Pallas

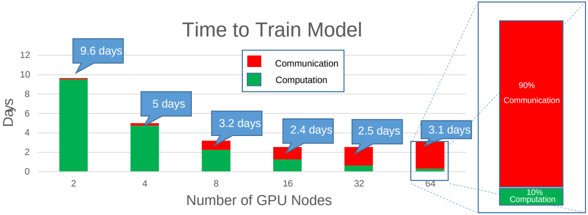
HPC Trace Analysis at scale

Catherine Guelque
Supervisors: Francois Trahay & Valentin Honoré

Télécom SudParis

Scalability issues in HPC

- Load-balancing
- Concurrent access to resources
- Interactions between threads
- Non-negligeable communication times



How to scale, debug and optimize applications on such systems ?

Tracing

Trace → timeline of the execution

Idea: Intercept function calls (MPI, OMP, CUDA) → **Event**

- **Timestamp**
- **Additional Data**: Arguments, callstack, etc.

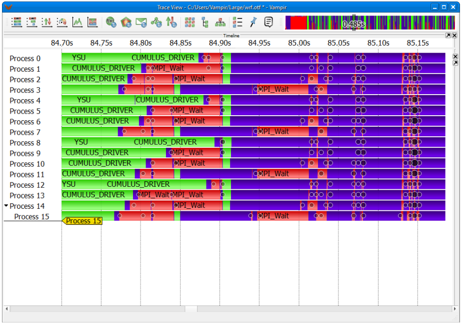


Figure: An OTF2 Trace visualised with Vampir.

Trace analysis

Visualisation aren't scalable (too many threads, memory issue)
Statistical analysis need to be done quick and cheap

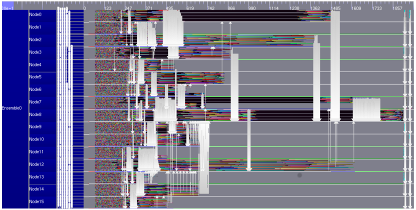


Figure: A sizable trace in ViTE

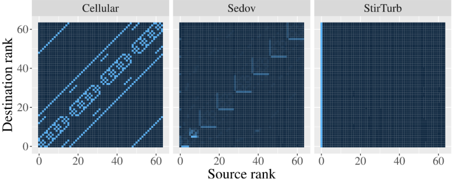


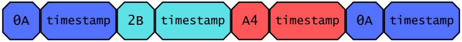
Figure: Communication matrices generated by Pilgrim.

Types of traces

Sequential

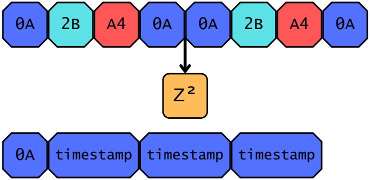
Array of events in chronological order

- Straightforward to read & write
- Redundancy → heavy traces



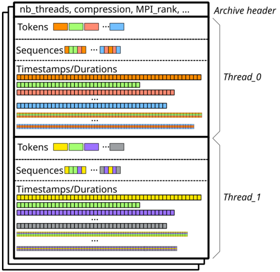
Structural
HPC apps are predictable → include the structure of the program

- Better compression
- More information
- Easier analysis



Pallas

- Structural trace format
- Efficient data retrieval
  - Independent data and metadata loading
  - Parallelisable read and write
- Efficient encoding and compression

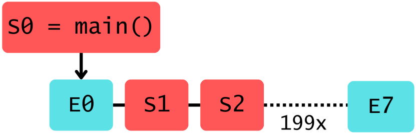


Appendix

General idea

- Events are intercepted and stored as Tokens
- Intercepted function calls create **Blocks** = Hierarchy

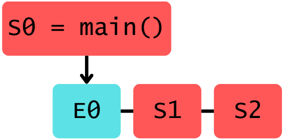
```c
void foo() {
  MPI_Send(...);
  MPI_Recv(...);
}

int main() {
  for (int i = 0; i < 200; i++) {
    foo();
  }
}
```



