PALLAS: a generic trace format for large HPC trace analysis

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

PALLAS: a generic trace format for large HPC trace analysis

Catherine Guelque
Télécom SudParis, Samovar, Inria
catherine.guelque@telecom-sudparis.eu
ORCiD: 0009-0002-0123-6958

Valentin Honoré
ENSIIE & Samovar
Evry, France
valentin.honore@ensiie.fr
ORCiD: 0000-0003-1028-5719

Philippe Swartvagher
Bordeaux INP, Inria
Talence, France
philippe.swartvagher@inria.fr
ORCiD: 0000-0003-3786-7364

Gaël Thomas
Inria
gael.thomas@inria.fr
ORCiD: 0000-0002-9444-107

François Trahay
Télécom SudParis, Samovar, Inria
francois.trahay@inria.fr
ORCiD: 0000-0001-7329-1812

***Abstract*—Identifying performance bottlenecks in a parallel application is tedious, especially because it requires analyzing the behaviour of various software components, as bottlenecks may have several causes and symptoms. For example, a load imbalance may cause long MPI waiting times, or contention on disk may degrade the performance of I/O operations.**

Detecting a performance problem means investigating the execution of an application and applying several performance analysis techniques. To do so, one can use a tracing tool to collect information describing the behaviour of the application. At the end of the execution, a trace file in a specific format is available to the application user, which can be used to conduct a complete post-mortem investigation.

Several challenges emerge from the generation and use of traces. Tracing applications may alter the performance of the application, and can create thousands of heavy trace files, especially at a large scale. Most importantly, the post-mortem analysis needs to load these thousands of trace files in memory, and process them. This quickly becomes impractical for large scale applications, as memory gets exhausted and the number of opened files exceeds the system capacity.

In this paper, we propose PALLAS, a generic trace format tailored for conducting various post-mortem performance analysis of traces describing large executions of HPC applications. During the execution of the application, PALLAS collects events and detects their repetitions on-the-fly. When storing the trace to disk, PALLAS groups the data from similar events or groups of events together in order to later speed up trace reading. We demonstrate that the PALLAS online detection of the program structure does not significantly degrade the performance of the applications. Moreover, the PALLAS format allows faster trace analysis compared to other evaluated trace formats. Overall, the PALLAS trace format allows an interactive analysis of a trace that is required when a user investigates a performance problem.

***Index Terms*—performance analysis, trace format**

I. INTRODUCTION

Improving the performance of a parallel application is tricky, because performance issues may have many root causes, from low-level hardware contention to algorithmic problems. Identifying which of these causes is to blame is challenging. Depending on the problem, it requires monitoring several software components, such as MPI [1], the task-based scheduling runtime system [2], or the OpenMP runtime [3]. There are a variety of performance metrics which may reveal the source of a performance problem [4]. Even if a particular issue can be identified with very little data, it is necessary to collect as much performance data as possible, because the component responsible for the performance bottleneck is usually unknown before the post-mortem analysis.

As a result, the performance analysis process is an investigation where one has to try several performance analysis techniques. Each analysis studies some of the collected performance data and may reveal a hint. A hint may guide the application developer to conduct the next performance investigation until they finally identify the cause of a performance problem.

Collecting complete execution traces can help developers investigate. Tracing tools such as Tau [5], Score-P [6], EZ-TRACE [7], or HPCToolKit [8] allow to run the application once and collect evidence that is stored in an execution trace. Then, the developers may conduct a forensic analysis of performance on this trace to find the component or the behaviour that degrades the performance.

Conducting a performance investigation with existing performance analysis tools is, however, costly. Performing a single trace analysis requires loading dozens of GiBs of events from thousands of files on disk, and allocating and processing them, which is resource and time-consuming. For example, as described in Section V, analyzing the performance of an AI model training application running at a medium scale (128 GPUs) requires the use of several analysis, some of which take up to 20 minutes with existing tools. We also had to modify the analysis program to be able to load thousands of files and analyze the trace. Such poor performance and the inability to natively load the trace prevent users from conducting a smooth and interactive investigation of their application performance problems and thus deter them from using said tools.

The difficulty of analyzing a large trace comes from the

fact that, while current tracing tools are optimized to avoid disturbing the profiled application, they do not consider how a trace is used during analysis. The tracing tools do their best to compress the trace [9]–[16] to reduce the I/O traffic and the trace size. However, since these compression techniques do not consider the analysis phase, they only worsen the time required to analyze the trace. As a result, while we can efficiently trace an execution, analyzing the trace itself remains long and difficult.

In this paper, we propose to revisit the way traces are stored to ease and accelerate the post-mortem trace analysis. For that, we observe that analyzing a trace often consists of macro-analysis in which the execution details are initially irrelevant, followed by micro-analysis in which the developer focuses more on the low-level details. Starting from this observation, we propose to design the trace hierarchically. At first level, the tracing tool stores an excerpt of the run. It summarizes the execution by building a structural view of the trace, consisting of a tree of nodes, as illustrated in Figure 1. A leaf of the tree (depicted as circles in the figure) represents an event type (*e.g.* a specific function call), and an internal node (depicted as a triangle or a square in the figure) represents the repetition of the events of its children. Based on this representation, the tracing tool associates macro metrics to the nodes at a second level. This representation is useful for a macro-analysis in which the developer wants to identify the abnormal sequence of events. At a third level, the tree can be flattened to obtain the exact sequence of events, as illustrated in Figure 2, which is how tracing tools usually expose events. Thanks to this, the developer can fine-tune an analysis to an abnormal sequence of events, which allows them to identify a performance bottleneck.

The paper makes the following contributions:

- a new trace format, PALLAS, that detects sequences of events on the fly and stores events in a structured way that allows fast post-mortem analysis;
- a set of ready-to-use analysis programs that leverage the PALLAS trace format, and allow to analyze large traces interactively;
- an extensive set of experiments on various application profiles (MPI, deep-learning), demonstrating that (1) tracing with PALLAS generates an overhead similar to other formats in the literature,
- and (2) PALLAS's associated tools can efficiently explore the trace content to find performance bottlenecks, hence enabling faster investigation for application developers.

The remainder of the paper is organized as follows. Section II presents related work on large-scale performance analysis. We present PALLAS design and implementation in Section III. We evaluate the performance of PALLAS on multiple applications in Section IV. In Section V, we present a case study that demonstrates how PALLAS can help a developer identify a performance problem in a deep-learning application. Finally, Section VI concludes the paper.

II. RELATED WORK

Execution traces of parallel applications can be large, and analyzing them is resource-consuming.

As a result, a common approach targets the reduction of the size of the traces. The Open Trace Format (OTF) [17] can reduce the size of traces by compressing the collected data with ZLib. Such compression methods are agnostic of the data to compress. PALLAS also provides size reduction for the durations of the events, which make up the bulk of a trace, and are well-suited for compression. Reducing the size of MPI traces has been abundantly studied. ScalaTrace [9] records traces of MPI events, and uses an intra- and inter-node compression technique. This allows ScalaTrace to replay the MPI communication patterns of large-scale programs running on hundreds of CPUs. PALLAS also detects event repetitions on a per-process approach, which allows to compress significantly the size of the structure of the application. We do not consider inter-thread compression yet, but envision it in future work.

