LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-656141

Ordering Traces Logically to Identify Lateness in Parallel Programs

K. E. Isaacs, T. Gamblin, A. Bhatele, M. Schulz,
B. Hamann, P. Bremer

June 25, 2014

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Ordering Traces Logically to Identify Lateness in Parallel Programs

Katherine E. Isaacs*, Todd Gamblin†, Abhinav Bhatele†, Martin Schulz†, Bernd Hamann*,
Peer-Timo Bremer†

*Department of Computer Science, University of California, Davis, California 95616 USA
†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California 94551 USA
E-mail: *{keisaacs, bhamann}@ucdavis.edu, †{tgamblin, bhatele, schulzm, ptbremer}@llnl.gov

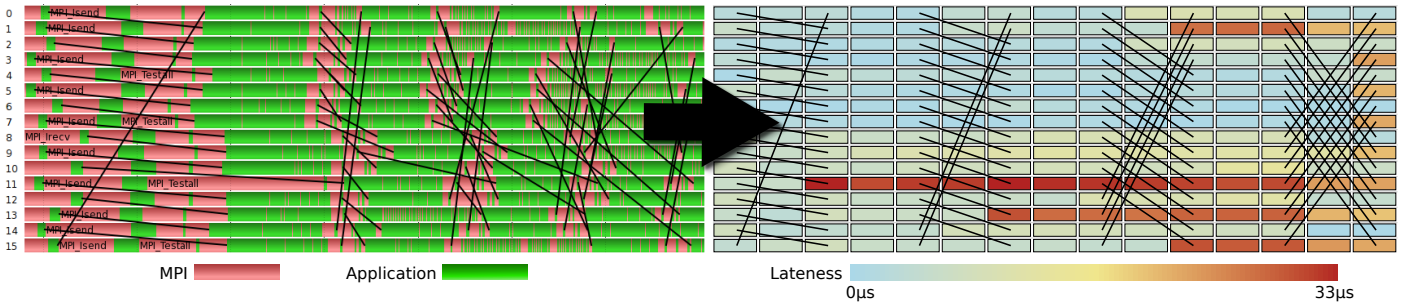


Fig. 1: Trace of a 16 process `MPI_Alltoall` using the dissemination implementation from libNBC [1]. From the raw physical time data, shown on the left using Vampir [2], we deduce a logical structure, visualized on the right, and use this structure to compute a novel lateness metric of each event, shown on the right using color. In this example, we can clearly see that the lateness from a receive on process 11 propagates to several other processes.

***Abstract*—Event traces are a valuable tool for understanding the behavior of parallel programs. Automatically analyzing a large trace, however, especially without a specific objective, is difficult. We aid this process by extracting a trace's *logical structure*, an ordering of events derived from their happened-before relationships. From this perspective, we can calculate delays relative to peers rather than duration and determine where they begin and how they propagate. The logical structure also acts as a platform for comparing and clustering processes as well as showing communication patterns in a trace visualization. We present an algorithm for determining this idealized logical structure from a trace and develop metrics to quantify delays and differences among processes. We implement this in our novel trace visualizer, Ravel, and apply our approach to several applications, demonstrating the accuracy of our extracted structure and its utility in analyzing these codes.**

I. Introduction

Writing an efficient, scalable parallel program that performs well on a range of parallel architectures is challenging. Achieving good performance usually involves several, often tedious, iterations of noticing performance problems, identifying their causes, and reworking the implementation accordingly. Tracing tools, which capture communication events and visualize them in timeline views (such as Vampir [2], which is shown in Fig. 1 on the left) have proven themselves as valuable tools, but also come with drawbacks limiting their wide spread use: Trace sizes are increasing rapidly as we run on a large number of processors or over a long duration (increasing the number of events traced) and complex communication patterns, common in many large scale codes, are hard to identify and comprehend. This is already evident even in the tiny example shown in the figure above.

We need new analysis and visualization tools to help users understand such complex traces and to aid in determining the information necessary to optimize the analyzed code. In this paper, we address this shortcoming by focusing on the underlying logical communication structure of a parallel program based on the happened-before relationships of all traced events at multiple granularities. This logical structure provides alignments of communication operations across processes, enabling clean visualizations that expose communication patterns, as can be seen in the right view of Fig. 1. Further, using the information from this logical structure, we define metrics that compare events to their logical peers, abstracting only the performance critical timing information. Specifically, we define *lateness*, which measures how delayed an event is relative to its peers, and *differential lateness* which measures how much delay was injected into the system at an event. Finally, we also provide a metric for comparing delay profiles of different processes.

Additionally, the logical structure provides the necessary basis for comparing and clustering processes and with that the ability to reduce the amount of information required to be shown in a trace visualization. This improves the scalability of both analysis and visualization and at the same time helps the user in focusing on the critical sections of the trace.

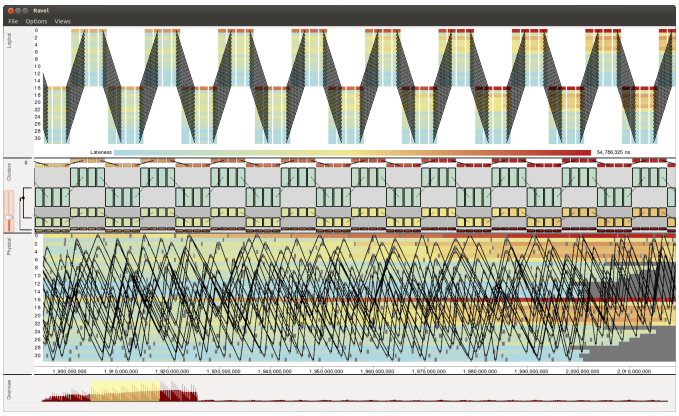


Fig. 2: Ravel interface showing multiple views of a pF3D trace colored by lateness.

We incorporate our structure extraction and metrics, as well as clustering approaches, in our interactive trace visualizer Ravel [3] (see Fig. 2).

In this work, we present our work within Ravel on an algorithm for determining an idealized logical structure from a trace and develop metrics to quantify delays and differences among processes. We define temporal metrics that we derive from the logical structure. In particular, we define "lateness" of events relative to the logical structure in a trace, which enables users to quickly identify delayed processes and the bottlenecks they cause. We use the LULESH shock hydrodynamics proxy application [4], the Multi-grid (MG) benchmark from the NAS Parallel Benchmarks (NPB) suite [5], [6], and pF3D [7], a laser-plasma interaction code, to illustrate our approach.

Further, we demonstrate the effectiveness of our logical structure extraction technique and the utility of our temporal metrics through three case studies. The applications used for these case studies are multiple implementations of collective operations in MPI [1], a sparse linear solver library [8], and an *in situ* analysis application for computing merge trees on a topological structure [9]. These applications are executed on an IBM Blue Gene/Q system and on an Infiniband Sandy Bridge cluster.

The major contributions of this work are:

- A set of rules and heuristics to extract the logical communication structure of a parallel program from its execution traces;
- Novel metrics, such as lateness, that help identify performance bottlenecks or delays in the execution;
- The integration of our analysis techniques into an interactive visualization tool, Ravel, to compute the structure and metrics above and visualize them; and
- Case studies demonstrating the accuracy of the new approach and how it detects and highlights a number of performance characteristics difficult to obtain from existing techniques.