Structure detection

Each Block has its own buffer of tokens
When a token is added → Basic loop detection algorithm
If repetition of tokens is detected:

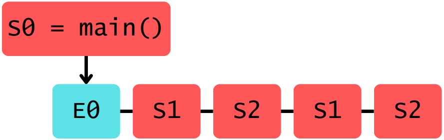
- Check already existing Sequences with hashing function
- Replace repeating Tokens with new Loop token



Structure detection

Each Block has its own buffer of tokens
When a token is added → Basic loop detection algorithm
If repetition of tokens is detected:

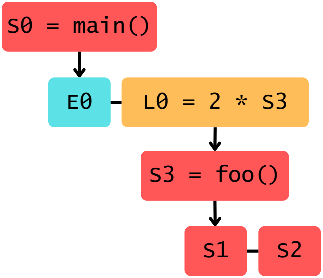
- Check already existing Sequences with hashing function
- Replace repeating Tokens with new Loop token



Structure detection

Each Block has its own buffer of tokens
When a token is added → Basic loop detection algorithm
If repetition of tokens is detected:

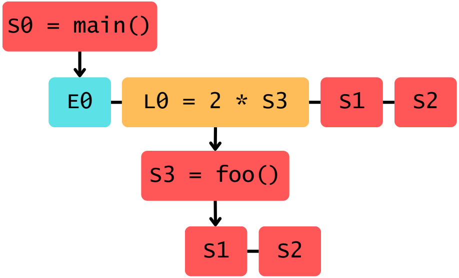
- Check already existing Sequences with hashing function
- Replace repeating Tokens with new Loop token



Structure detection

Each Block has its own buffer of tokens
When a token is added → Basic loop detection algorithm
If repetition of tokens is detected:

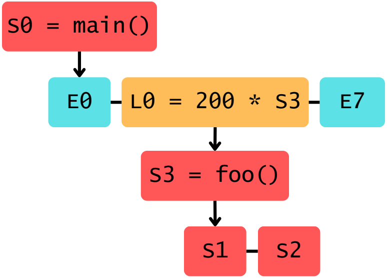
- Check already existing Sequences with hashing function
- Replace repeating Tokens with new Loop token



Structure detection

Each Block has its own buffer of tokens
When a token is added → Basic loop detection algorithm
If repetition of tokens is detected:

- Check already existing Sequences with hashing function
- Replace repeating Tokens with new Loop token



Reading the structure

The structure can be read independently from the timestamps → quick and lightweight overview of the program

```c
void ping_pong() {
  MPI_Send(...);
  MPI_Recv(...);
}

void main() {
  int i;
  for(i = 0; i < 100; i++) {
    ping_pong();
  }
  for(i = 0; i < 10 000; i++) {
    ping_pong();
  }
  MPI_Barrier();
}
```

```
Reading events for thread 0 (P#0T#0):
Tag      Event
S6       E0_S L1 Eb_S
├─E0_S   THREAD_BEGIN()
├┬L1     2 * S5 = L0 S4
│├┬L0    (100, 10_000) * S3 = S1 S2
││├┬S1   E1_E E2_S E3_L
│││├─E1_E        Enter 0 (MPI_Send)
│││├─E2_S        MPI_SEND(dest=1, comm=0, tag=0, len=16)
│││└─E3_L        Leave 0 (MPI_Send)
││└┬S2   E4_E E5_S E6_L
││ ├─E4_E        Enter 1 (MPI_Recv)
││ ├─E5_S        MPI_RECV(src=1, comm=0, tag=0, len=16)
││ └─E6_L        Leave 1 (MPI_Recv)
│└┬S4    E7_E E8_S E9_S Ea_L
│ ├─E7_E         Enter 2 (MPI_Barrier)
│ ├─E8_S         MPI_COLLECTIVE_BEGIN()
│ ├─E9_S         MPI_COLLECTIVE_END(op=0, comm=0, root=-1, sent=0, recved=0)
│ └─Ea_L         Leave 2 (MPI_Barrier)
└─Eb_S   THREAD_END()
```

