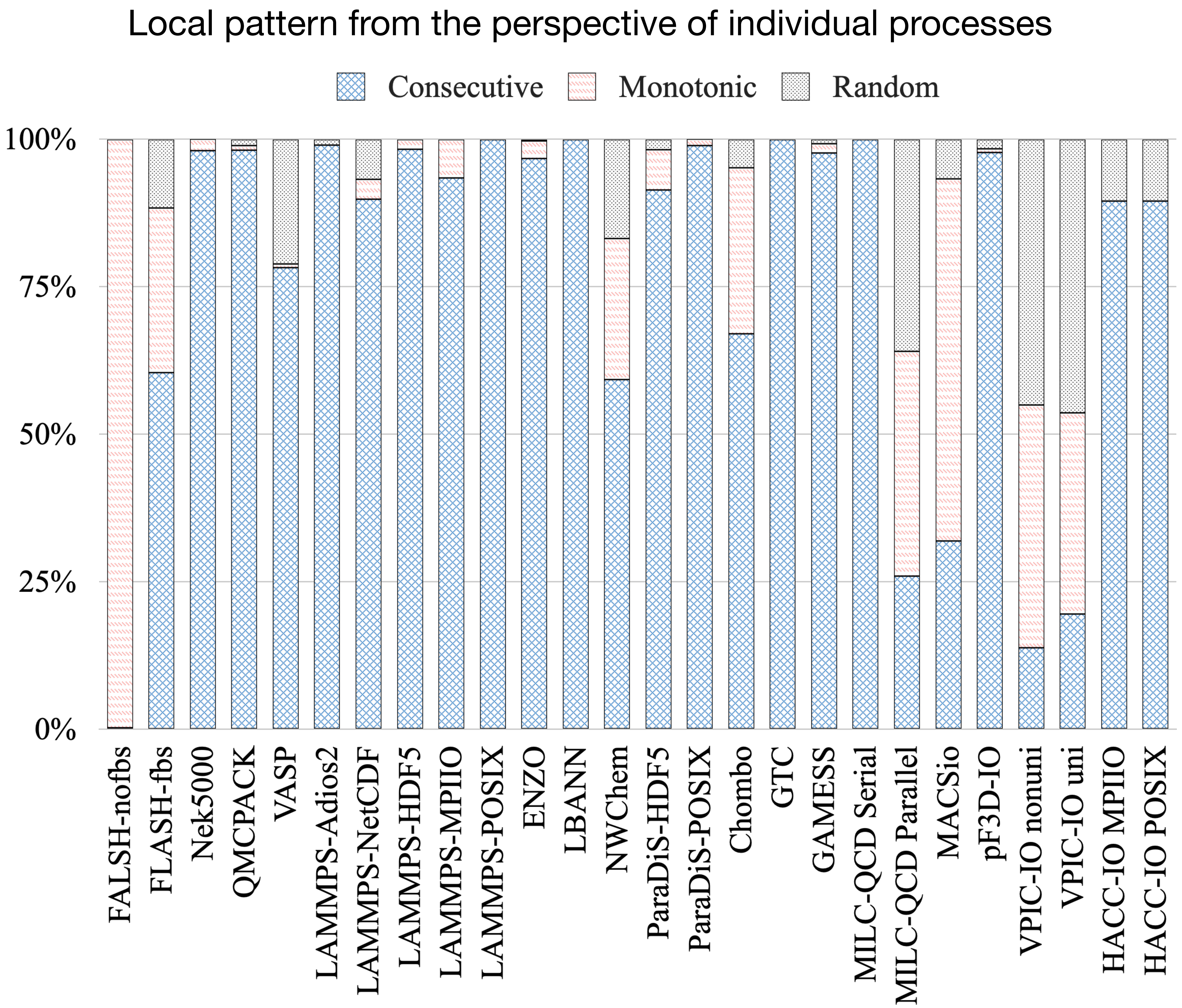
- Per-byte granularity.
- Three types:
  - Consecutive
  - Monotonic
  - Random



