Tracing task-based runtime systems: Feedbacks from the StarPU case

Alexandre Denis, Emmanuel Jeannot, Philippe Swartvagher, Samuel Thibault

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

Tracing task-based runtime systems: feedbacks from the StarPU case

Alexandre Denis[1,2] | Emmanuel Jeannot[1,2] | Philippe Swartvagher[3,2,1] | Samuel Thibault[2,1]

[1]Inria Bordeaux – Sud-Ouest, Talence, France
[2]LaBRI, University of Bordeaux, Talence, France
[3]Bordeaux INP, Talence, France

Correspondence
Email: philippe.swartvagher@inria.fr

Summary

Given the complexity of current supercomputers and applications, being able to trace application executions to understand their behaviour is not a luxury. As constraints, tracing systems have to be as little intrusive as possible in the application code and performances, and be precise enough in the collected data.
In this article, we present how works the tracing system used by the task-based runtime system StarPU. We study the different sources of performance overhead coming from the tracing system and how to reduce these overheads. Then, we evaluate the accuracy of distributed traces with different clock synchronization techniques. Finally, we summarize our experiments and conclusions with the lessons we learned to efficiently trace applications, and the list of characteristics each tracing system should feature to be competitive.
The reported experiments and implementation details comprise a feedback of integrating into a task-based runtime system state-of-the-art techniques to efficiently and precisely trace application executions. We highlight the points every application developer or end-user should be aware of to seamlessly integrate a tracing system or just trace application executions.

KEYWORDS:
High-Performance Computing, Task-based Runtime Systems, Tracing Systems, Distributed Clocks, Clock Synchronization

1 | INTRODUCTION

Increasing complexity of supercomputers brings several challenges, among them the difficulties to easily use all the computing power of the machines and to understand what exactly is happening during a program execution. The latter is for instance useful to investigate on performance-related issues.

Tracing systems are usually used to record details of an application execution, to then be able to precisely analyze and understand the execution. However, these tracing systems have a cost: some of them require code instrumentation (*i.e.* modification in the application code) and they can add an important performance overhead, sometimes changing the application behaviour when tracing is enabled, which can be dramatic if tracing the execution makes the developer actually observes different behaviours as the ones he wanted to understand originally. Challenges for tracing systems are thus to be as light as possible, as well as being able to bring insightful information to the application user or developer.

From the application front, task-based runtime systems aim at easily and efficiently exploiting current complex heterogeneous architectures. Since with task-based runtime systems, the application performance can depend as well on the runtime system

behaviour as the application behaviour itself, it is important to be able to understand how each component of the runtime system (scheduling, memory management, communications, data transfers, task executions or performance models) work, and thus have a well-integrated tracing system in the chosen runtime system. Some of them use their own tracing systems, while others rely on existing ones.

In this article, we present some existing challenges to efficiently integrate and use a tracing layer in a task-based runtime system. There are two main contributions: we identify some origins of performance overheads caused by tracing systems; and present the clock synchronization issue for precise traces of distributed executions. The goal is also to make users more aware of the possible pitfalls when tracing application executions.

The rest of the paper is organized as follows: Section 2 explains how generic tracing systems and task-based runtime systems work, Section 3 emphasizes on how the tracing system works with StarPU. Sections 4 and 5 focus respectively on the performance overhead caused by tracing and the challenge of using well-synchronized clocks to trace distributed executions. The last Sections 6 and 7 respectively gathers the lessons learned from this study and concludes the article.

2 | BACKGROUND: TRACING AND TASK-BASED RUNTIME SYSTEMS

This section presents task-based runtime systems, then tracing systems, how they work and the problems they are facing, and finally what the special requirements of task-based runtime systems are regarding tracing systems.

2.1 | Task-based runtime systems

With the increasing complexity of supercomputers (different heterogeneous multicore computing units, hierarchical memory, network), exploiting computing nodes at their full computational power is becoming more and more challenging. To tackle this issue, the *task-based* programming model has emerged. This model describes applications as a task graph (a DAG: *Directed Acyclic Graph*): the application is decomposed into small sub-routines called *tasks*; these tasks are edges of the task graph and vertices represent the data dependencies between the tasks. Task dependencies can also represent inter-process communications of data or synchronization. Once the application developer has written this decomposition, the runtime system executes the task graph by (1) respecting the dependencies of the graph, (2) transmitting the data required by the tasks to the memory node of the executing computing unit and (3) scheduling the tasks among the available resources. When such runtime system works on distributed memory, it also handles data transfers between hosts by exchanging messages (*e.g.* with an MPI library). Task-based runtime systems is a good solution to easily and efficiently exploit heterogeneous architectures (with CPU, GPU and other accelerators) and for applications with irregular patterns, since these runtime systems tend to avoid any synchronizations.

In this work, we use the task-based runtime system StarPU [1]. StarPU lets HPC applications submit a sequential flow of tasks, it infers data dependencies between tasks from that flow, and it schedules tasks concurrently while enforcing these dependency constraints. For each targeted architecture (for instance: CPU, GPU or FPGA), the application developer can provide implementations of the task to be executed by the computing unit. According to the available task implementations, the scheduler can decide, given a scheduling policy, which computing unit (called *worker* in StarPU's jargon) will execute which task, in which order. In its distributed version [2], the application gives an initial data distribution on the participating nodes, and every node submits the same flow of tasks to its local StarPU instance. Each StarPU instance then infers whether to execute a task or not from the piece of data the task writes to, and generates necessary send and receive communications to serve inter-instance data dependencies. This distributed execution model does not involve any master node or synchronization. Instead, all instances are implicitly coordinated by running the same state machine from the sequential task flow.

Since linear algebra is an important area where task-based runtime systems can bring performance improvements, the performance of different features of task-based runtime systems is usually measured by executing linear algebra algorithms. In our case, we use Chameleon [3], a dense linear algebra library built on top of StarPU, providing a set of tools to measure performance of classic linear algebra algorithms. We especially execute the Cholesky decomposition, an algorithm exhibiting a lot of interesting characteristics to exploit StarPU's features (several types of tasks with different behaviours on CPU or GPU, good parallelism, communication pattern).

2.2 | Generic tracing systems

Development of runtime systems and applications includes being able to trace their executions, especially to fix bugs or understand and improve performance, by having an overview of what is precisely happening during executions. Here, we focus on *offline* (or *post-mortem*) analysis: execute the application by recording a set of events describing the application behaviour. When the application terminates, files containing the execution *trace* are saved and can be exploited by tools dedicated to trace analysis. *Online* performance monitoring systems can have similar constraints than *offline* tools (*e.g.* minimal overhead on application performance), but can also present different challenges (for instance: quickly centralizing and analyzing collected information to take a decision)[4].

2.2.1 | The tracing workflow

The tracing workflow can be decomposed in several steps, each coming with their set of problematic and solutions:

1. Collecting information from application executions. This can be achieved mainly by manually putting probes into the code of the component to be traced (method called *instrumentation*) or by wrapping function calls to let the trace system catch them;

2. Storing collected information. Trace systems have to timestamp all events and save all collected data in a persistent format to make the trace data available to the user for the *post-mortem* analysis. Data has to be stored in a coherent format (keeping the chronological order of events and being able to store all possible kinds of additional data for each event), preferably minimizing the size of the trace files;

3. Converting raw trace files to more readable file formats. Because of the constraints on the previous step, the raw trace files are usually not directly exploitable (only the tool that generated the raw file can read it), and need some processing to be read and analyzed by other tools;

4. Analyzing the trace files. The converted files during the previous step can be read by tools to visualize the execution timeline and to highlight the performance bottlenecks and hotspots, for instance.

A large set of tools dedicated to the tracing workflow is available. Some tools focus only on a subset of the enumerated steps, while others take care of the whole workflow. For instance, software libraries can only collect information about application execution: by handling event probes added in the application code (like FxT[5] and LiTL[6]); or by wrapping function calls (like EZTRACE[7]): they store events when functions are entered and left. Libraries using the latter method can track all function calls (this is the case of PARLOT[8]) or only a set of defined functions, for instance functions of the MPI standard (PILGRIM[9] is one of such tool).

More complex tools such as TAU[10] and OPENSPEEDSHOP[11] execute the application to trace, process all the collected data and give information back to the user about the execution. Trace file formats resulting from a traced execution are usually a raw format, understandable only by the library which generated the file. However, once converted, the files depicting the execution can be in more common file formats, like PAJÉ[12] or OTF2[13], to be read by tools to view and analyze the trace, like VAMPIR[14], PARAVER[15], SCALASCA[16] or VITE[17]. Usually, tools focusing on a particular step of the tracing workflow are linked to specific tools focused on other steps. Some collaborations tend to reinforce these affinities, like SCORE-P[18], a joint performance measurement environment gathering, among others, TAU, SCALASCA and VAMPIR. All these tools can be used to trace any kind of application, even if they tend to focus on parallel applications.

2.2.2 | Tracing distributed applications: synchronizing clocks

With distributed executions, usually each process is traced locally, generating one trace file per process. A subsequent conversion step is then in charge of merging the trace files to generate a single exploitable trace file describing the behaviour of the whole application execution.

The main concern with distributed traces is the clock synchronization between nodes: each node usually has a different clock origin. Since each process uses its local clock to timestamp events stored in the trace file, clocks have to be synchronized between the different nodes. Even worse, clocks of different nodes can have different drifts, causing a single clock synchronization not to be accurate enough after some elapsed time.

In practice, badly synchronized clocks can break temporal order of events (the best example is a communication between two nodes appearing in the trace as being received before it was sent) and/or distort the durations of actions involving several nodes (a communication can for instance look faster than in the reality).

These phenomena are already well-covered in the literature. Among many works, BECKER *et al.* explain[19] how nonconstant clock drifts, caused for instance by processor frequency variations, can have a severe impact on distributed clock synchronization. They show that *post-mortem* linear interpolation of clock drift based on clock synchronizations before and after application execution is not enough to compensate for these clock variations, especially on long runs (after several minutes). JONES *et al.* statistically evaluate[20] the accuracy of time synchronization on several leadership class supercomputers in 2016, and report their clock synchronization is not as good as expected, for such top-world supercomputers.

There is no straightforward solution to synchronize distributed clocks, which satisfies all requirements: accurate, fast to initialize and to access, scalable with the number of processes, precise enough for a defined amount of time and without overhead for the application using this kind of clock. Moreover, many factors may influence the accuracy of synchronized clocks, from hardware characteristics (processor frequencies, network performance or computing load) to software features (algorithmic complexity, for instance).

The distributed clock synchronization problem is discussed in detail in Section 5.

2.3 | Tracing systems and task-based runtime systems

Even if task-based runtime systems can be traced and analyzed with existing tools, these tools are widely used for more classic applications, which are usually more regular (or even based on the *Bulk Synchronous Parallel* model), and can miss links between events reflecting the internal working of task-based runtime systems. For instance, one of the key components of such runtime systems is the scheduler, orchestrating the DAG execution onto the computing units: tracing its behaviour to control and understand its decisions requires to collect and visualize particular information, such as the number and types of tasks ready to be executed, which memory node the data buffers these tasks will use are on, or what the current status of each computing unit is. Moreover, since all the program execution relies on a DAG, saving this graph, *e.g.* all information about the tasks and the dependencies between them, is also important to understand the application structure and how the runtime system deals with it.

To trace its executions, STARPU relies sinces its beginning on FXT: its internal code contains probes handled by this tracing library. Execution traces can then be exploited with several tools or manually, as discussed in the next section. Regarding other task-based runtime systems, OMPSS[21] relies on the EXTRAE[22] library to generate traces, browsable by the PARAVER trace explorer, and PARSEC[23] uses an internal system to record events and provide a set of tools to convert resulting trace files in more convenient file formats, such as PAJÉ.

