A Unified I/O Monitoring Framework Using eBPF

Mahendra Paipuri

Head of Digital Projects, CDSP, SciencesPo Paris, CNRS

FOSDEM 2026 - eBPF Devroom

31th January 2026

SciencesPo

CENTRE DE DONNÉES SOCIO-POLITIQUES

cnrs

Context

AI is transforming HPC

Traditional HPC tools are focused on MPI workloads and cannot be used “out-of-the-box” with AI workloads

AI workloads are often limited by I/O

It is very important to understand application’s I/O pattern to optimize workflows

Non standardised telemetry of Parallel File Systems

Sometimes metrics are available only at the node level

Standard tools like Darshan are strongly coupled with MPI

Limited support for non MPI workloads and still require compile time modifications

Ideally what we need?

I/O monitoring agnostic of app and file system

Improves portability of the tool and helps in standardisation of telemetry systems

Minimal to no changes to apps either at compile time or runtime

Allows plug and play approach across different platforms

Minimally intrusive and negligible overhead

A bonus if this can be done on production workflows without losing performance

Ability to capture I/O events at any level in the stack

Tracing events both in kernel and user spaces.

eBPF - extended Berkeley Packet Filter

Extend capabilities of the kernel at RUNTIME

No need of kernel modules, no need to recompile kernel and no reboots required

Successor to Berkeley Packet Filter (BPF)

Classical BPF only dealt with network packet processing

Programs are triggered by kernel events

eBPF programs receives pointers to kernel and user space memory

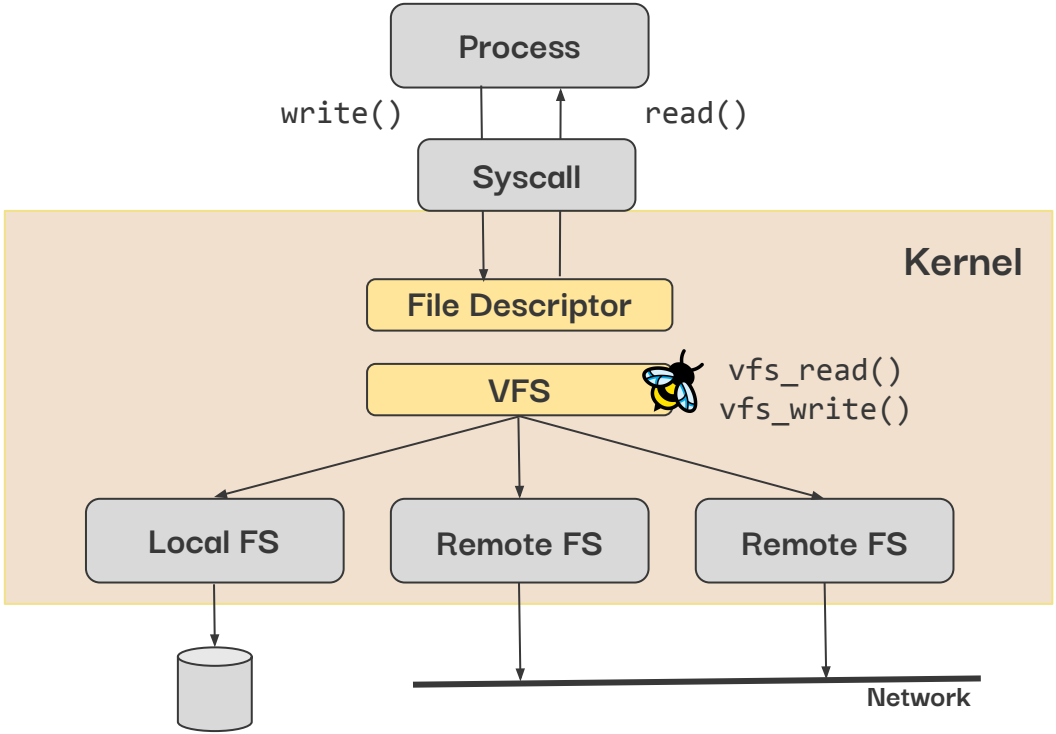
Maps allow sharing data between kernel and user space

Very efficient way to read data from and into kernel space from user space programs