Multiple implementations rely on Sequitur [10], a grammar-based compression algorithm [11], [12]. For example, PILGRIM encodes MPI functions and their parameters into *terminals*, and searches for rules to define a context-free grammar. PALLAS follows the same general idea, but it uses a more general loop finding algorithm. Even though our repetition detection is less fine-grained, we can reach a more stable performance overhead caused by tracing. Cypress [13] compresses MPI traces using a hybrid static-dynamic method that detects the application control flow graph during a compilation phase and completes this graph at runtime. Compression has also been applied to predict the behaviour of an application [14], or for prefetching I/O [15], [16]. These compression methods can significantly reduce the size of traces. However, they are usually not used for analyzing the performance of applications. They are mostly used for describing the program behaviour [14], to replay traces [9], [12] or to predict the application performance on a given hardware [13].

Reducing the duration of performance analysis has also been widely studied. Scalasca [18] generates OTF2 [19] traces during the execution of the application, and summarizes them into performance reports during the application call to `MPI_Finalize`. Thus, Scalasca uses the application's parallelism to conduct performance analysis. The generated performance reports can then be explored post-mortem. HPCToolkit generates statistical profiles of applications [8]. PerfExplorer [20] or VampirServer [21] use a client/server architecture to offload the processing of large traces on a cluster of machines, and summarized data can be visualized on a client node (typically a laptop). These tools rely on parallelism to speed up performance analysis, which is complementary to our approach. We propose a trace format tailored for fast performance analysis. This format could be used for collecting performance data, and performance analysis tools could parallelize their processing of the recorded data.

Overall, existing performance analysis tools rely on (and

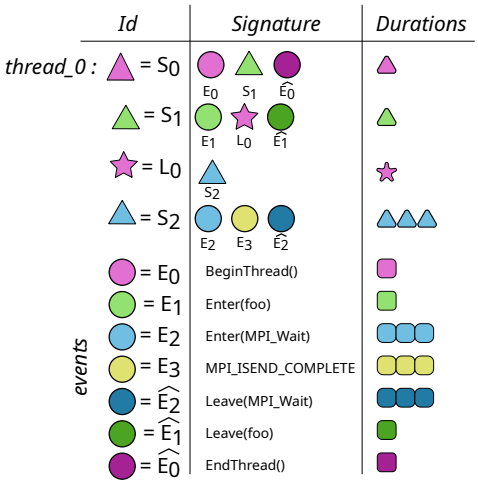

Fig. 1. PALLAS hierarchical representation of a thread trace. The trace starts with sequence S_0 that contains event E_0, then sequence S_1, then event $\hat{E_0}$. Each sequence is composed of a list of events (circles), sequences (triangles), or loops (stars).

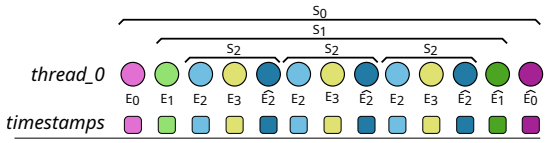

Fig. 2. Representation of a thread trace as a sequence of events. Events are described in Figure 1.

may be limited to) lists of typical problems that are usually found in parallel applications (*e.g.* late sender or load imbalance). We advocate for a post-mortem analysis carried out by the application developer that would collect hints on potential performance problems using existing tools, but would also inspect the statistics on the performance data, or develop their own performance analysis tailored for their application.

III. THE PALLAS TRACE FORMAT

We propose PALLAS [1], a new trace format whose design is driven by its capacity to ease post-mortem analysis and make it more interactive.

During the execution of the application, PALLAS collects events (*e.g.* calls to functions), and processes them on-the-fly to detect the main program constructs (such as functions, or loops). This is done when recording a new event, thus in the same thread triggering the event. The list of events of a thread is represented as a grammar-like structure that is composed of sequences. As illustrated in Figure 1, each sequence contains a list of *tokens* that may correspond to an event, a sequence, or a loop of a token. When writing the trace to disk at the end of the execution, PALLAS stores all of a thread's tokens in a small file that summarizes the thread behaviour, which we call a *structure file*. The rest of the performance data (such as the duration of each occurrence of each token) is stored in a separate file, called a *detail file*. This allows performance analysis tools to later read only small parts of the trace, which leads to a faster analysis.

[1]Available as open-source at https://gitlab.inria.fr/pallas/pallas

A. Detecting the program constructs

Tracing tools usually view traces as a flat list of events, as illustrated in Figure 2. PALLAS works differently and operates by using tokens, stored as integers. A token can represent an event, a sequence of tokens, or the repetition of a token (a *loop*). As such, the execution trace of a thread, which is a list of events, can be represented as a sequence of tokens, each of which can represent sub-sequences, loops, or events, as depicted in Figure 1.

A recorded event is uniquely identified by PALLAS by its signature, which is composed of the event type (inspired by OTF2 records, *e.g.* `Enter`, `Leave`, `MpiSend`, `MpiIsendComplete`, ...), and additional data such as the function name, or some relevant parameters (in the case of an `MPI_Send` for example, we store the receiver and the amount of data sent, as well as the communicator and the tag).This signature is then hashed into a 32-bit integer, which is used to quickly identify events, using a hashmap. Collisions are rare, and handled with a `memcmp` of the two events's signatures. When PALLAS detects a new event signature, a new token T_i^E is created, and associated to it.

PALLAS has two special event types called `Enter` and `Leave` that mark the beginning and the end of functions. They are used for starting and finishing custom sequences. Once the sequence is complete, *i.e.* when reaching a `Leave` event, the sequence's array of tokens is compared to all the known sequences. The sequence is then assigned to an existing token (if the same sequence already exists), or to a new token T_i^S.

Additionally, when adding a token in a sequence, PALLAS searches for loops by comparing the last series of tokens. If a loop $T_a \dots T_n T_a \dots T_n$ is found, PALLAS creates a sequence $T_k^S = T_a \dots T_n$, and a loop $T_i^L = 2 \times T_k^S$, and it replaces the events $T_a \dots T_n T_a, \dots T_n$ with T_i^L in the current sequence. When processing the next tokens, PALLAS may add iterations to the loop. Such a process is illustrated in Figure 3, where a repetition of three tokens is factorized into a single loop token. This loop detection and factorization is made during the execution of the program, and as such can cause some overhead. This is why it can be disabled for irregular applications that do not repeat sequences of events, and a maximum loop length can be specified to limit the impact of the $\mathcal{O}(n^2)$ complexity of the loop detection algorithm.

B. Storage of data in a PALLAS trace.

Many post-mortem analysis need to inspect the trace structure, but do not actually need any precise performance details. To take advantage of this characteristic, PALLAS uses a hierarchical trace format. We use what we call *headers* or *structure files* to store global information that can be quickly accessed. For example, at the root of the trace lies a header file for the whole execution, indicating the number of processes and other trace information. In addition, each process has its

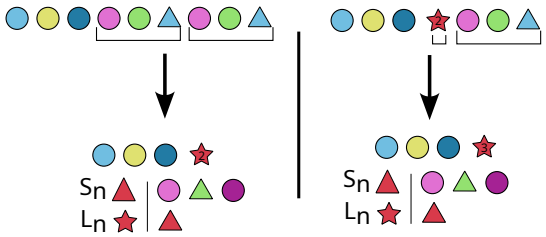

Fig. 3. Factorization of the repetition of a sequence token into a loop token, such as described by Figure 1. On the left, three tokens are repeated twice, and are replaced by a 2-iteration loop. On the right, three similar tokens appear after the loop. These tokens are removed and the iteration count is incremented.