2.4 | Contributions

This paper presents some challenges and solutions while integrating and using a tracing system. It makes the following contributions:

1. A presentation of three sources of performance overhead caused by tracing systems. For each source of overhead, we measure the performance penalty and propose solutions to reduce it;
2. An empirical evaluation of different distributed clock synchronization methods, along with implementation details to compute clock offsets between nodes;
3. A discussion from the different elements learned in the following sections about the method to efficiently trace applications and which requirements has to fit a generic competitive tracing system.

The whole study has been made within the context of the STARPU task-based runtime system. Experiences were made on three different machines. `bora` is composed of nodes with dual INTEL Xeon Gold 6240 at 2.6 GHz with 36 cores and 192 GB RAM, and equipped with INTEL OMNI-PATH 100 series network. `peabody` is a dual INTEL Xeon Gold 6230R at 2.1 GHz with 52 cores and 32 GB RAM. `zonda` is composed of nodes with dual AMD EPYC 7452 at 2.35 GHz with 64 cores and 256 GB RAM, and equipped with 10 Gb/s Ethernet network.

3 | TRACING STARPU'S BEHAVIOUR

The large number of concepts specific to task-based runtime systems (tasks, dependencies, memory transfer, scheduling) shows how complex the work of such runtime systems can be. Thus, it is important to be able to precisely analyze the runtime system behaviour, to check if it works as expected and to detect and investigate performance issues. Each previously enumerated concept can give valuable and ample information about application execution. A method to retrieve all this information is to record them during the application execution and exploit them later, in a *post-mortem* analysis.

To better understand following sections, this one explains more in depth how the trace gathering works within STARPU and how the collected data can then be exploited.

3.1 | Trace gathering

The big picture to explain STARPU's tracing mechanism is that the internal code of STARPU is riddled with probes to describe what the runtime system is about to do (for instance: launch a task, allocate some memory or unlock dependencies) or state changes that just happened (end of a task or reception of a network communication). These probes are instructions to save an event with a timestamp and additional provided information. These events are then stored in a file, to be analyzed later.

Let us consider the example of pushing a task to workers: all data dependencies of this task have been fulfilled, the task is ready to be executed by a worker. The function in charge of this action begins as follows:

```
int _starpu_push_task_to_workers(struct starpu_task *task)
{
    _STARPU_TRACE_JOB_PUSH(task, task->priority);

    // ... actually push the task to computing units
```

`_STARPU_TRACE_JOB_PUSH` will generate an event representing the push of the task given as a parameter, with the given priority. In fact, it is a preprocessor macro that checks whether tracing is enabled and then calls the tracing library to store the event.

STARPU relies on a third-party library, FXT, to record and store events. FXT is in charge of collecting events, possibly filtering them, timestamping them and saving them in a raw file. Then, FXT is also used to read the trace file and get all event information: timestamp, event type, thread ID and additional given information (in the previous example: the task and its priority).

Internally, FXT allocates a buffer to temporarily store events, before flushing this buffer to a disk, in the trace file. The buffer is flushed when it is full or when the application terminates. During a flush, other threads can continue to record events, thanks to a double-buffering system.

For distributed executions, a raw trace file per STARPU process is created.

In STARPU, all possible events belong to a category, for instance:

- `TASK`: task information: name, color, submission time, dependencies, number, throttling, *etc*;
- `WORKER`: computing unit activity: start and end of task execution, sleep, memory transfer to execute tasks, *etc*;
- `DSM`: all memory management made by STARPU: allocation, release, transfers between memory node, *etc*;
- `SCHED`: scheduler activity: new task to schedule, scheduled task, work stealing, *etc*;

At execution time, users can select which event categories they want to be recorded.

This tight integration of FXT in STARPU allows to capture in traces all relevant events and information about the internal state of the runtime system: for instance, this allows to understand why the scheduler took the observed decision. Using another tracing tool which requires instrumentation of the STARPU code would mean replace all tracing instructions in STARPU code with instructions of the new tool. Tracing tools, such as EZTRACE, which wrap function calls to record input and output times of function calls would miss many information about the internal working of the runtime system. Moreover, linking together recorded events to understand causes and consequences of each event is more difficult and requires more post-processing; while with tracing systems using code instrumentation, event probes can store all required information to ease the post-processing. Finally, the goal is not to focus on specific tools and libraries, but to present general problems all tracing systems can be facing.

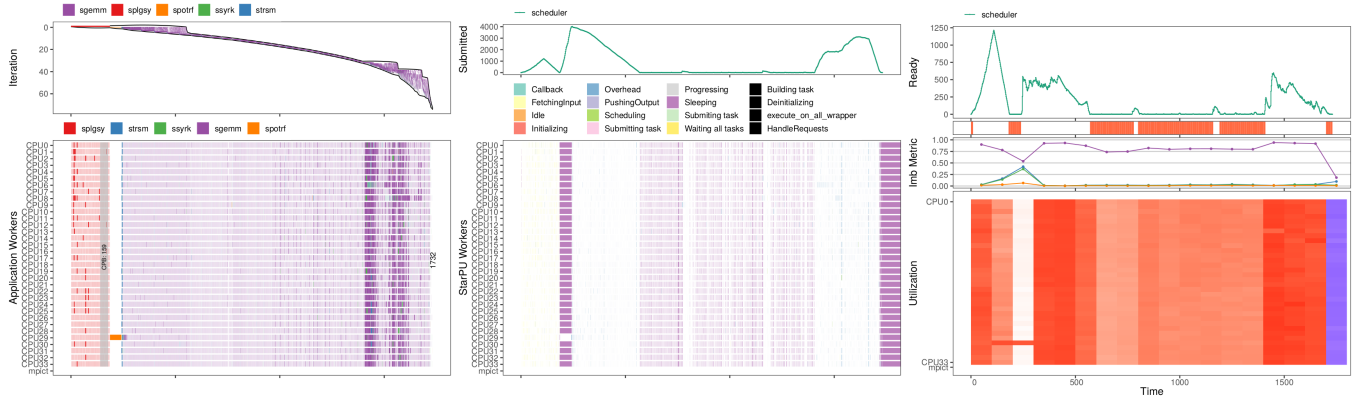

FIGURE 1 Example of visualization of an execution of the CHOLESKY algorithm with the STARVZ framework.

3.2 | Trace exploitation

Once the application execution has been traced, a raw trace file per STARPU process is left to the user for *post-mortem* analysis. Since these files are understandable only for FXT, STARPU provides the tool `starpu_fxt_tool` which reads the trace files, and transforms them into files with a more convenient format, for instance:

- `paje.trace`: the PAJÉ format stores timestamped events to describe application behaviour;
- Several `rec` files (a format similar to *Comma-Separated Values*) listing all communications, tasks and data buffers, with their respective characteristics;
- `dag.dot`: a `DOT` file representing the task graph of the application, executed by STARPU.

These different processed files can be exploited in different ways:

- The VITE software can be used to display the GANTT diagram described in the `paje.trace` file: it will statically represent along a timeline the activity driven by the runtime system: especially task executions by workers, memory and network data transfers. An example of the representation of a STARPU application by VITE is given by Figure 3.
- STARVZ[24] is an R framework, useful to manipulate data from the trace and to easily make all sorts of plots about information stored in the trace file. Figure 1 is an example of basic visualization rendered with STARVZ, where the different plots represent: the parallelization of CHOLESKY iterations, the task executions by workers, the number of submitted tasks, the worker status, the number of ready tasks, a metric representing work imbalance and worker utilization.
- Users can manually parse files generated by `starpu_fxt_tool` to produce their own analysis and plots to represent metrics they are interested in.

4 | REDUCING IMPACT ON PERFORMANCE

Tracing applications implies executing instructions for the original application processing, but also additional instructions to record the events. These additional instructions can add a performance overhead and thus reduce the application performance or, even worse, change the application behaviour. This section presents three sources of overhead caused by the trace recording and proposes solutions to reduce these overheads.

Regardless the considered runtime system or the tracing tool, many factors linked together can influence the performance overhead introduced by a tracing system, starting from the task granularity (the duration of each task): smaller tasks will generate more events (the runtime system will have to deal with of more tasks); threads generating more events will create contention when accessing internal structures of the tracing library; and more events means that event buffers will be full more quickly, requiring to flush them on the disk. The goal here is not to measure the overhead introduced specifically by FXT with STARPU applications, but enumerate the possible *sources* of overhead when tracing applications. Being aware of these sources of overhead can help to better read and analyze recorded traces.

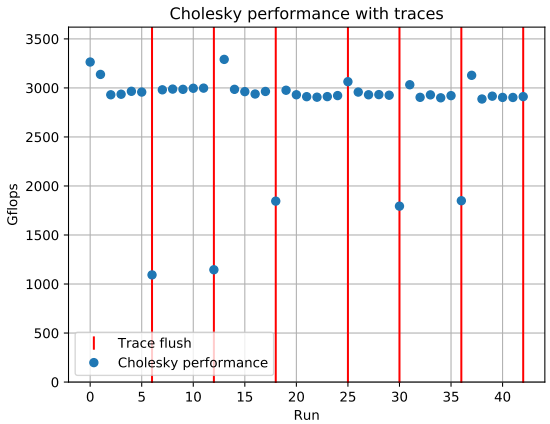

FIGURE 2 Impact of buffer flush on application performance, on a `bora` node.

4.1 | Avoid writing traces on the disk during execution

As mentioned earlier, FxT flushes its event buffer on the disk when it is full. FxT will notice the buffer is full when it will try to record a new event: if the buffer is full, writing the buffer in the file will increase the duration of the probe routine, as much as the necessary time to flush the buffer. This can affect application performance, if it happens during application execution, on a critical path.

We made an experiment to evaluate this possible source of tracing overhead on performance of applications. To generate a lot of events and observe how trace buffer flushes reflect in application performance, we execute several times the same Cholesky decomposition of a matrix of size $24\,000 \times 24\,000$ and plot the performance of each run. At the end, the trace file size is 7.1 GB while the trace buffer size is 1024 MB (*i.e.* flushes occurred during application execution). The trace file was recorded on a BeeGFS parallel filesystem. Results of this experiment are depicted on Figure 2: blue dots represent application performance in Gflops and runs during which a flush of the trace buffer occurred are highlighted with a vertical red line. When runs are not disturbed by a flush, application performance is around 3 Tflops (small variations may be caused by processor frequency variations, to avoid overheating). When a flush occurs during application execution, performance can be severely reduced (1.1 Tflops for runs 6 and 12, 1.8 Tflops for runs 18, 30 and 36) or not (3 Tflops for run 25).

Indeed, the impact of a trace buffer flush on the disk depends on when (and where in the StarPU's code) it happens. Figure 3 represents the Gantt chart of several executions of a Cholesky decomposition (here the size of the trace buffer was 512 MB and the resulting trace file weights 1.7 GB). The different executions are separated by the vertical white dashed lines and red areas represent idle computing units. Trace buffer flushes occurring at different times lead to different situations, highlighted in Figure 3:

- **A**: flush occurred during *overhead* (somewhere in StarPU's code, but not in a specific section): it did not disturb the application too much, because other workers were able to execute tasks and make the application progress;
- **B**: flush occurred during *progressing* (a memory transfer): in this situation, no computing unit was able to execute other tasks, because a lock was taken, preventing StarPU from launching tasks on other workers;
- **C**: the flush occurred during a task, other workers were able to work as long as the result of the blocked task was not necessary to process remaining tasks.

