

Pallas

A generic trace format for large HPC trace analysis

Catherine Guelque, Valentin Honoré, Philippe Swartvagher,
Gaël Thomas & François Trahay



PROGRAMME
DE RECHERCHE

NUMÉRIQUE
POUR L'EXASCALE



RÉPUBLIQUE
FRANÇAISE
*Liberté
Égalité
Fraternité*



PROGRAMME
DE RECHERCHE
NUMÉRIQUE
POUR L'EXASCALE



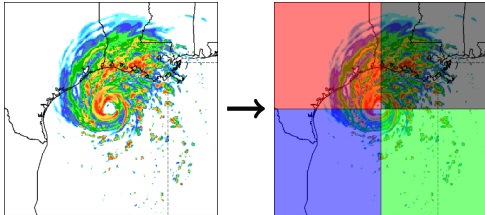
Inria

Introduction & Context

Performance issues in HPC

Performance scales with the number of cores...Right ?

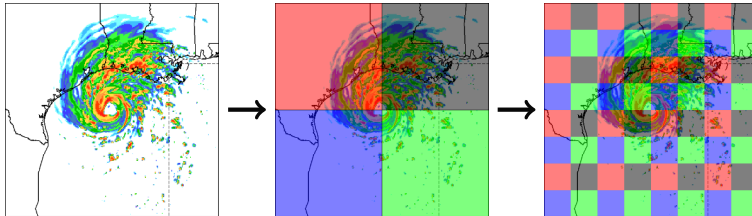
- 4 processors → 3.8 times faster 🥳



Performance issues in HPC

Performance scales with the number of cores...Right ?

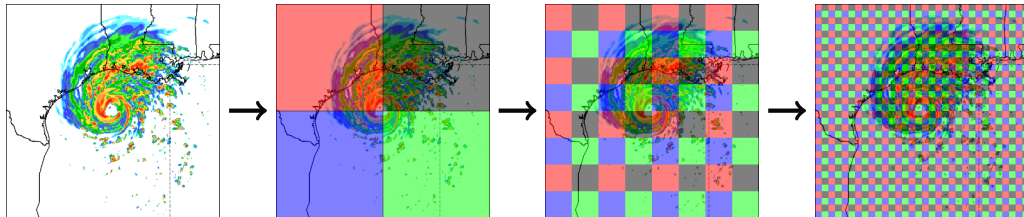
- 4 processors → 3.8 times faster 🥳
- 256 processors → 5.3 times faster 🤔



Performance issues in HPC

Performance scales with the number of cores...Right ?

- 4 processors → 3.8 times faster 🎉
- 256 processors → 5.3 times faster 🤔
- 1600 processors → 127 times **slower** 😭



To scale/debug/optimize these apps, **we need performance analysis tools !**

Traces & tracing tools

Traces

- Timeline of an execution
- Stores events (= POI) and metadata
 - Timestamps
 - Arguments (for function calls)
 - Callstack
 - ...

Tracing tools

Intercept known function calls (MPI, OMP, CUDA) and log them to create a trace



16-threads trace in Vampir

Traces & tracing tools

Traces

- Timeline of an execution
- Stores events (= POI) and metadata
 - Timestamps
 - Arguments (for function calls)
 - Callstack
 - ...

Tracing tools

Intercept known function calls (MPI, OMP, CUDA) and log them to create a trace



32-threads trace in Vite

Traces: limitations



Scalability issues

- Trace size scales with number of threads and execution time
 - Analysis time $\propto$ trace size
 - Resources needed $\propto$ trace size
- $\Rightarrow$ bigger traces need supercomputers to be analyzed !

We need a new, more **scalable** trace format, with:

- low overhead during the execution
- fast & scalable analysis after the execution



RÉPUBLIQUE
FRANÇAISE
*Liberté
Égalité
Fraternité*



PROGRAMME
DE RECHERCHE
NUMÉRIQUE
POUR L'EXASCALE



Inria

Pallas



Pallas

Goal

Accelerating post-mortem analysis:

- **Structural, generic** trace format
- Automatic sequence detection

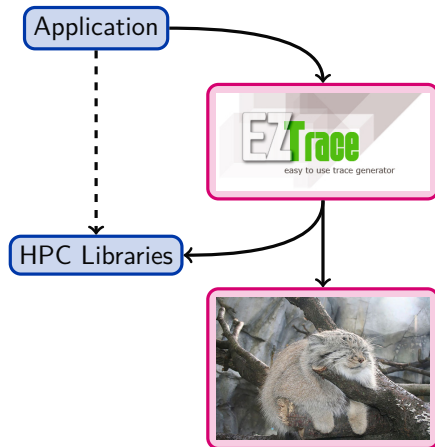
Software stack

EZTrace:

- Intercepts MPI/OMP/CUDA/other calls
- Builds OTF2 traces via OTF2 library

Pallas:

- Provides reading/writing API via C/C++ library
- Provides an OTF2 writing API



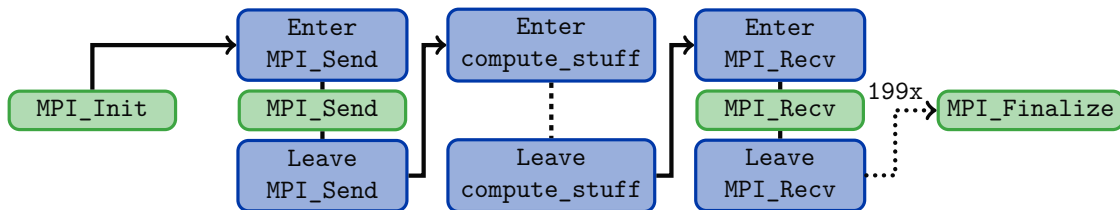
Example: Tracing with EZTrace

EZTrace

Intercepted functions:

- Enter and Leave events = scope
- Punctual event (e.g. message sent)

```
MPI_Init(...)
for _ in range(200):
    MPI_Send(...)
    compute_stuff(...)
    MPI_Recv(...)
MPI_Finalize(...)
```



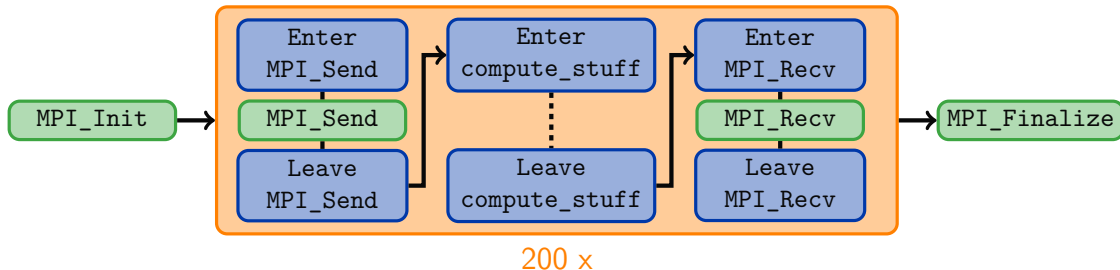
Example: Tracing with EZTrace

EZTrace

Intercepted functions:

- Enter and Leave events = scope
- Punctual event (e.g. message sent)

```
MPI_Init(...)
for _ in range(200):
    MPI_Send(...)
    compute_stuff(...)
    MPI_Recv(...)
MPI_Finalize(...)
```



Structure detection

Pallas Trace Format

- Everything is a **generic token**
- Specific information is stored in a **grammar**
- **Enter** and **Leave** events define **Sequences**
- Repetitions of tokens create **Loops**

E_0

Example: Grammar

- E_0 : MPI_Init

Structure detection

Pallas Trace Format

- Everything is a **generic token**
- Specific information is stored in a **grammar**
- **Enter** and **Leave** events define **Sequences**
- Repetitions of tokens create **Loops**



Example: Grammar

- E_0 : MPI_Init
- E_1 : Enter MPI_Send
- E_2 : MPI_Send
- E_3 : Leave MPI_Send
- S_1 : $E_1 E_2 E_3$ (MPI_Send)

Structure detection

Pallas Trace Format

- Everything is a **generic token**
- Specific information is stored in a **grammar**
- **Enter** and **Leave** events define **Sequences**
- Repetitions of tokens create **Loops**



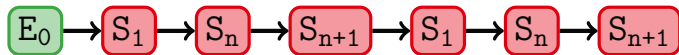
Example: Grammar

- E_0 : MPI_Init
- E_1 : Enter MPI_Send
- E_2 : MPI_Send
- E_3 : Leave MPI_Send
- S_1 : $E_1 E_2 E_3$ (MPI_Send)
- ...
- S_n : $E_4 \dots E_k$ (compute_stuff)
- ...
- S_{n+1} : $E_p E_q E_r$ (MPI_Recv)