Reading the timestamps

Durations can be loaded selectively
Timestamps are calculated on the fly using the sequences' durations

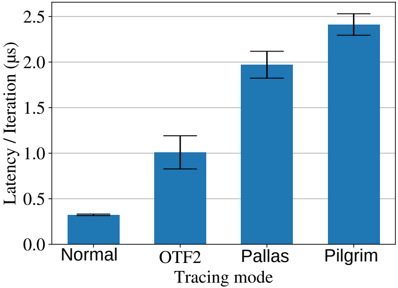
```
void ping_pong() {
  MPI_Send(...);
  MPI_Recv(...);
}

void main() {
  int i;
  for(i = 0; i < 100; i++) {
    ping_pong();
  }
  for(i = 0; i < 10 000; i++) {
    ping_pong();
  }
  MPI_Barrier();
}
```

```
Tag       Duration     Event
S7        2.00004882   E0_S S6 Ed_S
├─E0_S    0.00003671   THREAD_BEGIN()
├─S6      2.00000076   E1_E L1 Ec_L
│ ├─E1_E  0.00001130   Enter 1 (Working)
│ ├─L1    1.99998946   2 * S5 = L0 S4
│ │ ├─S5  0.09816532   E8_E E9_S Ea_S Eb_L
│ │ └─S5  1.90182414   E8_E E9_S Ea_S Eb_L
│ └─Ec_L  0.00001135   Leave 1 (Working)
└─Ed_S    THREAD_END()
```

Experimental parameters

- MPI Ping-Pong, NAS Parallel Benchmarks, AMG, MiniFE, Lulesh & Quicksilver
- 5 iterations, plotted the means with standard deviations
- Tested on
  - OTF2 using EZTrace
  - Pallas using EZTrace
  - Pilgrim

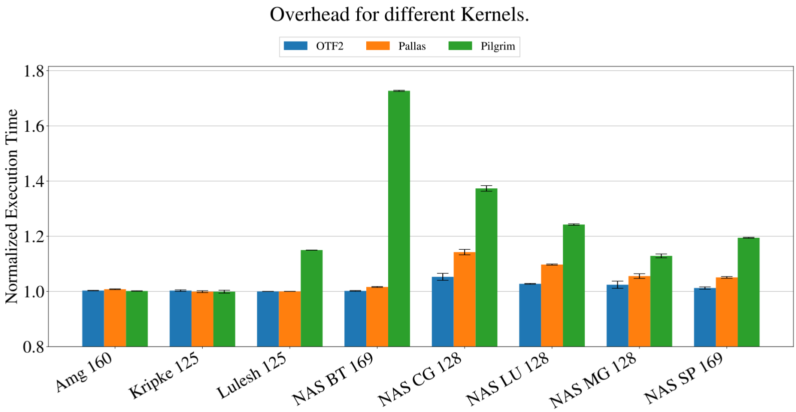
Event recording cost



	Added Latency (ns)	Events/Iter	Latency / Event (ns)
OTF2	686	6	114
Pallas	1 647	6	275
Pilgrim	2 089	2	1 044

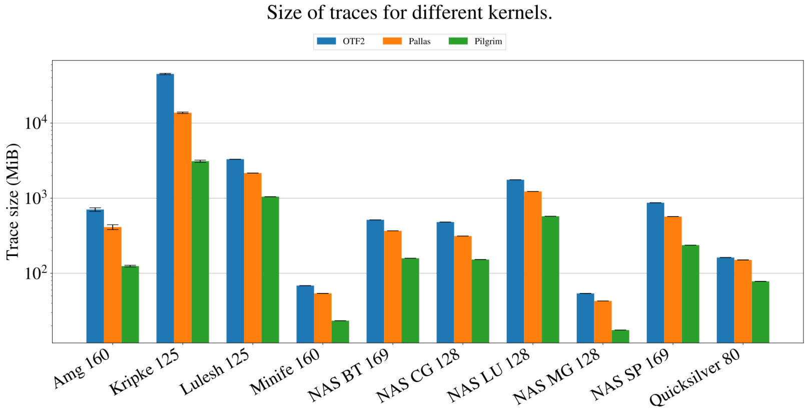
Figure: **Top**: Latency (in µs) of the MPI Ping-Pong program, using OTF2, Pilgrim et Pallas. **Bottom**: Table of the latency per event.