Heavily adopted in cloud-native landscape

Became an indispensable technology in Observability and Telemetry

Tracing VFS using eBPF



Function	Description
`vfs_read`	Read a file
`vfs_write`	Write a file
`vfs_open`	Open a file
`vfs_create`	Create a file
`vfs_mkdir`	Make a directory
`vfs_unlink`	Remove a file
`vfs_rmdir`	Remove a directory

Sample eBPF code to trace VFS

```
# include <linux/bpf.h>
# include <bpf/bpf_helpers.h>
# include <bpf/bpf_tracing.h>
```

Headers and type definitions

```
struct vfs_event_key {
      __u32 cid; /* cgroup ID */
      __u8 mnt[64]; /* Mount point */
};

struct vfs_rw_event {
      __u64 bytes; /* Bytes counter */
      __u64 calls; /* Call counter */
      __u64 errors; /* Error counter */
};
```

Struct definition for VFS event

```
struct {
      __uint(type, BPF_MAP_TYPE_LRU_HASH);
      __uint(max_entries, 16834);
      __type(key, struct vfs_event_key);
      __type(value, struct vfs_rw_event);
} write_accumulator SEC(".maps");
```

BPF map definition

```
char LICENSE[] SEC("license") = "GPL";
```

eBPF programs must be GPL compatible

Sample eBPF code to trace VFS

```
SEC("kprobe/vfs_write")
__u64 kprobe_vfs_write(struct pt_regs *ctx)
{
    struct file *file = (struct file *)PT_REGS_PARM1(ctx);
    __u64 bytes = (__u64)PT_REGS_PARM3(ctx);
```

Read args

```
    struct vfs_event_key key = { 0 };
    key.cid = (__u32)ceems_get_current_cgroup_id();
    get_mnt_path(&key, file);
```

Get cgroup ID and mount path of file

```
    event = bpf_map_lookup_elem(&write_accumulator, &key);
    if (!event) {
        struct vfs_rw_event new_event = { .bytes = bytes, .calls = 1, .errors = 0 };
        bpf_map_update_elem(&write_accumulator, &key, &new_event, BPF_NOEXIST);
        return 0;
    }

    __sync_fetch_and_add(&event->calls, 1);
    __sync_fetch_and_add(&event->bytes, (__u64)bytes);
}
```

Look up and increment counters

Tests

A NFS server has been created for tests

NFS server has been mounted on the same machine on loopback to eliminate noise with network

IOR benchmark has been chosen for evaluating current approach

Tested on one NFS client using transfer sizes of 1,2,4,8 and 16 MiB. Tests have been repeated for 8 times for POSIX and MPIIO.

CEEMS Exporter has been used for monitoring I/O using eBPF

Prometheus has been configured to scrape metrics from exporter for every 2 sec.

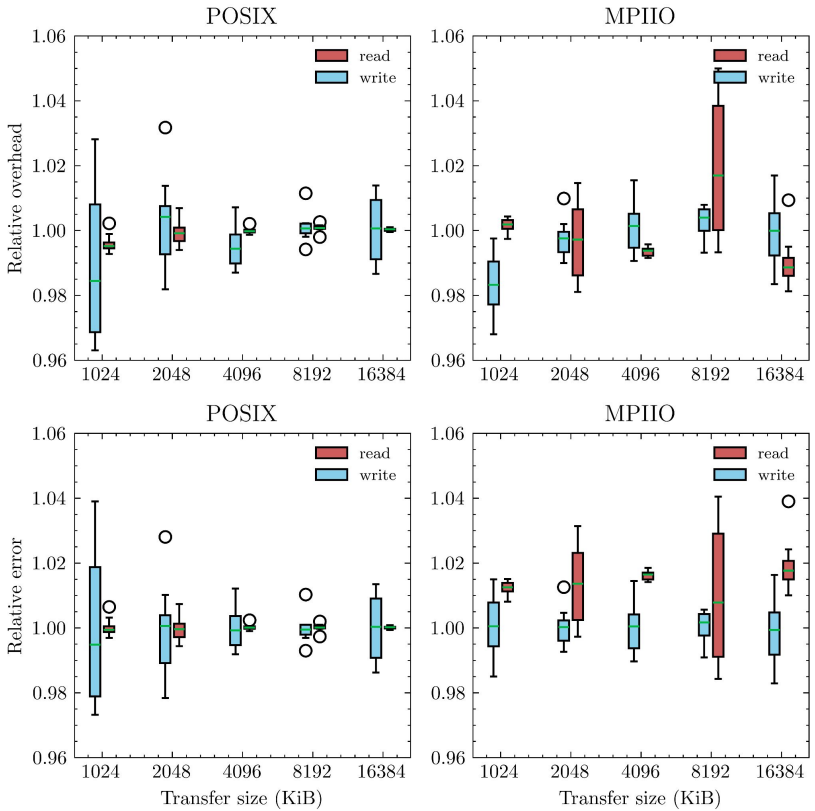
Relative overhead. Higher means more overhead.