Experiments

Access patterns



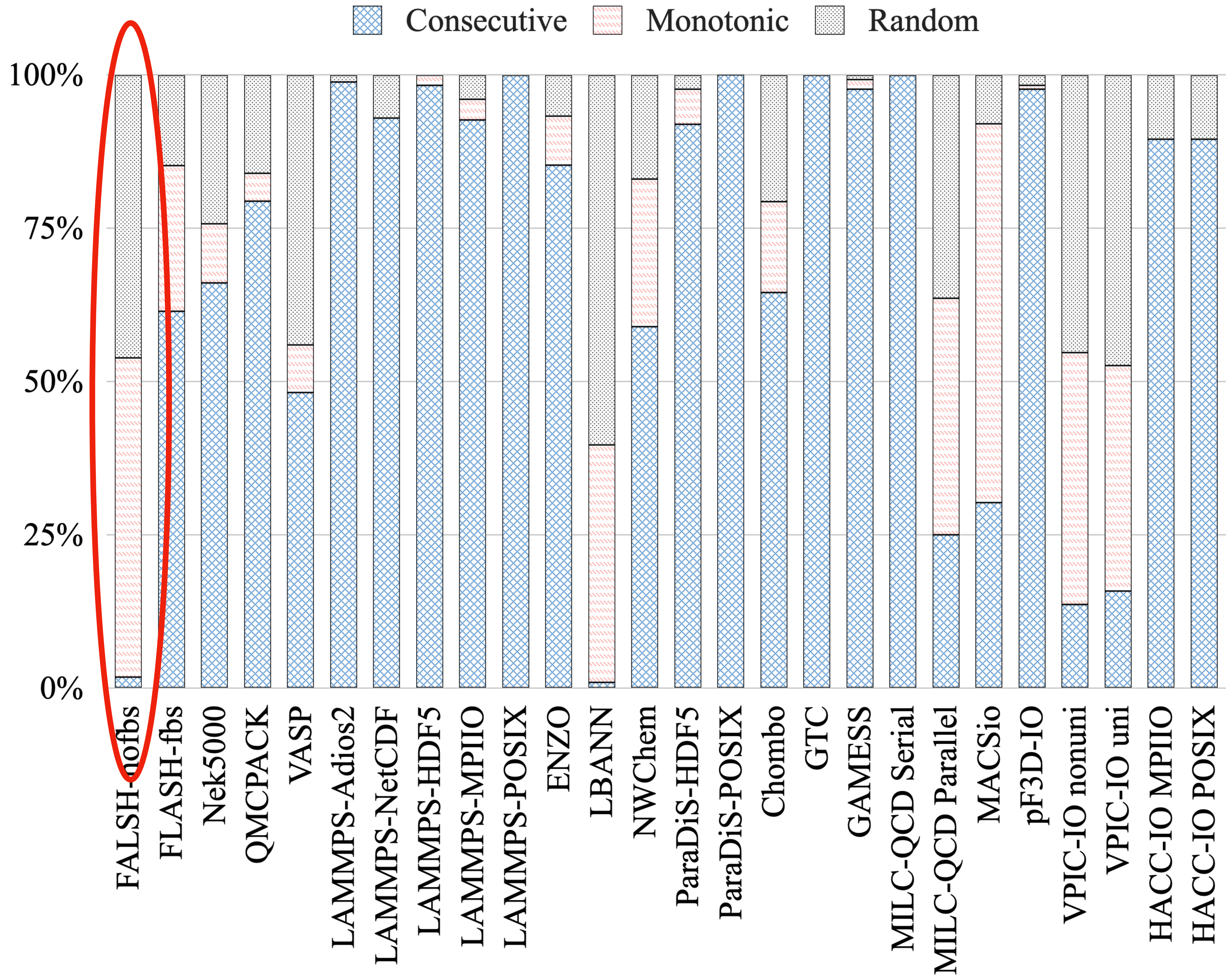


Experiments