Overhead



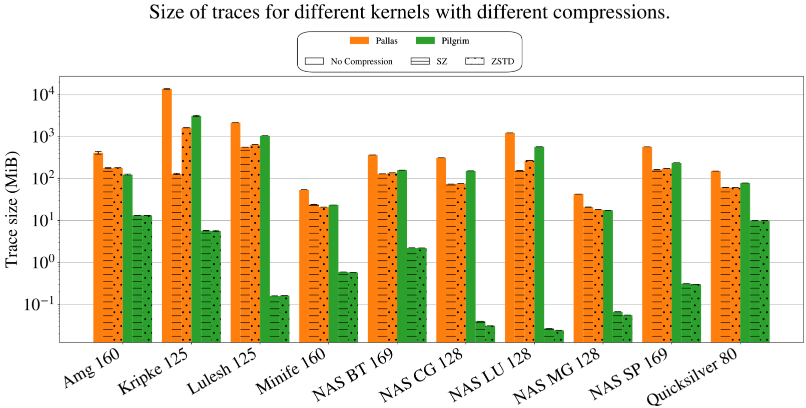
- OTF2, Pallas and Pilgrim almost have the same overhead.

Trace size



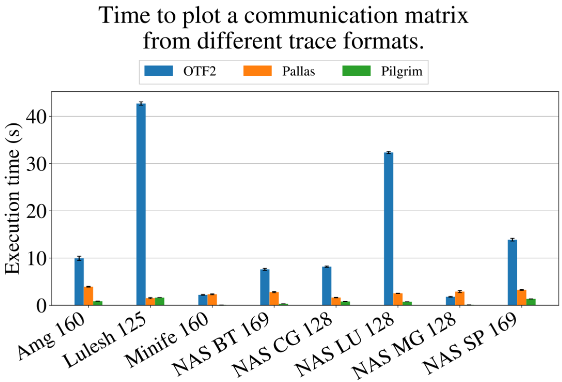
- Without compression, up to 10x difference between OTF2 and Pilgrim
- Possibility: Pilgrim collects less information than EZTrace (Strings, Thread names, etc.)

Trace size



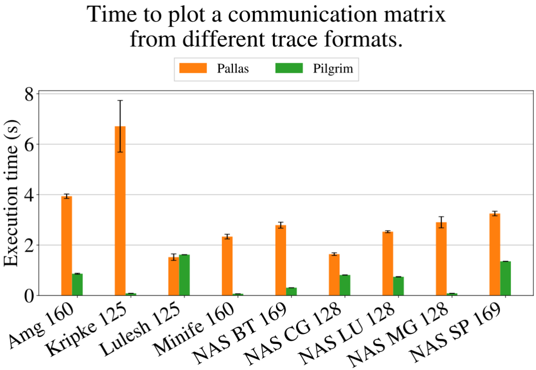
- With compression, Pilgrim shows substantially better results **because all the timestamps are compressed together**.

Analysis speed: Communication Matrix



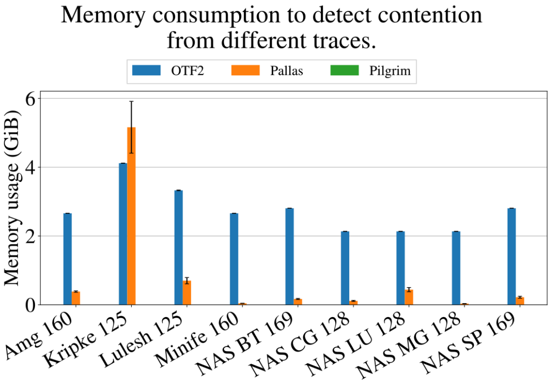
- No represented: Kripke analysis took 450s for OTF2.
- OTF2 (sequential trace) analysis is slow.

