INSTITUT POLYTECHNIQUE DE PARIS

ÉCOLE POLYTECHNIQUE

ENSTA

ENSAE

TELECOM Paris

TELECOM SudParis

Investigating IO bottlenecks with EZTrace

François Trahay

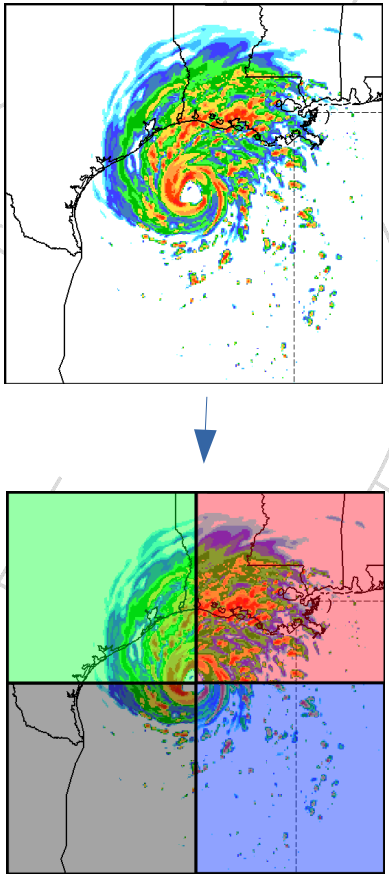
Per3S 2022

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

ip-paris.fr

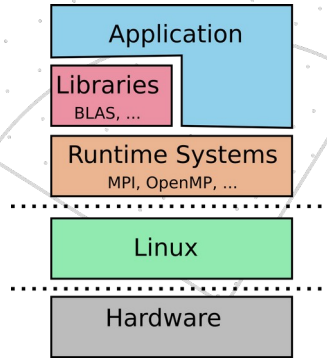
Parallel programming

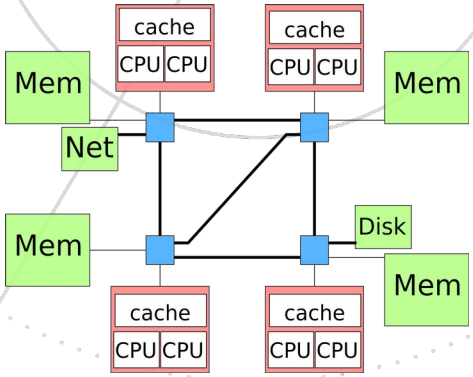
- *High Performance Computing*
  - Extensively used in weather forecasting, molecular modeling, physical simulation, ...
  - Need for a lot of computing power
- *Parallelizing an application*
  - Split a problem into sub problems
  - Distribute over several processors
  - Processors communicate their contribution through a network
- *Performance improvement*
  - 4 processors → 4 times faster
  - 128 processors → 128 times faster ?



Improving parallel applications

- *Many sources of parallel inefficiency*
  - Algorithmic issues
    - Number of synchronization increases at scale
  - Bad usage of hardware ressources
    - Memory access, Disk, ...
- *Improving performance is hard*
  - Need for tools to help developpers





Available resources on a computer

EZTrace

Tracing tool for parallel applications

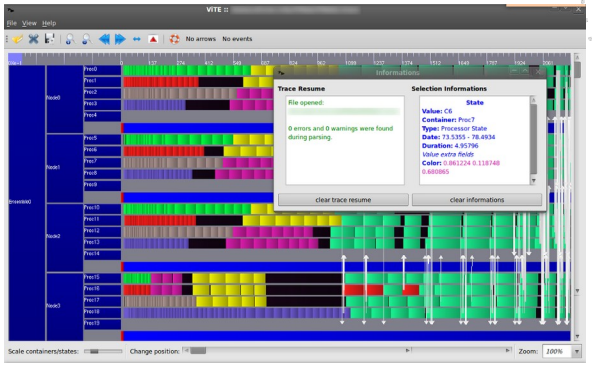
- *Automatically instrument applications*

