IEEE CLUSTER 2025
Edinburgh, Scotland • 2–5 September

Tutorial: Identifying Parallel I/O Performance Bottleneck Using IOP Tool

Genshen Chu
University of Science and Technology Beijing

Hands-on Tutorial @ CLUSTER25

Outline

- Introduction & Background
- How IOP works
- Evaluation
- Hands-on Tutorial
  - Install and use its API
  - Case Study – Hello world
  - Case Study – Application: MISA-MD (MPI-IO)
  - Case Study – Application: phiflow

Outline

- Introduction & Background
- How IOP works
- Evaluation
- Hands-on Tutorial
  - Install and use its API
  - Case Study – Hello world
  - Case Study – Application: MISA-MD (MPI-IO)
  - Case Study – Application: phiflow

Background

Cases

MD — Material simulation

KMC — Simulation of long time evolution of defects

CFD — 10-bilion meshes — Reactor: 61 pins

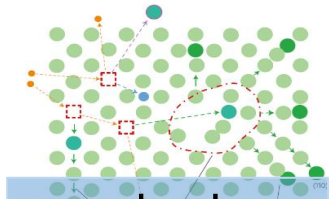
FEM — Structural mechanics computation

GPT-O4/Deepseek

Model parameters

Dataset

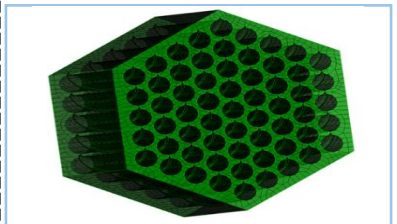
■Paticle-based Application



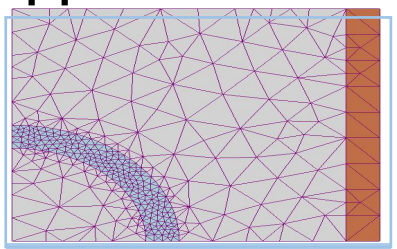
molecular dynamics(MD)

N-body

■Mesh-based applications



CFD



FEM

AI applications

LLM

LLM

I/O mount

10^{12} atoms
100B/atom
10000 steps
~ 100PB

CFD simulation:
~ 3.1TB

FEM:
~ 140GB

Train: ~PB
Inference: ~TB
Model load:
~GB

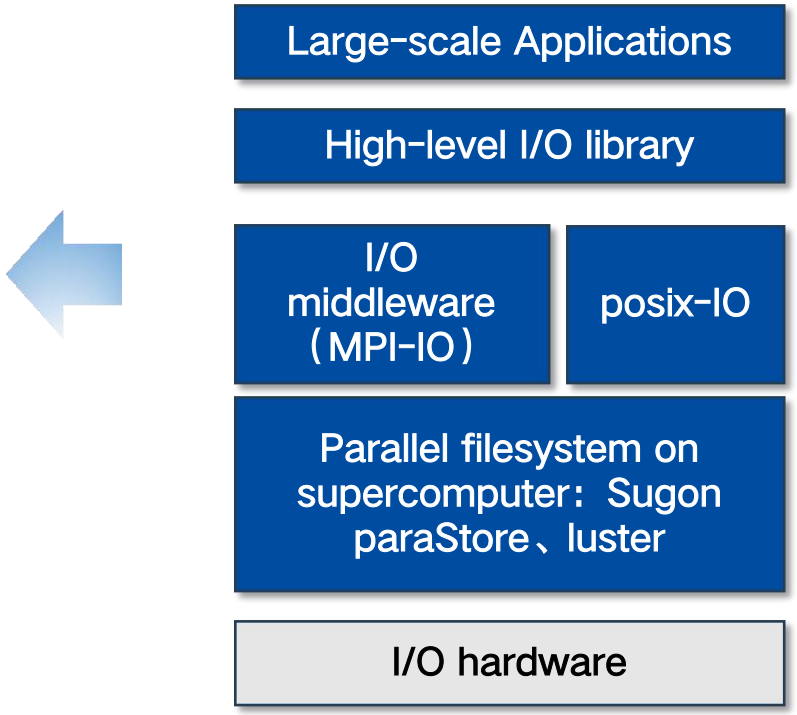
Many applications in engineering computing have a huge amount of i/o, and the application output usually reaches TB and PB levels.

Application performance testing of high-performance computing often focuses on computing performance and ignores I/O performance.

Background

The challenge of I/O performance analysis for massively parallel applications

- Existing I/O performance analysis tools do not support fine-grained I/O performance information collection for large-scale parallel programs.
- The existing I/O performance analysis and diagnosis tools are difficult to diagnose performance with high accuracy and low overhead in complex and large-scale parallel computing environments.



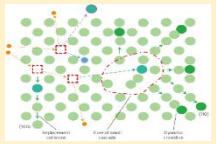
For Supercomputing system, a lightweight, fully functional, high accuracy, simple and easy-to-use tool set for online collection, analysis, diagnosis and optimization of I/O should be developed.

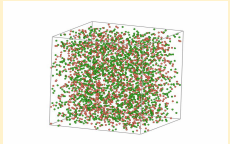
Outline

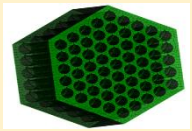
- Introduction & Background
- How IOP works
- Evaluation
- Hands-on Tutorial
  - Install and use its API
  - Case Study – Hello world
  - Case Study – Application: MISA-MD (MPI-IO)
  - Case Study – Application: phiflow

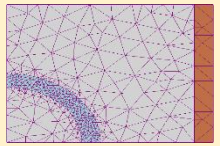
IOP tool: overview

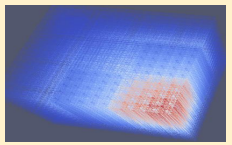
Applicat i-ons

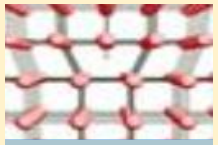

MD

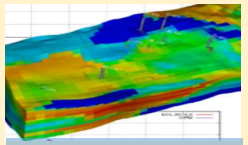

KMC


CFD


FEM

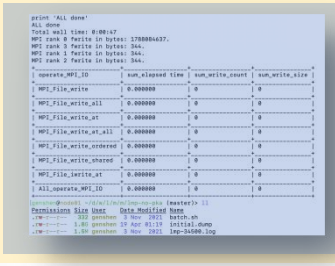

ANT-MOC


DDD

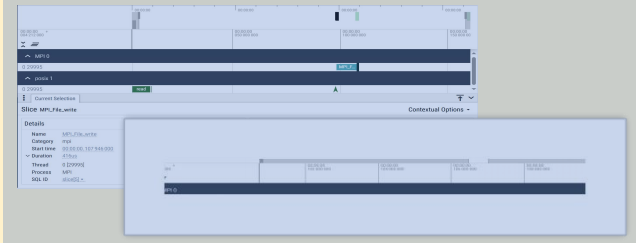

OPM

.....

Post-processing Tools



Performance data statistics table generation tool



Open Trace visualization format conversion tool

I/O Performance Analysis Tools

IOP v1.0: An I/O Performance Collection Tool for Domestic Large-Scale Parallel Programs

-Multi-level and fine-grained data collection
-Low overhead and high scalability
-Visualization and statistics table generation

I/O Performance Diagnosis Tool v1.0 Based on IOP Performance Data Format

-Load balancing performance issue diagnosis
-Scattered small I/O request identification
-Independent small I/O problem diagnosis

I/O Performance Prediction Tool v1.0 Based on Fine-grained I/O Performance Data

-I/O algorithm design and I/O performance prediction based on fine-grained I/O performance data records

