Ordering Traces Logically to Identify Lateness in Message Passing Programs

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

Abstract—Event traces are valuable for understanding the behavior of parallel programs. However, automatically analyzing a large parallel trace is difficult, especially without a specific objective. We aid this endeavor by extracting a trace's *logical structure*, an ordering of trace events derived from happened-before relationships, while taking into account developer intent. Using this structure, we can calculate an operation's delay relative to its peers on other processes. The logical structure also serves as a platform for comparing and clustering processes as well as highlighting communication patterns in a trace visualization. We present an algorithm for determining this idealized logical structure from traces of message passing programs, and we develop metrics to quantify delays and differences among processes. We implement our techniques in Ravel, a parallel trace visualization tool that displays both logical and physical timelines. Rather than showing the duration of each operation, we display where delays begin and end, and how they propagate. We apply our approach to the traces of several message passing applications, demonstrating the accuracy of our extracted structure and its utility in analyzing these codes.

Index Terms—trace analysis, performance

1 Introduction

Writing an efficient, scalable parallel program that performs well on different architectures is challenging. Achieving good performance on each new machine involves noticing performance problems, identifying their causes, reworking the implementation, and iterating tediously several times. Tracing tools are commonly used to help with this process. They capture communication events and display them in timeline views (as in Vampir [1]). However, two factors limit their effective use at scale. First, traces from large numbers of processes and long-running jobs are prohibitively large. Second, the complex communication patterns common in many large-scale codes are hard to comprehend when plotted with respect to wall-clock time. We need new analysis and visualization tools to help users understand complex parallel execution traces and to aid in determining what is necessary to optimize parallel code.

In this paper, we address the difficulty of trace analysis by focusing on the logical communication structure of message passing programs. We base this structure on happened-before relationships of traced events across parallel processes. This structure differs from previous analyses in logical time by considering the developers' intended organization of concurrent events. This allows us to deduce happened-before relationships at several granularities. Our structure allows us to define metrics in terms of relationships among logically simultaneous operations, extracting only the performance critical timing information. We define *lateness*, which measures an operation's delay relative to its peers, and *differential lateness*, which measures when delay is injected into the trace. This allows analysts to quickly identify delayed processes and the bottlenecks they cause.

Our approach leads to cleaner visualizations because logical structure aligns communication operations across processes, directly exposing communication patterns. Logical structure also provides a basis for comparing and clustering processes. Ravel [2], our interactive trace visualization tool, displays traces in both wall-clock and logical time. It uses clustering to improve visualization scalability and to help users focus on the most important parts of the trace.

We present our algorithm using traces of the LULESH shock hydrodynamics proxy application [3]; a communication proxy application for the pF3D laser-plasma interaction code [4]; the SMG2000 semicoarsening multigrid benchmark [5]; and the NAS MG benchmark [6], [7]. We demonstrate the effectiveness of our algorithm and our metrics through three case studies: implementations of collective operations in MPI [8], an *in situ* merge tree application [9], and the AMG2013 sparse linear solver [10], [11]. We conduct our studies on an IBM Blue Gene/Q system and on an Infiniband cluster with Intel Sandy Bridge processors.

The major contributions of this work are:

1) A set of techniques to extract the logical communication structure of a message passing program from its execution trace;
2) Novel metrics, such as lateness, that help identify performance bottlenecks or delays in the execution;
3) Case studies demonstrating that the new approach accurately detects and highlights performance characteristics difficult to obtain from existing techniques.

• *K. E. Isaacs and B. Hamann are with the Department of Computer Science, University of California, Davis, CA 95616.*
E-mail: {keisaacs, bhamann}@ucdavis.edu
• *T. Gamblin, A. Bhatele, M. Schulz and P.-T. Bremer are with the Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551.*
E-mail: {tgamblin, bhatele, schulzm, ptbremer}@llnl.gov

Manuscript received ; revised .

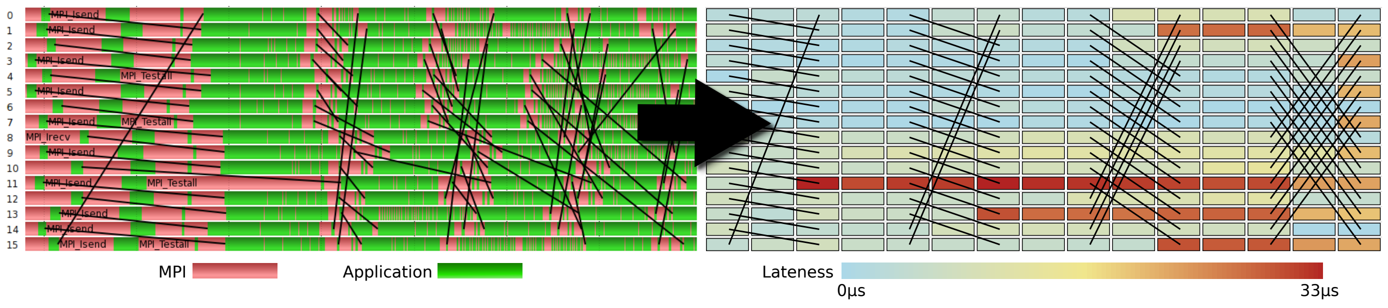


Fig. 1: Trace of a 16 process `MPI_Alltoall` using the dissemination implementation from libNBC [8]. From the raw physical time data, shown on the left using Vampir [1], we deduce a logical structure, visualized on the right, and use this structure to compute a novel lateness metric of each operation, 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.

2 Related Work

Parallel execution traces are used both for performance analysis and for debugging. Automated trace tools like Kojak [12] or its successor Scalasca [13] can detect 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 in identifying the locations at which the bottleneck exists, it typically does not make the context or root cause readily available. In particular, while these tools can pinpoint very local performance problems, they cannot identify transitive dependence chains or relations among processes easily. The addition of root cause analysis [14] allows the tracking of delay through dependencies, much like our differential lateness. However, the analysis remains limited to local waiting state calculations instead of taking into account the context of peer operations as done when calculating lateness.

Morajko et al. [15] build per-process causality graphs to discover structure and to detect root causes. They aggregate these graphs by identity and compress performance data on the representative graph. This can unduly emphasize boundary behavior of a small number of processes while hiding more extreme behavior of larger clusters.

Another way of analyzing trace data is to determine the critical path and analyze performance in that context [16], [17]. However, it is often unclear how operations on the critical path interact with the rest of the trace. Moreover, critical paths can be lengthy, often require costly reverse playback, and can obscure interesting subpaths.

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 [1]. Other trace visualizers like Jumpshot [18] or Paraver [19] 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. It also clutters the visualization and makes interpretation arduous, as dependencies are hard to follow. Logical time trace visualizations have been implemented, mostly for debugging [20], [21], but these incorporate few dependencies and thus place concurrent events as early as possible, and do not incorporate physical time information directly within the view. Our logical structure and metrics provide another way to visualize traces, the details of which can be found in [2]. In this paper, we employ Vampir as one example of an established trace visualization tool using a conventional timeline view. We employ Ravel to illustrate our logical structure and metrics.

