Are Noise-Resilient Logical Timers Useful for Performance Analysis?

Gregor Corbin*, Nour Daoud*, Bernd Mohr*, Gustavo de Morais†, Felix Wolf†
* *Forschungszentrum Jülich GmbH, Germany*
{*g.corbin, n.daoud, b.mohr*}*@fz-juelich.de*
† *Technical University of Darmstadt, Germany*
{*gustavo.morais, felix.wolf*}*@tu-darmstadt.de*

***Abstract*—In modern HPC systems, performance measurements are often disturbed by noise. Because repeating measurements to increase confidence in their results is costly, alternative noise-resilient techniques are desirable. Therefore, we implement a logical clock, which does not rely on real-time measurements, in Score-P. We explore several methods to model computational work with the clock increment, counting OpenMP loop iterations, LLVM basic blocks/statements, or hardware counters. We demonstrate the strengths and weaknesses of using logical time stamps in a trace analysis workflow with Score-P and Scalasca, by evaluating the performance problems we can find in three MPI+OpenMP mini-apps. By design, logical measurements reliably show algorithmic issues, such as load imbalance, but cannot capture external aspects of program execution, for example memory contention. In summary, logical-time based measurements are a specialized but valuable addition to the performance analyst's toolbox.**

***Index Terms*—F.1.2.d Parallelism and concurrency, C.4.c Measurement techniques, C.4.g Measurement, evaluation, modeling, simulation of multiple-processor systems, K.6.2.d Performance and usage measurement**

I. Introduction

With the ever-increasing scale of HPC systems, their hardware and software grow in complexity. To make the most of these resource-intensive machines, it is essential to assess and improve performance on all scales of the architecture. Experts rely on a multitude of specialized measurement and analysis tools for their performance assessments. In this work, we focus on a performance analysis workflow with Score-P [1] and Scalasca [2], [3]. Score-P collects traces and profiles from parallel programs, relying mainly on automatic instrumentation of the target applications during compilation and linking. Scalasca analyses the resulting traces to identify wait states and their origins in inter-process communication. It reports its findings as an application profile enhanced with additional metrics derived from the trace analysis. Both tools are designed to measure applications at production scale.

Even the simplest performance metric, the program wall clock time, can vary substantially between repetitions of the same computation due to effects outside the application's control. For example, Chunduri et al. observed a run-to-run variation of 70% [4] in the MILC application on a Xeon Phi cluster. Beni et al. [5] more recently observed the latency of an `MPI_Bcast` microbenchmark to vary by a factor of 6 on the Marconi100 cluster. This noise is present on all modern computers [6] and can have a multitude of origins. Ates et al. [7] classify noise by the hardware component it originates from as CPU [8], cache, memory, storage [9], or network [10], [11] noise. Noise makes the interpretation of performance data less reliable. For instance, did a change in the code really shorten the program run time, or was the baseline just negatively affected by noise? Does waiting time in an MPI collective come from an uneven distribution of work or from noise? Repeating measurements to increase confidence in results is potentially costly, as many repetitions are needed [12]. Although noise on the target system should be considered when optimizing an application [8], some aspects of the program execution—the aforementioned load imbalance, for instance—exist independently. Therefore, measurement techniques that are less sensitive to noise can identify these problems more reliably and cheaply.

In this work, we investigate a noise-resilient measurement technique based on the Lamport clock [13]. This algorithm is independent of a physical clock, relying instead on the logical order induced by messages between processes to assign timestamps. Consequently, the resulting program traces are independent of the actual timings and insensitive to noise. Lamport's algorithm works for any positive clock increment between events, which leaves room for adaptation. We explore various choices for the clock increment to model the effort spent in a given code region.

A. Contributions

We implement Lamport's logical clock algorithm in Score-P. Furthermore, we extend the basic algorithm with four methods to increment the clock between events, counting either OpenMP loop iterations, LLVM [14] basic blocks, LLVM statements, or CPU instructions. Based on measurements of three established OpenMP+MPI mini-apps, we evaluate the usefulness of these methods in the context of the Score-P/Scalasca workflow. The central question for evaluation is which performance problems can be detected from a Scalasca report based on logical event traces. On the one hand, we demonstrate that measurements with the logical clock are robust against noise and, therefore, can represent intrinsic aspects of application performance, such as load balance, more

This work is part of the ExtraNoise project funded by the Deutsche Forschungsgemeinschaft (DFG) under Grant Number 449683531.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00192

reliably than measurements with a physical clock. On the other hand, by the same design, the logical clock is agnostic to aspects outside the program's direct control, rendering it unable to detect some common performance problems related to these. In this work, we clearly establish the strengths and limitations of the proposed measurement techniques and pave the way for further improvement.

B. Related Work

The variety of hardware, the range of scales (from single thread and single function to entire programs on full clusters), and the multitude of programming languages and paradigms lead to many specialized tools for assessing application performance. The Virtual Institute – High Productivity Supercomputing Tools Guide [15] provides a structured overview of the available tools and the questions they help to answer. Other notable measurement tools are Tau [16] and Extrae [17].

Many tools for visual or semi-automatic trace analysis exist. Vampir [18] visualizes traces in a timeline. Ravel [19] employs the logical clock to visualize the structure of application traces. In contrast to our implementation, the logical time stamps are computed in a post-processing step. Extra-P [20] aims to identify possible bottlenecks at a larger scale from extrapolation of performance models obtained from a series of small-scale runs.

Extensive work exists on quantifying noise in HPC systems and its impact on application performance. We mention only a few publications: Zhai et al. use time variability in fixed work quanta already present in most applications to quantify noise [21]. Voevodin et al. develop a tool to monitor noise on a cluster on a daily basis by executing a small Netgauge workload after every job [22]. Ferreira et al. analyze the influence of system parameters on performance degradation due to OS noise [23]. Beni et al. investigate the bandwidth variability in the shared network of a cluster [5]. Tools like HPAS [7] allow the injection of additional, easily adjustable noise to study an application's sensitivity to various noise sources.

Despite this, to our knowledge, little work has been done to make performance analysis more robust to noise. Ritter, Tarraf et al. performed extensive measurements to identify hardware counters less sensitive to noise [24]. Ritter, Geiß et al. use neural networks to extract performance models from noisy performance data [25].

C. Outline

In Section II we introduce the logical clock and describe the various methods to model computational effort. We outline the implementation in the measurement tool Score-P [1]. Section III summarizes Scalasca's method of automatic wait state analysis and the implications of using a logical clock in this setting. Then, in Section IV, we describe the system specification, the benchmark applications, and the configurations used in our experiments. The main part of this work is in Section V, where we discuss experimental results in detail. Finally, we summarize our insights and discuss possible future work in Section VI.

II. Logical clock models

In the abstract sense, a clock is a function that assigns numbers (time stamps) to events. The logical clock described by Lamport in [13] introduces an order that respects causal relations to events in a distributed system (parallel program). Causality does not depend on physical time. It is sufficient to have a trace record of the events in all processes, and the messages passed between them. An event a could have influenced event b if either a and b are events on the same process and a is recorded before b, or a is the sending of a message to a different process and b is the receipt of that message. This is written as the transitive *happens before* relation $a \to b$. A logical clock C assigns the time stamps such that the clock condition holds: If $a \to b$, then $C(a) < C(b)$. This can be achieved via a process-local counter C_i that follows this simple algorithm:

Algorithm 1 Logical clock.

For event a on process i:

1) Increment the local counter C_i.
2) a) If a is the sending of a message M, attach the current value C_i to that message as $M.C$.
   b) If a is the receipt of message M, set $C_i \leftarrow \max(C_i, M.C + 1)$
3) Record the time stamp $C(a) = C_i$.

