Combing the Communication Hairball: Visualizing Parallel Execution Traces using Logical Time

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

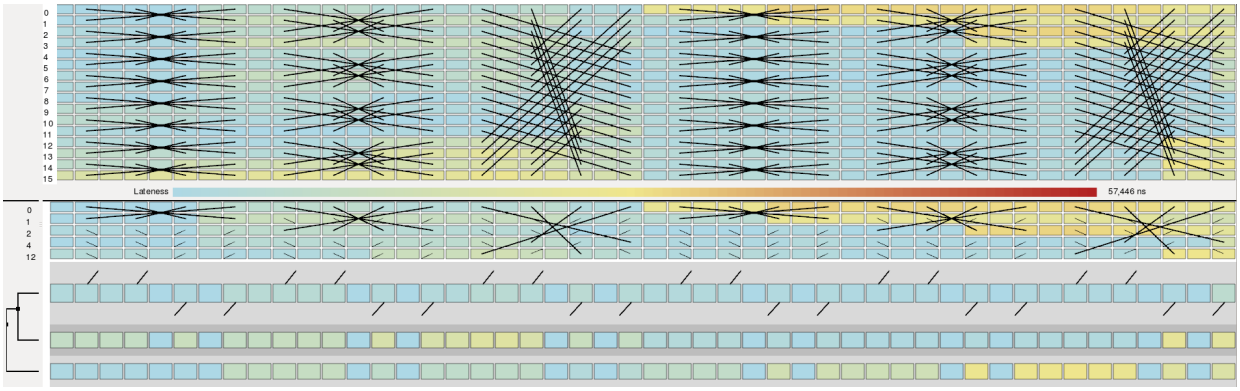


Fig. 1: Logical timeline and clustered logical timeline views from Ravel, a tool for visualizing parallel execution traces. Events are represented by boxes, colored by their wall-clock delay. The use of logical time reveals communication patterns and leverages developers' understanding of their program's structure. We use the logical time structure to cluster on any metric, which allow us to represent large-scale traces using explorable clusters while still depicting messages with full timelines for a subset of processes.

Abstract— With the continuous rise in complexity of modern supercomputers, optimizing the performance of large-scale parallel programs is becoming increasingly challenging. Simultaneously, the growth in scale magnifies the impact of even minor inefficiencies – potentially millions of compute hours and megawatts in power consumption can be wasted on avoidable mistakes or sub-optimal algorithms. This makes performance analysis and optimization critical elements in the software development process. One of the most common forms of performance analysis is to study execution traces, which record a history of per-process events and inter-process messages in a parallel application. Trace visualizations allow users to browse this event history and search for insights into the observed performance behavior. However, current visualizations are difficult to understand even for small process counts and do not scale gracefully beyond a few hundred processes. Organizing events in time leads to a virtually unintelligible conglomerate of interleaved events and moderately high process counts overtax even the largest display. As an alternative, we present a new trace visualization approach based on transforming the event history into *logical* time inferred directly from happened-before relationships. This emphasizes the code's structural behavior, which is much more familiar to the application developer. The original timing data, or other information, is then encoded through color, leading to a more intuitive visualization. Furthermore, we use the discrete nature of logical timelines to cluster processes according to their local behavior leading to a scalable visualization of even long traces on large process counts. We demonstrate our system using two case studies on large-scale parallel codes.

Index Terms—Information visualization, software visualization, timelines, traces, performance analysis.

1 Introduction

Large-scale simulations increasingly form the backbone of a wide variety of science and technology fields [19, 2]. These parallel applications can utilize hundreds of thousands of processors across tens of thousands of nodes. Understanding, let alone optimizing, the behavior of a massively parallel simulation code is a significant and largely unsolved challenge [12]. At the same time, high-end simulations monopolize supercomputers for days or weeks, so even minor runtime improvements translate into significant energy and cost savings, increased throughput, and more science per time.

One common approach to gain more insight into the behavior of a code is *tracing*, which logs events of interest during program execution, providing a detailed history for later analysis. The data can include function invocations for each process, interactions between processes, hardware counter states, or file operations. Even when recording only a subset of events or selected parts of the code, a trace of a moderately sized run may contain gigabytes or terabytes of data. In addition to the sheer scale, traces can be highly complex, making automatic analysis challenging. While there exist solutions to some problems, like identifying the critical execution path [36, 7], in many cases the root causes of performance problems are more subtle than, for example, a single late process. Moreover, fully automatic techniques have been of limited use. Instead, application developers and performance experts have turned to visualization in hopes of identifying patterns and forming new hypotheses to be tested.

Currently, the visual analysis of trace data tends to one of two extremes: on one end, visualizations used in practice attempt to show all of the data with little or no preprocessing [22, 31, 33], which makes the visualization difficult to comprehend even at modest pro-

- *Katherine E. Isaacs, Ilir Jusufi and Bernd Hamann are with University of California, Davis. E-mails: {keisaacs,jusufi,bhamann}@ucdavis.edu.*
- *Peer-Timo Bremer, Todd Gamblin, Abhinav Bhatele, and Martin Schulz are with Lawrence Livermore National Laboratory. E-mails: {ptbremer,tgamblin,bhatele,schulzm}@llnl.gov.*

Manuscript received 31 Mar. 2014; accepted 1 Aug. 2014; date of publication xx xxx 2014; date of current version xx xxx 2014.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

cess counts. At the other end of the spectrum are summary statistical graphs and analysis techniques, which heavily compress or process the data. While some analysis techniques focus exclusively on high-level structure [18, 17, 16], others identify problematic behavior locally [43], but discard the context necessary to discover initial causes or longer term effects. What is missing are visualization techniques that provide productive analysis that is abstract enough to handle large process counts but detailed enough to provide new insight.

Given the goals of performance experts — understand and optimize the behavior of codes — and the state of the existing trace analysis tools, we have identified three design goals for an improved trace visualization. A visualization must:

G1 *Preserve and highlight event patterns and dependencies.* Most applications will exhibit regular patterns in their traces caused by their iterative nature. Detecting these patterns is key to any performance analysis. A visualization must clearly show such patterns. For communication specifically, we must avoid the indecipherable tangle of lines common in current tools.

G2 *Be effective for large numbers of processes.* Many performance problems only manifest at large process counts, so the visualization must scale to be useful.

G3 *Provide multiple levels of detail and abstraction.* Analysts need overviews when little is known about a problem, and the ability to explore in more detail when necessary.

To meet these goals, we decided to visualize a trace in *logical time* inferred from the happened-before dependencies between events. This differs from *physical time*, which existing tools use, and has two main advantages: First, logical time, by construction, corresponds more closely to the code structure, with which a user is intimately familiar, resulting in a more intuitive visual encoding. Second, the discrete nature of logical time allows us to cluster and thus summarize chains of events across processes leading to a scalable, multi-resolution representation. We combine these two properties in *Ravel*, a new visualization tool for parallel execution traces. Ravel provides multiple linked and coordinated views of large-scale traces at different levels of abstraction. Unlike existing tools, our primary axis is logical time with other metrics that reflect physical time based properties, such as *lateness*, encoded using color maps. This allows developers to easily detect differences in structure between processes or different versions of the code as well as to understand differences in, for example, timing when comparing similar structures. In summary our contributions are:

1. A logical, rather than physical, parallel interactive timeline visualization that still captures physical timing data.
2. A distance metric for clustering logical timelines based on physical time behavior.
3. Ravel, a visualization tool for parallel execution traces using multiple coordinated views.

2 Background and Related Work

Modern supercomputers are composed of a network of nodes, where each node has compute cores and local memory. Large-scale parallel applications divide computational work among individual processes by assigning a piece of the overall work to each core. This is called *domain decomposition*. For the computation to proceed, processes typically must exchange data. A common model is to send *messages* between processes. The Message Passing Interface (MPI) [35] is the most widely used standard for this kind of programming model in high performance computing. Under MPI, developers explicitly call routines for sending and receiving messages between processes. Each process is assigned a unique ID for addressing messages and to determine what part of the larger problem to compute. While we focus on applications using MPI here, our approach is general enough that it could be applied to other message passing implementations and dependency-based programming models.