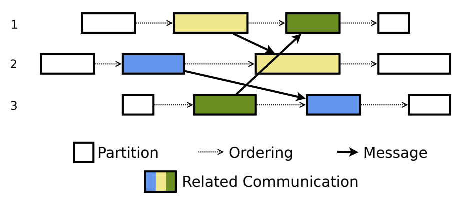
Many tools [22], [23], [24], [25] have focused on detecting high-level, statistical program behavior using clustering and wavelet techniques. While these numerical techniques provide useful high-level structure, they do not help programmers understand local logical dependence chains in communication threads. Such algorithms could easily be combined with our approach to display these aggregate metrics within our logical structure.

3 Extracting Logical Structure

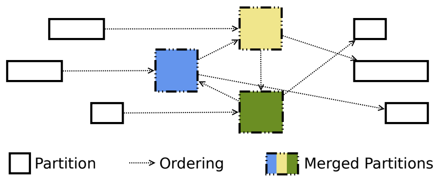
In contrast to existing approaches, we transform an event trace into a logical communication structure, and we perform analysis in this context. Throughout, we assume that a parallel trace for a message passing program consists of measured instantaneous events that are either function entry or exit, or communication such as sends, receives, or collectives. We call two matching enter and exit records an operation and two matching send and receive records a message. We further require, at a minimum, that an execution trace is a series of records of the enter and exit time of each operation that invokes communication, the send and receive time of each point-to-point message, and the processes associated with these operations and messages. In this paper, we focus on MPI, but aside from a few rules for specific MPI operations, our structure algorithm could be applied to any other message passing model.

The *logical structure* of a program is an ordering of operations consistent with that program, ideally reflecting the developers' *intended* organization. Due to either programming errors or ambiguities in assigning logical time steps, the structure may differ, but in general our goal is to determine which sets of operations are intended to happen simultaneously. This allows us to compare and analyze demonstrated versus ideal behavior of operations. While a traditional trace visualization relies on noticing deviations from the ideal in the layout, we can compute deviations according to various metrics described in Section 4.

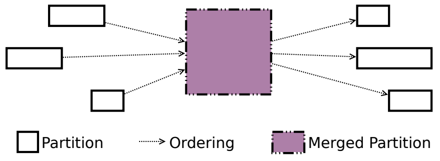
Fig. 1 (right) shows an example logical trace, arranged so that operations that *could* happen simultaneously are ver-



(a) Matching sends and receives indicate operations 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. 2: Mandatory Partitions

tically aligned, while the left side shows a traditional view. In both visualizations, time proceeds from left to right, but the real-time layout on the left obfuscates the performance problems. Our layout clearly highlights issues by coloring them accordingly, and it makes the relationship between the algorithm and the timing problems more clear. The remainder of this section describes the ordering constraints and layout rules that make this possible.

Lamport Clocks. Logical structure is built upon definitions from Lamport [26]. 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 is called the *clock condition*. Our assigned logical steps satisfy Lamport's clock condition, but we add further constraints when Lamport clocks only provide a partial ordering. Our constraints aim for a more intuitive alignment.

Simultaneous Operations. Rather than assuming that each event happens as early as possible, we aim to discover the operations (paired entry and exit events) that conceptually should happen simultaneously. For example, all send operations at one level of a binomial tree broadcast are conceptually simultaneous. As another example, consider MPI *collectives*. A collective communication operation is one in which all processes in an application must participate at once. Collectives are useful as they allow applications to easily leverage highly optimized implementations of complex, distributed algorithms. We assume all operations in the same blocking collective[1] occur simultaneously. From the developer's point of view, individual calls on each process occur together as a logical abstraction. Rooted collectives like `MPI_Bcast` imply an order between the call on the root and calls on all other participating processes, but we avoid defining collective-internal happened-before rules in order to match the way collectives are invoked.

Phases. 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. Our phases are thus very fine-grained. To match this intuition, we ensure that phases, whether detected by our algorithm or specified by the user, do not overlap. In terms of the clock condition, this means that phases $P \rightarrow Q$ with operations p_i, q_i, $C(p_i) < C(q_j)$, $\forall p_i \in P, q_j \in Q$. This condition ensures that the ordering within one phase is not affected by other phases.

1. We do not support non-blocking collectives currently. Hereafter, when we refer to 'collectives', we assume them to be blocking.

We focus on communication operations because they impose happened-before relations between processes, contributing to a global happened-before order built from single-process timelines. We create operations spanning the time between messages to represent the non-communication activities. In the remainder of this section, we describe the two steps of logical structure creation: phase partitioning and logical time step assignment.

3.1 Phase Partitioning

The primary reason for organizing all communication operations into phases is to match the intuition of developers with clear happened-before relations between phases. This step also has a number of practical advantages. For example, *partitioning* the trace into phases makes the computation and analysis a per-partition rather than a per-trace activity, significantly simplifying and accelerating the analyses of Sections 3.2 and 4. We present graph-based algorithms that first identify inseparable groups of operations, then merge them to define phases. In general, phase detection is a difficult challenge [27], [28], [29], [30], [31], [32], especially since the "correct" partitioning can be subjective or ill-defined. We give users the additional option to specify their own partitioning to accommodate application-specific details.

Mandatory Partitions. Our algorithm starts by identifying *mandatory partitions*: groups of operations that cannot be separated due to ordering constraints or semantic reasons. Given a set of MPI operations with their happened-before relations represented as a directed acyclic graph (DAG), we construct the partitioning in a bottom-up fashion. We initially assign each operation its own partition (Fig. 2a). Semantically, a matching send and receive, or each *message*, should belong to a single partition, and we merge their corresponding partitions (Fig. 2b). Similarly, the partitions of all operations in the same collective invocation should be merged. These merges can introduce cycles in the graph, as shown in the figure, which prevents a linear ordering among the partitions and thus the ability to apply happened-before ordering between partitions. To restore a linear order we merge all partitions that form a strongly connected component, restoring the partition graph to a DAG (Fig. 2c). The resulting partitions are minimal groups of operations that support a total order without separating messages and collectives. In practice, the resulting partitions are often fine-grained as even simple operations like allreduce can

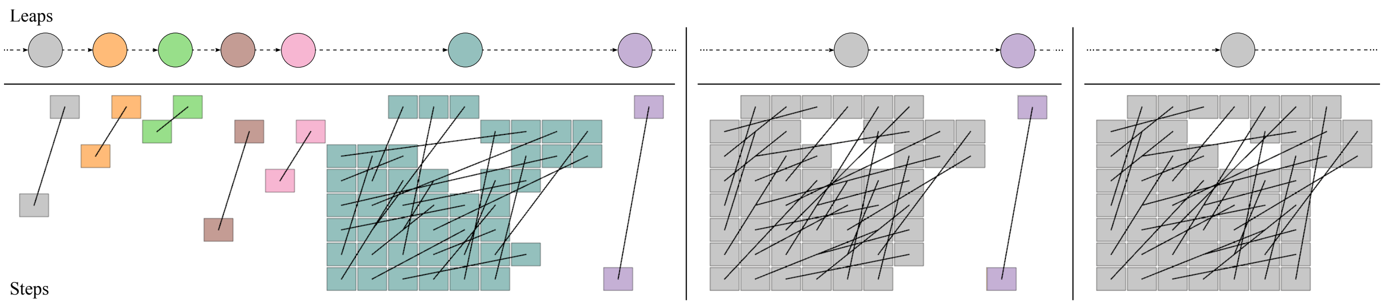


Fig. 3: Merging Leaps in LULESH. The graph in terms of leaps is on top. At bottom are the individual processes as they would be stepped with the leap graph. The result of the strongly connected component merge is the left image. The gray leap merges in succeeding leaps until it contains all processes, 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.

