PARALLEL I/O AND PORTABLE DATA FORMATS

PERFORMANCE ANALYSES

06.11.2024 I ARAVIND SANKARAN (A.SANKARAN@FZ-JUELICH.DE)

Agenda

- Understanding the touch points for I/O performance analysis.

- Learning to use the following tools for monitoring I/O accesses:
  - STrace
  - Darshan
  - LLview

- Apply the tools to analyze the following I/O access patterns:
  - Independent I/O to independent files.
  - Independent I/O to a shared file.
  - Collective I/O to a shared file.

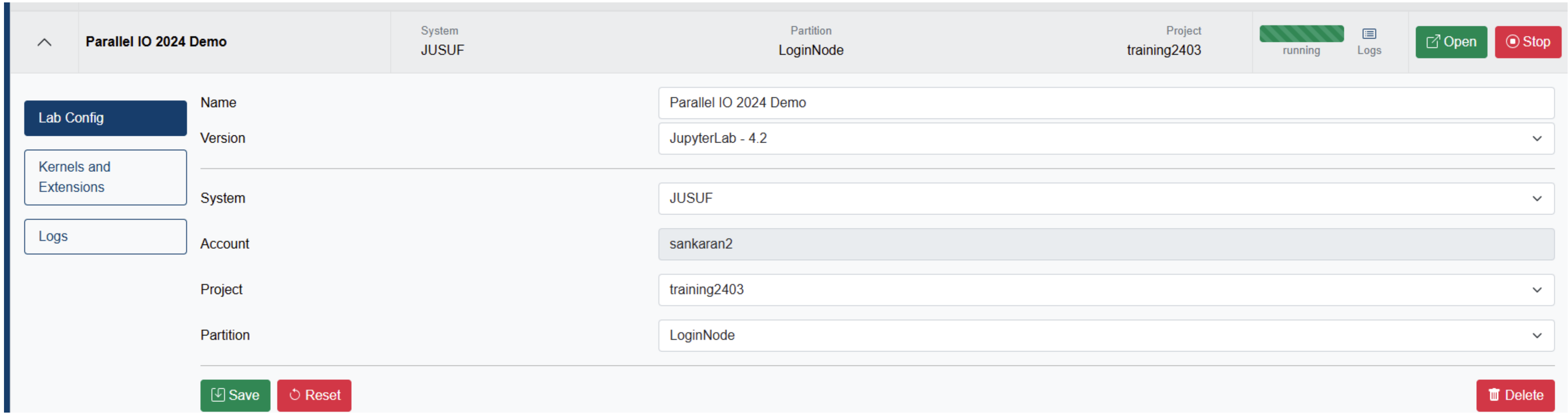
Time for Action

Make a copy of the exercise folder your project directory:

```
/p/project1/training2403/ParIO_course_material/exercises/Perf_Analysis
```

Time for Action: Spawn a Remote Instance of Jupyter Lab

- **OPTION 1: Use Jupyter-JSC** (https://jupyter.jsc.fz-juelich.de/hub/home)



Parallel IO 2024 Demo System JUSUF Partition LoginNode Project training2403 running Logs Open Stop

Name	Parallel IO 2024 Demo
Version	JupyterLab - 4.2
System	JUSUF
Account	sankaran2
Project	training2403
Partition	LoginNode

Time for Action: Spawn a Remote Instance of Jupyter Lab

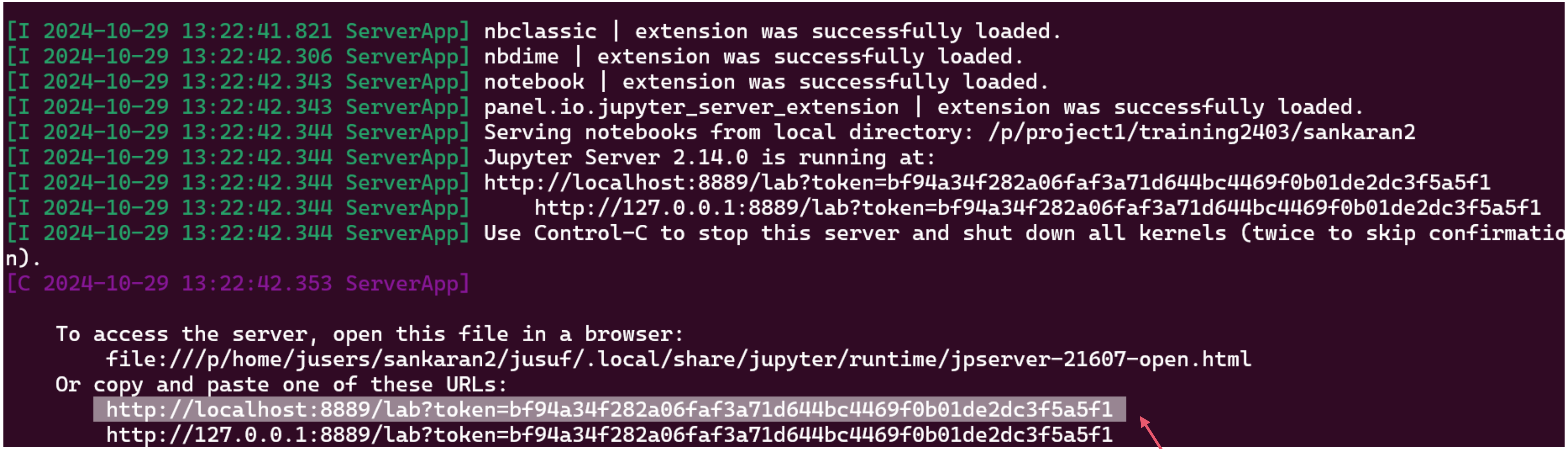
- **OPTION 2: Spawn the instance manually**

```
[local] ssh -L 8889:localhost:8889 -i [PATH_TO_KEY] [USERNAME]@jusuf.fz-juelich.de

[jusuf] cd /p/project1/training2403/[USERNAME]

[jusuf] module load Jupyter-bundle

[jusuf] jupyter lab --no-browser --port=8889
```

An arbitrary port number

Time for Action: Spawn a Remote Instance of Jupyter Lab

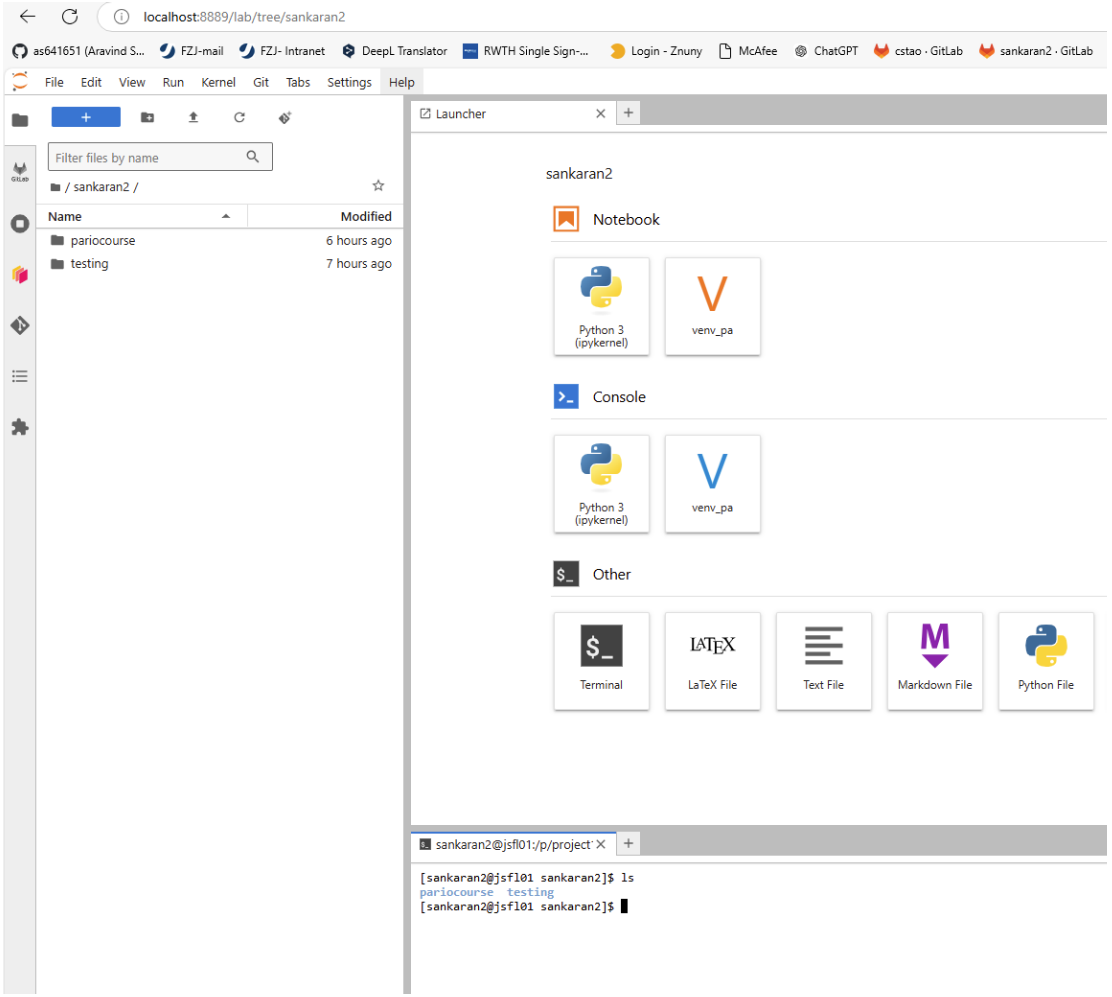
- **OPTION 2: Spawn the instance manually**



```
[I 2024-10-29 13:22:41.821 ServerApp] nbclassic | extension was successfully loaded.
[I 2024-10-29 13:22:42.306 ServerApp] nbdime | extension was successfully loaded.
[I 2024-10-29 13:22:42.343 ServerApp] notebook | extension was successfully loaded.
[I 2024-10-29 13:22:42.343 ServerApp] panel.io.jupyter_server_extension | extension was successfully loaded.
[I 2024-10-29 13:22:42.344 ServerApp] Serving notebooks from local directory: /p/project1/training2403/sankaran2
[I 2024-10-29 13:22:42.344 ServerApp] Jupyter Server 2.14.0 is running at:
[I 2024-10-29 13:22:42.344 ServerApp] http://localhost:8889/lab?token=bf94a34f282a06faf3a71d644bc4469f0b01de2dc3f5a5f1
[I 2024-10-29 13:22:42.344 ServerApp]     http://127.0.0.1:8889/lab?token=bf94a34f282a06faf3a71d644bc4469f0b01de2dc3f5a5f1
[I 2024-10-29 13:22:42.344 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmatio
n).
[C 2024-10-29 13:22:42.353 ServerApp]

    To access the server, open this file in a browser:
        file:///p/home/jusers/sankaran2/jusuf/.local/share/jupyter/runtime/jpserver-21607-open.html
    Or copy and paste one of these URLs:
        http://localhost:8889/lab?token=bf94a34f282a06faf3a71d644bc4469f0b01de2dc3f5a5f1
        http://127.0.0.1:8889/lab?token=bf94a34f282a06faf3a71d644bc4469f0b01de2dc3f5a5f1
```

Copy this link and paste it in the browser of your local machine

Time for Action: Spawn a Remote Instance of Jupyter Lab



Name	Modified
pariocourse	6 hours ago
testing	7 hours ago

```
[sankaran2@jsfl01 sankaran2]$ ls
pariocourse  testing
[sankaran2@jsfl01 sankaran2]$ 
```

Quiz

In C, what are the main difference between the following two sets of function calls?

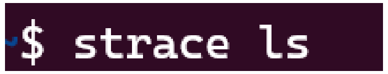
- `open(..), read(..), write(..)`
- `fopen(…), fread(…), fwrite(…)`

Tracing System Calls with STrace

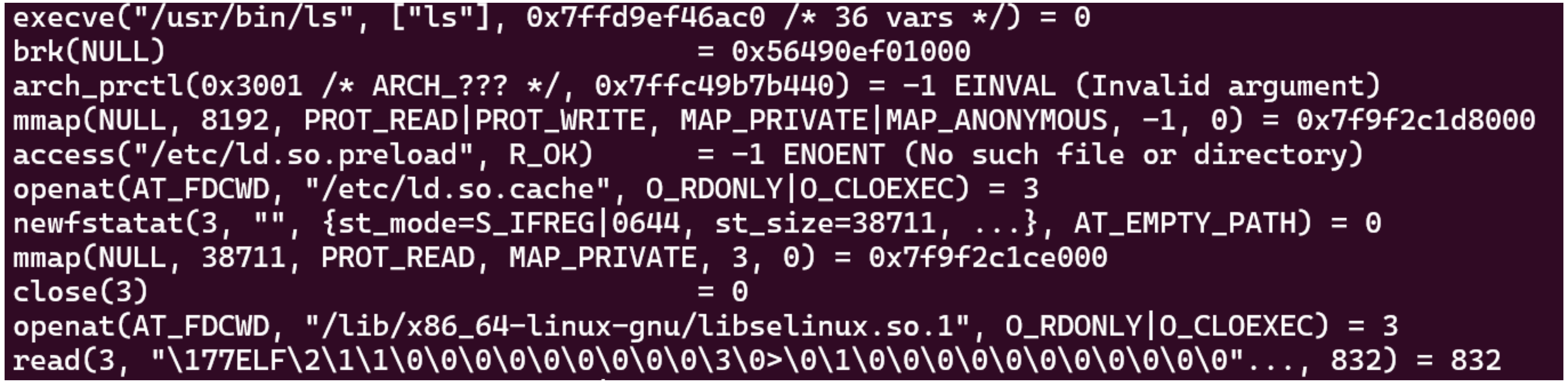
Basic Usage:

```
strace [COMMAND]
```

Example:



```
$ strace ls
```



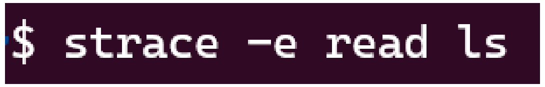
```
execve("/usr/bin/ls", ["ls"], 0x7ffd9ef46ac0 /* 36 vars */) = 0
brk(NULL)                               = 0x56490ef01000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc49b7b440) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9f2c1d8000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=38711, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 38711, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f9f2c1ce000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
```

Tracing System Calls with STrace

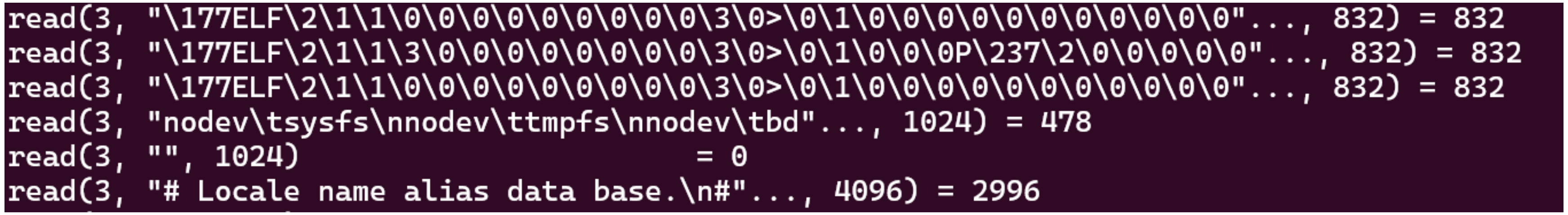
Basic Usage (Show only read calls):

```
strace -e read [COMMAND]
```

Example:

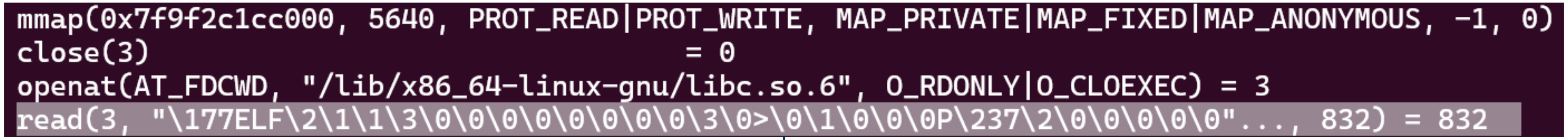


```
$ strace -e read ls
```

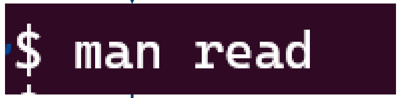


```
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832) = 832
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
read(3, "nodev\tsysfs\nnodev\ttmpfs\nnodev\tbd"..., 1024) = 478
read(3, "", 1024)                       = 0
read(3, "# Locale name alias data base.\n#"..., 4096) = 2996
```

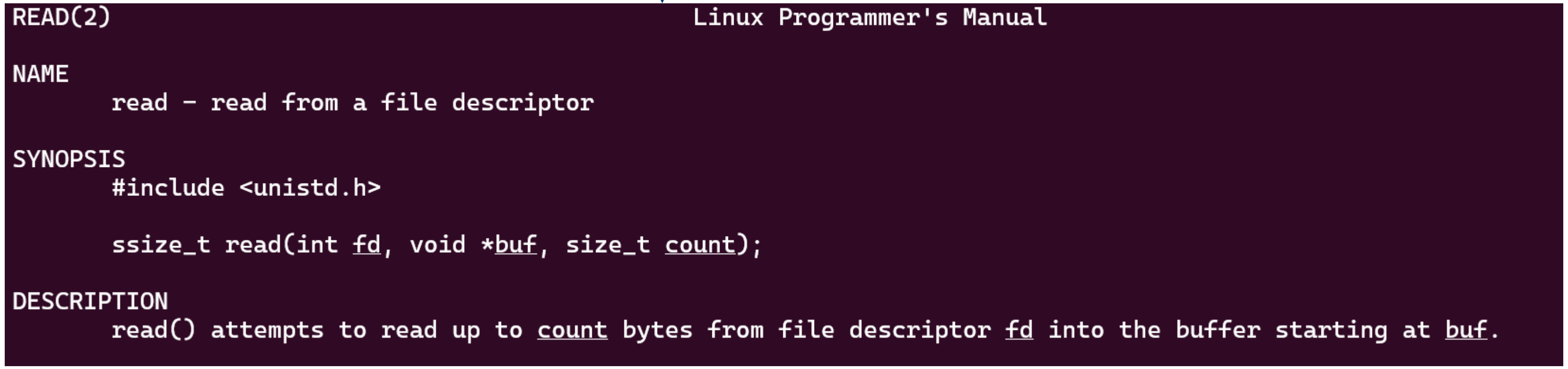
System Call Details



```
mmap(0x7f9f2c1cc000, 5640, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0)
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832) = 832
```



```
$ man read
```



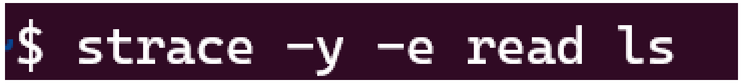
```
READ(2)                                               Linux Programmer's Manual

NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.
```

Tracing System Calls with STrace

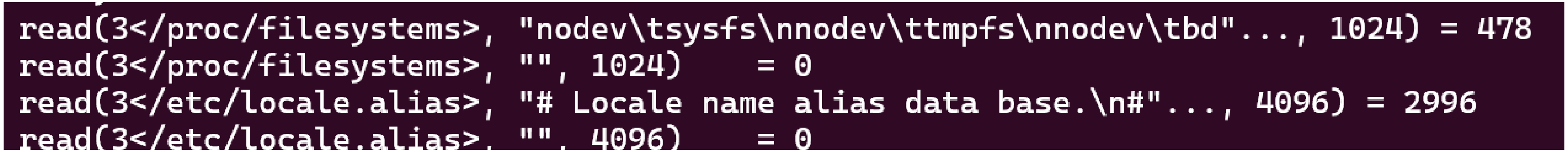
Basic Usage (Show only read calls with file paths instead of file descriptors):

```
strace -y -e read [COMMAND]
```

Example:

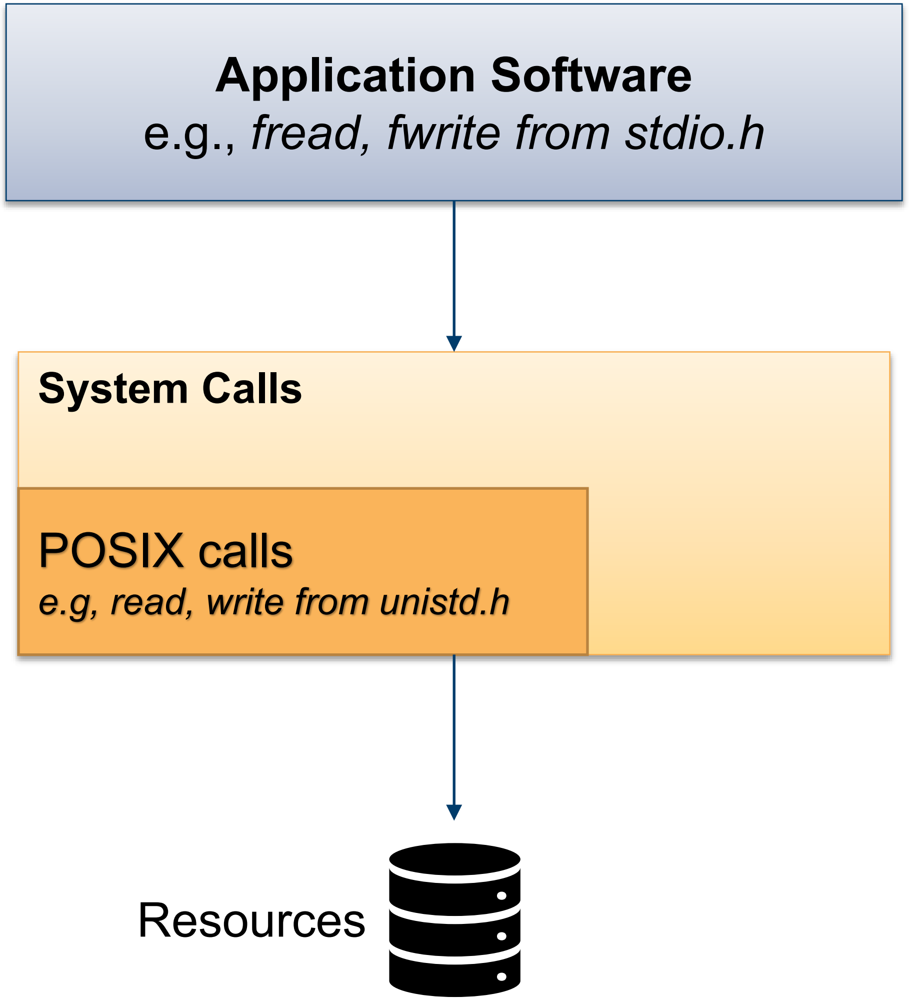


```
$ strace -y -e read ls
```



```
read(3</proc/filesystems>, "nodev\tsysfs\nnodev\ttmpfs\nnodev\tbd"..., 1024) = 478
read(3</proc/filesystems>, "", 1024)    = 0
read(3</etc/locale.alias>, "# Locale name alias data base.\n#"..., 4096) = 2996
read(3</etc/locale.alias>, "", 4096)    = 0
```

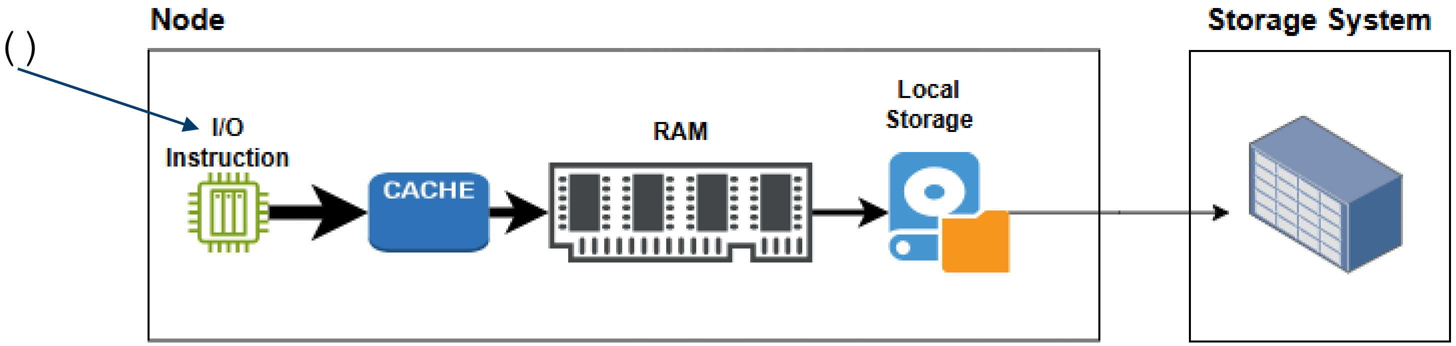
System Calls


Application Software
e.g., *fread, fwrite from stdio.h*
System Calls
POSIX calls
e.g, read, write from unistd.h
Resources

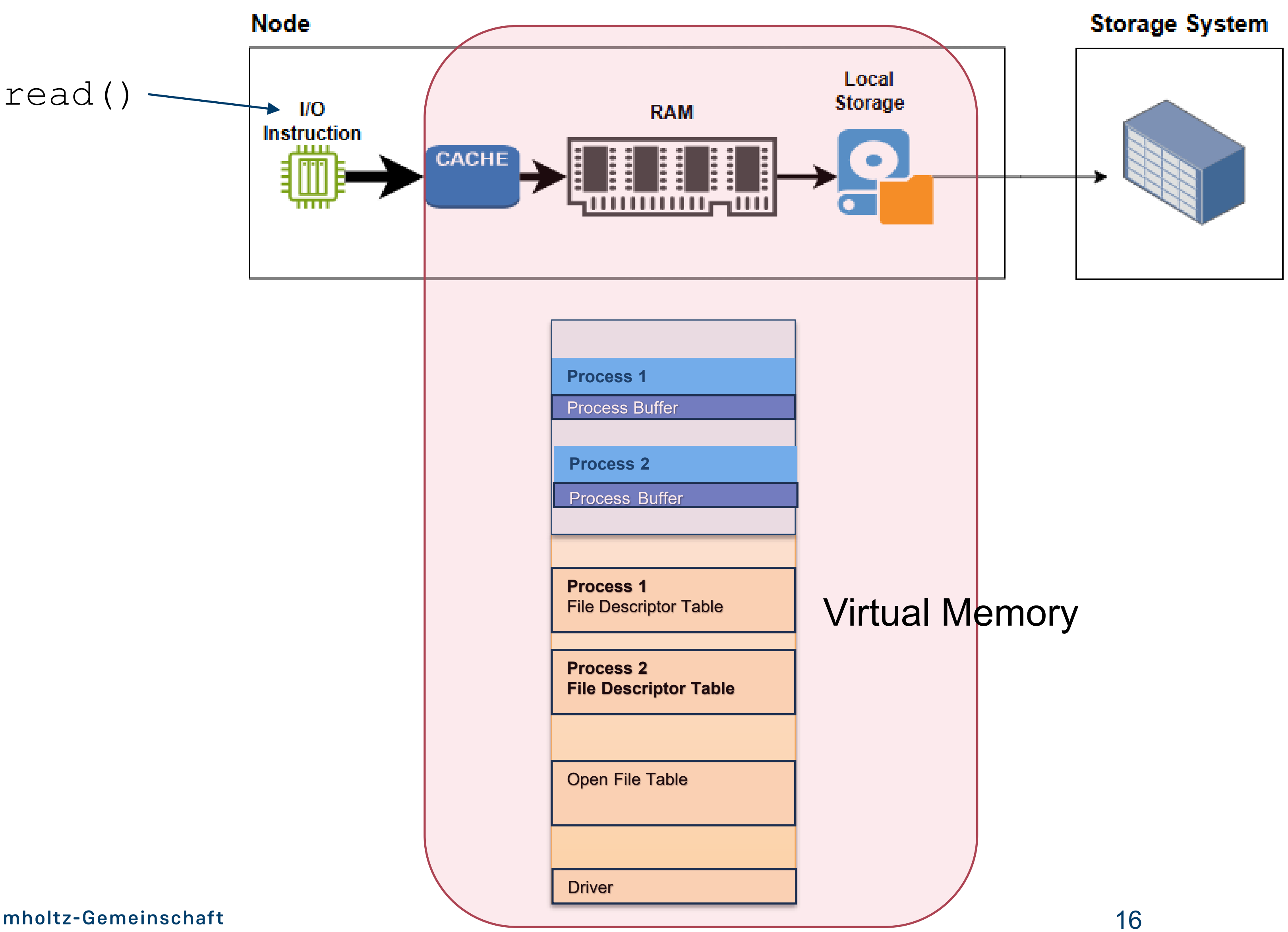
Why should I use fread, fwrite, when I can directly use read, write from unistd.h?