Tool Functions

Construction of typical application datasets

I/O performance bottleneck diagnosis based on typical application datasets

I/O strategy recommendation for large-scale systems

Hardware Support

Sugon ParaStore Filesystem

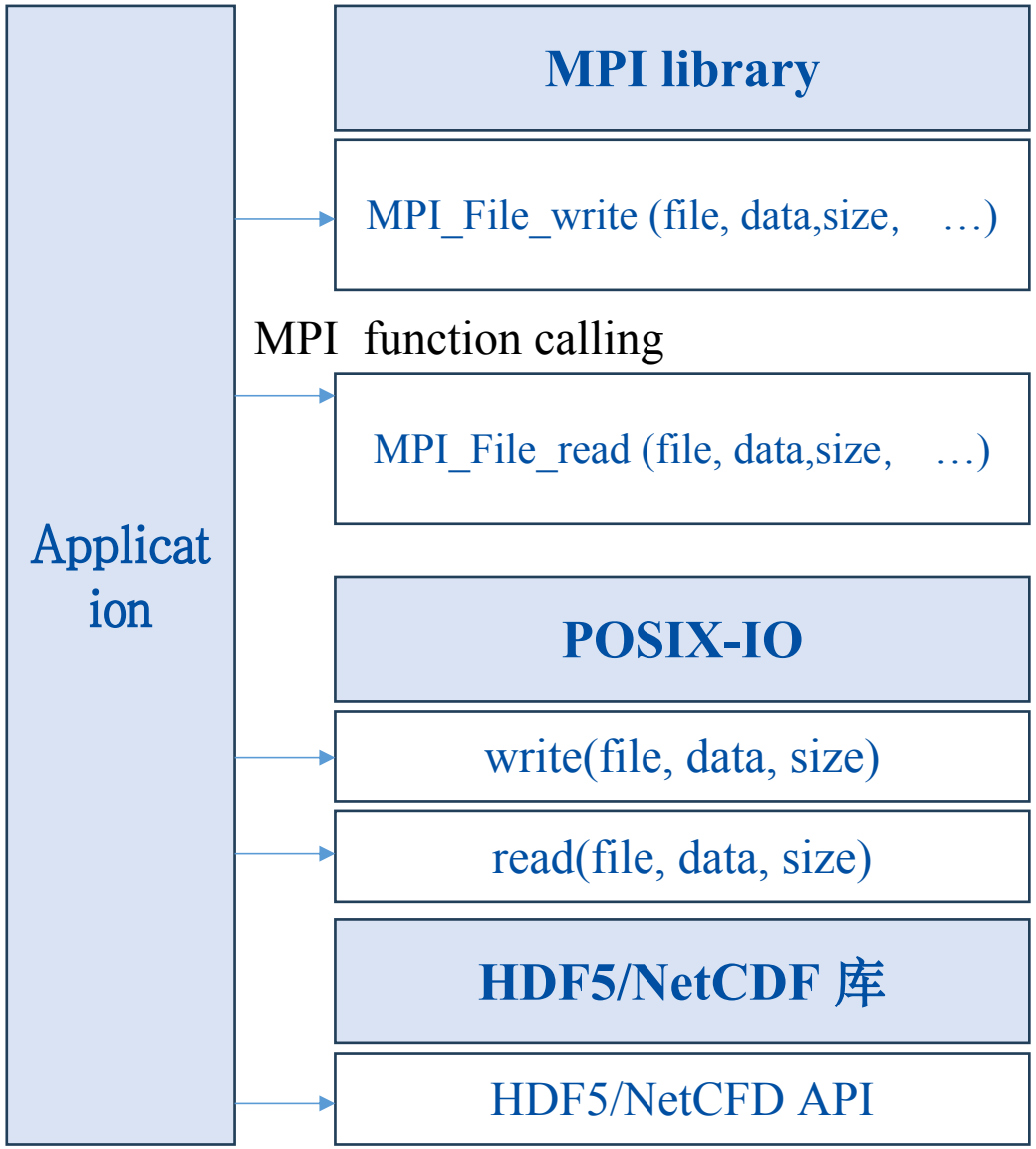
Tianhe Lustre Storage System

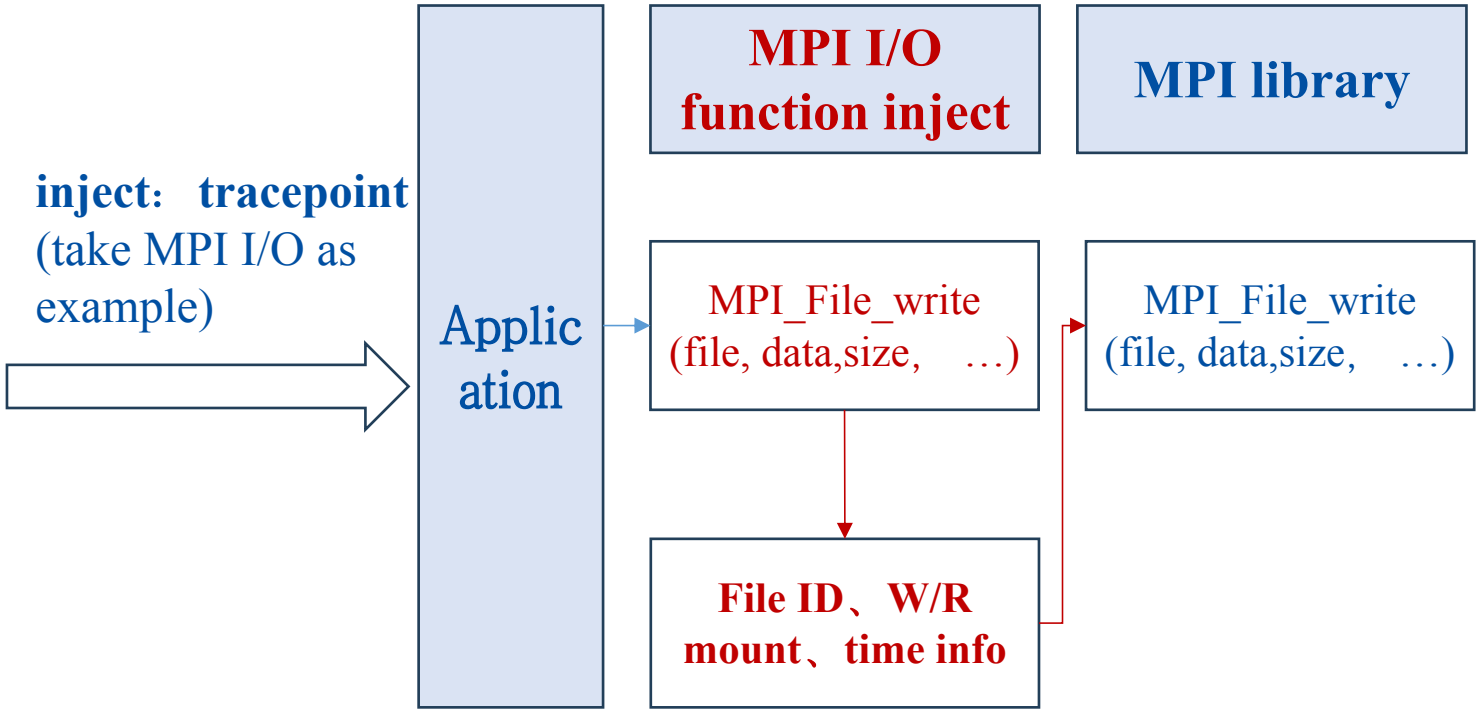
IntelImpi/MPICH Computing Libraries

.....