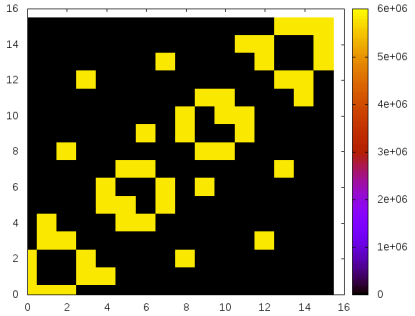
```
$ ./application foo bar
```

→

```
$ eztrace ./application foo bar
```

- *Generate execution trace for post-mortem analysis*





https://eztrace.gitlab.io/eztrace

EZTrace

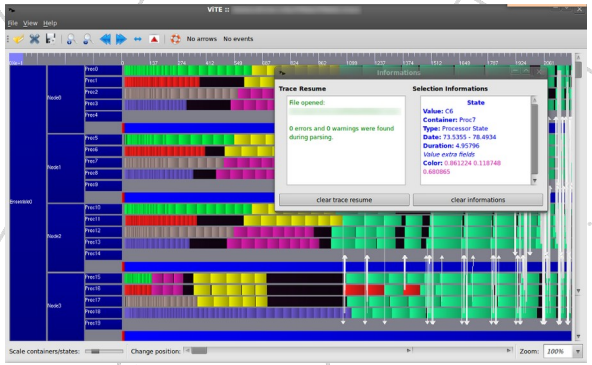
Features

- Plugin-based tracing tool
  - Pre-defined plugins (MPI, CUDA, OpenMP, pthread)
  - User-defined plugins (in C or using a DSL)
- Automatic instrumentation of applications
  - Dynamic interception with LD_PRELOAD
  - Injection of binary instructions in the application
- Lightweight recording of event
  - Generate OTF2 execution traces
  - Low overhead (typically ~2%)

```
int foo(int a, double b) {
    do_something();
}
```



```
int foo(int a, double b) {
    record_event("foo_entry", a);
    int ret = libfoo(a, b);
    record_event("foo_exit", ret);
}
```

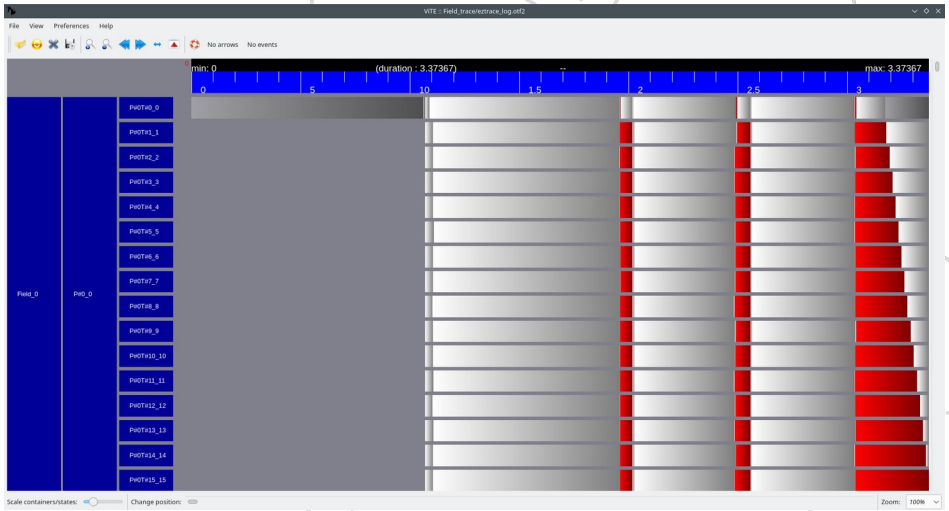


https://eztrace.gitlab.io/eztrace

Visualizing scalability issues with ViTE

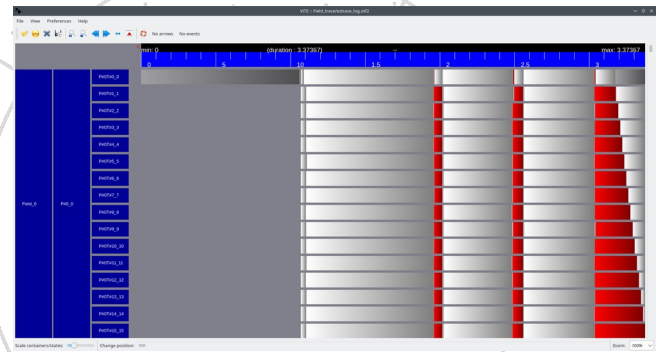
- Visualizing a trace with ViTE
  - Find obvious bottlenecks
  - Works for small traces