I/O Workflow

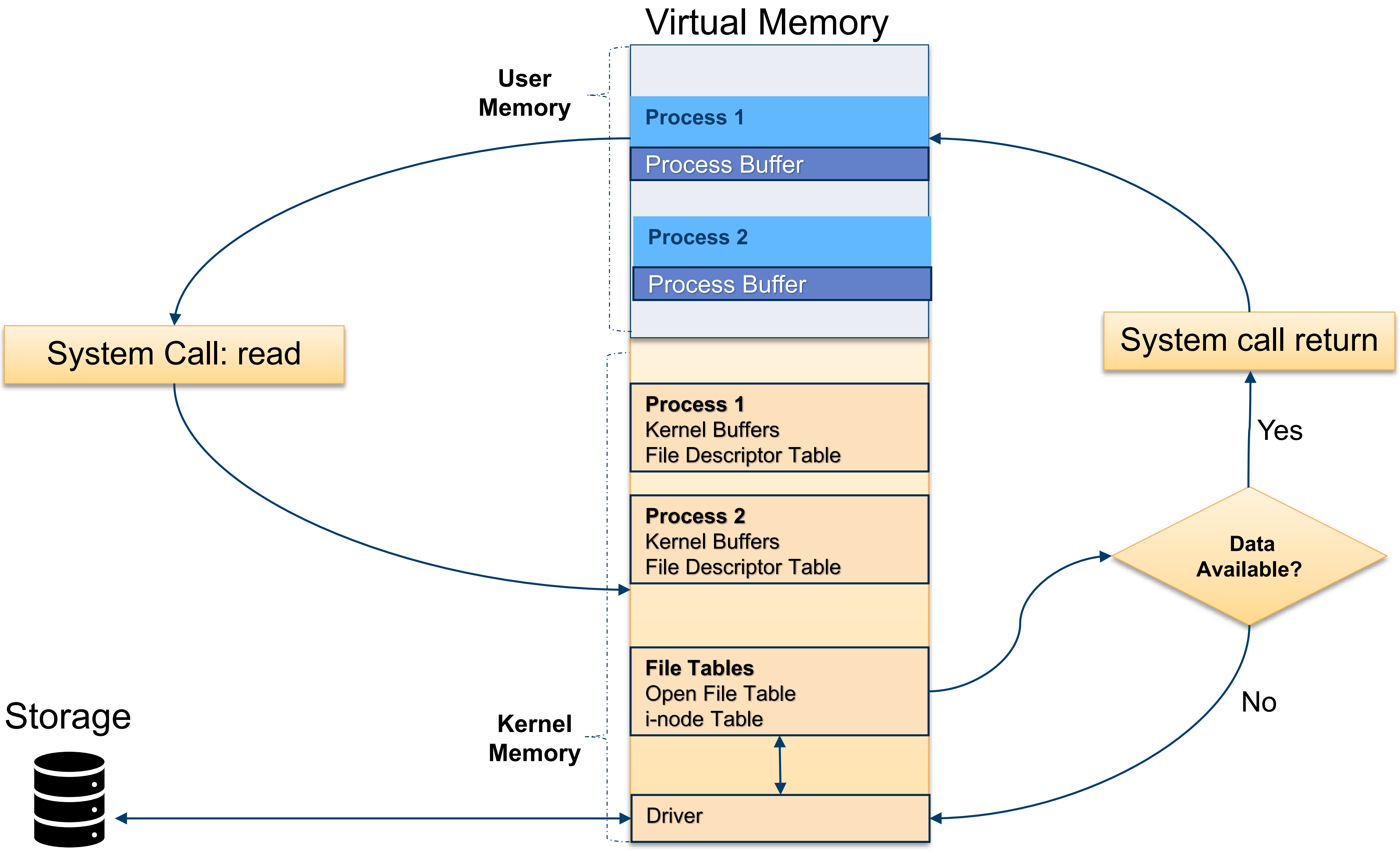
`e.g., read()`



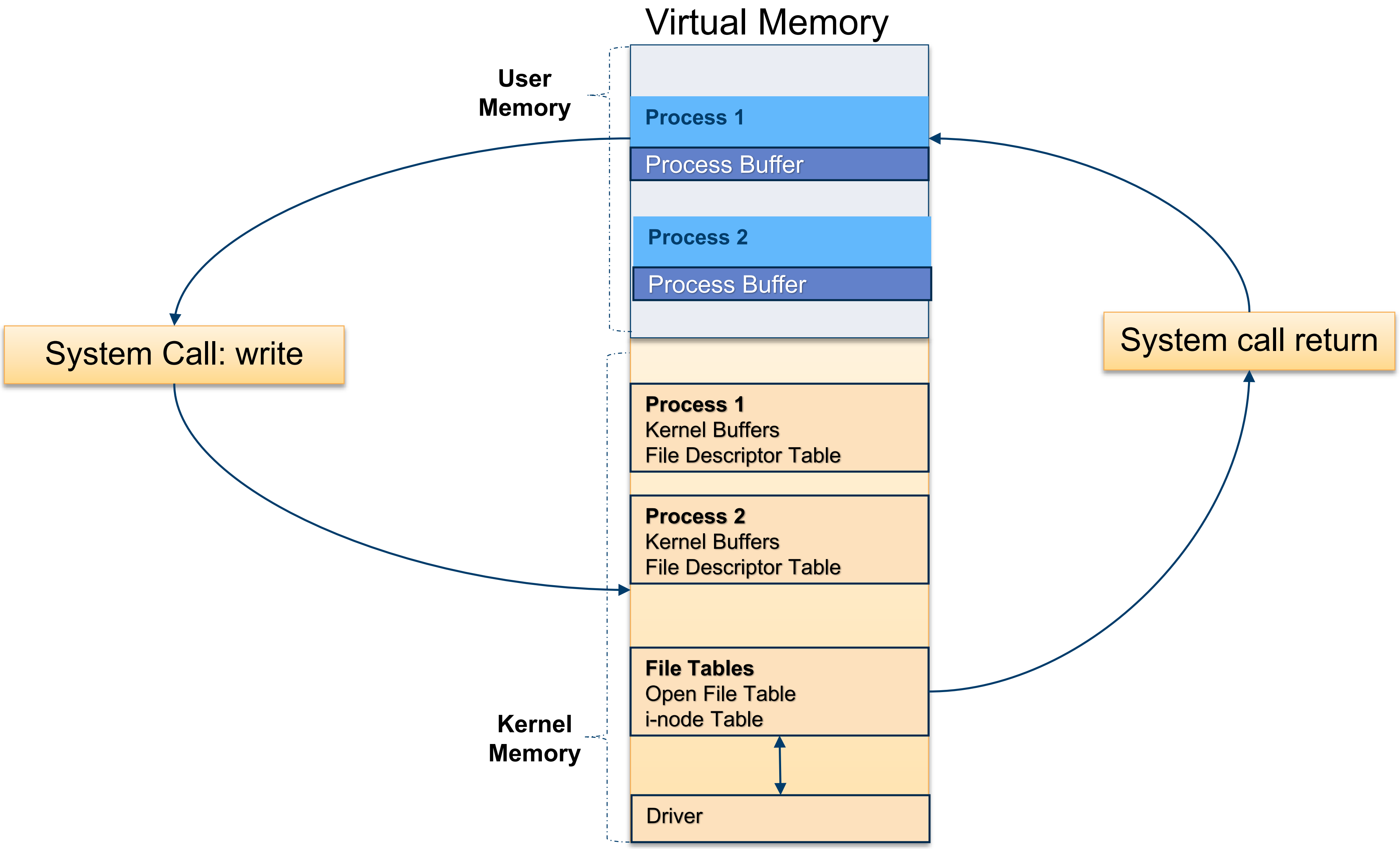
I/O Workflow


Node
Storage System
read()
I/O Instruction
CACHE
RAM
Local Storage
Process 1
Process Buffer
Process 2
Process Buffer
Process 1
File Descriptor Table
Process 2
File Descriptor Table
Open File Table
Driver
Virtual Memory

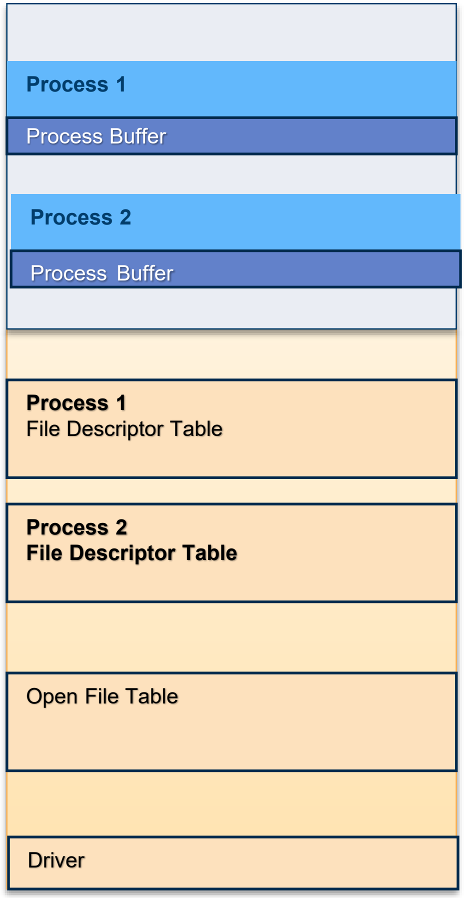
Read Workflow (OS View)



Write Workflow (OS View)



File Descriptor Table



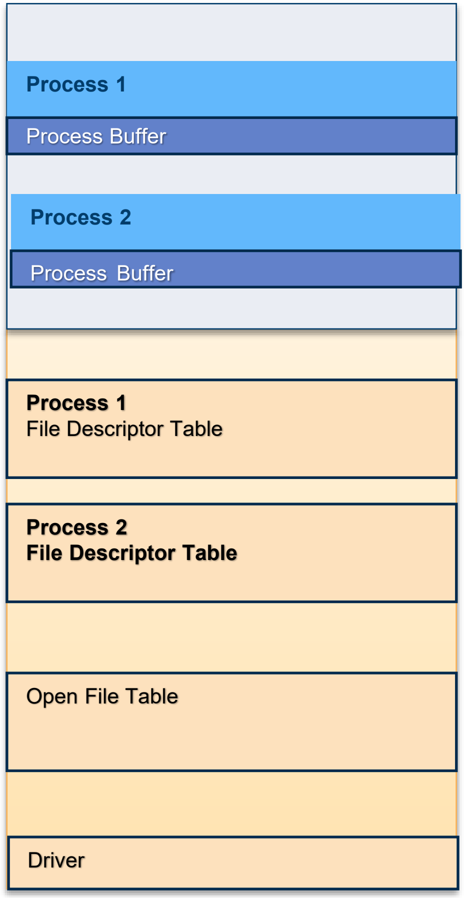
File Descriptor Table:

- Each Process has its own File descriptor table.

Some useful commands:

- `ls -l /proc/<PID>/fd`
  - Lists all the files opened by the process.
- `cat /proc/<PID>/fdinfo/<FD>`
  - Details of a particular file descriptor.

Open File Table



Process 1

Process Buffer

Process 2

Process Buffer

Process 1
File Descriptor Table

Process 2
File Descriptor Table

Open File Table

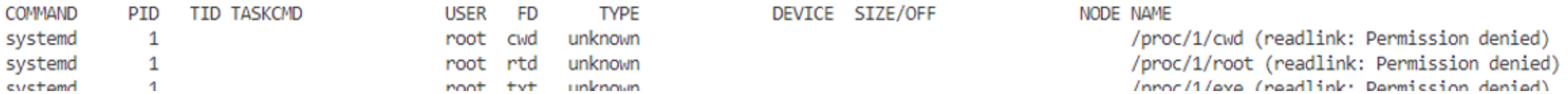
Driver

Open File Table:

- List of all open files in the system.
- Instead of File descriptor, points to the inode.

Some useful commands

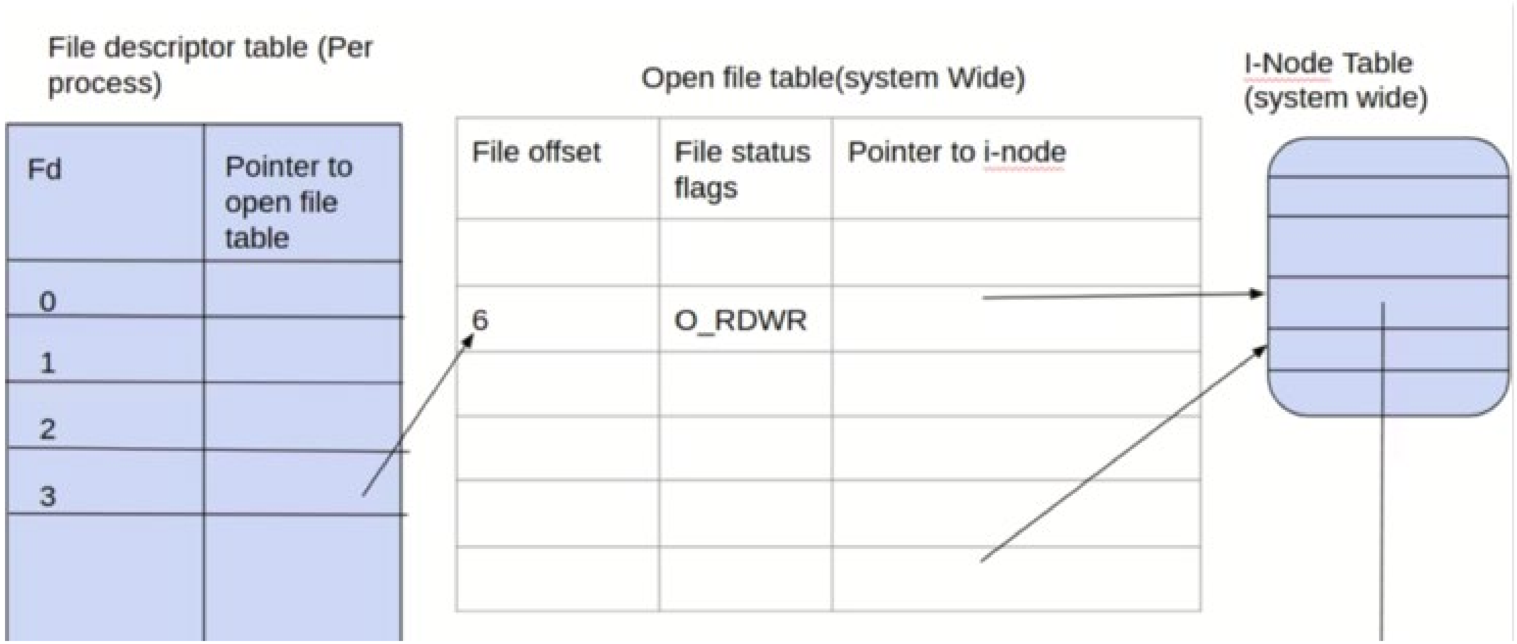
- `lsof`
  - Lists all open files



```
COMMAND     PID   TID TASKCMD               USER   FD      TYPE             DEVICE  SIZE/OFF              NODE NAME
systemd       1                             root  cwd   unknown                                                 /proc/1/cwd (readlink: Permission denied)
systemd       1                             root  rtd   unknown                                                 /proc/1/root (readlink: Permission denied)
systemd       1                             root  txt   unknown                                                 /proc/1/exe (readlink: Permission denied)
```

Relationship between the Tables

File Descriptor table has pointer to the **Open File Table** (system wide), which in turn has pointer to the **i-node table**, which has the data.



File descriptor table (Per process)

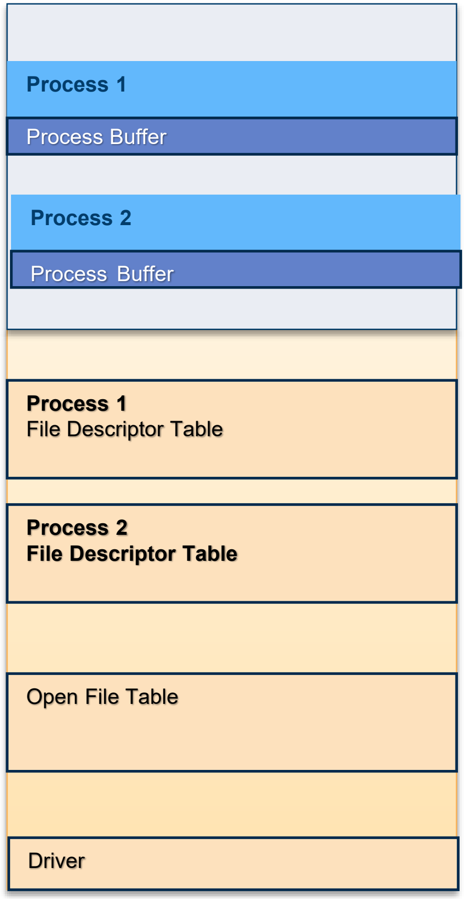
Fd	Pointer to open file table
0	
1	
2	
3	

Open file table(system Wide)

File offset	File status flags	Pointer to i-node
6	O_RDWR	

I-Node Table (system wide)

I/O Workflow



Process 1

Process Buffer

Process 2

Process Buffer

Process 1
File Descriptor Table

Process 2
File Descriptor Table

Open File Table

Driver

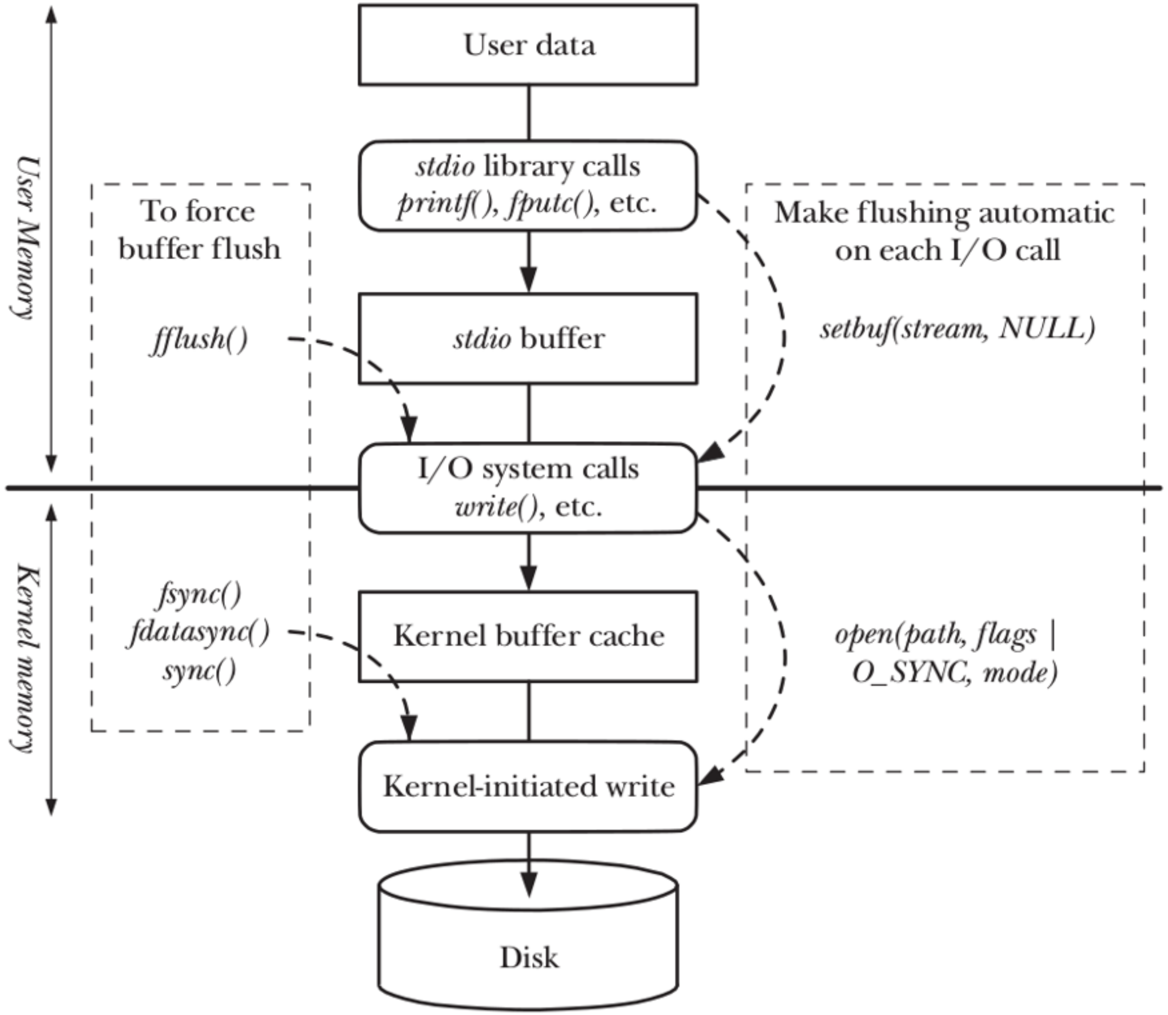
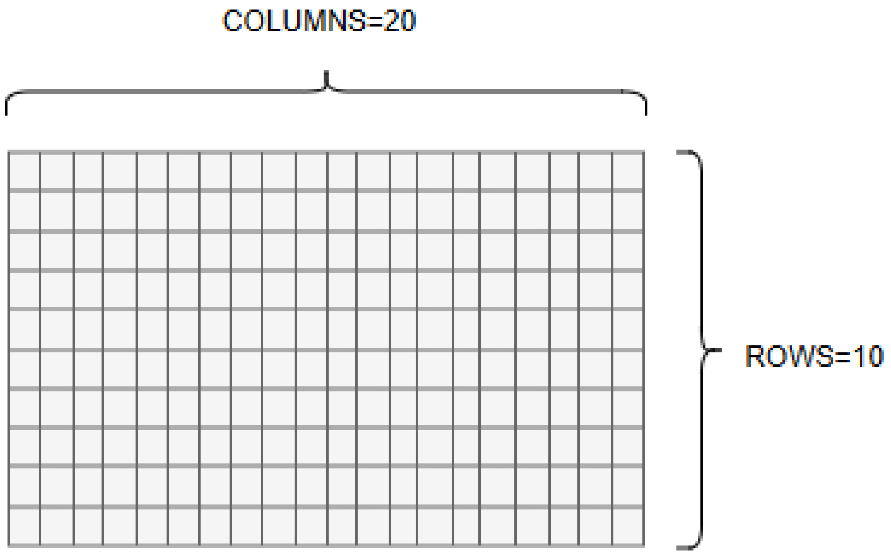


Fig ref: Kerrisk, Michael. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010

Independent I/O to Independent File



Type: `Int` (4 bytes)
Data size: 10x20x4 = 800 bytes

Warm Up:

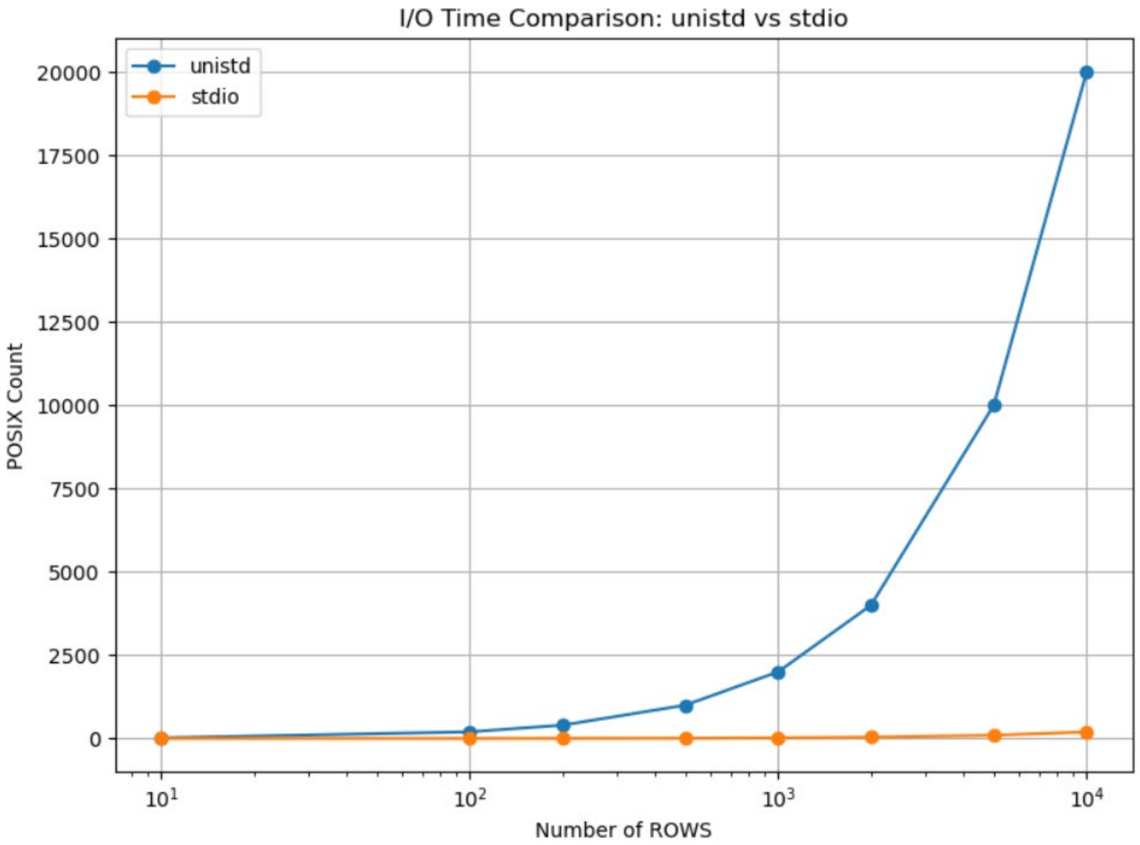
- Consider a two dimensional array with Integer values 1 up to 200.