II. Analyzing Parallel Execution Traces

We assume that a parallel trace for a message passing program consists of measured events that are either computation blocks, sends, or receives. We further assume, at a minimum, that an execution trace is a series of records of the enter and exit time of each event that invokes a send/receive call, the send and receive time of each message, and the processes associated with these calls and messages. We call two matching enter and exit records an event and two matching send and receive records a message. In this paper, we focus on MPI, but aside from a few optional steps, our structure algorithm could be applied to any other message-passing model.

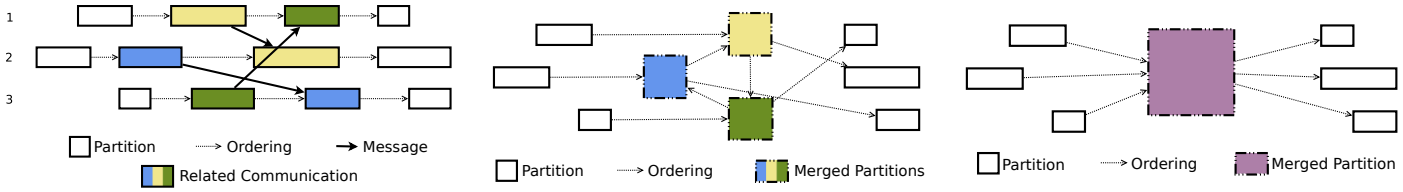
Parallel execution traces are used for performance analysis and debugging. Automated trace tools like Kojak [10] or its successor Scalasca [11] can detect given patterns of known performance problems, such as the late arrival of a message, and compute a severity score, which is mapped to source code. While this helps identifying the locations at which the bottleneck exists, it typically does not provide sufficient information about its context and root cause. In particular, while this can pinpoint very local performance problems, it cannot identify transitive dependence chains or relations among processes easily. Their addition of root cause analysis [12] allows the tracking of delay through dependencies, much like our differential lateness, however, their analysis is limited to local waiting state calculations instead of taking into account the context of peer events as we do when calculating lateness.

Morajko et al. [13] built per-process causality graphs for discovering structure and detecting propagation and root causes. They aggregated these graphs by identity, compressing performance information on the representative graph. This can unduly emphasize boundary behavior of a small number of processes while hiding the more extreme behavior of the larger clusters.

Traces are also used to determine the critical path through a program and analyze performance in that context [14], [15]. However, it can be unclear how the critical path interacts with the rest of the trace to cause issues. Further, critical paths can be lengthy, often require costly reverse playback, and can obscure subpaths of interest.

Manual trace analysis is often facilitated through timeline visualization: events are arranged in order of increasing time on the horizontal axis and in rank order by process id on the vertical axis. Fig. 1 (left) is a typical example taken from Vampir [2]. Other trace visualizers like Jumpshot [16] or Paraver [17] provide similar views. While such timelines allow the user to make inferences about timing directly from the spatial layout, excessive detail makes finding areas of interest difficult and clutters the visualization making interpretation arduous as dependencies are hard to follow. Our logical structure provides another way to visualize traces, the details of which can be found in [3]. In this paper we employ Vampir as one example of an established trace visualization tool using a conventional timeline view and Ravel to illustrate our logical structure and metrics.

Many tools [18], [19], [20] have focused on high-level, statistical detection of program behavior using clustering and wavelet techniques in traces. While these numerical techniques provide useful high-level structure, they do not help programmers understand local logical dependence chains in communication threads. Such algorithms could easily be combined



(a) Matching sends and receives indicate events are related and should be merged.

(b) Ordering relations for merged partitions are derived from the pre-merge neighbors.

(c) Strongly connected components are merged into single partitions.

Fig. 3: Mandatory Partitions

with our approach to interpret aggregate metrics in our logical structure.

In contrast to existing approaches, we transform the trace into its underlying logical structure of communication and perform analysis in the context of this structure. We derive metrics for events using the structure and the physical timing information from the trace. Fig. 1 (right) shows a visualization of our logical trace, where communication structure is uncovered and timing is incorporated by coloring via our lateness metric. This logical structure is obfuscated by performance problems in the figure on the left, so the relationship between the algorithm and the timing problems is not clear.

III. Extracting Logical Structure

We consider the logical structure of a program to be an ordering of events consistent with that program. Ideally, this structure reflects the developers' intended organization, but due to either programming errors or ambiguities in assigning logical time steps the structure may differ. In general, our goal is to determine which sets of events are intended to happen simultaneously for the purpose of comparing and analyzing their demonstrated versus ideal behavior.

We use the following relationships and definitions based on Lamport [21]. The happened-before relation ($\rightarrow$) is a partial order where (1) for events a, b of the same process, $a \rightarrow b$ if a occurs before b and (2) for matching send and receive events s, r, $s \rightarrow r$. Lamport calls events c, d *concurrent* if they are not ordered, i.e., $c \not\rightarrow d, d \not\rightarrow c$.

The Lamport clock is a function C mapping a number to each event such that for events a, b, $C(a) < C(b)$ if $a \rightarrow b$. This constraint is called the *clock condition*. Our assigned logical steps satisfy Lamport's clock condition, but we add further constraints. First, it is often intuitive to think of the communication of a program in terms of different *phases*, e.g., a neighborhood exchange or a global reduction (i.e., we assume a very fine grained definition of a phase). Assuming we have a set of phases, detected by our algorithm or specified by the user, we ensure they do not overlap. In terms of the clock condition this means that for phases $P \rightarrow Q$ with events p_i, q_i, respectively, $C(p_i) < C(q_j) \forall p_i \in P, q_j \in Q$. This condition ensures that the ordering in one phase is not affected by other phases. Second, rather than assuming that each event happens as soon as possible, we aim to discover the events that conceptually should happen simultaneously, e.g., all sends at one level of a binomial tree broadcast.

In particular, we focus on events representing messages since they impose happened-before relations between processes and thus contribute to a global happened-before structure built from the individual process timelines. All other (local per process) events are aggregated into a single event assumed to cover the entire time between messages, though more detailed assignments are possible. We create the logical structure in two steps: First, we infer the partitioning into phases (or sub-phases) and, second we assign logical time steps within and across partitions.

A. Partitioning

Organizing all communication events into phases not only matches the intuition of program developers, but also has a number of practical advantages. Most importantly, *partitioning* the trace into phases makes the computation and subsequent analysis a per-partition rather than a per-trace operation, which significantly simplifies and accelerates the process. We present graph-based algorithms to first identify inseparable groups of events and subsequently merge these to define phases. However, general phase detection is a difficult challenge [22], [23], [24], [25], [26], [27], especially since the "correct" partition is often subjective or not well-defined. Consequently, we optionally allow users to specify a partitioning directly to accommodate application specific details.

Mandatory Partitions. The algorithm starts by identifying *mandatory partitions* given by groups of events that due to ordering constrains or semantic reasons cannot be separated. Given a set of MPI events with their happened-before relations represented as a directed acyclic graph (DAG), we construct the partitioning in a bottom-up fashion that initially assigns each event its own partition (Fig. 3a). Semantically, each message (matching send and receive) should belong to a single partition and thus we merge their corresponding partitions (Fig. 3b). However, as shown in the figure, this can introduce cycles in the graph which prevents a linear ordering among the corresponding partitions. To restore a linear order we merge all partitions that form a strongly connected component, restoring the partition graph to a DAG (Fig. 3b). The resulting partitions are minimal groups of events that support a total order without separating messages. However, in practice, the resulting partitions are often very fine-grained as even simple operations such as `MPI_Allreduce` can be subdivided significantly. Since this typically does not match the intuition or intent of the developers we present additional techniques to further merge partitions if desired.