This general definition of a logical clock still leaves many choices for the concrete implementation. In particular, the definition of a process, what counts as an event, and the value of the increment in the first step. Here, we apply the concept to tracing parallel programs with Score-P. In the case of a hybrid OpenMP+MPI program, Score-P considers each OpenMP thread on each MPI rank as a separate location: threads correspond to processes in the logical time model. The events of the logical clock are exactly the events written by Score-P into the trace file. The selection of recorded events is highly configurable, e.g. via command line options and filters. Each (unfiltered) function entry and exit is recorded with automatic compiler instrumentation.

The logical clock has two main advantages over physical clocks. Firstly, the event time stamps are guaranteed to be in the right order because of the clock condition. Physical clocks can be out of sync, necessitating time-stamp correction algorithms that skew the results. Secondly, logical clocks are insensitive to noise, i.e., to factors outside of program control that influence the run-time behavior.

In programs relying on nondeterministic MPI semantics, such as wildcard receives, the happens-before relation is insufficient to detect all causalities. In this case, messages can be matched differently depending on the timing, therefore the event order and logical time stamps might vary between executions. Although we are aware of improved clock algorithms,

such as the vector clock or the lazy Lamport clock protocol [26], which remedy this situation partially, we rely on the original Lamport clock in this work. For the benchmarks in IV this is sufficient to produce deterministic traces.

A. Effort models

If the logical time stamps are only needed for ordering events across processes, an increment of one is sufficient. However, Scalasca's analysis also assigns additional meaning to the time stamps: Differences in time stamps translate to the duration of code regions and wait states. The increment in Algorithm 1 should, therefore, reflect the amount of work spent between two events. We investigate five variants to estimate the effort of code regions:

- lt_1: Increment the counter by one.
- $\mathsf{lt}_{\mathsf{loop}}$: Increment the counter by one before each event. Additionally, increment the counter in each OpenMP loop iteration.
- $\mathsf{lt}_{\mathsf{bb}}$: Increment by one plus the number of LLVM IR[1] basic blocks executed since the last recorded event. The extra increment by one ensures that time stamps are strictly increasing. Also, count calls into external functions as X basic blocks.
- $\mathsf{lt}_{\mathsf{stmt}}$: Same as $\mathsf{lt}_{\mathsf{bb}}$ but counting LLVM statements. Count calls into external functions as Y statements.
- $\mathsf{lt}_{\mathsf{hwctr}}$: Increment by the difference in the value of some hardware counter (e.g. number of instructions) since the last recorded event.

lt_1 is the original logical clock that serves as a baseline. $\mathsf{lt}_{\mathsf{loop}}$ is a coarse but easy-to-implement estimate under the assumption that most work in an OpenMP program happens inside parallel loops and is the same for all iterations. The modes $\mathsf{lt}_{\mathsf{bb}}$ and $\mathsf{lt}_{\mathsf{stmt}}$ rely on a plugin for LLVM [14], adding further instrumentation to the program to count the number of basic blocks or statements. The additional counts are an ad-hoc solution to estimate effort in regions outside the instrumented code. We increase the counter by a constant $X = 100$ basic blocks or $Y = 4300$ statements whenever there is a call to OpenMP (parallel, for, fork, join). We do not assign extra effort to other external functions. These numbers are fitted to our observations in the LULESH benchmark (see V-C3), which spends a significant portion of the time inside the OpenMP runtime. The estimates are specific to this particular experiment and not reliable in general. A more sophisticated model might base estimates on micro-benchmarks on the target system.

For $\mathsf{lt}_{\mathsf{hwctr}}$, we consider the Linux perf counter `PERF_COUNT_HW_INSTRUCTIONS`. Note that as shown by Ritter et al. [24], this counter is subject to noise but more resistant to it than run-time. An advantage of hardware counters is that they also count effort spent in regions not seen by the instrumentation, e.g. inside calls to library functions.

[1]LLVM Intermediate Representation: A hardware-independent abstract assembly language

B. Implementation in Score-P

In Score-P, the logical clock is implemented as an additional timer by storing a counter on each location and updating it during measurement according to the logical clock algorithm. Steps one and three of Algorithm 1 are implemented by incrementing the counter each time it is read.

Step two requires synchronization of counters between locations. To capture causalities correctly, it is important that the event model is compatible with the logical clock, i.e., all communication across threads and ranks has to be recorded as an event. In the current implementation, only MPI and OpenMP communication is supported. MPI communication is instrumented via the PMPI tools interface. We implemented point-to-point communication as well as non-blocking collectives on intra-communicators. Schulz et al. [27] discuss several mechanisms to attach piggyback messages to MPI point-to-point communication. We choose to send extra messages to synchronize counters, because it is easy to implement incrementally inside Score-P's existing MPI wrappers. The OpenMP instrumentation is implemented with the source-to-source translation tool Opari2. Our implementation supports barriers, loops, fork/join and critical regions.

III. Performance analysis

Additional tools are needed to gain insights from the huge amount of trace data. Scalasca [2] automatically searches for certain patterns in a trace and aggregates this information over time to produce a parallel profile.

A profile stores information along the three dimensions metric, call path, and system resource. The axes for these dimensions are trees: The root of the call path tree is the main function, each leaf corresponds to a stack frame. The root of the system tree represents the entire job allocation, whereas the leaves represent individual threads. Queries can be fine-grained or aggregate over children in each dimension. Examples are: How much time does thread 0 spend in function `foo`? How many CPU seconds does the program consume in total? The Cube browser [28] provides a convenient graphical interface to explore these profiles.

Patterns extracted by Scalasca are represented as additional metrics in such a profile. For example, the metric *time* can be divided into time spent in computation, MPI calls, OpenMP calls, and idle threads. Those are further subdivided into the various communication patterns and wait states. Figure 1 shows a selection of the child metrics of *time* used in the analysis part of this work.

The classical example of a wait state is the MPI late sender: A point-to-point communication in which the sending of the message is started later than the matching reception. The receiver is blocked waiting at least until the sender initiates the message transfer. The severity assigned to the wait state is the difference in time stamps between entering the `MPI_Send` and the `MPI_Recv`.

Scalasca also produces higher-order analysis results that are not grouped under *time* but are presented as additional metrics in the output profile. In the evaluation of our experiments, we

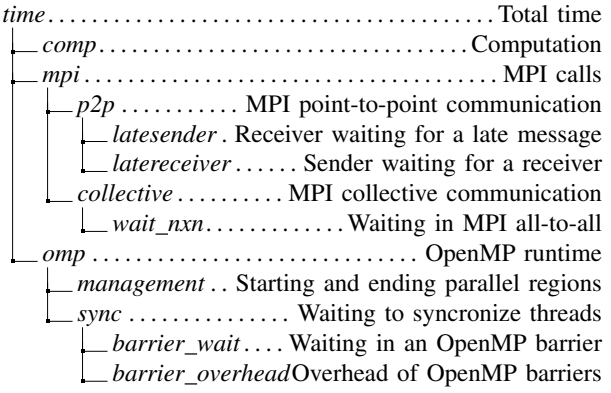

Fig. 1: Selected metrics in the analysis.

consider one such class of metrics: delay costs. For each wait state metric, there is a corresponding *delay cost* metric that shows the root causes for this wait state. For instance, the delay cost for the late sender shows the call paths and locations that are responsible for the wait time at the receiver. The delay costs are often more useful in pinpointing performance problems than the wait state metrics. For example, the wait time in an MPI all-to-all exchange is attributed to the MPI call itself, while the corresponding delay cost highlights the imbalanced functions causing the wait time.

Intervals measured with different clocks have different interpretations and severity values for metrics cannot be compared directly. We normalize all values by the total severity of the *time* metric to obtain dimensionless values. These values should be interpreted as fractions of the total reported effort for a given effort model. When describing measurements, we use 'time' to refer to the *time* metric.