Tracing is a technique that chronologically records actions taken by an application during its execution. We call the resulting record an execution trace. The recorded actions include entering and exiting functions, as well as sending and receiving messages. Events are timestamped. In parallel programs, the process in which each event occurs is also recorded. We combine matching function entry and exit actions and deem the time spent in a function an *event*. We also associate messages with the functions that send and receive them. We call these *send events* and *receive events*.

Most parallel trace visualizations plot stacked per-process event timelines with process or thread numbers mapped to the non-time axis. Functions are represented as rectangular bars that span their lifetime along the time axis, a technique used in Gantt charts. Single process trace visualizations often show the active call stack for each process by arranging function bars similar to an icicle plot. Trumper et al. [41] aligned multiple of these views to show threads, but targeted situations where only a few threads need to be viewed at a time. When several processes are shown, there is not enough space for the call stack, so only the active function is represented. Zinsight's [13] event flow view follows this depiction with functionality to expand process timelines into those of the composite processing elements. However, these traces do not include communication between processes.

Visualizations of this type that depict messages use lines between the two communicating process timelines positioned based on their send and receive times. Actively maintained visualization tools of this type include Vampir [31], Paraver [33], and Projections [22], the latter focusing on Charm++ as an alternative to the MPI model. For a more complete survey of visualizations of various types of traces, including the ones we discuss here, we refer the reader to [21].

As more processes are added to the parallel physical timeline view, the message lines can become an inscrutable hairball, which makes it impossible for users to see, let alone understand, any pattern present in the trace. While *Ravel* uses the same familiar paradigm of parallel timelines, events as rectangles and messages as lines between processes, our approach uses logical timelines rather than physical ones. This untangles the hairball and can reveal messaging patterns. Physical time information, which is still critical for performance analysis, is added back to the plot by color coding the logical time view. We include a traditional physical timeline view, but it is mainly for comparison for users familiar with existing tools as well as for detailed exploration of local areas of interest discovered using our other views.

Ariadne [11], PVaniM [40], and Growing Squares [14] animated message events between processes in logical time for debugging and program comprehension purposes, but did not include physical time information needed for performance analysis. Furthermore, these visualizations portrayed at most tens of processes.

Muelder et al. [29] tackled the issue of scale by changing the vertical axis from processes to event duration and using opacity scaling to plot the events from all processes on top of each other, visually clustering similarly timed events but anonymizing all processes, thereby omitting message dependencies between them. Sigovan et al. [38] plotted event duration versus process ID, incorporating time through animation. This shows individual processes, but not messages between them. Our visualization allows users to explore communication patterns at a level of detail that preserves these relationships.

Vampir includes an option for applying k-means clustering to processes, choosing k by how many clusters can fit on screen [9]. The clustering is done on events in physical time and messages are omitted. We also use clustering, but we use temporal metrics to cluster in logical time. Projections shows outlier timelines when it is not possible to draw all processes. Our clustering view is similar, showing a process of interest and its messaging neighborhood.

Timeline visualizations in other domains share some similarities with this work. An overview of time-series visualization techniques can be in found in [1]. LifeFlow [44] aggregates and aligns numerous event sequence records, placing groups at their mean event sequence time. EventFlow [28] extends this visualization to durational events. The timelines in this data do not have relationships between them.

Several techniques depict interactions between multiple timelines, such as participation in the same event [30], marriages between persons [24], or edits to the same file [32], by changing the proximity

between the timelines. In history flow [42], Wikipedia author contributions to a single article were positioned vertically by the change along the article's text. This visualization could use either physical or per-revision time. These techniques however do not depict multiple separate interactions occurring concurrently, which is the case with many messaging patterns in our traces.

3 Logical Timelines

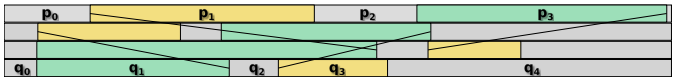
As time is a central element of traces, many trace visualizations represent the time data of events using position on a common scale, the most effective visual channel [10]. While this choice gives the viewer the most fidelity with respect to event start and end time differences, the larger communication structure composed of those events and messages is easily lost. This structure is an important context for understanding the underlying application code. We prioritize this structure by switching from a physical time scale to a logical time scale where events are positioned based on a logical happened-before ordering defined by the semantics of the communication events (e.g., a message has to be sent before it can be received). At the same time, we retain physical time, which is important to understand the overall performance, by color encoding with metrics derived from physical time. In particular, we have had success using a metric we call *lateness*, the delay each event experiences relative to its logical assignment. We explain logical time in Section 3.1 and lateness in Section 3.2. Logical time and lateness further provide a basis for handling issues of scale, which we discuss in Section 3.3.

3.1 Logical Time

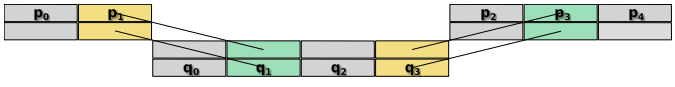
Logical time is a concept based on the order in which the events took place *with respect to each other* rather than compared to physical or wall clock time. Lamport [25] defines the happened-before relation and the Lamport clock, which serve as an initial basis for the logical time we use. Happened-before is a transitive relation ($\rightarrow$) such that: (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, respectively, $s \rightarrow r$. The Lamport clock is a function C that assigns a number to each event such that for events a, b, $C(a) < C(b)$ if $a \rightarrow b$.

We use a clock scheme with additional constraints designed to extract communication structures from the trace [20]. This scheme has a happened-before relation between phases, blocks of execution composing some larger sub-task of the program. Division of the trace events into communication phases can be given by the user or determined computationally based on connected components in the event happened-before graph. Then it enforces the condition that for any two phases P, Q, if $P \rightarrow Q$, then $C(p) < C(q) \forall$ events $p \in P$ and $q \in Q$.

Another feature of this scheme is that, instead of assigning the least possible value required by the conditions thus far, it prioritizes assigning the same values to send events, arguing that these are more important to the structure of the communication. As communication is the focus, non-messaging events between each pair of communication events in a process are aggregated into a single logical event. Following the terminology of this scheme, we refer to logical time values as *logical steps*. We also use the phases determined by the algorithm for clustering (see Section 3.3.1).



(a) Physical timelines.



(b) Logical timelines.

Fig. 2: Portion of a four-process trace. Sends events are green. Receive events are yellow. Non-messaging events are gray. Messages are drawn as black lines between processes. The communication pattern is easily discernible in the logical time view.

Fig. 2a shows a portion of a four-process trace visualized using physical time. Fig. 2b shows the the same events in logical time. In both views, green and yellow rectangles are used to represent messaging events and lines between them represent messages. Non-messaging time is shown in gray. In the physical time view, the width of the rectangle represents duration. In the logical time view, the widths are kept constant to avoid obscuring the communication structure. The communication structure is more apparent using logical time.

3.2 Lateness

As we replace the physical time scale with a logical timeline, we must retain the information captured by physical time data by encoding it in a different way. We do this by coloring the events with a novel metric that shows delays of processes compared to others.

The most straightforward metric for timing data is event exit time, the actual time at which the event completed. Generally, event exit time provides the most useful information since it shows when computation following a communication event is delayed, though we optionally enable other choices, such as enter time or duration, to allow performance analysts to study delay times. A greater exit time value with respect to other events around the same step indicates that the event is behind schedule relative to the others. The greater the discontinuity in color between an event and its predecessor, the greater the duration of the event.

Discontinuities can be difficult to discern when coloring by absolute exit time, especially when viewing a large number of steps. To further highlight the discontinuities, we focus on a metric that represents them using the delay of an event relative to its peers. We call this the *lateness* of the event and calculate it as the difference in exit time between the event and the earliest event sharing its timestep.

Fig. 3 shows a portion of a 16 process MG [3] trace with events colored by lateness. We observe lateness beginning in the tenth process and propagating to the other processes along message lines. Note that because lateness is calculated relative to the other events in a timestep, once the delay has propagated to all processes, each process's lateness drops as soon as all processes are synchronized. We call this phenomenon "resetting." This behavior is advantageous in the visualization because it prevents late events that occur at the beginning of the trace from masking those that occur further towards the end.