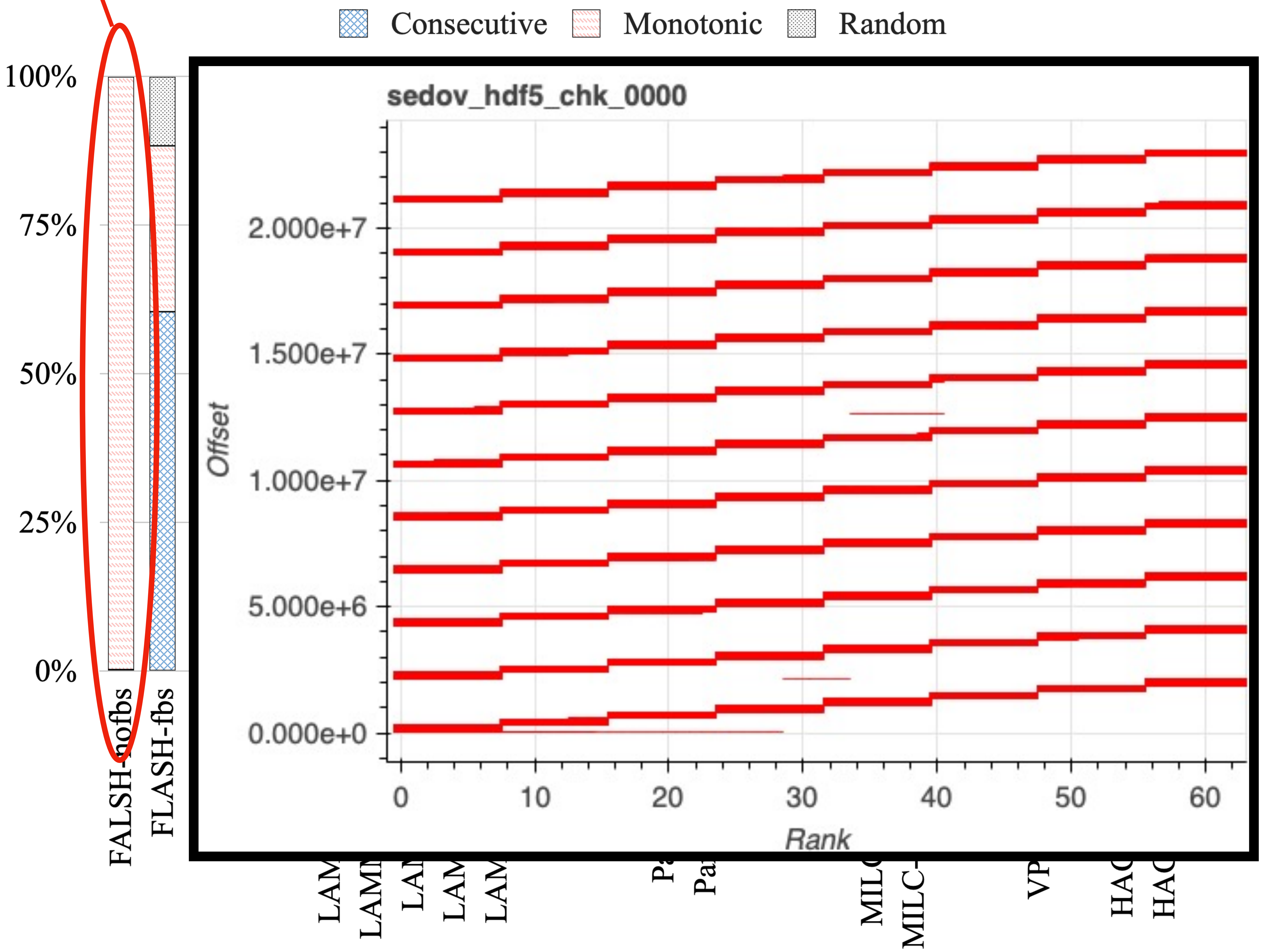
Access patterns

Uses **independent** MPI-IO (HDF5)