We begin the exploration of a measurement generally asking two types of questions. Firstly, 'What percentage of the time is spent in useful computations vs. MPI calls, idle threads, etc.?' Secondly, 'What call paths contribute the most to these metrics?', e.g. 'Which functions are responsible for most of the idle time?' The answers determine the direction for deeper investigations. Answers to the first type of question, e.g. 'The application spends $5\%_T$ in MPI', involve fractions of the total *time* in percent, denoted with the symbol $\%_T$. In the Cube browser, one sets the metric view to 'Own root percent' to obtain these values. The second type of question could be answered with 'The functions `comm_boundaries` and `reduce_results` are responsible for $80\%_M$ and $10\%_M$ of the MPI time'. Here, the symbol $\%_M$ denotes a fraction relative to the value of a given metric. To display these values in Cube, one sets the call path view to 'Metric selection percent'. Usually, the ranking of call paths and the order of magnitude of their contributions is more important than the exact percentages.

The investigation usually proceeds with the highlighted wait states and corresponding call paths. Visual comparisons are, therefore, useful in this study because they represent the exploration done in a typical performance analysis. To compare measurements obtained from different clocks, we plot the results as stacked bars next to each other. For instance, Figure 5a compares contributions of various call paths to the metric *comp* in the `MiniFE-1` experiment.

Scalasca helps to show where and how parallel execution of a program is inefficient. It is still up to a human expert to interpret the analysis results and decide how to modify the code and configuration. Thus, the central questions for the evaluation of the following experiments are

- Can we draw useful conclusions from a Scalasca report based on logical event traces?
- Which kinds of performance problem can we detect?
- In which settings does the method fail, i.e., when do the analysis results become inconclusive or even misleading?
- How closely is real-time in each code region modeled by the clock increment?

IV. Benchmarks and configurations

A mini-app is a program designed to represent the performance-critical workload of an HPC application while keeping code complexity to a minimum. We select three such mini-apps, all proxies for grid-based physics simulations, for our investigation: MiniFE [29], LULESH [30], and the C++ port of TeaLeaf [31]. Currently, the logical clock is implemented only in MPI and OpenMP, which excludes codes using other parallel paradigms. Because the implementation of $\mathsf{lt}_{\mathsf{bb}}$ and $\mathsf{lt}_{\mathsf{stmt}}$ depends on LLVM, we restrict the selection to C/C++ codes.

A. Hardware specification

All measurements are taken on the standard nodes of the Jureca-DC cluster [2], with the following hardware specification:

- Processors: $2 \times$ AMD EPYC 7742 (2×64 cores@2.25 GHz)
- Memory: 512 GB DDR4@3200 MHz (8 NUMA domains with 64 GB each)
- Interconnect: InfiniBand HDR100 (NVIDIA Mellanox Connect-X6)

B. General workflow

Many factors influence benchmark performance. On the job level, influential options are the number of MPI ranks, the number of OpenMP threads per rank, the distribution of ranks/threads on the hardware, and the pinning of threads to CPUs. Application-specific options may influence the total amount, the type, and the distribution of work onto ranks. We run each benchmark without instrumentation with varied configurations and collect the benchmark's performance results. All benchmarks report at least the time to completion. MiniFE and LULESH also report a Figure of Merit, i.e., the rate at which the application completes a unit of work. We use these figures to carry out preliminary scaling studies, which already

[2]https://apps.fz-juelich.de/jsc/hps/jureca/configuration.html

indicate possible causes for performance loss. Then, we select a few interesting configurations for the detailed analysis.

To obtain reference timings, the application is run five times without instrumentation. Then, we perform an instrumented measurement and Scalasca trace analysis with the physical clock provided by the time stamp counter of x86-64 (denoted tsc), and each of the logical clocks lt_1, $\mathsf{lt}_{\mathsf{loop}}$, $\mathsf{lt}_{\mathsf{bb}}$, $\mathsf{lt}_{\mathsf{stmt}}$, $\mathsf{lt}_{\mathsf{hwctr}}$. Time measurements, needed to determine measurement overhead, are subject to noise. Additionally, tsc and $\mathsf{lt}_{\mathsf{hwctr}}$ measurements are noise sensitive. Therefore, we repeat every measurement five times. We base our evaluation in Section V on the arithmetic mean of the five call-path profiles from the repeated measurements.

In the following, we briefly describe the mini-apps and the configurations we chose for our experiments.

C. MiniFE

MiniFE is designed to model performance characteristics of typical finite element codes. First, a sparse linear system is assembled, which is then solved by an iterative method—conjugate gradient (CG) descent without preconditioner. There is an option to introduce imbalance across MPI ranks. An imbalance of 50% means that one-half of the ranks is assigned three times as many elements as the other half.

We choose these two configurations:

a) `MiniFE-1`*:* Single node, 8 MPI ranks (one per NUMA domain), one thread per rank. The grid has 400^3 elements in total. We set artificial imbalance to 50%. This configuration is somewhat untypical because it leaves most of the node unused. However, we select it because it presents a single, well-defined performance problem that should be detected easily by logical measurements.

b) `MiniFE-2`*:* This configuration is the same as `MiniFE-1`, except with 16 threads per rank. Because the job uses one entire node, it is more representative of a typical system use. In addition to the imbalance, there is another performance problem that should be detected by the logical measurements: parts of the matrix assembly are single threaded. We can also expect a performance decrease in CG due to contention for memory bandwidth among the threads.

D. LULESH

LULESH is a proxy app for hydrodynamics simulations [30] in a suite developed by the Lawrence Livermore National Laboratory.

The cubic domain is decomposed with a regular hexahedral grid, where some field quantities are associated with the nodes and some with the elements. Each time step has three phases: First, the global time step size is computed in function `TimeIncrement`, relying on an `MPI_Allreduce`, thus synchronizing ranks. Then, the node-centered quantities are updated in `LagrangeNodal`, and finally, the element-centered quantities are updated in `LagrangeElements`. Processes exclusively use point-to-point communication to access grid quantities on neighbors.

LULESH requires a cube number of MPI ranks such that each subdomain is a cube and each rank has the same number of elements. There is an option to introduce artificial work imbalance into the routine `ApplyMaterialPropertiesForElems`.

We set the number of grid elements per rank to 50^3 and use four threads per rank.

a) `LULESH-1`*:* This configuration uses 64 ranks and fills exactly two nodes. The default artificial imbalance is enabled. We can expect inefficiencies from multiple sources: load imbalances across ranks, contention for memory access, OpenMP overhead and waiting times, and sequential code.

b) `LULESH-2`*:* We chose this configuration because the logical measurements should be unable to detect the prominent performance problem. Artificial imbalance is disabled. The job uses 27 ranks on one node, which cannot be distributed evenly across NUMA domains. Three NUMA domains are filled completely with four ranks (16 threads) each. The other five domains are assigned three ranks (12 threads) each. The main performance problem is the uneven contention for memory bandwidth.

E. TeaLeaf

TeaLeaf, originally a Fortran code that has been ported to C++ [31], solves the heat conduction problem in two space dimensions. Spatial derivatives are approximated via five-point finite differences. The time steps are computed with an implicit method, relying on an iterative CG solver for the linear system.