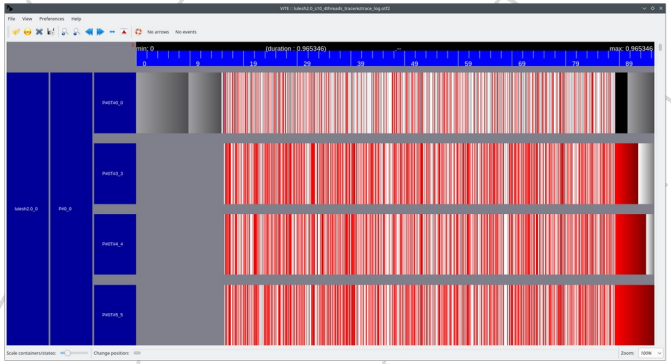
```
$ eztrace -t ompt ./Field

$ vite Field_trace/eztrace_log.otf2
```



Limitations of trace visualization

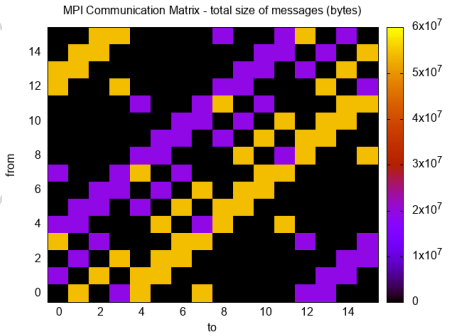
- Loading a large trace requires a lot of RAM
- Some performance problems are hard to spot visually
  - High impact, appear once
    - Easy to sport
  - Low impact, happens often
    - Need to browse the trace
- Solution: automatic trace analysis





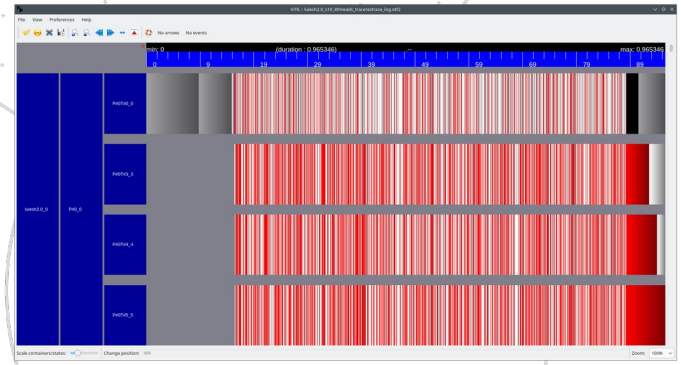
Easy Trace Analyzer

- Collection of tools that analyze traces
  - Support for multiple trace formats (OTF2, Pajé, ...)
  - eta_profile – Profiler
  - eta_mpi_stats – MPI analysis
  - eta_perf_model – Extract/replay a trace
  - eta_merge – Merge traces



https://parallel-and-distributed-systems.gitlab.io/easytraceanalyzer/eta_web/

Extracting a profile from a trace



```
$ eta_profile -t -e eztrace_log.otf2
[…]
Thread P#0_T#3:
|Function                      |   Duration|  % runtime|      Count|        Min|        Max|    Average|        SCI|
|------------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
|OpenMP implicit barrier       |     0.3938|    51.0332|     126819|   5.83e-07|  0.0425763|3.10521e-06|   0.414518|
|OpenMP loop                   |   0.183595|    23.7924|     142296|    1.6e-07|0.000518895|1.29024e-06|    0.20842|
|OpenMP implicit task          |   0.159475|    20.6667|     113420|   1.28e-07|0.000514263|1.40606e-06|   0.187853|
Total duration: 0.771654
```