Waitall Partitions. One common construct in parallel applications is `MPI_Waitall`, which causes a process to wait until a given set of prior MPI calls have completed. We merge all partitions containing events associated with the

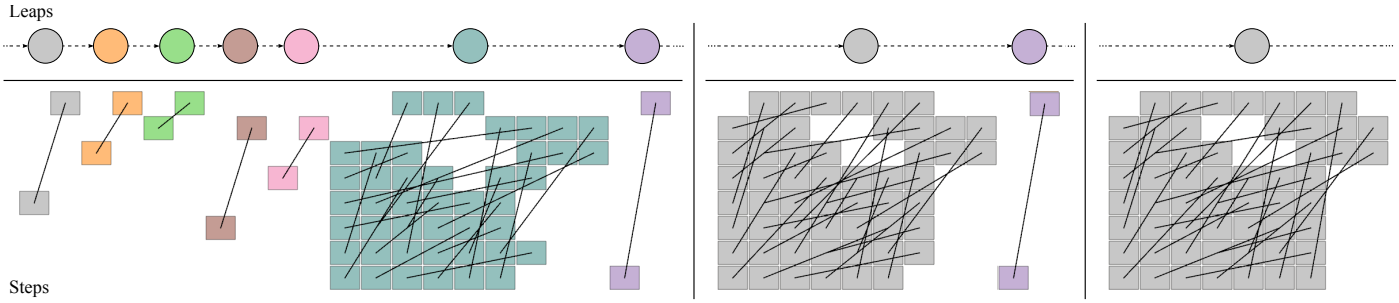


Fig. 4: Merging Leaps in LULESH. The graph in terms of leaps is shown on top. The bottom shows the individual processes as they would be stepped with that leap graph. The result of the strongly connected component merge is the left image. The gray leap merges in its succeeding leaps until it contains all processes, resulting in the center image. The remaining purple leap is significantly closer to the gray leap than the next one (not shown), so it is merged backwards, resulting in the right image.

same `MPI_Waitall` as an additional semantic constraint. However, the set of events handled by each `MPI_Waitall` is not directly available in our traces, so we determine the set heuristically and make this merging optional. We assume that all calls associated with the `MPI_Waitall` are not interspersed with receive events or collective, wait, or test calls and thus we assume that all send events between the last such call and an `MPI_Waitall` belong to that `MPI_Waitall`. The reason receive events are included in the former list is that in the trace any receive call associated with the `MPI_Waitall` ends during that `MPI_Waitall`.

```
complete_leaps (partitions);
    all_leaps = compute_leaps (partitions);
    k = 0;
    while k < |all_leaps| do
        leap = all_leaps[k];
        changed = TRUE;
        while changed and not complete (leap) do
            changed = FALSE;
            for p in partitions (leap) do
                a,b = compute_leap_distances (p);
                if b ≪ a then
                    merge_backward (p);
                    changed = TRUE;
                else
                    if will_expand (p) then
                        merge_at_leap (p);
                        changed = TRUE;
                    end
                end
            end
        end
        if not complete (leap) and force_merge then
            force_full_merge (leap);
        else
            k = k+1;
        end
    end
```

Algorithm 1: Complete leaps through merging partitions.

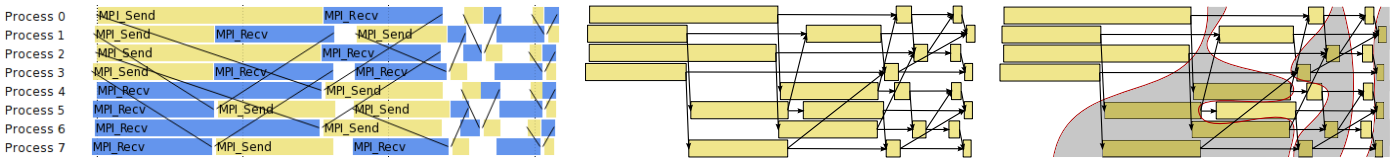
Leap Partitions. There may exist messages that do not result in a strongly connected component in the sense of Fig. 3, yet nevertheless logically belong together as part of the same phase. Fig. 4 shows an example taken from an eight process trace of LULESH [4]. Many of the communication events have been grouped, resulting in the large blue partition. However, a few messages on either side are isolated and thus are not included. In bulk synchronous codes, such as LULESH, we expect all processes to participate in each communication phase and thus we provide the option to merge partitions until this property is satisfied. More formally, we define a *leap* as all the partitions with the same graph distance from the source of the partition DAG.[1] We merge partitions until each leap contains events from all processes using Algorithm 1.

Starting from the first leap, we determine whether it is complete (contains events from all processes). Should this not be the case, we begin processing its member partitions. Each partition computes its *leap distance* as the minimum of the first event entry time for each of its processes and the event exit time of their previous event in the partition's previous-leap neighbors. By construction, any previous leap is complete and thus we prefer merging at the leap – pulling in unprocessed partitions from the next leap – to merging backwards – extending completed leaps. In practice, we typically only consider merging backwards if the incoming leap distance is more than an order of magnitude (factor of ten) smaller than the outgoing one. Once the direction of a potential merge has been established we always execute a backward merge, but only merge in from the next leap if this expands the set of processes participating in the merge. In this manner the current leap can shrink or grow and we repeat this process until the leap stabilizes. Depending on the application, the resulting stable leap may still not contain all processes. In this case we allow the user to either force all corresponding partitions to merge in all their successors before restarting the process or to accept the incomplete leap and continue.

Another option during the leap merge concerns the treatment of collectives. As some collectives act in a synchronizing manner, and thus our phase clock condition holds on either side of them, we also give the user the option to disallow any merge between partitions that fall on opposite sides of these collectives.

In the example of Fig. 4, the gray partition on the left successively merges in the succeeding partitions until it has merged in the blue one and thus contains all eight processes. Subsequently, the purple partition merges backward since in this case it is significantly closer to the incoming gray

[1] Intuitively, the leap is similar to *rank* in a graded poset, but we avoid that term due to confusion with MPI ranks.



(a) A Vampir visualization of a stencil in the pF3D communication benchmark.

(b) We create a happened-before graph containing only the send events.

(c) Stride boundaries are set based on graph distance.

(d) We use strides to position sends in logical time.

(e) Sends are positioned at the end of their stride. Receives are filled from the earliest step that meets happened-before constraints.

(f) We insert aggregated events (green) before each communication event. This spans all real time which is used for analysis in Section IV.

Fig. 5: Step Assignment. Sends are yellow. Receives are blue. Alternating white and gray backgrounds denote stride boundaries.

one than to the outgoing leap (not shown). The particular threshold to decide the merge direction and whether to force completed partitions should reflect the user's knowledge (or expectations) of the application in order to create the most intuitive partitioning.

While the algorithms described above may not accurately detect all phases, they are simple to implement, easy to adapt, and in our experience create intuitive partitions well-aligned with the developer's intention for all practical cases.