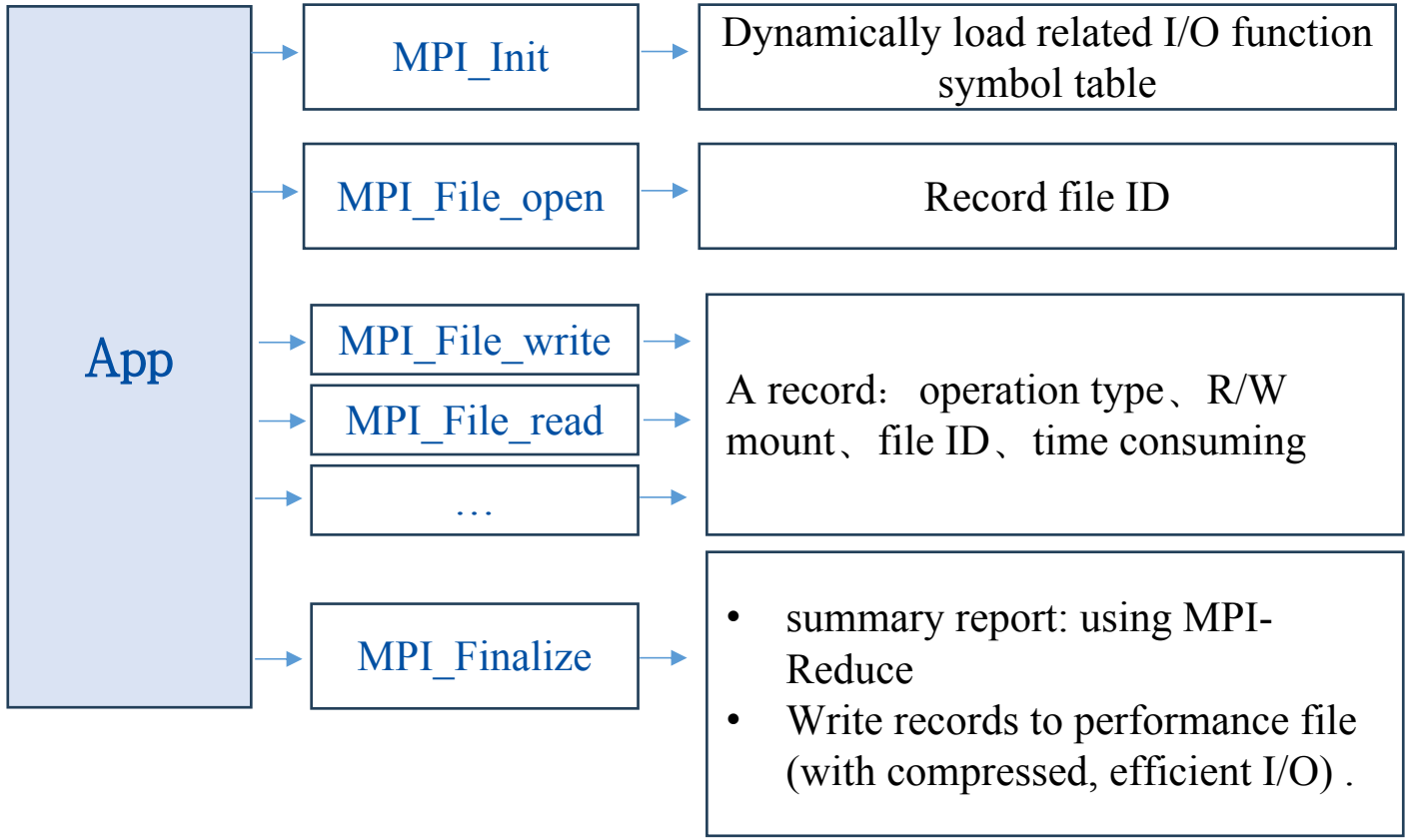
How IOP works

Principle: Linux dynamic library loading mechanism.
Advantages: 1) Only a small amount of overhead is involved in the I/O part, and other parts of the program are not affected.
2) Collect and analyze the performance of MPI-IO, parallel IO libra and POSIX I/O.