Structure detection

Pallas Trace Format

- Everything is a **generic token**
- Specific information is stored in a **grammar**
- **Enter** and **Leave** events define **Sequences**
- Repetitions of tokens create **Loops**



Example: Grammar

- E_0 : MPI_Init
- E_1 : Enter MPI_Send
- E_2 : MPI_Send
- E_3 : Leave MPI_Send
- S_1 : $E_1 E_2 E_3$ (MPI_Send)
- ...
- S_n : $E_4 \dots E_k$ (compute_stuff)
- ...
- S_{n+1} : $E_p E_q E_r$ (MPI_Recv)

Structure detection

Pallas Trace Format

- Everything is a **generic token**
- Specific information is stored in a **grammar**
- **Enter** and **Leave** events define **Sequences**
- Repetitions of tokens create **Loops**



Example: Grammar

- E_0 : MPI_Init
- E_1 : Enter MPI_Send
- E_2 : MPI_Send
- E_3 : Leave MPI_Send
- S_1 : $E_1 E_2 E_3$ (MPI_Send)
- ...
- S_n : $E_4 \dots E_k$ (compute_stuff)
- ...
- S_{n+1} : $E_p E_q E_r$ (MPI_Recv)
- L_m : $2 \times S_1 S_n S_{n+1}$

Structure detection

Pallas Trace Format

- Everything is a **generic token**
- Specific information is stored in a **grammar**
- **Enter** and **Leave** events define **Sequences**
- Repetitions of tokens create **Loops**



Example: Grammar

- E_0 : MPI_Init
- E_1 : Enter MPI_Send
- E_2 : MPI_Send
- E_3 : Leave MPI_Send
- S_1 : $E_1 E_2 E_3$ (MPI_Send)
- ...
- S_n : $E_4 \dots E_k$ (compute_stuff)
- ...
- S_{n+1} : $E_p E_q E_r$ (MPI_Recv)
- L_m : $2 \times S_1 S_n S_{n+1}$

Structure detection

Pallas Trace Format

- Everything is a **generic token**
- Specific information is stored in a **grammar**
- **Enter** and **Leave** events define **Sequences**
- Repetitions of tokens create **Loops**



Example: Grammar

- E_0 : MPI_Init
- E_1 : Enter MPI_Send
- E_2 : MPI_Send
- E_3 : Leave MPI_Send
- S_1 : $E_1 E_2 E_3$ (MPI_Send)
- ...
- S_n : $E_4 \dots E_k$ (compute_stuff)
- ...
- S_{n+1} : $E_p E_q E_r$ (MPI_Recv)
- L_m : $200 \times S_1 S_n S_{n+1}$
- E_{10} : MPI_Finalize

Trace format

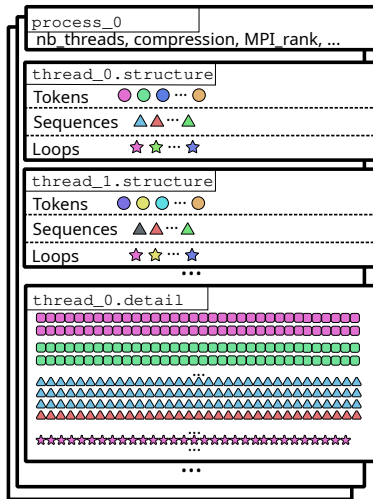
Structure built for HPC

One folder / thread → allows 1k+ threads to write simultaneously on NFS

Smart data storage & retrieval

Data is segregated between:

- Program structure (MB)
 - Execution information & trace metadata
 - Grammar
 - Events / Sequences statistics
- Timestamps and durations (GB/TB)
 - Compressible
 - Loaded on-the-fly



Post-mortem analysis tools

pallas_contention

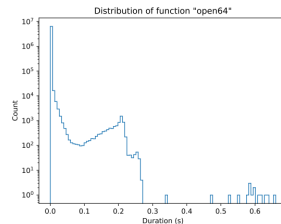
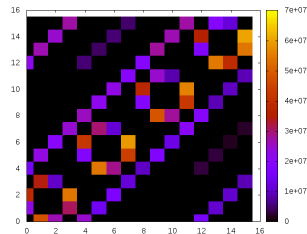
- Compute a contention score based on sequence statistics
- Only needs the program structure

pallas_comm_matrix

- Creates a communication matrix based on the grammar
- Only needs the program structure

pallas_histogram

- Creates a histogram of the distribution of a function's duration
- Only needs the sequence's durations