https://parallel-and-distributed-systems.gitlab.io/easytraceanalyzer/eta_web/

EZIOTracer: tracing the whole I/O stack

→ *Collecting user-level events with EZTrace*

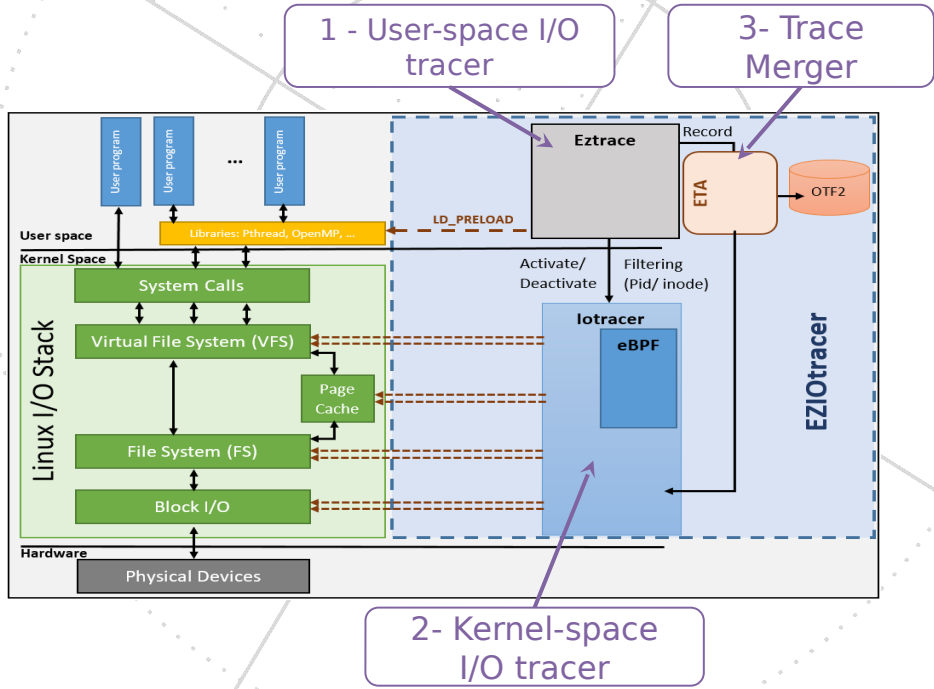
- Helps locating bottlenecks in the application

→ *Collecting kernel-level I/O events with IOTracer*

- Locate the bottleneck in the I/O stack

→ *Merge traces with Easy Trace Analyzer (post-mortem)*

- Unified view of the application + I/O stack



https://gitlab.com/idiom1/eziotrace

Details in the paper:
EZIOTracer: Unifying Kernel and User Space I/O Tracing for Data-Intensive Applications
Mohammed Islam Naas et al.
CHEOPS 2021: Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems
https://hal.archives-ouvertes.fr/hal-03215663v1/

Evaluation

- Hardware

  2 x Intel Xeon Gold 5220R CPUs @ 2.20 GHz (total: 48 cores/96 threads)

  192 GiB of DRAM

  2 x 1.2 TiB 10K RPM disks (RAID1) with ext4 FS

- Software

  Linux kernel 5.10

  3 different run types:

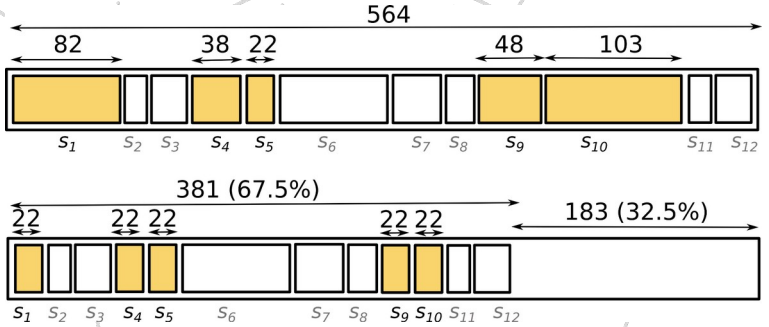
  - ***Vanilla*** *configuration*: Application runs without tracing;
  - ***EZTrace*** configuration: Application runs with **EZTrace** (user-space) tracing only;
  - ***EZIOTracer*** configuration: Application runs with **EZIOTracer** (user- and kernel-level) tracing.