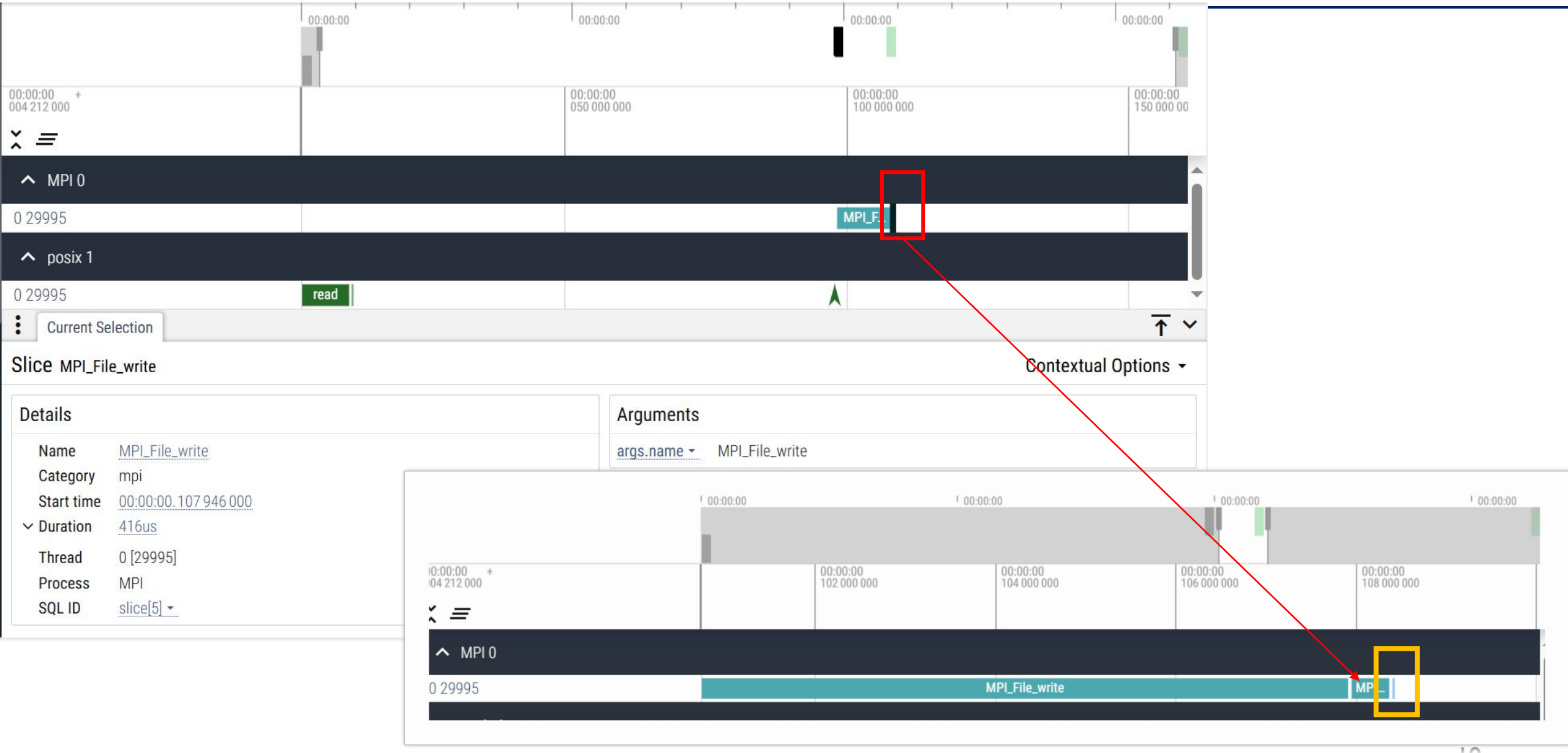


How IOP works: the workflow

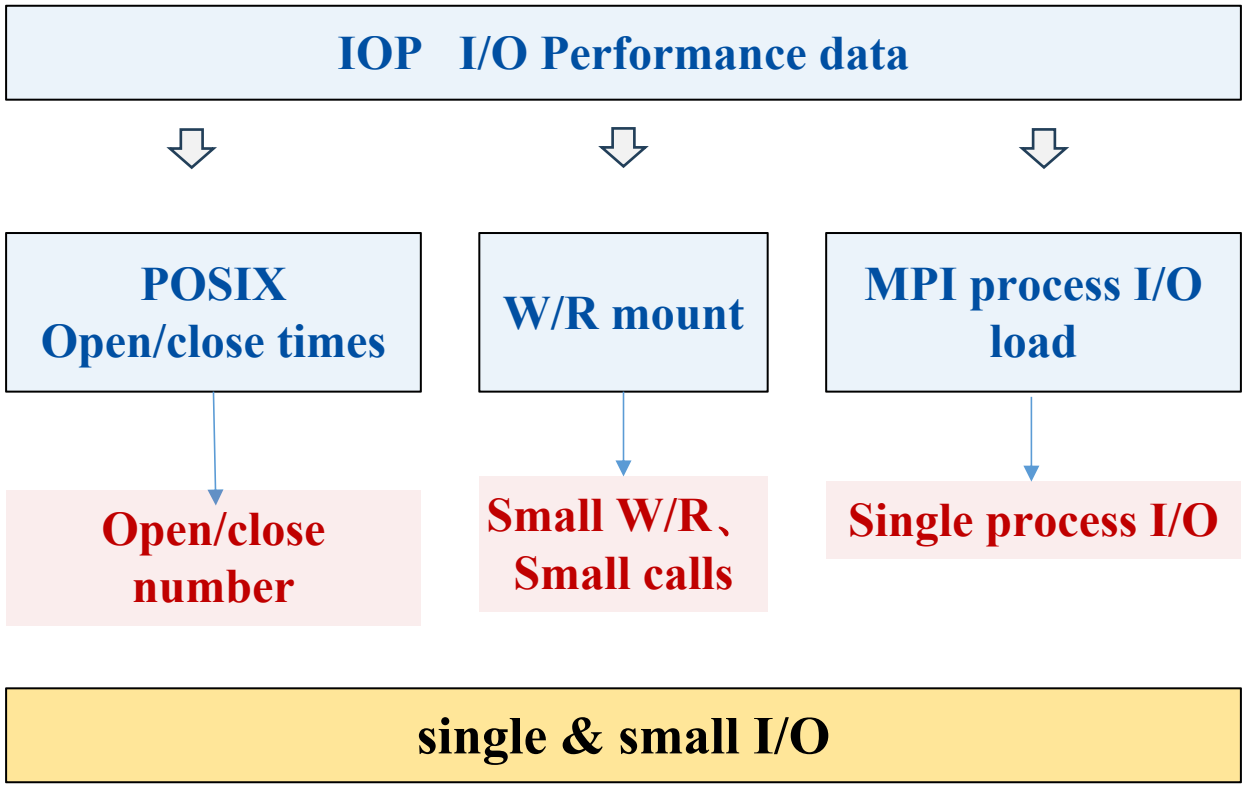


The workflow of IOP tool with Linux **dlsym**

How IOP works: Timeline visualization



How IOP works: the diagnosis of small file I/O



Outline

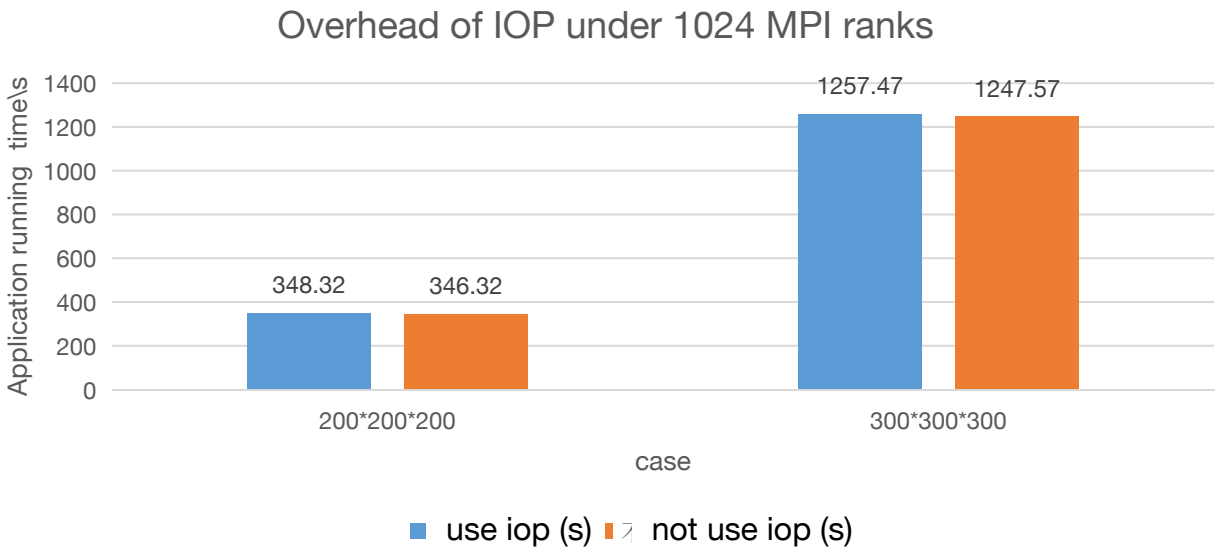
- Introduction & Background
- How IOP works
- Evaluation
- Hands-on Tutorial
  - Install and use its API
  - Case Study – Hello world
  - Case Study – Application: MISA-MD (MPI-IO)
  - Case Study – Application: phiflow

Experimental Setup

- **Server:**
  - 128 GB DDR4 memory & Hygon X86 CPU (32 cores)
  - Linux Centos 7
- **Representative programs**
  - **Parallel filesystem:** Sugon parStore
- **Compiler:** GCC 7.3.1, OpenMPI 4.0

IOP tool: overhead

IOP overhead evaluation



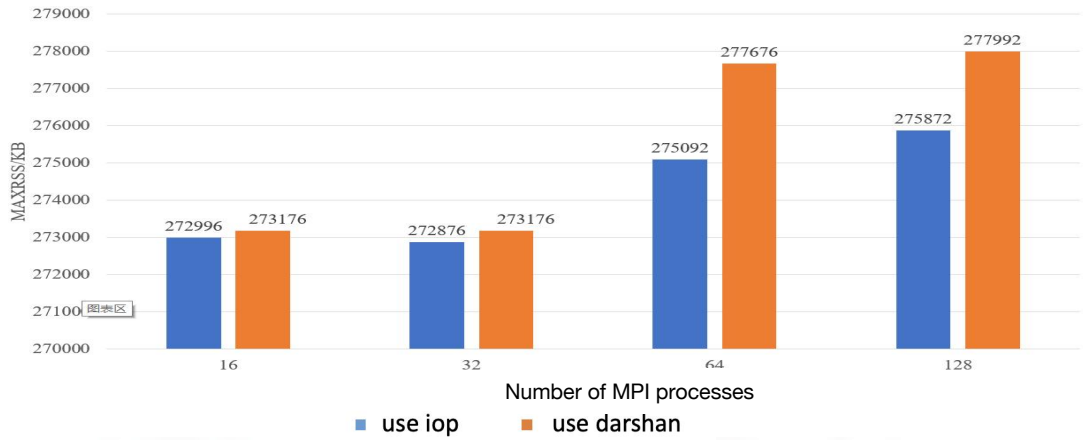
Small scaling: collection+ analysis overhead < 5%.

MISA-MD application	raw app			Overhead of IOP+app			
Case:	MPI ranks	Total running time	I/O time	MPI ranks	Total running time	I/O time	**IOP overhead percentage**
200*200*200	1024	**346.32**	343.39	1024	**348.32**	344.82	**0.50%**
300*300*300	1024	**1247.57**	1239.05	1024	**1257.47**	1246.91	**0.80%**

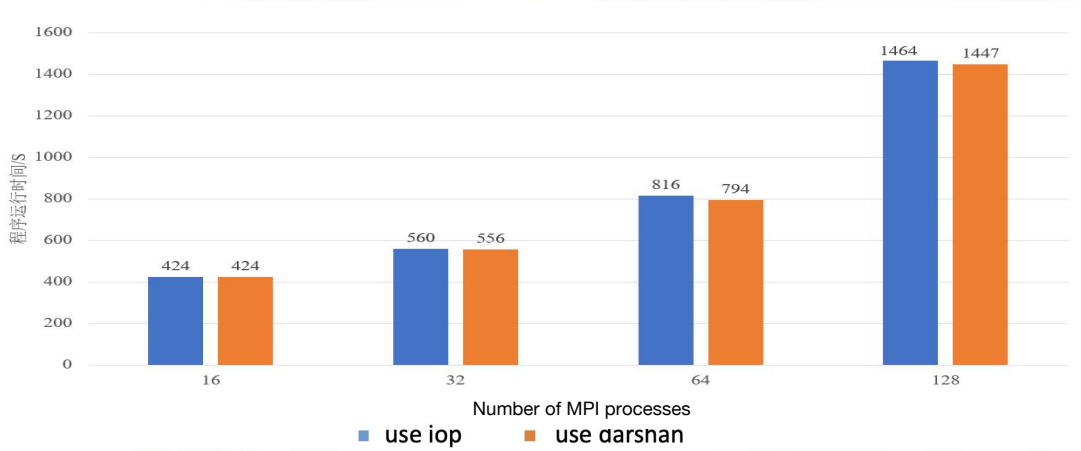
14

IOP overhead（VS darshan）

Memory usage



Time overhead



IOP tool: comparison

tool	IOP	Darshan	IO-PIN	Recorder
purpose	I/O performance analyzing for HPC application	I/O characterization and performance analyzing	Dynamic binary instrumentation analyzing	System-level performance data logging
Supported I/O interface	MPI-IO, Parallel FS, POSIX-IO, High-level I/O lib	MPI-IO, Parallel FS, POSIX-IO, High-level I/O lib	MPI-IO, Parallel FS, POSIX-IO, High-level I/O lib	MPI-IO, Parallel FS, POSIX-IO, High-level I/O lib
How to collect I/O performance	Dynamically capture key performance indicators	Capture I/O access patterns and performance metrics	Use runtime instrumentation for collecting detailed information	Record system calls and kernel events
Trace position	Multi-level **fine-grained (function call level)** insertion	Dynamically insert code probes when a file is opened	Dynamically insert code probes based on user requirements	System-level instrumentation without modifying the source code
Performance data	• file opening, writing, and closing; • **Fine-grained I/O details**: per-I/O function call data (e.g., MPI read/write bytes and rank info, operation name, I/O timestamp, file descriptor); • Statistical analysis of all I/O information.	• **Coarse-grained I/O statistical performance**: I/O operation count, I/O bytes, I/O latency, I/O bandwidth, file size • File access patterns	• Function call counts, duration, and memory access patterns; • Per-I/O-call details: I/O latency at software stack layers, disk access counts, throughput, and I/O calls to PVFS servers • **Constrained by specific file systems**	• System call counts, duration, and resource consumption; • Detailed function trace information (including parameters) for HDF5, MPI, and POSIX I/O calls

For more details, please refer to our source code and online docs.