B. Local and Global Step Assignment

In Section III-A we created a DAG of partitions containing related communication events. Subsequently, we assign logical steps first within each partition locally and then globally across partitions, defining the logical structure. The local step assignment follows two simple principles: First, all happened-before relationships must be strictly maintained, i.e., $a \rightarrow b$ implies $step(a) < step(b)$; and second, send events have a greater impact than receives on the communication structure. The latter is a consequence of the fact that the order of receives is not always uniquely defined by the program and some events, such as `MPI_Waitall`, may serve as the receive for multiple sends. Consequently, we initially use only the sends to define the communication structure. Once the local (per-partition) order for all sends has been determined we introduce the receives and ultimately the compute events to the per-partition step assignment. We use an eight process run of the pF3D communication benchmark, shown in Fig. 5, as a working example throughout our explanation.

Send Strides. The send events in a trace typically define most of the communication structure and thus we start the local step assignment by grouping sends into *strides*. Strides are defined by the graph distances within the partition considering only the send events. More specifically, we create a sparse version of the happened-before graph of each partition that contains only the sends and their aggregated dependencies (Fig. 5b). We subsequently group sends according to their stride (Fig. 5c. We align the strides in logical time and assign preliminary steps accordingly (Fig. 5d). Not all processors contain a send in all strides. Next, we re-introduce the receives such that the ordering is preserved, all sends within a stride are assigned the same step, and receives are placed as early as possible while still maintaining their happened-before relationships (Fig. 5e). Finally, we insert aggregated *computation* events representing all processing between communication events (Fig. 5f). Defining these aggregated events allows us to account for all physical time spanned by the trace, which is helpful when defining the temporal metrics in Section IV. To assign global steps, we shift the local step assignment within each partition to be after all the steps occupied by its predecessors in the partition DAG, thereby enforcing global happened-before relationships.

IV. Temporal Metrics

By design, we avoid relying on wall-clock timing information when determining the logical structure of a trace. This produces a more abstract and easier to process representation of a trace, in addition to avoiding problems with clock skew and synchronization. Ultimately, however, the timing of events determines where delays or bottlenecks occur and which part of the program is responsible. We therefore restore the temporal information by computing various temporal metrics and map these metrics onto the logical structure as attributes for easy interpretation.

Lateness. Simple examples of such metrics are event-based indicators, such as entry or exit time or duration, all of which can be computed directly and without the logical structure. However, the true power of our technique comes from comparing such simple event-centric metrics across logical partitions. For example, comparing the exit times of events in the same partition allows one to track delays. In particular, we define the *lateness* of an event as the difference in exit time between itself and the earliest event sharing the step in the partition:

$$l_e = e.exit - \min\{x.exit | e, x \in P, x.step = e.step\}$$

where P is the set of events within a partition. By restricting the computation to partitions we take advantage of the leap merging. In bulk-synchronous codes leaps typically contain events from all processes and thus lateness is calculated

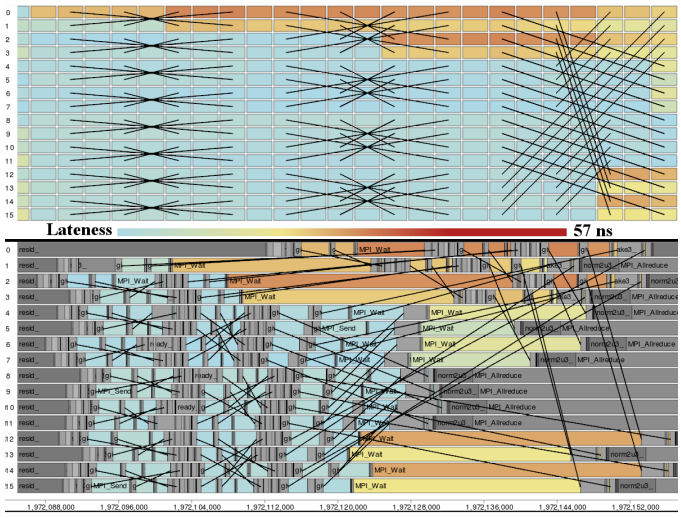
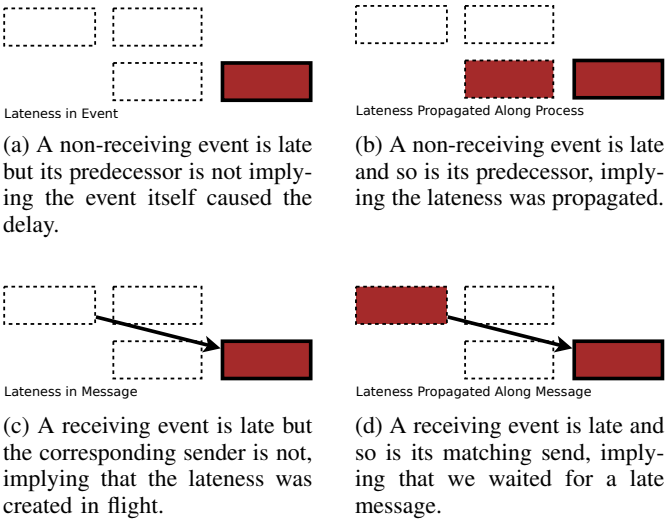


Fig. 6: Logical (top) and physical (bottom) time visualization of a 16 process execution of MG. Communication events are colored using the lateness metric. The first process becomes late during an aggregated non-message event. The lateness spreads through messages to the other processes.



(a) A non-receiving event is late but its predecessor is not implying the event itself caused the delay.

(b) A non-receiving event is late and so is its predecessor, implying the lateness was propagated.

(c) A receiving event is late but the corresponding sender is not, implying that the lateness was created in flight.

(d) A receiving event is late and so is its matching send, implying that we waited for a late message.

Fig. 7: Creation and propagation of lateness for non-receiving (a,b) and receiving (c,d) events.

globally. This can also be enforced by a post-stepping merge across shared global steps. However, for codes with different process-groups that perform separate and distinct actions, the partition ensures that only related events are compared. For more complex arrangements the user can also manually specify groups of partitions to consider when computing metrics.

Fig. 6 shows a portion of a 16 process MG trace visualized in Ravel with communication events colored by the lateness metric. Ravel displays both a traditional physical timeline and a logical timeline. Both views show a delay in a non-communication event on the first process propagates to other processes. The logical time view highlights a propagation of lateness along processes and along messages to other processes. This leads us to classify the conditions that contribute to the lateness of an event depending on whether the event in question is receiving a message or not. A late, non-receiving event whose predecessor is not late is likely responsible for the delay, perhaps due to load imbalance in the computation (Fig. 7a). If the predecessor is late as well (Fig. 7b), lateness has been propagated and was likely caused upstream. Similarly, a late receiving event whose corresponding sender is not late (Fig. 7c) indicates that the message has either been delayed in flight, e.g., due to contention in the network, or is late because of the processing needed to perform a receive, which could be caused by e.g., a slow buffer allocation. Finally, a late receive with a matching late send indicates lateness propagation across processes (Fig. 7d). The aggregated non-message events created in global step assignment are necessary to differentiate between in-process and across-process lateness. One interesting and useful property of lateness is that it naturally "resets" once all processes become equally late. For example, a reduction down to a single process resets the lateness, as would a barrier or simply a load imbalance that, through neighbor exchanges, has propagated globally.