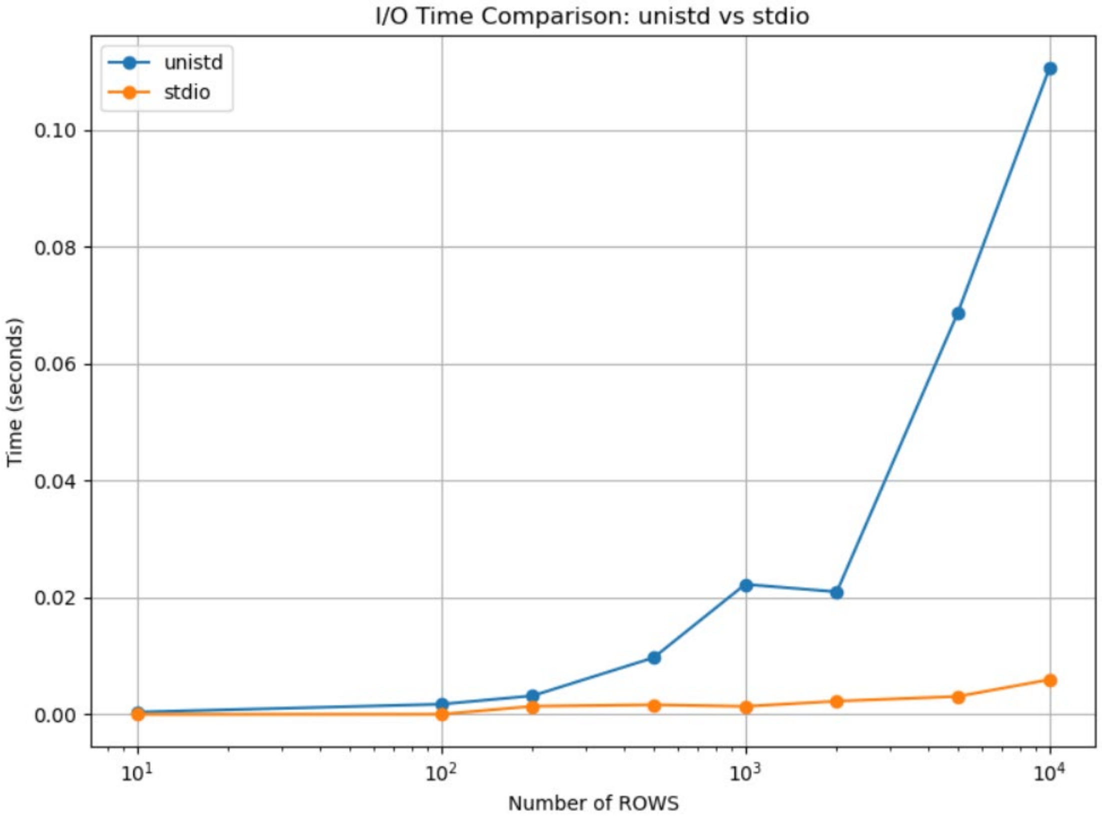
We will analyze the following interfaces:

- UNISTD (e.g., read, write)
- STDIO (e.g., fread, fwrite)

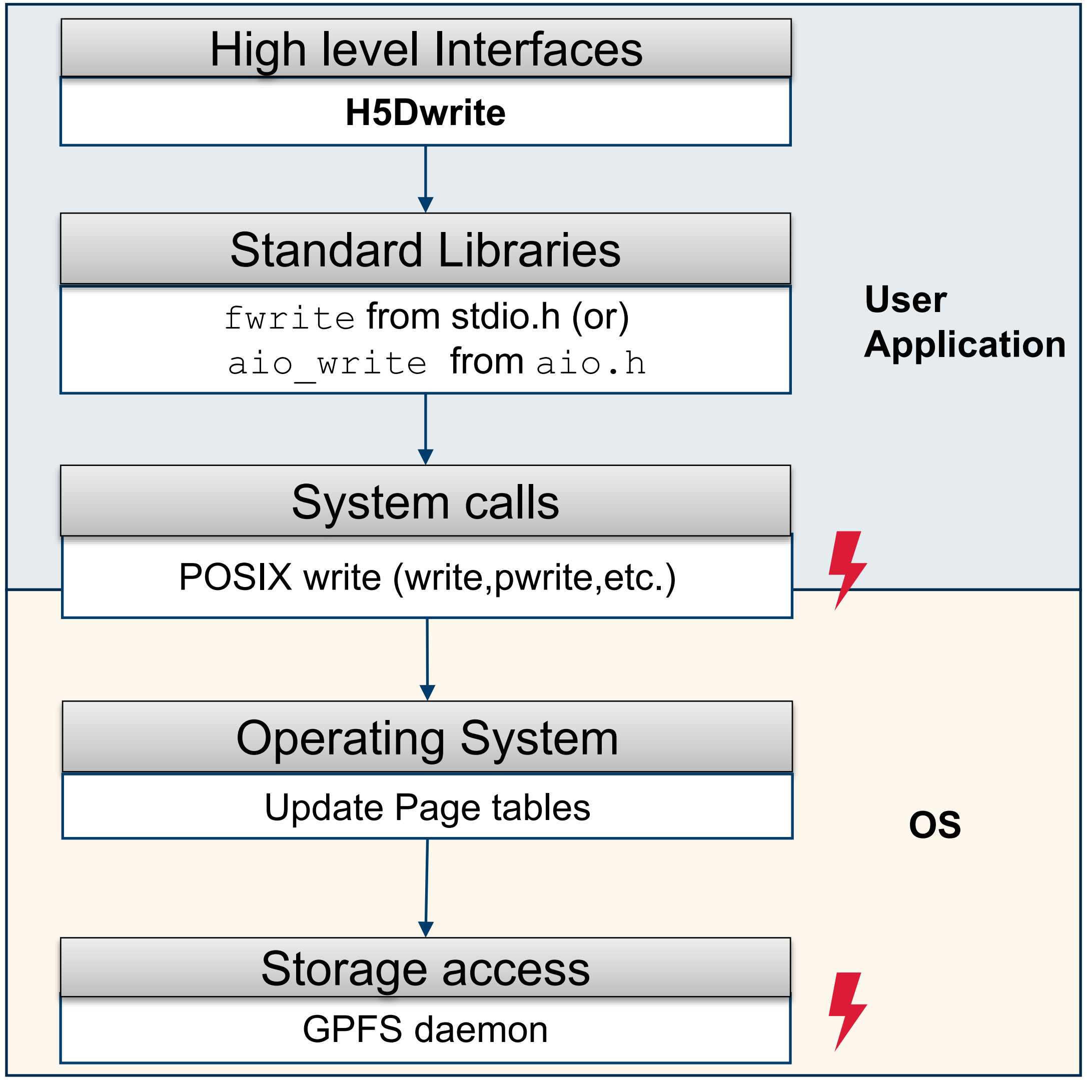
Follow the notebook:
`02_II_posix_stdio.ipynb (Task 1)`

Independent I/O to Independent File





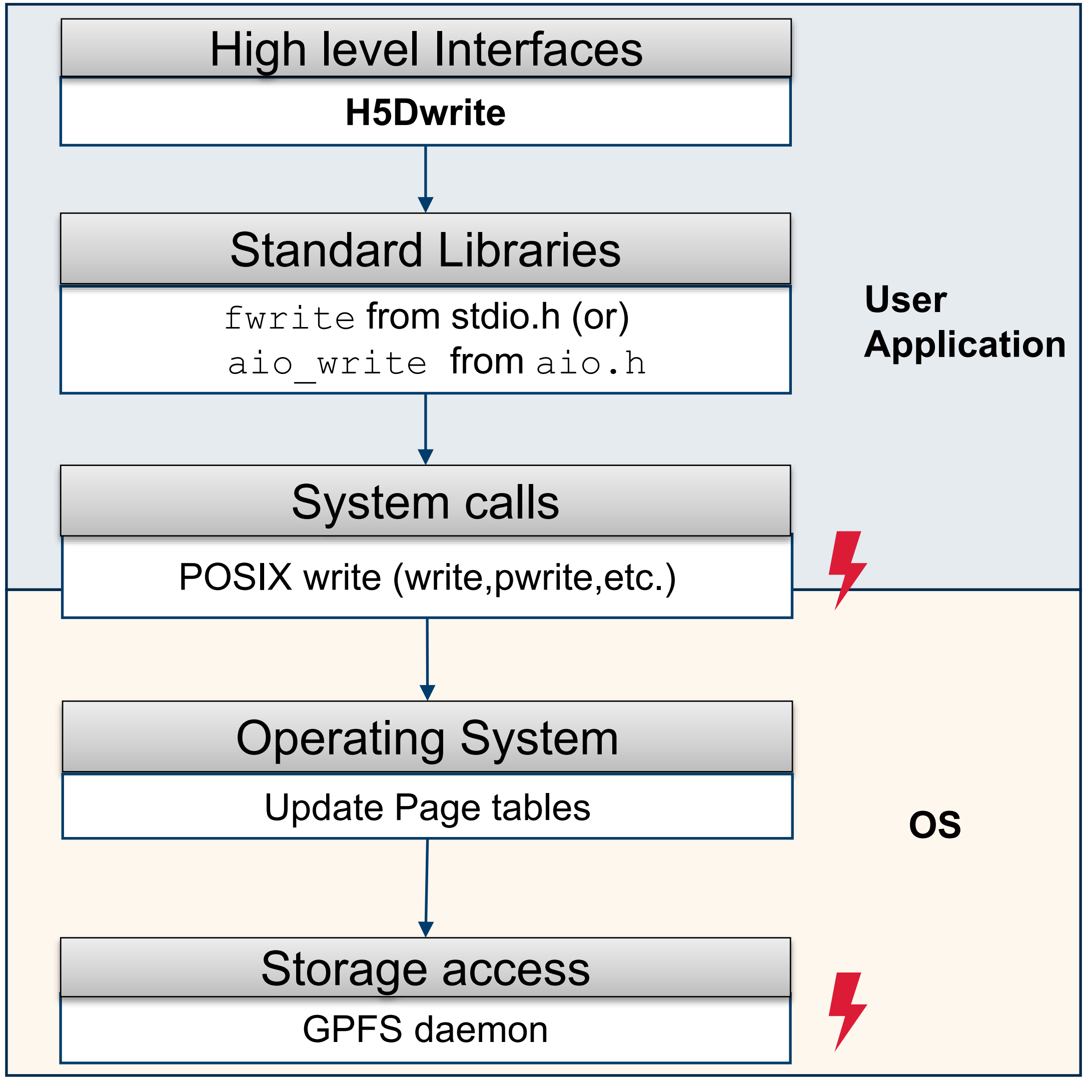
The Touch Points for I/O Performance Analyses



POSIX calls and Storage Access

- To perform I/O, a user application should eventually issue requests to the OS. These requests are called system calls, and they typically conform to POSIX standard.
- The OS performs read/write operations to a copy of the file in the memory (i.e., in the page table) and returns.
- The return of the system calls are significantly high if storage accesses are involved.
  - **read:** The storage access is done if the data is not in the page table.
  - **write:** The storage access is done asynchronously (except when explicitly synchronized).

The Touch Points for I/O Performance Analyses



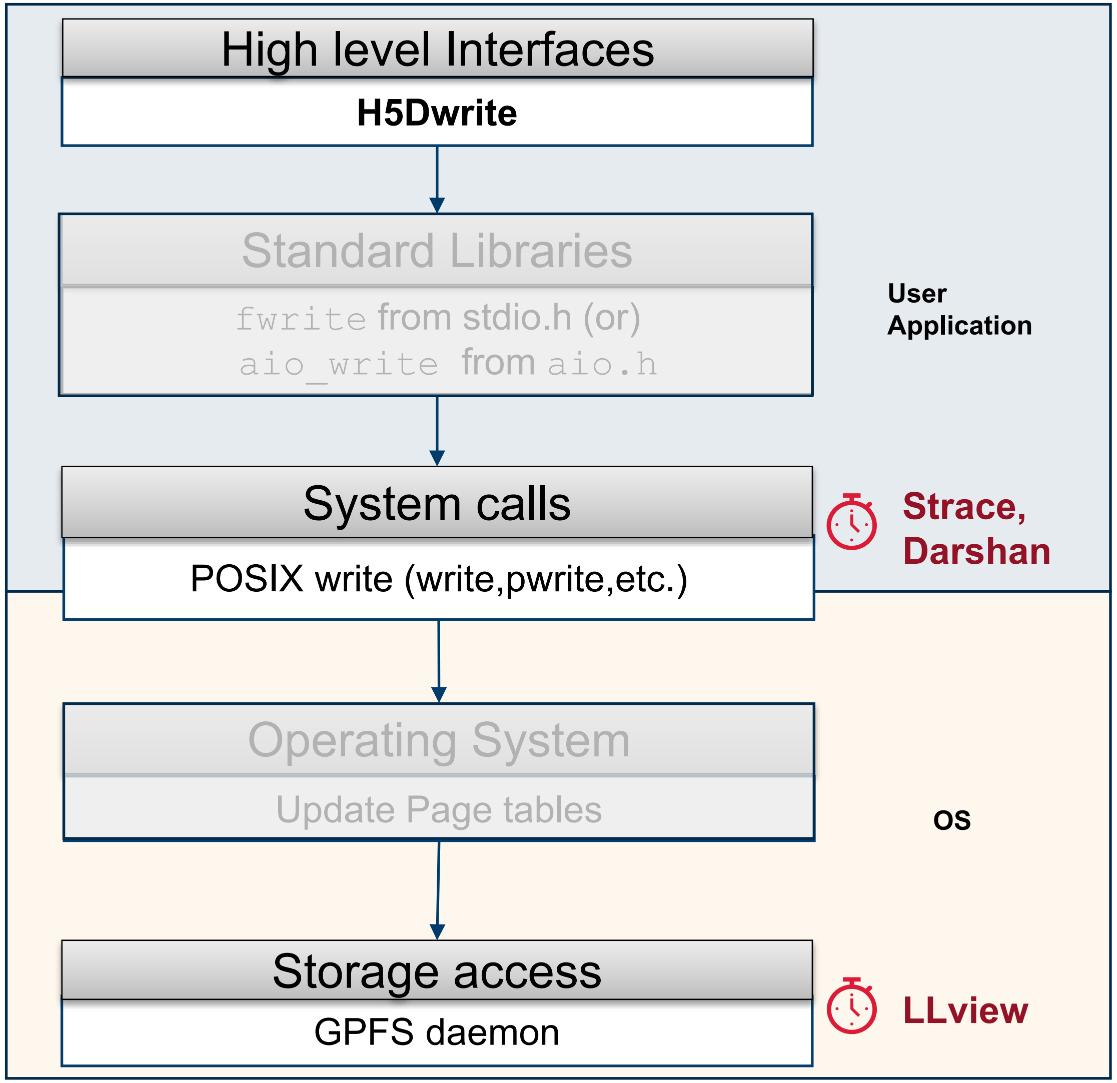
Optimization of I/O

- User applications should minimize the number of POSIX calls with small access size.
- Modern OS internally minimizes the number of storage accesses by optimally repacking one or more POSIX calls.
- Therefore, to optimize I/O, both POSIX calls and storage access counts should be considered.

Monitoring of POSIX calls and storage accesses

- The user application typically cannot monitor beyond the POSIX calls, and does not directly know whether or not the storage access was done.
- For monitoring of storage accesses, site specific tools can be used.

Tools for monitoring POSIX and GPFS accesses



High level Interfaces

H5Dwrite

Standard Libraries

`fwrite` from stdio.h (or) `aio_write` from `aio.h`

System calls

POSIX write (write,pwrite,etc.)

Operating System

Update Page tables

Storage access

GPFS daemon

User Application

Strace, Darshan

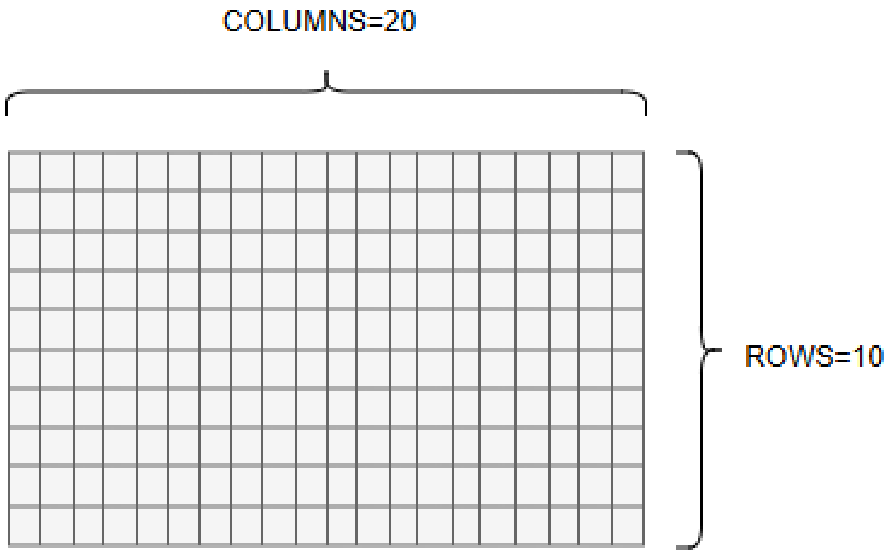
OS

LLview

What is covered in this session:

- **Application level monitoring:**
  - **STrace**: Linux utility to traces the sequence of POSIX calls. Simple to use, and provides the raw data that can be used to infer application performance.
  - **Darshan:** Utilizes the traces of POSIX calls and also the calls from standard library (STDIO) and MPI-IO, computes statistics and provide a high level overview of application performance.
  - There are more tools such as Score-P that are not discussed in this session.
- **System level monitoring:**
  - **LLview:** Provides an overview of GPFS accesses for each SLRUM job in JSC systems.

Independent I/O to Independent File



Type: `Int` (4 bytes)
Data size: 10x20x4 = 800 bytes

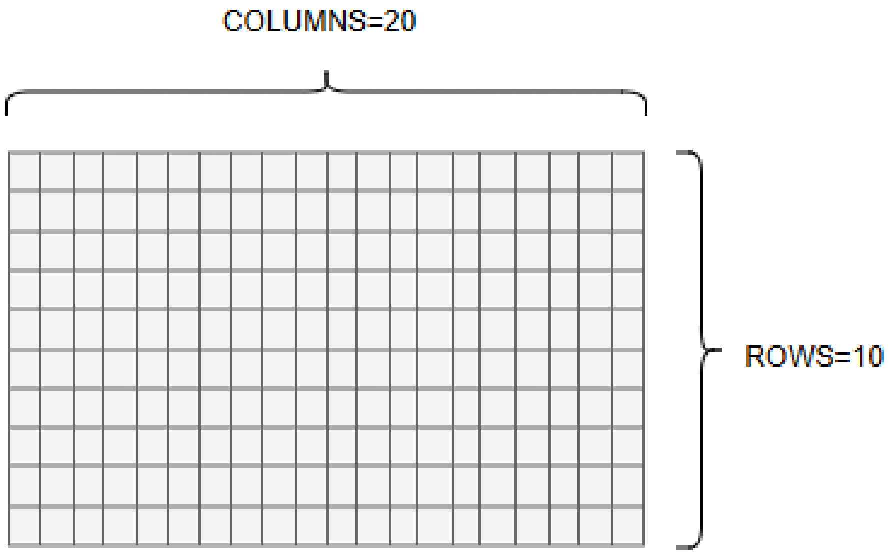
Consider a two dimensional array with Integer values 1 up to 200.

We will analyze the following interfaces:

- Python built-in I/O interfaces
- Python Pickle

Follow the notebook:
`02_II_posix_stdio.ipynb (Task 2)`

Independent I/O to Independent File



Type: `Int` (4 bytes)
Data size: 10x20x4 = 800 bytes

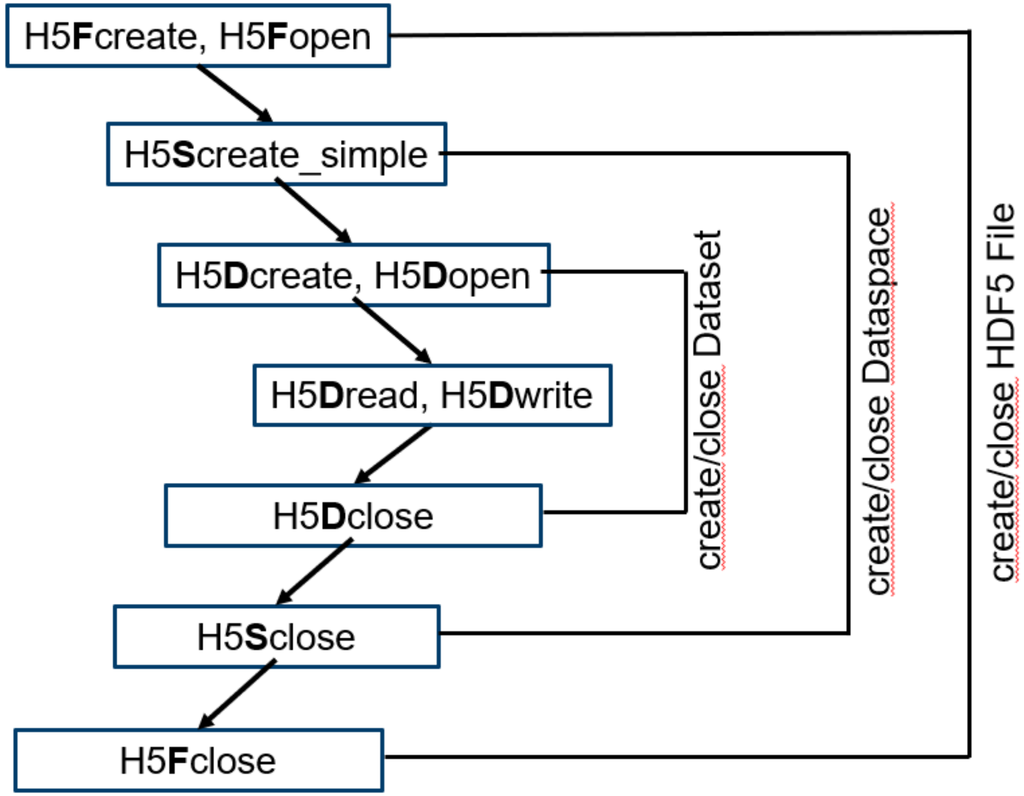
Consider a two dimensional array with Integer values 1 up to 200.

We will analyze the following interfaces:

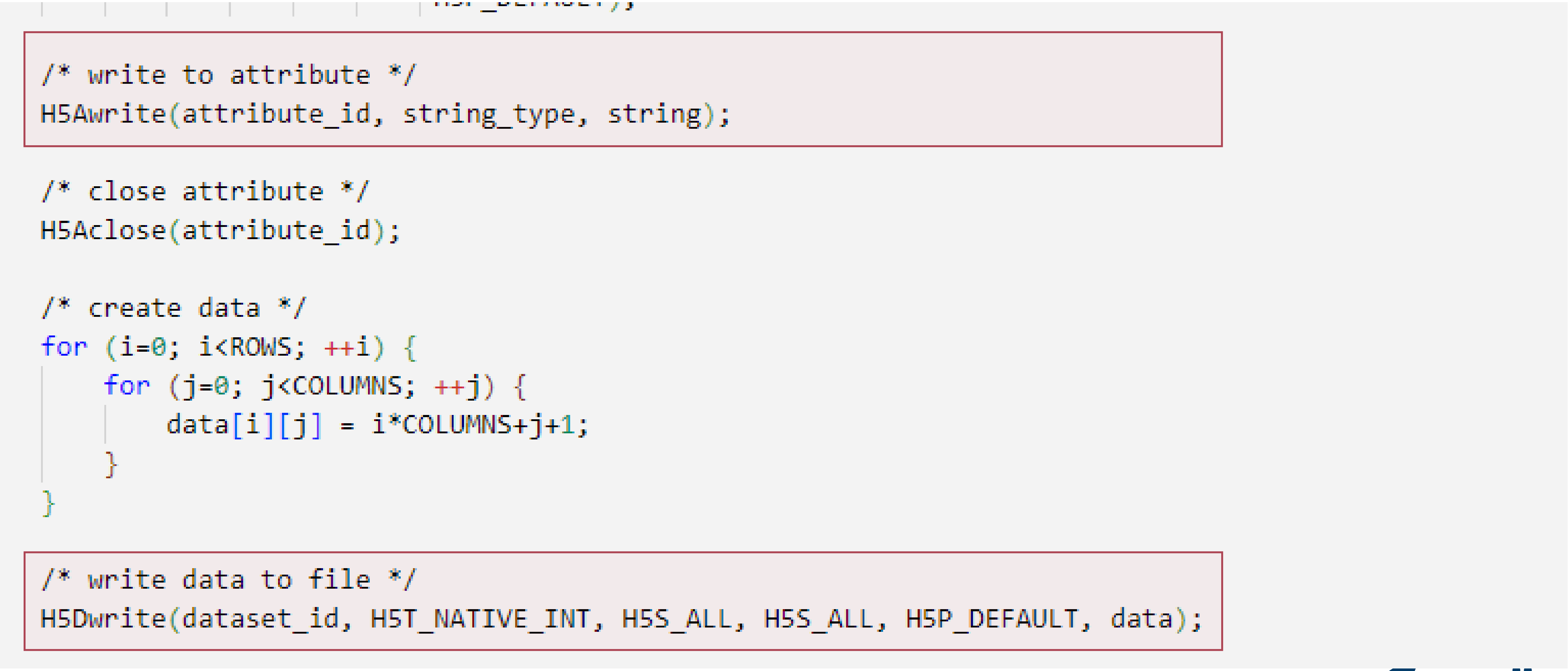
- HDF5 interfaces (both C and Python)

Follow the notebook:
`03_II_h5.ipynb`

Warm Up: Usage of HDF5 Interfaces

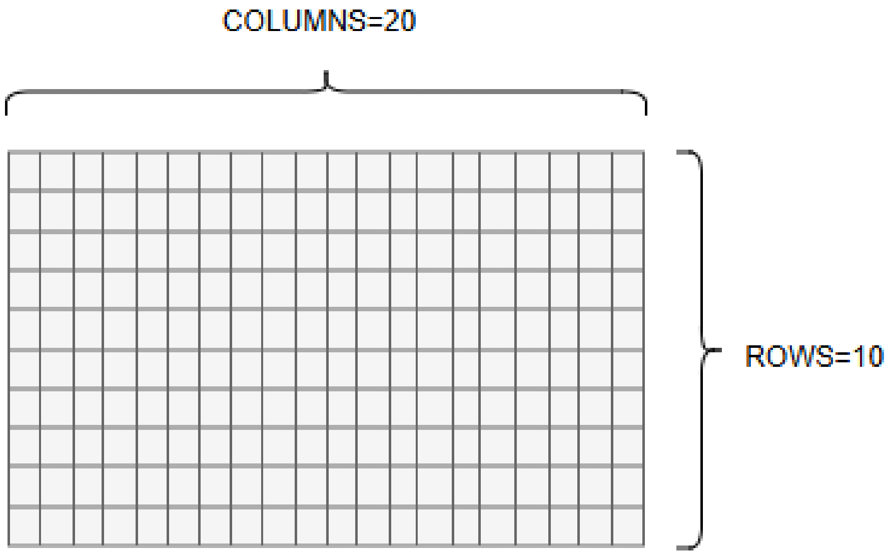


Warm Up: Usage of HDF5 Interfaces



```
                                    H5P_DEFAULT);

/* write to attribute */
H5Awrite(attribute_id, string_type, string);

/* close attribute */
H5Aclose(attribute_id);

/* create data */
for (i=0; i<ROWS; ++i) {
    for (j=0; j<COLUMNS; ++j) {
        data[i][j] = i*COLUMNS+j+1;
    }
}

/* write data to file */
H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, data);
```

I/O Analyses with STrace



Type: `Int` (4 bytes)
Data size: 10x20x4 = 800 bytes

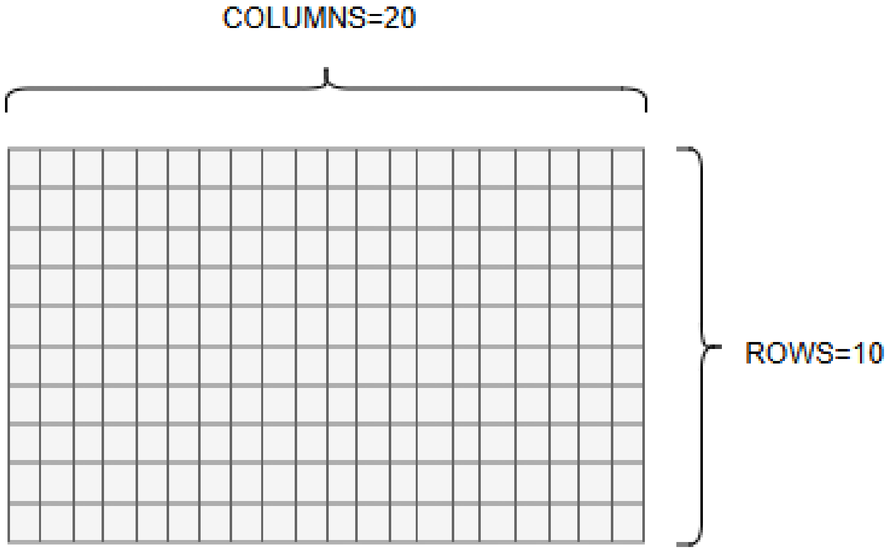
Usage:

- Prepend your command with `strace` to log the sequence of system calls.

Example command:

```
strace -y -o trace.log ./main
cat trace.log | grep “matrix.h5"
```

I/O Analyses with STrace



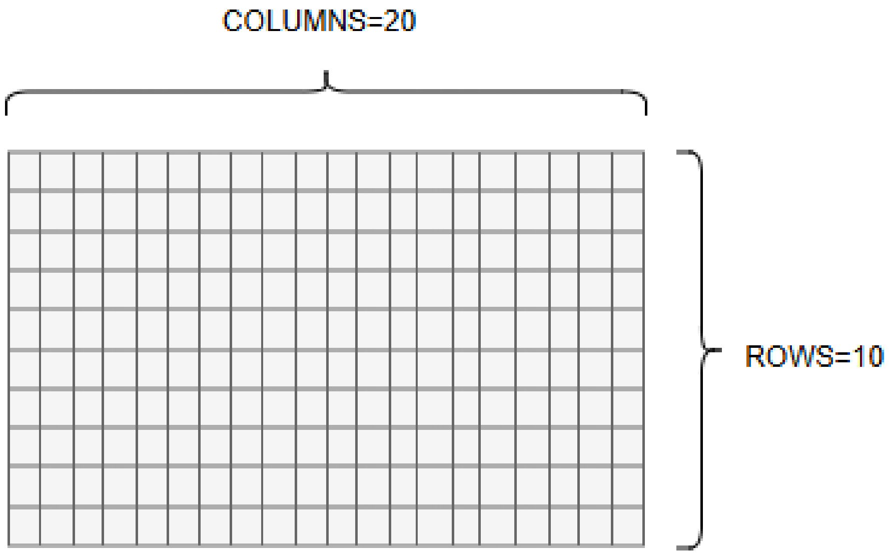
Type: `Int` (4 bytes)
Data size: 10x20x4 = 800 bytes

Revisiting the serial HDF5 program:

- Consider a two dimensional array with Integer values 1 up to 200.