The chosen configurations compute the predefined benchmark `5e_4_4` (4000^2 cells, simulated time $4s$) on one compute node:

- `TeaLeaf-1`: 1 MPI rank, 128 OpenMP threads per rank: Distributes threads across sockets.
- `TeaLeaf-2`: 2 MPI ranks, 64 OpenMP threads per rank: Each rank fills exactly one socket. This is the optimal configuration on Jureca-DC.
- `TeaLeaf-3`: 8 MPI ranks, 16 OpenMP threads per rank: Each rank fills one NUMA node.
- `TeaLeaf-4`: 128 MPI ranks, 1 OpenMP thread per rank: Loses performance in the MPI all-to-all exchanges

The problem fits neatly into L3 cache: There is $8 \times 4 \times 16MB = 512MB$ L3 cache on the node, and the main calculation operates on $4000^2 \times 4 = 64M$ double values. Table II shows timings for the configurations and the overhead of instrumentation with tsc. The optimal configuration filling one node is 2 MPI ranks with 64 OpenMP threads each. In this configuration, each rank occupies one of the two sockets. This reduces the cost of the frequent MPI all-to-all exchanges and avoids thread communication across sockets.

V. Results

First, we investigate the measurement overhead of the various timer modes. Then, we compare trace analysis results obtained with different timers with the Jaccard score before we discuss the results for each benchmark in detail.

Mode	Overhead / %				
	MiniFE-2			LULESH-1	TeaLeaf-2
	init	solve	total		
tsc	−14.3	0.3	−6.5	3.1	41.5
lt_1	−12.2	0.3	−5.3	3.6	40.5
lt_{loop}	−15.7	0.2	−6.9	4.3	42.5
lt_{bb}	97.8	0.2	47.9	23.5	48.0
lt_{stmt}	94.5	0.2	46.6	23.9	43.7
lt_{hwctr}	89.9	0.4	41.5	14.7	56.5

TABLE I: Measurement overheads for selected configurations and the various clocks.

A. Overhead

Using the reference measurement, we determine the overhead of the various modes, as shown in Table I. Keeping overhead small is especially important for physical measurements because overhead degrades their accuracy. With filter files and various other options, Score-P can be configured to ignore events, thus incurring less overhead and consuming less memory. As a rule of thumb, we specified filters to keep the overhead for tsc measurements reasonably small, i.e., at approximately 5% or below penalty to total run-time. This is not always possible, though.

The MiniFE benchmark consists of two main distinct phases: matrix initialization and CG-solver. The sensitivity of their run-time to noise and/or overhead is different. Measurement overhead stems to a large part from the initialization phase, whereas the overhead in the solver phase is negligible. The total overhead depends on the ratio of initialization time to solver time, which depends on the benchmark configuration in turn. Figure 2 shows individual and averaged run times of the initialization phase in MiniFE-2 for each measurement method. For this particular experiment, tsc and the low-effort logical timers lt_1 and lt_{loop} even produce a negative overhead on average, i.e., the program runs faster with instrumentation. A possible explanation is that measurement induces a desynchronization between threads, which has also been observed by Afzal et al. [32] to increase performance in memory-bound codes. In contrast, the logical clocks lt_{bb}, lt_{stmt}, and lt_{hwctr} introduce overhead on the order of 100% in this phase. We do not show a plot for the CG phase, as none of the methods produce any significant overhead, and the run-to-run variation between timings is negligible.

In the LULESH experiments, we observed below 1% run-to-run variation in timings. The overhead of the lt_1 and lt_{loop} methods is slightly greater than in tsc measurements. lt_{bb}, lt_{stmt} have the greatest overhead.

Despite filtering, the TeaLeaf experiment suffers from high overhead, between 40% and 56%. We observed the greatest overhead with the lt_{hwctr} counter. This is caused by Score-P interfering with the cache, as we discuss in more detail in Section V-C5.

The accuracy of logical measurements does not suffer from overhead because they are invariant to the timing of events. However, greater overhead means that experiments are more

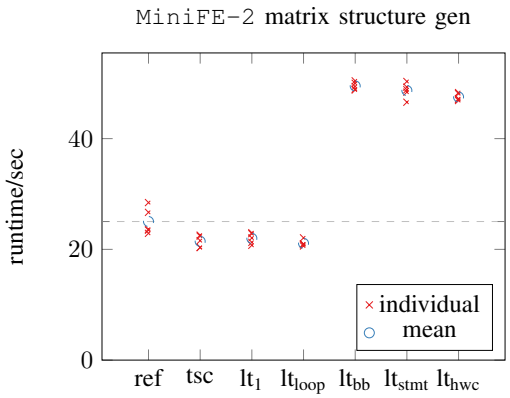

Fig. 2: MiniFE-2: Run-time for matrix structure generation.

costly to run. Broadly speaking, the methods lt_1 and lt_{loop} are less intrusive than lt_{bb}, lt_{stmt}, and lt_{hwctr}. Compared to tsc, the additional overhead of the less intrusive methods is negligible. In some instances, the overhead is even smaller than with tsc, despite the additional work done in OpenMP and MPI synchronizations. With the more intrusive methods, one should be prepared for 1.5 times longer runs compared to tsc.

B. High-level comparison of methods

We describe the three dimensions of a profile, i.e., metric, call path, and system, in Section III. A profile provides mappings from those dimensions to normalized severity values, for instance, $(\text{metric}, \text{call path}) \mapsto \text{severity}$ in $\%_T$. We use a generalized Jaccard score to measure the similarity of two such mappings from different measurements.

The original Jaccard score is a measure of similarity between two sets. It is computed as the ratio of the size of the intersection to the size of the union:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}. \tag{1}$$

Here, we generalize this idea to functions: Given a discrete definition set X and two functions $A, B : X \mapsto \mathbb{R}_{\geq 0}$, their Jaccard score is computed as in Equation 1, with adapted notions of intersection and union:

$$\begin{aligned} |A \cap B| &:= \sum_{x \in X} \min(A(x), B(x)), \\ |A \cup B| &:= \sum_{x \in X} \max(A(x), B(x)). \end{aligned}$$

This definition is analogous to Costa's generalization of the Jaccard score to multi-sets [33], which are functions mapping to $\mathbb{N}_0$ instead of $\mathbb{R}_{\geq 0}$. The score is zero when the functions have disjoint support, one when the functions are identical, and otherwise, it assumes any value in between.

In the following, we consider the scores:

- $J_{(M,C)}$ for the mapping from $(\text{metric}, \text{call path})$ pairs to their contributions to run time (in $\%_T$). Figure 3 shows $J_{(M,C)}$ scores comparing the various logical timers to tsc

in the MiniFE and LULESH experiments. Figure 4 shows the same for TeaLeaf.

- J_C^{metric} for the mapping from call paths to their relative contributions to a given metric (in $\%_M$). We show these scores in the bar plots 5, 6 and 9.

According to the $J_{(M,C)}$ score, lt_1 and lt_{loop} perform generally worse than the more advanced methods lt_{bb}, lt_{stmt} and lt_{hwctr}. In almost all experiments, lt_1 has the lowest score. But there is no clear best method for all benchmarks. lt_{bb} has the highest score of all methods in `MiniFE-1` and all TeaLeaf experiments, but the lowest score in the LULESH experiments. lt_{stmt} scores highest in `MiniFE-2` and `LULESH-2`, lt_{hwctr} is best in `LULESH-1`.

The configurations `MiniFE-1`, `MiniFE-2`, `LULESH-1`, and `LULESH-2` were designed to be increasingly challenging for logical methods. Indeed, for each of the lt_{bb}, lt_{stmt} and lt_{hwctr} methods, the score decreases in the same order. The lt_{stmt} method performs most consistently across benchmarks, whereas lt_{bb} shows the largest difference across benchmarks.