IOP is open-source: https://github.com/hpcde/IOP

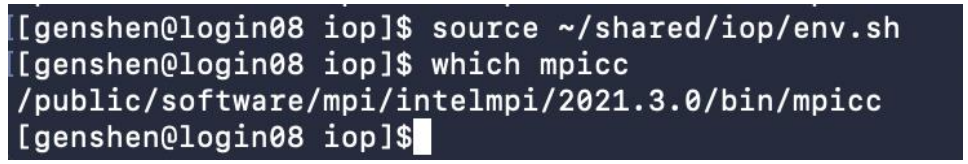
Outline

- Introduction & Background
- How IOP works
- Evaluation
- Hands-on Tutorial
  - Install and use its API
  - Case Study – Hello world: mpi_trunk_wirte
  - Case Study – Application: MISA-MD (MPI-IO)
  - Case Study – Application: phiflow CFD

Installation – IOP (optional)

- **Prerequisites for Compilation:**
  - environment: C++ 11 ,MPI standard 2.0+, cmake >3.19.1.
    - To load the above tools, just:

```
source ~/shared/iop/env.sh
which mpicc
```



  - IOP Source Code (optional): git clone https://github.com/hpcde/iop.git
  - Dependencies manger tool **pkg**(optional): download from https://github.com/genshen/pkg/releases

Installation – IOP (optional)

- Prepare:

```
mkdir my_iop_work_dir/
cd my_iop_work_dir/
export IOP_PATH=$(pwd)
```

- Compilation and Install :
  - Get Source Code:

```
cd $IOP_PATH && cp -r ~/shared/iop/iop/code/ ./iop_code
cd iop_code
```

  - Install dependencies:

```
pkg fetch && pkg install
```

  - Compilation and install:

```
cmake -B ./build -S ./
cmake --build ./build -j 2
cmake --install ./build --prefix=$IOP_PATH/iop_install
```

- Check IOP Installation:
  - libiop_hook.so:

```
ls -alh $IOP_PATH/iop_install/lib/libiop_hook.so
```

```
[genshen@login08 iop_code]$ls $IOP_PATH/iop_install/lib/libiop_hook.so
/public/home/wangxg/25-9-20-can-rm/my_iop_work_dir/iop_install/lib/libiop_hook.so
[genshen@login08 iop_code]$
```

Or just **copy pre-built iop binary** b

```
cp -r ~/shared/iop/iop_pre-built/  $IOP_PATH/iop_install
```

How to Use IOP

- **Assumption:** IOP has been successfully
- **.so building path:** $IOP_*PATH/iop_install/lib/libiop*_hook.so
- **Usage:**
- Runtime injection via `LD_PRELOAD` (recommended)

```
export LD_PRELOAD=$IOP_PATH/iop_install/lib/iop_hook.so.
```

- **Or:** link during compilation if you are using CMake

```
target_link_libraries(YOUR_TARGET
                PRIVATE {PATH/TO/libiop_hook.so}
)
```

Linking Order Requirement: If other shared libraries are being linked, IOP must be
placed at the top (outermost) layer during linking.

Case Study – hello world: mpi_trunk_write

- Testing the Built-in Test Programs of IOP

Run code path: `cd $IOP_PATH/iop_code/tests/mpi-io-app`

Compile and run:

compile: mpicxx -o mpi_trunk_write mpi_trunk_write.cc -lmpi

run:
LD_PRELOAD=$IOP_PATH/iop_install/lib/libiop_hook.so ./mpi_trunk_write

Case Study – hello world: mpi_trunk_write

- Testing the Built-in Test Programs of IOP

Run code path: `cd $IOP_PATH/iop_code/tests/mpi-io-app`

Compile and run:

compile: mpicxx -o mpi_trunk_write mpi_trunk_write.cc -lmpi

run:

LD_PRELOAD=$IOP_PATH/iop_install/lib/libiop_hook.so ./mpi_trunk_write

Run with multiple mpi processes:

vi run.sh
chmod +x run.sh
mpirun -n 4 ./run.sh

```
#!/bin/bash

# Note: you may set $IOP_PATH or change LD_PRELOAD here.
export LD_PRELOAD=$IOP_PATH/iop_install/lib/libiop_hook.so

./mpi_trunk_write
```

Case Study – mpi_trunk_write

- After the program execution completes, an I/O Operations and Communication Statistics Summary Table will be printed to the console.

Communication Function	Count	Total Size (bytes)	Total Time (s)	Avg Time (s)
MPI_Send	0	0	0.000000	0.000000
MPI_Recv	0	0	0.000000	0.000000
MPI_Isend	0	0	0.000000	0.000000
MPI_Irecv	0	0	0.000000	0.000000
MPI_Bcast	0	0	0.000000	0.000000
MPI_Reduce	0	0	0.000000	0.000000
MPI_Allreduce	0	0	0.000000	0.000000
MPI_Gather	0	0	0.000000	0.000000
MPI_Scatter	0	0	0.000000	0.000000

I/O Operations Summary Table

Communication Function	Count	Total Size (bytes)	Total Time (s)	Avg Time (s)
MPI_Send	0	0	0.000000	0.000000
MPI_Recv	0	0	0.000000	0.000000
MPI_Isend	0	0	0.000000	0.000000
MPI_Irecv	0	0	0.000000	0.000000
MPI_Bcast	29	482440	9.349668	0.322402
MPI_Reduce	0	0	0.000000	0.000000
MPI_Allreduce	5	64	0.336348	0.067270
MPI_Gather	0	0	0.000000	0.000000
MPI_Scatter	0	0	0.000000	0.000000

I/O Communication Summary Table

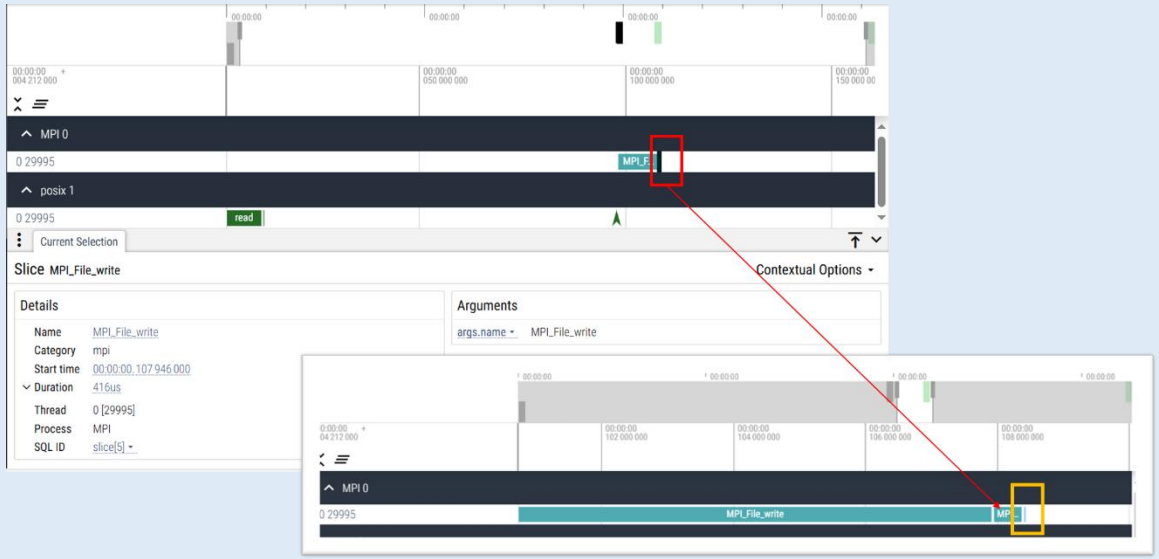
Case Study – mpi_trunk_write

IOP run will generate binary performance data file **mpi_iomsg.dat** and **posix_iomsg.dat**, which contain I/O call information for MPI-IO and POSIX-IO, respectively (in compressed storage format)

offset 0 absolute relative

address	00 01 02 03 04 05 06 07	08 09 10 11 12 13 14 15	Ascll	unsig
00000000	1d 01 00 00 ff ff ff ff	9b 58 e7 67 00 00 00 00	▯.....X.g....	
00000010	28 93 09 00 00 00 00 00	9b 58 e7 67 00 00 00 00	(.....X.g....	
00000020	33 93 09 00 00 00 00 00	3c 00 00 00 00 00 00 00	3.....<.....	
00000030	00 00 00 00 00 00 00 00	1b 01 00 00 ff ff ff ff	.....▯.....	
00000040	9b 58 e7 67 00 00 00 00	63 93 09 00 00 00 00 00	.X.g....c.....	
00000050	9b 58 e7 67 00 00 00 00	64 93 09 00 00 00 00 00	.X.g....d.....	
00000060	20 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
00000070	1a 01 00 00 ff ff ff ff	9b 58 e7 67 00 00 00 00	▯.....X.g....	
00000080	68 93 09 00 00 00 00 00	9b 58 e7 67 00 00 00 00	h.....X.g....	
00000090	6a 93 09 00 00 00 00 00	50 00 00 00 00 00 00 00	j.....P.....	
000000a0	00 00 00 00 00 00 00 00	1d 01 00 00 ff ff ff ff	.....▯.....	
000000b0	9b 58 e7 67 00 00 00 00	53 b5 09 00 00 00 00 00	.X.g....S.....	
000000c0	9b 58 e7 67 00 00 00 00	55 b5 09 00 00 00 00 00	.X.g....U.....	
000000d0	3c 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	<.....	
000000e0	1b 01 00 00 ff ff ff ff	9b 58 e7 67 00 00 00 00	▯.....X.g....	
000000f0	65 b5 09 00 00 00 00 00	9b 58 e7 67 00 00 00 00	e.....X.g....	
00000100	66 b5 09 00 00 00 00 00	20 00 00 00 00 00 00 00	f.....	
00000110	00 00 00 00 00 00 00 00	1a 01 00 00 ff ff ff ff	.....▯.....	