One way to avoid troubles caused by trace buffer flushes during critical moments is to be able to set the size of the trace buffer. When users execute the application a first time, they are warned for each flush occurring during the execution; at the end of execution, the user can look at the size of the trace file to have a rough idea of the required size of the trace buffer to avoid flushes during execution. Then, users execute the application again, but with specifying the size of the trace buffer. Figure 4 presents performance with a trace buffer of 8192 MB (as said previously, the trace file for this experiment has a size of 7.1 GB). There is no outliers and the remaining small variations are probably caused by processor frequency variations.

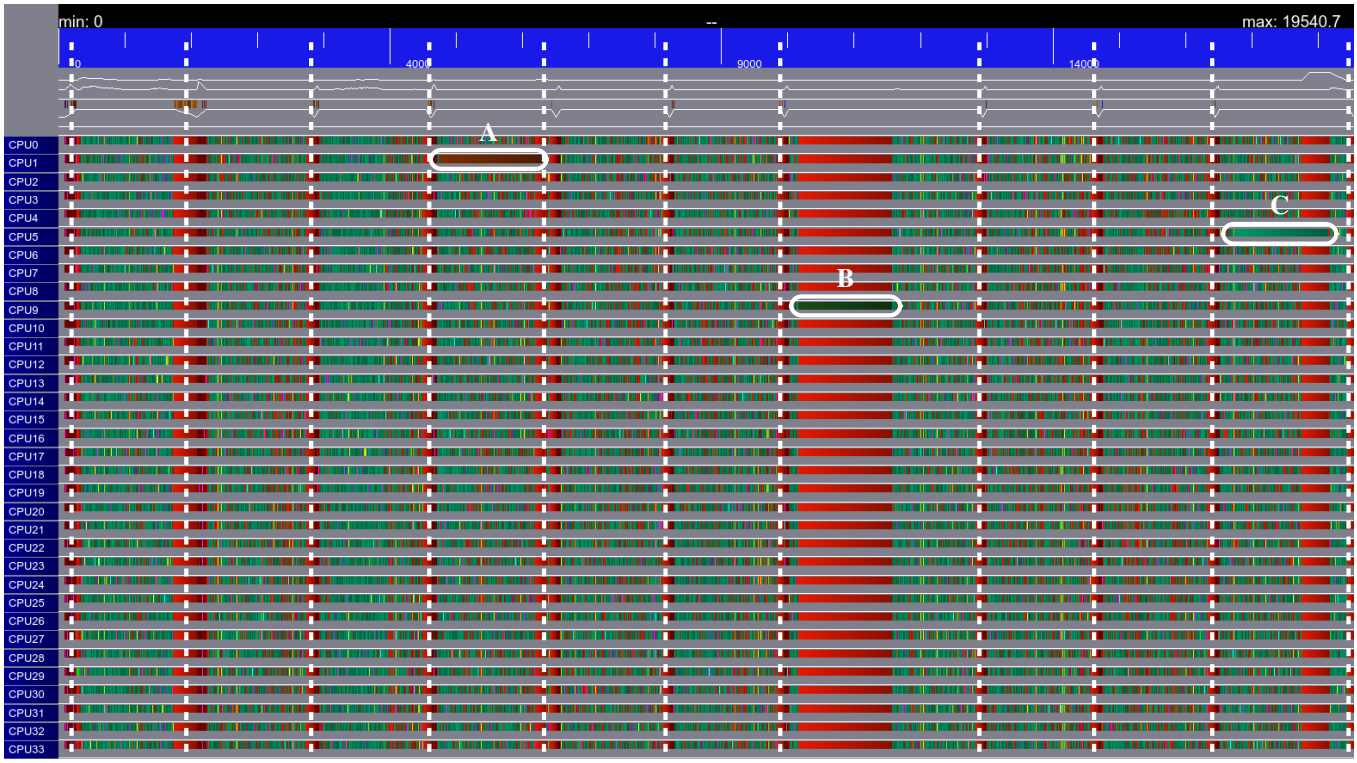

FIGURE 3 Trace of several CHOLESKY factorizations, different runs being separated with vertical white dashed lines. Time progresses from left to right and the different cores are stacked from top to bottom. In the activity of cores, green areas correspond to task executions, red color means an idle core and other colors are for STARPU's internal activities. Highlighted tasks **A**, **B** and **C** are those impacted by event buffer flushes.

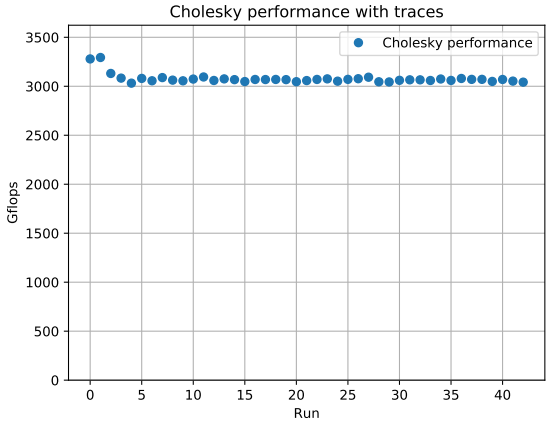

FIGURE 4 Performance of several runs without interrupting buffer flushes, on a `bora` node.

Another (not implemented) idea to avoid disturbing buffer flushes is to dedicate a non-bound thread to flush the buffer. Since FXT has a double-buffering system, enabling to record events in a second buffer while the first one is being flushed, the thread could write the buffer on the disk without disturbing other important threads. Moreover, this thread would be performing only I/O activities, requiring few CPU resources. This could avoid having to manually specify a buffer size to avoid the problem.

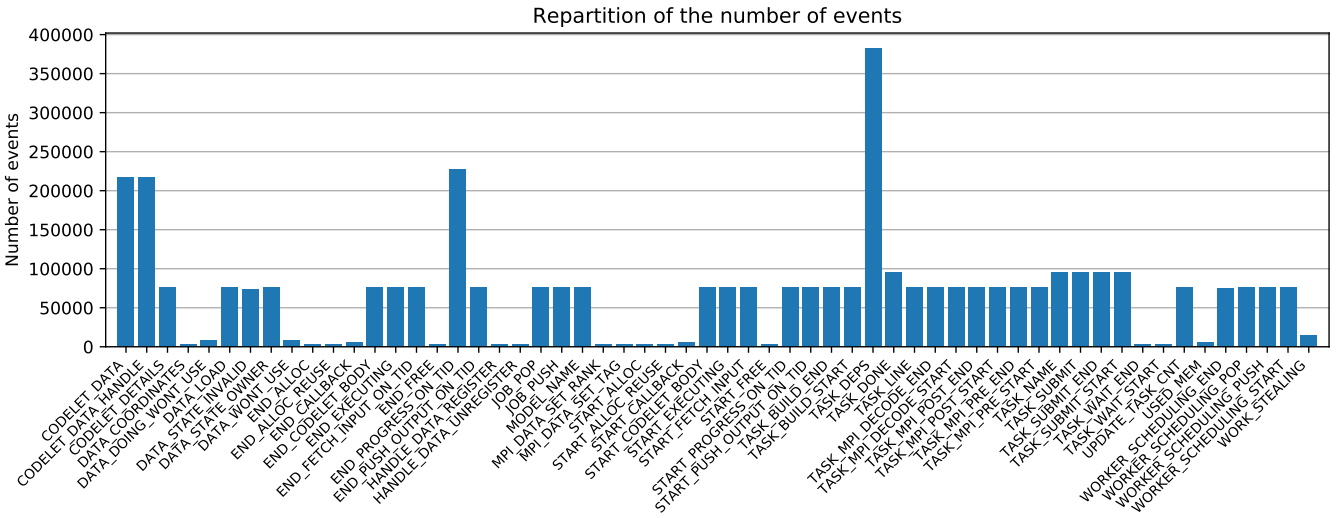

FIGURE 5 Number of events according to their type.

4.2 | Number of recorded events

The more events are recorded, the more time is spent in the tracing library and we can presume the overhead will be more important. By default, all available event types in StarPU are recorded. The resulting number of recorded events in the trace file can be considerable.

Figure 5 depicts the number of events according to their type, for one run of the Cholesky decomposition of a matrix of size of $24\,000 \times 24\,000$. The trace file weights 170 MB and contains 3 887 676 events. On the histogram, only events with more than 2000 occurrences are considered. We can notice some event types are more represented than others: for instance, `TASK_DEPS` (records dependencies between tasks), `END_PROGRESS_ON_TID` (records end of memory transfers), `CODELET_DATA` and `CODELET_DATA_HANDLE` (both record information about the data buffers used by tasks) make the majority.

Recorded events also depict the potential different phases of the analyzed application. Thus, the number and type of recorded events can change during the application execution. Figure 6 represents the number of events generated during the application execution. Even without knowing in details which events are recorded, we can notice four phases: (A) data and problem initialization, (B) task executions, (C) task graph submission, and (D) data release. There are more events during the phase C, because the task graph submission is overlapped by the task executions, which are two different StarPU's activities, each generating their own events. If we look at the same plot, but with details about which types of events are recorded (Figure 7), our hypothesis is confirmed: events corresponding to task graph submission occur only in phase C, while events about task execution occur during the whole phase B.

From figures 6 and 7, we can determine which event types are dominant in the trace file and thus which ones have to be filtered out in priority, if we want to lighten the tracing activity. The difficulty in selecting which event types to drop during trace recording is finding the good trade-off between acceptable trace overhead (caused by an important number of events to collect) and enough events in the trace file to be able to do insightful *post-mortem* analysis.

If the user knows on which events to focus to analyze the trace of an execution, the set of recorded events can be reduced to keep only the interesting events, and thus reduce the tracing overhead. There are several possible approaches:

- Using the environment variable `STARPU_FXT_EVENTS` to specify which event categories have to be recorded:

  ```
  export STARPU_FXT_EVENTS="TASK|DATA|WORKER"
  ```

- Manually changing in the source code of StarPU which events will be recorded (by removing some tracing probes, for instance). This can be much more complicated than the previous solution, but it allows a more fine-grain selection of events than just filtering out whole categories.

Figure 8 depicts the tracing overhead according to which events are recorded, which changes the number of recorded events (as reported by the red dots to be read on the right Y-axis). One can notice that building StarPU with the trace support, without

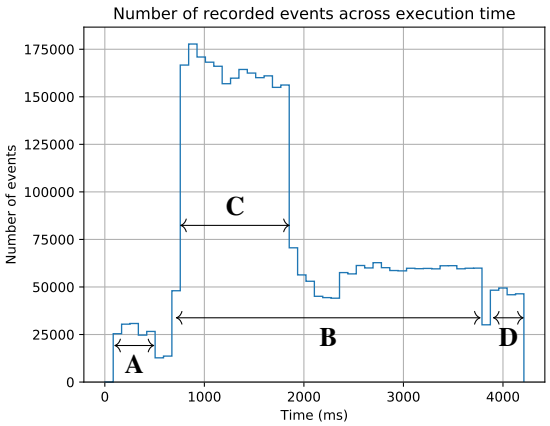

FIGURE 6 Number of events across time (see Figure 7 for details about the number of events).