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 MPI applications is `MPI_Waitall`, which causes a process to wait until a given set of prior MPI operations has completed. As an additional semantic constraint, we merge all partitions containing operations associated with the same `MPI_Waitall`. If these associations were not recorded in our traces, we determine the set heuristically and make this merging optional in case the heuristic does not apply. We assume that all calls associated with the `MPI_Waitall` are not interspersed with receive, collective, wait, or test operations. Thus, we assume that all send operations between the last such call and an `MPI_Waitall` belong to that `MPI_Waitall`. The reason receive operations are included in the first list is that in the trace any receive associated with the `MPI_Waitall` ends during the `MPI_Waitall`.

Leap Partitions. There may exist messages that do not result in a strongly connected component as in Fig. 2, but still logically belong together in the same phase. Fig. 3 shows an example taken from an eight process trace of LULESH [3]. Many of the communication operations have been grouped, resulting in the multi-message blue partition. However, a few messages on either side are isolated and are thus excluded. In bulk synchronous codes like LULESH, we expect all processes to participate in each communication phase, and we optionally merge partitions until this property is satisfied. More formally, we define a *leap* as all the partitions with the same graph distance from the sources of the partition DAG[2]. We merge partitions until each leap contains operations from all processes using Algorithm 1.

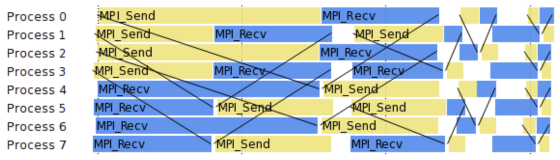
Starting from the first leap, we determine whether it is complete (contains operations from all processes). If it is not, we begin to process its member partitions. Each partition computes its incoming *leap distance* as the minimum of the first operation entry time for each of its processes and the operation exit time of their previous operation in the partition's previous-leap neighbors. Similarly, its outgoing leap distance is the minimum of the last operation exit time for each of its processes and the operation entry time of their next operation in the partition's next-leap neighbors. By

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

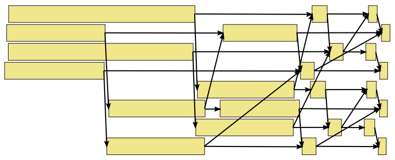
```
complete_leaps (partitions);
    all_leaps = compute_leaps (partitions);
    k = 0;
    while $k < |all\_leaps|$ do
        leap = all_leaps[k];
        changed = TRUE;
        while changed and not complete (leap) do
            changed = FALSE;
            for p in partitions (leap) do
                incoming = leap_distance (p, k-1);
                outgoing = leap_distance (p, k+1);
                if incoming $\ll$ outgoing then
                    merge_into_previous_leap (p);
                    changed = TRUE;
                else
                    for c in children (p) do
                        if will_expand (c, leap) then
                            absorb_partition (p, c);
                            changed = TRUE;
                        end
                    end
                end
            end
        end
        if not complete (leap) and force_merge then
            absorb_next_leap (leap);
        else
            k = k+1;
        end
    end
```
Algorithm 1: Complete leaps through merging partitions.

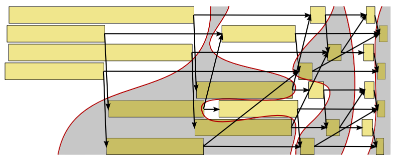
construction, any previous leap is complete. We thus prefer merging at the leap (absorbing un-processed partitions from the next leap) to merging backwards (extending completed leaps). In practice, we only consider merging backwards if the incoming leap distance is more than an order of magnitude smaller than the outgoing one.

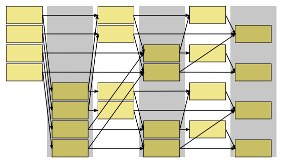
Once the direction of a potential merge has been established, we always merge with all previous-leap parents when merging backwards, but we only absorb a partition from the next leap if it would expand the set of processes in the current leap. Thus, the current leap can shrink or grow. We repeat this 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

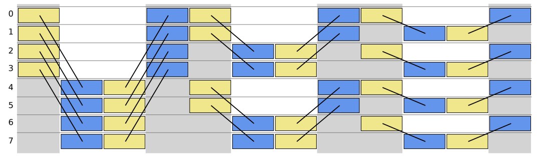


(a) Vampir rendering of a stencil in the pF3D communication benchmark.


(b) We create a happened-before graph of only send operations.


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


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


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

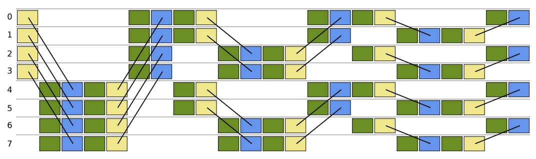

(f) We insert aggregated operations (green) before each communication operation, so all physical time is represented as operations. We use this for analysis in Section 4.

Fig. 4: Step Assignment. Send operations are yellow, receive operations are blue, and alternating white and gray denotes stride boundaries.

successors before restarting the leap merge algorithm or to accept the incomplete leap and continue.

In the example of Fig. 3, 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. As this completes the leap, the algorithm moves onto the next leap which contains only the purple partition. This partition merges backward since in this case it is significantly closer to the incoming gray 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.

Leap merging should be performed when the user suspects that the application generally engages all processes at a coarse scale. This is true in bulk synchronous codes where it is reasonable to assume the partition DAG is a path and each phase is understood to contain all processes. However, this is also true in cases where the partition DAG branches, but every process can be assumed to be active in one of the parallel executing phases. Forcing the merge should be done when the user is confident all processes are active throughout the trace at phase-granularity.

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

3.2 Local and Global Step Assignment

In Section 3.1 we create a DAG of partitions containing related communication operations. Next, we assign logical steps within each partition locally, then globally across partitions, defining the logical structure.
The local step assignment follows three simple principles:

1) All happened-before relationships must be strictly maintained, i.e., $a \rightarrow b$ implies $step(a) < step(b)$;
2) All operations of the same collective invocation happen simultaneously; and
3) Send operations have a greater impact than receive operations on the communication structure.