  FIO benchmark with 2 settings:

Metric	Buffered I/O	Non Buffered I/O
I/O block size	4 KiB	4KiB
Number of threads	1,2,4,8,16,32,64	1,2,4,8,16,32,64
Data file size	100 MiB	10 MiB

Evaluation: methodology

- Generate traces with EZTrace/EZIOTracer
- Quantify the contention with the SCI score
  - Slowdown Caused by Thread interference score*
  - SCI = 0 → no contention
  - SCI=1 → Threads spend all their time waiting for a resource
- Manual analysis to identify the source of contention



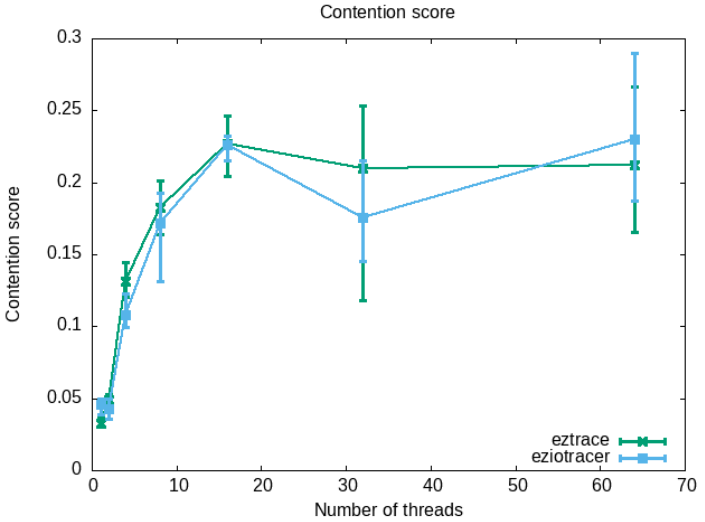
* see **"Using differential execution analysis to identify thread interference."** Mohamed Saïd Mosli Bouksi, François Trahay, et al. In IEEE TPDS 2019 -- https://hal.archives-ouvertes.fr/hal-02179717/

Case study #1 : FIO with non-buffered I/O

- *Low SCI score (~20%)*

  → low contention

- *Each write syscall reaches the block layer*
  - Bottleneck at the lowest layer of the kernel
  - Not taking advantage of the page cache



Level	Non-buffered I/O (10 MiB)
User	81921
VFS	80867
FS	80868
Block	86458

Number of traced events for the 32 threads experiment.

```
./fio --rw=write --ioengine=sync --fdatasync=1 --directory=$data_dir –name=mytest \
--thread  --direct=1 --fsync=1 --size=10m
```