- This array is written into an empty HDF5 dataset using the C API.

I/O Analyses with STrace



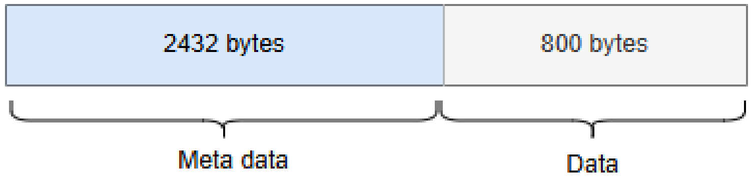
Type: `Int` (4 bytes)
Data size: 10x20x4 = 800 bytes

COMMAND:

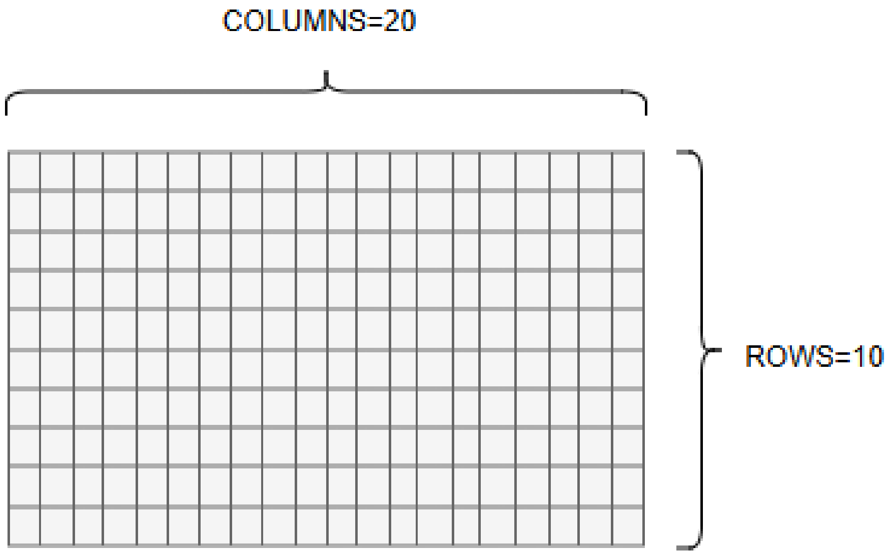
```
strace -y -o trace.log ./main
cat trace.log | grep “matrix.h5"
```

OUTPUT:

```
pwrite64(.., BYTES_REQ, OFFSET) = BYTES_WRITTEN;
pwrite64(.., 96, 0) = 96
pwrite64(..,800, 2432) = 800
pwrite64(..,2432, 0) = 2432
pwrite64(..,96, 0) = 96
```



Exercise

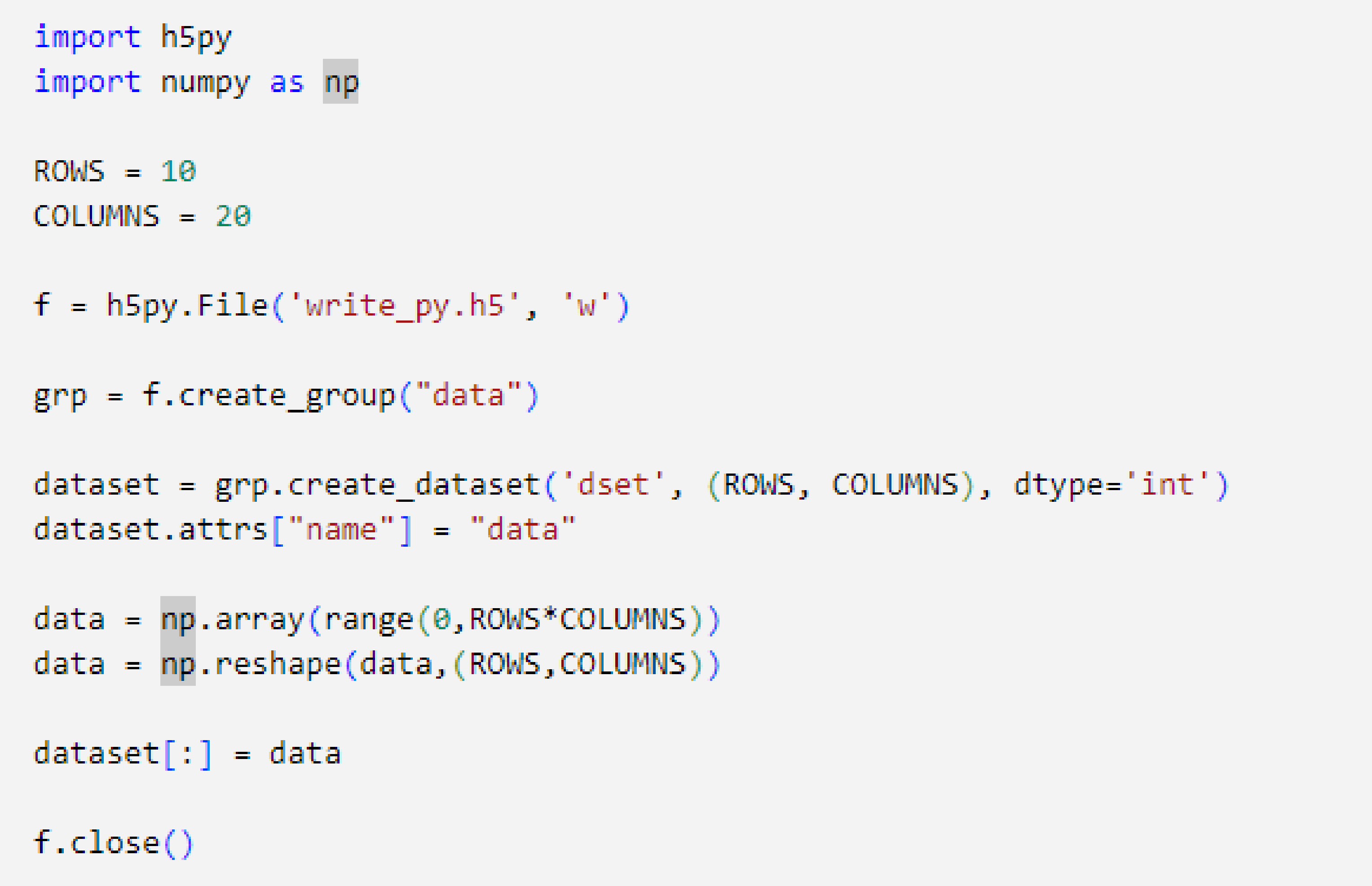


Type: `Int` (4 bytes)
Data size: 10x20x4 = 800 bytes

- Consider the writing of same matrix with the Python API and analyze with STrace. Identify and explain the differences in data and meta data sizes.

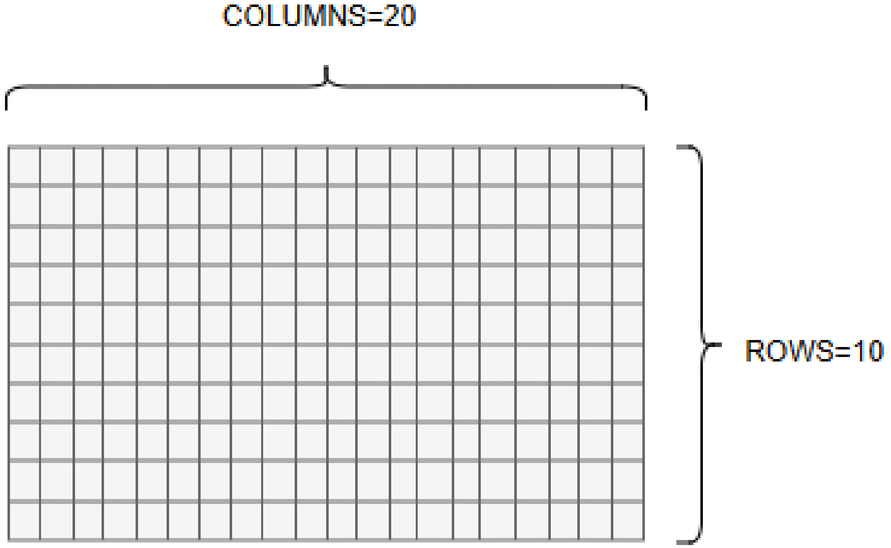
- Follow the instructions in the notebook: `03_II_h5.ipynb (Task 4)`

Exercise



```python
import h5py
import numpy as np

ROWS = 10
COLUMNS = 20

f = h5py.File('write_py.h5', 'w')

grp = f.create_group("data")

dataset = grp.create_dataset('dset', (ROWS, COLUMNS), dtype='int')
dataset.attrs["name"] = "data"

data = np.array(range(0,ROWS*COLUMNS))
data = np.reshape(data,(ROWS,COLUMNS))

dataset[:] = data

f.close()
```

Exercise



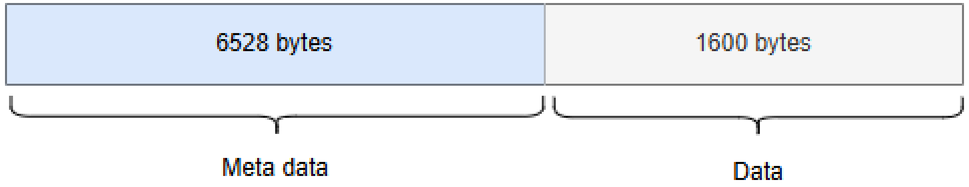
Type: `Int` (4 bytes)
Data size: 10x20x4 = 800 bytes

COMMAND:

```
strace -y -o trace.log python main.py
cat trace.log | grep "matrix_py.h5"
```

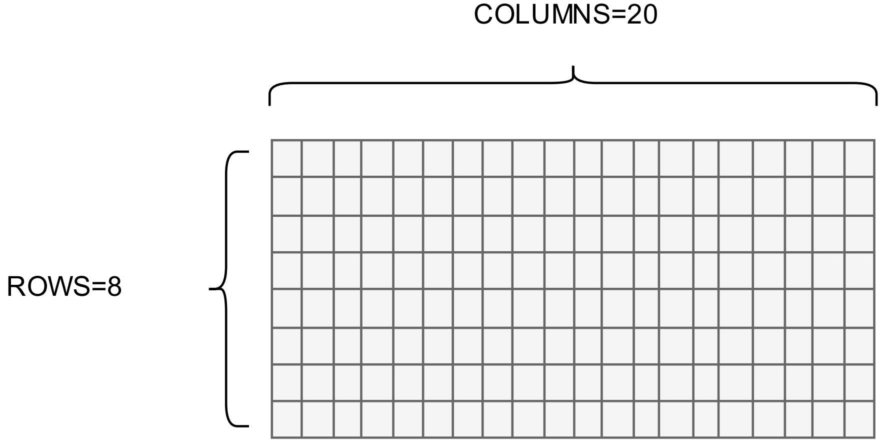
OUTPUT:

```
pwrite64(.., BYTES_REQ, OFFSET) = BYTES_WRITTEN;
pwrite64(.., 96, 0) = 96
pwrite64(..,1600, 6528) = 1600
pwrite64(..,4096, 2432) = 4096
pwrite64(..,2432, 0) = 2432
pwrite64(..,96, 0) = 96
```

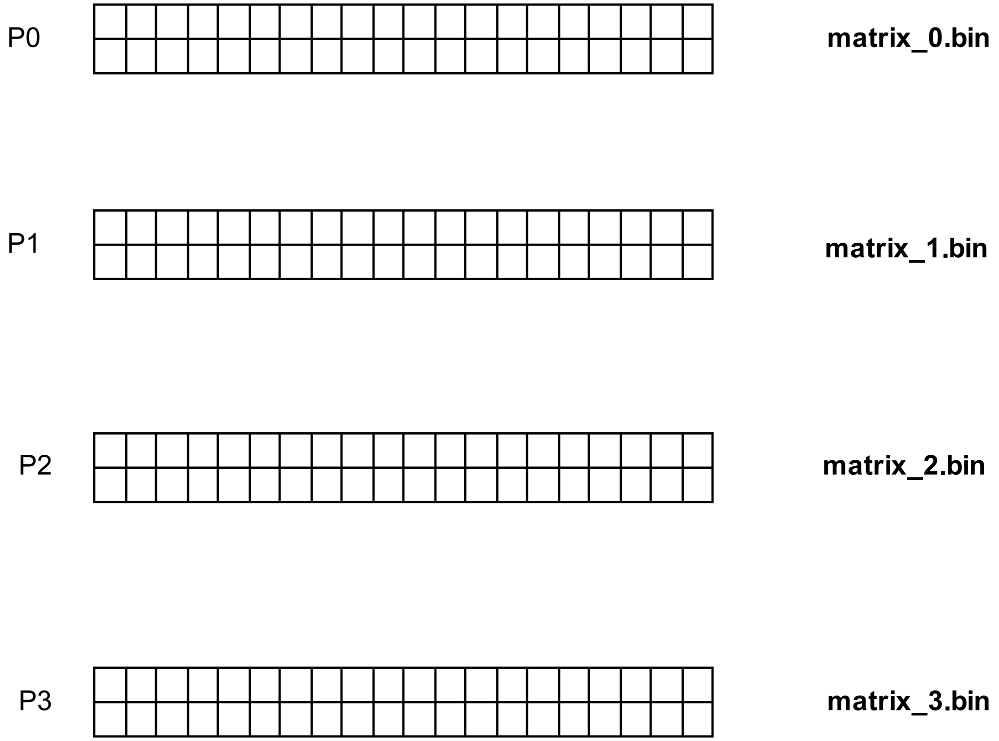


Questions:
1. Why is the data size 2x?
2. Why is the meta data size significantly high?

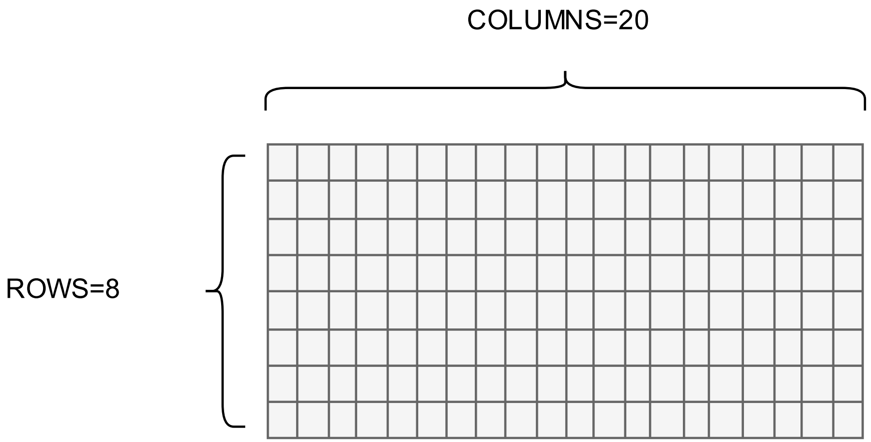
Independent I/O to Independent File (Parallel)



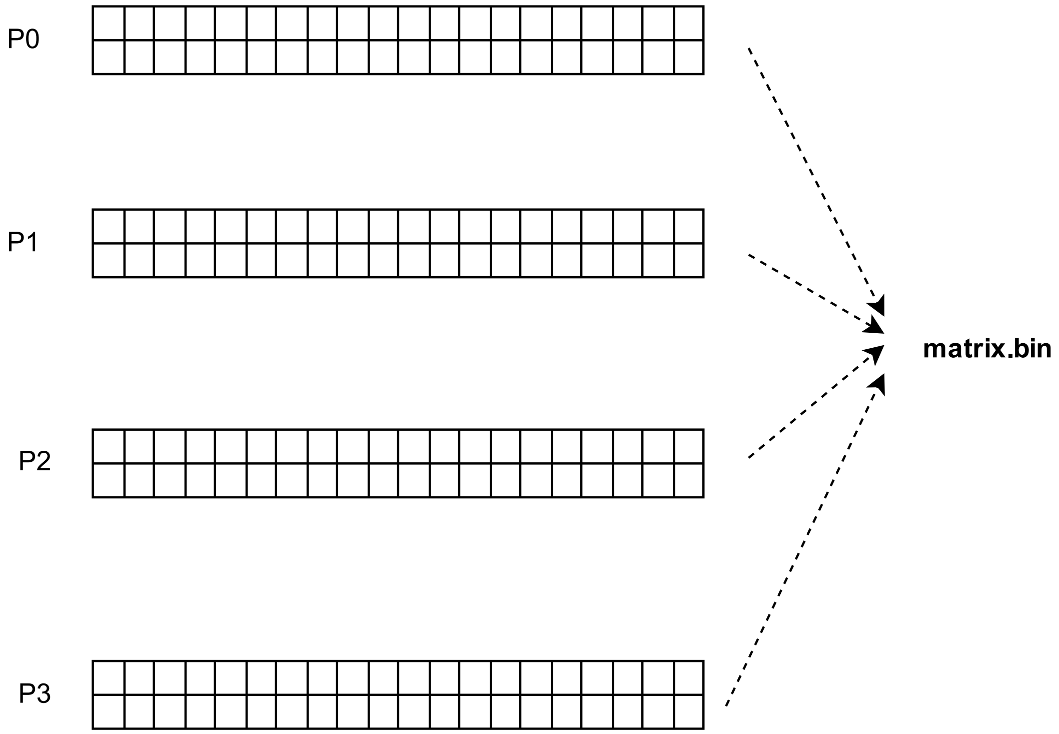
Type: `Int` (4 bytes)
Data size: 8x20x4 = 640 bytes
Size per process = 2x20x4 = 160 bytes



Independent I/O to a Shared File (STDIO)



Type: `Int` (4 bytes)
Data size: 8x20x4 = 640 bytes
Size per process = 2x20x4 = 160 bytes



Follow the notebook:
`04_IS_posix_stdio.ipynb`

Independent I/O to a Shared File (STDIO)

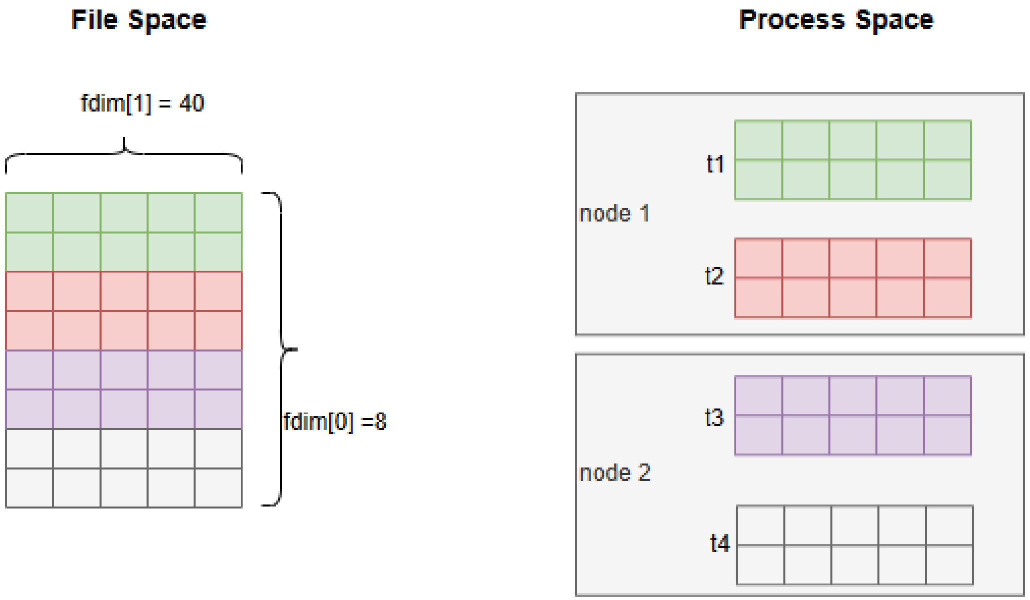
The main problem with STDIO in doing Parallel I/O:

- STDIO is not aware of which parts of a shared file other processes in the MPI communicator are trying to accesses. Hence, it cannot optimize I/O by combining accesses from multiple processes.

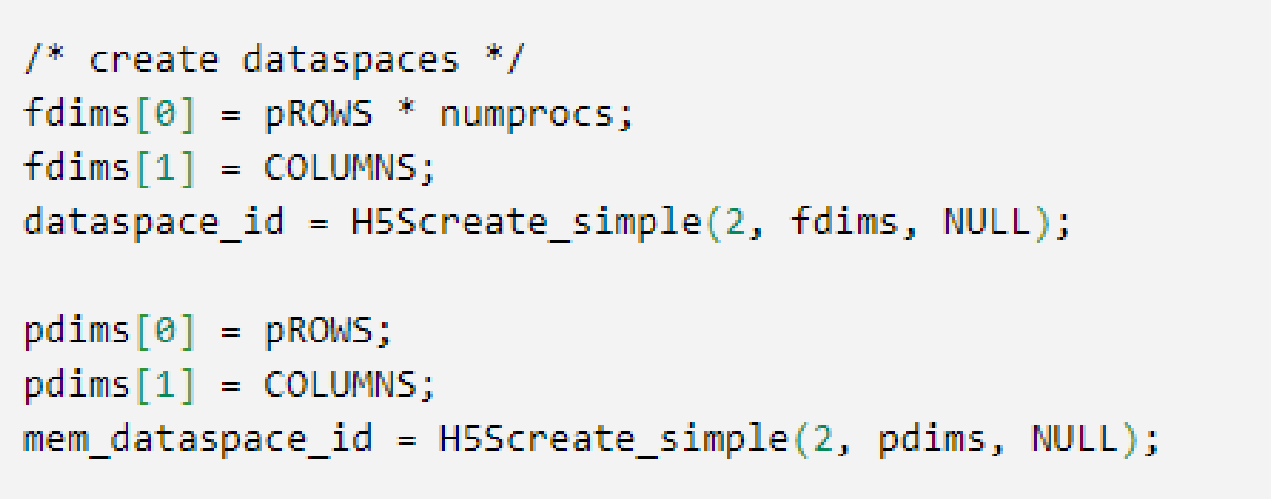
Collective I/O to a Shared File (HDF5)

Revisiting the Parallel HDF5 program:

- Consider a two dimensional array with Integer values, and the rows are split among 4 processes.
- **Follow the notebook**: `05_CS_h5_row_split.ipynb`

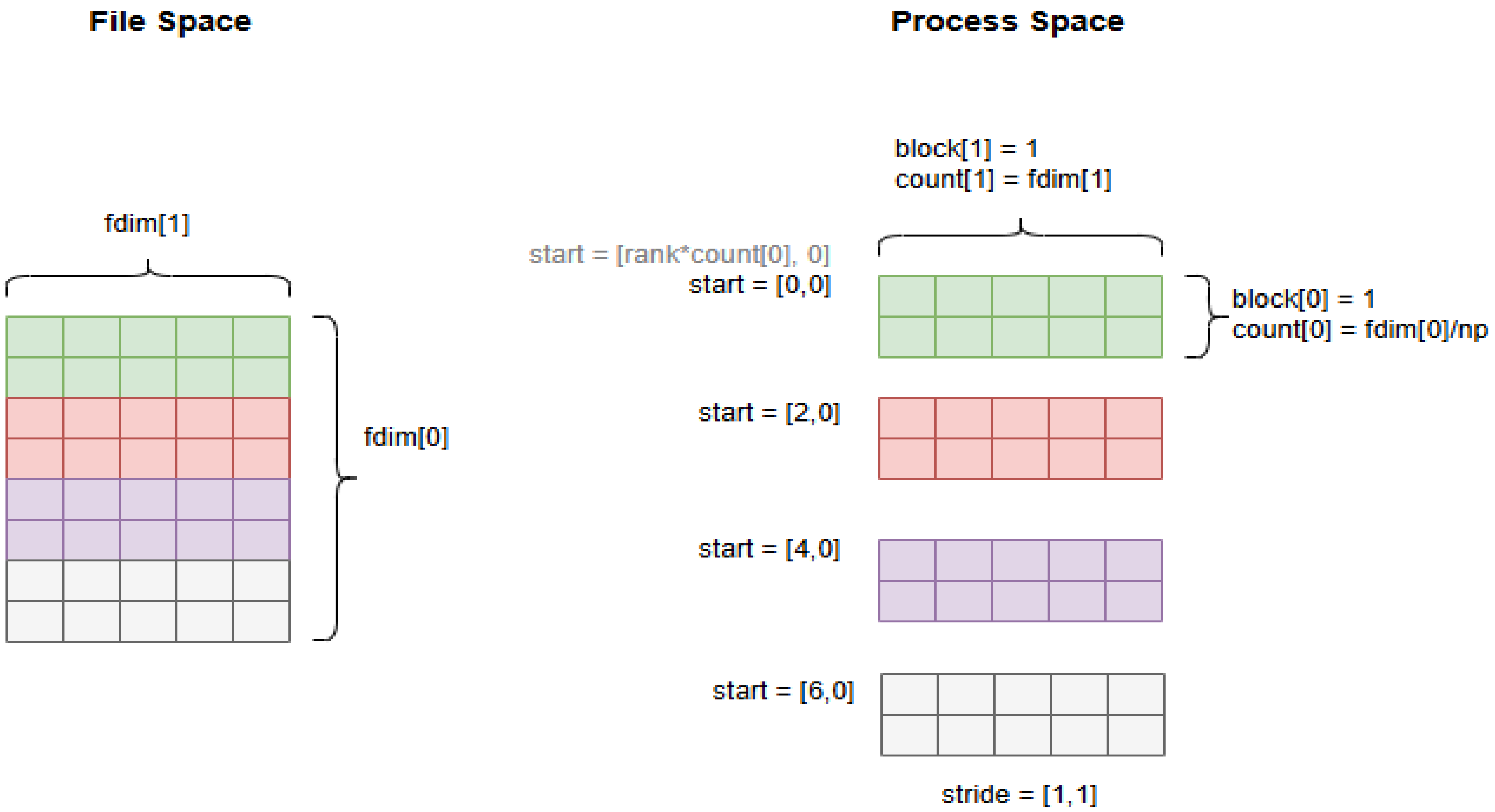


Collective I/O to a Shared File (HDF5)



```
/* create dataspaces */
fdims[0] = pROWS * numprocs;
fdims[1] = COLUMNS;
dataspace_id = H5Screate_simple(2, fdims, NULL);

pdims[0] = pROWS;
pdims[1] = COLUMNS;
mem_dataspace_id = H5Screate_simple(2, pdims, NULL);
```

Collective I/O to a Shared File (HDF5)