Item 3 is a consequence of the fact that the order of receive operations is not always uniquely defined by the program and some operations, such as `MPI_Waitall`, may serve as the receiving operation for multiple send operations. Consequently, we initially use only the send and collective operations to define the communication structure. Once the local (per-partition) order for these operations has been determined we introduce the receive operations and ultimately the non-communication operations to the per-partition step assignment. We use an eight process run of the pF3D communication benchmark, shown in Fig. 4, as a working example throughout our explanation.

Strides. The send and collective operations in a trace typically comprise most of the communication structure. We start to assign local steps by grouping these operations into *strides*. Strides are defined by the graph distances from the beginning of the partition, considering only send and collective operations. More specifically, for each partition, we create a sparse version of the happened-before graph, containing only send operations, collective operations, and their aggregated dependencies (Fig. 4b). To maintain the condition that collectives occur simultaneously, we ensure that any happened-before dependencies of one operation in the collective apply to *all* operations in the collective. We next group send and collective operations according to their stride (Fig. 4c). We align the strides in logical time and assign preliminary steps accordingly (Fig. 4d). Not all processes must contain an operation in all strides.

After positioning send operations, we re-introduce receive operations so that the ordering is preserved. We assign all send and collective operations within a stride to the same step, and we place receive operations as early as possible while still maintaining their happened-before relationships (Fig. 4e). Finally, we insert aggregated *non-communication* operations representing all processing between communication operations (Fig. 4f) such as computation or idling.

Aggregated operations allow us to account for all phys-

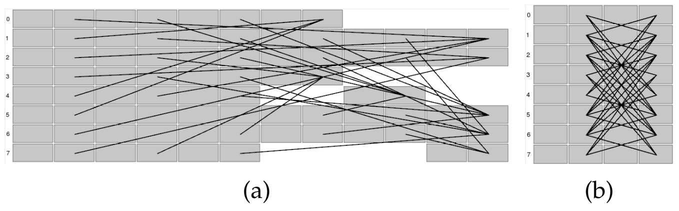


Fig. 5: One partition of an eight process SMG2000 trace, (a) without and (b) with coalescing of `MPI_Isend`s.

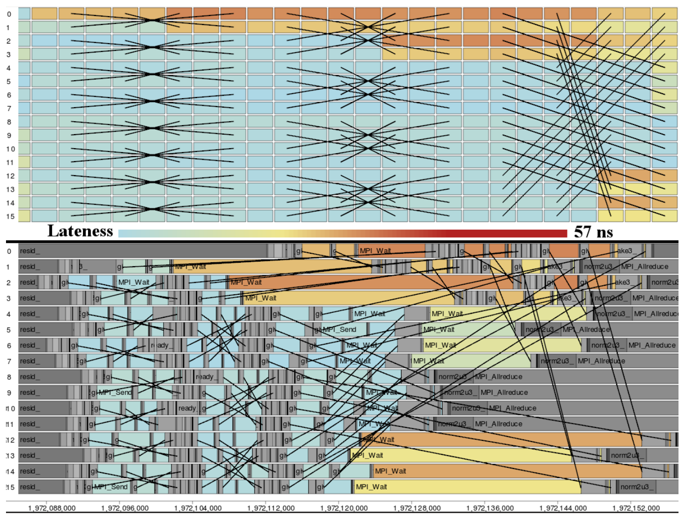


Fig. 6: Logical (top) and physical (bottom) time visualization of a 16 process execution of MG. Communication operations are colored by lateness. The first process becomes late during an aggregated non-communication operation. The lateness spreads through messages to the other processes.

ical time spanned by the trace, which is helpful when defining temporal metrics (Section 4). 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, enforcing global happened-before relationships.

3.3 Isend Coalescing

Throughout our structure extraction pipeline, we have defined an operation to be the matching enter and exit of a single procedure call. However, we have found it useful to optionally coalesce uninterrupted, unbounded sequences of neighboring `MPI_Isend` enter and exit records into a single operation before starting our extraction routine. The resulting operation has an enter time of the first `MPI_Isend` and the exit time of the last. Messages associated with any of the composite calls become associated with the coalesced operation. As individual `MPI_Isend` operations are short and non-blocking, coalescing them can reveal structure that is obscured when processes have different `MPI_Isend` invocation counts without misrepresenting dependencies. In general, we recommend coalescing, unless the user requires more detail about the individual calls and knows they should be aligned at the level of the individual operation.

Fig. 5 shows the results of our structure extraction for a single partition of an eight process SMG2000 [5] trace with and without coalesced `MPI_Isend`s. In Fig. 5a, the `MPI_Waitall` operation happens at different steps for each process depending on how many messages that process is waiting for (which depends on the part of the boundary the process computes) and where in the order of the sending processes each of those send operations falls. Fig. 5b effectively eliminates these concerns, perfectly aligning all send operations and all receive operations. This depicts the symmetry of the exchange. Encouraging comparisons between like operations without violating ordering is also beneficial when metrics based on these comparisons are applied (see Section 4 and Section 5.3).

4 Temporal Metrics

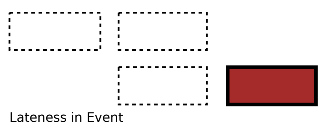
By design, we avoid relying on wall-clock timing information when determining the logical structure of a trace. This produces a well-aligned version of the trace where the relationship between potentially simultaneous operations is clear. It also avoids problems with clock skew and synchronization. Ultimately, however, the timing of operations determines where delays or bottlenecks occur and which part of the program is responsible. We therefore preserve the temporal information by computing metrics from timestamps. In the visualization, these metrics can be mapped onto the logical structure as highlights. This makes delays easy to see without obscuring the logical layout.

Lateness. Simple metrics, such as the entry time, exit time, or duration of an operation, can be computed directly without the logical structure. However, the true power of our technique comes from comparing such simple per-operation metrics within logical partitions. For example, comparing the exit times of operations in the same logical step allows one to track delays. In particular, we define the *lateness* of an operation as the difference between its own exit time and that of the earliest operation in its step in the partition:

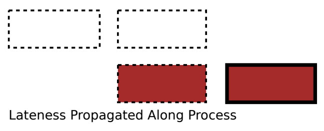
$$l_{op} = op.exit - \min\{x.exit | op, x \in P, x.step = op.step\}$$

where P is the set of operations within a partition. In bulk-synchronous codes leaps typically contain operations from all processes and thus lateness is calculated 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 operations are compared.

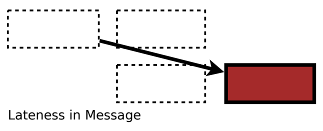
Fig. 6 shows a portion of a 16 process MG trace visualized in Ravel with communication operations colored by the lateness metric. Ravel displays both a traditional physical timeline and a logical timeline. Both views show a delay in a non-communication operation on the first process which propagates 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 operation depending on whether the operation in question is receiving a message or not. A late, non-receiving operation 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 operation whose corresponding sending operation



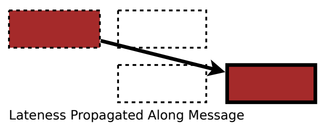
(a) A non-receiving operation is late but its predecessor is not, implying the operation itself caused the delay.