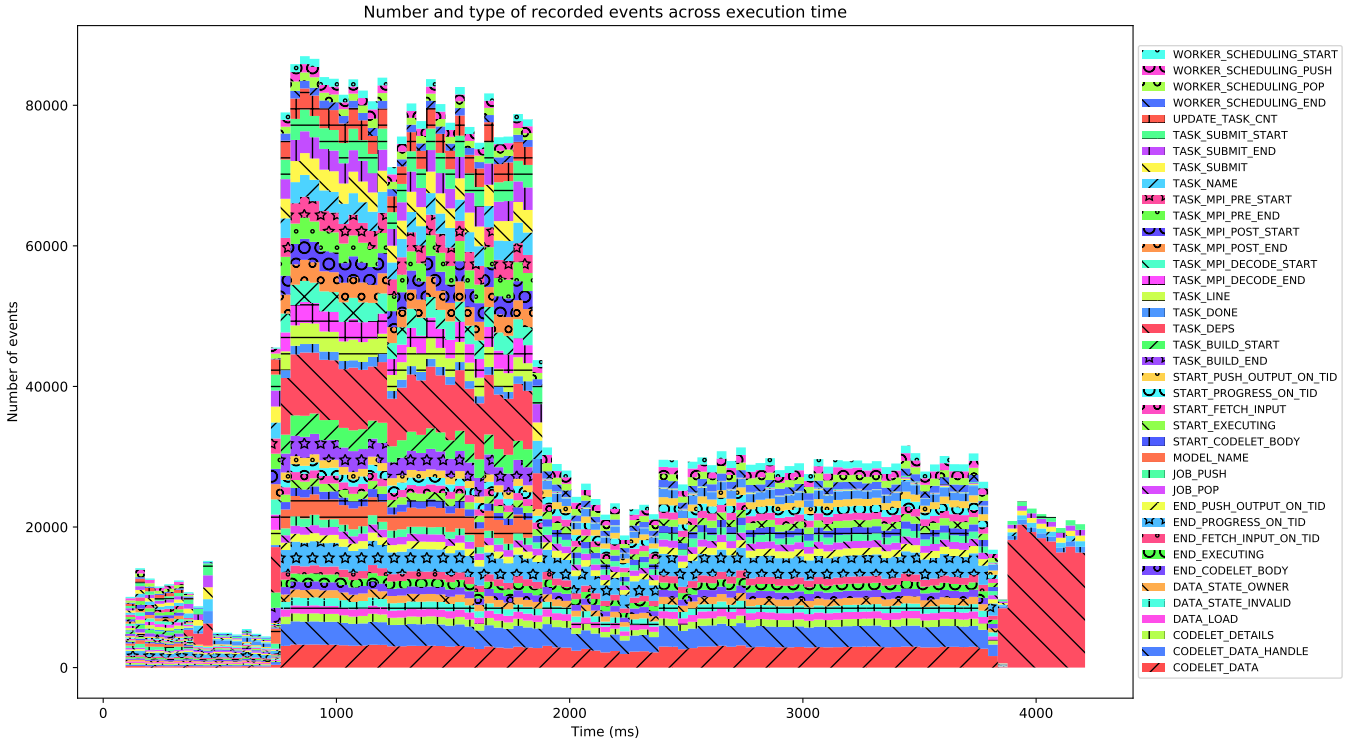

FIGURE 7 Number of events across time, for each event type (detailed version of Figure 6).

enabling trace recording at the runtime, does not add an overhead. Then, as expected, the more there are recorded events, the more the impact on application performance is important. It should be noted that this seems to be relative to the runtime system behaviour: in this case, the task graph submission is longer than task execution, thus workers were actively waiting for new tasks to execute. In an execution with another configuration, where the task graph submission is shorter than task execution (because of bigger tasks lasting longer), the overhead of traces is almost negligible (see Figure 9). We can conclude that the trace overhead is mainly caused by events to record on the runtime system critical path, especially when this critical path is under pressure.

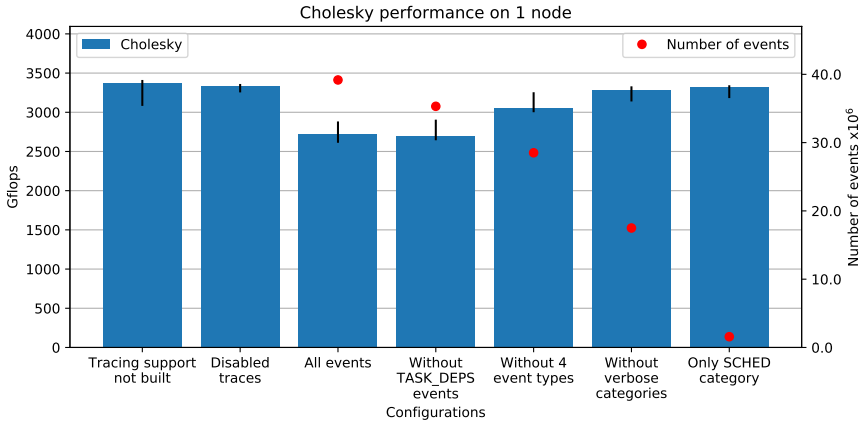

FIGURE 8 Impact of the number of recorded events on the trace overhead. The *4 event types* are the ones previously mentioned being the most numerous in the trace: `TASK_DEPS`, `CODELET_DATA`, `CODELET_DATA_HANDLE` and `END_PROGRESS_ON_TID`.

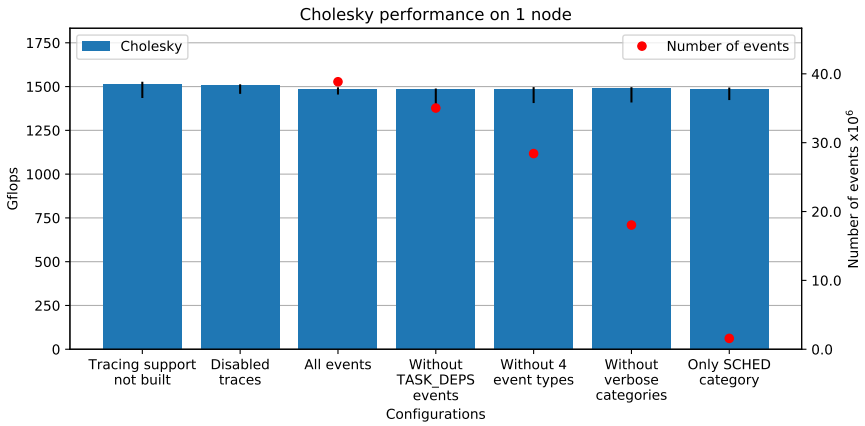

FIGURE 9 Similar to Figure 8, but in this case the tasks took more time to complete, reducing the pressure on the runtime system, thus tracing had less impact on performance.

This observation also shows the impact of task granularity mentionned earlier: bigger tasks reduce the performance overhead of the tracing process.

4.3 | Scalability of the number of recording cores

The number of workers (threads bound on CPU cores, for instance) used by StarPU to execute tasks can be set by the user at execution time. The default configuration is to put one thread per processor core. All these threads produce events to be recorded.

By observing the performance of the strong scaling of the Cholesky decomposition, we can notice that the more there are threads recording events, the higher the impact seems to be on performance: Figure 10 shows the results on `peabody`, an Intel machine.

On AMD `zonda` nodes, when the MKL library (the library providing routines called by tasks to actually make the linear algebra computations) is used with its default settings, the maximal reached performance is 1 Tflops, and there is no impact on performance when traces are enabled, regardless of the number of computing cores (see Figure 11). However, when the MKL is correctly set up to use all features of the AMD processor (Figure 12), the maximal reached performance is 3 Tflops, and there is an impact on performance with enabled traces, starting from 53 computing cores (out of 64). This makes sense since faster

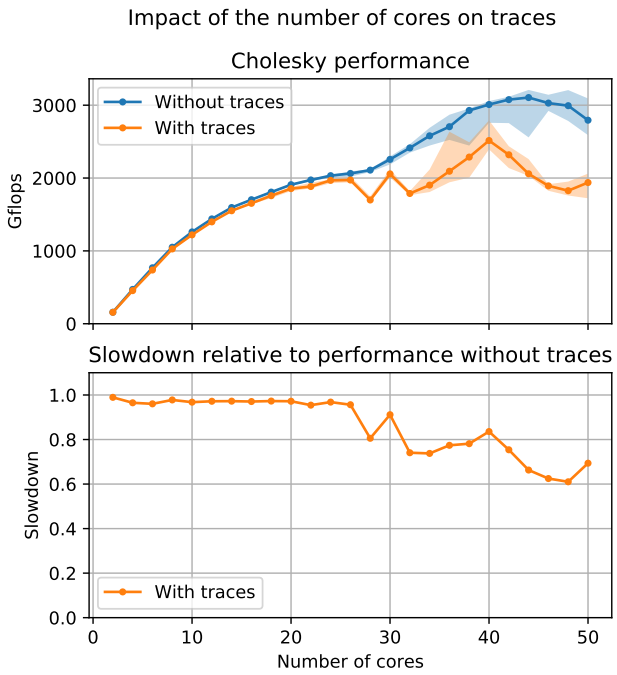

FIGURE 10 Impact of the number of cores on performance of CHOLESKY factorization with traces on `peabody`, with INTEL processor.

execution of tasks means a higher throughput of events to record. We can notice here the tracing overhead also depends on performance of the analyzed application, and not only from the implementation of the tracing library or the runtime system.

The observed phenomenon comes from a lock that protects the single list of recorded events in the FXT library. This single list allows to easily keep the temporal order of recorded events in the trace file. With a list of events per core, there would be no lock (and thus no contention on waiting for this lock), but the events would then need to be correctly reordered before any possible exploitation: during the writing of the trace file on the disk or during the conversion of the trace file with `starpu_fxt_tool`. The reordering could be based on the timestamps of the events, which need proper synchronization even when running on a single node, as detailed in Section 5.3.2. Despite the event reordering complexity, using an event buffer per core is a good solution: for instance, LITL has an event buffer per core and scales better regarding the number of cores recording events[6].

The CHOLESKY factorization is rather a compute-bound algorithm. To measure the impact of tracing when computations are more memory-bound, we consider the case of a matrix copy. Figure 13 presents the results of the same experiments presented previously in this section, but a matrix copy is executed instead of a CHOLESKY decomposition. The maximum performance of 46.3 GB/s is reached with 8 computing cores and with more cores, performance decreases until 16.4 GB/s with 50 cores, due to contention in the memory system. Since trace recording systems also require memory for their internal mechanisms, enabling them can also have an important impact on performance of memory-bound applications, as observed in the figure: even before memory contention impacts the application without traces, recording traces reduces the application performance up to 36 %. Again, this type of performance overhead can be mitigated by tuning the task granularity.

4.4 | Summary about the tracing impact on performance

The various experiments and results presented in this section explored three sources of disturbances caused by the tracing system, impacting application performance and behaviour. Writing on the disk the buffer containing all recorded events can have a severe impact on the application performance, depending on where the flush occurs. Recording many events means an important intrusion in the runtime system behaviour, and an increased activity of the tracing library, which can increase the performance overhead caused by the tracing system. Similarly, many cores recording events or memory-bound applications can generate contention on getting access to the tracing library.

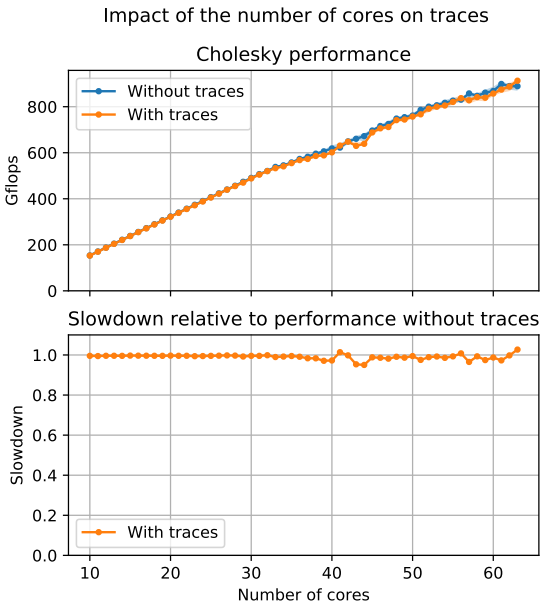

FIGURE 11 Impact of the number of cores on performance of CHOLESKY factorization with traces on `zonda`, with AMD processor and badly configured MKL library.