Ratio of observed bandwidth reported by IOR without exporter to observed bandwidth reported by IOR with exporter enabled.

Relative error. Closer to 1 means less error.

Ratio of observed bandwidth reported by IOR to mean of the instantaneous bandwidth reported by the exporter

Tests


POSIX
MPIIO
read
write
Relative overhead
Relative error
1.06
1.04
1.02
1.00
0.98
0.96
1024
2048
4096
8192
16384
Transfer size (KiB)

Tests

Multi node IOR test on Jean Zay HPC platform

4 nodes with 16 MPI processes with transfer size of 1 MiB on LUSTRE file system for POSIX and MPIIO

Darshan has been used to compare results from current approach

Darshan reported aggregate metrics with a bin interval of ~52 sec.

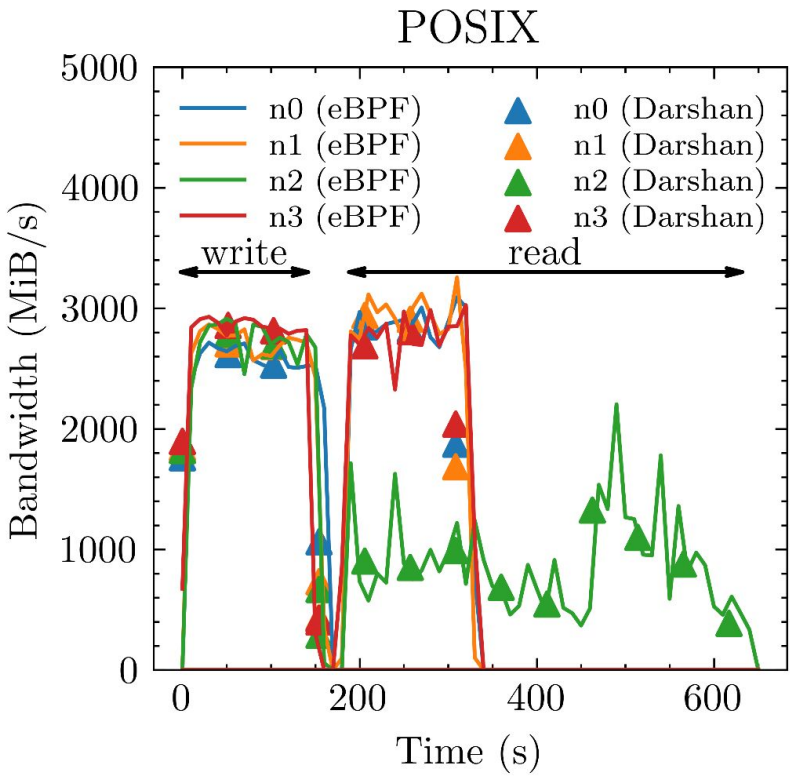
CEEMS Exporter has been used for monitoring I/O using eBPF