Global pattern from the perspective of the PFS



Local pattern from the perspective of individual processes



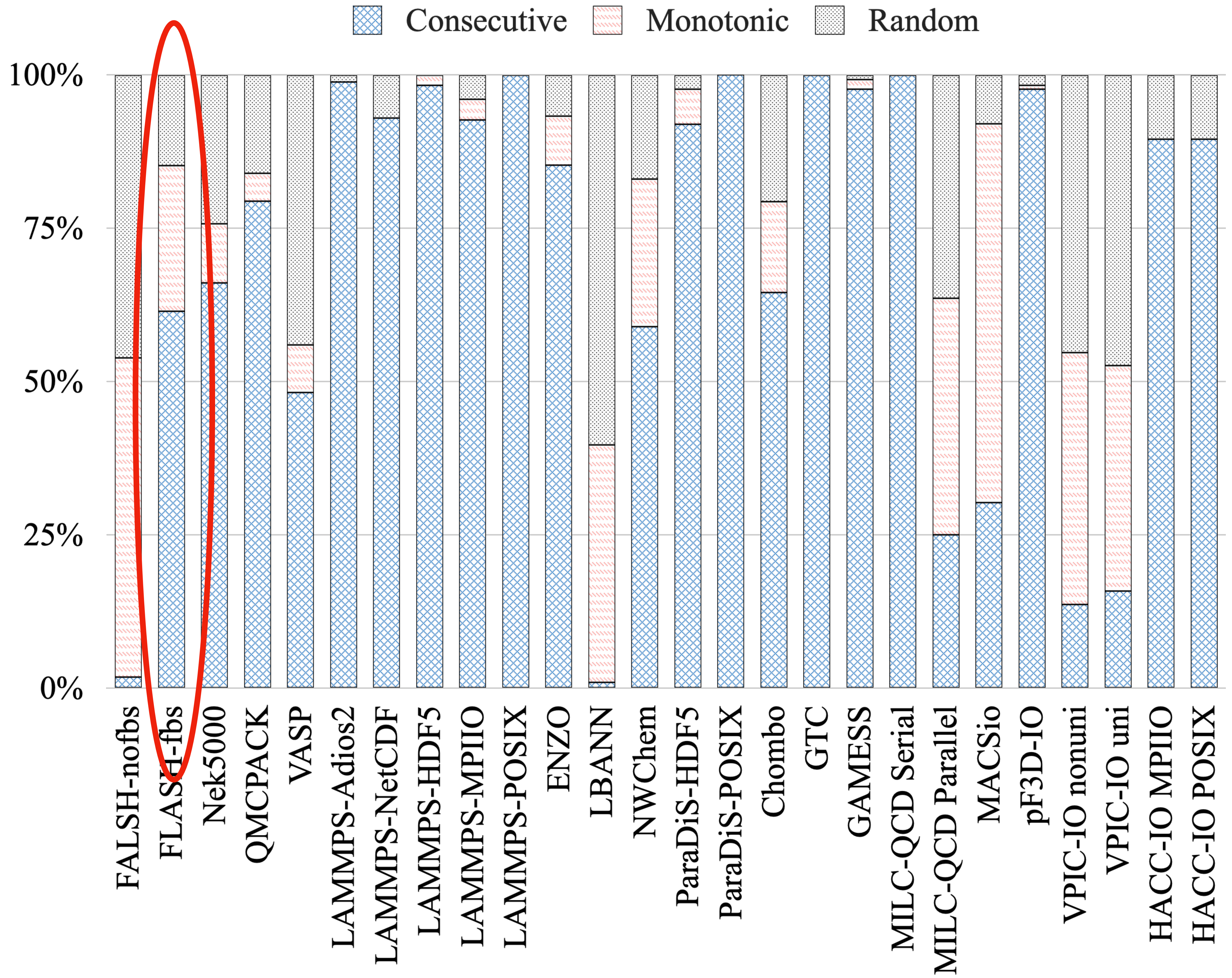
Experiments

Access patterns

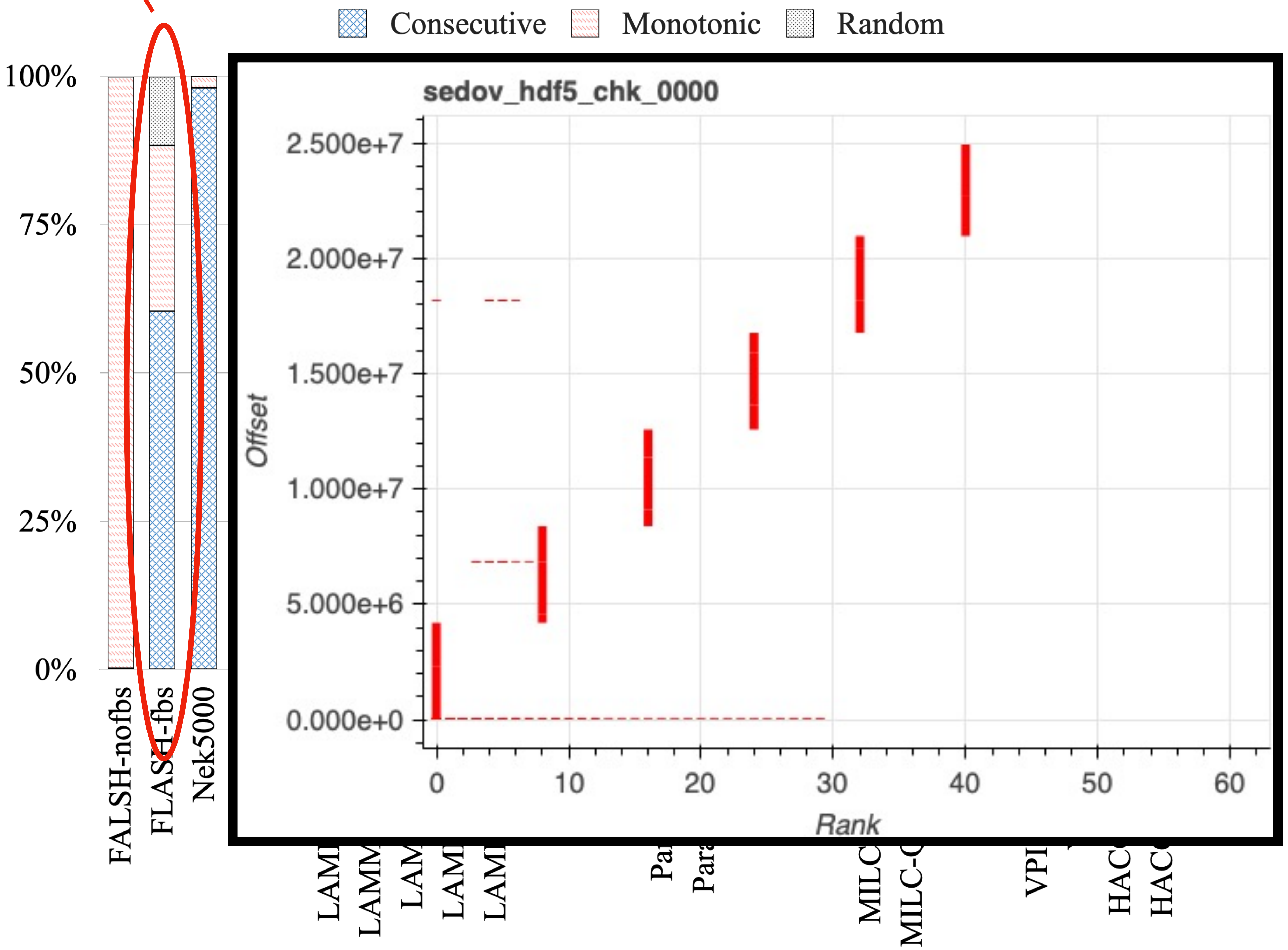
MPI hint: cb_nodes;
Lustre stripe count.

Uses **collective** MPI-IO (HDF5)

Global pattern from the perspective of the PFS



Local pattern from the perspective of individual processes



Experiments

Conflicts observed when using session semantics

- Most conflicts occur on the same process, which are normally handled correctly by the PFS (except BurstFS).
- Only one application (FLASH) has conflicts across processes. Others do not require POSIX semantics, they can run correctly on session semantics.
- The WAW-D conflict of FLASH can be fixed easily (one line change).

Application	I/O library	WAW		RAW	
		S	D	S	D
FLASH	HDF5	✓	✓		
ENZO	HDF5			✓	
NWChem	POSIX	✓		✓	
pF3D-IO	POSIX			✓	
MACSio	Silo	✓			
GAMESS	POSIX	✓			
LAMMPS	ADIOS	✓			
	NetCDF	✓			
	HDF5				
	MPI-IO				
	POSIX				
MILC-QCD	POSIX				
ParaDiS	HDF5				
	POSIX				
VASP	POSIX				
LBANN	POSIX				
QMCPAC	HDF5				
Nek500	POSIX				
GTC	POSIX				
Chombo	HDF5				
HACC-IO	MPI-IO				
	POSIX				
VPIC-IO	HDF5				