File Space

Process Space

fdim[1]

fdim[0]

block[1] = 1
count[1] = fdim[1]

start = [rank*count[0], 0]

start = [0,0]

block[0] = 1
count[0] = fdim[0]/np

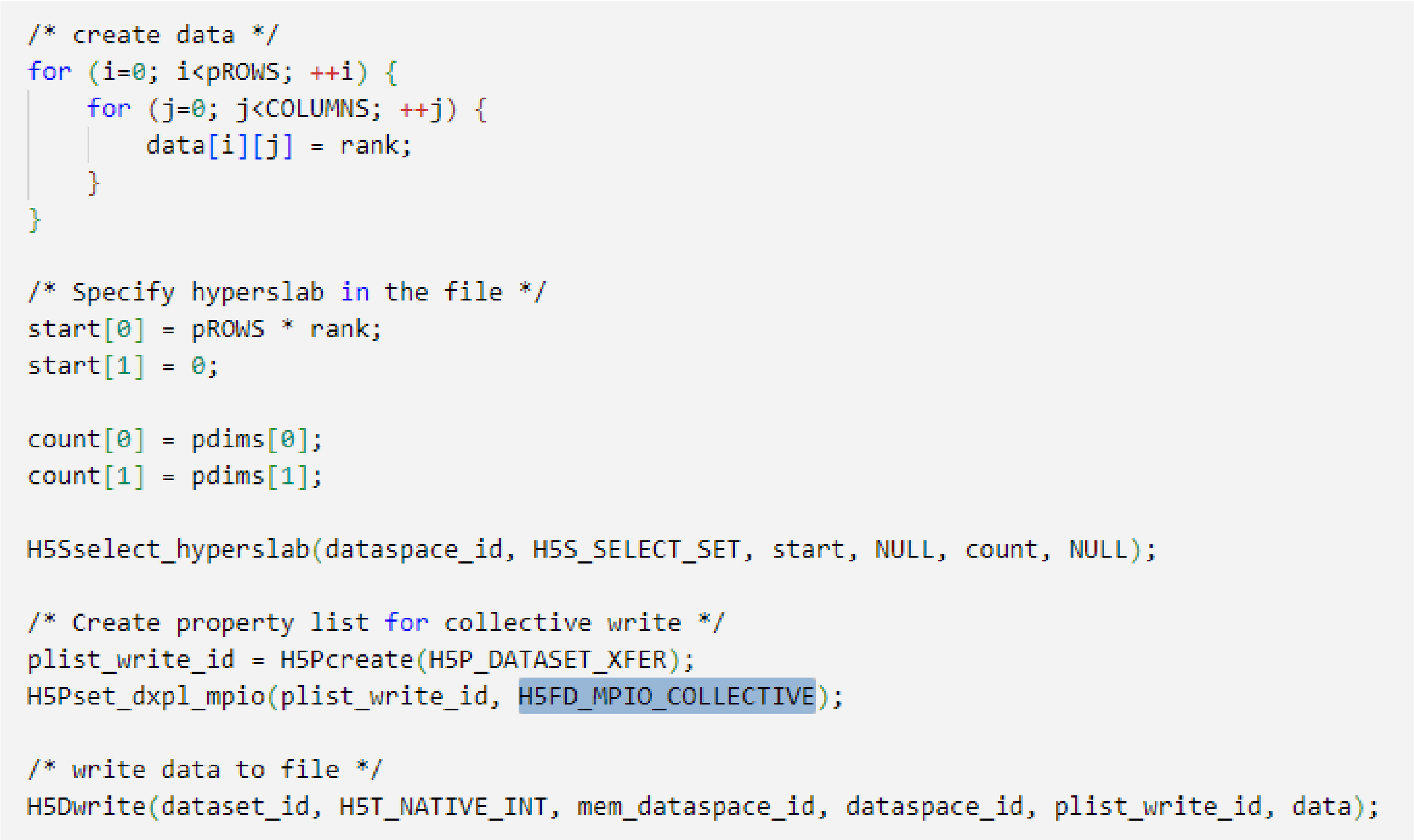
start = [2,0]

start = [4,0]

start = [6,0]

stride = [1,1]

Collective I/O to a Shared File (HDF5)



```
/* create data */
for (i=0; i<pROWS; ++i) {
    for (j=0; j<COLUMNS; ++j) {
        data[i][j] = rank;
    }
}

/* Specify hyperslab in the file */
start[0] = pROWS * rank;
start[1] = 0;

count[0] = pdims[0];
count[1] = pdims[1];

H5Sselect_hyperslab(dataspace_id, H5S_SELECT_SET, start, NULL, count, NULL);

/* Create property list for collective write */
plist_write_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_write_id, H5FD_MPIO_COLLECTIVE);

/* write data to file */
H5Dwrite(dataset_id, H5T_NATIVE_INT, mem_dataspace_id, dataspace_id, plist_write_id, data);
```

Collective I/O to a Shared File (HDF5)

```
#!/bin/bash
#SBATCH --job-name=phdf5_st
#SBATCH --output=log.out
#SBATCH --error=log.err
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2
#SBATCH --time=00:05:00
#SBATCH --partition=batch
#SBATCH --account=ACCOUNT

module purge
module load Stages/2024
module load GCC ParaStationMPI HDF5
module load strace

mpicc -o write_phdf5 write_phdf5.c -lhdf5
srun -n 4 --cpus-per-task=1 strace [OPTIONS] ./write_phdf5
```

Collective I/O to a Shared File (HDF5)

Node/task	Sys call	POSIX size	offset
Node 1 (t1)	pwrite64	1280	2140
Node 1 (t1)	pwrite64	96	0
Node 1 (t2)	pwrite64	128	680
Node 2 (t3)	pwrite64	328	1054
Node 2 (t4)	pwrite64	272	1832
Node 2 (t4)	pwrite64	328	4152
..	..	..	..

Data

MD

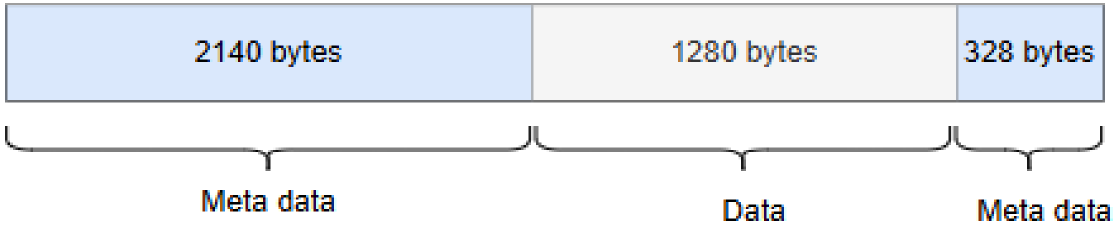
Type: `Int` (4 bytes)
Data size: 8x40x4 = 1280 bytes

Collective I/O to a Shared File (HDF5)

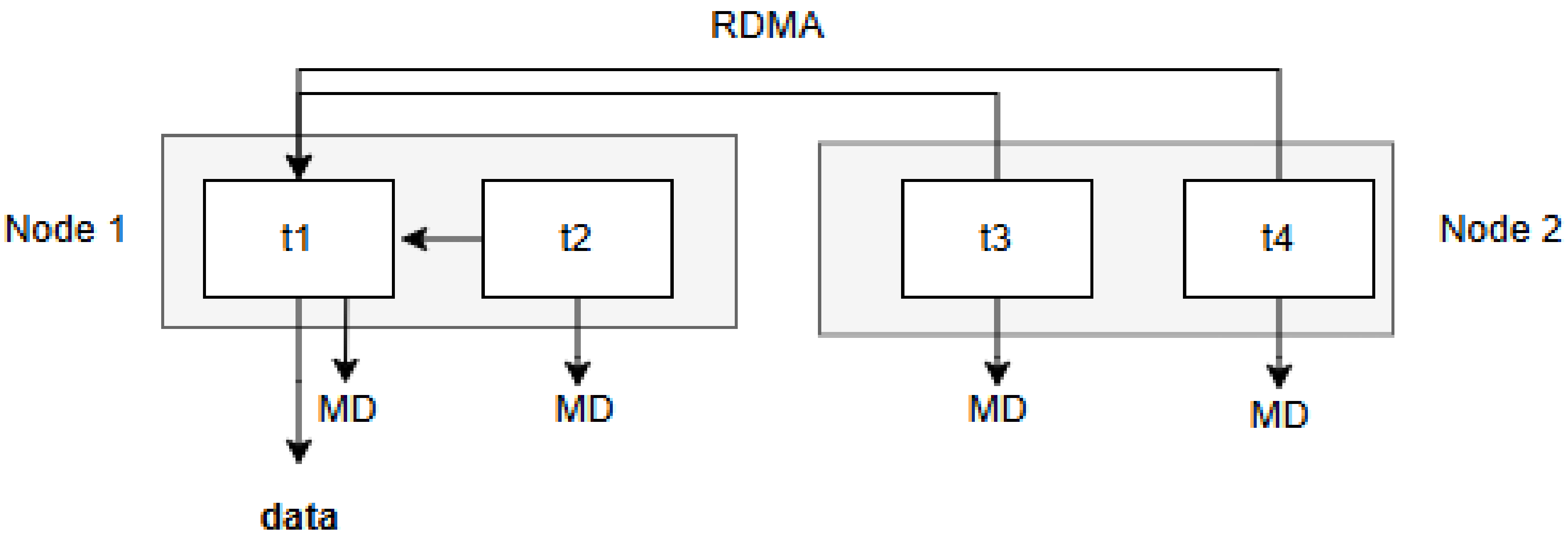
Node/task	Sys call	POSIX size	offset
Node 1 (t1)	pwrite64	1280	2140
Node 1 (t1)	pwrite64	96	0
Node 1 (t2)	pwrite64	128	680
Node 2 (t3)	pwrite64	328	1054
Node 2 (t4)	pwrite64	272	1832
Node 2 (t4)	pwrite64	328	4152
..	..	..	..

Data

MD

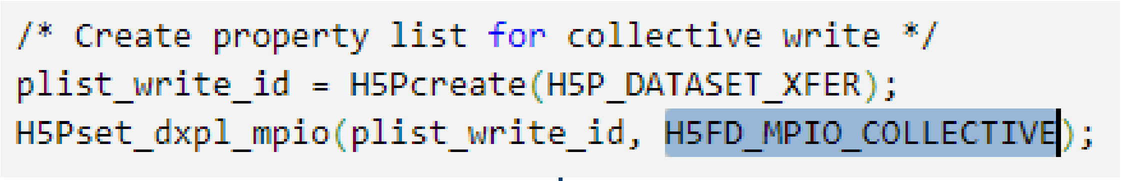


Collective I/O to a Shared File (HDF5)

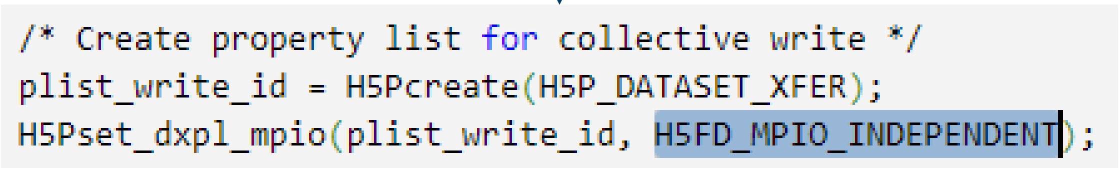


Independent I/O to a Shared File (HDF5)

- Repeat the program with **MPI IO independent accesses.**



```
/* Create property list for collective write */
plist_write_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_write_id, H5FD_MPIO_COLLECTIVE);
```



```
/* Create property list for collective write */
plist_write_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_write_id, H5FD_MPIO_INDEPENDENT);
```

Independent I/O to a Shared File (HDF5)

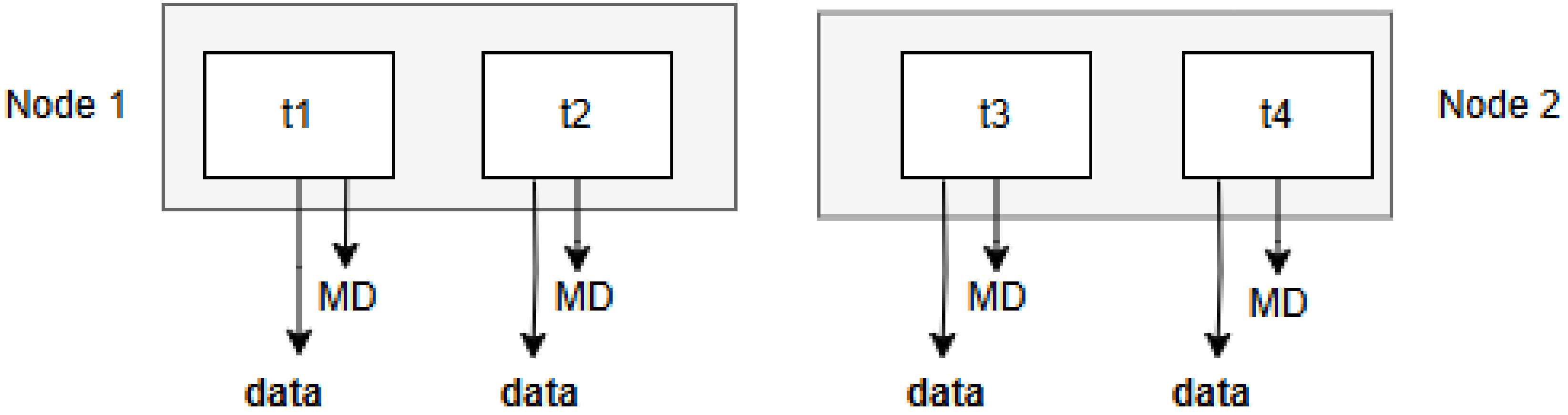
Node/task	Sys call	POSIX size	offset
Node 1 (t1)	pwrite64	320	2104
Node 1 (t2)	pwrite64	320	2424
Node 2 (t3)	pwrite64	320	2744
Node 2 (t4)	pwrite64	320	3064
Node 2 (t3)	pwrite64	272	1832
Node 2 (t4)	pwrite64	328	4152
..	..	..	..

Data

MD

Type: `Int` (4 bytes)
Data size: 8x40x4 = 1280 bytes
Num procs = 4
Independent data access size = 1280/4 = 320

Independent I/O to a Shared File (HDF5)



Writing to a Shared File (HDF5)

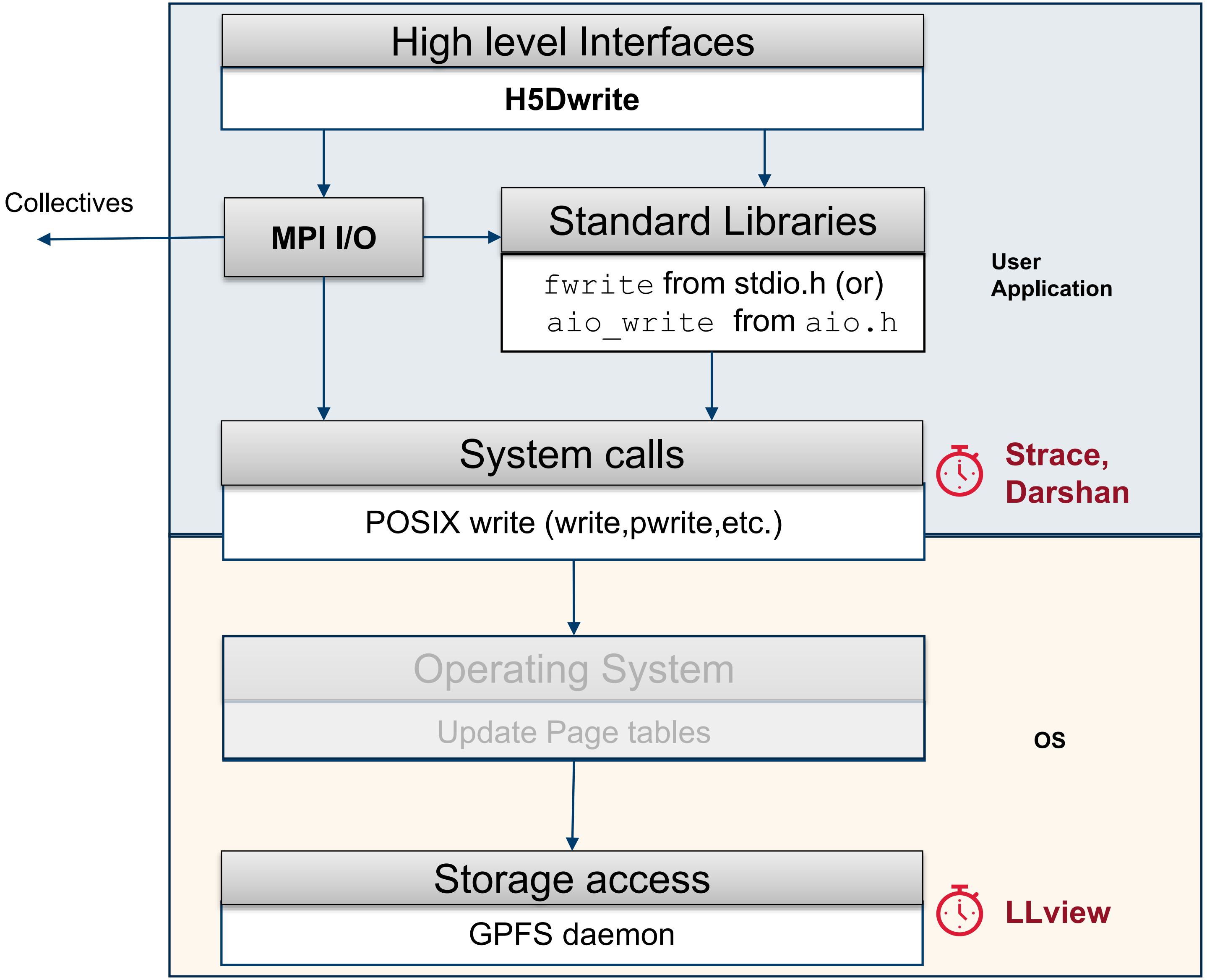
Collective I/O observations:

- The data is gathered by one process, which then **issues one POSIX call**.
- Every process writes some meta data.

Independent I/O observations:

- Each process issues one POSIX call to write its portion of data.
- Every process writes some meta data.

Discussion

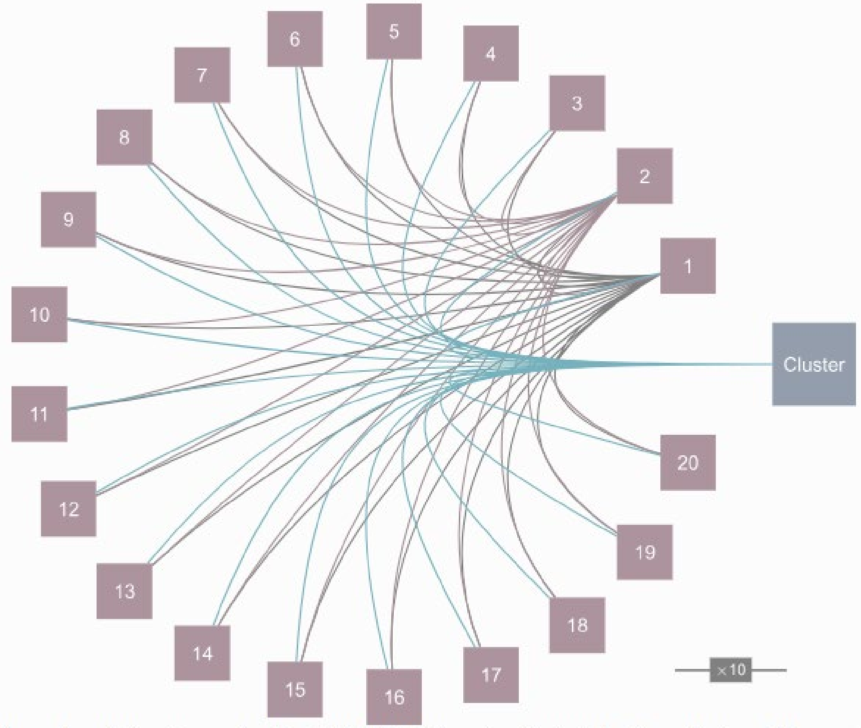


The I/O operations passing through the MPI-IO collective interface may either translate to a POSIX request or invoke an MPI communication.

Discussion

- **Collectives:** I/O contentions are often realized when scaled to large number of nodes.

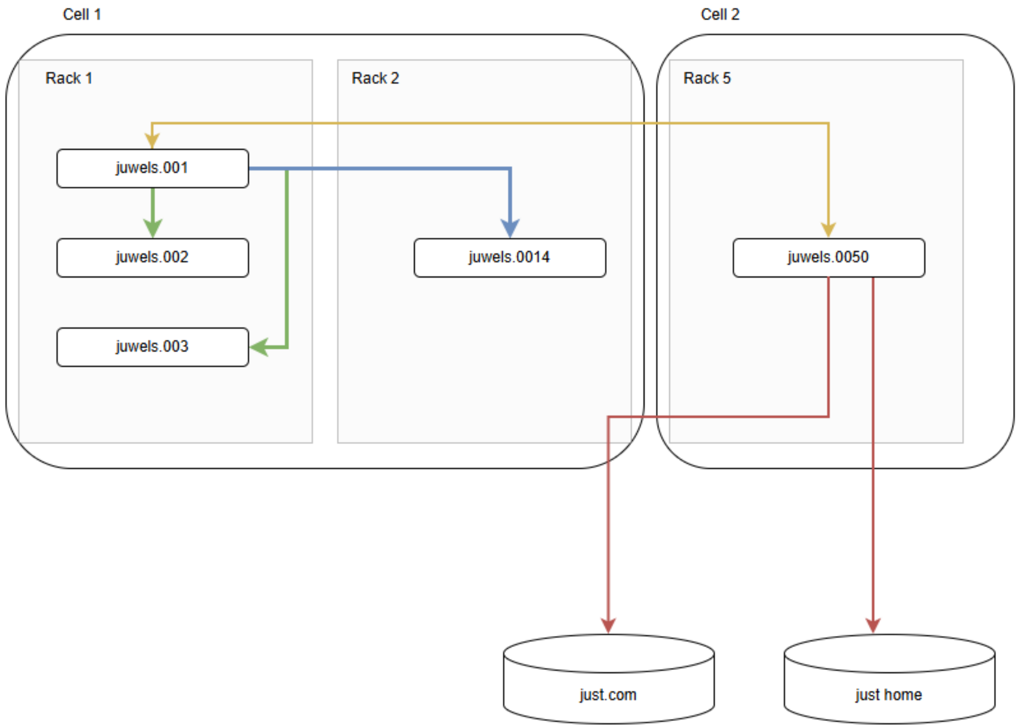
- **Independent:** Scalable, but contentions are realized when multiple processes try to access the same file system block.



Sketch of the network topology of cells of JUWELS Booster. Only links for cells 1 and 2 are shown as an example.

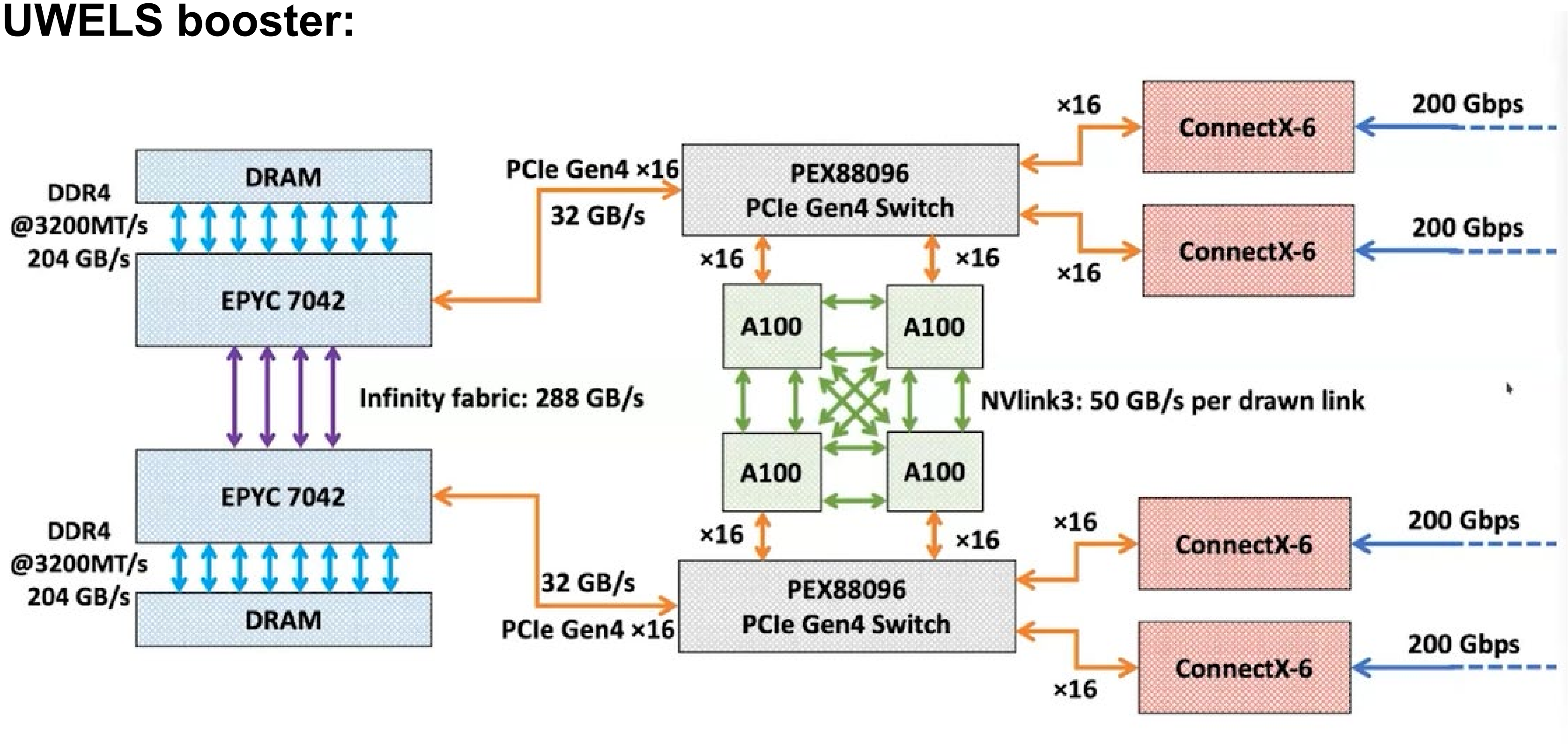
Discussion

- **Collectives:** I/O contentions are often realized when scaled to large number of nodes.
- **Independent:** Scalable, but contentions are realized when multiple processes try to access the same file system block.



The Trade-offs (Indepedent vs Collective with a node)

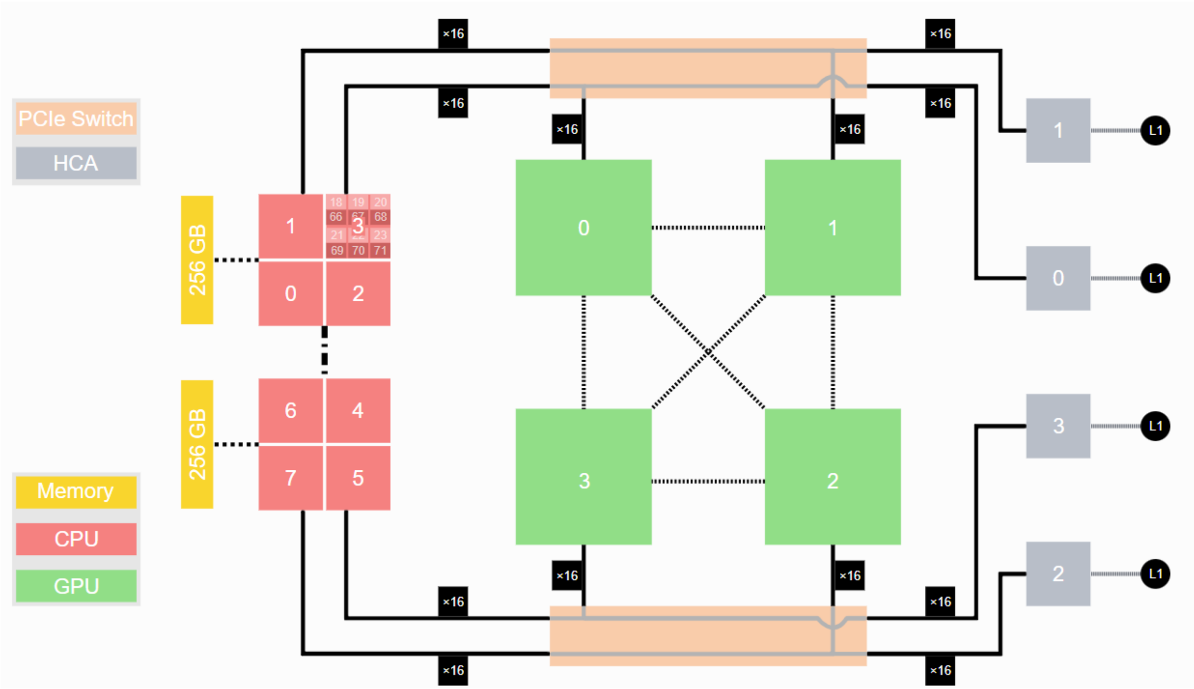
A node of JUWELS booster:



- 4 NIC cards (ConnectX-6); each with 200 Gbps or 25 GB/s link. Total: 25x4 = 100 GB/s.
- Can a single user process capitalize on all 4 NIC cards? Depends on the OS.