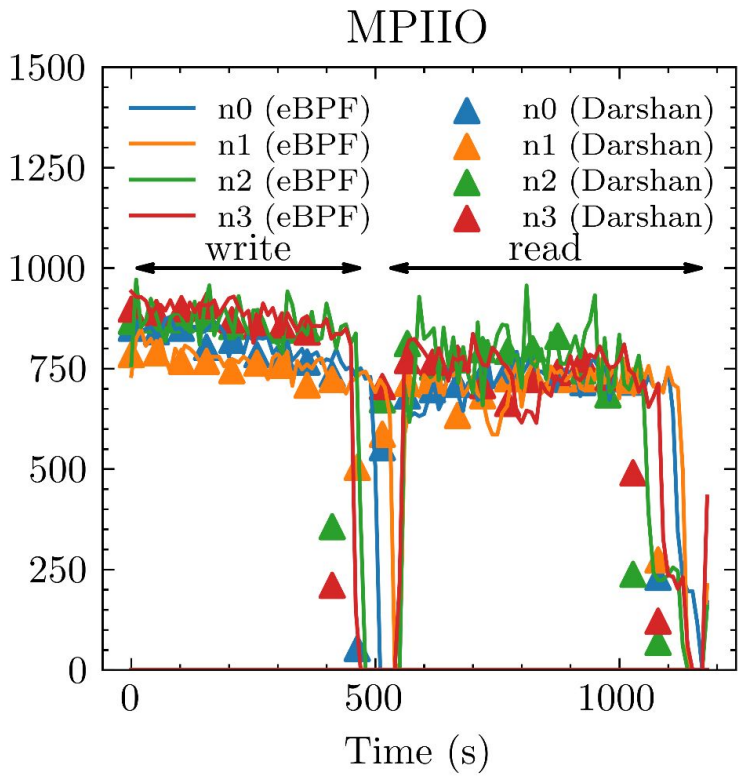
Prometheus has been configured to scrape metrics from exporter for every 10 sec.

Instantaneous bandwidths between current approach and Darshan have been compared

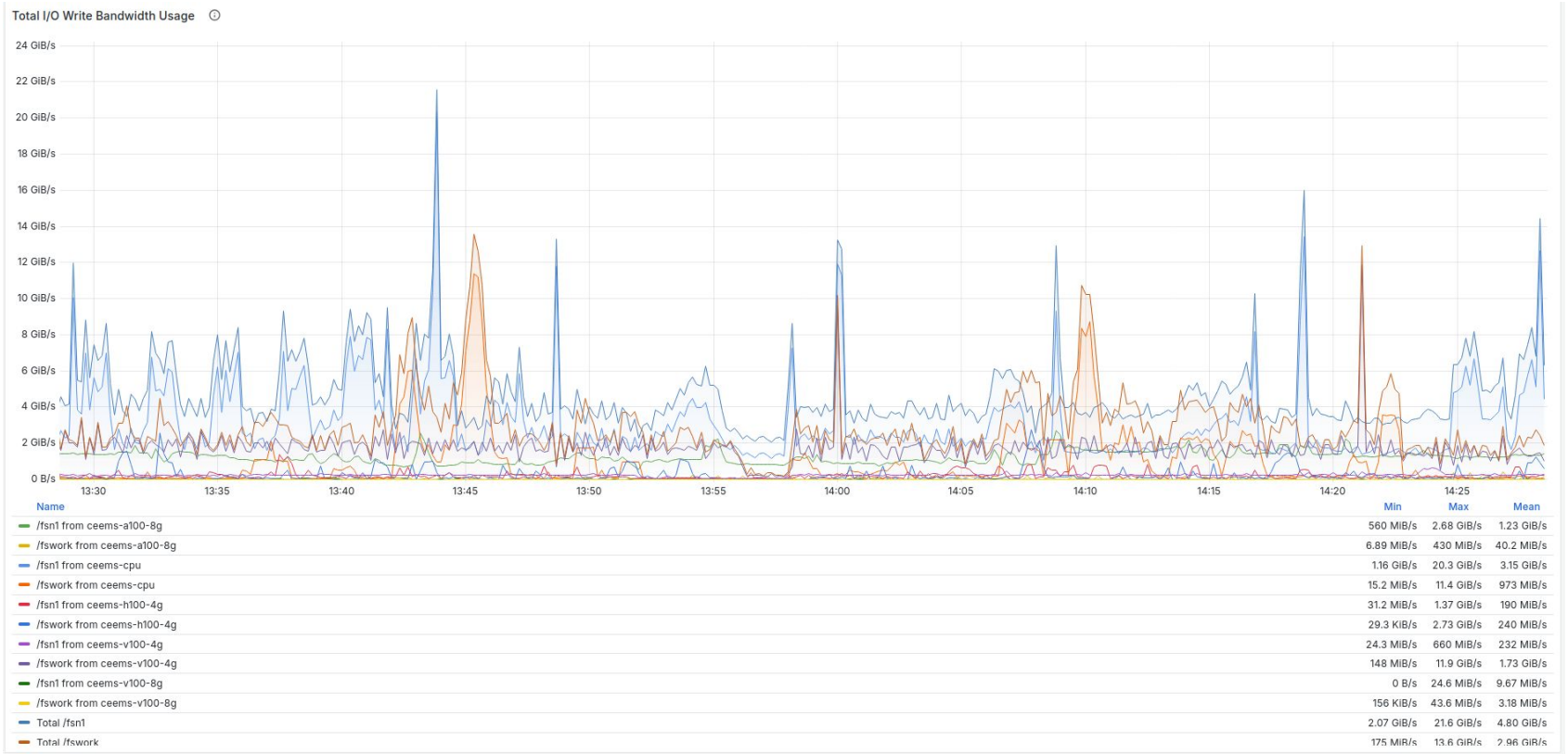
Number of bytes written/read within a bin interval (~52 sec for Darshan and 10 sec for CEEMS exporter)