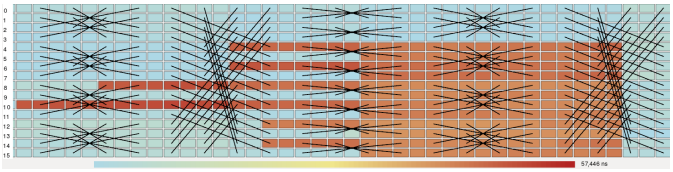


Fig. 3: Logical timeline with events colored by *lateness*, the difference in the wall-clock exit time between the event and the earliest event sharing the timestep. Lateness spreads starting from the tenth process to others waiting via message dependencies until it reaches all processes, at which point it "resets".

3.3 Scale

The amount of screen space that can be devoted to each timeline diminishes as the process count increases. Logical time and lateness provide a convenient platform with which to cluster, allowing us to represent large numbers of processes. We discuss clustering in Section 3.3.1. Our default cluster representations show aggregate sends and receives at each logical time step. We include a mechanism for detecting user-defined motifs and changing their cluster representation to better portray them. We explain this mechanism in Section 3.3.2.

3.3.1 Clustering

We cluster to group processes according to the metric of interest (by default, lateness). Many processes exhibit different behavior between logical time phases, so we apply clustering on a per-phase basis.

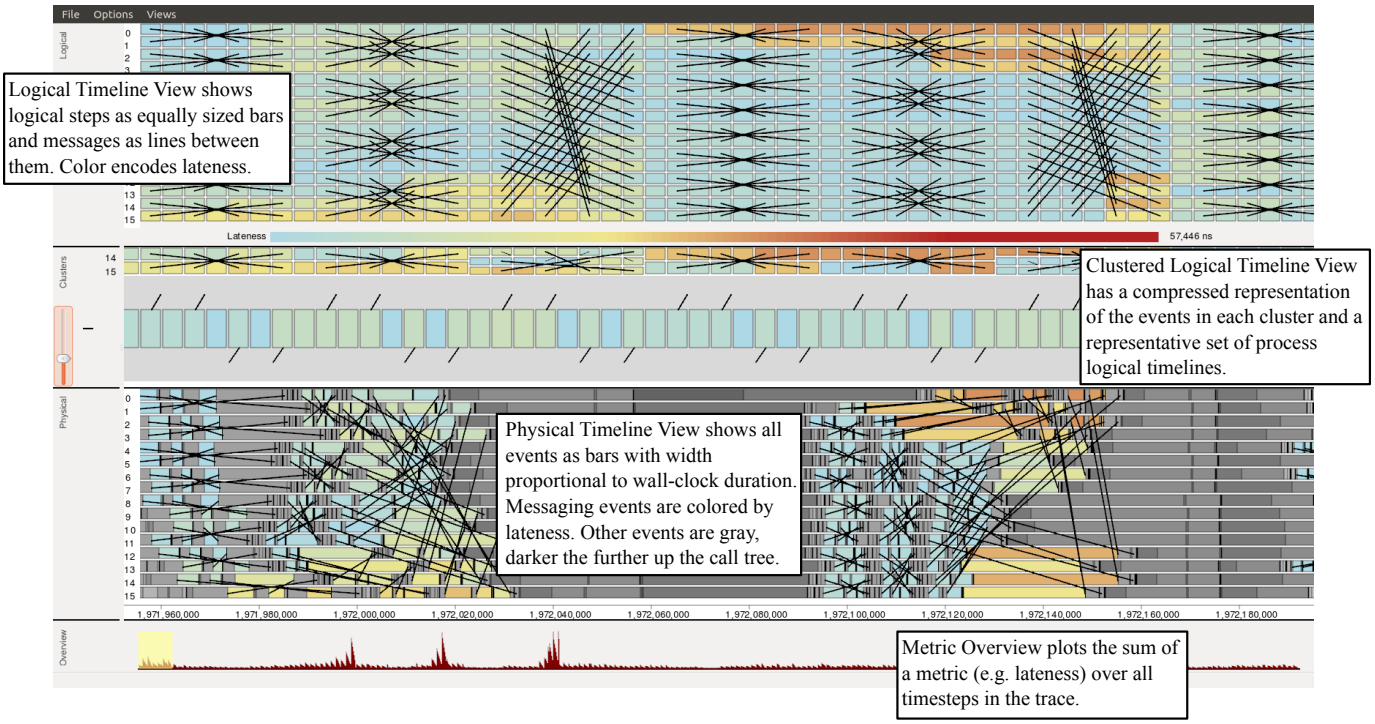


Fig. 5: Overview of Ravel, a tool for visualizing parallel execution traces.

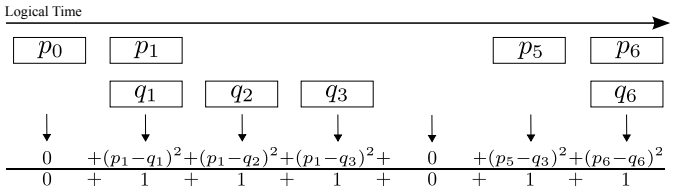


Fig. 4: Distance between two logical timelines with metrics p_i, q_i. When both processes exhibit an event, we use the squared difference. When only one process exhibits an event, we use the difference between the event and a prior event if it exists.

The distance metric for our clustering takes advantage of the discrete logical timesteps, considering the phase's span of steps as a vector of values. However, we do not calculate a pure Euclidean distance on this vector, as not every process will have an event at every timestep in a phase. We expect physical time-based metrics like lateness to be fairly stable until perturbed by another event, so we use previous values to fill in gaps where applicable. At any step where one process has an event and another does not, if the process without the event has a previous event in the phase, we use the metric value of that event, otherwise we skip the step. We also skip any steps where both processes are inactive. As we cluster on metric and not structure, we do not want to penalize processes for not having as many matching events. Therefore, we take an average of the squared distances between each step counted. A sample distance calculation is shown in Fig. 4.

In Ravel, we employ two-stage clustering. We first rapidly generate a moderate number of clusters using CLARA (Clustering for Large Applications) [23], a sampled k-medoids method, as implemented in Muster [15, 16], though other clustering algorithms could be used. In the second stage, we create a navigable cluster hierarchy of the CLARA clusters using single linkage hierarchical clustering [37], allowing users to explore different clusterings.

CLARA is $O(p)$ where p is the number of processes. The hierarchical clustering algorithm is $O(c^2)$ where c is the (small) number of CLARA clusters. Calculating per phase rather than over the length of the trace further decreases clustering costs. Not all phases may contain all processes and phase breaks mean not every event-less step will have a suitable previous event to use for a value. We cluster the entire trace using lateness during preprocessing. If the user changes the metric later, we recalculate the clustering for phases only as they come into view, spreading out and only incurring computation cost as needed. The user may also elect not to perform clustering. In this case the cluster view will not be shown.

3.3.2 Customization

We can improve the visualization by detecting patterns of interest and customizing their cluster representation. We have developed a modular interface to support patterns in Ravel. Each module defines a pattern, and we run pattern detection in all phases during preprocessing. On successful detection, we call the module's draw functions to visualize the cluster of the phase containing the detected pattern.

As an example, we have implemented a module to detect exchanges. Exchanges are a prevalent pattern, commonly referred to as *stencils* in scientific simulations and iterative solvers. An exchange occurs in phases where every process sends and receives from the same set of other processes, e.g., its neighbors within the application's domain decomposition. Each process may have a different neighborhood size, as long as the sends and receives match.

There are several common ways of performing an exchange. One is to alternate sends and receives. Another is to have all processes initiate all of their sends and then handle all of their receives. A third is to have all processes initiate their receives, perform their sends, and finally collect the incoming messages in an `MPI_WaitAll`. We detect all three of these flavors and develop a custom cluster visualization for each. These are described in Section 4.2.

4 Ravel

Our trace visualization tool, Ravel, shown in Fig. 5, is composed of four coordinated views. The logical time view, discussed in Sections 3.3.2 and 4.1, displays the logical timelines of all processes. This view is useful for moderate numbers of processes and overviews. The clustered logical time view, discussed in Sections 3.3 and 4.2, shows timeline clusters and representative timeline neighborhoods. This view is useful for large numbers of processes. The physical time view, discussed in Section 4.3, is similar to the main view in prevailing trace visualization tools. It should be familiar to experienced users.