Compressed storage performance logs



Visualization Outputs

- Through post-processing scripts, it can be converted into visualization format.

```
cd $IOP_PATH && cp -r ~/shared/iop/iop/generate_iotimeline ./ && cd generate_iotimeline
./iop_timeline_generator -input $IOP_PATH/iop_code/tests/mpi-io-app/mpi_iomsg.dat
```

- [skip] Download the converted .json file to local via sftp/scp, and open it in https://ui.perfetto.dev

Case Study – Application: MISA-MD (MPI-IO)

- MISA-MD Build & Deployment Tutorial (optional): https://github.com/misa-md/MISA-MD
  - MISA-MD is Molecular dynamics software for material simulation. It successfully simulated 30 trillion atoms on Sunway supercomputer.
- Script of running MISA-MD with IOP: copy **pre-built MISA-MD** binary and run batch.sh

```
cd $IOP_PATH
cp -r  ~/shared/iop/MISA-MD/ ./
cd  MISA-MD/example/
sbatch batch.sh
```

```
#!/bin/bash

#SBATCH –J md–48
#SBATCH –o md–48–%J.log
#SBATCH –e md–48–%J.err
#SBATCH –n 48
#SBATCH –c 1
#SBATCH  ––exclusive

module purge
module load compiler/devtoolset/7.3.1 mpi/intelmpi/2021.3.0

export LD_PRELOAD=$IOP_PATH/iop_install/lib/libiop_hook.so

mpirun –n 48 ../build/bin/misamd –c config.yaml
```

Num of MPI processes

Exclusive mode(Avoid resource competition)

Loading environment

Runtime injection via `LD_PRELOAD` to load iop

Case Study – Application: MISA-MD (MPI-IO)

Continued:

- Script of running MISA-MD with IOP: copy **pre-built MISA-MD** binary and run batch.sh

```
[genshen@login08 example]$ sbatch batch.sh
Submitted batch job 16013937
[genshen@login08 example]$ ll
total 2237
-rw-r--r-- 1 genshen genshen 1215317 Aug 23 00:38 FeCuNi.eam.alloy
-rw-r--r-- 1 genshen genshen    1111 Aug 23 00:38 batch.2.sh
-rw-r--r-- 1 genshen genshen     311 Aug 23 00:39 batch.sh
-rw-rw-r-- 1 genshen genshen     112 Aug 23 00:54 bump.10.out
-rw-rw-r-- 1 genshen genshen     112 Aug 23 00:54 bump.11.out
-rw-rw-r-- 1 genshen genshen     112 Aug 23 00:54 bump.12.out
-rw-rw-r-- 1 genshen genshen     112 Aug 23 00:54 bump.5.out
-rw-rw-r-- 1 genshen genshen     112 Aug 23 00:54 bump.6.out
-rw-rw-r-- 1 genshen genshen     112 Aug 23 00:54 bump.7.out
-rw-rw-r-- 1 genshen genshen     112 Aug 23 00:54 bump.8.out
-rw-rw-r-- 1 genshen genshen     112 Aug 23 00:54 bump.9.out
-rw-r--r-- 1 genshen genshen    3643 Aug 23 00:38 config.yaml
-rw-rw-r-- 1 genshen genshen    2754 Aug 23 00:54 md-48-16013937.err
-rw-rw-r-- 1 genshen genshen   24987 Aug 23 00:54 md-48-16013937.log
-rw-rw-r-- 1 genshen genshen     112 Aug 23 00:54 misa_mdl.2.out
-rw-rw-r-- 1 genshen genshen     112 Aug 23 00:54 misa_mdl.4.out
-rw-rw-r-- 1 genshen genshen 1014496 Aug 23 00:54 mpi_iomsg.dat
-rw-rw-r-- 1 genshen genshen   20104 Aug 23 00:54 posix_iomsg.dat
[genshen@login08 example]$ vi md-48-16013937.err
[genshen@login08 example]$ vi md-48-16013937.log
```

Input file: potential file

Slurm script

Output files of MISA-MD

Slurm std error and std output file

Output files of MISA-MD

Performance record data of IOP

Convert to .json timeline format:

```
$IOP_PATH/generate_iotimeline/iop_timeline_generator -input ./mpi_iomsg.dat
```

Case Study – MISA-MD load balance analyze

Assumption: MISA-MD has been successfully && generate mpi_io_msg.dat
run iop_timeline_generator to convert mpi_iomsg.dat to .json timeline format

Run analyze script:

```
cd $IOP_PATH/MISA-MD/example/
$IOP_PATH/iop_install/bin/mpi_io_analysis ./io_timeline.json
```

- Possible Results: Process-level and file-level load balance

```
============================================================
Process-based I/O Load Balance Analysis
============================================================

Process I/O Load Statistics:
--------------------------------------------------------------------------------
Process ID Total I/O (bytes) Total Duration (us) Avg Bandwidth (MB/s) Runtime (ms)
--------------------------------------------------------------------------------
0          13,922,112        37,121            3.71                 3576.62
12         12,985,952        74,111            3.45                 3588.17
3          12,985,952        72,848            3.45                 3589.01
15         12,781,832        71,759            3.40                 3588.14
21         12,722,912        71,040            3.38                 3586.94
6          12,722,912        71,873            3.38                 3586.97
45         12,722,912        74,772            3.38                 3587.69
42         12,722,912        72,709            3.39                 3584.19
39         12,722,912        74,492            3.38                 3587.81
36         12,722,912        70,747            3.38                 3584.61
33         12,722,912        71,882            3.38                 3584.98
27         12,722,912        74,929            3.38                 3589.07
24         12,722,912        71,209            3.38                 3584.48
30         12,722,912
9          12,722,912
18         12,722,912
4          12,362,912
40         12,362,912
41         12,362,912
13         12,362,912

Process Load Balance Assessment:
  Maximum I/O Volume: 13,922,112 bytes
  Minimum I/O Volume: 12,362,912 bytes
  Average I/O Volume: 12,520,083 bytes
  Load Imbalance Index: 11.20%
  Coefficient of Variation (CV): 2.26%
  Load Balance Rating: Excellent
```

```
============================================================
File-based I/O Load Balance Analysis
============================================================

File I/O Load Statistics:
--------------------------------------------------------------------------------
File FD  Total I/O (bytes) Total Duration (us) Avg Bandwidth (MB/s) Access Time Range (ms)
--------------------------------------------------------------------------------
10       72,001,840        11,773            9781.53              7.02
3        72,001,840        12,598            9770.39              7.03
4        72,001,840        10,971            9280.48              7.40
5        72,001,840        11,187            10744.22             6.39
6        72,001,840        11,069            10827.23             6.34
7        72,001,840        10,327            11946.12             5.75
8        72,001,840        11,262            10501.04             6.54
9        72,001,840        9,554             13538.31             5.07
-1001    23,160,192        3,224,767         26.38                837.23
1        894,532           187,430           84.57                10.09
2        894,532           5,650             195.62               4.36

File Load Balance Assessment:
  Maximum I/O Volume: 72,001,840 bytes
  Minimum I/O Volume: 894,532 bytes
  Average I/O Volume: 54,633,089 bytes
  Load Imbalance Index: 31.79%
  Coefficient of Variation (CV): 55.46%
  File Load Balance Rating: Poor
```

Note: IOP can analyze I/O performance logs to generate file view and process view load information.

Case Study –MISA-MD I/O Bandwidth analyze

Assumption: MISA-MD has been successfully && generate mpi_io_msg.dat
run iop_timeline_generator to convert mpi_iomsg.dat to .json timeline format

Run analyze script:

```
cd $IOP_PATH/MISA-MD/example/
$IOP_PATH/iop_install/bin/mpi_io_analysis ./io_timeline.json
```

Commands are the same as Previous page.

- Possible Results: Process-level and file-level I/O bandwidth analyze.

```
Process Average Bandwidth Ranking:
-------------------------------------------------
Process ID Avg Bandwidth (MB/s) Max Bandwidth (MB/s)
-------------------------------------------------
13         14950.86             35156.25
45         14636.54             35156.25
29         14611.34             35156.25
20         14593.07             35156.25
2          14592.10             35156.25
16         14531.16             35156.25
15         14514.21             35156.25
17         14506.61             35156.25
36         14492.39             35156.25
4          14489.23             35156.25
```

```
File Average Bandwidth Ranking:
-------------------------------------------------
File FD  Avg Bandwidth (MB/s) Max Bandwidth (MB/s)
-------------------------------------------------
10       16139.88             35156.25
9        16039.76             35156.25
6        16035.88             35156.25
5        16026.73             35156.25
4        15917.43             35156.25
8        15536.97             35156.25
7        15201.42             35156.25
3        15196.80             35156.25
1        2476.50              29296.88
2        2186.58              29296.88
```

Note: IOP can analyze I/O performance logs to generate file view and process view Bandwidth information.