Figures 3 and 4 also show the minimal Jaccard score between any pair of the five repetitions of each measurement. This shows how much the analysis results vary from run to run, i.e., how susceptible the analysis is to noise. In all experiments, this minimal run-to-run score is above 0.9 for tsc measurements. The lt_{hwctr} measurements show generally a slightly larger variation, i.e., a lower run-to-run score than tsc. In the `TeaLeaf-2` experiment, we observe a minimal score of only 0.67, meaning that the lt_{hwctr} measurement is much more susceptible to noise than tsc. At first sight, this observation is in contradiction with the results from Ritter, Tarraf et al. [24], who observe a lesser run-to-run variability in instruction counters compared to time measurements. However, their measurements also indicate a high counter variability for a few call paths that contribute little to total run time. Their evaluation is concerned with plain profiles recording the total time/total counter per call path, whereas our evaluation also includes the additional metrics from Scalasca's wait state analysis. Our findings indicate that wait state analysis is influenced differently by noise than plain profiling. Of course, all other logical measurements are exactly the same in each repetition, and therefore, the minimal run-to-run score is one.

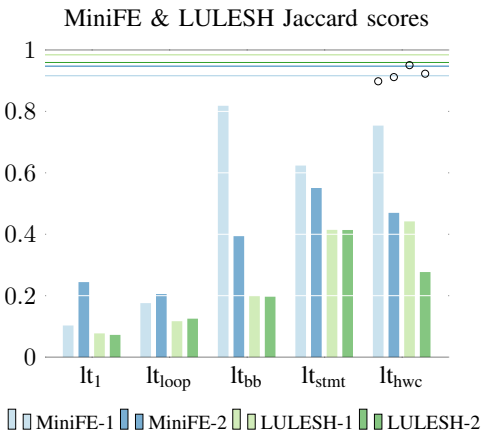

Fig. 3: Similarity of the logical measurements to tsc, according to Jaccard score for (metric, call path) contributions. Bars—grouped by measurement method and color-coded by experiment—show $J_{(M,C)}$ between tsc (average) and the logical measurement (average). The lines show the minimal $J_{(M,C)}$ between the repetitions of tsc, color-coded by experiment. Circles show minimal $J_{(M,C)}$ between repetitions of the lt_{hwctr} measurements.

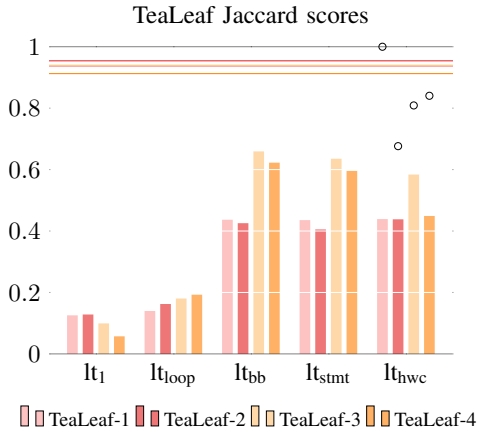

Fig. 4: Similarity of logical measurements to tsc, according to (metric, call path) contributions ($J_{(M,C)}$). See Fig. 3 for a detailed description.

C. Detailed analysis

1) `MiniFE-1`: The tsc measurement reports that only $60\%_T$ of the time is spent in computation. The first bar in Figure 5a shows a breakdown of computation time by call path. The various stages of matrix assembly contribute slightly more than $50\%_M$ to the computation time, while the rest is spent in the CG solver, mainly in computing matrix-vector products ($37\%_M$). Most of the remaining time ($38\%_T$) is spent waiting in MPI all-to-all exchanges. As shown in Figure 6a, the three most important call paths are `make_local_matrix` ($44\%_M$), `cg_solve/dot` ($31\%_M$), and `generate_matrix_structure` ($20\%_M$). The waiting does not manifest in the compute-intensive call paths but in those procedures that rely on MPI all-to-all exchanges. However, the delay-cost analysis (not shown here) highlights the compute-intensive functions.

Conclusions from tsc: There is a large computational imbalance in all parts of the code, which is, relative to compute time, worse in the matrix assembly than in the solver.

The logical measurements attribute similar portions of the time to computation (between $62\%_T$ and $68\%_T$) and all-to-all wait time (between $28\%_T$ and $38\%_T$). That is, all measurements indicate a computational imbalance between MPI ranks. The lt_{loop} method also reports late sending of point-

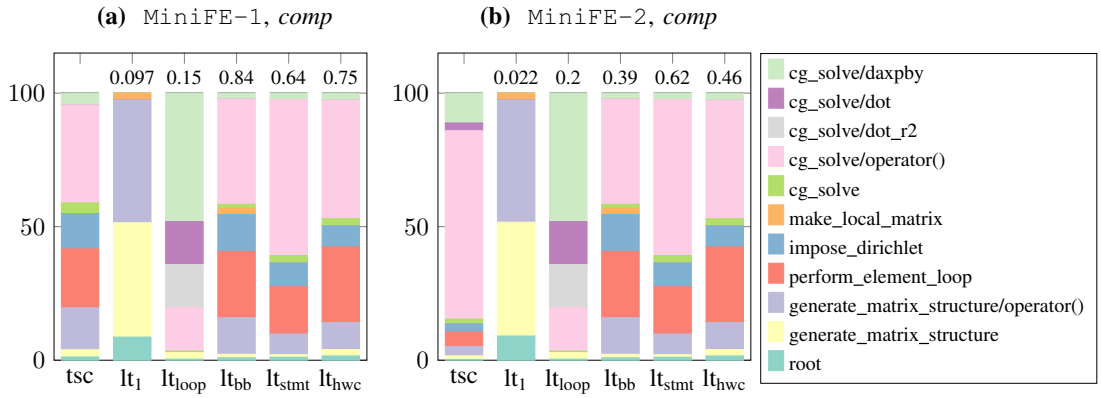

Fig. 5: `MiniFE-1` and `MiniFE-2`: Contributions of selected call paths to user computation (metric *comp*, in $\%_M$).

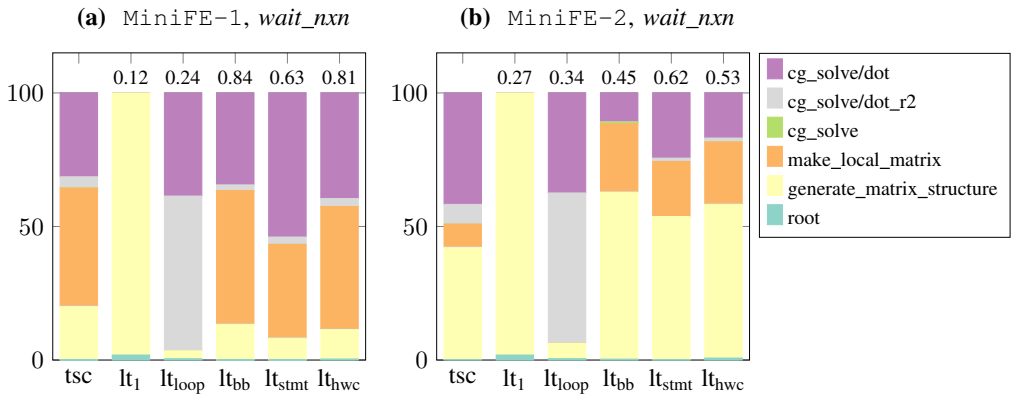

Fig. 6: `MiniFE-1` and `MiniFE-2`: Contributions of selected call paths to all-to-all wait time (metric *wait_nxn*, in $\%_M$).

to-point messages as a minor problem with $6\%_T$. This is not seen in tsc and therefore misleading.

However, the attribution of these times to the call paths differs significantly between the measurements, as seen in the other bars in Figures 5a and 6a. Unsurprisingly, lt_1 highlights parts of the code that contain many inexpensive function calls, i.e., the matrix assembly. The lt_{loop} measurement overemphasizes regions with many inexpensive loop iterations, i.e., the vector operations in the CG solver. These methods fail to estimate the effort of code regions even in this simple setting and cannot be trusted to identify call paths responsible for the imbalance. The lt_{bb}, lt_{stmt}, and lt_{hwctr} measurements are in good agreement with tsc regarding the contribution of call paths to the metrics. Results obtained with these methods support the same interpretation as the tsc measurement.