Once an area of interest has been found in the other views, the physical time view provides detail in wall-clock time. Finally, the metric overview, discussed in Section 4.4, allows users to see the distribution of a metric like lateness over the entire length of the trace and navigate to time spans of interest. Ravel chooses which timeline windows will be opened at the start based on the number of processes in the trace and the clustering options.

The three timeline views, logical, clustered logical, and physical, have linked panning and zooming through time. The selected portion of the metric overview will also update to reflect these actions. The logical and physical timelines also have linked panning and zooming through processes, which are stacked on the vertical axis in both views. Individual MPI events may be selected in one timeline view and will be highlighted in all views in which they are visible. Hovering on events will bring up a tooltip with more information. Users have the option to omit the display of aggregated non-communication events in the logical and clustered logical views.

4.1 Logical Time View

In the logical timeline view, individual process timelines are stacked in rows, ordered by their ID in MPI. We preserve this order in all timeline views as it likely has meaning to the developers, e.g., it is often easily mapped to the application domain decomposition.

The horizontal axis represents logical time. Events are drawn as equally sized boxes as the time scale is logical and not physical. This drawing choice also emphasizes communication patterns. Messages are drawn as lines of equal width between their send and receive event.

When small numbers of processes or timesteps are shown, events are drawn with spacing and borders. As the number of processes or timesteps increases, the spacing and borders are omitted to conserve space. With large numbers of processes, message lines are omitted as well. They may also be turned off by the user. When the number of processes exceeds the pixel height of the view, overplotting techniques are used so that even sparse logical timestep patterns will be visible.

4.2 Clustered Logical Time View

The clustered logical time view is split into two sections. The clusters are shown in the bottom section, while timelines for a focus subset of processes are shown in the top section. The latter is useful to give viewers a hint as to the detailed pattern of communication, which the clusters are unable to show. Fig. 6 demonstrates this feature.

Each cluster is displayed as a timeline of events matching the logical time scale of the other views. The communication events are represented by a rectangular glyph (Fig. 7) subdivided vertically into sections. The top section shows sends while the bottom shows receives. Inactive processes are an uncolored section in the middle. The height of each section corresponds to the portion of the cluster performing that action at that timestep. For sparse data, the inactive section can be removed. To further emphasize the communication, a send line is drawn from the top and a receive line from the bottom. The thickness of this line is determined by the portion of the cluster that is sending or receiving. Each event is colored by the average metric of the events it

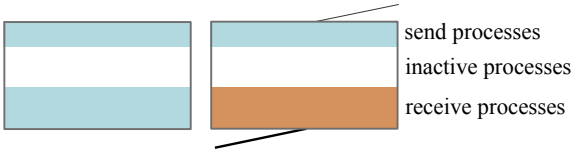


Fig. 7: Cluster event glyphs. The top bar and line are proportional to the number of sends and colored based on the average metric for sends. The bottom bar and line is the same for receives. The blank area is proportional to inactive processes. The left glyph represents the non-communication event preceding the right. In this example, receiving processes experience an increase in the metric value on average.

represents. Customized depictions for special communication patterns are discussed in Section 4.2.1.

A dendrogram of the clustering of the phase in focus is displayed in the left panel. Users can expand or contract the clustering dendrogram to explore clusters. The vertical space assigned to each cluster is proportional to the number of processes in that cluster. Alternating background colors demarcate cluster boundaries. Single-process clusters are drawn as they would be in the logical timeline view, with message lines drawn between the leaf timeline and the cluster containing the other end of the message.

The user can select a cluster of interest. Doing so will highlight all members of that cluster in the focus processes and in the other timeline views. Highlighting of processes is done by decreasing the opacity of all non-selected processes and events.

The focus process timelines are drawn similarly to the logical timeline view, except messages between a focus process and an undrawn process are drawn completely in the focus process's event, starting at the center and ending in the corner closest to where it would be drawn in the full view, to indicate whether it is a send or a receive and communicating between a process with a higher or lower ID.

Initially, the process containing the event with the greatest metric value and that process's messaging neighborhood are chosen as focus processes. The user can set radius of the messaging neighborhood can be set by the user or change the focus processes to the neighborhood of the maximum metric process or centroid process in a selected cluster.

4.2.1 Customized Cluster Drawing for Exchanges

In Section 3.3.2, we introduced a module that detects exchange communications and changes the cluster drawing if the exchange falls into one of three types. The general cluster drawing is suitable for the type where each process posts all its sends and then handles receives individually. For the case where all receives are collected into a single `MPI_Waitall`, we replace the drawing of the receive line with a quarter pie indicating how many receives are handled at that step on average compared to the maximum handled in that phase. We also quantify that average with a label. This is shown in Fig. 8.

In the case of a large volley of send-receive pairs, we verify that the majority of the messages have their receive event two steps after their send. Then we represent the cluster as a pair of timelines, exchanging

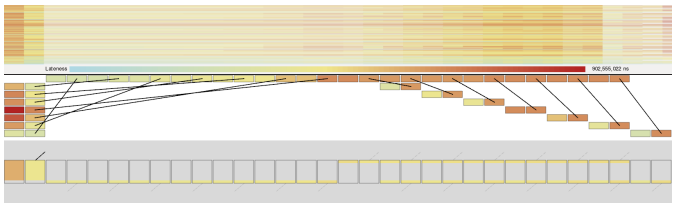


Fig. 6: Logical (top) and Cluster (middle/bottom) view of a merge tree gather, further discussed in Section 5.1. The bottom row shows the unopened root of the cluster hierarchy, summarizing the events in the cluster: many processes active in the first two steps, few in the rest. Above the cluster are individual timelines showing the communication neighborhood of one process of interest. The focus processes show structure which can no longer be seen in the logical view.

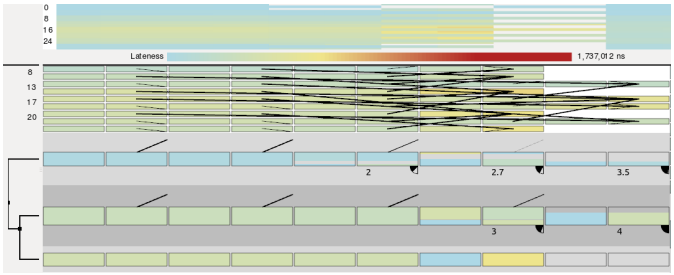


Fig. 8: Cluster view of an exchange in SMG2000 [34], a semicoarsening multigrid solver, with a customized cluster representation. The small pie charts below the communicating tasks show that the earlier `MPI_Waitall` calls collect fewer receives than the later ones.

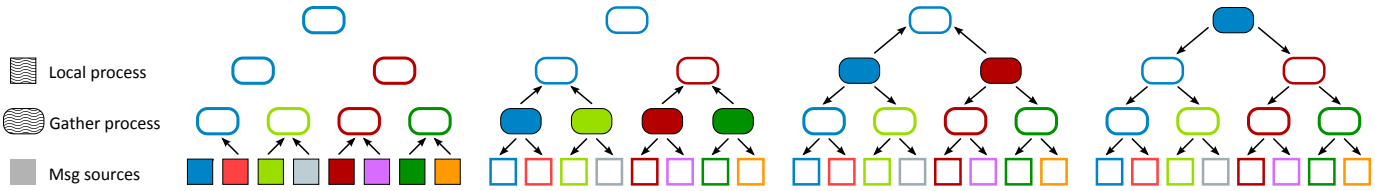


Fig. 9: Communication pattern of the parallel merge tree algorithm for eight processes using a binary gather. The data is distributed among the eight processes (identity indicated by the coloring) and the gather processes are overloaded onto a subset of processes. The solid shapes highlight the processes responsible for the communication in each round. At each round of communication data is sent upwards to the root and downward to the children in a point-to-point fashion.

messages back and forth. In some clusters, certain events in this portrayal may not be present in the data, e.g., a cluster of all first-senders will not have any representative first-receivers. In that case, we depict the unrepresented events as an empty box with a dashed border. The send-receive pattern is maintained in this drawing, but no non-present events are shown. This cluster representation is used in Section 5.2.