RÉPUBLIQUE
FRANÇAISE
*Liberté
Égalité
Fraternité*



PROGRAMME
DE RECHERCHE
NUMÉRIQUE
POUR L'EXASCALE



Inria

Benchmarks and Evaluations

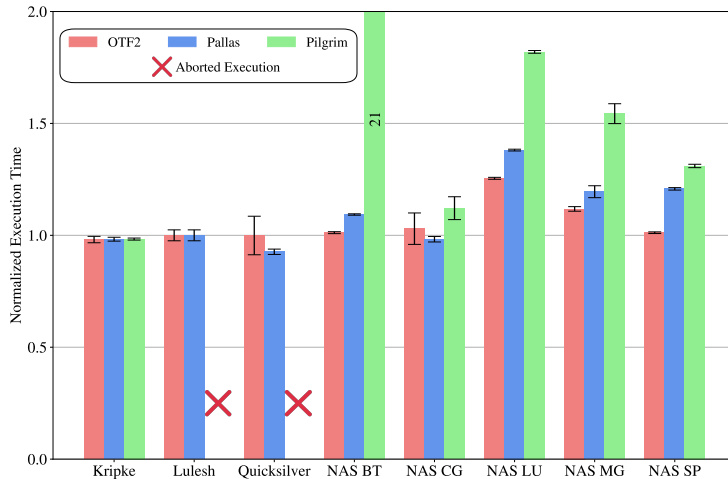
Experimental parameters

- Platform: **Jean-Zay** supercomputer
 - Scalar partition: 2x20 cores, 192 GiB RAM
 - Execution using 4096 MPI ranks
 - GPU partition: 2x20 cores, 192 GiB RAM, 4 NVIDIA V100 GPUs
 - Execution using 128 MPI ranks
- We compare
 - Vanilla: no tracing
 - OTF2: tracing with EZTrace in OTF2
 - Pallas: tracing with EZTrace in Pallas
 - Pilgrim: tracing with Pilgrim
- Applications: Lulesh, Quicksilver, Kripke, NAS Parallel Benchmark
 - Tracing the MPI functions

Overhead

Key points

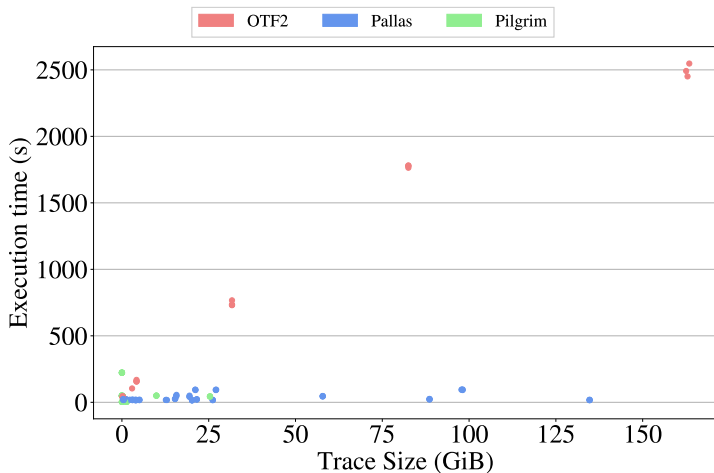
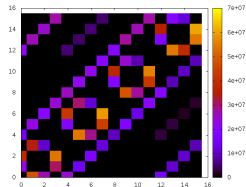
- **Lower is better !**
- Pallas adds very little overhead compared to OTF2
- Pilgrim struggles when tracing complex applications



Analysis speed: Communication Matrix

Key points

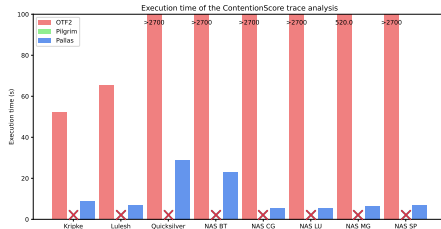
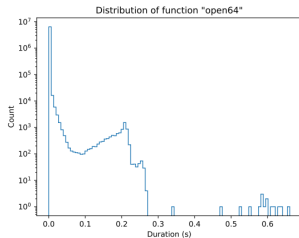
- Pilgrim/Pallas $\lll$ OTF2
- Pilgrim/Pallas $\rightarrow$ scalable