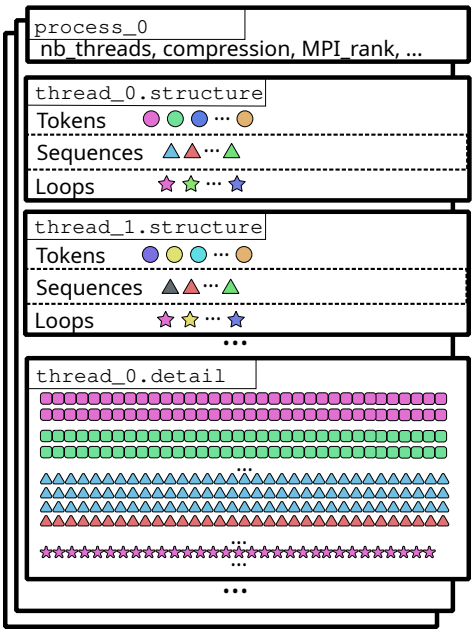

Fig. 4. PALLAS trace format.

own folder with a header file indicating the number of threads and such; finally, each thread has its own subfolder with a header file, storing the grammar (or structure) of the thread execution. These thread structure files also store some basic statistical performance data about the recorded events, such as their minimum, maximum and average duration. They are very light: in our experiments, they seldom weight more than 300 KiB. This allows analysis tools to access this synthetic data without reading and parsing GiB of data they would have no use for.

PALLAS also stores precise performance data, such as the duration of all the occurrences of an event or a sequence, in a separate *detail file*. These files may take several GiB, representing over 90% of the trace size, but they are only needed for analysis that require precise performance data of each individual event. Timestamps of similar events or sequences are stored together, as an array, which allows analysis tools to compare them. The location of the performance data related to a particular event in the detail file is stored in the structure file, which means it can easily be loaded when needed. This allows for some very efficient trace analysis, since only the relevant data can be loaded by the analysis tool. This organization of a PALLAS trace format is shown in Figure 4.

Currently, PALLAS only writes the trace at the end of the execution. Implementing a mechanism to write the trace to disk when buffers are full would take little engineering effort, since we would only need to edit the header files to point to multiple locations in the detail files. The flushed buffers would be written, as always, in the detail files.

Before writing the trace to the disk, PALLAS can apply a compression algorithm to reduce the size of the vectors of duration in the trace. PALLAS supports saving traces without compression, with lossy compression using SZ [22] or ZFP [23], or with lossless compression using ZSTD [24].

C. Tracing applications using the PALLAS format

PALLAS provides a library that can be used by tracing tools to write PALLAS traces. Tracing tools are in charge of intercepting important function calls (*e.g.* calls to MPI functions, CUDA events, etc.) and submitting corresponding events to PALLAS.

To make PALLAS compatible with existing tracing tools, it also implements a subset of the OTF2 API. Thus, a tracing tool that relies on OTF2, such as EZTRACE [7], can use PALLAS as a storage backend without any modification. Similarly, an OTF2-compatible trace visualization tool, such as VITE [25], can use the PALLAS implementation of the OTF2 reading interface to display PALLAS traces. As a generic trace format, PALLAS lets its user create custom events at runtime, through a dedicated event type. It is also fairly simple to modify the PALLAS source code in order to trace such custom events. One only needs to add the corresponding event in an enum, and write a short (less than 20 lines) function to handle that event's type registration.

D. Post-mortem analysis of PALLAS traces

Once a PALLAS trace is generated, a user can investigate the performance issues of their application by running several types of trace analysis. Our PALLAS trace format has been designed to ease the processing of a trace by accelerating a macro-analysis while allowing the analyst to perform micro-analysis when needed. PALLAS comes with a collection of ready-to-use programs allowing to perform post-processing of PALLAS traces. We describe in the following three macro-analysis programs (only accessing structure files), and one micro-analysis (requiring access to detail files). We intend to expand the number of such programs in the future, so that PALLAS can adapt to an even wider range of scenarios.

a) `pallas_print:` basic explorer of a PALLAS trace, providing a concise display of a trace content, with an overview of the structure of the trace. Many options are available to customize the amount of information displayed, for instance to explore more in-depth the structure of the trace (definitions, sequences, etc) or get only the information specific for a given thread (number of events, their duration, etc.).

b) `pallas_contention`: [2] tool to compute a contention score [26] for each sequence of the trace. PALLAS only accesses the structure files of the trace, and does not access the detail files. This metric estimates the impact of contention in a piece of code (typically, a function) on the performance of the application. A contention metric close to 1 indicates that most of the execution time of the thread is due to the variations of the duration of the measured function, and these variations may be caused by contention on a shared resource such as a lock, or a disk. However, if that piece of code always runs in the same amount of time, then there is no contention at play, and the score is close to 0. The metric was originally defined as

$$\frac{\sum_i (d_i - \bar{d})}{thread\_duration}$$

where $\bar{d}$ is the fastest occurrence of the considered function, and d_i is the duration of the i^{th} occurrence of the function. Since PALLAS collects basic statistics about sequence duration, in the structure file, that include their minimum (d_{min}), maximum (d_{max}), and average duration (d_{avg}), we can rewrite this contention metric as:

$$\frac{n \times \sum_i \frac{d_i - \bar{d}}{n}}{thread\_duration} = \frac{n \times (d_{avg} - d_{min})}{thread\_duration}$$

where n is the number of occurrences of the sequence. This transforms n computations into a single one and requires to read only 4 values instead of n timestamps.

c) `pallas_comm_matrix`: [3] tool to generate the communication matrix of an MPI program that describes the quantity of data transferred between pairs of MPI processes, as illustrated by Figure 7. In parallel programming, process placement on nodes is critical for performance. Two processes that have an important volume of communication should be located on the same computing resources to reduce communication overhead [27]. To determine the affinity between processes, obtaining a communication matrix of the processes is a common approach.

The communication matrix is generated by summing the amount of data sent with `MPI_Send` events. However, rather than computing that sum, one can also simply multiply the number of `MPI_Send` by their message size. Since the amount of data sent is one of the parameters used to differentiate `MPI_Send` events in PALLAS, it is stored within the event data and the tool only has to access the trace structure files, without browsing the whole trace files.

d) `pallas_histogram`: [4] tool to compute the distribution of duration of a sequence. Unlike the three previous programs, `pallas_histogram` performs a micro-level analysis, and must access the details associated to sequences and events. It reads the sequence's duration from the detail files and computes their distribution, which can then be used to plot a histogram, as illustrated in Figure 8.

Since these durations are stored as a vector for each sequence, PALLAS reads a vector of elements stored in contiguous memory, which can be fast, even for large traces. As these histograms are only generated for a few particular sequences, PALLAS does not have to load or read the whole trace to compute them, saving both time and resources.

E. Discussion

PALLAS has been developed to provide both the capacity to trace applications and fast performance analysis of the generated traces. In this objective, PALLAS separates the storage of the program structure (that is, the description of sequences/events) and the data associated to the structure (that is, the information relative to the duration of sequences/events). We also store statistics concerning the data in the structure of the trace to allow quick analysis of the different phases of applications for profiling.

We developed a catalog of macro- or micro-analysis programs with PALLAS that leverage the trace format to quickly query application structure and performance data.

As data are grouped by sequences in the structure, PALLAS is able to give efficient access to the associated data when the analysis program requires it. As a result, an application developer can perform several analysis in a matter of seconds, which makes the performance investigation smoother. Additionally, one can complete the list of analysis tools to fit their need, by exploiting the PALLAS trace reading API.