The Trade-offs (Indepedent vs Collective with a node)

A node of JUWELS booster:

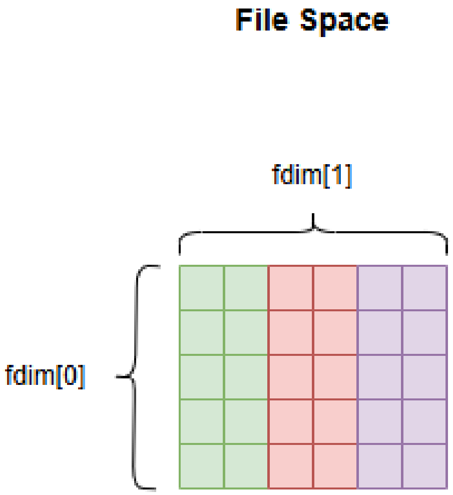


- 4 NIC cards (ConnectX-6); each with 200 Gbps or 25 GB/s link. Total: 25x4 = 100 GB/s.
- Can a single process capitalize on all 4 NIC cards? Depends on the OS.

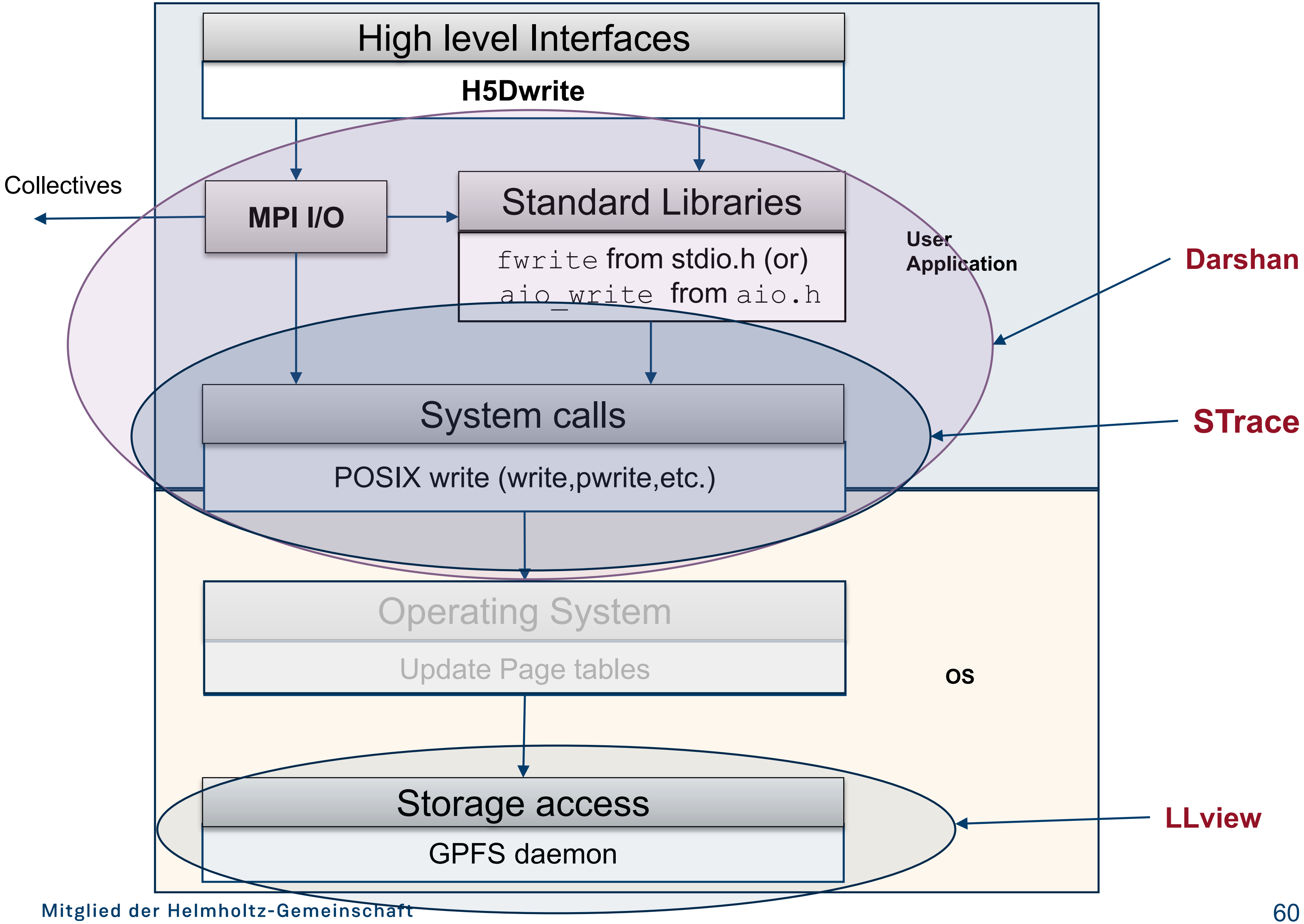
Exercise (HDF5 Collective Column split)

- Instead of rows, now split the columns of the matrix among the processes and repeat the analyses with STrace. Identify the differences.

- Follow the instructions in the notebooks: `07_h5_col_split.ipynb`



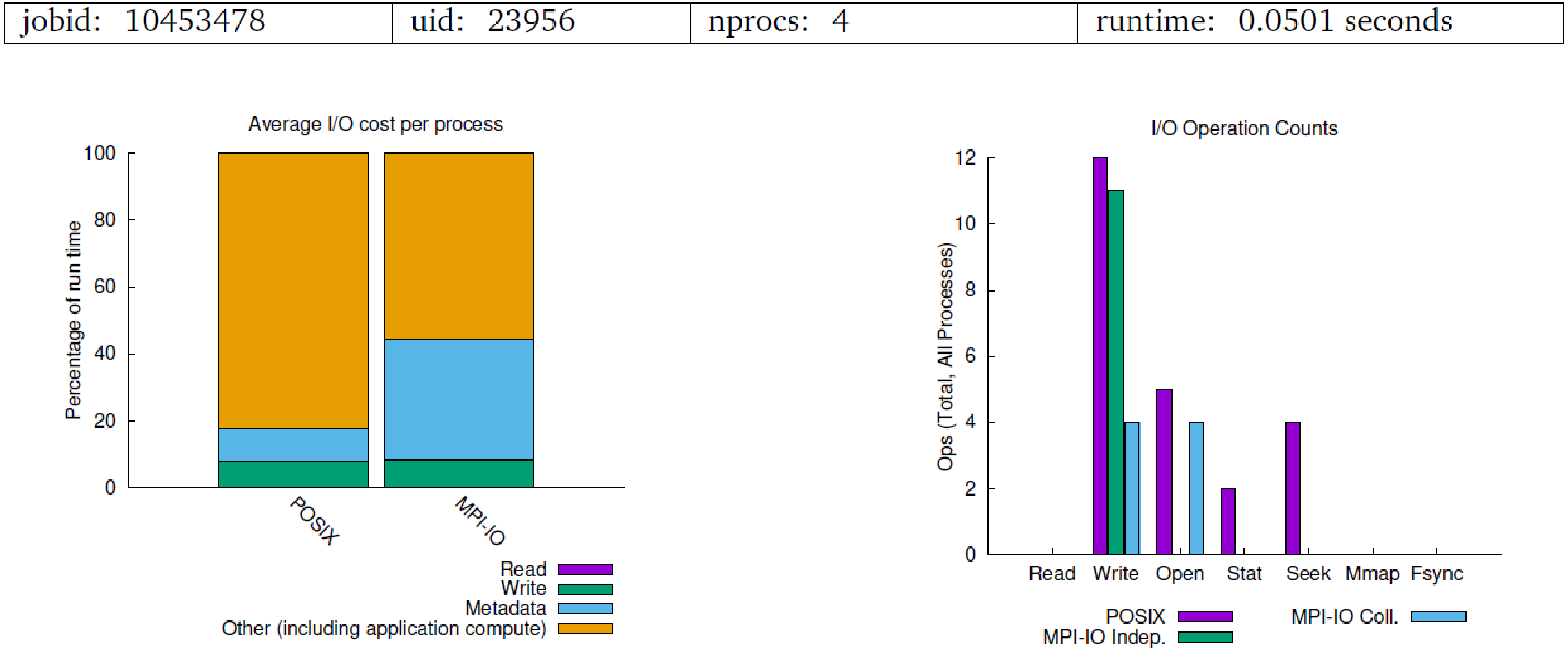
Tools in their Perspectives



Profiling with Darshan

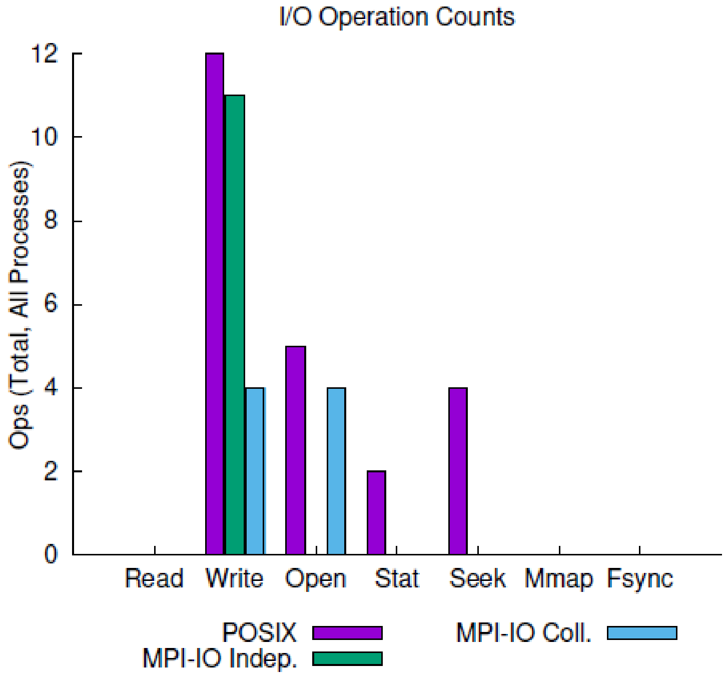
- I/O profiling tool for parallel applications
  - http://www.mcs.anl.gov/research/projects/darshan/
- Integration by using LD_PRELOAD:
  - `LD_PRELOAD=.../lib/libdarshan.so`
- `DARSHAN_LOG_PATH` points to target log directory
- `DXT_ENABLE_IO_TRACE=1` allows task specific tracing
- Analyse tools:
  - `darshan-parser:` command line access
  - `darshan-dxt-parser:` trace data access
  - `darshan-job-summary.pl:` PDF reportMore details: https://www.mcs.anl.gov/research/projects/darshan/docs/darshan-runtime.html

Darshan



jobid: 10453478	uid: 23956	nprocs: 4	runtime: 0.0501 seconds

Darshan (Operation Counts)



Row Split, MPI IO Collective

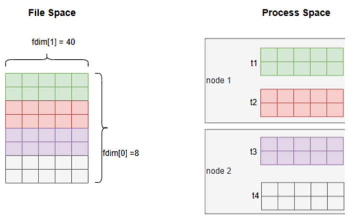
Node/task	Sys call	POSIX size	offset
Node 1 (t1)	pwrite64	1280	2140
Node 1 (t1)	pwrite64	96	0
Node 1 (t2)	pwrite64	128	680
Node 2 (t3)	pwrite64	328	1054
Node 2 (t4)	pwrite64	272	1832
Node 2 (t4)	pwrite64	328	4152
..	..	..	..

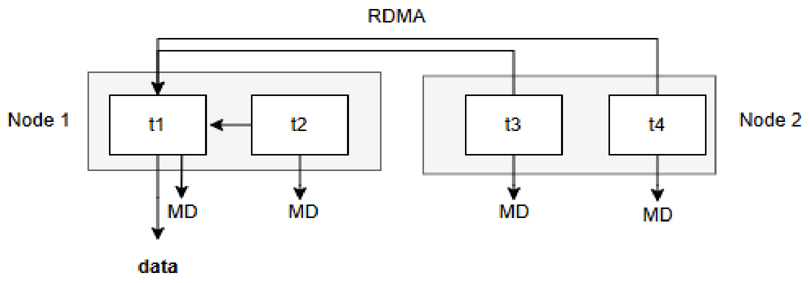
1x Data Op

11x MD Op

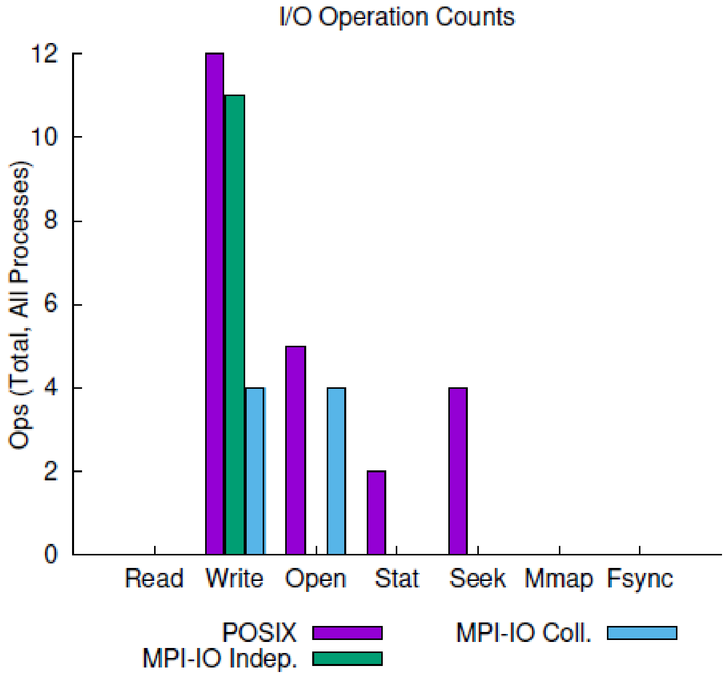
Question:

- How does the 12 POSIX calls translate to 11 MPI-IO independent and 4 MPI-IO collective calls?





Darshan (Operation Counts)



Row Split, MPI IO Collective

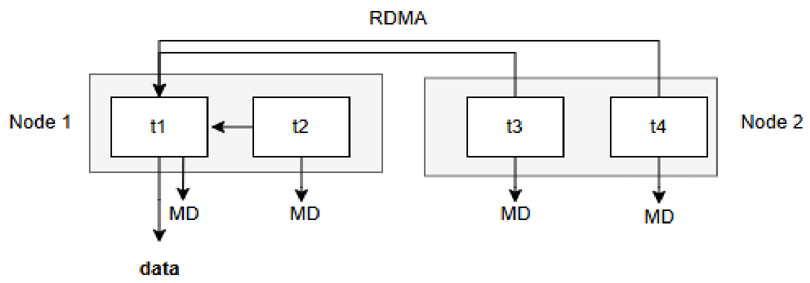
Node/task	Sys call	POSIX size	offset
Node 1 (t1)	pwrite64	1280	2140
Node 1 (t1)	pwrite64	96	0
Node 1 (t2)	pwrite64	128	680
Node 2 (t3)	pwrite64	328	1054
Node 2 (t4)	pwrite64	272	1832
Node 2 (t4)	pwrite64	328	4152
..	..	..	..

1x Data Op

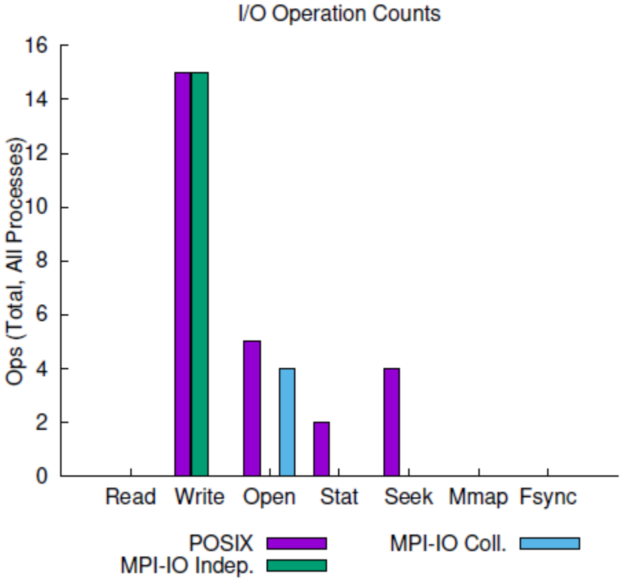
11x MD Op

Observations:

- 12 POSIX calls in total.
- 11 POSIX calls for meta data writes → 11 Independent MPI-IO calls to a shared file.
- 1 POSIX call for data transfer → after collective MPI-IO from each process.



Darshan (Access Sizes)



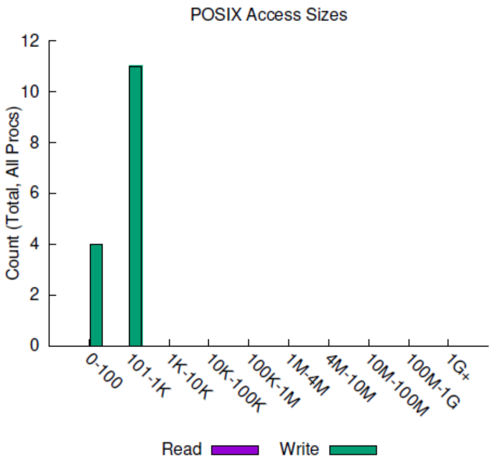
One-to-One correspondence b/w POSIX and MPI-IO

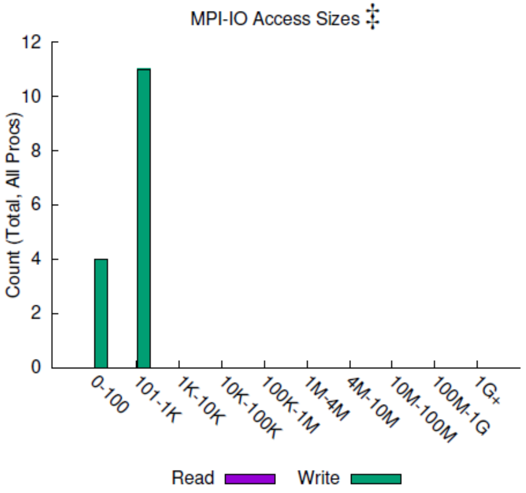
Row Split, MPI IO Independent

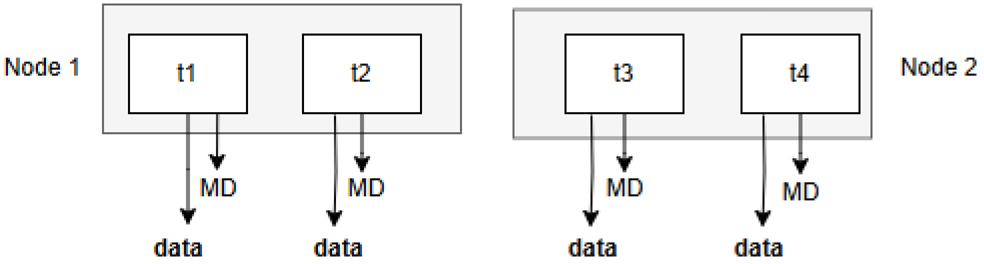
Node/task	Sys call	POSIX size	offset
Node 1 (t1)	pwrite64	320	2104
Node 1 (t2)	pwrite64	320	2424
Node 2 (t3)	pwrite64	320	2744
Node 2 (t4)	pwrite64	320	3064
Node 2 (t3)	pwrite64	272	1832
Node 2 (t4)	pwrite64	328	4152
..	..	..	..

4x Data

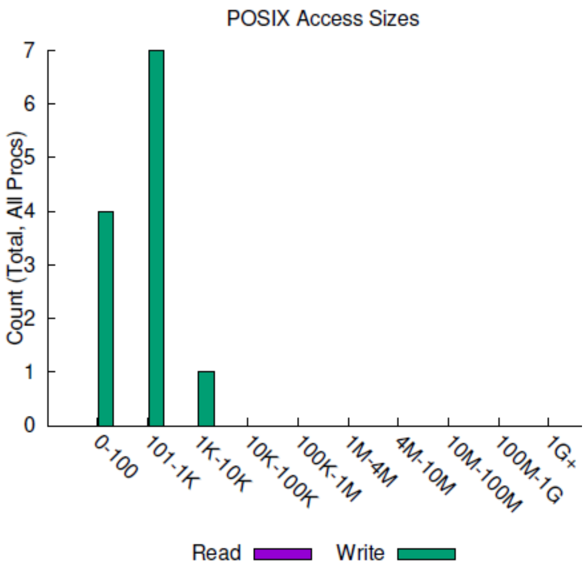
11x MD

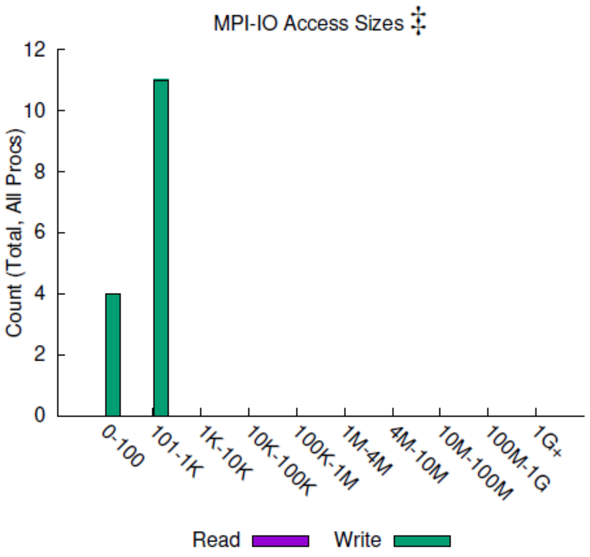


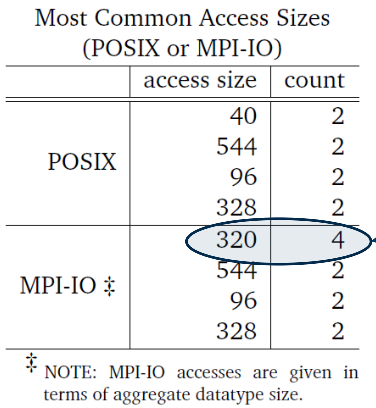




Darshan (Access Sizes)







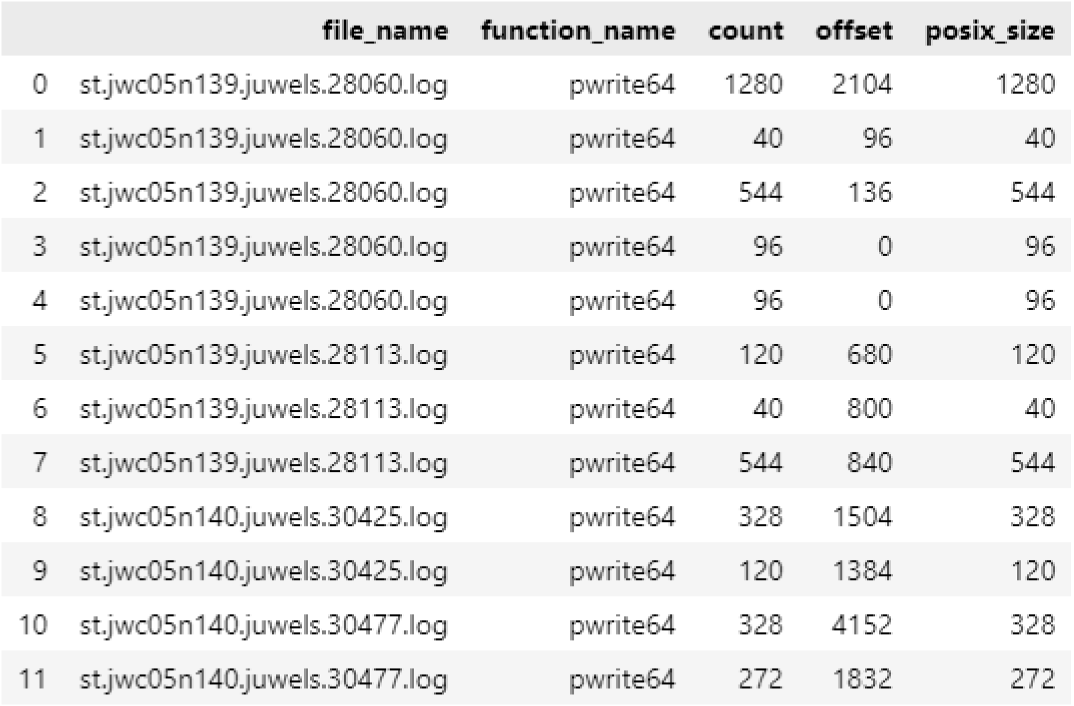
Most Common Access Sizes
(POSIX or MPI-IO)

	access size	count
POSIX	40	2
	544	2
	96	2
	328	2
MPI-IO ‡	320	4
	544	2
	96	2
	328	2

‡ NOTE: MPI-IO accesses are given in terms of aggregate datatype size.