Case Study – MISA-MD LOAD BALANCE analyze

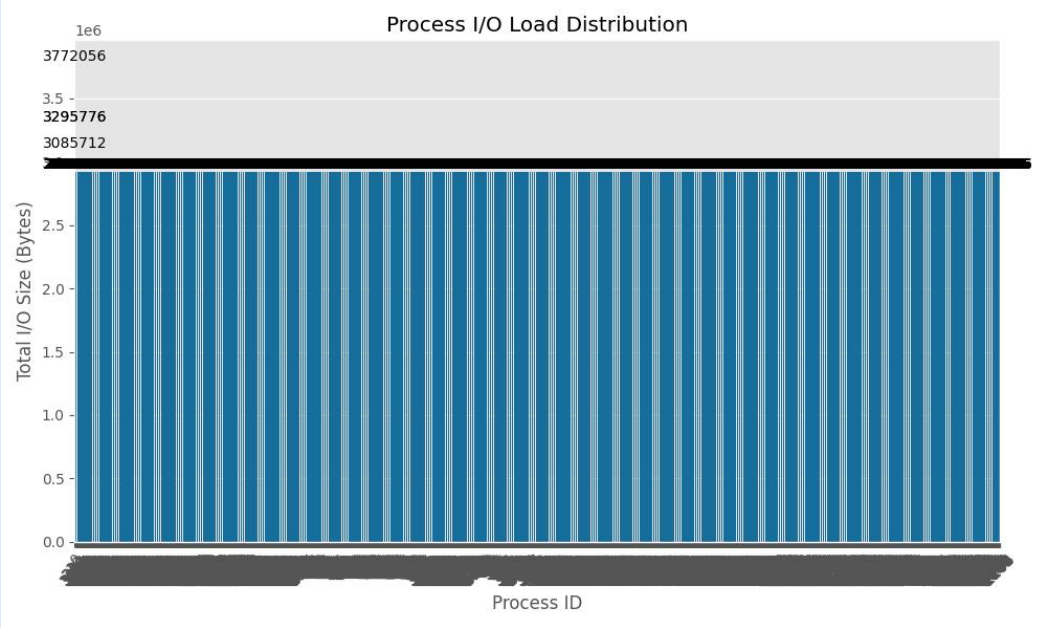
Assumption: MISA-MD has been successfully && generate mpi_io_msg.dat

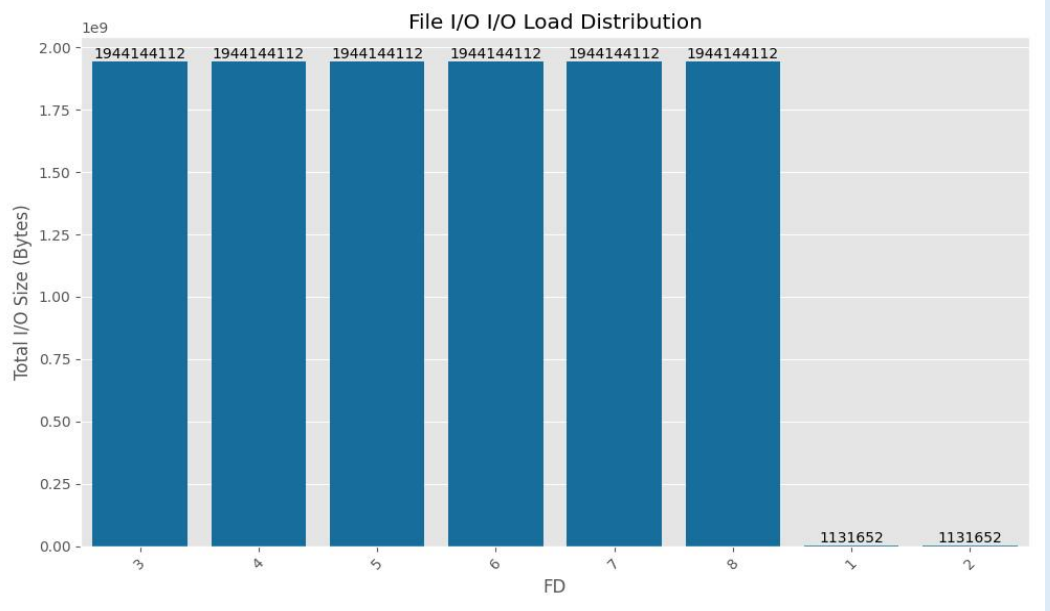
run iop_timeline_generator to convert mpi_iomsg.dat to .json timeline format

Run analyze script:

```
cd $IOP_PATH/MISA-MD/example/
$IOP_PATH/iop_install/bin/mpi_io_analysis.py ./io_timeline.json
```

- Download the analyses result **file** to local via sftp/scp and view it:





Note: IOP can analyze I/O performance logs to generate file view and process view load information.

Case Study –MISA-MD I/O Bandwidth analyse

Assumption: MISA-MD has been successfully && generate mpi_io_msg.dat

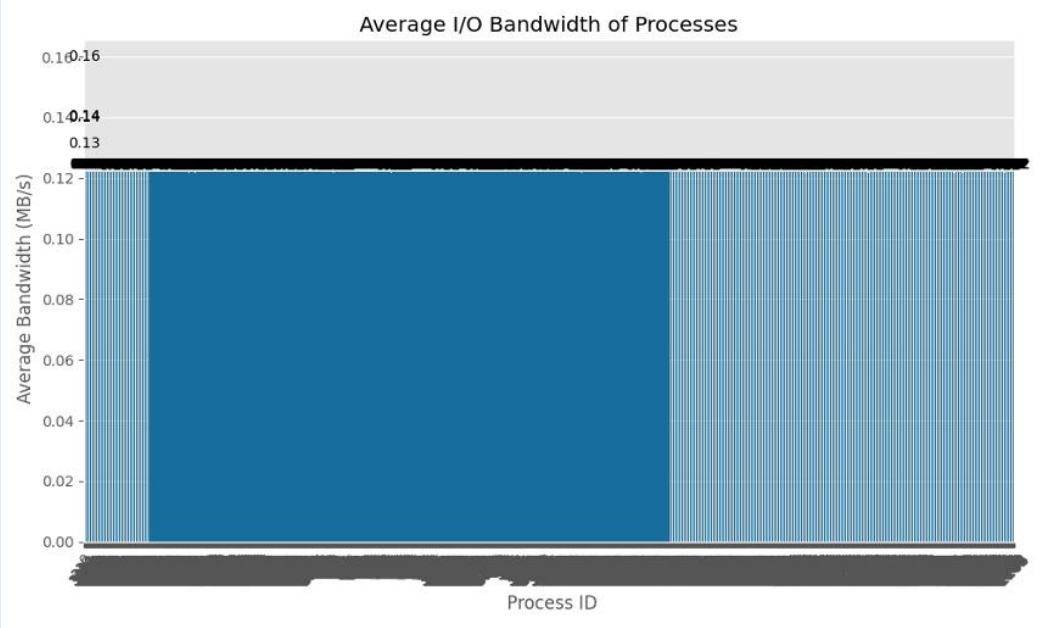
run ./iop_timeline_generator -input $MISA-MD_PATH/exapmle/mpi_io_msg.dat

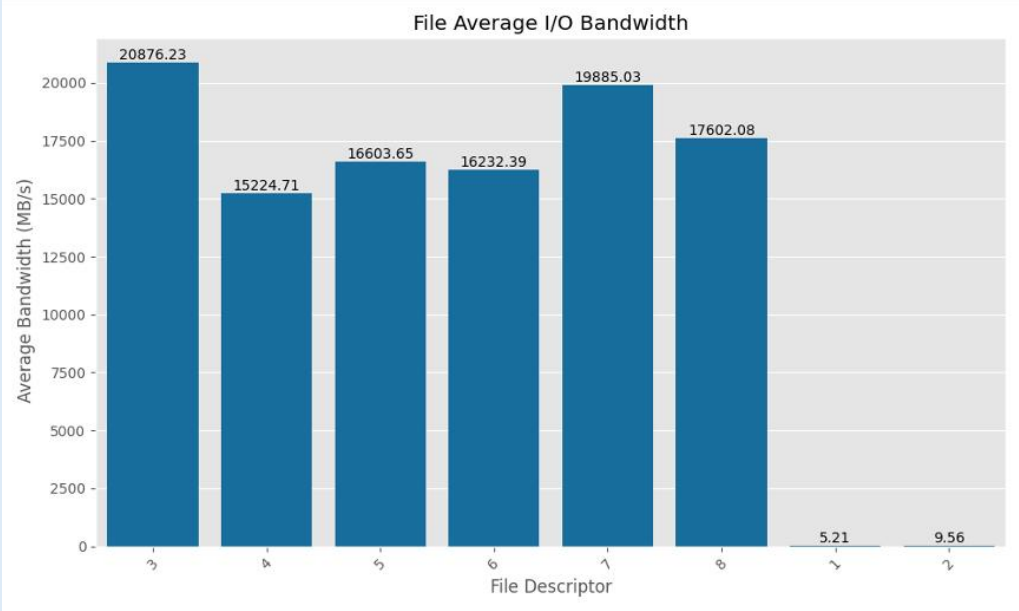
Run analyze script:

```
cd $IOP_PATH/MISA-MD/example/
$IOP_PATH/iop_install/bin/mpi_io_analysis.py ./io_timeline.json
```

Commands are the same as Previous page.