However, using PALLAS has some drawbacks. Since it is not a sequential trace format, the reading of a PALLAS trace adds some complexity inherent to the exploration of the trace structure. The software needs to keep the count of each event previously seen in the trace, and it often has to compute the timestamps (by summing durations), instead of just reading them. This would make PALLAS less suited for reading a whole trace linearly. With the structure detection of the program, PALLAS induces slightly more overhead at runtime than its sequential counterparts (such as OTF2), as we will see in Section IV-B. However, for large-scale traces with thousands of processes, storage using a sequential trace format becomes impractical anyway. Thus, these drawbacks of using PALLAS are balanced by its large-scale usability and better compression.

IV. PERFORMANCE EVALUATION

In this section, we evaluate the performance of the PALLAS trace format on different aspects: impact of tracing on the application performance, size of the generated traces, and execution time of post-mortem analysis tools presented above.

A. Experimental Setup

a) Experimental platform: We run experiments on two types of nodes offered by the French Jean Zay machine: CPU-only nodes with 2xIntel Cascade Lake 6248 with 20 cores each (40 cores per node in total) running at 2.5 GHz, with

[2] Available as open-source at https://gitlab.inria.fr/pallas/analysis-tools/pallas-contention

[3] Available as open-source at https://gitlab.inria.fr/pallas/analysis-tools/pallas-communication-matrix

[4] Available as open-source at https://gitlab.inria.fr/pallas/analysis-tools/pallas_profiling

192 GiB of RAM; and GPU nodes, identical to the CPU-only nodes, but with 4 additional Nvidia Tesla V100 GPUs with 32 GiB of memory per GPU. Hyperthreads are not used during our experiments. All nodes are interconnected with an Intel Omni-Path network 100 GB/s with one NIC per CPU-only node and four NICs per GPU node. We used Intel MPI 2021.9 to compile the programs.

b) Tracing format and setup: We present below the different tracing configurations that we use in our experiments.

- *Vanilla*: baseline, without tracing tools;
- PILGRIM [12]: both a tracer and a trace format; It only intercepts MPI events.
- EZTRACE/OTF2: using EZTRACE [7] for tracing, which records OTF2 traces. EZTRACE can intercept a wide range of events (MPI, OpenMP, CUDA, etc);
- EZTRACE/PALLAS: using EZTRACE [7] for tracing. EZTRACE uses the PALLAS implementation of the OTF2 API to record PALLAS traces.

In our experiments, we only use the MPI module of EZTRACE, in order to compare ourselves fairly with PILGRIM.

When tracing executions with PILGRIM and PALLAS, we also compare the different trace compression each tool supports. Note that the way PILGRIM handles compression leads to some unavoidable crashes due to not being able to allocate a large amount of memory at once. This leads some PILGRIM runs to not generate traces, even though they have correctly traced the rest of the execution.

c) Applications: In our experiments, we evaluate performance metrics of different trace formats on MPI applications. Focusing on MPI applications allows us to include PILGRIM in our study, which is not able to trace other types of paradigms. All the code, parameters, and scripts to reproduce those experiments will be made available in a reproducibility artifact. We consider the following applications:

- **Lulesh**: an application that approximates hydrodynamics equations by solving a Sedov blast problem;
- **Quicksilver**: an application implementing a simplified dynamic Monte Carlo particle transport problem. Quicksilver is known to be an irregular application, with specific communication patterns that evolve during the execution;
- **Kripke**: a simple deterministic particle transport code;
- **NAS Parallel Benchmark**: we use the MPI versions of the following kernels: BT, LU, MG, CG and SP. NAS Parallel Benchmarks are classical applications used to evaluate the performance of supercomputers.

NAS benchmarks are regular kernels, representing classical communication patterns in HPC applications. The three other applications represent different types of kernels (linear algebra, Monte Carlo simulations, etc), with different communication patterns. We ran these applications on 4096 MPI processes, with one MPI process per core. When presenting execution times, we report the average duration over 4 executions, and figures report the standard deviation of those measures.

TABLE I
NUMBER OF EVENTS IN THE TRACE OF THE CONSIDERED APPLICATIONS ON 4096 PROCESSES.
"ABORTED" MEANS THAT WE COULD NOT GENERATE A TRACE.

Benchmark	EZTRACE/OTF2	EZTRACE/PALLAS	PILGRIM
Kripke	12 452 559	12 444 356	433 145 931
Lulesh	17 664 768	17 664 768	ABORTED
NAS BT	72 634 880	72 634 880	23 974 656
NAS LU	11 870 199 900	11 870 199 900	3 956 218 044
NAS MG	254 125 472	254 125 472	84 554 622
NAS SP	7 859 904 512	7 859 904 512	2 353 285 935
NAS CG	1 861 689 344	1 861 689 344	620 564 480
Quicksilver	8 016 217 924	8 016 217 924	ABORTED

B. Tracing Applications with PALLAS

a) Overhead: We first evaluate the impact of tracing on the application performance. Figure 5 reports the execution time of each tracing scenarios for the different applications. This time is extracted from counters within the app, and then normalized by its vanilla execution time. Note that these tracing tools write the trace to the disk at the end of the program. As such, this step is not taken into account in the reported execution time.

We observe that EZTRACE with PALLAS has an overhead in the same order of magnitude as with OTF2, albeit a bit higher. The same could be said for PILGRIM compared to PALLAS, however, it sometimes induces a very important slowdown. This comes from how PILGRIM creates the structure of the program: it generates a context-free grammar for each process to compress and store the sequence of MPI calls made by that process. If a lot of different sequences are detected (a function call with a different signature will be a different event), the size of the grammars keeps increasing, and the cost to detect if a sequence is already in the grammar becomes prohibitive. We observe that PALLAS online detection of the program structure does not significantly degrade the application performance in most cases. In some cases (such as Quicksilver or NAS CG) it appears as though PALLAS actually slightly reduces the execution time. We could not explain such a phenomenon.

b) Trace size: We now evaluate the size of the traces generated in the different scenarios. Table I reports the number of recorded events for each application traced with the different considered formats, over 4096 MPI processes.

We observe that both the OTF2 and the PALLAS formats record and store roughly the same number of events, which is expected as they used the same tracing software, EZTRACE. In most cases, PILGRIM records fewer events than EZTRACE. This is because each MPI call is translated to a single event by PILGRIM, where EZTRACE would generate 3 OTF2 records. For example, invoking the `MPI_Irecv` function would generate `Enter(MPI_Irecv)` (that indicates the beginning of function `MPI_Irecv`), `MpiIrecvRequest` (that indicates a new `Irecv` request), and `Leave(MPI_Irecv)` (that indicates the end of function `MPI_Irecv`). PILGRIM also records calls to `MPI_Comm_rank` and `MPI_Comm_size`, which EZTRACE does not. These two factors explain the small variations of event count between PILGRIM and EZTRACE. In