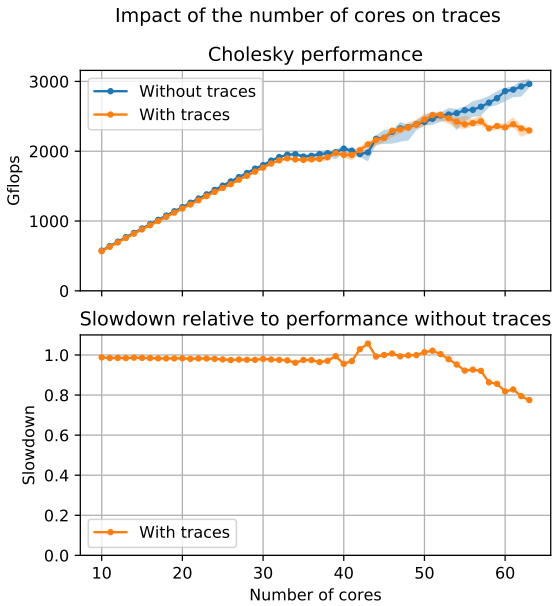

FIGURE 12 Impact of the number of cores on performance of CHOLESKY factorization with traces on `zonda`, with AMD processor and correctly configured MKL library.

Along each evaluated source of overhead, we proposed solutions to reduce the impact on performance: the generic advice is to limit the number of recorded events, and especially their throughput. The number of events can be reduced by changing the task granularity, although it can have other effects (the possible parallelism or the overhead of the runtime system itself). Another solution to reduce the tracing impact could be to compress on-the-fly[9,25] recorded information to minimize the required memory dedicated to traces. Since FXT does not provide such feature, we consider this idea as beyond the scope of this study and left it for future work.

All these experiments were made on a single node. In distributed executions, the distributed aspect does not bring additional specific overhead coming from the tracing system. However, it has also its set of difficulties, as explained in the next section.

5 | PRECISE DISTRIBUTED TRACES

When distributed applications are traced, there is a need for precise synchronized clocks between nodes to keep a temporal coherency between events. If clocks are not synchronized, the event order can be wrong and, for instance, network data transfers can appear as being received before they are sent! A software adjustment is necessary to avoid these artifacts.

This section presents the problems requiring synchronized clocks, our implementation of precise distributed clocks to trace STARPU applications and its empirical evaluation.

5.1 | Motivation for synchronized clocks

Usually, when tracing distributed applications, each process uses the local clock to timestamp events recorded in the trace file. To keep a correct temporal coherency between events (an event on a node occurring after another event on another node is presented as such in the post-processed trace files), clocks of all nodes have to be synchronized: all clocks need to have the same origin and the same speed.

Unfortunately, such perfectly synchronized clocks are usually not present on computing clusters. The local clocks have as origin the start of the node, which is hardly the same on all nodes (nodes are sometimes rebooted independently from each other,

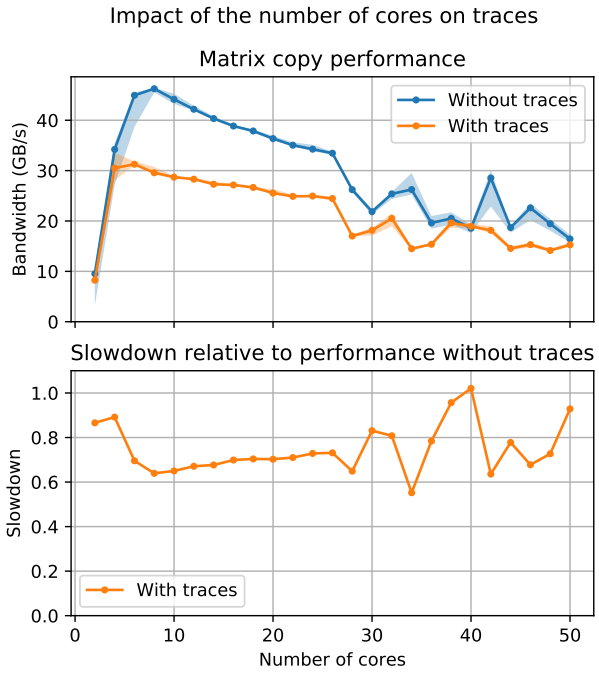

FIGURE 13 Impact of the number of cores on performance of matrix copies with traces on `peabody`, with INTEL processor.

for maintenance tasks, for instance). They can also have different speeds, depending on differences in crystal manufacturing, temperatures, and voltage variations.

Synchronization methods exist (such as NTP[26]) to have the current time available on all nodes, but they are too much coarse-grain for the tracing requirements. If we need duration of communications reported in traces, the allowed error on the clock synchronization has to be lower than the minimum network latency (approximately 1 μs on INFINIBAND networks).

All in all, if we want precise and coherent distributed traces, the problem of distributed clocks to be precisely synchronized has to be considered in tracing systems. The rest of this section presents the related work about this topic and how we addressed this issue in STARPU.

5.2 | Related works

The problem of distributed synchronized clocks, applied to tracing systems or other, has already been covered in the literature, to explain the origin of clock differences in distributed systems, to propose algorithmic solutions, and to present solutions used by applications.

The problem of synchronizing distributed clocks is not only present in the HPC area. For commodity computers, the NTP[26] (*Network Time Protocol*) is used by operating systems to have a correct clock, but with a coarse-grain precision: only around 200 μs on local networks. In research about distributed systems, for instance, CLÉMENT and DAGENAIS synchronize[27] event timestamps to trace events in OS kernel during distributed executions.

In the HPC area, there are two main features requiring accurate synchronized distributed clocks: correctly timestamping events to trace distributed executions and benchmarking inter-node communications. Both suffer from the same problems and can usually be solved by similar solutions, but there are still some differences.

In tracing systems, one of most important requirements is to keep sequential consistency in the traces, for instance to avoid communications appearing as received before they are sent (this kind of artifacts are sometimes called *tachyons*). In addition to correctly synchronizing clocks in order to accurately compute the clock offset during post-processing, some tools also rely on logical clocks, following the ideas introduced by LAMPORT[28]: they look for timing inconsistencies and try to correct them by changing their timestamp to preserve the correct chronological event order[29]. With VAMPIR, two barriers are used before and

after the application execution, and then the event timestamps are corrected by interpolating the clock offset[30]; SCALASCA use additional logical clocks to fix remaining inaccuracies[31,32].

For communication benchmarks, especially *collective* communications (involving several processes), the accuracy of a synchronized clock is of paramount importance to have precise measurements and to be able to correctly analyze the results. The problem lies more in being able to start an MPI operation at the exact same time on all nodes, rather than measuring the duration of an action taking place over several nodes. If all processes are able to start at the exact same time, we can use local clocks to measure the duration of the local action, and then aggregate the duration of all local events to get an overview of the global duration. Many articles[33,34,35,36,37,38] explore what different methods are used in MPI benchmark sets to synchronize clocks. Lots of tools just use MPI barriers in each loop iteration to start the MPI function at the same time on all processes, despite the inaccuracy MPI barriers can suffer from, as pointed out in several papers[37,35]. A common practice is also to use the same process (and thus the same clock, not requiring distributed synchronization) to collect the start and end time of the routine execution to be benchmarked. SKAMPI was the first MPI benchmark[39] to implement the most efficient technique to start a function on several distributed processes at the exact same time, with a so-called *window-based synchronization*.

In both cases, tracing or benchmarking, synchronizing clocks requires efficient algorithms, to compute clock offset as fast as possible. The literature contains some work about the communication patterns to use to synchronize clocks[40,37,41,38,35], implementation techniques[42] or statistical approaches[43,44].

In the MPI standard, the function `MPI_Wtime` returns the time in seconds since an arbitrary time in the past. The origin of the clock used by `MPI_Wtime` is guaranteed not to change during the life of the process. However, the used clock does not have to be necessarily synchronized with other processes in the MPI job. In other words, having a global synchronized clock is left to the appreciation of MPI library developers, which is currently not the case in OPENMPI, MVAPICH or INTEL MPI.

5.3 | Synchronized clocks in STARPU

This section presents in detail how we implemented state-of-the-art techniques to have distributed traces with events precisely timestamped within STARPU. We also report how it improved the accuracy of event timestamps, by comparing the different solutions.

To evaluate the accuracy of several synchronized methods, we will execute several times a ring communication pattern between all nodes, with 4-byte-long messages.

5.3.1 | Implementation overview

The general idea is to record events timestamped with the local clock; record an event at the exact same time on all nodes; and during trace file post-processing, use this event to compute clock offsets and adjust the timestamps of all events.

A naive approach to execute an event at the exact same time on all nodes is to perform an MPI barrier, and record the event just after the exit of the barrier. An MPI barrier is a function, provided by the MPI standard, that blocks as long as not all processes called the function. However, an MPI barrier is not precise enough to synchronize clocks, because all nodes do not leave the barrier at the same time, as illustrated by Figure 14: the event recorded just after processes left the barrier will not happen at the exact same time ($t_0 \neq t_1 \neq t_2 \neq t_3$), while it will be considered as such ($t_0 = t_1 = t_2 = t_3$) to compute clock offsets. In practice, this can actually make communications being received after they are sent, as shown by Figure 15: `starpu_fxt_tool` considered that all MPI processes have left the MPI barrier at the same time (as indicated by the four vertically aligned white circles). However, this was not true: the MPI process 2 left the barrier a moment after the other processes.

Using a precise barrier

To make all nodes leaving the barrier at the same time, we use a *synchronized barrier*, based on a window-based synchronization[39]. During application execution, after computing clock offsets (see below), the node 0 (arbitrary choice) decides at which time all nodes will leave the barrier. This time is broadcasted to all nodes, each node applies the previously computed clock offset to get the decided time in its local clock and then waits in the barrier until the deadline. Thus, all nodes leave the barrier at the exact same time. The next instruction after the barrier is to record the event which will be used to compute clock offsets during the trace post-processing.

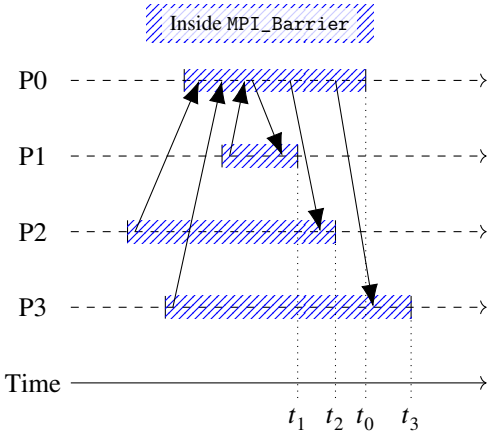

FIGURE 14 `MPI_Barrier`: Not all processes leaves the barrier at the same time.