Differential Lateness. Lateness provides a good high level overview of potential root causes of delays. Especially when coupled with the visualizations in Ravel, a user can quickly find the first late event and continue a more detailed analysis from there. However, in very large or complex traces, identifying these patterns becomes challenging, meaning techniques are necessary to directly identify the likely cause of a problem. To this end we propose to analyze *differential lateness*: the difference between the lateness attributed to prior events and the lateness at exit time:

$$d_e = \max\{l_e - \max\{l_x | x \to e, \nexists y \text{ s.t. } x \to y \wedge y \to e\}, 0\}$$

Instead of showing all events that are late, differential lateness highlights the events and processes that cause the lateness. Note, though, that we do not allow negative lateness: while it sometimes can highlight events that compensate for earlier problems, negative lateness primarily occurs at reset boundaries leading to somewhat confusing and difficult to interpret configurations.

Clustering. In addition to intuitive metrics, the logical structure also provides new opportunities to compare per-process traces and to use this information to cluster processes into classes of similar behavior or to summarize that behavior to ease analysis. This feature has been used in Ravel to aid visual exploration at higher process counts. It is difficult to cluster processes in physical time as there exists no direct correspondence between traces from different processes. Instead, logical time provides this correspondence and provides a simple way to define various similarity metrics.

For example, one can compare per-process traces using their delay profile. To this end, we calculate the distance between two processes by the average of their squared difference in lateness at each logical step. At steps where an event is present in one process, but not the other, we substitute the last known lateness value if it exists. While the process is absent in that step, we assume that throughout a process, lateness remains the same unless some event occurs to change it. Steps where neither process is active are skipped. We average the squared differences at each step as some processes may

have more matching steps than others. In the case where no steps yield a difference value, we assume the processes have maximum distance. Fig. 8 provides an example calculation. This metric is based on temporal behavior, not on the specific calls made by the process. This means processes performing different actions could be clustered together if they are similarly delayed, regardless of whether they match function calls exactly.

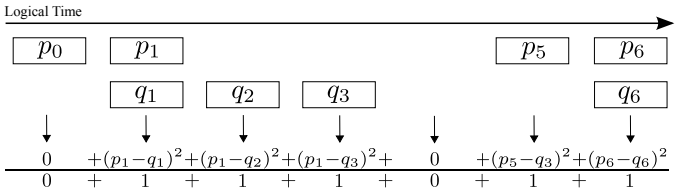


Fig. 8: Example calculation of distance between processes.

This metric may be used over all logical steps in the trace. However, in some cases it may be more beneficial to calculate process distances in each partition in the logical structure individually. Partitions are meant to be communication phases and phases may exhibit different behavior. Thus, when examining a subset of the trace, it may make more sense to examine this metric at the partition level. Depending on the partitioning options used, there may be several partitions at each logical step. The distance metric can be applied to each of those partitions separately or across all the partitions active over the same set of steps.

V. Results

We execute applications on two radically different architectures: a large Blue Gene/Q installation as well as an Infiniband cluster with 12 Sandy Bridge cores per node split into two sockets. The former system uses IBM's compute node kernel stack and MPI implementation, while the latter system uses a Red Hat derived Linux distribution combined with MVAPICH. On both machines we obtain our traces using VampirTrace [28] and store them in the Open Trace Format (OTF) [29].

A. MPI Collective Operations

Collective algorithms are an important part of MPI because they allow groups of processes to work together for efficient global communication. For example, MPI provides an `MPI_Allreduce` algorithm that performs a distributed parallel sum (or other associative operation) and puts its result on all processes. Collectives have dense communication patterns, and they pose a challenge for existing trace tools, especially when the system is noisy and dependence chains are perturbed across MPI processes. We use these as a case study of our tool to demonstrate its ability to correctly determine logical structure and to showcase how we can display collective operations in an intelligible manner. For our experiments, we used libNBC [1], an open source implementation of non-blocking MPI collective operations. We chose libNBC because the algorithms it implements for its collectives are well-understood, allowing us to verify our logical structure.

We first consider the binomial tree implementation of `MPI_Allreduce`. Fig. 9 shows the unprocessed trace as visualized by Vampir and its extracted logical structure as

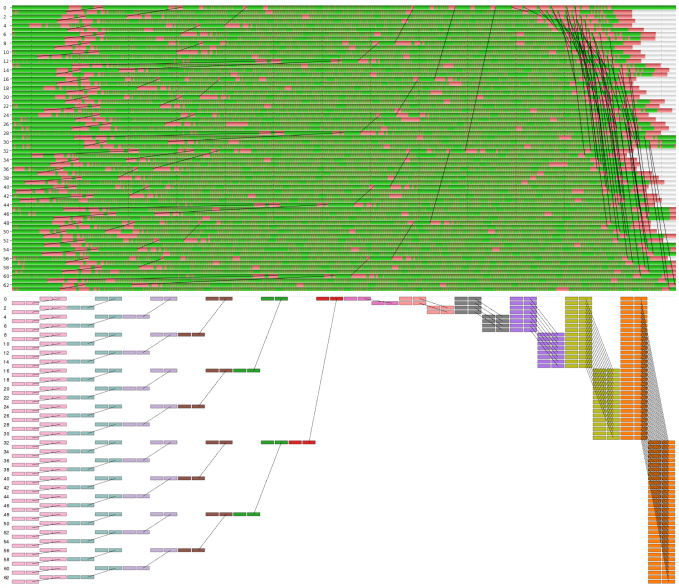


Fig. 9: Visualization of 64 process binomial tree `MPI_Allreduce` in physical time by Vampir (top) and logical time by Ravel (bottom). In logical time, we color by communication partition. We are able to identify the binomial tree levels though they overlap in physical time.

visualized by Ravel. The algorithm performs a parallel reduction with a binomial tree embedded in the MPI ranks, then it broadcasts the global sum back along another binomial tree. Our logical structure captures the send-receive pairs at each level of the tree, despite the overlap observed in the physical time visualization. All of our partitioning options yield the same logical steps but different partitions. In the figure we show the partitioning resulting from mandatory merging and merging across global steps.

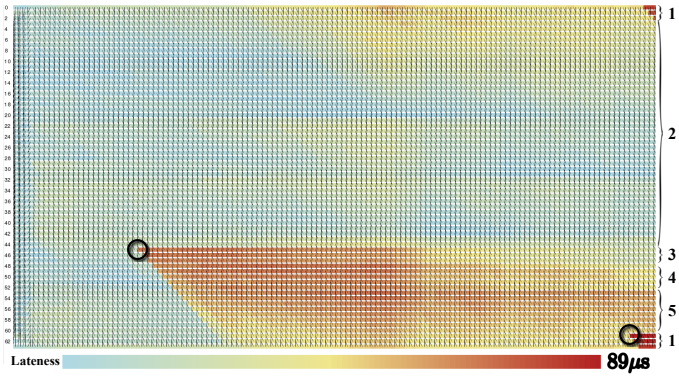


Fig. 10: Ring algorithm `MPI_Allreduce` on 64 processes. Coloring is done by lateness, showing propagation. We find two events with high differential lateness (circled). A clustering of five groups is labeled on the right.