1280/4 = 320

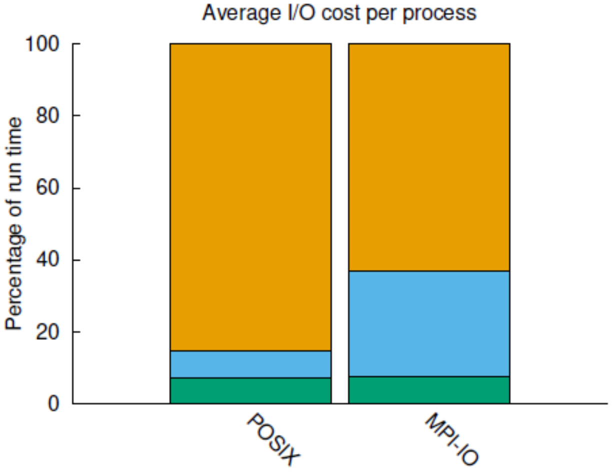
Row Split, MPI IO Collective



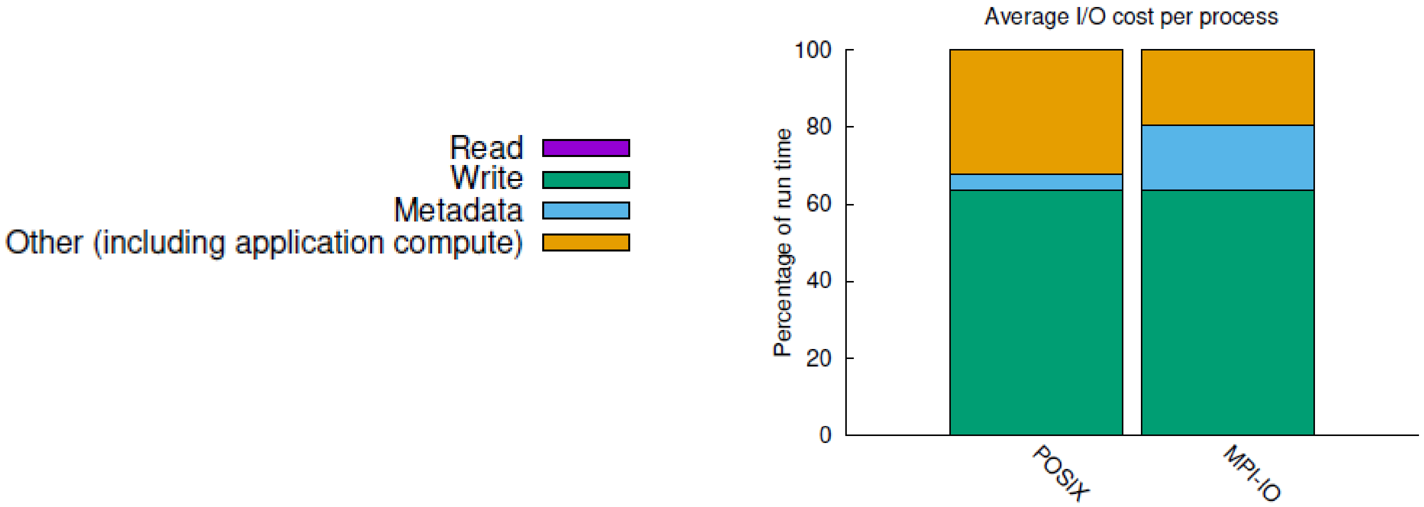
	file_name	function_name	count	offset	posix_size
0	st.jwc05n139.juwels.28060.log	pwrite64	1280	2104	1280
1	st.jwc05n139.juwels.28060.log	pwrite64	40	96	40
2	st.jwc05n139.juwels.28060.log	pwrite64	544	136	544
3	st.jwc05n139.juwels.28060.log	pwrite64	96	0	96
4	st.jwc05n139.juwels.28060.log	pwrite64	96	0	96
5	st.jwc05n139.juwels.28113.log	pwrite64	120	680	120
6	st.jwc05n139.juwels.28113.log	pwrite64	40	800	40
7	st.jwc05n139.juwels.28113.log	pwrite64	544	840	544
8	st.jwc05n140.juwels.30425.log	pwrite64	328	1504	328
9	st.jwc05n140.juwels.30425.log	pwrite64	120	1384	120
10	st.jwc05n140.juwels.30477.log	pwrite64	328	4152	328
11	st.jwc05n140.juwels.30477.log	pwrite64	272	1832	272

Darshan

Collective



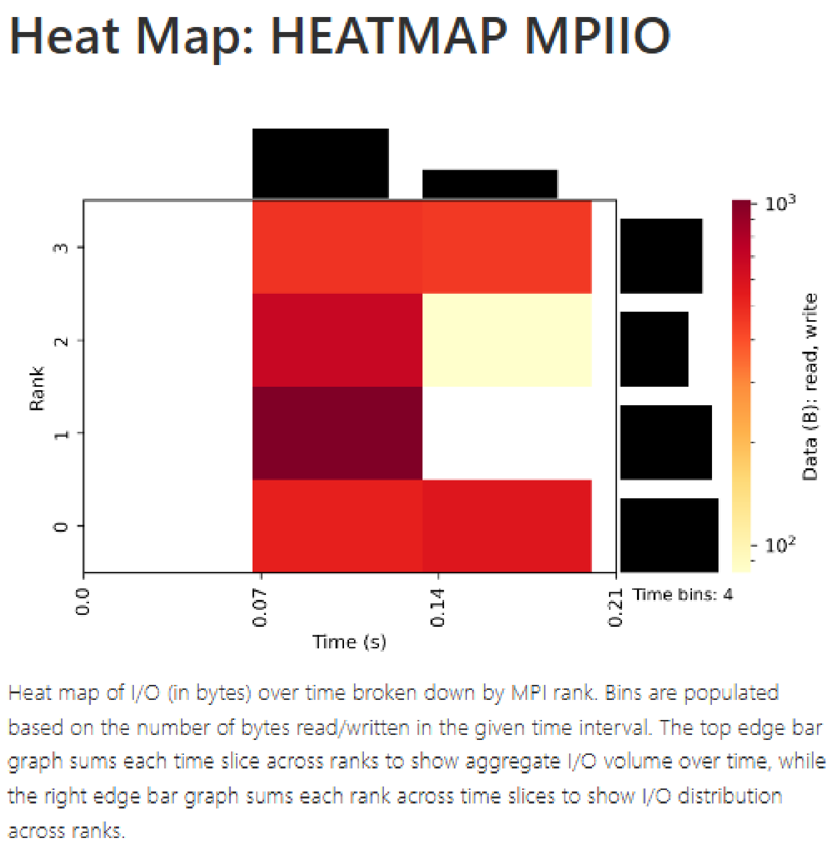
Independent



Note:

- The high overhead for independent access is not because of increased number of POSIX writes, but due to file lock contentions resulting from multiple processes trying to update the same file system block.

Darshan (Heat Map)



Heat Map: HEATMAP MPIIO

Heat map of I/O (in bytes) over time broken down by MPI rank. Bins are populated based on the number of bytes read/written in the given time interval. The top edge bar graph sums each time slice across ranks to show aggregate I/O volume over time, while the right edge bar graph sums each rank across time slices to show I/O distribution across ranks.