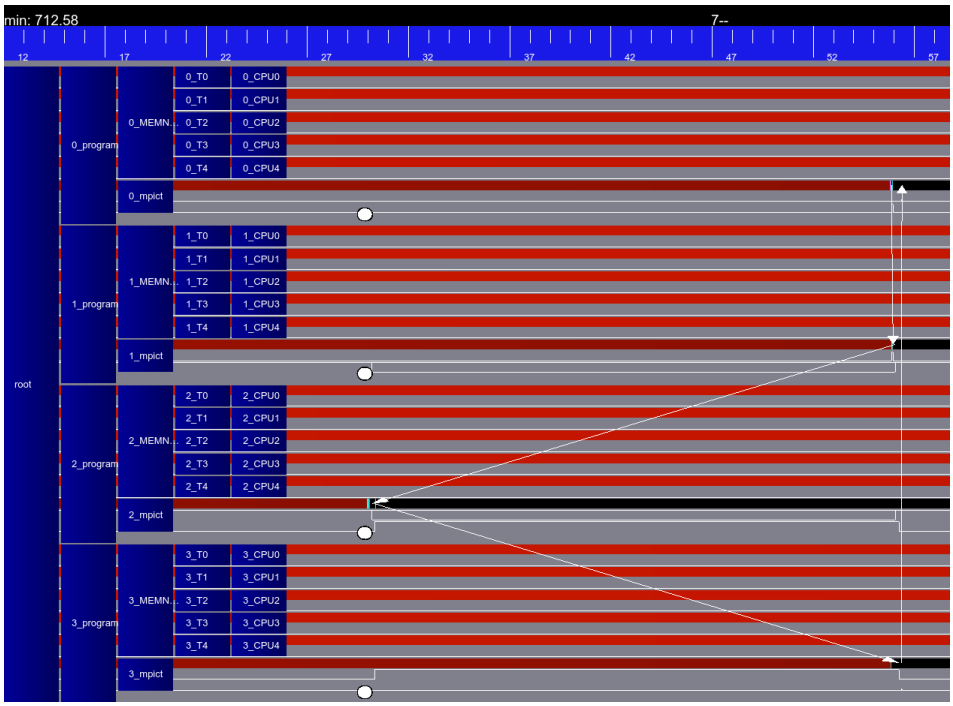

FIGURE 15 The communication from node 1 to node 2 is received before it is sent!

Computing clock offset

To compute the clock offset between nodes, for each node to convert the time received from the node 0 to its local clock, clocks of the two nodes are compared, with the following protocol, illustrated by Figure 16:

1. The node 0 saves the current time t_{start};
2. The node 0 sends a message to the node n;
3. Just after the node n receives the message from the node 0, it saves the current time t_{middle};
4. The node n sends the time t_{middle} to the node 0;
5. Just after the node 0 receives the message from the node n, it saves the current time t_{end};

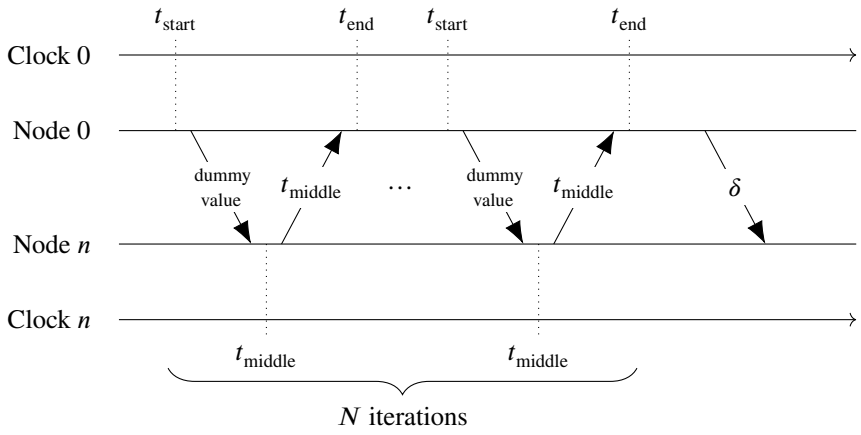

FIGURE 16 How clock offset δ is computed between nodes 0 and n.

6. The node 0 can now compute the clock offset δ between the nodes 0 and n: $\delta = \frac{t_{start}+t_{end}}{2} - t_{middle}$;
7. The previous steps are repeated N times and we select the clock offset obtained with the minimal difference $t_{end} - t_{start}$;
8. The node 0 sends to the node n the selected offset δ.

Let's explain some details of the protocol. To compute the clock offset between the nodes 0 and n, we need to send the time given by the clock n to the node 0, so it can compare with its own local clock (step 4). However, the time required to send t_{middle} to the node 0 has to be taken in account when comparing the two clocks. To do so, we need to know the duration of the communication transferring t_{middle}. Since the only accurate way to achieve this is to use the same clock to read off the times before and after the communication, we measure on node 0 the necessary duration to send a dummy value with the same size of the transferred message from node 0 to n (step 2) and receive back t_{middle}. Then, the average of t_{start} and t_{end} should be aligned with t_{middle}, and the difference between this average and t_{middle} gives the clock offset between the nodes 0 and n. This is based on the assumption that the communications from the node 0 to the node n and inversely are exactly symmetric: this is why we send a dummy message from the node 0 to the node n (instead of an empty message), and we take the clock offset obtained with the minimum duration of the exchanges between the nodes 0 and n (the smallest duration is when the fluctuation of network latency is minimal, thus a better symmetry of the two communications).

Taking into account clock drift

With this synchronized barrier, we can have one reference point in the trace files to compute clock offsets between nodes. However, due to clock drift being different on each node, the computed clock offset is valid only to adjust timestamps of events recorded a short time after the synchronized barrier. Then, clocks follow different drifts and the synchronization is not valid anymore.

A solution to take into account clock drifts is to perform two synchronized barriers, delimiting the period requiring precise well-synchronized timestamps and then, in the trace file conversion step, interpolate the clock offset to apply to each timestamp, with the events recorded after the two synchronized barriers as reference points.

Figure 17 shows the difference between using one (at the beginning of application execution) or two (at the beginning and at the end) synchronized barriers. This figure plots the duration of communications from the rings mentioned earlier: the send and receive times are taken from two different nodes (the sender and the receiver), thus having a precise synchronized clock is crucial here for a good estimation of communications duration. Since each communication has the same message size, communications duration should be constant. With only one synchronized barrier at the beginning of the execution (blue dots), measured durations are constant, but for less than one second. Then, the durations follow lines with non-zero slopes, coming from clock drifts. The different slopes come from the clock drift differences between pairs of nodes (clock drift difference between nodes 0 and 1 is different from the clock drift difference between nodes 1 and 2). With two synchronized barriers (before and after the region of interest, orange dots), measured durations are, as in the reality, constant. Durations measured with a simple MPI barrier are not represented, because they are much higher, and the chart scale prevents from seeing the interesting differences between one or two synchronized barriers.

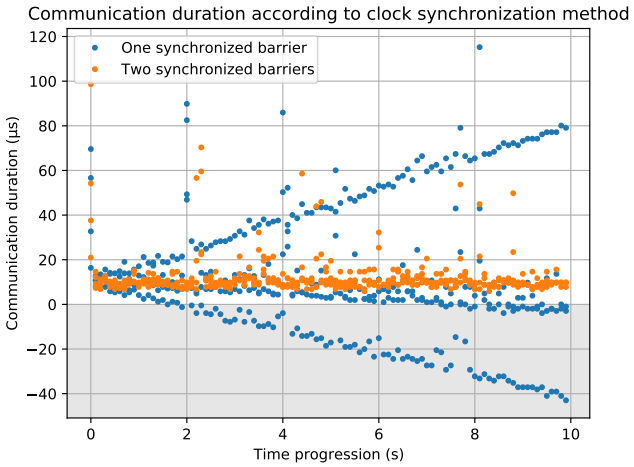

FIGURE 17 Communications duration over time: two synchronized barriers are required to take into account clock drifts.

However, linearly interpolating clock drift assumes the clock drift is linear, which in practice is not the case after several minutes. Thus, this method can be used to trace only short executions.

5.3.2 | Using the right clock

To get a sufficient resolution for the clock and reduce drift, we must use the right clock source. There are multiple sources for clock in a regular computer:

- RTC: this is the basic clock available in every PC/AT-compatible machine since 1984. It has been considered legacy for a long time. Its resolution is very poor.
- ACPI_PM: this is the clock device from the ACPI Power Management specification[45]. The frequency is hardwired to 3.579545 MHz which makes it a better resolution than RTC but still poor to timestamp events in the range of the gigahertz. It is considered legacy as a clock source.
- HPET (*High Precision Event Timer*)[46]: this is a clock that was introduced especially to get a precise and steady clock source. It guarantees a resolution higher than 10 MHz. Its resolution is usually average and it costs a system call to be read.
- TSC (*Time Stamp Counter*): this high resolution timer was introduced[47] in the INTEL PENTIUM PRO family. It is synchronized with the instruction counter and thus has a resolution sufficient for tracing. Moreover, it is very cheap to consult, using a single unprivileged instruction. However, it is not guaranteed to be synchronized between cores, and its frequency varies with the processor frequency (which can affected by Turbo Boost or energy saving strategies), which makes it an unreliable source for timing.
- Invariant TSC: more recent CPUs feature an *invariant* TSC, which is based on the ART (*Always Running Timer*), running at the crystal clock frequency. This flavor of TSC is synchronized between cores and uses a constant frequency, even when the CPU changes its frequency for energy saving.

Hence, the most relevant clock to use for traces is invariant TSC, when available. It has a good resolution, is steady, and cheap to consult. When it is not available, the second-best choice is HPET, though the resolution is usually much lower.

To get access to the clocks, multiple interfaces are available. The good old `gettimeofday` interface cannot express high resolution, and the clock it provides access to is affected by NTP adjustments, which makes it not steady. It is not suitable for our use case. Then there is the direct access to a hardware clock. However, direct reading from `/dev/hpet` is slow (involves a system call) and usually reserved to root. The direct read of TSC (with the `rdtsc` instruction) is fast but is non-portable and requires non-trivial code to check its properties (mostly whether it is invariant TSC or not).

The best solution is to use the POSIX.1-2001 function `clock_gettime` that gives access to various clocks of the system: `CLOCK_REALTIME` is actually the same clock as `gettimeofday` and should not be used for tracing; `CLOCK_MONOTONIC` is fast

and monotonic, derived from the default system clock (usually TSC, sometimes HPET if TSC is not invariant) but affected by NTP (no jumps, but not steady); `CLOCK_MONOTONIC_RAW` is a direct portable way (although LINUX-only) to get direct access to the default clock.

The main goal of NTP is to compensate for the drift between the local clock and the real time. It adjusts the rate of local clock to reach sync with the root clock, thus locally it causes small fluctuations in the speed of the local clock. Therefore, to avoid those fluctuations, it is relevant to use a raw clock when we compensate for the drift ourselves, since we do not need features from NTP and it would only add noise in the clock. However, in cases where we use a single barrier with offset, we need to use a regular clock (with NTP roughly compensating for the drift) since we do not compensate for the drift ourselves.

As a consequence, we use `clock_gettime(CLOCK_MONOTONIC)` when using a single offset, and `clock_gettime(CLOCK_MONOTONIC_RAW)` when interpolating between two synchronized barriers.

5.3.3 | Protection from preemption between synchronized barrier and event probe

As mentioned earlier, one of the most efficient ways to have an event recorded at the exact same time on all nodes is to perform a synchronized barrier and then execute the probe corresponding to the synchronizing event. In the code, it can look like this:

```
mpi_sync_barrier();
_STARPU_MPI_TRACE_BARRIER(rank);
```

Since the barrier is synchronized, we are sure each process will leave the barrier function call at the same time and should execute the next instruction (here: the trace probe to later compute clock offsets) also at the same time. However, a thread preemption can happen between the end of the barrier call and the trace probe: this could prevent the event from being recorded at the exact same time on each distributed process. To avoid such problem, the solution is to get the local time the barrier is waiting to unblock, use this time as an additional data for the trace probe, and then use this time as the event timestamp, instead of the timestamp set by the tracing library to compute the clock offset:

```
mpi_sync_barrier(&local_sync_time);
_STARPU_MPI_TRACE_BARRIER(rank, local_sync_time);
```

In fact, the interesting value to correctly synchronize distributed clocks is not the timestamp of the event indicating this synchronization, but the local time the barrier was waiting for.