Experiments

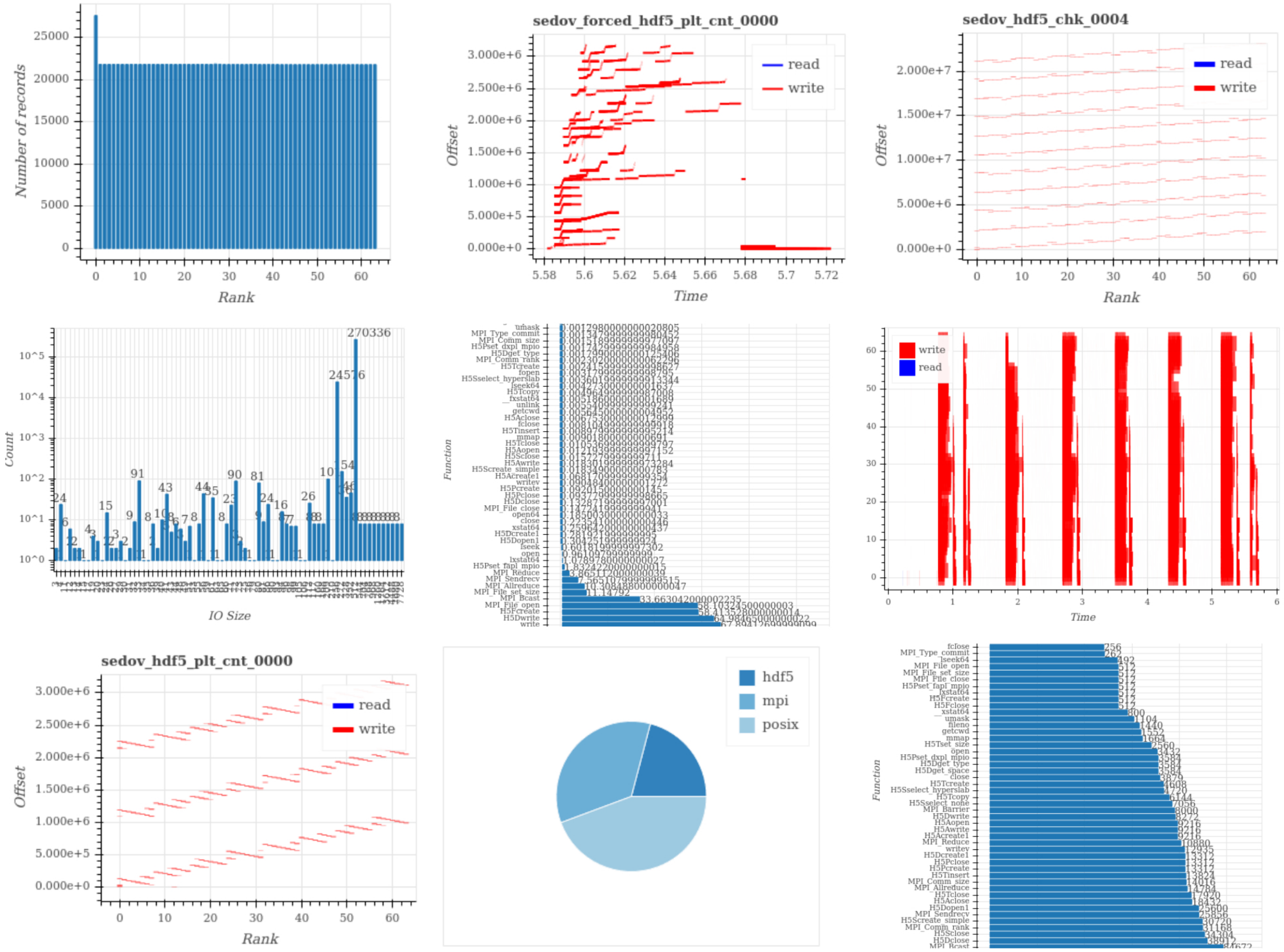
Metadata operations

- Since metadata operations can introduce performance bottleneck, PFS developers may choose to relax POSIX metadata requirements. E.g., atime.
- But what metadata operations are actually used?
- Are they used by the application or I/O libraries?

Experiments

More analysis

- Visualization reports available for all traces.
  - Function counters
  - I/O size histogram
  - Access pattern vs. Rank.
  - Access pattern vs. Time.
- More analysis using our traces & recorder-viz.