2) `MiniFE-2`: This configuration uses 16 OpenMP threads per rank. The tsc result shows that the program spends most of its run time in idle threads ($58\%_T$) and only $39\%_T$ in useful computation. Waiting in all-to-all exchanges takes $2\%_T$ of the total time. Since MPI is single-threaded in this application, the wait time is responsible for 15 times as much idle time in the worker threads, i.e., $30\%_T$ of the total time. This is the computational imbalance we have seen before in `MiniFE-1`. But the waiting time can explain only half of the idle time. The tsc measurement shows that `generate_matrix_structure/operator()` also contributes to idle threads with $35\%_M$, but does only $4\%_M$ of the computation. This procedure is single-threaded and, therefore, a candidate for easy performance gains. The `make_local_matrix` routine is also single-threaded and contributes another $6\%_M$ to idle threads.

The first bar in Figure 5b shows the contributions of call paths to computation time. Compared to the `MiniFE-1` experiment, the matrix-vector products contribute a greater portion ($70\%_M$). This indicates a reduced efficiency due to memory contention between threads.

According to tsc, the application spends only $0.6\%_T$ time in OpenMP and almost all of that in waiting at OpenMP barriers. The overhead of OpenMP constructs is negligible. There is little potential for optimization here.

Figure 7 shows the contributions of paradigms to the total run time for each method. All measurements report a similar time in MPI, but the ratio of idle threads to useful computation varies. Because most work in the OpenMP parallel regions does not involve additional function calls, lt_1 shows no effort in the worker threads ($93\%_T$ idle threads = $15\times6\%_T$ com-

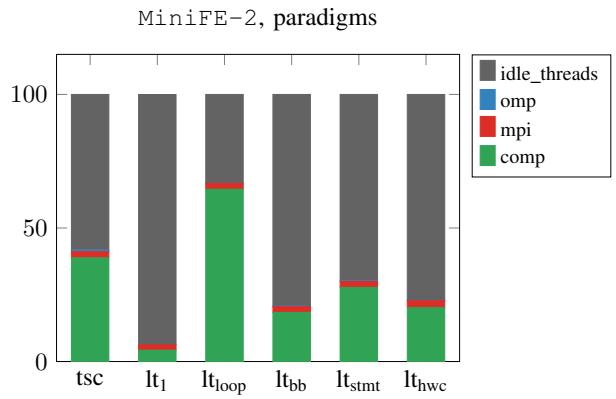

Fig. 7: MiniFE-2: Time spent in user computation, OpenMP, MPI, and idle threads, relative to total run time (metrics *comp, mpi, omp, idle_threads*, in $\%_T$).

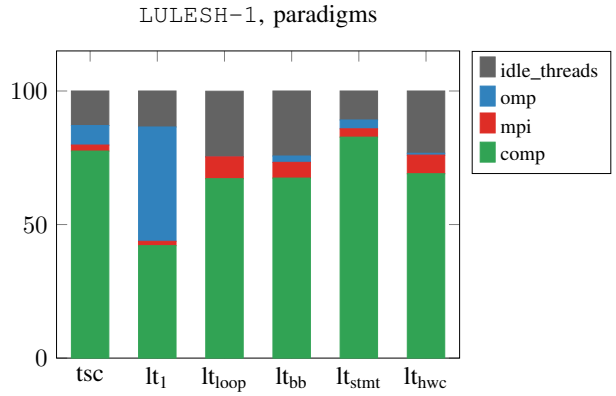

Fig. 8: LULESH-1: Time spent in user computation, OpenMP, MPI, and idle threads, relative to total run time (metrics *comp, mpi, omp, idle_threads*, in $\%_T$).

putation + MPI). In contrast, lt_{loop} focuses on the work done in OpenMP loops. In this measurement, the $2.1\%_T$ MPI time explains almost all the total idle time($33\%_T$). Therefore, lt_{loop} is unable to detect idle threads due to serial regions. The other methods lt_{bb}, lt_{stmt}, and lt_{hwctr} are closer to tsc in this regard but tend to overestimate effort in the serial regions of the code. Regarding computational imbalance and serial regions, the lt_{bb}, lt_{stmt}, and lt_{hwctr} measurements support the same conclusions as tsc.

Figure 5b shows the contributions of call paths to the computation time for the various timer methods. The values reported by the logical measurements are the same as in Figure 5a for MiniFE-1. Since the code has no parallelization overhead that can be detected by the logical clocks, i.e., outside the OpenMP runtime, the total computational effort is the same. Consequently, the logical measurements cannot detect the memory contention issue.

3) LULESH-1: Figure 8 shows how the execution time is distributed to the paradigms. The tsc measurement reports that most time ($78\%_T$) is spent in computation. The program spends $2\%_T$ in MPI. Since MPI runs single-threaded, this explains already $6\%_T$ idle time. Another significant portion of the time is spent in OpenMP ($7\%_T$). The rest of the idle time indicates serial sections of the code. Both MPI and OpenMP are good candidates to look for optimizations.

According to tsc, half the MPI time ($1\%_T$) comes from waiting at all-to-all exchanges, while late senders in point-to-point communication ($0.5\%_T$) are a lesser problem. The rest of the time in this category is spent in the MPI library.

Figure 9b shows the call paths that cause the all-to-all wait time. Two functions are highlighted by the tsc measurement: CalcForceForNodes and ApplyMaterialPropertiesForElems. The second is the function where the artificial imbalance is applied. The first does not have an artificial imbalance. However, the first bar in Figure 9a shows that this function is responsible for most of the computation time. Therefore, minor imbalances in this function still cause most of the all-to-all wait time. The same call path is also responsible for most ($89\%_M$) of the wait time in point-to-point communication.

tsc attributes the $7\%_T$ in OpenMP mostly to waiting at barriers ($5\%_T$) and to a lesser extent ($2\%_T$) to overhead of the OpenMP runtime. Waiting at barriers is approximately proportional to computation time and, therefore, predominantly occurs in CalcForceForNodes. The overhead is caused mostly by ApplyMaterialPropertiesForElems, which contains many OpenMP loops doing little work each.

The logical measurements yield varied results. The lt_1 measurement reports similar waiting time in MPI all-to-all and late senders as tsc but does not show the call paths causing these issues correctly. This method does not find the computational imbalance across MPI ranks. Overhead of OpenMP runtime, and therefore the overall time in OpenMP is strongly overestimated. However, the overhead is correctly attributed to the material update routine, which contains many small OpenMP loops and is also responsible for most of the overhead shown in the tsc measurement. For the same reason, computation time is attributed to the wrong call paths.

lt_{loop} estimates computational effort slightly better but still overestimates the call path with many inexpensive OpenMP loop iterations. In MPI, the measurement reports only the all-to-all wait state, which it overestimates. Corresponding delay costs point to the material update routine, which contains the artificial imbalance. The measurement, therefore, enables the analyst to find the work imbalance. However, the wait time caused by the nodal calculations is not found. These calculations are perfectly balanced in terms of OpenMP loop iterations. Thus, the imbalance must have another cause, most likely timing variations of memory accesses. The lt_{loop} method cannot measure time inside the OpenMP runtime and therefore shows no OpenMP overhead. The measurement also does not report any waiting time in OpenMP, i.e., the number of loop iterations is balanced across threads and cannot explain the waiting at OpenMP barriers seen with tsc.