Even if it can seem obvious, it is important to mention that the clock used for timestamping events and the one used for the time as additional data of the synchronized event have to come from the same clock and have the same origin. Since timestamping events is done by the tracing library and the synchronized barrier is done by another library, this might not be trivial, and requires proper support: either being able to specify the same clock to be used by the both libraries or to convert the time given by the synchronized barrier to the one used by the tracing library.

5.3.4 | Compute clock offsets to adjust event timestamps

A synchronized barrier followed by a recorded event is done in the STARPU initialization and release phases, thus delimiting the application execution requiring events with precise timestamps. In order to merge the distributed traces, `starpu_fxt_tool`, the tool in charge of converting raw trace files to exploitable ones, now has two tasks to achieve: computing the clock offsets between the nodes from the events of the synchronized barriers, and then interpolating the clock offsets to apply the timestamp adjustment on each event. Figure 18 describes how the clock offsets corresponding to one synchronized barrier are computed:

- Clock origins can be different on each node, because the clocks we are using have as origin the start of the node. Then, the application does not start at the exact same time on all nodes (the instruction to start the application is not synchronized to be executed at the exact same time on all nodes). In the runtime system initialization, once all distributed processes are launched, we execute a first synchronized barrier to record the event used as reference point by all processes.

- **First step**: we consider the event following the synchronized barrier as the local time origin. Thus, the synchronized time $\widetilde{t_i}$ of an event on process i with a timestamp t_i is: $\widetilde{t_i} = t_i - t_i^{\text{Barrier}}$. This also allows having small timestamp values, easier to analyze, since the clock origin will now be the beginning of the application execution.

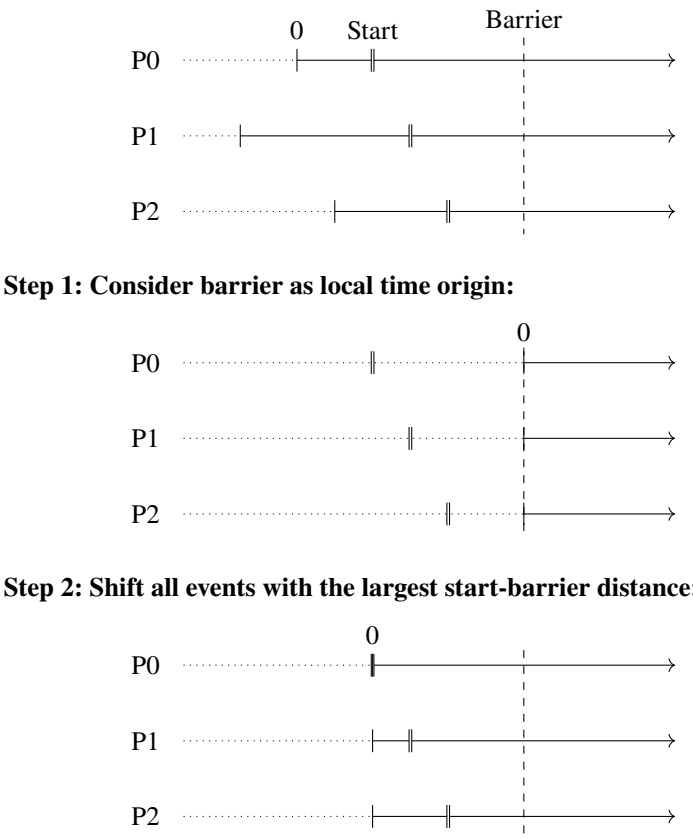

FIGURE 18 Clock offset computation.

- **Second step**: the previous step makes the events occurring before the barrier having a negative adjusted timestamp. To avoid this, we shift the time origin, to the first application start, which is the maximum of the distances between application starts and synchronized barriers. Now the transformation to apply is: $\widetilde{t_i} = t_i - t_i^{\text{Barrier}} + \max_j \{t_j^{\text{Barrier}} - t_j^{\text{Start}}\}$.

As one may notice, there is no one node, designated as having the reference clock, before beginning computing clock offsets.

When two synchronized barriers are used, we follow the same steps: the two barriers give two sets of clock offsets between nodes, and then the offset to apply to an event timestamp is linearly interpolated. Outside of the interval delimited by the two barriers, we cannot extrapolate the clock offset, because there is no guarantee a linear extrapolation will fit the real clock drift. Thus, in some cases, applying the extrapolated offset can lead to negative timestamps. Figure 19 illustrates this phenomenon. The black vertical lines represent the first and last events recorded in the trace. The synchronized barriers and the offsets computed for each of them are represented by the blue dots on the upper plot. The time range between these two dots is the range where interpolating clock offsets gives valid results, while extrapolating clock offsets outside of this window can lead to negative timestamps, as illustrated by the orange curve on the lower plot, before local time of 0.45. If we only interpolate offsets in the synchronized window and use the clock offset of the first synchronized barrier for events before it and the clock offset of the second synchronized barrier for events after it, we get acceptable timestamps even outside the synchronized window, as illustrated by the green curve. The resulting inaccuracy is acceptable outside the synchronized window, because timestamp of events outside of the window do not have to be very precise, they are usually more descriptive than indicative of a state change.

5.4 | Conclusion on synchronizing distributed traces

In this section, we presented why we need clock synchronization to trace distributed executions, the related work and how we ended up with a working solution within StarPU.

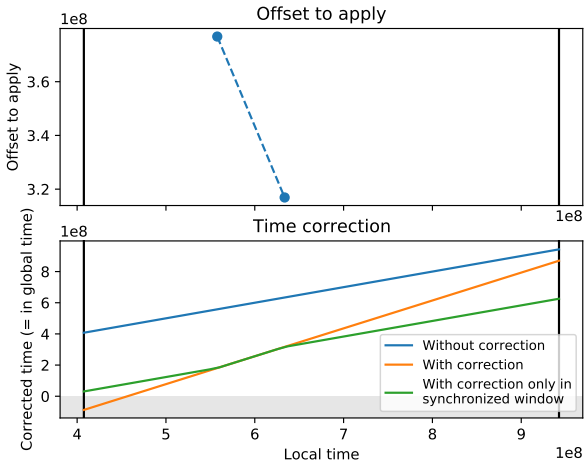

FIGURE 19 Interpolating offsets outside of the synchronized window can lead to negative timestamps.

The first motivation for clock synchronization is the use of clocks having as origin the start of the node to timestamp events. A coarse-grain synchronization can be done with a classic MPI barrier. With a real use-case, the presented experiences show, and confirm what is reported in the literature, that a simple MPI barrier is not precise enough to synchronize clocks, and the clock drift has to be taken into account by interpolating the clock offsets between two reference points. The given implementation details specify which clock we use, how we avoid thread preemption between synchronized barrier and event probe, and how we compute clock offsets during post-processing to merge distributed traces.

The solution we implemented in StarPU is valid while the clock drift is linear, *i.e.* only for short executions. This is not limiting, since we usually only trace short executions, to ease their analysis. We do not use logical clocks, because it would require lots of development, and the correction of chronological order of events would not reach the precision we are looking for in distributed executions.

6 | LESSONS LEARNED

This section summarizes the insights learned by exploring the issues previously described to deal with when tracing applications: it proposes a methodology to apply to efficiently trace applications and a list of requirements for a satisfying tracing system.

6.1 | Methodology to apply when tracing applications

As observed previously, the overhead caused by tracing systems can impact application performance, and even modify the behaviour we want to observe in the *post-mortem* analysis. Thus, the user has to be sure that the potential tracing overhead is acceptable regarding the purpose of the traces. To do so, the best way is to compare the application performance with and without tracing it. If the impact on performance is too important, the user can try to reduce it with several methods:

- Set a correct trace buffer size to avoid applications being disturbed by the flushes on the disk, such as explained in Section 4.1;
- Reduce the number of recorded events by focusing only on important ones, such as explained in Section 4.2;
- If it appears to be a source of overhead, reduce the number of workers (see experiments in Section 4.3). However, this can also change the application behaviour.

Reducing the number of events to record is also useful to reduce the size of trace files, making them much more convenient to manipulate (to transfer between clusters, to convert and exploit, for instance).

If the user is interested in precise timestamps of events occurring on several nodes, using a synchronized barrier to synchronize trace files is a good solution to trace fast executions ($\mathcal{O}$(several seconds)). An easy way to verify the clock synchronization is correct is to compare the observed communication duration to the theoretical duration (for instance, on INFINIBAND networks, a transfer of small data lasting less than 1 μs might be suspect).

6.2 | Requirements for an efficient tracing system

All these highlighted possible problems appearing when tracing applications allow us to suggest a list of requirements for an efficient tracing system. Of course, such systems have to feature the lowest performance overhead and be precise enough, which can be possible with:

- A good scalability of the number of cores recording events: this excludes all general locks protecting a resource accessed by all cores at the same time;
- Accurate-enough timestamps of events, especially on distributed executions, but this is also true if each core records events in its own buffer: at the end, events have to be presented ordered to exploit the trace;
- A good control of trace buffer flushes on the disk, to avoid them occurring in the traced region of interest;
- A user-friendly system to easily filter which events have to be recorded, to reduce the tracing overhead and the trace file size;
- A completely transparent tracing mechanism from the point of view of the user application: only the runtime system has to be aware whether tracing is enabled or not;
- Regarding the integration of a tracing system in the runtime system, the runtime developer has to be careful about where to put the probes in the code, to avoid overloading critical paths. Also, if the tracing system allows it, it might be interesting to distinguish events requiring a timestamp (state changes or actions which duration is an important information) from others, usually more descriptive (for instance the dependencies of a task, which scheduling policy is selected, or the machine's characteristics), since the latter do not require to be correctly ordered to keep a temporal coherency. Thus, it can reduce the contention of resources in charge of respecting the temporal order of events.

Correctly and efficiently implementing all these features in a tracing system is not straightforward, this is why developers wanting to add a tracing system to a runtime system should look for using already existing tracing projects instead of implementing their own from scratch!

6.3 | Extension to other runtime systems

We discussed previously the specificities of task-based runtime systems when it comes to tracing their executions: especially information about the application DAG and its scheduling has to be recorded in the trace. However, the majority of phenomenons covered in this paper is also valid in the context of other runtime systems than task-based ones. We took the occasion of improving the tracing system used by STARPU to evaluate its capabilities, but in the end, the reported issues are not specific to task-based runtime systems: performance overhead caused by buffer flushes or the number of events to record has to be considered with every tracing system, the number of cores recording events has to be considered for parallel applications, while distributed synchronized clocks is a general problem when tracing distributed applications.

7 | CONCLUSION

Tracing application executions helps to understand their behaviour and improve them, it is a valuable technique given the current complexity of supercomputers. We presented how we integrated a tracing layer in the STARPU task-based runtime system.

We evaluated the different sources of performance overhead coming from the tracing system: writing the event buffer on the disk during the execution, the number of recorded events, and of recording cores, are all responsible for penalizing the application performance. The tracing overhead can depend of the application behaviour, but can also change it; thus the user should always

try to minimize the overhead caused by the tracing system, or at least be aware of the performance difference when traces are recorded.

We improved the clock synchronization mechanism in StarPU by implementing state-of-the-art techniques, and observed the different timestamp accuracy when different techniques are used to synchronize clocks. This work also confirms that using accurate synchronized clocks is necessary when dealing with distributed applications.