Fig. 10 shows the logical time view of a ring implementation of `MPI_Allreduce`, colored by lateness. In this linear-time algorithm, each of the P processors sends to its neighbor and accumulates a sum from each rank's contribution. Again our logical structure accurately determines the P rounds of this communication. We also observe the spread of lateness

from the 45th process and its continued effects through the remainder of the rounds. Also visible are the handful of late processes in the final rounds. The circled steps were the only ones calculated to have high differential lateness, indicating these were the sources of the other delays. Closer examination reveals lateness is injected at a messaging event, possibly due to congestion on the network. We hierarchically cluster the processes and explore the result in Ravel. The clustering for five groups is shown in the figure denoted by brackets on the right side. The first cluster captures the processes at both ends of the rank space that become very late in the final rounds. The largest cluster represents the processes largely unaffected by the two delay sources.

B. Algebraic Multigrid

Algebraic multigrid techniques solve sparse linear systems that may or may not be associated with an actual spatial grid. The method begins with a fine-grained grid or matrix that is successively coarsened until it can be solved with reasonable error. It interpolates back from the coarsened solution to the fine-grained one. This so-called V-cycle is repeated until it converges upon a solution. We examine an algebraic multigrid method implemented in the *hypre* scalable solver library [8], via the AMG2013 benchmark, which is part of the CORAL [30] benchmark suite. This gives us an opportunity to verify our structure algorithm on a more complicated example.

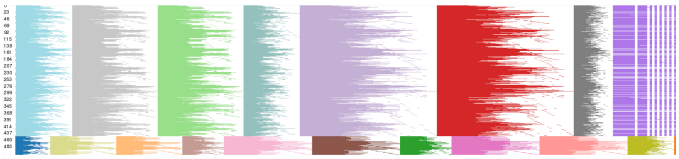


Fig. 11: Partitioning of AMG2013 solve run on 512 processes. The last 64 processes partition independently of the others.

Fig. 11 shows a portion of the logical structure we extracted, using the leap merge option, from a 512 process trace of the AMG2013 solver algorithm. The events are colored by partition. We omit the message lines to focus on the partitions. Our structure separates the first 448 processes into distinct partitions from the remaining 64. This led us to examine the rest of the structure and discover that the two process groups never interact. Upon consulting with the development team and verifying the results, we learned that the final 64 processes are assigned to the anisotropic portion of the domain, which is why they behave differently and independently, as found by our logical structure.

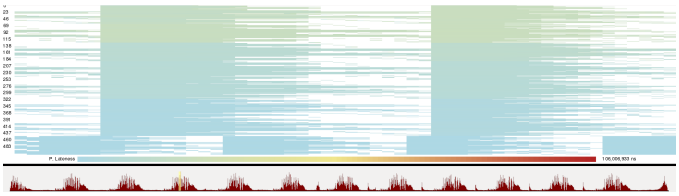


Fig. 12: Logical structure of AMG2013 solve on 512 processes. The overview histogram on the bottom shows twelve repetitions of the lateness profile, corresponding to the iterations of the V-cycle.

Fig. 13: Partitioning of a single iteration of the AMG2013 solve, focusing on the first 448 processes. The individual partitions match well with what we expect from the levels of the V-cycle. We see the amount of communication increase as processes in coarser levels gain more neighbors. Eventually some processes become inactive due to coarsening, resulting in white lines across the long blue partitions.

Examining lateness in the trace, shown in Fig. 12, the overview histogram of lateness at the bottom of the figure revealed a recurring pattern over logical time. Differential lateness (not shown) spikes at the beginning of the late step regions. All of the events with high differential lateness are computation rather than communication, probably due to the imbalance caused by coarsening. The number of repetitions correspond to the iterations of the V-cycle reported by the run.

Having noticed iteration boundaries, we narrow our focus to a single iteration, shown in Fig. 13, once again colored by partition. For simplicity we show only the 448 process partition. In the solve, each level performs a relaxation step and one to two matrix-vector products. Our algorithm separates these in partitioning except where there are processes without work, at which point the aggressive merging combines a relaxation step with a matrix-vector product. Initially, the partitions are short, but as the grid gets coarser, participating processes need to send information to more neighbors, resulting in longer partitions. At the same time, the coarsening leaves some processes without work, seen as white lines (no events) in the wide blue partitions. This behavior is expected, suggesting the logical structure extracted by our partitioning approach is consistent with AMG's algorithm design.

C. Massively Parallel Merge Trees

Merge trees are topological structures that can aid in the analysis of large scale simulations [31], [32], [9]. We analyze a massively parallel algorithm to compute them *in situ*, which avoids the limitations and penalties of writing out the data to be analyzed post-mortem.

The algorithm begins with each process handling its local portion of the data. As the computation portion is data-dependent and the data decomposition follows that of the simulation it runs with, load imbalance can be an issue. After the local computation step, each process sends its partial results to a designated gather process. The gather processes combine the results from their children and send them along to both their gather process at the next level of the tree and back down toward the leaves of the gather tree. This continues until the root of the gather tree has integrated the results and sent them back to the leaves. At each gather level, the messages to the

leaves are sent along the tree structure to divide the messaging burden amongst the processes.

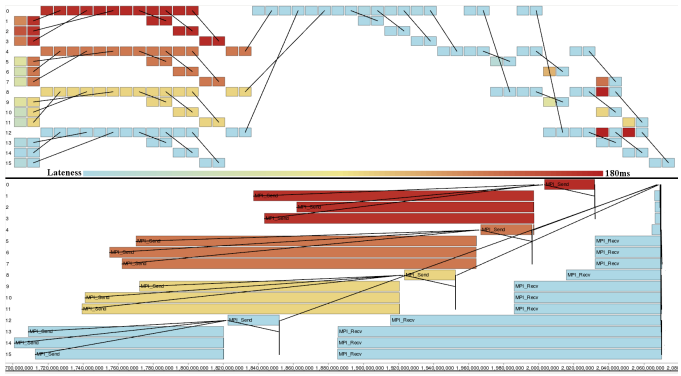


Fig. 14: Complete trace of a 16 process, 4-ary merge tree in the default layout, i.e., with the root at process 0, shown in logical time (top) and physical time (bottom) and colored by lateness.

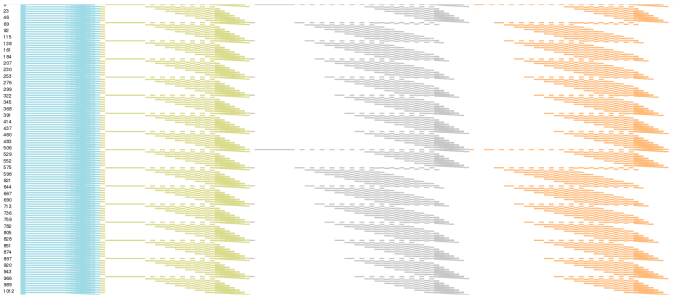
The leaves of the gather tree comprise all processes generating data from the simulation, which is usually all processes running the application. As such, some processes have to take on extra roles as the internal vertices of the gather tree. These roles are assigned modulo-k when the gather is done via a k-ary tree. For example, in a binary tree, the leaves first send to gather processes $0, 2, 4 \ldots$ and those processes in turn send to the next level gather processes of $0, 4, 8, \ldots$, and so forth.