Analysis speed: Contention & Histogram

Key points

- Pilgrim has no API for analysis
- Pallas → scalable, almost constant
- OTF2 needs to read a lot more data



Case study: Distributed training of a ML model

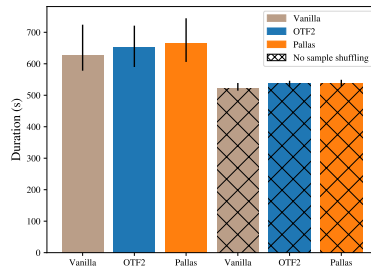
Resnet50

- 5 epochs over the ImageNet dataset over 128 GPUs
- Known issue: sample shuffling causes I/O contention
- Tracing with OTF2 & Pallas
 - MPI, PosixIO, some Python functions

Results

- Vanilla has significant variation for its duration
- Pallas' time to solution is significantly shorter

	EZTrace/OTF2	EZTrace/Pallas
Trace Size	6.1GiB	1.4GiB
Total Time	1366s	160s





RÉPUBLIQUE
FRANÇAISE
*Liberté
Égalité
Fraternité*



PROGRAMME
DE RECHERCHE
NUMÉRIQUE
POUR L'EXASCALE



Inria

Conclusion

Conclusion

Current features

- ✓ Low Overhead
- ✓ Structure detection
 - ✓ Leveraged in quick and scalable analyses
- ✓ Efficient storage
- ✓ Efficient compression
- ✓ On demand-trace loading and exploration
- ✓ Python API and basic trace visualization

Future endeavours

- Trace visualization:
 - Open a 50 GiB trace instantaneously
 - Displaying 1k threads on an average screen
- Improve trace analysis API (Python, R)
- Improve tracing performance

Come check us out at:
gitlab.inria.fr/pallas/pallas

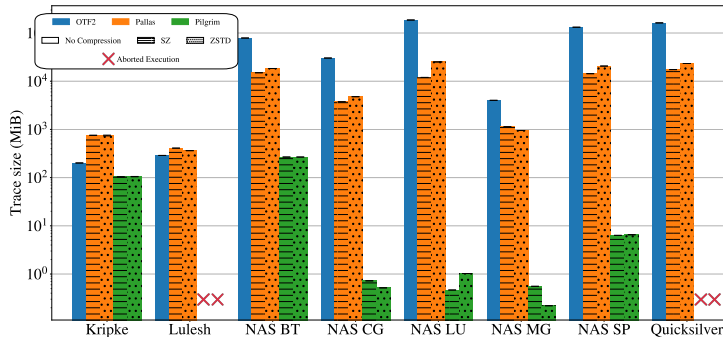


Appendix

Trace size & compression

Key points

- **Lower is better* !**
- $\text{OTF2} \approx 10 \cdot \text{Pilgrim}$
- Pilgrim collects less information than EZTrace
- Pilgrim **compresses all the timestamps together.**



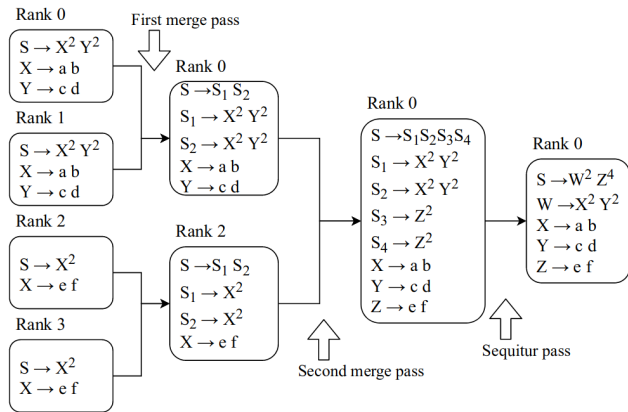
Timestamp compression & encoding

Durations are similar → easily compressible

Different storage options:

- No timestamps (Structure only)
 - Encoding:
 - Removed leading 0s
 - Replace leading 0s (as presented before)
 - Compression:
 - ZSTD
 - SZ
 - ZFP
 - Bin-based (similar to QSDG)
 - Histogram-based (same thing but Gaussian distribution)
- } Lossy compression

(Pilgrim) Inter-trace compression



Using ncurses