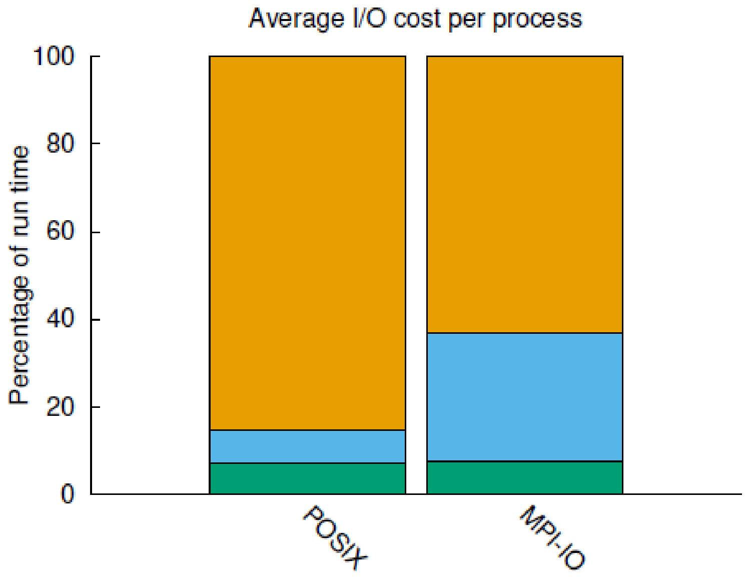
Darshan: Usage summary

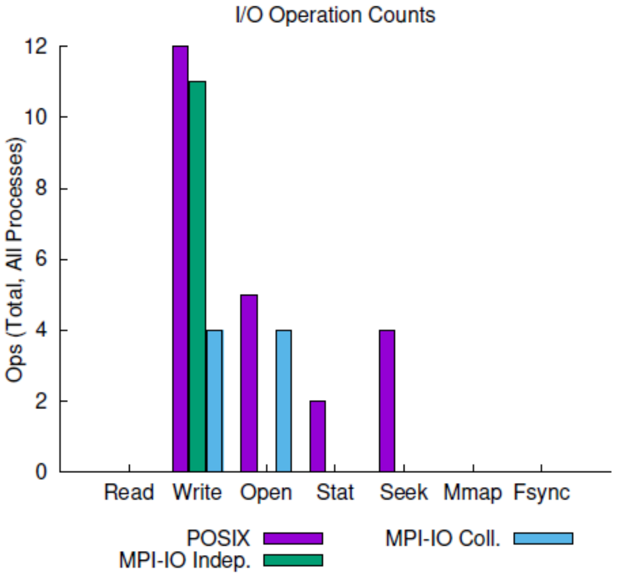
- Load module
  - `module load darshan-runtime`
- Tell srun to use Darshan (in submit script)

  -
    ```
    LD_PRELOAD=$EBROOTDARSHANMINRUNTIME/lib/libdarshan.so \
    DARSHAN_LOG_PATH=/path/to/your/logdir \
    srun … ./executable
    ```
- Analyse output
  - `module load darshan-util`
  - `darshan-job-summary.pl <logfile>.darshan`

STrace Inspector

- Darshan considers only a set of files directly accessed by the application through POSIX or high level library interfaces and aggregates the I/O operations related to those files.

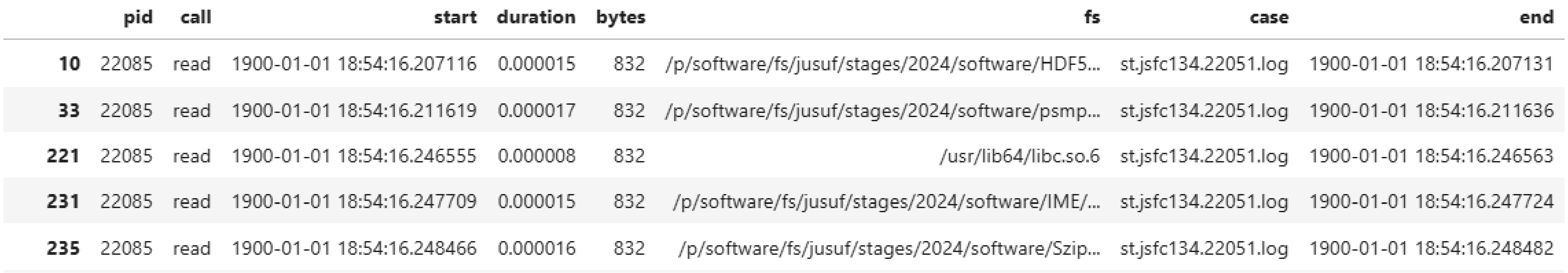




- However, the application accesses a lot more files and the system call traces hold those information!

STrace Inspector

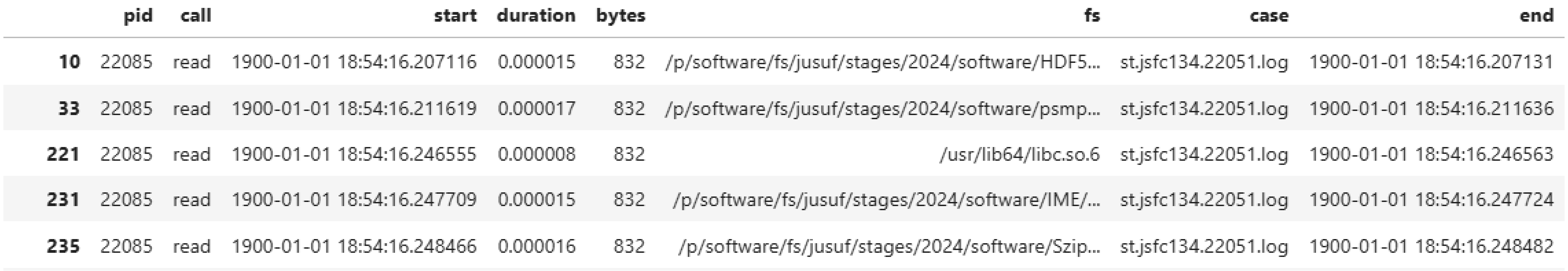
Challenge: How do you extract useful information from large amounts of information in the system call traces?



	pid	call	start	duration	bytes	fs	case	end
10	22085	read	1900-01-01 18:54:16.207116	0.000015	832	/p/software/fs/jusuf/stages/2024/software/HDF5...	st.jsfc134.22051.log	1900-01-01 18:54:16.207131
33	22085	read	1900-01-01 18:54:16.211619	0.000017	832	/p/software/fs/jusuf/stages/2024/software/psmp...	st.jsfc134.22051.log	1900-01-01 18:54:16.211636
221	22085	read	1900-01-01 18:54:16.246555	0.000008	832	/usr/lib64/libc.so.6	st.jsfc134.22051.log	1900-01-01 18:54:16.246563
231	22085	read	1900-01-01 18:54:16.247709	0.000015	832	/p/software/fs/jusuf/stages/2024/software/IME/...	st.jsfc134.22051.log	1900-01-01 18:54:16.247724
235	22085	read	1900-01-01 18:54:16.248466	0.000016	832	/p/software/fs/jusuf/stages/2024/software/Szip...	st.jsfc134.22051.log	1900-01-01 18:54:16.248482

⋮

STrace Inspector



	pid	call	start	duration	bytes	fs	case	end
10	22085	read	1900-01-01 18:54:16.207116	0.000015	832	/p/software/fs/jusuf/stages/2024/software/HDF5...	st.jsfc134.22051.log	1900-01-01 18:54:16.207131
33	22085	read	1900-01-01 18:54:16.211619	0.000017	832	/p/software/fs/jusuf/stages/2024/software/psmp...	st.jsfc134.22051.log	1900-01-01 18:54:16.211636
221	22085	read	1900-01-01 18:54:16.246555	0.000008	832	/usr/lib64/libc.so.6	st.jsfc134.22051.log	1900-01-01 18:54:16.246563
231	22085	read	1900-01-01 18:54:16.247709	0.000015	832	/p/software/fs/jusuf/stages/2024/software/IME/...	st.jsfc134.22051.log	1900-01-01 18:54:16.247724
235	22085	read	1900-01-01 18:54:16.248466	0.000016	832	/p/software/fs/jusuf/stages/2024/software/Szip...	st.jsfc134.22051.log	1900-01-01 18:54:16.248482

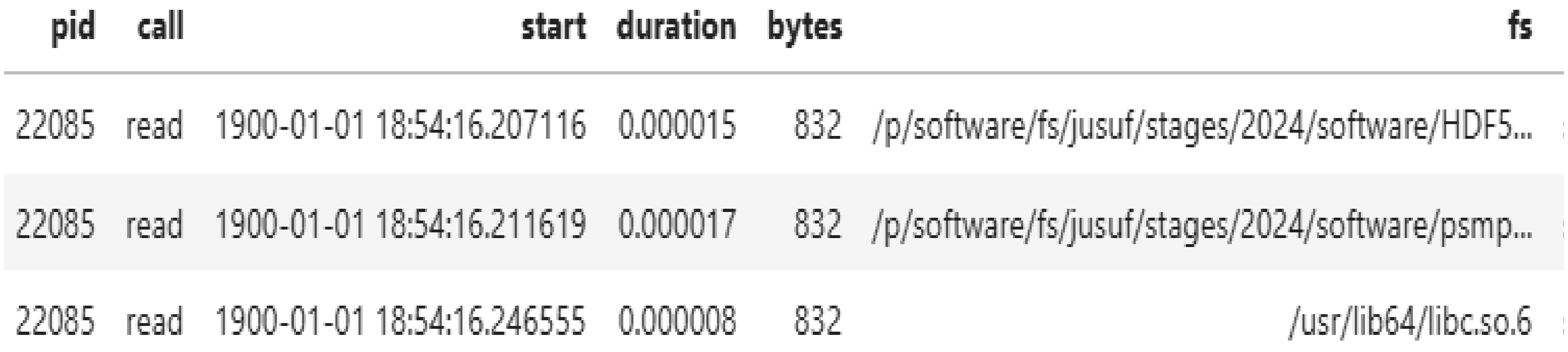
Typical questions one could ask looking at the above data:

- What is the total read time spent on the directory **/p/software**?
- How much I/O time is spent on system activities, i.e., under **/sys/?**

STrace Inspector

Idea:

- Classify each row to a string that helps answer your question. We call this string “Activity”.
- Apply grouping based on activities and compute statistics.
- Identify dependency relations (e.g., directly-follows relation) between the activities.



pid	call	start	duration	bytes	fs
22085	read	1900-01-01 18:54:16.207116	0.000015	832	/p/software/fs/jusuf/stages/2024/software/HDF5...
22085	read	1900-01-01 18:54:16.211619	0.000017	832	/p/software/fs/jusuf/stages/2024/software/psmp...
22085	read	1900-01-01 18:54:16.246555	0.000008	832	/usr/lib64/libc.so.6

Activity

→ read+/p/software

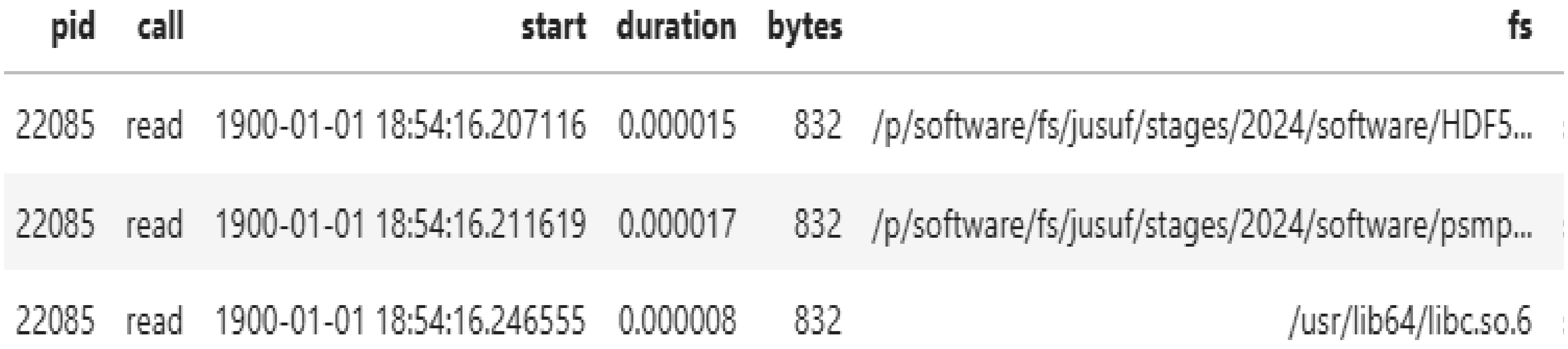
→ read+/p/software

→ read+/usr/lib64

STrace Inspector

Idea:

- Apply Process Mining techniques.
- Ref: W. M. P. Van Der Aalst, “Foundations of process discovery,” in Process Mining Handbook, DOI: https://doi.org/10.1007/978-3-031-08848-3_2



pid	call	start	duration	bytes	fs
22085	read	1900-01-01 18:54:16.207116	0.000015	832	/p/software/fs/jusuf/stages/2024/software/HDF5...
22085	read	1900-01-01 18:54:16.211619	0.000017	832	/p/software/fs/jusuf/stages/2024/software/psmp...
22085	read	1900-01-01 18:54:16.246555	0.000008	832	/usr/lib64/libc.so.6

<u>Activity</u>

→ read+/p/software

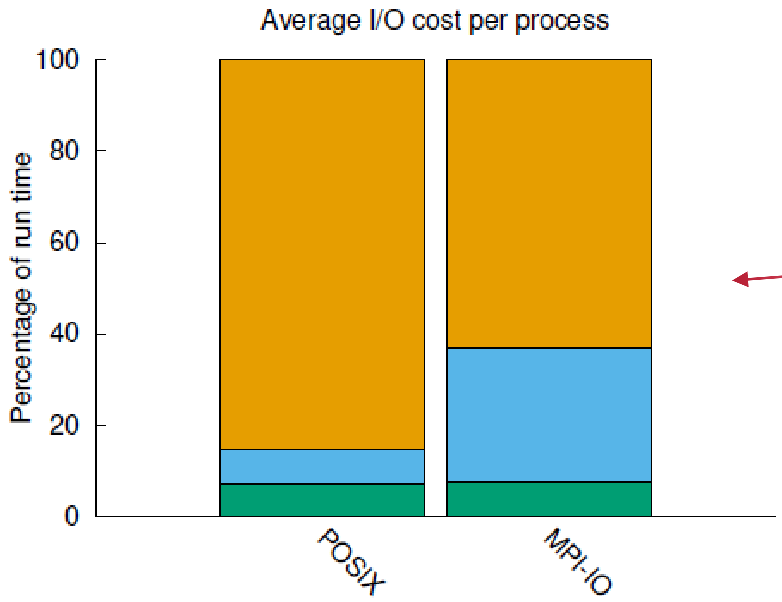
→ read+/p/software

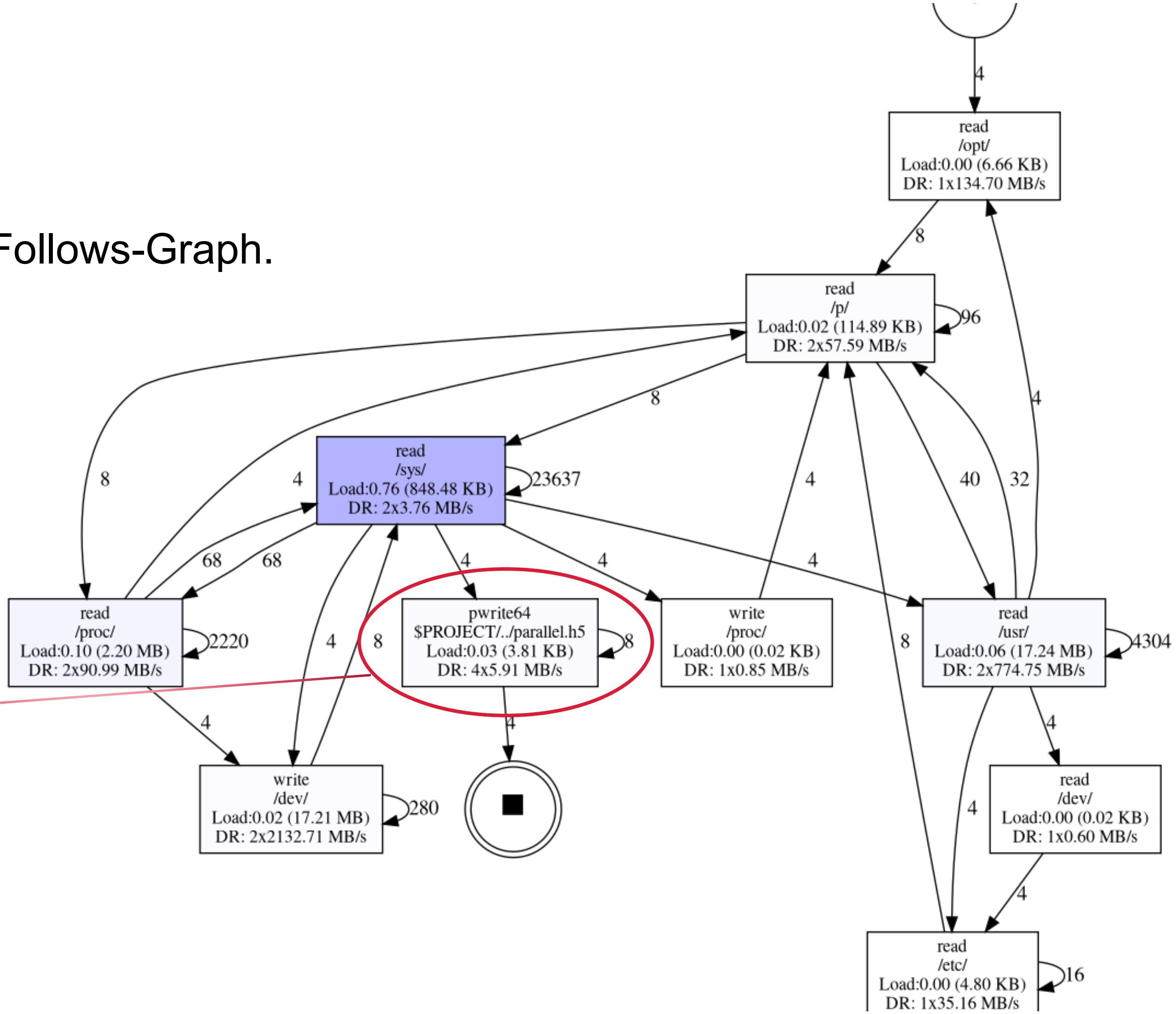
→ read+/usr/lib64

STrace Inspector

Consider Row split, MPI IO Collective

- I/O Operations represented as a Directly-Follows-Graph.
- Ref: https://arxiv.org/abs/2408.07378

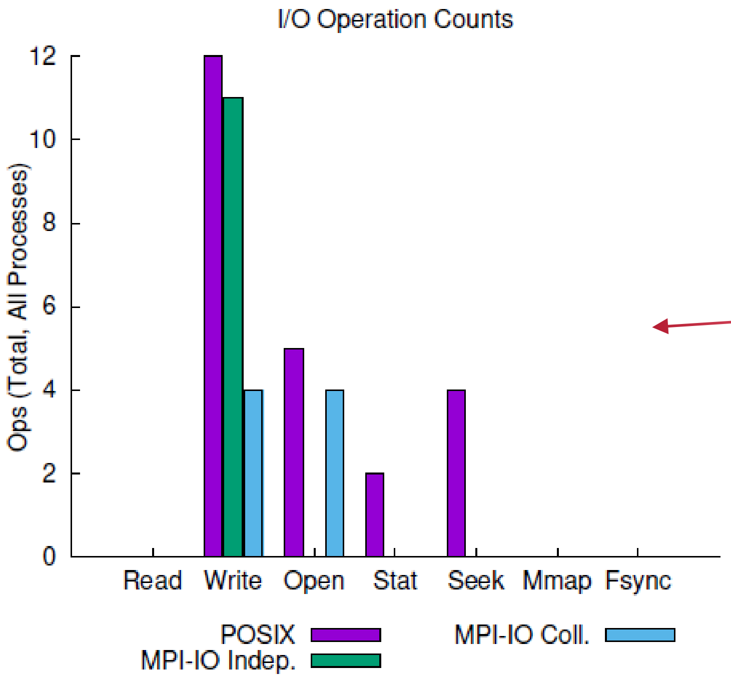


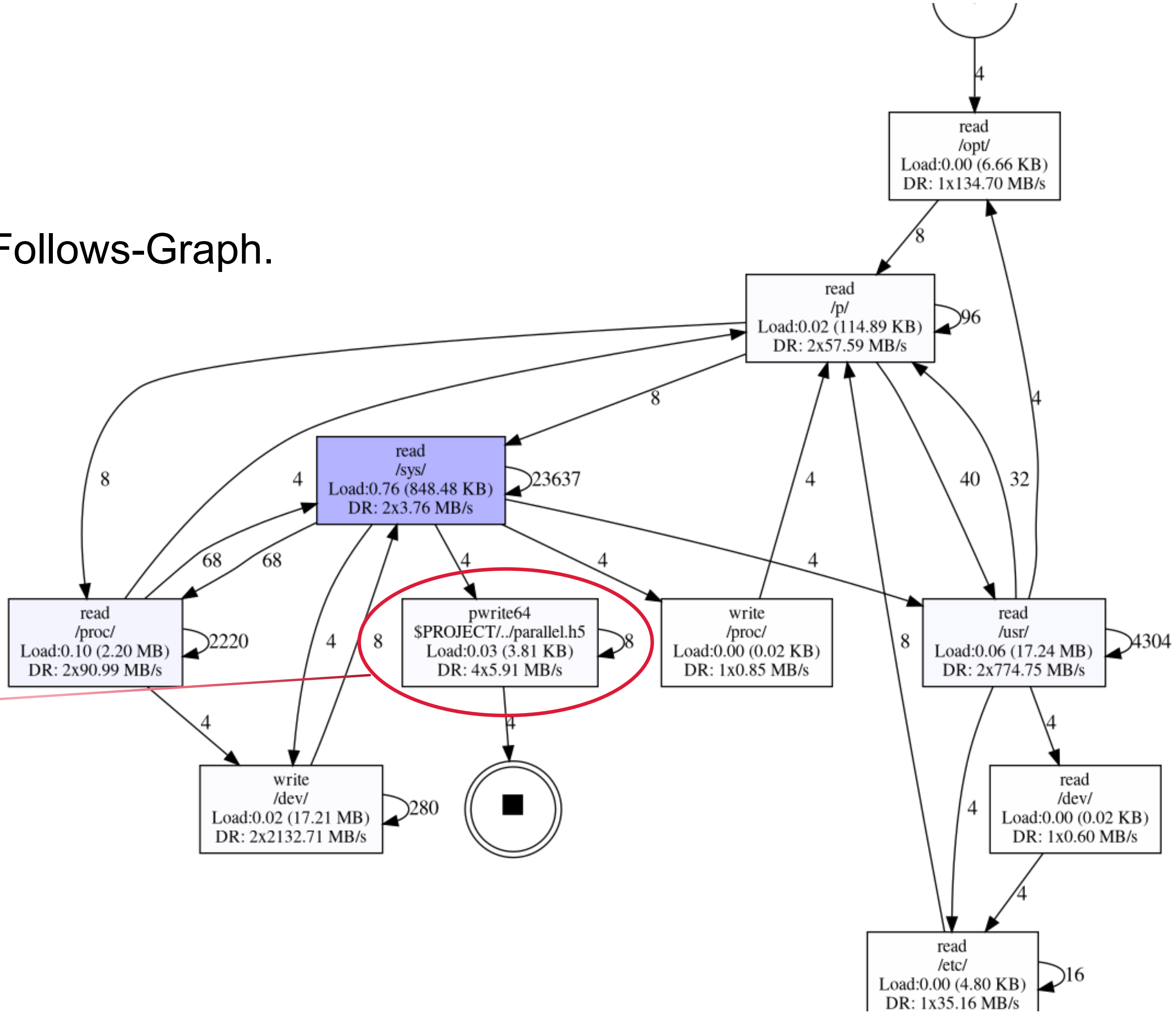


STrace Inspector

Consider Row split, MPI IO Collective

- I/O Operations represented as a Directly-Follows-Graph.
- Ref: https://arxiv.org/abs/2408.07378

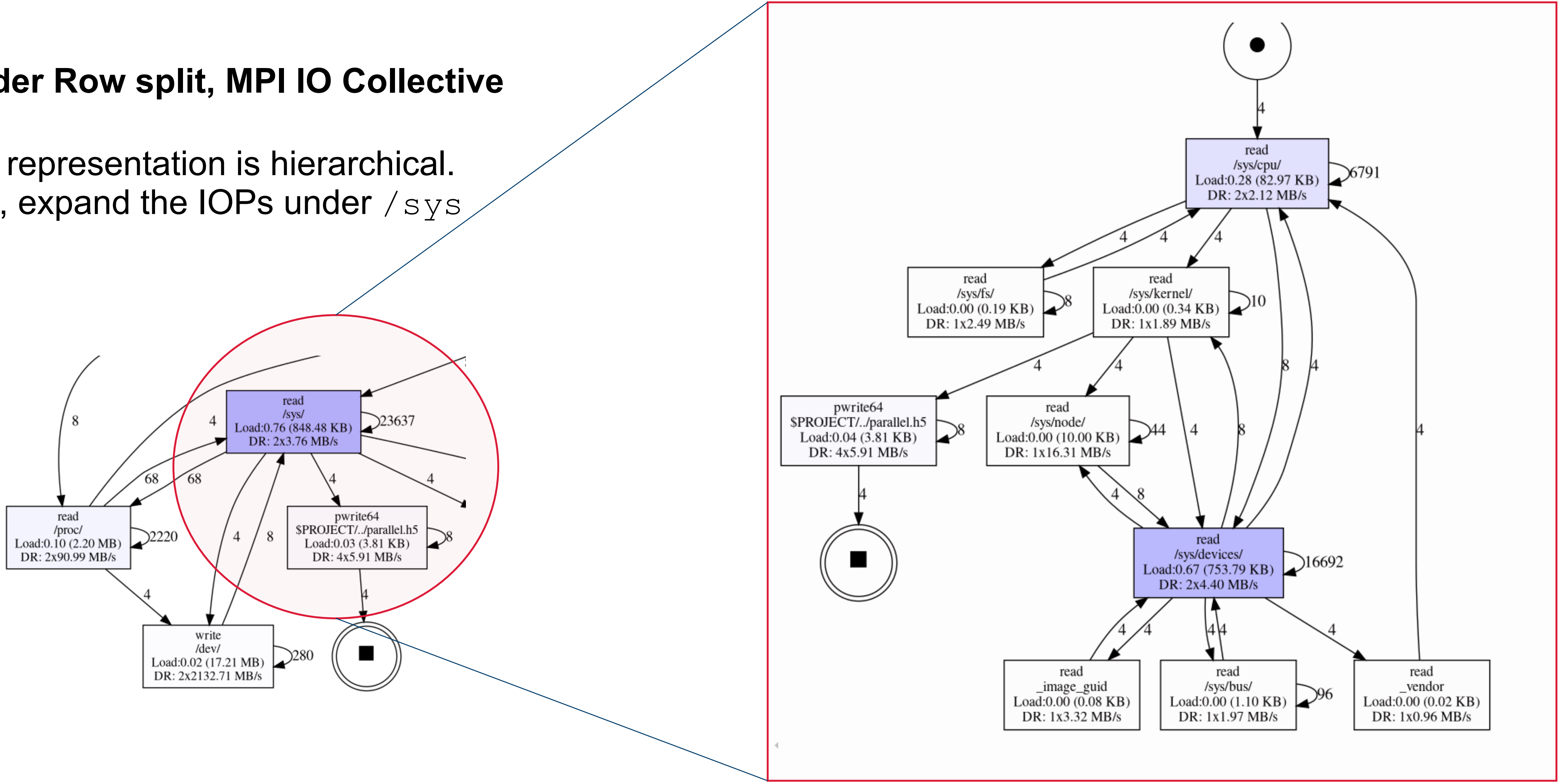




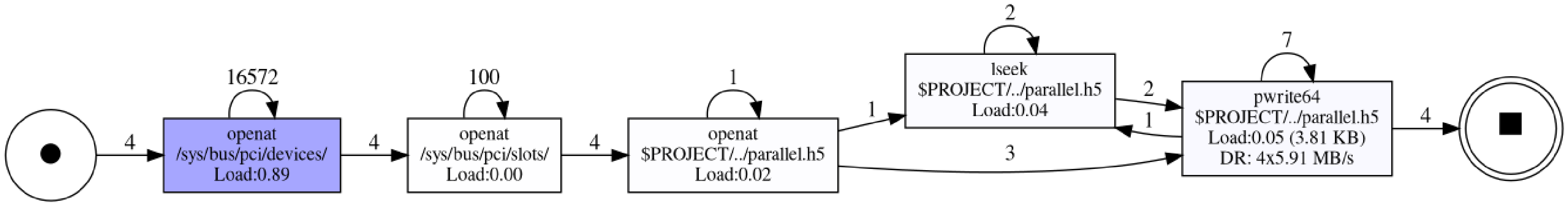
STrace Inspector

Consider Row split, MPI IO Collective

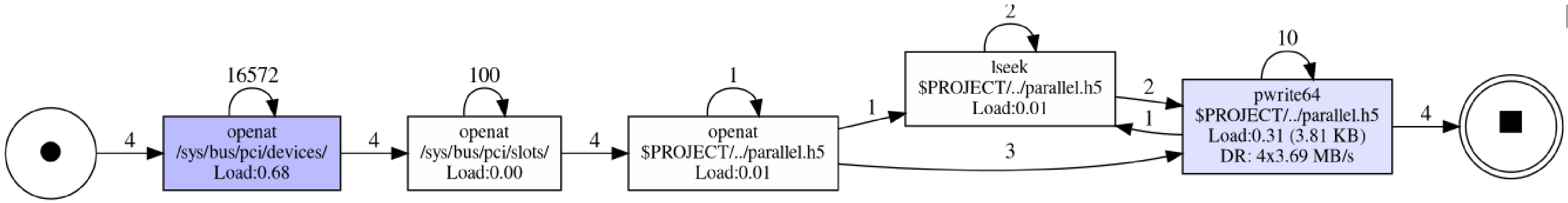
- The representation is hierarchical.
- E.g., expand the IOPs under `/sys`



STrace Inspector

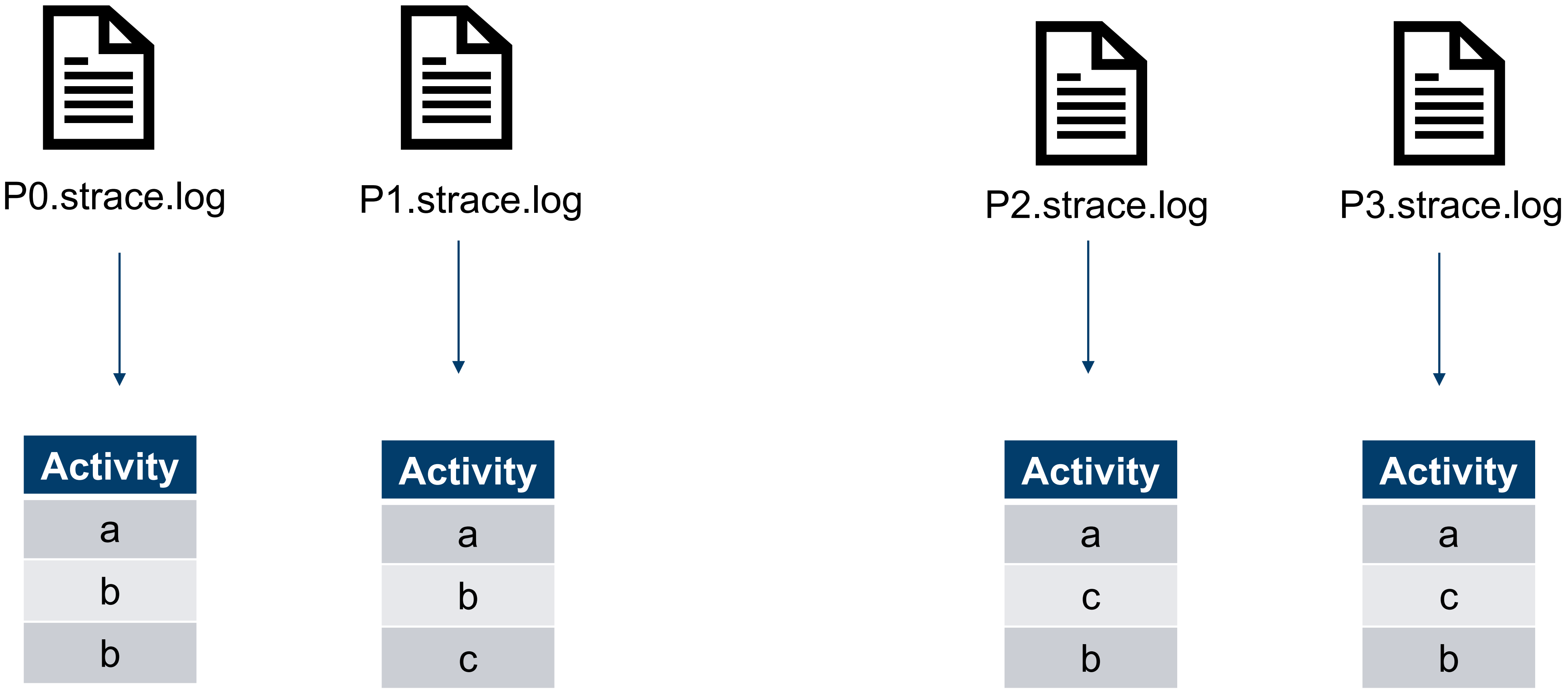


Row split, MPI IO Collective



Row split, MPI IO Independent

STrace Inspector: DFG Construction



P0.strace.log

Activity
a
b
b

P1.strace.log

Activity
a
b
c

P2.strace.log

Activity
a
c
b

P3.strace.log

Activity
a
c
b

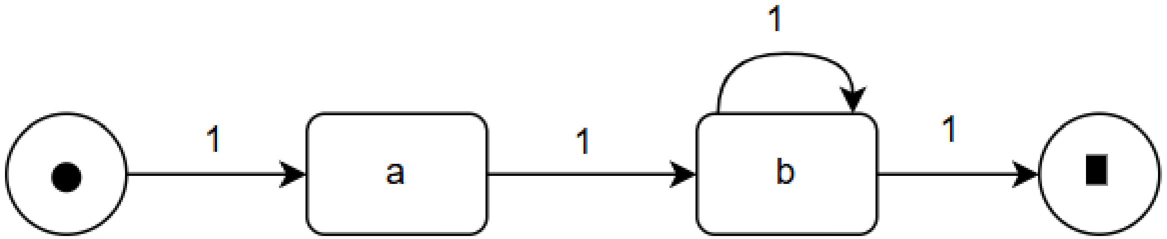
STrace Inspector: DFG Construction

Activity
a
b
b

Activity
a
b
c

Activity
a
c
b

Activity
a
c
b



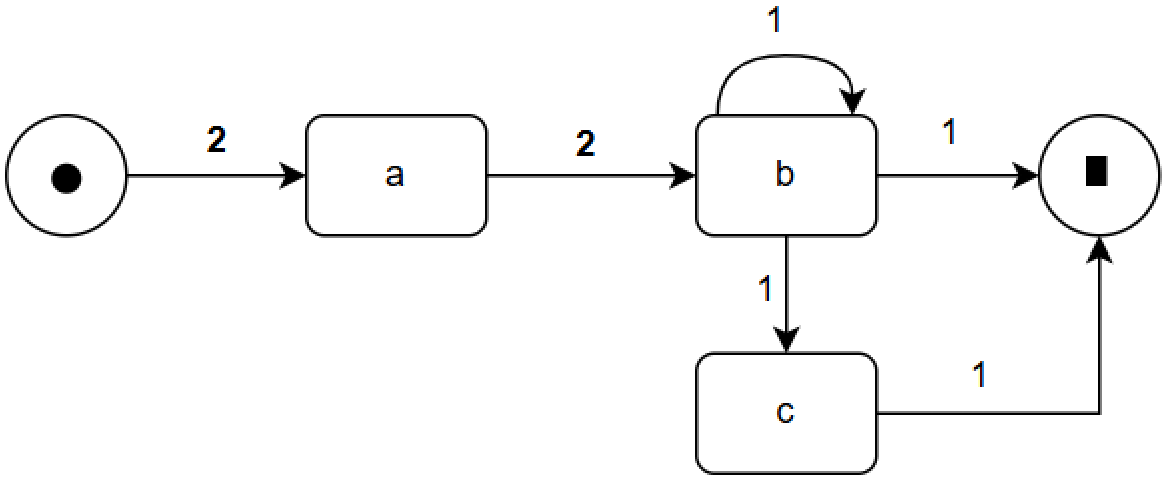
STrace Inspector: DFG Construction

Activity
a
b
b

Activity
a
b
c

Activity
a
c
b

Activity
a
c
b



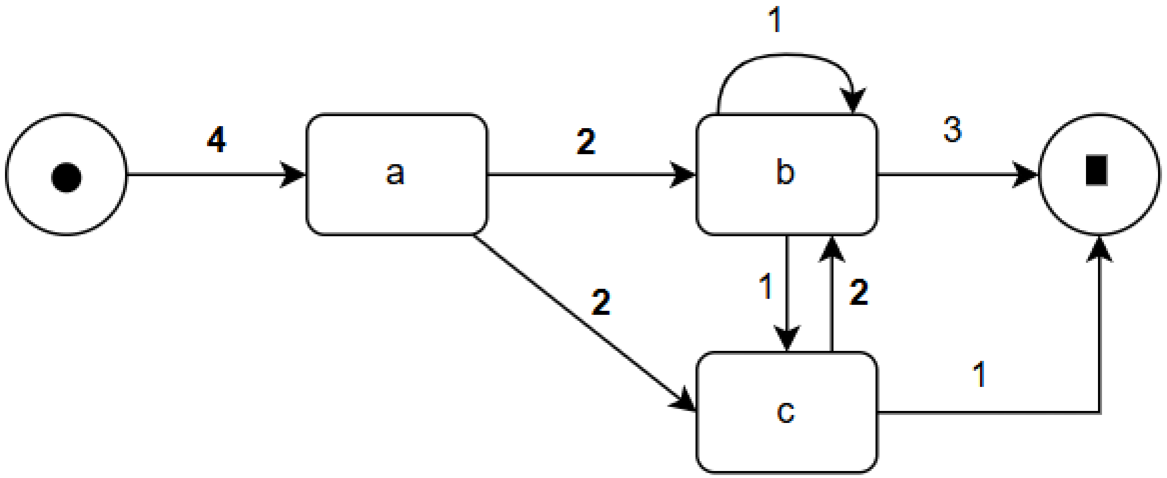
STrace Inspector: DFG Construction

Activity
a
b
b

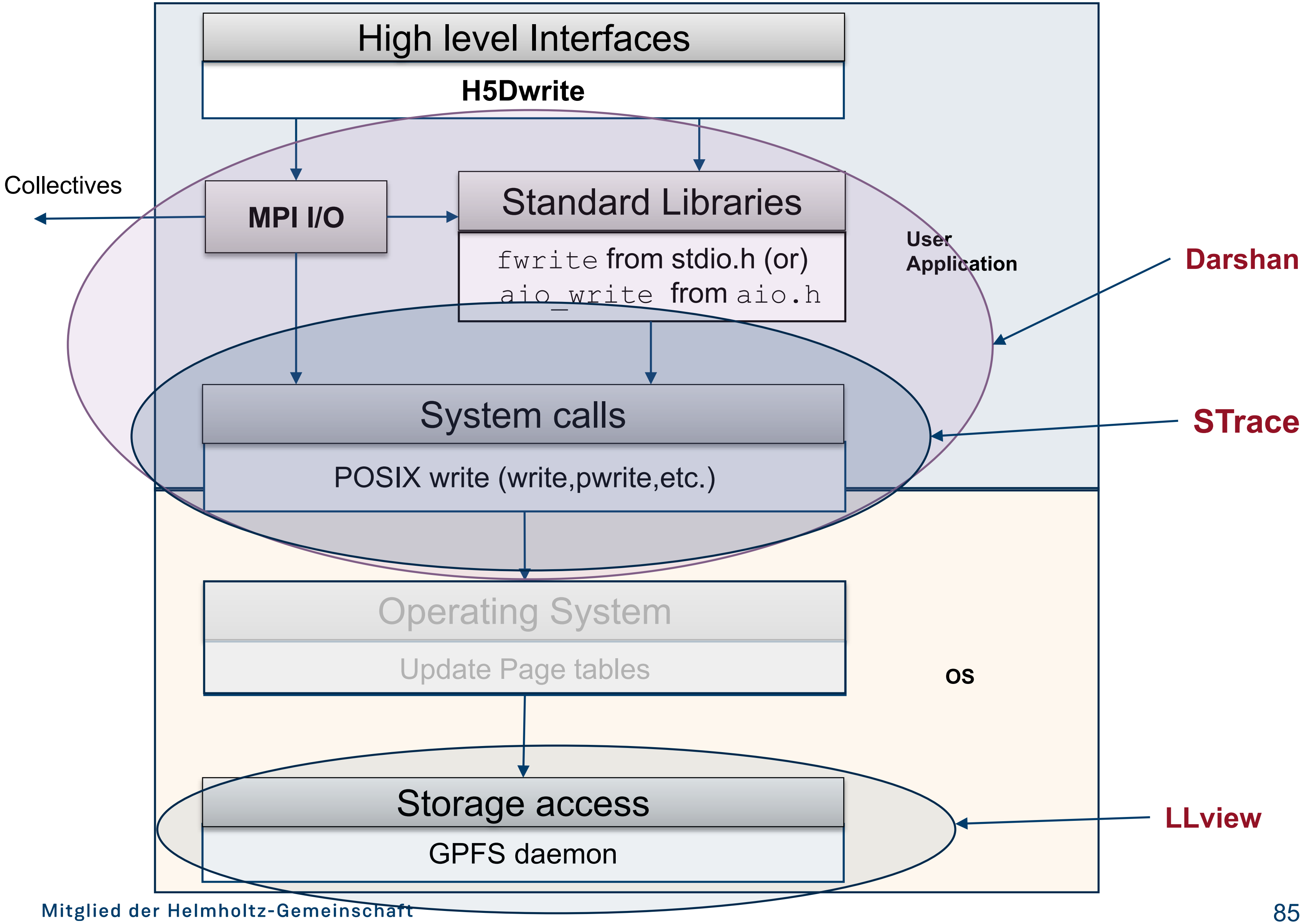
Activity
a
b
c

Activity
a
c
b

Activity
a
c
b

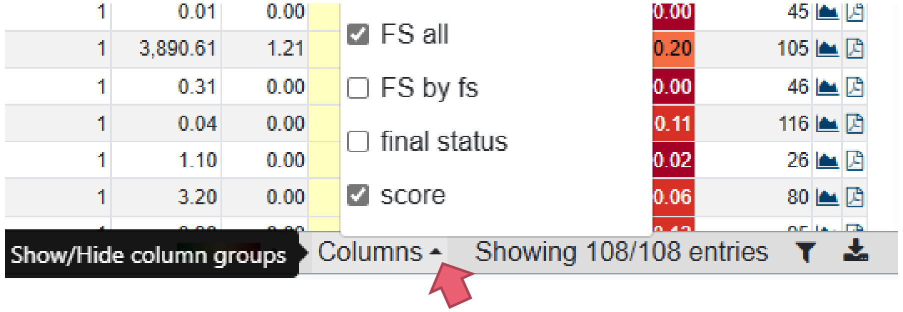


Tools in their Perspectives



Monitoring GPFS accesses with LLView

- Enable the view of File system I/O operations (FS all)



1	0.01	0.00	0.00	45
1	3,890.61	1.21	0.20	105
1	0.31	0.00	0.00	46
1	0.04	0.00	0.11	116
1	1.10	0.00	0.02	26
1	3.20	0.00	0.06	80

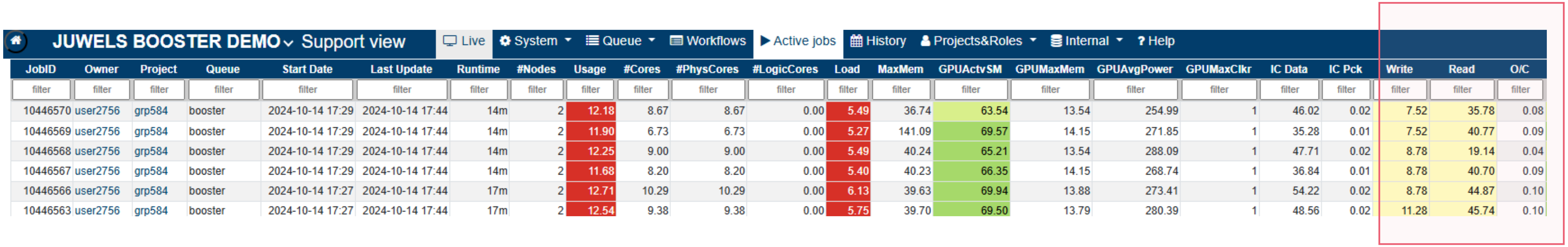
☑ FS all

☐ FS by fs

☐ final status

☑ score

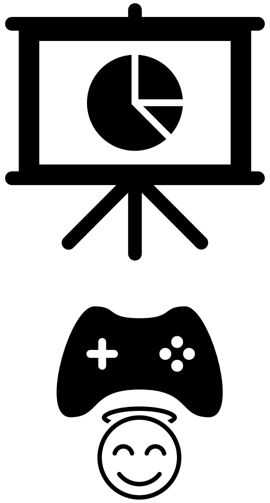
- The total GPFS read/write (in GiB) during the run time of the job are displayed.



JobID	Owner	Project	Queue	Start Date	Last Update	Runtime	#Nodes	Usage	#Cores	#PhysCores	#LogicCores	Load	MaxMem	GPUActvSM	GPUMaxMem	GPUAvgPower	GPUMaxClkr	IC Data	IC Pck	Write	Read	O/C
filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter	filter
10446570	user2756	grp584	booster	2024-10-14 17:29	2024-10-14 17:44	14m	2	12.18	8.67	8.67	0.00	5.49	36.74	63.54	13.54	254.99	1	46.02	0.02	7.52	35.78	0.08
10446569	user2756	grp584	booster	2024-10-14 17:29	2024-10-14 17:44	14m	2	11.90	6.73	6.73	0.00	5.27	141.09	69.57	14.15	271.85	1	35.28	0.01	7.52	40.77	0.09
10446568	user2756	grp584	booster	2024-10-14 17:29	2024-10-14 17:44	14m	2	12.25	9.00	9.00	0.00	5.49	40.24	65.21	13.54	288.09	1	47.71	0.02	8.78	19.14	0.04
10446567	user2756	grp584	booster	2024-10-14 17:29	2024-10-14 17:44	14m	2	11.68	8.20	8.20	0.00	5.40	40.23	66.35	14.15	268.74	1	36.84	0.01	8.78	40.70	0.09
10446566	user2756	grp584	booster	2024-10-14 17:27	2024-10-14 17:44	17m	2	12.71	10.29	10.29	0.00	6.13	39.63	69.94	13.88	273.41	1	54.22	0.02	8.78	44.87	0.10
10446563	user2756	grp584	booster	2024-10-14 17:27	2024-10-14 17:44	17m	2	12.54	9.38	9.38	0.00	5.75	39.70	69.50	13.79	280.39	1	48.56	0.02	11.28	45.74	0.10

Summary

Performance Analysis



Expectation

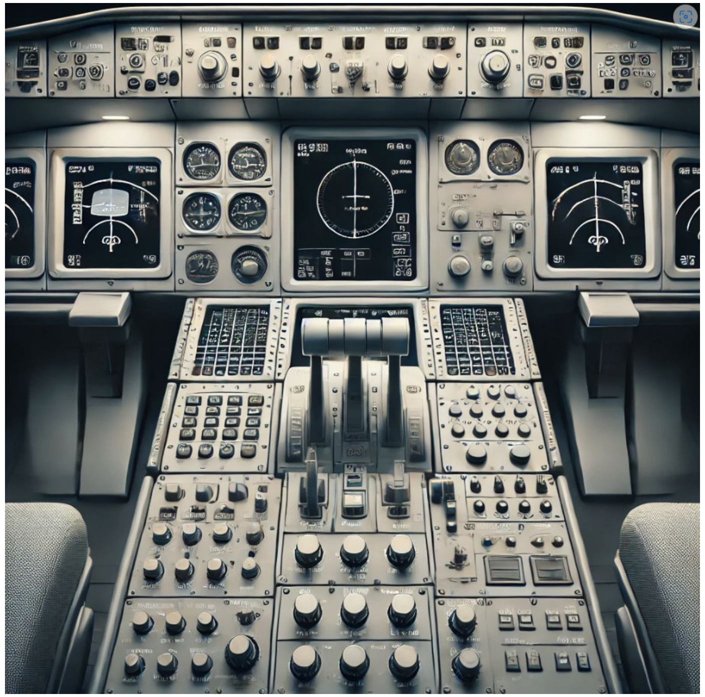
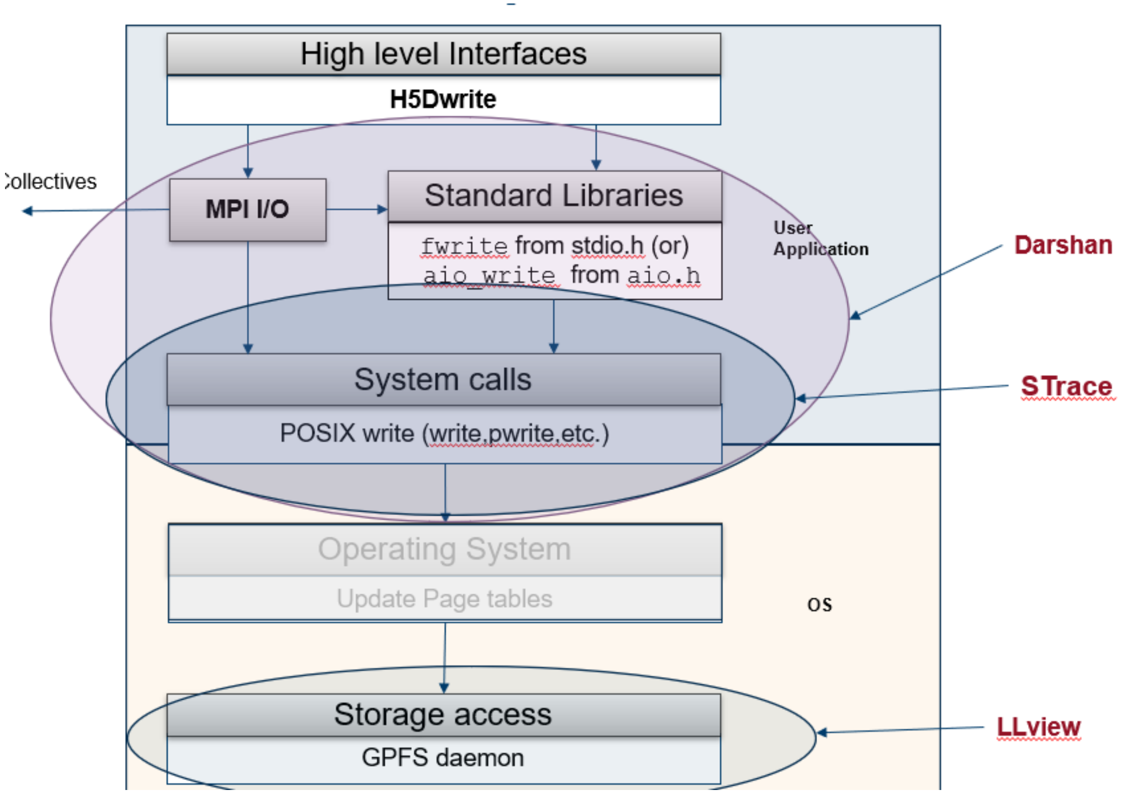


Image generated by OpenAI's DALL-E model

Reality

Summary

- LLView: To identify the stress due to file system activities.
- Darshan: For aggregated statistics on application I/O performance.
- STrace Inspector: For hierarchical analysis of application IOPs.


High level Interfaces
H5Dwrite
Collectives
MPI I/O
Standard Libraries
fwrite from stdio.h (or)
aio_write from aio.h
User Application
Darshan
System calls
POSIX write (write,pwrite,etc.)
STrace
Operating System
Update Page tables
OS
Storage access
GPFS daemon
LLview