Analysis speed: Communication Matrix



- Analysis speed **of the structure** isn't quite as fast as Pilgrim yet.
- Analysis speed uncorrelated with actual trace size (MiniFE was 10-50 times lighter than Lulesh)
- We're supposed to be faster... (Work in Progress)

Memory consumption: Contention detection



- Very low consumption compared to OTF2... Except for Kripke.
- No data for Pilgrim yet (should consume more memory).
- Some debugging of Pallas is still in order.

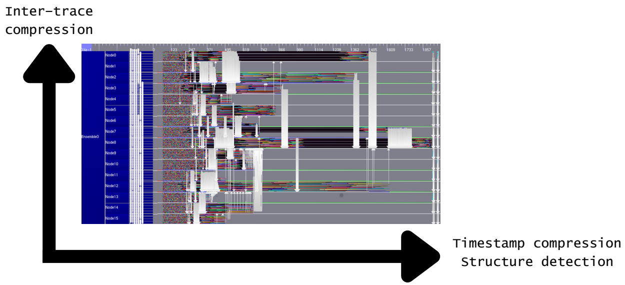
Conclusion

Pallas:

- ✓ Low Overhead
- ✓ Structure detection
- ☞ Compact structures
- ✓ Efficient timestamp storage with compression / encoding
- ✗ Efficient compression
- ✓ Basic scalable analysis
- ☞ Good performance for analysis
- ✓ On demand-trace loading and exploration

Future developments

- Inter-trace compression → "Vertical" scalability
- More efficient compression techniques
- Better trace reading performance
- Tracing non-MPI kernels
- More complex and scalable analysis



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

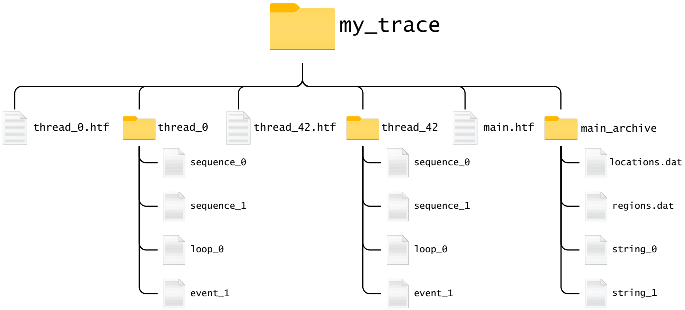
Folder & file structure

Each Event/Loop/Sequence are stored separately:

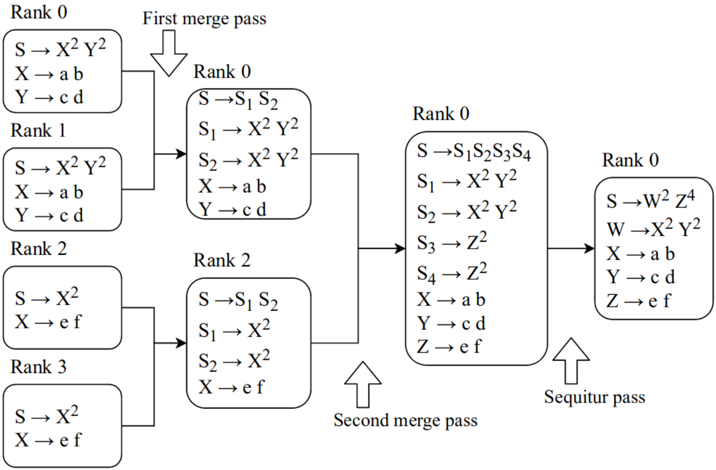
- Loops and runtime variable: stored as-is
- Sequences and Events: stored with array of **durations**