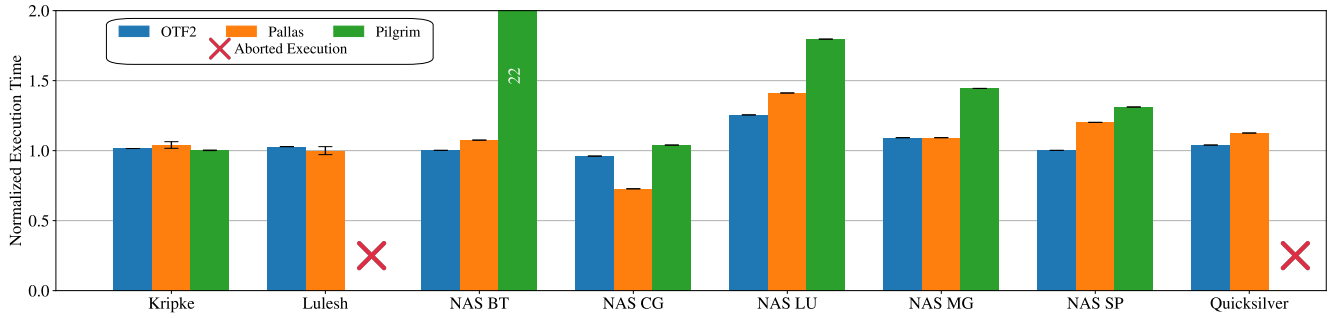

Fig. 5. Execution time of the different tracing scenario, normalized by the vanilla run of the application (no tracing), for the different applications over 4096 MPI processes.

the case of Kripke, PILGRIM records many `MPI_Testany` events (419 million events) due to a busy-wait loop, while EZTRACE does not log this function. Thus, PILGRIM records 433 million events while EZTRACE only records 12 million events. Notes that if we ignore the `MPI_Testany` events, PILGRIM records 14 million events, which is similar to EZTRACE traces.

Figure 6 presents the size of the traces over 4096 MPI processes. For PILGRIM, and EZTRACE/PALLAS, we report the size of trace when using SZ and ZSTD compression, as well as no compression, as discussed in Section III-B. In most cases, OTF2 generates traces with a size an order of magnitude larger than PALLAS and PILGRIM traces, event without compression. This can be explained by the fact that the PALLAS and PILGRIM trace formats store the program events as a grammar-like structure. Without compression, PILGRIM traces are on the same order of magnitude as PALLAS ones, although they tend to be lighter. However, with compression, PILGRIM generates far more compact traces than PALLAS. This comes from the fact that PILGRIM keeps the duration information of all events in a single buffer, which allows for high compression rates. This has two drawbacks: first, it means that PILGRIM needs to allocate a single buffer to store this compressed data when writing the trace, causing the process to crash after the execution when it is unable to do so. This compression (and allocation) is made by each process, and then written on a common file using MPI. Secondly, such a system makes it impossible to read a single duration without decompressing all of the other duration as well. On the contrary, PALLAS groups those data by sequence of events, which limits the compression rates, but eases access when reading. We plan to work on better compression techniques for PALLAS traces in the future.

C. Post-Mortem Performance Analysis

In this section, we evaluate the performance of analysis programs developed with PALLAS, presented in Section III-D.

1) Experimental settings: To be able to compare the performance of the analysis programs with the three trace formats, we have developed the same tools for OTF2 and PILGRIM formats, that mimic their counterpart in PALLAS format. For PILGRIM, only the communication matrix tool was implemented; we could not implement the contention and histogram tools because, to the best of our knowledge, it does not provide an API to access the data associated to each event (that is, duration of events/sequences).

In Table III, we display the number of lines in the source code of the different programs for each trace format. We note that the PALLAS API is notably more compact than the one for the OTF2 format. Especially, the PALLAS API eases the exploration of a trace, either on a by-event or by-sequence approach, making the design of analysis programs more flexible. PILGRIM also offers efficient access to the structure of the application.

In the following, we evaluate the time-to-solution of the different analysis programs, alongside their maximum memory consumption (measured with the GNU `time` command). We use the traces presented in Figure 6, for each format. When available, we analyzed the traces compressed with the ZSTD algorithm. Since it is a lossless algorithm, it does not change the outcome of the analysis.

2) Communication matrix: We natively propose with PALLAS a program, `pallas_comm_matrix`, that generates a file that is used to generate a communication matrix. Figure 7 depicts such a matrix for NAS LU on 512 MPI processes. We made the effort to implement the same program for OTF2 [5] and extended PILGRIM [6] to provide the same feature.

Table II presents the execution time of the analysis programs. Since some traces could not be generated, we denote this with a red cross. ABORTED indicates that the analysis program in the given format took more than 45 min to run. For PALLAS and PILGRIM, we used traces compressed with ZSTD. The completion time includes the cost of decompression when opening these traces.

We observe that OTF2 can generate a communication matrix from a small trace (such as the Lulesh, or Kripke traces) in a few dozens seconds. For larger traces, it becomes impracticable: it can take dozens of minutes to load thousands of files, and to explore the whole trace. For PALLAS,

[5]Available as open-source at https://gitlab.com/eztrace/analysis-tools/otf2-communication-matrix

[6]Available as open-source at https://github.com/valentinhon/pilgrim

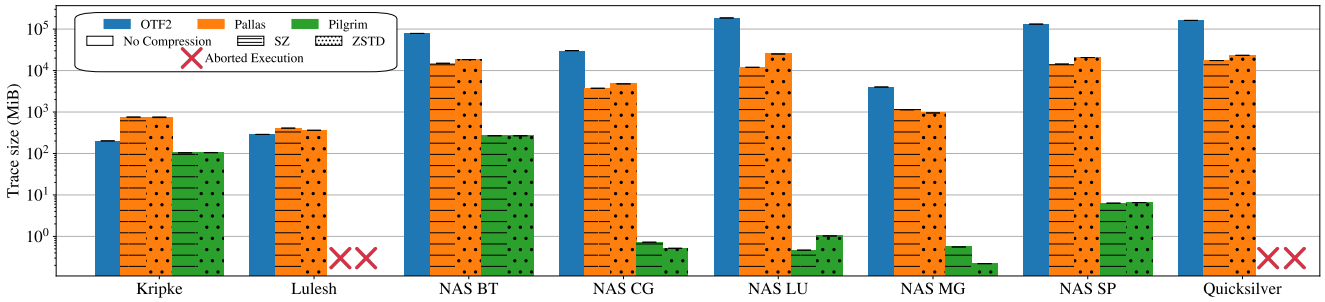

Fig. 6. Comparison of trace size for different trace formats, when tracing the different applications over 4096 MPI processes. A red cross means that the execution of the application aborted when tracing in the considered format.

TABLE II
TRACE SIZE, MEMORY PEAK CONSUMPTION AND EXECUTION TIME TO COMPUTE A COMMUNICATION MATRIX FOR THE DIFFERENT TRACE FORMAT AND DIFFERENT APPLICATIONS. TRACES USED ZSTD COMPRESSION WHEN AVAILABLE.
"ABORTED" MEANS THAT THE EXECUTION TIME WAS MORE THAN 45 MINUTES. X MEANS THAT WE COULD NOT GENERATE A TRACE TO ANALYZE.

Benchmark (4096 MPI processes)	**EZTRACE/OTF2**			**EZTRACE/PALLAS**			**PILGRIM**		
	Trace Size (GiB)	**Memory Peak** (GiB)	**Duration** (sec)	**Trace Size** (GiB, compressed)	**Memory Peak** (GiB)	**Duration** (sec)	**Trace Size** (GiB, compressed)	**Memory Peak** (GiB)	**Duration** (sec)
Kripke	0.226	67.30	34.35	0.83	1.96	18.32	0.14	0.60	3.88
Lulesh	0.343	67.30	35.20	0.46	0.70	10.09	X	X	X
Quicksilver	158.4	ABORTED	ABORTED	22.8	10.21	24.38	X	X	X
NAS BT	76.9	81.46	1913.69	18.0	8.50	23.08	0.31	14.03	47.88
NAS CG	29.6	67.30	739.50	4.7	0.36	9.16	0.03	4.93	50.02
NAS LU	181.7	ABORTED	ABORTED	24.6	0.31	8.23	0.03	0.16	49.41
NAS MG	3.9	67.30	152.10	1.2	0.62	8.45	0.03	0.16	3.20
NAS SP	128.8	ABORTED	ABORTED	20.2	1.68	13.73	0.04	18.51	223.05

TABLE III
NUMBER OF LINES OF CODE OF THE ANALYSIS PROGRAMS PRESENTED IN SECTION III-D FOR DIFFERENT TRACE FORMATS.