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



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



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

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

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 the receive, which could be caused by, e.g., a slow buffer allocation. Finally, a late receive operation with a matching late send operation indicates lateness propagation across processes (Fig. 7d). The aggregated non-message operations 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 tree-structured reduction rooted at a single process resets lateness, as does a barrier or a simple load imbalance that propagates globally through neighbor exchanges.

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 operation 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 operations and the lateness at exit time:

$$d_{op} = \max\{l_{op} - \max\{l_x | x \to op, \nexists y \text{ s.t. } x \to y \to op\}, 0\}$$

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

5 Evaluation

We execute applications on two radically different architectures: a large Blue Gene/Q (BG/Q) system and an Infiniband cluster with 12 Sandy Bridge cores per node. The Intel cores are split across two sockets, with 6 cores per socket. The BG/Q system uses IBM's compute node kernel OS and a custom MPI implementation. The Infiniband cluster uses a Red Hat derived Linux distribution and the MVAPICH MPI implementation. On both machines we obtain our traces in Open Trace Format 2 (OTF2) [33] using Score-P [34] or Open Trace Format (OTF) [35] using VampirTrace [36].

Our structure extraction algorithm is implemented in Ravel, a desktop application. Our algorithm operates on messaging operations, so the runtime primarily depends on the number of messages, not the number of processes or the total timespan of the trace. The current implementation relies on a preprocessing step to match all messages; this can be memory intensive for large message counts.

5.1 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. From a visualization perspective, collectives have dense communication patterns, with many messages sent between processes at around the same time. This poses a challenge for existing trace tools, especially when the system is noisy and dependence chains across MPI processes are perturbed. We use these as a case study of our tool to demonstrate its ability to correctly determine logical structure and to show how we can display collective operations in an intelligible manner. For our experiments, we used libNBC [8], 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. 8 shows the unprocessed trace as visualized by Vampir and its extracted logical structure as 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 operation 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. 9 shows the logical time view of a ring implementation of `MPI_Allreduce`, colored by lateness. In this $O(P)$ algorithm, each of the P processes 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 further observe the spread of lateness from the 45th process and its continued effects through the remainder of the rounds. Also visible are a 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 exami-

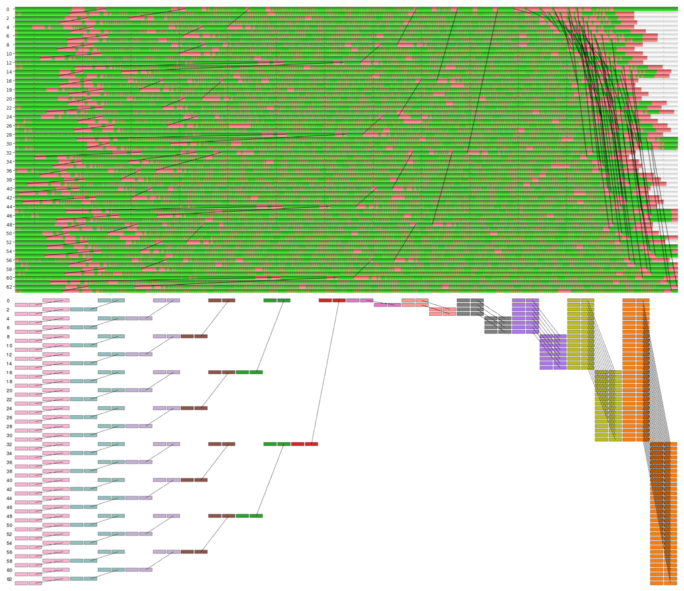


Fig. 8: Visualization of 64 process binomial tree `MPI_Allreduce` in physical time by Vampir (top) and logical time by Ravel (bottom). In logical time, we color by communication partition. We are able to identify the binomial tree levels though they overlap in physical time.

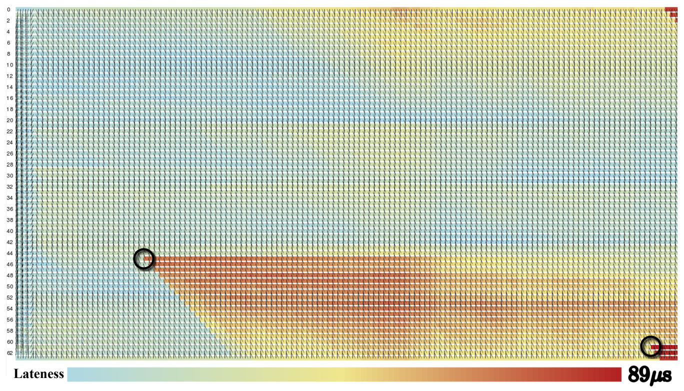


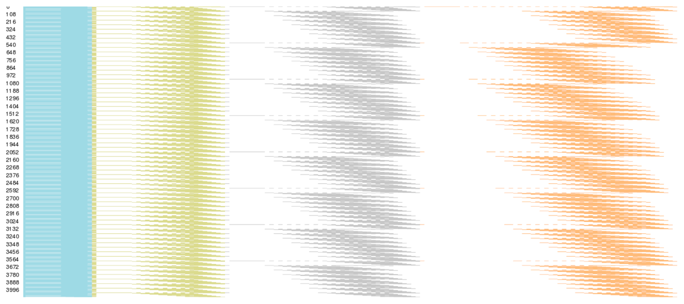
Fig. 9: Ring algorithm `MPI_Allreduce` on 64 processes. Coloring is done by lateness, showing propagation. We find two operations with high differential lateness (circled).

nation reveals lateness is injected at a messaging operation, possibly due to congestion on the network.

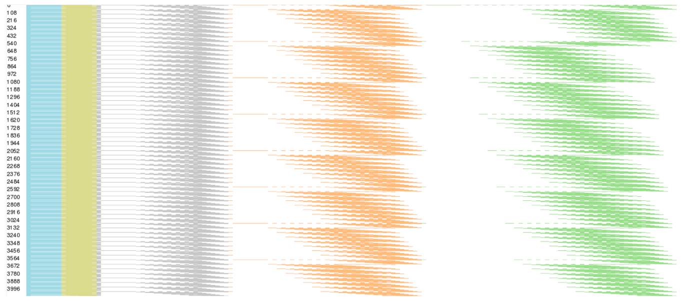
5.2 Massively Parallel Merge Trees

Merge trees are topological structures that can aid in the analysis of data from large parallel simulations [37], [38], [39]. We consider a massively parallel algorithm to compute them *in situ* [9], which avoids the limitations and penalties of writing out the data to be analyzed post mortem. Previously, we demonstrated how visual analysis of logical structure helped locate a sub-optimal message order in an early merge tree implementation [2]. Here, we analyze the validity of the logical structure extracted from this code.

In the merge tree algorithm, each process running the simulation must compute a local merge tree over features of interest in the simulation data. These local merge trees are sent to *gather processes* which combine the local trees and



(a) Logical steps resulting from developer partitions.