Fig. 14 shows the logical and physical time visualizations of a complete 16 process, 4-ary merge tree, both colored by lateness. As expected, the lateness comes from data-dependent load imbalance in the initial computation. Our structure extraction successfully retrieves the communication pattern, showing successive groups of four processes sending up the tree, as well as the updates sent back down. However, this also reveals a potential inefficiency in the implementation. Once all the information has been gathered at the root, the update is sent to the root's leaf (initial gather) processes rather than its higher-level gather processes (4, 8, 12).

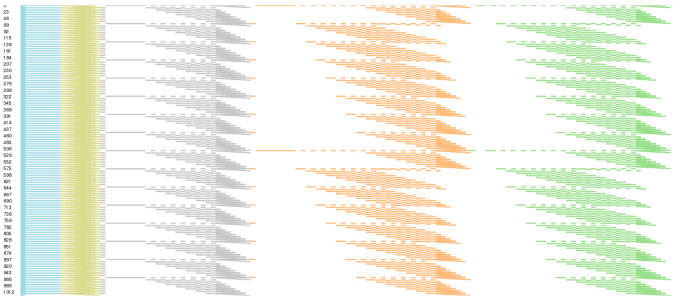
Further examination with 8-ary gathers at 1,024 (Fig. 15a) and 16,384 (Fig. 15b) processes, respectively, indicates this problem occurs at every level of the gather. This manifests as a repeating skewed parallelogram motif, the panhandle showing the sends to the leaf processes nearby in rank space first. Furthermore, at many steps very few processes are active, suggesting that not only are gather processes prioritizing updates to their leaves over their higher level gather children, they are also prioritizing all of the downward updates over sending information upwards to be combined.

After these artifacts were brought to the attention of the developers, they incorporated an improved communication ordering into the next iteration of their algorithm. The resulting traces are shown in Fig. 15c and Fig. 15d, respectively. The panhandle is no longer present and in most steps more of the processes are participating.

The 1,024 process traces exhibit more lateness than the 16,384 process ones. This is because they were executed on the same simulation data. As the local data in the 16,384 process run is smaller in size, there is less variability in the initial computation, resulting in less lateness.



(a) Logical steps resulting from developer partitions.



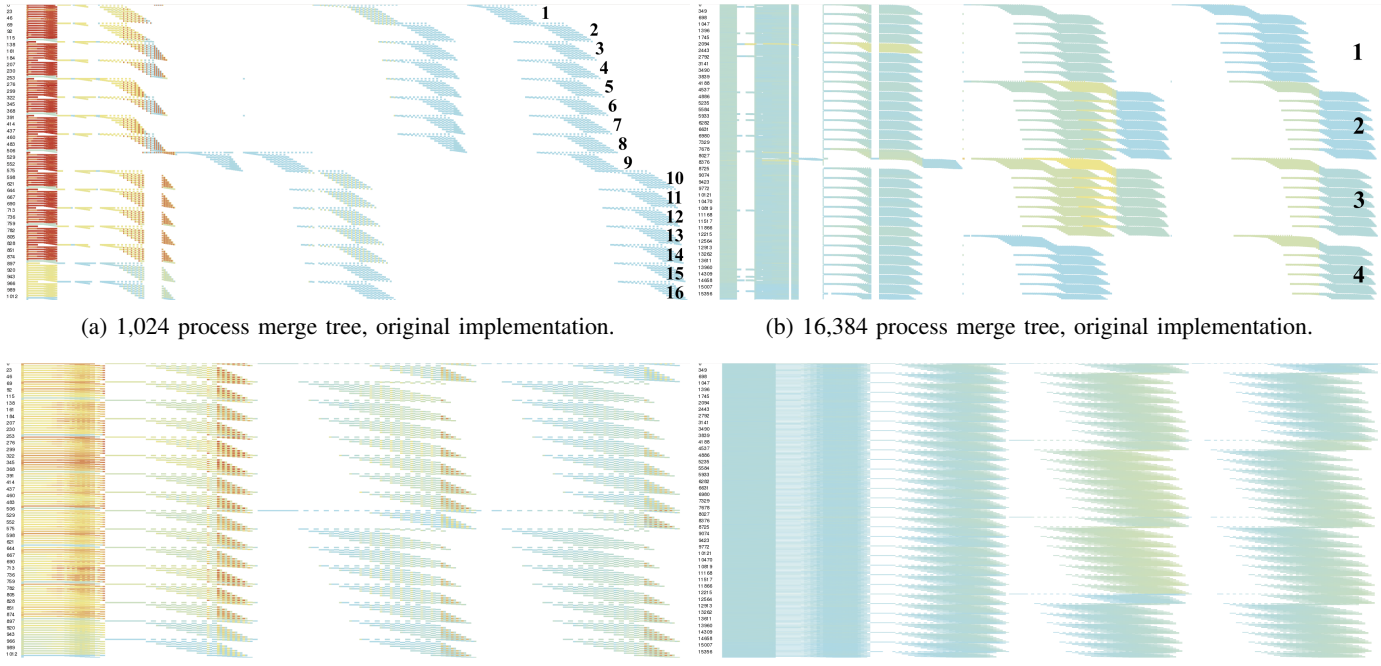
(b) Logical steps resulting from our partitions.

Fig. 16: 1,024 process merge tree, improved implementation, colored by partitioning. In (a) we use partitions from the merge tree developers. In (b) we derive partitions using our algorithm. The partitioning and resulting logical structure is highly similar.

Our logical structure finds the gather tree in both the original and improved implementations. In a 1,024 process 8-ary gather tree, the root has only two children as 1,024 is not a power of eight, and thus the next level has 16 children total. In the later rounds of the gather tree, we see 16 parallelogram groupings, as marked in Fig. 15a. Similarly, in a 16,384 process 8-ary gather tree, the root has only four children. These four major groups are also apparent in our logical structure, marked in Fig. 15b.

We obtained communication phase information from the developers and compared it to our partitioning with the leap merge and merging across global steps. Fig. 16 shows the main difference is that our algorithm breaks the initial phase into the up and down partitions, but even with this difference, the resulting logical structures are highly similar.

Closer examination of the original merge tree implementation reveals some of the shortcomings of a strict adherence to the Lamport happened-before ordering. Fig. 17 shows a zoomed-in view of the tree from Fig. 15a with a single communication line drawn to highlight the problem. In the first step, the (level-0) leaves send to their level-1 gather process which responds with corrections and subsequently sends the result to the level-2 process. However, the blue branch completes its level-1 gather and sends to its level-2 result before the topmost level-1 gather has received all messages from its children. Due to the over-provisioning, the level-2 gather as well as the topmost level-1 gather are handled by the same process. As a



(a) 1,024 process merge tree, original implementation.

(b) 16,384 process merge tree, original implementation.

(c) 1,024 process merge tree, improved implementation.

(d) 16,384 process merge tree, improved implementation.

Fig. 15: Logical structure of an 8-ary merge tree, original and improved implementations, with 1,024 and 16,384 processes. The skewed parallelograms are cascading updates towards the leaves. The original implementation, (a) and (b), has fewer processes active at each step than the improved implementation, (c) and (d). In (a) and (b), we mark high-level subtrees in the gather.