Analysis program	**OTF2**	**PALLAS**	**PILGRIM**
Communication matrix	250	210	173
Contention score	336	186	N/A
Histogram	409	257	N/A

generating a communication matrix almost always takes less than an minute, even for large traces (such as NAS LU). This is because PALLAS only reads the thread's structure files to collect information on the number of MPI events, and it does not need to read the event duration, which would be time consuming for large traces.

PILGRIM performs better than OTF2, but worse than PALLAS, in all cases. In some cases, such as for SP, both the analysis time and the memory consumption grow to prohibitive levels.

3) Contention: A parallel application may suffer from contention when accessing a shared resource such as a disk, a NIC, or a lock. To detect functions that may degrade the application performance due to contention, we compute a contention score for each function of each thread. The contention score is inspired by the SCI score [26] and presented in Section III-D.

For PALLAS traces, we use the `pallas_contention`, which reads the threads structure files, and computes the contention score of all the functions based on their duration statistics. We developed a tool that reads an OTF2 trace and computes the contention score of all the functions [7].

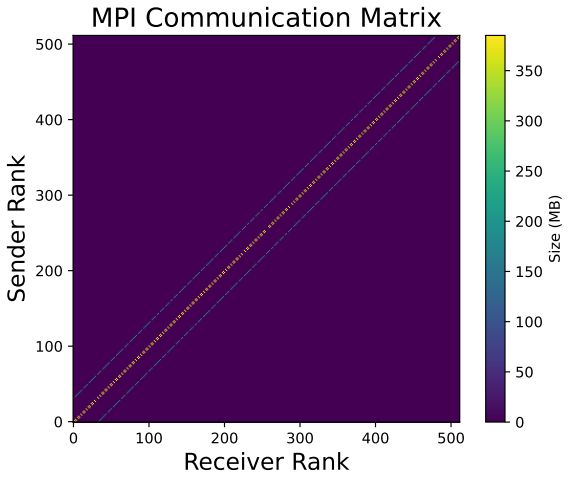

Fig. 7. Communication matrix generated by the `pallas_comm_matrix` analysis program, for NAS LU in MPI version, over on 512 MPI processes.

[7]Available as open-source at https://gitlab.com/eztrace/analysis-tools/otf2-profile

TABLE IV
EXECUTION TIME AND MEMORY PEAK CONSUMPTION TO COMPUTE A CONTENTION SCORE FOR OTF2 AND PALLAS, FOR TRACES OF DIFFERENT APPLICATIONS. TRACES USED ZSTD COMPRESSION WHEN AVAILABLE. ABORTED MEANS THAT THE EXECUTION TIME WAS MORE THAN 45 MINUTES.

Benchmark (4096 MPI processes)	EZTRACE/OTF2			EZTRACE/PALLAS		
	Trace Size (GiB)	Memory Peak (GiB)	Duration (sec)	Trace Size (GiB, compressed)	Memory Peak (GiB)	Duration (sec)
Kripke	0.226	67.19	52.1	0.83	1.72	9.06
Lulesh	0.343	68.2	65.3	0.46	0.52	6.77
Quicksilver	158.4	ABORTED	ABORTED	22.8	9.79	28.8
NAS BT	76.9	ABORTED	ABORTED	18.0	8.23	22.8
NAS CG	29.6	ABORTED	ABORTED	4.7	0.21	5.21
NAS LU	181.7	ABORTED	ABORTED	24.6	0.166	5.27
NAS MG	3.9	67.1	520.0	1.2	0.45	6.37
NAS SP	128.8	ABORTED	ABORTED	20.2	1.52	6.8

TABLE V
EXECUTION TIME AND MEMORY PEAK CONSUMPTION TO GENERATE A HISTOGRAM DATA FILE OF THE MOST FREQUENT FUNCTION FOR OTF2 AND PALLAS, FOR TRACES OF DIFFERENT APPLICATIONS. ABORTED MEANS THAT THE EXECUTION TIME WAS MORE THAN 45 MINUTES.

Benchmark (4096 MPI processes)	EZTRACE/OTF2			EZTRACE/PALLAS		
	Trace Size (GiB)	Memory Peak (GiB)	Duration (sec)	Trace Size (GiB)	Memory Peak (GiB)	Duration (sec)
Kripke	0.226	0.06	75.00	0.83	1.77	24.36
Lulesh	0.343	67.30	35.20	0.46	0.61	10.00
Quicksilver	158.4	ABORTED	ABORTED	22.8	9.8	18.2
NAS BT	76.9	ABORTED	ABORTED	18.0	8.35	17.3
NAS CG	29.6	0.06	1919.64	4.7	0.21	22.75
NAS LU	181.7	ABORTED	ABORTED	24.6	0.16	19.02
NAS MG	3.9	0.06	268.86	1.2	0.45	15.86
NAS SP	128.8	ABORTED	ABORTED	20.2	1.52	17.64

This tool reads the `Enter` and `Leave` events of the OTF2 trace to measure the duration of each function call. As for `pallas_contention`, the contention score of a function is then computed based on the function duration. We could not implement such a tool for PILGRIM traces, because PILGRIM does not provide an API to retrieve the relevant data for each event.

Table IV describes the performance of the contention programs in OTF2 and PALLAS for the different applications. For traces smaller than GiB (Lulesh and Kripke), OTF2 computes the contention score in about a minute. For a 4 GiB trace (NAS MG), it takes 8 minutes to perform the analysis. However, for larger traces, the analysis fails to deliver a solution within 45 minutes (NAS BT, NAS LU, NAS SP, NAS CG, Quicksilver). This is because OTF2 loads and iterates over the entire trace, which becomes too costly as the trace size grows. Thus, performing this analysis during a performance investigation is inpracticable.

PALLAS is able to process all the traces in less than 30 seconds, and most traces are processed in less than 10 seconds. This is because `pallas_contention` leverages the PALLAS trace format, and only reads the threads metadata files to compute the contention scores. As a result, a user could us this analysis program interactively while searching for the software component that degrades the application performance.

4) Histogram: In this analysis, we explore the time to extract the duration of each occurrence of a function, in order to generate an histogram displaying the variations of the function execution time.

We have developed `pallas_histogram` that extracts the duration of each appearance of a set of functions from a PALLAS trace. As presented in Section III-D, this program reads the thread's structure files that contain the duration of each sequence. We implemented a similar analysis program for OTF2 traces [8]. This program reads the `Enter` and `Leave` events in an OTF2 trace in order to compute the functions duration. Both programs generate csv files that contain the list of durations of a function as well as the thread that called this function. A user can then plot the content of these files (*e.g.* with a Python script) based on their needs (*e.g.* grouping threads, or comparing the histogram of different threads), as illustrated in Figure 8. As with the contention analysis experiment, we could not implement such a tool for PILGRIM traces.