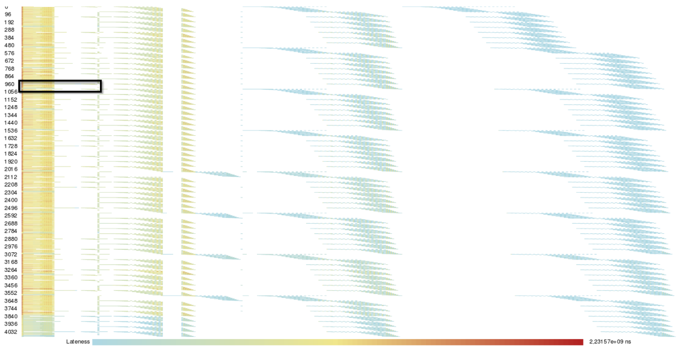
(b) Logical steps resulting from our partitions.

Fig. 10: 4,096 process merge tree, 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 are highly similar.

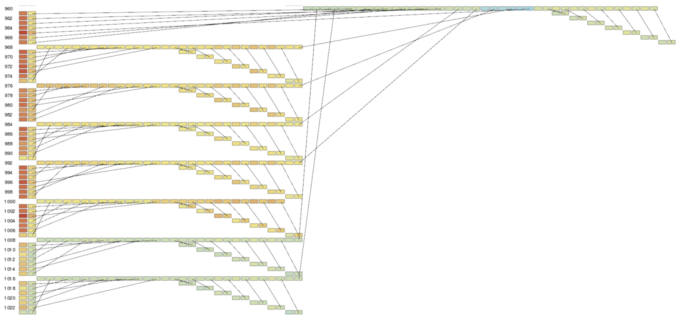
send to the next level of gather processes. The algorithm iterates until a global merge tree is computed. The gather processes are organized as a k-ary tree, where one process of a set of k siblings at one level acts as the group's gather process at the next level. At the end of each round, the intermediate merge tree is sent not only to this gather process, but also down along the tree to the leaves.

We obtained communication phase information from the developers and compared it to our partitioning using the leap merge and merging across global steps. Fig. 10 shows a 4,096 process, 8-ary merge-tree where operations are colored by membership in their logical partition as determined by the developer in Fig. 10a and our algorithm in Fig. 10b. The main difference is that our algorithm breaks the initial phase into the up and down partitions (to and from the gather process). Even with this difference, the resulting logical structures are very similar. In both images, the first level of the gather tree structure is apparent from the eight parallelograms stacked vertically in the process space – indicating the eight parallel subtrees.

The parallel merge tree algorithm uses asynchronous primitives for its communication. However, an early development version of the algorithm used synchronous primitives. Closer examination of this early implementation reveals some of the pitfalls of strict adherence to happened-before ordering. Fig. 11b shows eight gather groups of eight leaves at the beginning of a 4,096 process, 8-ary merge tree trace with partitioning given by the developers. 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 gather process. However, the top set of eight leaves is later than its sibling groups for the level-



(a) Entire trace.



(b) Detail shows shifting of steps based on happened-before ordering.

Fig. 11: Trace from an older merge tree implementation, on 4,096 processes. In (a), the entire merge tree trace is shown colored by lateness. The portion in the black box is shown in more detail in (b). The top gather process receives messages from the gather processes of faster groups. To enforce the true ordering of operations, send operations of the initial gather must be shifted toward the right, preventing logically parallel communication from being assigned the same step.

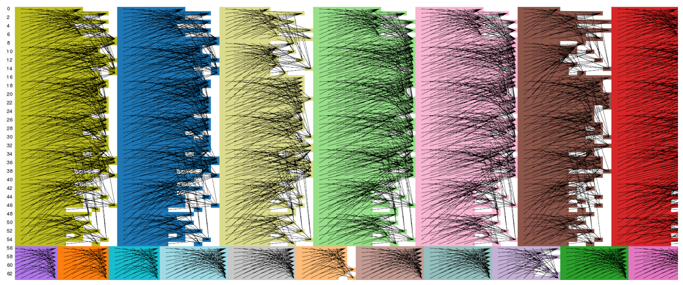


Fig. 12: Partitioning of AMG2013's solver algorithm executed on 64 processes. The last eight processes partition independently of the others.

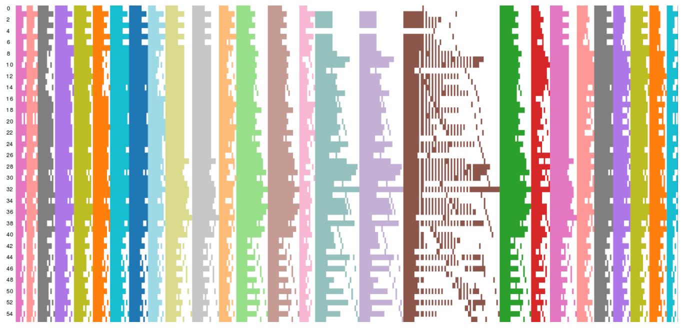


Fig. 13: Partitioning of a single iteration of the AMG2013 solve, focusing on the first 54 processes. The individual partitions match well with what we expect from the levels of the V-cycle. We see the amount of communication increase as processes in coarser levels gain more neighbors. Eventually some processes become inactive due to coarsening, resulting in white gaps across the blue and lavender partitions preceding the brown one.

1 gather. The level-1 gather processes of those siblings send their results to the topmost process (which will act as a level-2 gahter process) before the it receives all messages from its (level-0) children in its role as a level-1 gather process. Thus, the early receive causes the steps to be misaligned. The remaining communication operations of the top-most process are shifted towards the right. We observe this effect among multiple gather groups with varying severity. This is seen as horizontal lines of different lengths in the full view (Fig. 11a). Sends are shifted back to the leaves, several steps to the right.

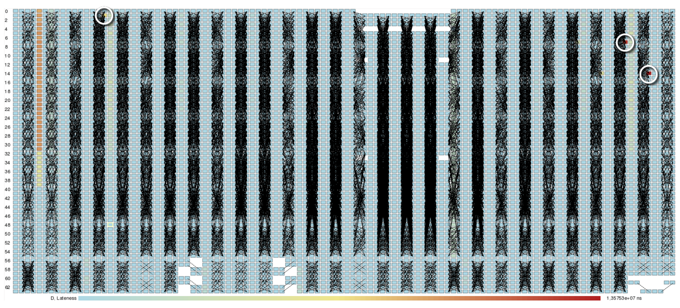
This effect is an unavoidable consequence of the happened-before ordering. However, in this case the early message does not actually change the order of computation. Allowing the stepping algorithm to violate the happened-before relation to create the expected regular patterns could potentially result in a more intuitive visualization and more meaningful metrics. However, it is not clear under which circumstances such a re-ordering should be permissible. This will be the subject of future research.