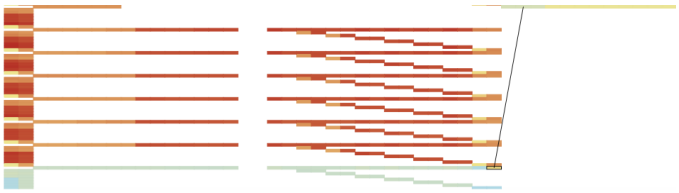


Fig. 17: A small portion of Fig. 15a highlighting a potential problem with strictly adhering to the happened-before relations. The top gather process receives a premature message from the gather process of a fast group on the bottom. To enforce the true ordering of events, all remaining sends must be shifted toward the right, preventing logically parallel communication from being assigned the same step.

result the early receive causes a severe misalignment of the steps with all remaining communication events of the top-most process being shifted towards the right. This is a direct consequence of the happened-before ordering and thus cannot be avoided. Nevertheless, in this case the early message does not actually change the order of computation. Thus, potentially allowing the stepping algorithm to violate the happened-before relation to create the expected regular patterns would likely result in a more intuitive visualization and more meaningful metrics. However, it is not clear under which circumstances such a re-ordering should be permissible and this will be the subject of future research.

VI. Conclusion

We have presented a new approach for analyzing execution traces obtained from parallel programs. We extract a logical structure, meant to capture the intended ordering of event in a program. This technique utilizes happened-before relationships not only on the individual event scale, but also on the scale of communication phases and even concurrent sends. Since the logical structure, on purpose, hides timing information, we explicitly define temporal metrics and map them onto the logical structure. In particular, we exploit the logical structure by capturing delay experienced by events relative to their peers, providing an abstract view of lateness. Using the happens-before relationship encoded in the logical structure, we use this information to both pinpoint the original cause of a bottleneck and to study its propagation.

Through a series of case studies, we have demonstrated the fidelity of our algorithm in identifying structures across a variety of communication profiles. We have also shown the correctness of our metrics in locating both load imbalance (merge tree) and communication (libNBC) delays. We intend to expand the types of events and dependencies handled by our structure extraction algorithm and further leverage the structure for detection of performance issues.

Acknowledgment

The authors would like to thank Ulrike Yang and Aaditya Landge for their guidance regarding AMG2013 and the parallel merge tree application respectively.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-TR-656141.

REFERENCES

[1] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI," in *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[2] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996.

[3] 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," Under Review.

[4] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[5] "NAS parallel benchmarks (NPB)." [Online]. Available: https://www.nas.nasa.gov/publications/npb.html

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, and R. A. Fatoohi, "The NAS parallel benchmarks," *The International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[7] C. H. Still, R. L. Berger, A. B. Langdon, D. E. Hinkel, L. J. Suter, and E. A. Williams, "Filamentation and forward brillouin scatter of entire smoothed and aberrated laser beams," *Physics of Plasmas*, vol. 7, no. 5, pp. 2023–2032, 2000.

[8] R. Falgout, J. Jones, and U. Yang, "The design and implementation of hypre, a library of parallel high performance preconditioners," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, A. Bruaset and A. Tveito, Eds. Springer-Verlag, 2006, vol. 51, pp. 267–294.

[9] J. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *Proc. ACM/IEEE Conference on Supercomputing (SC12)*, 2012.

[10] B. Mohr and F. Wolf, "KOJAK: A tool set for automatic performance analysis of parallel programs," in *9th International Euro-Par Conference (EUROPAR)*, Klagenfurt, Austria, Aug. 2003.

[11] F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi, "Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications," in *Tools for High Performance Computing*. Springer Berlin Heidelberg, 2008, pp. 157–167.

[12] D. Böhme, M. Geimer, F. Wolf, and L. Arnold, "Identifying the root causes of wait states in large-scale parallel applications," in *Proc. of the 39th International Conference on Parallel Processing (ICPP), San Diego, CA, USA*. IEEE Computer Society, Sep. 2010, pp. 90–100.

[13] O. Morajko, A. Morajko, T. Margalef, and E. Luque, "On-line performance modeling for mpi applications," in *Euro-Par 2008 Parallel Processing*, ser. Lecture Notes in Computer Science, E. Luque, T. Margalef, and D. Bentez, Eds. Springer Berlin Heidelberg, 2008, vol. 5168, pp. 68–77.

[14] M. Schulz, "Extracting critical path graphs from mpi applications," in *Cluster Computing*. IEEE International, September 2005, pp. 1–10.

[15] D. Boehme, F. Wolf, B. R. de Supinski, M. Schulz, and M. Geimer, "Scalable critical-path based performance analysis," *Parallel and Distributed Processing Symposium*, pp. 1330 – 1340, 2012.

[16] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *High Performance Computing Applications*, vol. 13, no. 2, pp. 277–288, Fall 1999.

[17] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," 1995.

[18] J. Gonzalez, J. Gimenez, and J. Labarta, "Automatic detection of parallel applications computation phases," in *Proc. of the 23rd IEEE Intl. Parallel and Distributed Processing Symp.*, 2009, pp. 1–11.

[19] T. Gamblin, R. Fowler, and D. A. Reed, "Scalable methods for monitoring and detecting behavioral equivalence classes in scientific codes," in *Proc. of the 22nd IEEE Intl. Parallel and Distributed Processing Symp.*, 2008, pp. 1–12.

[20] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed, "Clustering performance data efficiently at massive scales," in *Proc. of the 24th ACM Intl. Conf. on Supercomputing*. New York, NY, USA: ACM, 2010, pp. 243–252.

[21] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[22] M. Casas, R. M. Badia, and J. Labarta, "Automatic phase detection of MPI applications," *Parallel Computing: Architectures, Algorithms, and Applications*, vol. 38, pp. 129–136, 2007.

[23] ——, "Automatic structure extraction from MPI applications tracefiles," in *13th International Euro-Par Conference*, vol. 4641/2007, Rennes, France, August 28-31 2007, pp. 3–12.

[24] J. Gonzalez, J. Gimenez, and J. Labarta, "Automatic detection of parallel applications computation phases," in *International Parallel and Distributed Processing Symposium (IPDPS'09)*, Rome, Italy, May 25-29 2009.

[25] G. Llort, H. Servat, J. Gonzalez, J. Gimenez, and J. Labarta, "On the usefulness of object tracking techniques in performance analysis," in *Supercomputing 2013 (SC'13)*, Denver, CO, November 17-22 2013.

[26] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, October 5-9 2002, pp. 45–47.

[27] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, November-December 2003.

[28] TU Dresden Center for Information Services and High Performance Computing (ZIH), "VampirTrace 5.14.2 user manual," `http://www.tu-dresden.de/zih/vampirtrace`, March 2013.

[29] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the open trace format (OTF)," in *Proc. of 6th Int. Conf. on Comp. Sci.*, ser. ICCS'06. Springer-Verlag, 2006, pp. 526–533.

[30] "Collaboration of Oak Ridge, Argonne, and Livermore benchmark codes," https://asc.llnl.gov/CORAL-benchmarks.

[31] P.-T. Bremer, G. Weber, J. Tierny, V. Pascucci, M. Day, and J. B. Bell, "Interactive exploration and analysis of large scale simulations using topology-based data segmentation," *IEEE Trans. on Visualization and Computer Graphics*, vol. 17, no. 9, pp. 1307–1324, 2011.

[32] J. Bennett, V. Krishnamurthy, S. Liu, V. Pascucci, R. Grout, J. Chen, and P.-T. Bremer, "Feature-based statistical analysis of combustion simulation data," *IEEE Trans. Vis. Comp. Graph.*, vol. 17, no. 12, pp. 1822–1831, 2011.