4.3 Physical Time View

The physical time view is the common visualization of traces, with process timelines stacked in rows and physical time on the horizontal axis. Events are represented as bars, their widths proportional to duration. Messages are shown as lines between two process timelines.

If recorded as part of the trace, this view shows details of non-communication, and possibly non-MPI, events. This can include function invocations made at any level, so some events contain others. A gray scale is applied to non-messaging events with calls made deeper in the tree colored lighter and drawn on top of their parent calls. If present, the `main` procedure will be darkest and overlaid with other events. Messaging events will always be at the leaves of their call tree. These are colored as in the other timeline views.

As the horizontal scale in this view is physical time rather than logical time, a mapping between steps to physical time must be made to keep it synchronized with the other views. During preprocessing, this view creates such a map, storing for each step the start and stop time. The start time is the least enter time of any event occurring at that step. The stop time is the greatest exit time of any events occurring at that step. When mapping between steps and physical time, we assume the largest span calculated. This causes irregular overlaps resulting in logical timelines scrolling non-smoothly when the physical timeline is being panned or zoomed and vice versa. However, this is necessary to assure that the other views contain the entirety of messaging information in the one being manipulated.

4.4 Metric Overview

The metric overview displays the sum of the active metric across all processes for all timesteps in the trace. The horizontal axis is logical time. The height of the bars are scaled by the step exhibiting the greatest sum. Users can select time spans on the metric overview to quickly navigate to any portion of the trace.

5 Case Studies

We examine two large scale parallel codes to show the utility of Ravel. In the first case study, we show how Ravel was used to discover a non-optimal message ordering in an *in situ* analysis application and then verify the improved implementation. We also demonstrate the cluster view's ability to show messaging patterns at multiple scales. In the second case study, we use Ravel to explore messaging delay patterns under two different run configurations of a parallel physics simulation code at three process counts and reveal a scaling bottleneck.

5.1 Massively Parallel Merge Trees

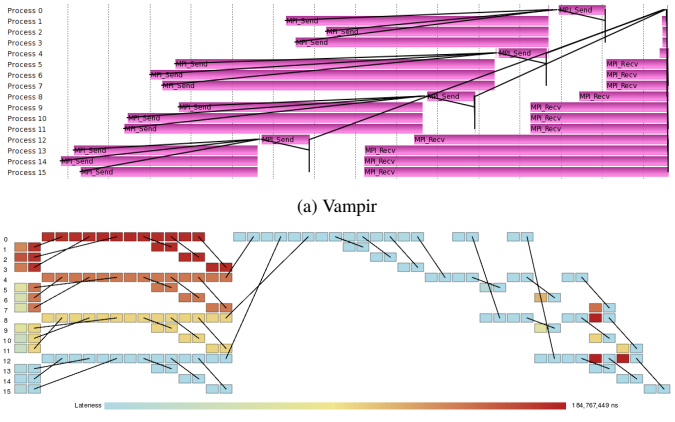
As the gap between available memory and the effective file I/O rates increases, many tasks that have traditionally been performed in post processing will have to be executed *in situ*. This is particularly problematic for many of the common data analysis and visualization algorithms, which are typically global in nature, file I/O or memory bandwidth bound, and often unstructured. Parallelizing such approaches, especially to tens or hundreds of thousands of cores utilized by today's simulations, is a significant challenge.

Here, we analyze a development version of a massively parallel algorithm [27] to compute *merge trees*, a topological structure that has recently been used to analyze some of the largest combustion simulations [8, 5, 4]. The algorithm relies on a global gather-scatter approach similar to a V-cycle in a traditional multi-grid solver with the corresponding problem of low parallel efficiency as most processes are kept idle some of the time. Further, as with many analysis problems, the computation is highly data dependent causing severe load imbalance in some cases.

The general data flow for a merge tree using a binary gather is shown in Fig. 9. Each process is assigned an equal portion of the data with the decomposition typically dictated by the corresponding simulation code. All processes then perform a local computation step and send the results to their designated gather processes. These integrate the information and send the results both upwards to the next level merge and back downwards towards the leaves which must integrate it. This process is repeated until all information has been globally integrated by the root of the gather. To avoid any one node sending an excessive number of messages, e.g., the root to all leaves, the scattering is routed backwards along the gather tree using multiple hops.

Since virtually all simulation codes distribute data among all available processes, the gather tree will contain as many leaves as there are processes in the simulation. The gather processes are therefore overloaded onto the processes in a straightforward modulo-k type fashion; e.g., in a binary gather the first level of gather will be assigned to processes $0, 2, 4, \ldots$, the level 2 gather to processes $0, 4, 8, \ldots$, and so on.

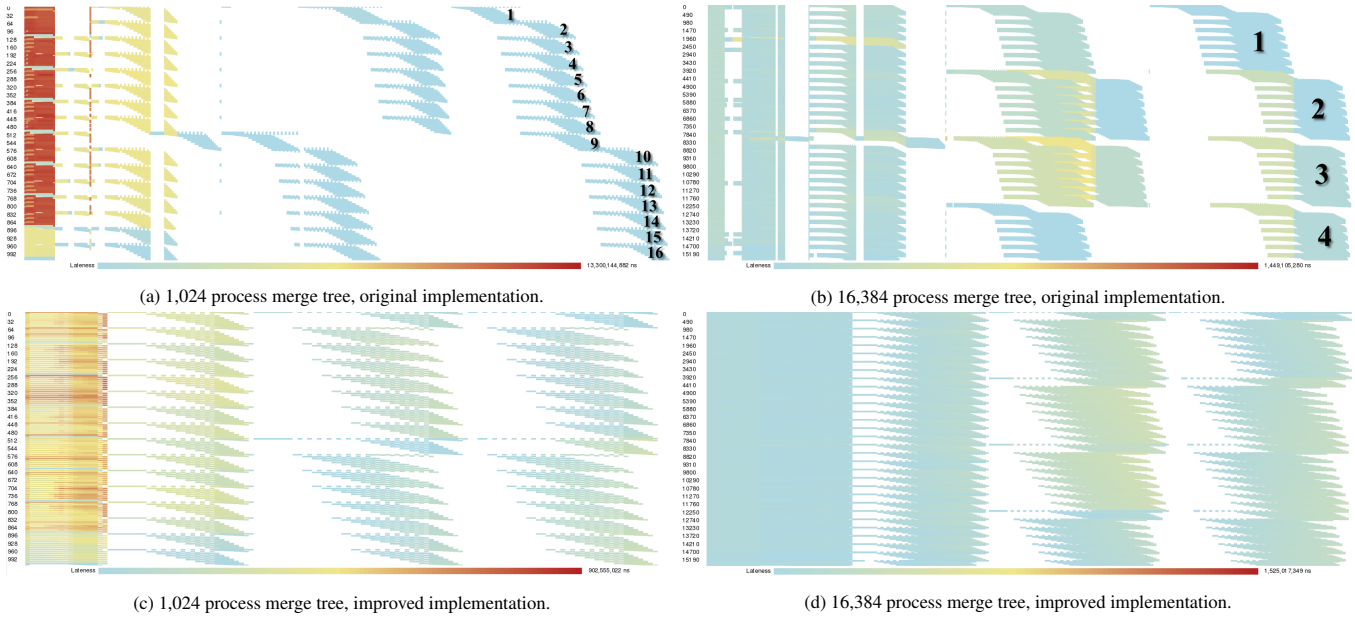
Fig. 10 shows the Vampir and Ravel visualizations of a complete 16 process, 4-ary merge tree. The logical step view is colored by lateness. In this case, the initial step is late, meaning the lateness is due to a



(a) Vampir

(b) Ravel

Fig. 10: Complete trace of a 16 process, 4-ary merge tree rooted at process 0. The Vampir view is dominated by the first round of receives waiting for the local compute; little other structure is readily discernible. In contrast, the Ravel logical time view directly reveals the communication structure expected from Fig. 9.



(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. 11: Ravel visualizations of an 8-ary merge tree calculation using both the original and improved implementations on 1,024 and 16,384 processes, respectively. The skewed parallelograms correspond to cascading sends towards the leaves. Branching in the gather three, numbered in (a) and (b), is clear. In the original implementation, (a) and (b), the parallelograms have a panhandle and there is a lot of white space, indicating most processes are often waiting. After this was discovered in Ravel, the developers improved the implementation to more optimally order messages. Traces of the new implementation, (c) and (d), show greater parallelism of the communication operations.