Conclusion and Future Work

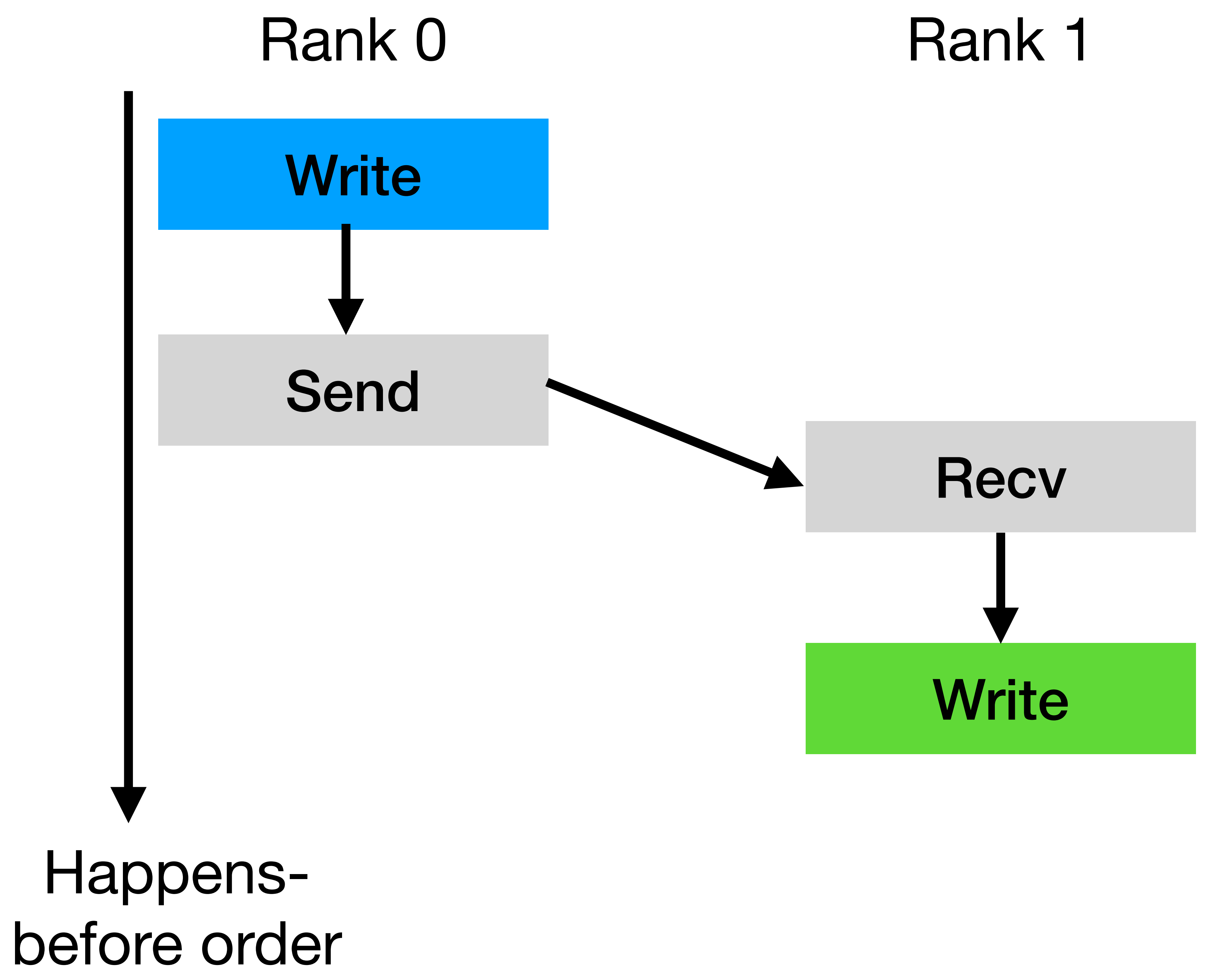
- Takeaways:
  - As commonly expected, most applications do not require POSIX semantics. A weaker model, such as session semantics or commit semantics is enough.
  - Application configurations, I/O libraries, PFS configuration can both change the access pattern.
- We hope our approach can help the community to determine the semantics and operations needed by applications and provided by PFSs.
  - Better documentation of the supported operations, deviations from POSIX semantics, and more uniformity in terminology across PFSs will greatly impact the HPC community.
- We plan to extend the conflicts detection algorithm to include metadata operations and handle complex HPC workflows consisting of multiple applications.

Thanks!

Backup

No conflicts with strong consistency semantics; conflicts are properly synchronized

- Recorder traces include MPI calls also.
- Match MPI calls and generate a graph representing the happens-before order.
- Every potential conflicts are properly synchronized and not concurrent.
- Maximum clock skews in the traces are less than 20 microseconds. Timestamp order of conflicts match their execution order.



Backup

Why isn't WAR a potential conflict

- Assume conflicts are properly synchronized, otherwise, the behavior is undefined even with POSIX semantics.
- Read will finish before write starts.