From the observations and conclusion we made, we proposed a methodology to follow when an application is traced, to minimize the phenomena we described; and we suggested a list of features a tracing system should implement to ease the application integration and user manipulation.

Finally, even if our feedback comes from experiments with StarPU and FxT, most of our observations and conclusions are valid also to trace applications that do not rely on a task-based runtime system.

As future work, there is room for improvements of the current implementation. As already mentioned, FxT could dedicate a thread to flush its buffer, to avoid blocking the thread calling the probe. Moreover, the proposed method to dimension the trace buffer is not perfect, especially if the required buffer size is bigger than the available RAM memory. The scalability of FxT regarding the number of cores recording events could be improved, probably by using a buffer of events per core. For distributed executions, the adjustment of event timestamps could be improved to support longer executions. A convenient feature would also be being able to adjust the timestamp *online*, directly during the event recording.

Software Availability

We endeavor to make our experiments reproducible. A public companion[†] contains the instructions to reproduce our study.

AUTHOR CONTRIBUTIONS

Alexandre Denis: Software, Conceptualization, Writing - Review & Editing.
Emmanuel Jeannot: Validation, Writing - Review & Editing.
Philippe Swartvagher: Software, Data Curation, Investigation, Methodology, Visualization, Writing - Original Draft.
Samuel Thibault: Software, Writing - Review & Editing.

ACKNOWLEDGEMENTS

This work is supported by the Agence Nationale de la Recherche, under grant ANR-19-CE46-0009.

This work is supported by the Région Nouvelle-Aquitaine, under grant 2018-1R50119 *HPC scalable ecosystem*.

Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr/).

References

1. Augonnet C, Thibault S, Namyst R, Wacrenier PA. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 2011; 23(2): 187-198. doi: https://doi.org/10.1002/cpe.1631

2. Agullo E, Aumage O, Faverge M, et al. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems* 2017. doi: 10.1109/TPDS.2017.2766064

3. Agullo E, Augonnet C, Dongarra J, et al. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In: Wen-mei W. Hwu ., ed. *GPU Computing Gems*. 2. Morgan Kaufmann. 2010.

4. Nataraj A, Malony AD, Morris A, Arnold DC, Miller BP. A framework for scalable, parallel performance monitoring. *Concurrency and Computation: Practice and Experience* 2010; 22(6): 720-735. doi: https://doi.org/10.1002/cpe.1544

[†] https://gitlab.inria.fr/pswartva/paper-starpu-traces-r13y, archived on https://www.softwareheritage.org/ with the ID `swh:1:snp:a49125c783b3807ba645337c95b1d1a5e5504977`

5. Danjean V, Namyst R, Wacrenier PA. An Efficient Multi-level Trace Toolkit for Multi-threaded Applications. In: Cunha JC, Medeiros PD., eds. *Euro-Par 2005 Parallel Processing*Springer Berlin Heidelberg; 2005; Berlin, Heidelberg: 166–175

6. Iakymchuk R, Trahay F. LiTL: Lightweight Trace Library. technical report, INF - Département Informatique; Telecom SudParis: 2013.

7. Trahay F, Rue F, Faverge M, Ishikawa Y, Namyst R, Dongarra J. EZTrace: a generic framework for performance analysis. In: Venkatasubramanian N, Lee C., eds. *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*; 2011; Newport Beach, CA, USA: 618-619

8. Taheri S, Devale S, Gopalakrishnan G, Burtscher M. ParLoT: Efficient Whole-Program Call Tracing for HPC Applications. In: Bhatele A, Boehme D, Levine JA, Malony AD, Schulz M., eds. *Programming and Performance Visualization Tools*Springer International Publishing; 2019; Cham: 162–184.

9. Wang C, Balaji P, Snir M. Pilgrim: Scalable and (near) Lossless MPI Tracing. In: SC '21. Association for Computing Machinery; 2021; New York, NY, USA

10. Shende S. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications* 2006; 20: 287-311. doi: 10.1177/1094342006064482

11. Schulz M, Galarowicz J, Maghrak D, Hachfeld W, Montoya D, Cranford S. OpenSpeedShop: An Open Source Infrastructure for Parallel Performance Analysis. *Sci. Program.* 2008; 16(2–3): 105–121. doi: 10.1155/2008/713705

12. de Oliveira Stein B, Chassin de Kergommeaux J. Pajé trace file format. online; 2003.

13. Eschweiler D, Wagner M, Geimer M, Knüpfer A, Nagel W, Wolf F. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In: De Bosschere K, D'Hollander E, Joubert G, Padua D, Peters F, Sawyer M., eds. *Applications, Tools and Techniques on the Road to Exascale Computing, Proceedings of the conference ParCo 2011, 31 August - 3 September 2011, Ghent, Belgium*. 22 of *Advances in Parallel Computing*. IOS Press; 2011: 481–490

14. Nagel W, Arnold A, Weber M, Hoppe HC, Solchenbach K. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer* 1996; 63, Vol. XII(1): 69-80.

15. Pillet V, Labarta J, Cortes T, Girona S. PARAVER: A Tool to Visualize and Analyze Parallel Code. *WoTUG '96: Proceedings of the 19th World Occam and Transputer User Group Technical Meeting on Parallel Processing Developments* 1996.

16. Geimer M, Wolf F, Wylie B, Ábrahám E, Becker D, Mohr B. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience* 2010; 22(6): 702–719. doi: https://doi.org/10.1002/cpe.1556

17. Coulomb K, Degomme A, Faverge M, Trahay F. An open source tool chain for performance analysis. In: Brunst H, Müller M, Nagel W, Resch M., eds. *5th Parallel Tools Workshop*Springer; 2011; Dresden, Germany: 37–48

18. Knüpfer A, Feld C, Mey D, et al. *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir*: 79-91; Berlin, Heidelberg: Springer Berlin Heidelberg . 2012

19. Becker D, Rabenseifner R, Wolf F. Implications of non-constant clock drifts for the timestamps of concurrent events. In: Ishikawa Y., ed. *2008 IEEE International Conference on Cluster Computing*; 2008; Tsukuba, Japan: 59-68

20. Jones T, Ostrouchov G, Koenig G, Mondragon O, Bridges P. An evaluation of the state of time synchronization on leadership class supercomputers. *Concurrency and Computation: Practice and Experience* 2018; 30(4): e4341. doi: https://doi.org/10.1002/cpe.4341

21. Duran A, Ayguadé E, Badia R, et al. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* 2011; 21(02): 173-193. doi: 10.1142/S0129626411000151

22. Extrae. https://tools.bsc.es/extrae; . Accessed: 2021-10-01.

23. Bosilca G, Bouteiller A, Danalis A, Herault T, Luszczek P, Dongarra J. Dense Linear Algebra on Distributed Heterogeneous Hardware with a Symbolic DAG Approach. *Scalable Computing and Communications: Theory and Practice* 2013: 699-735.

24. Garcia Pinto V, Mello Schnorr L, Stanisic L, Legrand A, Thibault S, Danjean V. A visual performance analysis framework for task-based parallel applications running on hybrid clusters. *Concurrency and Computation: Practice and Experience* 2018; 30(18): e4472. doi: https://doi.org/10.1002/cpe.4472

25. Aguilar X, Fürlinger K, Laure E. Online MPI Trace Compression Using Event Flow Graphs and Wavelets. *Procedia Computer Science* 2016; 80: 1497-1506. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USAdoi: https://doi.org/10.1016/j.procs.2016.05.471

26. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305; 1992

27. Clément E, Dagenais M. Traces synchronization in distributed networks. *Journal of Computer Systems, Networks, and Communications* 2009; 2009.

28. Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 1978; 21(7): 558–565. doi: 10.1145/359545.359563

29. Ellwood L, Heath M. A Tracing Environment for MPI. 1995.

30. Knüpfer A, Brunst H, Doleschal J, et al. The Vampir Performance Analysis Tool-Set. In: Resch M, Keller R, Himmler V, Krammer B, Schulz A., eds. *Tools for High Performance Computing*Springer Berlin Heidelberg; 2008: 139–155.

31. Becker D, Rabenseifner R, Wolf F. Timestamp Synchronization for Event Traces of Large-Scale Message-Passing Applications. In: Cappello F, Herault T, Dongarra J., eds. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*Springer Berlin Heidelberg; 2007; Berlin, Heidelberg: 315–325.

32. Becker D, Linford J, Rabenseifner R, Wolf F. Replay-Based Synchronization of Timestamps in Event Traces of Massively Parallel Applications. In: Feng W., ed. *2008 International Conference on Parallel Processing - Workshops*; 2008: 212-219

33. Hamid NAWA, Coddington P. Comparison of MPI Benchmark Programs on Shared Memory and Distributed Memory Machines (Point-to-Point Communication). *The International Journal of High Performance Computing Applications* 2010; 24(4): 469-483. doi: 10.1177/1094342010371106

34. Hamid N, Coddington P, Vaughan F. Comparison of MPI benchmark programs on an SGI Altix ccNUMA shared memory machine. In: Spirakis P, Siegel H., eds. *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*; 2006

35. Hoefler T, Schneider T, Lumsdaine A. Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale. *International Journal of Parallel, Emergent and Distributed Systems* 2010; 25(4): 241-258. doi: 10.1080/17445760902894688

36. Lastovetsky A, Rychkov V, O'Flynn M. MPIBlib: Benchmarking MPI Communications for Parallel Computing on Homogeneous and Heterogeneous Clusters. In: Lastovetsky A, Kechadi T, Dongarra J., eds. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*Springer Berlin Heidelberg; 2008; Berlin, Heidelberg: 227–238.

37. Hunold S, Carpen-Amarie A. On the Impact of Synchronizing Clocks and Processes on Benchmarking MPI Collectives. In: EuroMPI '15. Association for Computing Machinery; 2015; New York, NY, USA

38. Hunold S, Carpen-Amarie A. MPI Benchmarking Revisited: Experimental Design and Reproducibility. *CoRR* 2015; abs/1505.07734.

39. Worsch T, Reussner R, Augustin W. On Benchmarking Collective MPI Operations. In: Kranzlmüller D, Volkert J, Kacsuk P, Dongarra J., eds. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*Springer Berlin Heidelberg; 2002; Berlin, Heidelberg: 271–279.

40. Doleschal J, Knüpfer A, Müller M, Nagel W. Internal Timer Synchronization for Parallel Event Tracing. In: Lastovetsky A, Kechadi T, Dongarra J., eds. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*Springer Berlin Heidelberg; 2008; Berlin, Heidelberg: 202–209.

41. Hunold S, Carpen-Amarie A. Hierarchical Clock Synchronization in MPI. In: Nikolopoulos D, de Supinksi B., eds. *2018 IEEE International Conference on Cluster Computing (CLUSTER)*IEEE Computer Society; 2018; Los Alamitos, CA, USA: 325-336

42. Jones T, Koenig G. Clock synchronization in high-end computing environments: a strategy for minimizing clock variance at runtime. *Concurrency and Computation: Practice and Experience* 2013; 25(6): 881-897. doi: https://doi.org/10.1002/cpe.2868

43. Maillet E, Tron C. On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing* 1995; 28(1): 84-93. doi: https://doi.org/10.1006/jpdc.1995.1090

44. Cristian F. Probabilistic clock synchronization. *Distributed computing* 1989; 3(3): 146–158.

45. UEFI Forum, Inc. . Advanced Configuration and Power Interface Specification, version 6.4.; 2021.

46. Intel Corporation . IA-PC HPET (High Precision Event Timers) Specification, revision 1.0a.; 2004.

47. Intel Corporation . Pentium® Pro Family Developer's Manual.; 1995.