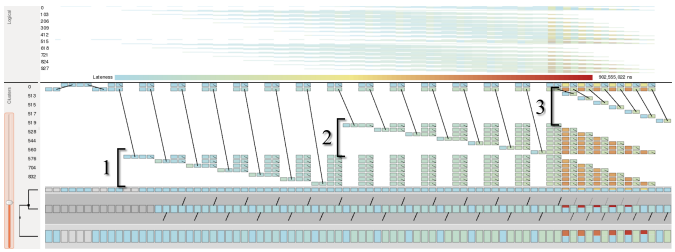
severe load imbalance caused by the input data characteristics. As process 0 is imbalanced, lateness is propagated to the corresponding gather as well as the resulting re-broadcast. Once the root of the tree is reached in the second gather stage lateness resets – there are no other events in the logical step with which to compare. The logical steps clearly highlight the gather tree structure of the algorithm and provide immediate insight into the overall behavior of the code. None of this information is easily accessible from the traditional trace visualization as the time-bound layout obscures the underlying structure.

Figs. 11a and 11b show two more realistic cases – a 1,024 and 16,384 process run using an 8-ary gather with the same data. There is more lateness in the 1,024 run than the 16,384 process run. As in the small 16 process example, the lateness is due to load imbalance caused by the data. The 1,024 process run divides the data into larger portions per process than the 16,384 process run. The larger data portions are able to exhibit more variance in computational requirements, which is why the 1,024 process run experiences more lateness.

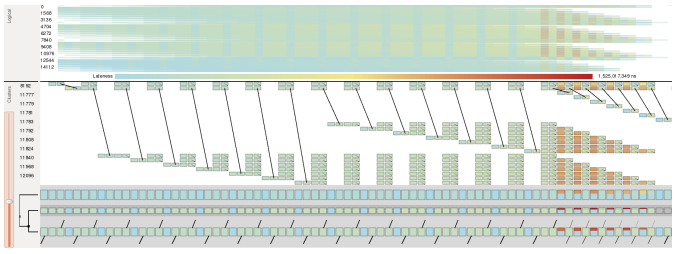
Since 1,024 and 16,384 are not powers of eight, the level closest to the root only contains two gather processes at 1,024 nodes (and thus 16 processes at the next) and 4 gather processes at 16,384 nodes. In Fig. 11a, we can see a division in connection between the top and the bottom half of the image, with each half containing eight parallelograms, 16 groups in total. Similarly, the right side of Fig. 11b contains four large groupings. However, the repeating motif of a parallelogram with a panhandle results in many timesteps where few processes are active. A closer analysis reveals a potential flaw in the algorithm. The panhandle occurs when a process sends to its leaves, those closest in rank space, before sending to its higher level children. Furthermore, this means the gather processes first send the information back to the leaves before sending it onward towards the root of the tree. This motif manifests at each level of the tree, the largest example being the events in the middle of the logical steps on the higher half of the MPI ranks (lower half of the visualization). The observed ordering misses an opportunity for a more aggressive pipelining of the computation. No process can finish until the root of the gather has been reached and thus the gather should be prioritized over the scatter.

Upon discovering the non-optimal message ordering in Ravel, the merge tree developers changed their implementation. Figs. 11c and 11d are traces of the improved application using the same input parameters, but showing significantly more overlap in the communication.

For the large examples, the logical view can no longer meaningfully represent the messages, which are not drawn. The focus processes in the cluster view can be used to get a sense of the communication pattern. Fig. 12 features the Ravel cluster view for one phase of the improved merge tree algorithm run at both 1,024 and 16,384 processes. Though the two traces differ in scale, the messaging pattern conveyed

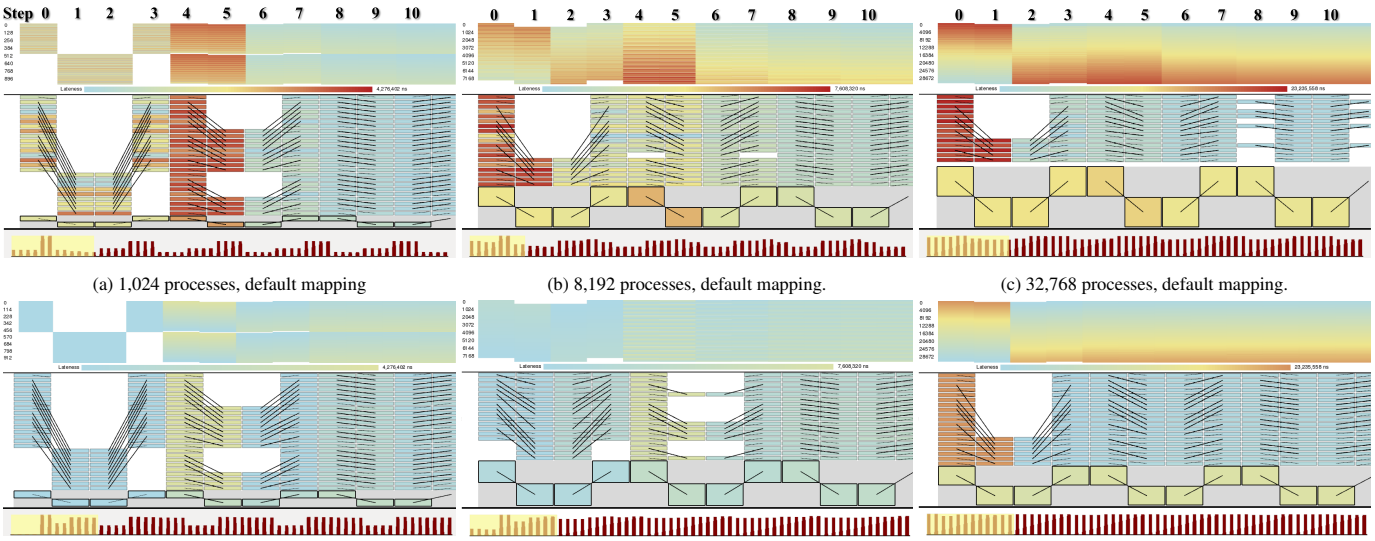


(a) 1,024 process merge tree, improved implementation



(b) 16,384 process merge tree, improved implementation

Fig. 12: One phase of the improved merge tree algorithm. Despite the difference in process count, the cluster view reveals the same message pattern. The focus processes exhibit the motif of one process sending to three different groups of eight processes, labeled in (a). As soon as the message is received by a child, it in turn begins sending, except in the final group. This indicates those are the gather tree leaves and thus the messages are ordered optimally.



(a) 1,024 processes, default mapping

(b) 8,192 processes, default mapping.

(c) 32,768 processes, default mapping.

(d) 1,024 processes, round-robin mapping

(e) 8,192 processes, round-robin mapping.

(f) 32,768 processes, round-robin mapping.

Fig. 13: Ravel visualizations of the pF3D near-neighbor exchange using both the default mapping and the round-robin mappings at 1,024, 8,192 and 32,768 processes. Timestep labels have been added at the top for clarity. The traces from the round-robin mapping runs exhibit lower lateness throughout. These visualizations also show a curious pattern in lateness, causing a gradient in the logical view. An examining of details in the cluster view shows this is not a drawing artifact. We hypothesize the gradients may be due to round trip behavior required by the sends which can be completed faster by processes that only receive in a send-receive timestep pair.

is equally visible in both views.

The focus processes show the optimized messaging pattern. The second process sends to three different groups of eight children, a group for each level of the gather tree for which that process is active. After the child receives the message, it in turns starts sending, except for the last group. This indicates the final group of eight which do not send represent the leaves, verifying the updated merge tree implementation handles the leaves last.

The coloring of the cluster visualization shows that the processes have been grouped by lateness in the aggregate non-communication steps late in the phase.

5.2 Laser-Plasma Interaction Simulations

pF3D [39] is a highly scalable parallel application used to study laser-plasma interactions in experiments at the National Ignition Facility at LLNL. This code is routinely run on high-end supercomputers, such as the IBM Blue Gene/Q and Cray XE6 at very high processor counts. The communication-heavy nature of this application often leads to performance problems at such scales.