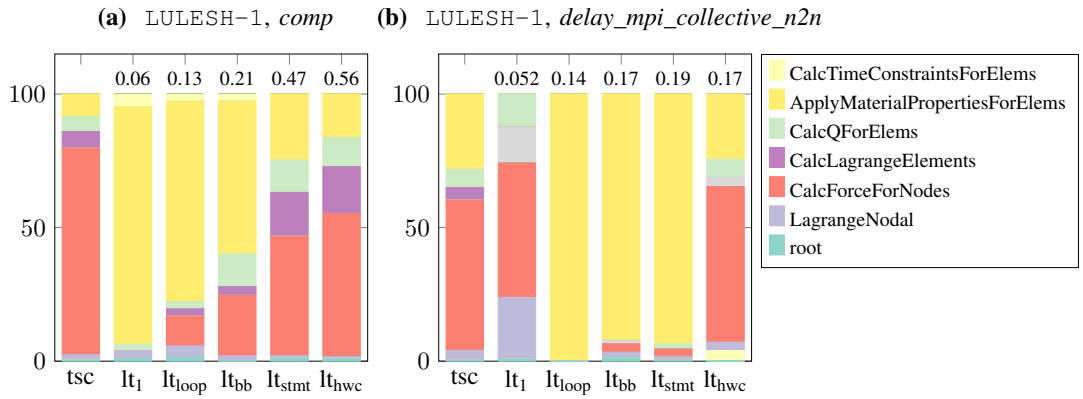

Fig. 9: LULESH-1. Contributions of selected call paths to user computation (metric *comp*, in $\%_M$) 9a, and to delay cost for MPI all-to-all wait states (metric *delay_mpi_collective_n2n*, in $\%_M$) 9b.

The lt_{bb} and lt_{stmt} measurements estimate time spent in computation, MPI, OpenMP and idle threads similarly to tsc, although overestimating the MPI time. With respect to MPI wait states, both measurements support the same interpretation as lt_{loop}, i.e., a computational imbalance in the material updates. Figure 9a shows that lt_{bb} and lt_{stmt} yield better estimates for the computational effort in individual call paths than lt_{loop}. lt_{bb} only presents a small improvement, whereas the lt_{stmt} estimates are significantly better. Because we fitted the constants from II-A, which estimate effort in the OpenMP runtime, to this particular experiment, both measurements report a similar time for OpenMP overhead as tsc. With this tuning, the measurements correctly highlight `ApplyMaterialPropertiesForElems` as the origin of most of the overhead. Similarly to the lt_{loop} result, the computations are balanced in terms of basic blocks and statements across threads.

The lt_{hwctr} measurement reports wait time for MPI all-to-all communication ($2.3\%_T$) and waiting for late senders ($2.6\%_T$). Both patterns are overestimated, and in contrast to tsc, the late sender pattern is more severe. As shown in 9b, delay costs for all-to-all wait time point to the right call paths in approximately the right amount. That is, lt_{hwctr} is the only logical measurement that shows an imbalance resulting from the nodal calculations. But the low Jaccard score ($J^{\text{delay}}_{(C)} = 0.17$)) for that metric indicates that lt_{hwctr} associates the delay costs with different procedures inside the `CalcForceForNodes` call path. In particular, tsc points to various OpenMP loops doing computation, whereas lt_{hwctr} points to an `MPI_Waitall` call. A possible explanation is that the nodal calculations are balanced in terms of instructions, but timing variations lead to waiting time, which shows as extra instructions inside the `MPI_Waitall` call. The computational imbalance in the material updates is also found correctly. Additionally, lt_{hwctr} is the only logical method that shows effort in the MPI library ($2\%_T$). No waiting in OpenMP barriers is reported, meaning that computations are balanced across threads with respect to instruction count. The OpenMP overhead is also found by lt_{hwctr} and correctly attributed to the material update routine. The hardware counters also yield the best estimate for computation effort among the logical timers.

Name	Ranks	time / sec.		overhead / %
		Ref.	tsc	
`TeaLeaf-1`	1	58.8	83.9	42.8
`TeaLeaf-2`	2	41.5	58.7	41.5
`TeaLeaf-3`	8	53.1	58.1	9.4
`TeaLeaf-4`	128	54.2	62.3	14.9

TABLE II: Run times and tsc measurement overheads for TeaLeaf.

4) `LULESH-2`: We included `LULESH-2` as an example of a configuration where the logical measurements fail to detect the dominant and obvious performance problem. The artificial imbalance is disabled in this experiment, and the work is distributed evenly across ranks. However, the ranks are not distributed evenly across the hardware of the node. Ranks on the fully occupied NUMA domains have less bandwidth available than the ranks on the partially occupied domains. In the tsc measurement, this manifests as a late sender wait state with $3.3\%_T$ that is caused mostly by `CalcForceForNodes`. lt_{hwctr} is the only logical measurement that reports the late sender wait state as the dominant issue, but locates it in the wrong call paths.

5) `TeaLeaf-1,2,3,4`: The tsc measurement of `TeaLeaf-2` reports that a large portion of the application run-time ($39\%_T$) is spent in OpenMP, which is classified as $11\%_T$ waiting at barriers and $28\%_T$ overhead. These observations on their own indicate major issues with the load distribution to threads and suboptimal use of OpenMP in general. However, Table II shows that the measurement caused a 40% penalty to overall run time in this configuration. With instrumentation, we do not observe a significant difference between the optimal `TeaLeaf-2` and the other configurations with more ranks, `TeaLeaf-3` and `TeaLeaf-4`.

The instrumentation consumes additional memory and

pushes the computation out of the cache. The resulting overheads are large enough to skew the analysis and are most likely the cause for the observed time in OpenMP. In this case, the tsc result is potentially misleading.

The lt_{bb}, lt_{stmt} and lt_{hwctr} report all below $2\%_T$ time as OpenMP overhead and between $2.3\%_T$ and $2.6\%_T$ waiting in barriers, therefore work in terms of statements or instructions is distributed almost evenly between threads.

In the `TeaLeaf-2` experiment, which uses only 2 MPI ranks, none of the measurements shows significant time in MPI. In configuration `TeaLeaf-3`, wait time in MPI all-to-all exchanges becomes noticeable ($0.4\%_T$ in tsc), and in `TeaLeaf-4`, which employs 128 ranks, this becomes the dominant problem ($12\%_T$ in tsc). Even with almost perfect load balance, minor timing variations lead to some waiting time in all-to-all exchanges. Only the lt_{hwctr} measurement shows the same problem($44\%_T$), whereas lt_{bb} and lt_{stmt} show little to no time in MPI. The wait time in barriers, which is not caused by uneven work distribution, is not shown by the logical measurements.

VI. Discussion

In this work, we investigated how useful time stamps from logical clocks are in a performance analysis workflow based on Scalasca trace analysis. Therefore, we extended the Score-P measurement system to record logical instead of physical time stamps. We implemented four simple models to estimate work done in a logical time interval based on counting OpenMP loop iterations, LLVM basic blocks, LLVM statements, and CPU instructions.

To compare the different clock models, we measured a few selected mini-applications with each clock. We used the Jaccard score to evaluate how much results overlap with the traditional time based methods. However, while Scalasca detects wait states and their causes automatically in large trace files, interpretation by human experts is still needed to understand application performance. Therefore, in addition to the scoring approach, we used our experience to interpret the measurements and compare the timer methods.

A. Conclusions

It is time to summarize the conclusions we draw from our exploratory experiments. Firstly, none of the simple logical time based techniques is a drop-in replacement for a physical clock. We consider these methods to be specialized tools to be used in conjunction with traditional measurements. Their usefulness very much depends on the specific scenario.

Wait states caused by contention for system resources, e.g. main memory, are not detected by design. If such a problem is present, the logical measurements are skewed and might be misleading. However, using the combined results from a physical and a logical measurement, it is possible to differentiate intrinsic wait states caused by uneven work distribution from extrinsic wait states due to uneven resource distribution.