Table V reports the execution time of the histogram programs when extracting the duration of the most frequent function of a trace. The smallest OTF2 traces (up to 4 GiB) are processed in less than 5 minutes, but larger traces can take up to or more than 30 minutes (such as NAS CG, which is 30 GiB). Larger traces could not be analyzed within 45 minutes.

`pallas_histogram` can analyze each trace in less than 30 seconds. This is because the program does not need to read the whole trace: it only loads the duration of the sequence to analyze. Since these durations are grouped in the thread's structure files, PALLAS only reads a subset of these files, which is faster than reading all the content of the trace.

5) Discussion: In this section, we evaluated several trace analysis programs that have been developed with PALLAS. Many other tools could be envisioned, we do not pretend to give an exhaustive list of them in this paper. PALLAS offers the possibility to design many different types of analysis programs, going from high-level approaches based on the structure of the trace to more complex ones studying spatio-temporal aspects in a trace.

PALLAS performance makes performance analysis interactive, allowing the user to test various hypotheses and gather clues to help pinpoint the source of the performance problem. By contrast, OTF2 trace analysis works for small traces, but becomes far too time-consuming for larger traces. Furthermore, in the course of these experiments, we had to modify most OTF2 analysis tools in order to be able to read the thousands of OTF2 trace files.

V. CASE STUDY: A DEEP-LEARNING APPLICATION

This case study involves a distributed deep-learning application that trains the ResNet50 model over the ImageNet dataset [28]. The application is written in Python and uses the PyTorch machine-learning library, and the Horovod framework for distributing computation on several workers. In our experiments, Horovod relies on OpenMPI v4.1.5 for communication.

[8] Available as open-source at https://gitlab.com/eztrace/analysis-tools/otf2-histogram

TABLE VI
STATISTICS OF DIFFERENT TRACING TOOLS FOR RESNET

Metric	OTF2	PALLAS
Performance overhead	4.2 %	6.4 %
Number of collected events	428,415,180	429,008,944
Number of files	21,003	42,255
Total trace size	6.1 GiB	1.4 GiB
Time to profile	40.46 s	8.37 s
Time to compute histograms	1285.34 s	144.67 s
Time to estimate contention	40.46 s	7.93 s

We run the application with 128 MPI processes on 32 nodes of the GPU partition presented above. In order to study the performance of the application, we only perform 5 epochs, instead of the 80+ epochs required to reach the best accuracy of the model.

We used three different settings for the application code:

- *Vanilla*: the application runs without tracing;
- OTF2: the application runs with EZTRACE over OTF2;
- PALLAS: the application runs with EZTRACE over PALLAS (using the OTF2 interface), without any compression done to the trace.

We run each configuration at least 6 times and present the average training time.

In this experiment, EZTRACE traces the application calls to MPI functions, POSIXIO functions, and the main Python functions of the training application (`forward`, `backward`, `update`, ...)

Figure 10 presents the average training time we measured for several configurations. Table VI reports statistics on the generated traces and on their analysis.

The *Vanilla* version executes in 626.3 s with a significant variation from one execution to another (min: 578.0 s, max: 724.2 s). The OTF2 version executes in 661 s (min: 597.7 s, max: 721.1 s) and generates a 5.8 GiB trace. The PALLAS version executes in 666.4 s (min: 605.9 s, max: 744.6 s) and generates a 1.4 GiB trace.

Both the OTF2 and PALLAS traces show approximately 10,000 threads created by the 128 processes. Most of the threads (79%) have a duration smaller than 1 second, or only generate a few events (90% of these threads have less than 200 events). We guess that these threads are created by PyTorch, and are mostly sleeping during the program execution. Identifying these threads is straightforward and almost instantaneous with both OTF2 and PALLAS.

The first step of our analysis is to profile the application to determine which functions are the most time-consuming. As reported in Table VI, PALLAS returns the list of all the threads functions and their duration in 8.4 s. Collecting the same information from the OTF2 trace takes 40.5 s. PALLAS performs better because it only reads the structure files without reading the duration of events. This first analysis shows that 8 functions are responsible for most of the execution time.

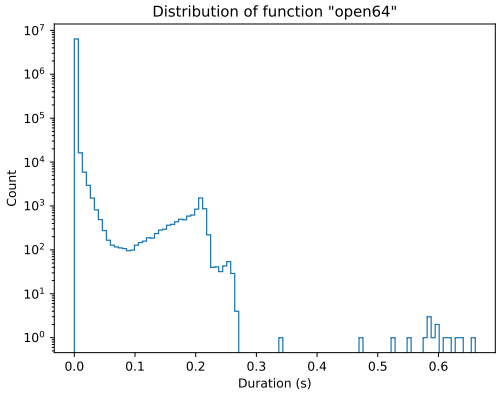

Fig. 8. Histogram of function `open64` in the ResNet experiment.

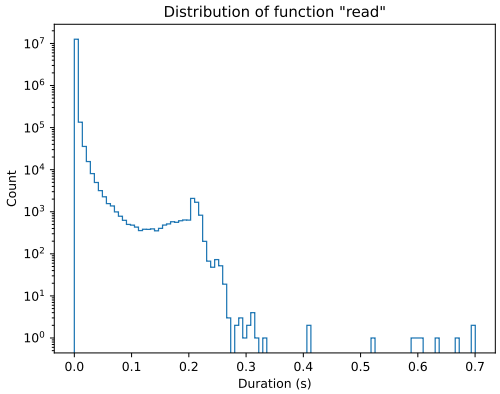

Fig. 9. Histogram of function `read` in the ResNet experiment.

The second step of our analysis is to study the duration of these 8 functions. For each of them, we plot a histogram that shows the distribution of that function's duration. Table VI reports the duration of `pallas_histogram` and its OTF2 counterpart. We do not take into account the performance of the Python script that processes this data to plot a graph. PALLAS extracts the histogram data from the trace in 144 s, while it takes 1285 s to extract similar information from the OTF2 trace. To collect this information, PALLAS only reads the duration of the requested functions, while OTF2 needs to read the whole trace. This second analysis shows that the duration of `open64` and `read` varies, but most durations are in the first bin of the histogram, as shown by figures 8 and 9.

The third step of our analysis is to estimate the contention in `open64` and `read`, and its impact on the application's execution time. We compute the contention score described in Section III-D, and show that `read` may suffer from contention. As reported in Table VI, this takes 40 s with OTF2, and 8 s with PALLAS.

Manual inspection of the application source code reveals that, by default, the workers shuffle the dataset at each epoch. Since the images of the dataset are accessed by the 128 workers through a centralized Lustre filesystem, the random

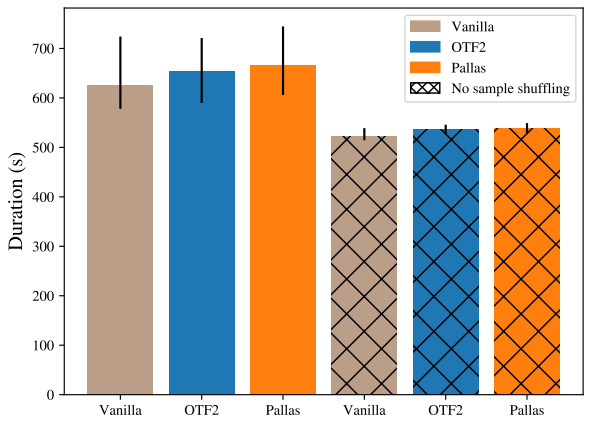

Fig. 10. Execution time of the ResNet application with and without sample shuffling.

access generates I/O contention. This problem was identified by Nguyen et al [29], who show that removing the shuffling phase improves the training execution time without degrading the accuracy of the final model. To validate this hypothesis, we disable sample shuffling and run the experiment again. Figure 10 reports the measured execution time we obtained with and without sample shuffling. We observe that disabling shuffling reduces the training time by 16%, and that performance variation is significantly reduced.