5.3 Algebraic Multigrid

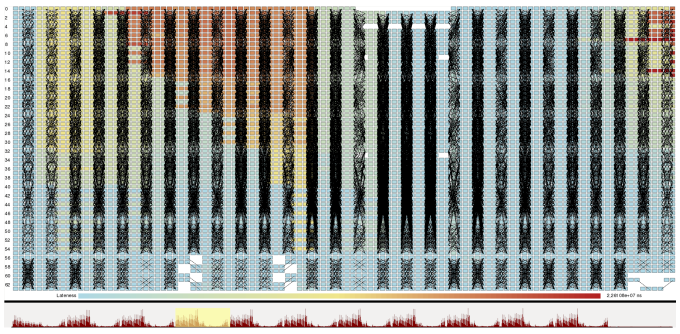
Algebraic multigrid techniques solve sparse linear systems that may or may not be associated with an actual spatial grid. The method begins with a fine-grained grid or matrix that is successively coarsened until it can be solved with reasonable error. It interpolates back from the coarsened solution to the fine-grained one. This so-called V-cycle is repeated until it has converged. We examine an algebraic multigrid method implemented in the *hypre* scalable solver library [10], via the AMG2013 benchmark, which is part of the CORAL [11] benchmark suite. This gives us an opportunity to verify our structure algorithm for a more complicated example.

Fig. 12 shows a portion of the logical structure we extracted, using the leap merge option, from a 64 process trace of the AMG2013 solver algorithm executed on the BG/Q machine. The operations are colored by partition. Our structure separates the first 54 processes into distinct partitions from the remaining eight. This led us to examine the rest of the structure and discover that the two process groups never interacted within the solve. Upon consulting with the development team and verifying the results, we learned that the final eight processes are assigned to the anisotropic portion of the domain, which explains why they behave differently and independently, as found by our logical structure.

We narrow our focus to a single iteration, shown in Fig. 13, once again colored by partition. For simplicity we show only the 54 process partition and omit message lines. In the solve, each level performs a relaxation step and



(a) The first 40 processes exhibit high differential lateness early on due to greater computational requirements. A few `MPI_Waitall` operations (annotated with white circles) also show high differential lateness.



(b) The overview on the bottom shows eleven repetitions of the lateness profile, corresponding to the iterations of the V-cycle. Lateness spreads from the operations of high differential lateness in Fig. 14a.

Fig. 14: Logical structure of AMG2013 cycle on 64 processes.

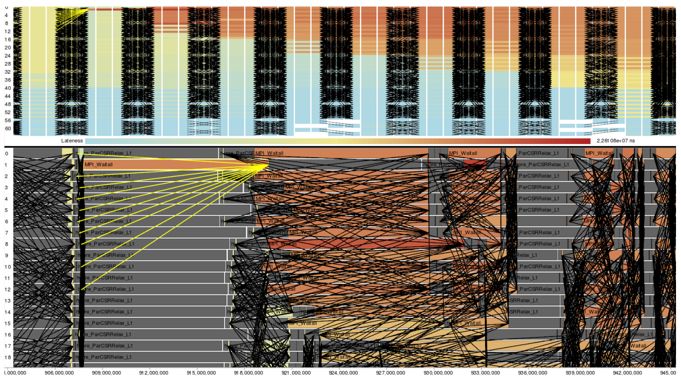


Fig. 15: Ravel logical and physical view of an `MPI_Waitall` with high differential lateness in the AMG2013 solve cycle. Messages to the `MPI_Waitall` are highlighted in yellow.

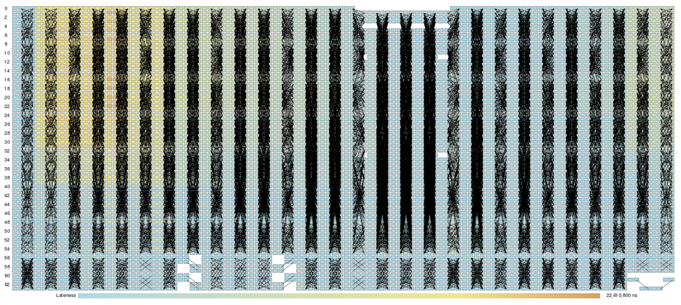


Fig. 16: Logical structure of an AMG2013 solve cycle on 64 processes with an asynchronous progress thread running. Compared to Fig. 14b, the lateness due to `MPI_Waitall`s is gone, leaving only the computation lateness.

one to two matrix-vector products. Our algorithm separates these in partitioning. Initially, the partitions are short, but as the grid gets coarser, participating processes need to send information to more neighbors, resulting in longer partitions representing the increased communication. At the same time, the coarsening leaves some processes without work, seen as white gaps (no operations) across the blue and lavender partitions preceding the long brown one. This behavior is expected, suggesting the logical structure extracted by our partitioning approach is consistent with AMG's algorithm design.

The AMG2013 solve cycle generally uses a sequence of `MPI_Irecv`s, `MPI_Isend`s, and an `MPI_Waitall`. This makes it a good candidate for isend coalescing (Section 3.3) to examine lateness. Fig 14a shows the operations of a single solve cycle colored by differential lateness. The coarse partitions where some processes do not participate are apparent from the white gaps in the operations across ranks 0 and 4, as well as from the denser lines in those steps among other processes that are sending to more neighbors. We observe the first 40 processes exhibit high differential lateness in the computation following the first communication volley, indicating an imbalance. We also note high differential lateness in a few `MPI_Waitall` operations, circled in white.

Examining lateness instead (Fig. 14b), we see the propagation of lateness from the abnormal `MPI_Waitall`s. Note that the lateness overview at the bottom reveals a periodic pattern of lateness over logical time. The number of repetitions corresponds to the number of iterations of the V-cycle reported by the run, indicating this is a recurrent issue.

We focus on one of these `MPI_Waitall` operations in Ravel's physical time view (Fig. 15). The operation is selected in both the logical and physical time views with the associated message lines highlighted in yellow. From our logical steps and lateness metrics, we were able to locate this aberrant ∼12ms `MPI_Waitall` quickly and learn that its expected behavior is like the short `MPI_Waitall` calls before it, rather than the longer ones after it. When differential lateness is high for a receiving operation, we expect the message itself to be the cause (Fig. 7c). In this case, all the messages come from processes on the same node, so the network could not have contributed to the lateness. Furthermore, other `MPI_Waitall` calls complete much more rapidly. We note that the late `MPI_Waitall` does not complete until the next round of `MPI_Waitalls` begin on its senders. We hypothesize that this may be an asynchronous progress issue – the `MPI_Waitall` occurs late enough that the MPI implementation does not handle the outstanding `MPI_Isends` until the next step's `MPI_Waitalls`. We run AMG2013 again, this time setting the `PAMID_ASYNC_PROGRESS` environment variable, which uses a separate thread to make asynchronous progress at the cost of potentially greater latency. The differentially late `MPI_Waitalls` are no longer present in the resulting trace (Fig. 16), leaving only computation-based lateness.