Tests

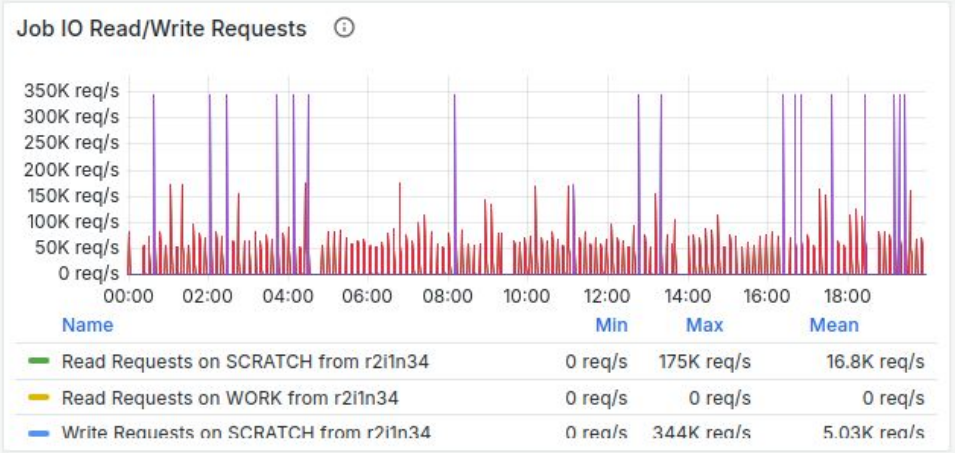

POSIX
n0 (eBPF)
n1 (eBPF)
n2 (eBPF)
n3 (eBPF)
n0 (Darshan)
n1 (Darshan)
n2 (Darshan)
n3 (Darshan)
write
read
Bandwidth (MiB/s)
Time (s)
0
1000
2000
3000
4000
5000
0
200
400
600


MPIIO
n0 (eBPF)
n1 (eBPF)
n2 (eBPF)
n3 (eBPF)
n0 (Darshan)
n1 (Darshan)
n2 (Darshan)
n3 (Darshan)
write
read
Time (s)
0
250
500
750
1000
1250
1500
0
500
1000

eBPF based I/O monitoring on Jean Zay



eBPF based I/O monitoring on Jean Zay



What more can we do with eBPF?

Trace user space MPI libraries

Almost all the MPI functions (OpenMPI and/or Intel MPI) can be traced to build profiles

eBPF ring buffer is used to send events

A circular multiple producer single consumer buffer that can preserve the event order coming from multiple CPUs