- Possible Results:





Note: IOP can analyze I/O performance logs to generate file view and process view Bandwidth information.

Case Study – Application: Phi-Flow

- Phi-flow overview:
  - It is a CFD application developed for nuclear reactor simulation.
  - In phi-flow, MPI processes read mesh files as input.
- **Slurm script to run phi-flow and generate iop performance record (skip):** the record data is placed in ~/shared/iop/phi-flow-pre-build/sfio/ direcotry

```
#!/bin/bash
#SBATCH –p normal
#SBATCH –o cfd–96–%J.log
#SBATCH –e cfd–96–%J.err
#SBATCH –J channel138
#SBATCH –n 92

module purge

module unload mpi/hpcx/2.11.0/gcc–7.3.1
module load mpi/intelmpi/2021.3.0

export LD_PRELOAD=$IOP_PATH/iop_install/lib/libiop_hook.so

# Run time
srun ––mpi=pmi2 –n 96 ${Phi_Flow/BUILD_PATH}/bin/nvwa_c
 channel30
```

Num of processes

Exclusive mode (Avoid resource competition)

Loading environment

Runtime injection via `LD_PRELOAD` load iop

Case Study – Phi-Flow Small File I/O analyze

Assumption: Phi-Flow has been successfully && generate mpi_io_msg.dat
run ./iop_timeline_generator –input $Phi-Flow_PATH/run/chanel30/mpi_io_ms

Run code path:

```
cd $IOP_PATH
cp -r ~/shared/iop/phi-flow-pre-build/sfio/ ./small_file_io
cd small_file_io
$IOP_PATH/iop_install/bin/small_file_io ./phi_io_timeline.json
```

- Possible results:

```
(mpi_io_analysis) [genshen@login08 small_file_io]$ python3 $IOP_PATH/iop_install/bin/small_file_io.py ./phi_io_timeline.json
Analyzing file: ./phi_io_timeline.json

================================================================
MPI I/O Fragmented Small I/O Request Diagnosis
================================================================

------------------------------------------------
Analyzing operation type: MPI_File_write_all
------------------------------------------------
  Request Count: 56
  Total Data Volume: 4,141,992.0 bytes (3.95 MB)
  Average Request Size: 73,964 bytes
  Request Size Coefficient of Variation: 1.390
  Small Request Ratio: 66.1%

----------------------------------------------------------------
Detailed Analysis Report
----------------------------------------------------------------

 Request Size Distribution Analysis:
Size Range   Requests Ratio    Total Data
------------------------------------------------
< 1KB        19       33.9   % 264.0
1KB - 8KB    18       32.1   % 36,000.0
8KB - 64KB   0        0.0    % 0.0
64KB - 1MB   19       33.9   % 4,105,728.0
> 1MB        0        0.0    % 0.0
```

- **IOP analyzes the relationship between I/O request size and duration,also have frequency distribution of I/O request sizes.**

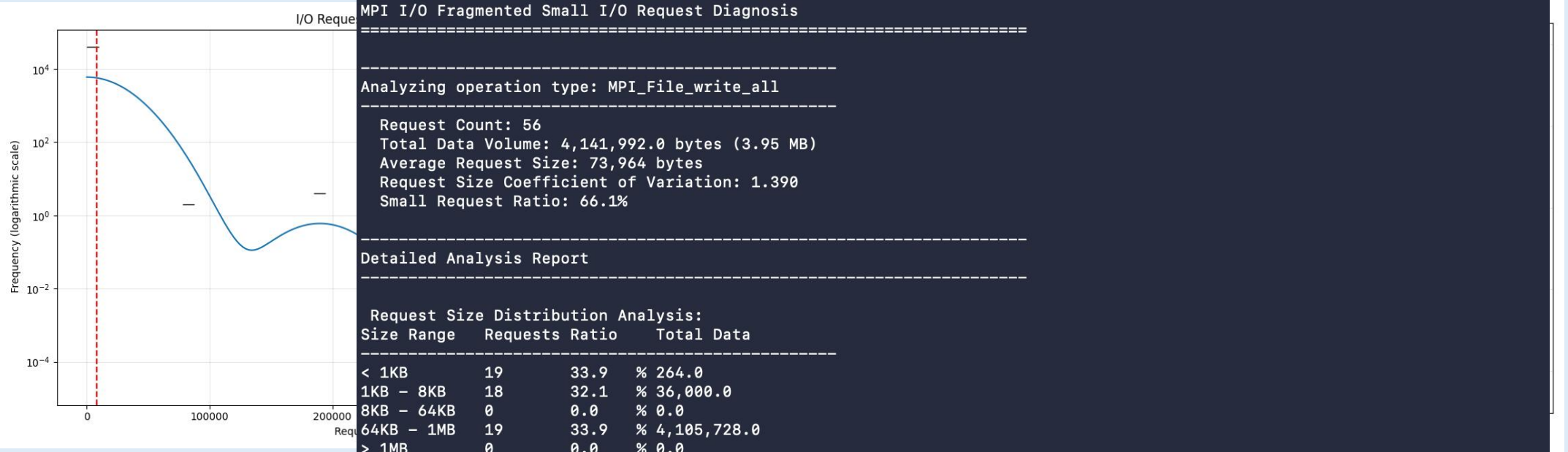
Case Study – Phi-Flow Small File I/O analyse

Assumption: Phi-Flow has been successfully && generate mpi_io_msg.dat
run ./iop_timeline_generator –input $Phi-Flow_PATH/run/chanel30/mpi_io_ms

Run code path:

```
cd $IOP_PATH
cp -r ~/shared/iop/phi-flow-pre-build/sfio/ ./small_file_io
cd small_file_io
$IOP_PATH/iop_install/bin/small_file_io.py ./phi_io_timeline.json
```

- Download the anal




```
(mpi_io_analysis) [genshen@login08 small_file_io]$ python3 $IOP_PATH/iop_install/bin/small_file_io.py ./phi_io_timeline.json
Analyzing file: ./phi_io_timeline.json

======================================================================
MPI I/O Fragmented Small I/O Request Diagnosis
======================================================================

--------------------------------------------------
Analyzing operation type: MPI_File_write_all
--------------------------------------------------
  Request Count: 56
  Total Data Volume: 4,141,992.0 bytes (3.95 MB)
  Average Request Size: 73,964 bytes
  Request Size Coefficient of Variation: 1.390
  Small Request Ratio: 66.1%

----------------------------------------------------------------------
Detailed Analysis Report
----------------------------------------------------------------------

 Request Size Distribution Analysis:
Size Range    Requests Ratio    Total Data
--------------------------------------------------
< 1KB         19       33.9    % 264.0
1KB - 8KB     18       32.1    % 36,000.0
8KB - 64KB    0        0.0     % 0.0
64KB - 1MB    19       33.9    % 4,105,728.0
> 1MB         0        0.0     % 0.0
```

- **IOP analyzes the relationship between I/O request size and duration,also have frequency distribution of I/O request sizes.**

Thanks!

Genshen Chu

Contect me: cgs@ustb.edu.cn

Case Study –MISA MD I/O Bandwidth analyse

Assumption: MISA-MD has been successfully && generate mpi_io_msg.dat

run ./iop_timeline_generator –input $MISA-MD_PATH/exapmle/mpi_io_msg.dat

Run code path:

```
cd $IOP_PATH/load_balance_analyse
./mpi_io_analysis.py $MISA-MD_PATH/exapmle/mpi_io_msg.json
```

Case Study – Application: Phi-Flow

- Build && Deployment Tutorial

  https://git.hpcer.dev/source-flow/source-flow/Readme

- RUN scprit

```
#!/bin/bash
#SBATCH –p normal
#SBATCH –o cfd–96–%J.log
#SBATCH –e cfd–96–%J.err
#SBATCH –J channel138
#SBATCH –n 192

module purge

module load  GCC/9.3.0  cmake/3.30.4–gcc9.3.0
mpich/mpi–n–gcc9.3.0
module load openblas/0.3.12–gcc9.3.0

export LD_PRELOAD=/public/home/cgs/zhuhaoran/iop–develop/build/lib/libiop_hook.so

# Run time
#srun ––mpi=pmi2 –n 96 ${Phi_Flow/BUILD_PATH}/bin/nvwa_c
 channel30
```

Num of processes

Exclusive mode(Avoid resource competition)

Loading environment

Runtime injection via `LD_PRELOAD` load iop