6 Conclusions and Future Work

We have presented a new approach for analyzing the execution traces of message passing programs. We extract a logical structure to capture the developer's intended ordering of operations. This technique uses happened-before relationships not only on the scale of individual events, but also on the scale of communication phases and even concurrent send operations. We explicitly hide timing information in our logical view. This clearly shows relationships among operations, and we use temporal metrics mapped onto this structure to highlight timing problems without clutter. In particular, we calculate each operation's delay relative to its logical peers, providing an abstract view of lateness. Using the happened-before relationship encoded in the logical structure, we are able to pinpoint the cause of a bottleneck and to study its propagation.

Through a series of case studies, we have demonstrated that our algorithm can identify structures across a variety of communication profiles. We have shown that our metrics correctly identify communication delays and their sources. We also found an example where derived structure was not ideal (Fig. 11). In future work, we plan to improve our heuristics for message passing programs, expand the types of operations and dependencies handled by our structure extraction algorithm, and further leverage logical structure for detection of performance issues. We will further investigate how our visual approach compares to automatic approaches in trace analysis, documenting the strengths, limitations, and best-suited problems for each.

Acknowledgments

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 and supported by Office of Science, Office of Advanced Scientific Computing Research as well as the Advanced Simulation and Computing (ASC) program.

References

[1] 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.

[2] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann, "Combing the communication hairball: Visualizing large-scale parallel execution traces using logical time," *IEEE Trans. on Vis. and Comp. Graphics, Proc. InfoVis '14*, no. 12, 2014.

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

[4] 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.

[5] Accelerated Strategic Computing Initiative, "The SMG2000 benchmark," 2001.

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

[7] 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.

[8] 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.

[9] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P.-T. Bremer, "In-situ feature extraction of large scale combustion simulations using segmented merge trees," *Proc. ACM/IEEE Conf. on Supercomputing (SC14)*, SC'14. Nov. 2014.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] C. Schaubschläger, D. Kranzlmüller, and J. Volkert, "Event-based program analysis with DeWiz," in *Proceedings of the Fifth International Workshop on Automated Debugging AADEBUG2003*, 2003.

[21] T. J. LeBlanc, J. M. Mellor-Crummey, and R. J. Fowler, "Analyzing parallel program executions using multiple views," *J. Parallel Distrib. Comput.*, vol. 9, no. 2, pp. 203–217, Jun. 1990.

[22] 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.

[23] 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.

[24] T. Gamblin, B. R. de Supinski, M. Schulz, R. J. Fowler, and D. A. Reed, "Clustering performance data efficiently at massive scale," in *International Conference on Supercomputing*, Tsukuba, Japan, June 1-4 2010.

[25] ——, "Scalable load-balance measurement for SPMD codes," in *Supercomputing 2008 (SC'08)*, Austin, Texas, November 15-21 2008, pp. 46–57.

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

[27] 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.

[28] ——, "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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] D. Eschweiller, M. Wagner, M. Geimer, A. Kn upfer, W. E. Nagel, and F. Wolf, "Open Trace Format 2: The next generation of scalable trace formats and support libraries," in *Applications, Tools, and Techniques on the Road to Exascale Computing*, ser. Advances in Parallel Computing, K. De Bosschere, E. H. D'Hollander, G. R. Joubert, D. Padua, F. Peters, and M. Sawyer, Eds. IOS Press, 2012, pp. 481–490.

[34] A. Knüpfer, C. Rössel, D. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Springer Berlin Heidelberg, 2011, pp. 79–91.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

Katherine E. Isaacs is a Ph.D. candidate at the University of California, Davis researching information visualization techniques for performance analysis. In 2012 she was awarded a Department of Energy Office of Science Graduate Fellowship (DOE SCGF). She completed a B.S. in computer science and a B.A. in mathematics at San José State University and a B.S. in physics at the California Institute of Technology.

Todd Gamblin is a Computer Scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research focuses on scalable tools and algorithms for measuring, analyzing, and visualizing the performance of massively parallel applications. He leads several projects in these areas, and has been at LLNL since 2008. Todd received the Ph.D. and M.S. degrees in Computer Science from the University of North Carolina at Chapel Hill in 2009 and 2005. He received his B.A. in Computer Science and Japanese from Williams College in 2002. He has also worked as a software developer in Tokyo and held graduate research internships at the University of Tokyo and IBM Research. Todd recently received an Early Career Research Award from the U.S. Department of Energy.

Abhinav Bhatele is a computer scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His interests lie in performance optimizations through analysis, visualization and tuning and developing algorithms for high-end parallel systems. His thesis was on topology aware task mapping and distributed load balancing for parallel applications.

Abhinav received a B. Tech. degree in Computer Science and Engineering from I.I.T. Kanpur, India in May 2005 and M.S. and Ph.D. degrees in Computer Science from the University of Illinois at Urbana-Champaign in 2007 and 2010 respectively. Abhinav was an ACM/IEEE-CS George Michael Memorial HPC Fellow in 2009. He has received several awards for his dissertation work including the David J. Kuck Outstanding MS Thesis Award in 2009, a Distinguished Paper Award at Euro-Par 2009 and the David J. Kuck Outstanding PhD Thesis Award in 2011. Recently, a paper that he co-authored with LLNL and external collaborators was selected for a best paper award at IPDPS in 2013.

Martin Schulz is a Computer Scientist at the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL). He earned his Doctorate in Computer Science in 2001 from the Technische Universität München (Munich, Germany) and also holds a Master of Science in Computer Science from the University of Illinois at Urbana Champaign. He has published over 175 peer-reviewed papers and currently serves as the chair of the MPI forum, the standardization body for the Message Passing Interface. He is the PI for the Office of Science X-Stack project "Performance Insights for Programmers and Exascale Runtimes" (PIPER) as well as for the ASC/CCE project on Open|SpeedShop, and is involved in the DOE/Office of Science exascale projects CESAR, ExMatEx, and ARGO. Martin's research interests include parallel and distributed architectures and applications; performance monitoring, modeling and analysis; memory system optimization; parallel programming paradigms; tool support for parallel programming; power-aware parallel computing; and fault tolerance at the application and system level. Martin was a recipient of the IEEE/ACM Gordon Bell Award in 2006 and an R&D 100 award in 2011.

Bernd Hamann is a professor of computer science at the University of California, Davis. He studied mathematics and computer science at the Technical University of Braunschweig, Germany, and received a Ph.D. in computer science from Arizona State University in 1991. His main teaching and research interests are data visualization, data analysis and geometric modeling.

Peer-Timo Bremer is a member of technical staff and project leader at the Center for Applied Scientific Computing (CASC) at the Lawrence Livermore National Laboratory (LLNL) and Associated Director for Research at the Center for Extreme Data Management, Analysis, and Visualization at the University of Utah. His research interests include large scale data analysis, performance analysis and visualization and he recently co-organized a Dagstuhl Perspectives workshop on integrating performance analysis and visualization. Prior to his tenure at CASC, he was a postdoctoral research associate at the University of Illinois, Urbana-Champaign. Peer-Timo earned a Ph.D. in Computer science at the University of California, Davis in 2004 and a Diploma in Mathematics and Computer Science from the Leibniz University in Hannover, Germany in 2000. He is a member of the IEEE Computer Society and ACM.