One of the dominant communications in pF3D is a nearest-neighbor exchange where each process communicates with six nearest neighbors in a three-dimensional (3D) process grid. The default mapping of pF3D processes to cores on some architectures can make this exchange inefficient due to the relatively bad placement of communicating neighbors on the interconnect topology. For example, Blue Gene/Q (BG/Q) uses a five-dimensional (5D) torus interconnect as the network between processing nodes. Different mappings of the 3D logical process grid of pF3D onto the 5D torus of BG/Q can make a significant difference in how messages are routed on the network and in turn, performance [6]. The effect was previously studied on a 3D torus interconnect with the aid of a visualization specific to that interconnect topology [26]. As such visualizations are not available for all interconnects, including the BG/Q's 5D torus, we explore performance under different process mappings using tracing and trace visualization.

The default process mapping assigns process IDs to all 16 cores on a node before moving onto the next node. We can also apply a different mapping that assigns process IDs to one core on each node in a *round-robin* fashion. Runs of the pF3D communication benchmark using the round-robin mapping finished 1.25 to 2.5 times faster than those run with the default mapping. Fig. 13 shows trace visualizations of both mappings applied to 1,024, 8,192, and 32,768 process executions.

At each scale, the round-robin mapping exhibits much less lateness at all steps, which is expected given its better overall performance. The first two send-receive pairs, corresponding to timesteps 0 through 3, are an exchange in the z direction of the domain decomposition. Under the default mapping, this means all processes send off-node to the exact same neighbor node, contending for the same network link. As such, we expected to see lateness vary among groups of 16 processes in the default mapping and then this lateness to propagate along the processes. We also expected the round-robin mapping would not have as much lateness variance in groups of 16 during the z exchange, but what small variance it did have would help alleviate contention further along the exchange where conflicts could occur, leading to lower lateness overall. In the 1,024 process scenario, Ravel shows behavior that supports this theory during the z exchange: the first four steps of Fig. 13a show a lot of variance in lateness between neighboring processes, visible in both the logical and clustered view, and lateness propagating along message lines, visible in the focus processes of the clustered view. The first four steps of Fig. 13d show no lateness as expected. However, after the first four steps, we see unexpected gradient patterns along the processes under both mappings. These gradients are also present in the larger runs.

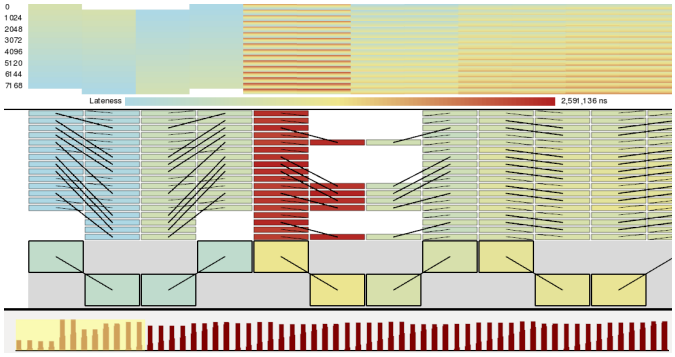
The unexpected gradients are most visible in Figs. 13b, 13c and 13f. Also, rather than lateness propagating along a process, it inverts between the first two and the second two timesteps in these same examples. When we adjust the color map of the 8,192 process round-robin trace (Fig. 14), we also find similar problems. We verified that these observations were not drawing artifacts by examining focus processes in the cluster view and zooming in on the logical view. These effects could not be seen in a standard physical time visualization (Fig. 15).

In the first two send-receive pairs (timesteps 0 - 3), lateness starts at the send, but is not entirely caused by contention from the mapping, because that does not explain the gradient. We hypothesize another contributor to this lateness. The MPI send call used can require the receiver to acknowledge the request. In the first two steps, only the last processes, those corresponding to the final xy plane in the domain decomposition, do not send. This is visible as the white space at the bottom of step 0 and the top of step 2. By not sending, those processes are able to more promptly respond to the request. Thus, their senders receive acknowledgement first and can complete their send. This ef-

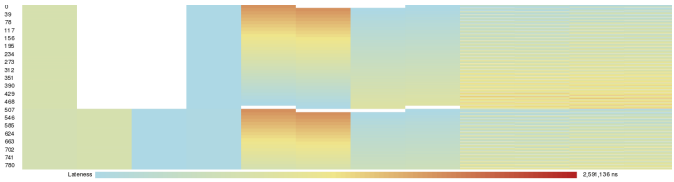
fect cascades along the process IDs, resulting in the observed gradient. Fig. 16 illustrates this effect. The second send-receive timestep pair has the same problem inverted. In the 1,024 process runs, no process both sends and receives in the first two timesteps, so no cascade effect occurs which explains why no gradient appears.

The send-receive timestep pairs in the y and x direction have similar boundary conditions, but are tiled in the process ID space. This leads to the striping patterns seen in some of the later steps. However, if the effect of the earlier delays is great enough, the effects of the later steps may not be significant enough to show up in this view.

The developers were previously unaware of this cascading dependency problem. They were able to fix the problem leading to a significant performance benefit. Fig. 17 shows the difference in time spent in communication for the full (non-benchmark) application before and after the change with all other parameters fixed.

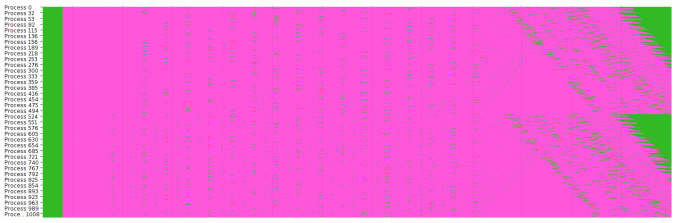


(a) 8,192 processes, round-robin mapping.

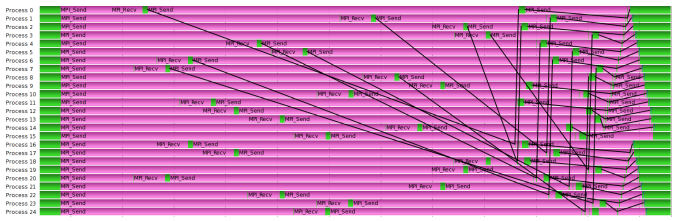


(b) Zoom to show gradient in later steps.

Fig. 14: Trace for an 8,192 process, round-robin mapping pF3D benchmark run with the lateness range relative to maximum lateness in the step. The same striping patterns seen in the default mapping trace in Fig. 13b are visible here as well.



(a) Vampir visualization of all 1,024 processes.



(b) Vampir visualization of subset of 1,024 processes.

Fig. 15: Vampir visualizations of the pF3D exchange as shown in Fig. 13. Even with relatively few processes, it is difficult to understand the communication structure and behavior.

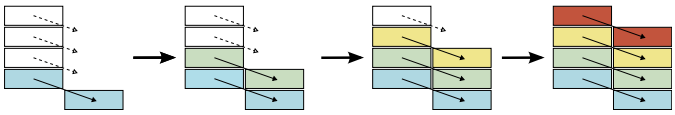


Fig. 16: Sends cannot complete until the matching receive is called. Therefore they must wait until the receiver finishes its own send. This causes a cascading effect which shows up as gradients in Ravel.

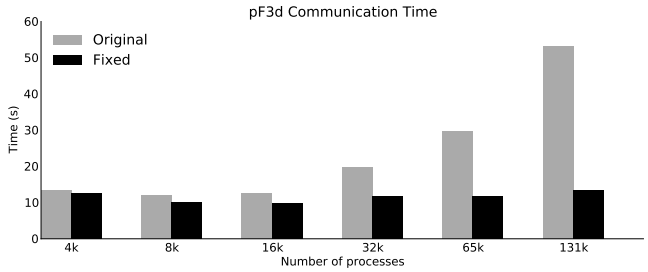


Fig. 17: Time pF3D spent in communication using the original and fixed implementation. Removing the dependency issue found using Ravel resulted in a significant reduction in time.