Timestamps are calculated when reading the trace

1 file & folder / thread → (almost) no concurrent writing



(Pilgrim) Inter-trace compression



Pallas_info

```
Archive fffffffe:
    trace_name: ping_pong.htf
    Strings {.nb_strings: 26} :
        // Définitions de 26 Strings

    Regions {.nb_regions: 6} :
        // Définitions de 6 Régions (Sections de code)

    Location_groups {.nb_lg: 2}:
        // Définitions des 2 Location Groups (Processus)

    Locations {.nb_loc: 2}:
        // Définitions des 2 Locations (Fils d'exécutions)

    Threads {.nb_threads: 2}:
        0:    {.archive=0,    .nb_events=12, .nb_sequences=7, .nb_loops=2}
        // 2nd threads, identique au premier

    Archives {.nb_archives: 2}

Thread 0 {.archive: 0}
    Events {.nb_events: 12}
        E0      THREAD_BEGIN()                          {.nb_events: 1}
        E1      Enter 0 (MPI_Send)                      {.nb_events: 1100}
        E2      MPI_SEND(dest=1, comm=0, tag=0, len=16) {.nb_events: 1100}
        E3      Leave 0 (MPI_Send)                      {.nb_events: 1100}
        E4      Enter 1 (MPI_Recv)                      {.nb_events: 1100}
        E5      MPI_RECV(src=1, comm=0, tag=0, len=16)  {.nb_events: 1100}
        E6      Leave 1 (MPI_Recv)                      {.nb_events: 1100}
        E7      Enter 2 (MPI_Barrier)                   {.nb_events: 2}
        E8      MPI_COLLECTIVE_BEGIN()                  {.nb_events: 2}
        E9      MPI_COLLECTIVE_END(op=0, comm=0, root=-1, sent=0, recved=0)
                                                        {.nb_events: 2}
        Ea      Leave 2 (MPI_Barrier)                   {.nb_events: 2}
        Eb      THREAD_END()                            {.nb_events: 1}
    Sequences {.nb_sequences: 7}
        S0      {S6}        S1      {E1, E2, E3}
        S2      {E4, E5, E6}
        S3      {S1, S2}
        S4      {E7, E8, E9, Ea}
        S5      {S3, S4}
        S6      {E0, S5, Eb}
    Loops {.nb_loops: 2, .nb_allocated_loops: 2}
        L0      {.nb_loops: 2, .token: S3, .nb_iterations: [100, 1000]}
        L1      {.nb_loops: 1, .token: S5, .nb_iterations: [2]}
```

Using ncurses

```
ncdu 1.11 ~ Use the arrow keys to navigate, press ? for help
--- /etc ----------------------------------------------------------------------
    3,6 MiB [##########] /brltty
    1,8 MiB [#####     ] /apparmor.d
.   1,1 MiB [###       ] /ssl
  636,0 KiB [#         ] /asciidoc
  524,0 KiB [#         ] /xdg
  496,0 KiB [#         ] /X11
  404,0 KiB [#         ] /init
  392,0 KiB [#         ] /fonts
  344,0 KiB [          ] /ssh
  332,0 KiB [          ] /sane.d
  328,0 KiB [          ] /init.d
  232,0 KiB [          ] /ImageMagick-6
  220,0 KiB [          ] /dbus-1
  168,0 KiB [          ] /lynx
  152,0 KiB [          ] /java-8-openjdk
  144,0 KiB [          ] /console-setup
  140,0 KiB [          ] /default
  136,0 KiB [          ]  ld.so.cache
  120,0 KiB [          ] /pam.d
  112,0 KiB [          ] /systemd
  100,0 KiB [          ] /ppp
 Total disk usage:  14,9 MiB  Apparent size:   9,2 MiB  Items: 3311
```

```
--- Reading bran_quicksilver.htf ---------------------------------
In S6
    100 µs  [                            ] E0_S       THREAD_BEGIN()
   5.53  s  [############################] L1         3 * S5 = L0 S4
    100 µs  [                            ] Eb_S       THREAD_END()

==================================================================
--- Reading bran_quicksilver.htf ---------------------------------
In S6/L1
   525  ms  [###                         ] S5         L0(1.000) S4
   4.00  s  [############################] S5         L0(8.000) S4
   1.00  s  [#######                     ] S5         L0(2.000) S4
```