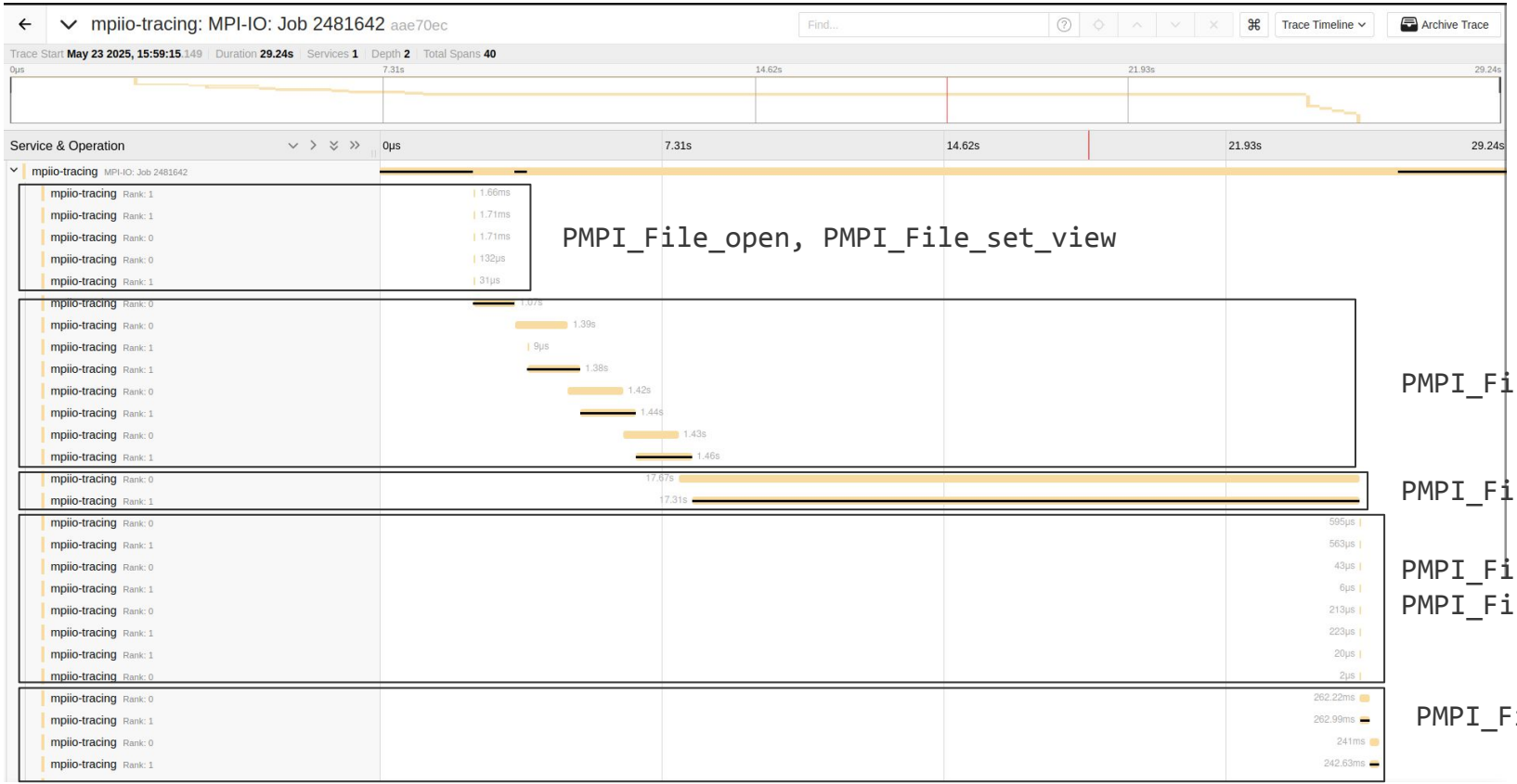
OpenTelemetry SDKs can be used to generate profile traces

These traces can be stored in any OpenTelemetry compatible backend like Prometheus, Jaeger, *etc.*

A rudimentary PoC using Jaeger and OpenTelemetry SDK for MPI-IO

MPI-IO functions have been traced to generate spans using OpenTelemetry Go SDK and sent them to Jaeger for visualization

MPI-IO Tracing Tests



PMPI_File_open, PMPI_File_set_view

PMPI_File_write

PMPI_File_close

PMPI_File_open,
PMPI_File_set_view

PMPI_File_read

Closing Remarks

A zero instrumentation I/O monitoring framework

Agnostic of application and filesystem and capable of performing system wide monitoring

Leverage existing cloud-native tools

Storage and visualization of monitoring data can be done using existing tools like Grafana, Prometheus, Jaeger for better UI/UX

Ability to trace any function or library

Both user and kernel space functions can be traced to generate spans

Can be combined with eBPF based continuous profiling

This gives a complete monitoring and profiling solution that is portable and agnostic across HPC platforms

SciencesPo
CENTRE DE DONNÉES SOCIO-POLITIQUES

cnrs

Thank you

mahendra.paipuri@cnrs.fr

CEEMS Exporter:

GitHub Repository

Docs