VI. Conclusion

We have presented PALLAS, a trace format that rethinks the way to store a trace in order to ease and accelerate the post-mortem analysis. During the execution of the application, PALLAS detects sequences of events that repeat and groups them. PALLAS separates the storage of the trace structure from the associated performance data so that they can be queried on-demand during the analysis. We offered a catalog of analysis programs with PALLAS to demonstrate the practicality of the PALLAS format in both micro- and macro-level analysis, and have evaluated the performance of PALLAS on a set of applications that were representative of the diversity of HPC applications against other formats used in the field or in the literature. Our experiments show that the online detection of the application structure does not significantly degrade performance, whilst making the analysis of the traces significantly faster and smoother, by allowing both efficient macro- and micro-level analysis.

Future work will focus on enhancing the performance of the PALLAS trace format to provide smoother trace analysis. For longer executions, PALLAS might collect too much information and cause a memory consumption overhead that would make it prohibitive. We will look into detecting such issues at runtime, and provide ways to write the trace during the execution, in order to free some memory. Comparing the structure of multiple threads or processes to study their difference seems to be a promising way to detect performance problems. We will investigate the visualization of PALLAS traces with on-demand exploration of a trace, using the PALLAS API. A complementary approach using spatio-temporal aggregation over computing resources could also be beneficial for macro-analysis of traces.

Software Availability: We endeavor to make our experiments reproducible. A public companion[9] contains the instructions to reproduce our study.

Author Contributions:

- **Catherine GUELQUE**: Software, Conceptualization, Investigation, Validation, Writing - editing, review & experiments.
- **Valentin HONORÉ**: Software, Investigation, Validation, Writing – original draft, Writing – review & editing.
- **Philippe SWARTVAGHER**: Software, Investigation, Writing – review & editing.
- **Gaël THOMAS**: Software, Writing – original draft, Writing – review & editing.
- **François TRAHAY**: Software, Conceptualization, Investigation, Writing – original draft, Writing – review & editing.

Acknowledgment

As part of the "France 2030" initiative, this work has benefited from a national grant managed by the French National Research Agency (Agence Nationale de la Recherche) attributed to the Exa-SofT project of the NumPEx PEPR program, under the reference ANR-22-EXNU-0003.

This project was provided with computing and storage resources by GENCI at IDRIS thanks to the grant 2024-AD011014249 on the supercomputer Jean Zay's V100 & CSL partitions.

References

[1] A. Denis, E. Jeannot, and P. Swartvagher, "Interferences between Communications and Computations in Distributed HPC Systems," in *ICPP 2021 - 50th International Conference on Parallel Processing*, Chicago / Virtual, United States, Aug. 2021.

[2] L. L. Nesi, V. G. Pinto, L. M. Schnorr, and A. Legrand, "Summarizing task-based applications behavior over many nodes through progression clustering," in *2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2023.

[3] A. Daumen, P. Carribault, F. Trahay, and G. Thomas, "ScalOMP: analyzing the Scalability of OpenMP applications," in *IWOMP 2019: 15th International Workshop on OpenMP*. Auckland, New Zealand: Springer, Sep. 2019.

[4] L. Carrington, M. Laurenzano, A. Snavely, R. L. Campbell, and L. P. Davis, "How well can simple metrics represent the performance of HPC applications?" in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005.

[5] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, May 2006.

[6] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[9] https://gitlab.inria.fr/pallas/IPDPS2025_Reproducibility, archived on https://www.softwareheritage.org/ with the ID `swh:1:snp:33e88daab8e2448b1160725f6062a3c95c8d815e`

[7] F. Trahay, F. Rue, M. Faverge, Y. Ishikawa, R. Namyst, and J. Dongarra, "EZTrace: a generic framework for performance analysis," in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2011.

[8] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, 2009.

[9] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, "ScalaTrace: Scalable compression and replay of communication traces for high-performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 8, Aug. 2009.

[10] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *Journal of Artificial Intelligence Research*, vol. 7, Sep. 1997.

[11] S. Krishnamoorthy and K. Agarwal, "Scalable communication trace compression," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. Melbourne, Australia: IEEE, 2010.

[12] C. Wang, Y. Guo, P. Balaji, and M. Snir, "Near-lossless mpi tracing and proxy application autogeneration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 1, Jan. 2023.

[13] J. Zhai, J. Hu, X. Tang, X. Ma, and W. Chen, "Cypress: Combining static and dynamic analysis for top-down communication trace compression," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. New Orleans, USA: IEEE, 2014.

[14] A. Colin, F. Trahay, and D. Conan, "Pythia: an oracle to guide runtime system decisions," in *Cluster 2022*, Heidelberg, Germany, 2022.

[15] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'IO: A Grammar-Based Approach to Spatial and Temporal I/O Patterns Prediction," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[16] L.-M. Nicolas, S. Mimouni, P. Couvée, and J. Boukhobza, "GrIOt: Graph-based Modeling of HPC Application I/O Call Stacks for Predictive Prefetch," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. Denver CO USA: ACM, Nov. 2023.

[17] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the open trace format (otf)," in *Computational Science – ICCS 2006*, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Berlin, Heidelberg: Springer, 2006.

[18] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and computation: Practice and experience*, 2010.

[19] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "Open trace format 2: The next generation of scalable trace formats and support libraries," in *Applications, Tools and Techniques on the Road to Exascale Computing*. IOS Press, 2012.

[20] K. Huck and A. Malony, "Perfexplorer: A performance data mining framework for large-scale parallel computing," in *ACM/IEEE SC 2005 Conference (SC'05)*. Seattle, WA, USA: IEEE, 2005.

[21] M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel, "Developing scalable applications with Vampir, VampirServer and VampirTrace," in *Parallel Computing: Architectures, Algorithms and Applications*, 2007.

[22] S. Di and F. Cappello, "Fast error-bounded lossy HPC data compression with SZ," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016.

[23] P. Lindstrom, "Fixed-Rate Compressed Floating-Point Arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, 08 2014.

[24] Y. Collet and M. Kucherawy, "Zstandard Compression and the 'application/zstd' Media Type," RFC 8878, Feb. 2021.

[25] K. Coulomb, A. Degomme, M. Faverge, and F. Trahay, "An Open-Source Tool-Chain for Performance Analysis," in *Tools for High Performance Computing 2011*. Springer Berlin Heidelberg, 2012.

[26] M. S. M. Bouksiaa, F. Trahay, A. Lescouet, G. Voron, R. Dulong, A. Guermouche, E. Brunet, and G. Thomas, "Using differential execution analysis to identify thread interference," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 12, Dec. 2019.

[27] E. Jeannot and G. Mercier, *Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[28] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 2009.

[29] T. T. Nguyen, F. Trahay, J. Domke, A. Drozd, E. Vatai, J. Liao, M. Wahib, and B. Gerofi, "Why globally re-shuffle? Revisiting data shuffling in large scale deep learning," in *IPDPS 2022: 36th International Parallel & Distributed Processing Symposium*. Lyon (virtual), France: IEEE, May 2022.