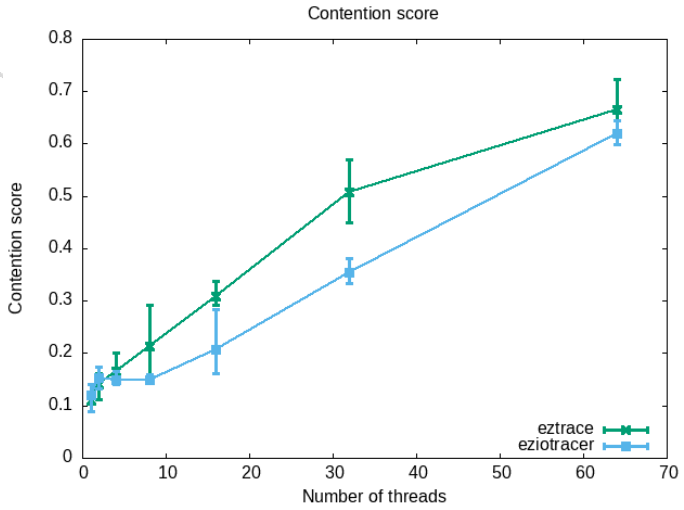
Case study #2: FIO with buffered I/O

- *High SCI score (> 50%)*

  → Most of the time is lost due to contention

- *Calls to write do not reach the block layer*

  – The write traffic is absorbed by the page cache

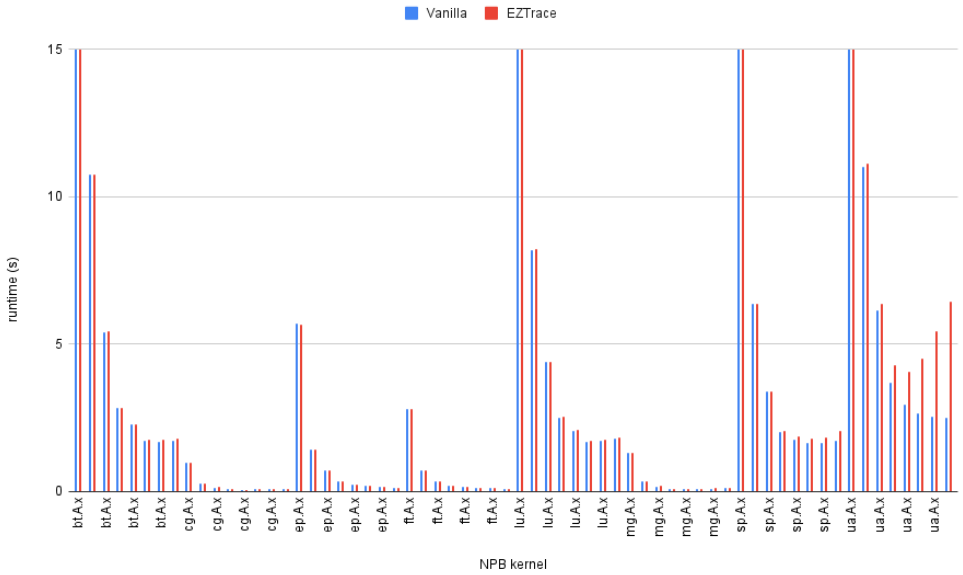


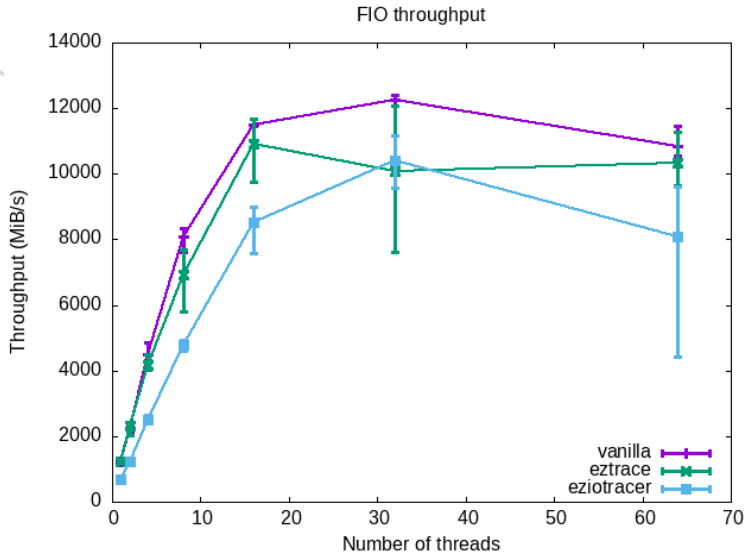
Level	Buffered I/O (100 MiB)
User	819201
VFS	673193
FS	673193
Block	0

Number of traced events for the 32 threads experiment.

```
./fio --rw=write --ioengine=sync --directory=$data_dir --name=mytest --thread --size=100m
```

Overhead evaluation





FIO -- Buffered I/O

- EZTrace overhead < 2% (in most cases)
- High overhead for event intensive apps
  - UA: 5M event/second

- Max throughput = 12 GiB/s
- EZTrace overhead < 5% (in most cases)
- EZIOTracer overhead = 26%

Conclusion

- A tool chain analyzing the performance of parallel applications
  - EZTrace https://eztrace.gitlab.io/eztrace/ / EZIOTracer https://gitlab.com/idiom1/eziotrace
    - Capturing the program behavior/performance
  - ViTE https://solverstack.gitlabpages.inria.fr/vite
    - Trace visualization
  - EasyTrace Analyzer https://gitlab.com/parallel-and-distributed-systems/easytraceanalyzer
    - Collection of trace analysis tools