The less intrusive methods lt_1 and lt_{loop} introduce little to no additional overhead compared to tsc. Overhead for the more intrusive methods is generally larger than for physical measurement, most notably with lt_{bb} and lt_{stmt}, which increase run time by 1.5 in some cases. Still, a single logical measurement is faster than repeating the experiment to account for noise. When the measurement overhead negatively impacts performance, the physical clock measurement might be misleading. In this case, the logical clocks have an advantage, as they are insensitive to their own overhead.

The logical clock lt_1 without an additional effort model is barely useful. Estimating work by the OpenMP loop count is sufficient to detect clearly defined load imbalances across ranks and threads when most work is done inside parallel loops of roughly the same complexity. However, the lt_{loop} model overestimates loops with many inexpensive iterations and imbalances in these regions. Additionally, it does not detect idling threads due to serial regions. This model is probably not useful in most scenarios.

The lt_{bb} and lt_{stmt} models make limited use of the compiler's representation of the code. Counting basic blocks or statements results in similar models with minor differences in the weighting of call paths. Both correctly show wait states caused by load imbalances and serial regions. However, time spent in library calls is not recorded, for instance, the overhead of MPI routines or OpenMP regions. In principle, one can improve the measurements by adding an effort model for library calls, as we did for OpenMP. In practice, this must be more sophisticated than assuming a constant amount of work in each library call.

The lt_{hwctr} model also correctly shows wait states caused by load imbalances and serial regions. Additionally, it can record effort in code not seen by the instrumenter at compile-time. This leads to a better representation of effort in MPI and OpenMP but comes at the cost of making measurements susceptible to noise again.

B. Future work

Towards employing logical measurement techniques in the performance analysis of production code, further work is required.

The LLVM-based clocks need better effort models for MPI, OpenMP, and other library calls. As [34], [35] have shown, such models would need to be hardware and vendor-dependent to be accurate. Assigning different weights for different kinds of statements might improve the model further. We considered only the number of instructions in the hardware counter model. Experiments with different hardware counters and combinations of hardware counters might lead to a better model. The validation of more complex effort models requires gathering and evaluating more data in a more automatic and quantitative way. Comparing measurements with the Jaccard score is a first step in this direction. An analysis method that combines physical time and logical time measurements could provide more insight than individual measurements.

References

[1] Score-P developer community. (2023, Apr.) Scalable performance measurement infrastructure for parallel codes (Score-P). [Online]. Available: https://doi.org/10.5281/zenodo.7817192

[2] M. Geimer, F. Wolf, B. J. Wylie, and B. Mohr, "A scalable tool architecture for diagnosing wait states in massively parallel applications," *Parallel Computing*, vol. 35, no. 7, pp. 375–388, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819109000398

[3] I. Zhukov, C. Feld, M. Geimer, M. Knobloch, B. Mohr, and P. Saviankou, "Scalasca v2: Back to the future," in *Tools for High Performance Computing 2014*, C. Niethammer, J. Gracia, A. Knüpfer, M. M. Resch, and W. E. Nagel, Eds. Cham: Springer International Publishing, 2015, pp. 1–24.

[4] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, "Run-to-run variability on xeon phi based cray xc systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–13.

[5] M. Salimi Beni, S. Hunold, and B. Cosenza, "Analysis and prediction of performance variability in large-scale computing systems," *The Journal of Supercomputing*, vol. 80, pp. 1–28, 03 2024.

[6] O. H. Mondragon, P. G. Bridges, S. Levy, K. B. Ferreira, and P. Widener, "Understanding performance interference in next-generation hpc systems," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 384–395.

[7] E. Ates, Y. Zhang, B. Aksar, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Hpas: An hpc performance anomaly suite for reproducing performance variations," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[8] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003, p. 55.

[9] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing variability in the io performance of petascale storage systems," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–12.

[10] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: performance degradation due to nearby jobs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[11] A. Jokanovic, G. Rodriguez, J. C. Sancho, and J. Labarta, "Impact of inter-application contention in current and future hpc systems," in *2010 18th IEEE Symposium on High Performance Interconnects*. IEEE, 2010, pp. 15–24.

[12] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807644

[13] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, 07 1978.

[14] LLVM admin team. (2023) LLVM website. Accessed 2023/08/21. [Online]. Available: https://llvm.org/

[15] (2023, Nov) VI-HPS Tools Guide. Accessed 2023/11/29. [Online]. Available: https://www.vi-hps.org/cms/upload/material/general/ToolsGuide.pdf

[16] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[17] Extrae. Accessed 2023/11/29. [Online]. Available: https://tools.bsc.es/extrae

[18] Vampir. Accessed 2023/12/05. [Online]. Available: https://vampir.eu/

[19] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann, "Combing the communication hairball: Visualizing parallel execution traces using logical time," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2349–2358, 2014.

[20] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA*. ACM, November 2013, pp. 1–12.

[21] J. Zhai, Y. Jin, W. Chen, and W. Zheng, *Lightweight Noise Detection*. Singapore: Springer Nature Singapore, 2023, pp. 165–197. [Online]. Available: https://doi.org/10.1007/978-981-99-4366-1_7

[22] V. Voevodin and D. Nikitenko, "Recurrent monitoring of supercomputer noise," *Supercomputing Frontiers and Innovations*, vol. 10, pp. 27–35, 01 2024.

[23] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. T. Pedretti, "The impact of system design parameters on application noise sensitivity," *Cluster computing*, vol. 16, pp. 117–129, 2013.

[24] M. Ritter, A. Tarraf, A. Geiß, N. Daoud, B. Mohr, and F. Wolf, "Conquering noise with hardware counters on hpc systems," in *2022 IEEE/ACM Workshop on Programming and Performance Visualization Tools (ProTools)*, Nov 2022, pp. 1–10.

[25] M. Ritter, A. Geiß, J. Wehrstein, A. Calotoiu, T. Reimann, T. Hoefler, and F. Wolf, "Noise-resilient empirical performance modeling with deep neural networks," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 23–34.

[26] A. Vo, G. Gopalakrishnan, R. M. Kirby, B. R. De Supinski, M. Schulz, and G. Bronevetsky, "Large scale verification of mpi programs using lamport clocks with lazy update," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 330–339.

[27] M. Schulz, G. Bronevetsky, and B. Supinski, "On the performance of transparent mpi piggyback messages," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 09 2008, pp. 194–201.

[28] P. Saviankou, A. Visser, and Cube developer community. (2023, Mar.) Cubegui: Graphical explorer. [Online]. Available: https://doi.org/10.5281/zenodo.7737411

[29] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[30] I. Karlin, J. Keasler, and R. Neely. (2023) LULESH website. Accessed 2023/05/26. [Online]. Available: https://asc.llnl.gov/codes/proxy-apps/lulesh

[31] (2023) TeaLeaf. Revision 5ee7d75. [Online]. Available: https://github.com/UoB-HPC/TeaLeaf/tree/master

[32] A. Afzal, G. Hager, and G. Wellein, "Desynchronization and wave pattern formation in mpi-parallel and hybrid memory-bound programs," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds. Cham: Springer International Publishing, 2020, pp. 391–411.

[33] L. da F. Costa, "Further generalizations of the jaccard index," 2021.

[34] C. Iwainsky, S. Shudler, A. Calotoiu, A. Strube, M. Knobloch, C. Bischof, and F. Wolf, "How many threads will be too many? on the scalability of openmp implementations," in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 451–463.

[35] S. Shudler, Y. Berens, A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf, "Engineering algorithms for scalability through continuous validation of performance expectations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1768–1785, Aug 2019.