6 Conclusion

We have presented a scalable, informative visualization for parallel execution traces that utilizes the concept of logical time. By shifting the data from physical to logical time, we are able to maintain and clarify communication (messaging) and dependency relationships between processes and illuminate messaging patterns among processes, meeting our first design goal (G1). By applying metrics derived from physical time, we depict sources, patterns, and evolution of delays in an execution.

Another benefit of logical timelines is the way they lend themselves to clustering, allowing us to summarize behavior at even larger scales. We created a representation for clusters that includes messaging information along with structural cues from focus process neighborhoods to meet our second design goal (G2), efficacy at scale, without abandoning the first.

We combined our logical timelines and clusters with a traditional physical time visualization and an overview, coordinated across a single logical time scale, giving users a range of detail from the coarse full timeline overview to the highly detailed physical timeline, with the logical timelines and clusters existing in the middle. By coordinating these views, we achieve our final design goal (G3) of providing exploration at several levels of abstraction. We demonstrated the effectiveness of this visualization through two case studies, discovering and explaining performance issues in a parallel analysis application and a large-scale scientific simulation.

Acknowledgments

The authors would like to thank Aaditya Landge for his help with the parallel merge tree application, Nikhil Jain and Steven H. Langer for their help with pF3D, and Steven Mueller for his multimedia technical expertise.

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-JRNL-657418. This research is supported in part by the Department of Energy Office of Science Graduate Fellowship Program, administered by ORISE-ORAU under contract no. DE-AC05-06OR23100.

References

[1] W. Aigner, S. Miksch, H. Schumann, and C. Tominski. *Visualization of Time-Oriented Data*. Human-Computer Interaction Series. Springer, 2011.

[2] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, T. Mezzacappa, P. Moin, M. Norman, R. Rosner, V. Sarkar, A. Siegal, F. Streitz, A. White, and M. Wright. ASCAC subcommittee report: The opportunities and challenges of exascale computing. Technical report, United States Department of Energy, Fall 2010.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, and R. A. Fatoohi. The NAS parallel benchmarks. *Intl. J. of Supercomputer Applications*, 5(3):63–73, 1991.

[4] 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 Conf. on Supercomputing (SC12)*, Nov. 2012.

[5] 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. on Vis. and Comp. Graphics*, 17(12):1822–1831, 2011.

[6] A. Bhatele, T. Gamblin, S. H. Langer, P.-T. Bremer, E. W. Draeger, B. Hamann, K. E. Isaacs, A. G. Landge, J. A. Levine, V. Pascucci, M. Schulz, and C. H. Still. Mapping applications with collectives over sub-communicators on torus networks. In *Proc. ACM/IEEE Conf. on Supercomputing (SC12)*, SC '12, Nov. 2012.

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

[8] 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 Vis. and Comp. Graphics*, 17(9):1307–1324, 2011.

[9] H. Brunst, D. Hackenberg, G. Juckeland, and H. Rohling. Comprehensive performance tracking with Vampir 7. In M. S. Mller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 17–29. Springer Berlin Heidelberg, 2010.

[10] W. Cleveland and R. McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *J. American Statistical Association*, 79:531 – 554, 1984.

[11] J. Cuny, G. Forman, A. Hough, J. Kundu, C. Lin, L. Snyder, and D. Stemple. The Ariadne debugger: Scalable application of event-based abstraction. In *Proc. 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, PADD '93, pages 85–95, 1993.

[12] J. Daly, J. Hollingsworth, P. Hovland, C. Janssen, O. Marques, J. Mellor-Crummey, B. Miller, D. Quinlan, P. Roth, M. Schulz, D. Skinner, and J. Vetter. Tools for exascale computing: Challenges and strategies. Technical report, United States Department of Energy, Oct. 2011.

[13] W. De Pauw and S. Heisig. Zinsight: A visual and analytic environment for exploring large event traces. In *Proc. 5th Intl. Symp. on Software Vis.*, SOFTVIS '10, pages 143–152, 2010.

[14] N. Elmqvist and P. Tsigas. Growing Squares: Animated visualization of causal relations. In *Proc. 2003 ACM Symp. on Software Vis.*, SOFTVIS '03, pages 17–ff, 2003.

[15] T. Gamblin. Muster: Massively scalable clustering. https://github.com/scalability-llnl/muster, accessed Mar. 2014.

[16] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed. Clustering performance data efficiently at massive scales. In *Proc. ACM Intl. Conf. on Supercomputing (SC10)*, pages 243–252, 2010.

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

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

[19] A. Hoisie, D. Kerbyson, R. Lucas, A. Rodrigues, J. Shalf, and J. Vetter. ASCR modeling and simulation of exascale systems and applications workshop. Technical report, United States Department of Energy, Aug. 2012.

[20] K. E. Isaacs, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. Ordering traces logically to identify lateness in parallel programs. Technical Report LLNL-TR-656141, Lawrence Livermore National Laboratory, June 2014.

[21] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. State of the art of performance visualization. In *Eurographics/IEEE Conf. on Vis. State-of-the-Art Reports*, EuroVis '14, 2014.

[22] L. V. Kalé, S. Kumar, G. Zheng, and C. W. Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, Intl. Conf. on Computational Science (ICCS)*, pages 23–32, June 2003.

[23] L. Kaufman and P. J. Rousseeuw. *Clustering Large Applications (Program CLARA)*, pages 126–163. John Wiley & Sons, Inc., 2008.

[24] N. W. Kim, S. K. Card, and J. Heer. Tracing genealogical data with TimeNets. In *Proc. Intl. Conf. on Advanced Visual Interfaces*, AVI '10, pages 241–248, 2010.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[26] A. G. Landge, J. A. Levine, K. E. Isaacs, A. Bhatele, T. Gamblin, M. Schulz, S. H. Langer, P.-T. Bremer, and V. Pascucci. Visualizing network traffic to understand the performance of massively parallel simulations. *IEEE Trans. on Vis. and Comp. Graphics*, 18(12):2467–2476, 2012.

[27] 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. *To appear in Proc. ACM/IEEE Conf. on Supercomputing (SC14)*, SC'14. Nov. 2014.

[28] M. Monroe, R. Lan, H. Lee, C. Plaisant, and B. Shneiderman. Temporal event sequence simplification. *IEEE Trans. on Vis. and Comp. Graphics*, 19(12):2227–2236, 2013.

[29] C. Muelder, F. Gygi, and K.-L. Ma. Visual analysis of inter-process communication for large-scale parallel computing. *IEEE Trans. on Vis. and Comp. Graphics*, 15(6):1129–1136, 2009.

[30] R. Munroe. XKCD #657. http://xkcd.com/657/, Dec. 2009.

[31] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[32] M. Ogawa and K.-L. Ma. Software evolution storylines. In *Proc. 5th Intl. Symp. on Software Vis.*, SOFTVIS '10, pages 35–42, 2010.

[33] V. Pillet, J. Labarta, T. Cortes, and S. Girona. Paraver: A tool to visualize and analyze parallel code. Technical Report UPC-CEPBA 95-3, Polytechnic University of Catalonia, 1995.

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

[35] Message Passing Interface Forum. MPI: A message-passing interface standard, 2012.

[36] M. Schulz. Extracting critical path graphs from MPI applications. In *Cluster Computing*, pages 1–10. IEEE International, Sept. 2005.

[37] R. Sibson. SLINK: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

[38] C. Sigovan, C. W. Muelder, and K.-L. Ma. Visualizing large-scale parallel communication traces using a particle animation technique. *Computer Graphics Forum*, 32(3pt2):141–150, 2013.

[39] 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*, 7(5):2023, 2000.

[40] B. Topol, J. T. Stasko, and V. Sunderam. PVaniM: a tool for visualization in network computing environments. *Concurrency: Practice and Experience*, 10(14):1197–1222, 1998.

[41] J. Trümper, J. Bohnet, and J. Döllner. Understanding complex multithreaded software systems by using trace visualization. In *Proc. 5th Intl. Symp. on Software Vis.*, SOFTVIS '10, pages 133–142, 2010.

[42] F. B. Viégas, M. Wattenberg, and K. Dave. Studying cooperation and conflict between authors with history flow visualizations. In *Proc. SIGCHI Conf. on Human Factors in Computing Systems*, CHI '04, pages 575–582, 2004.

[43] 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*, pages 157–167. Springer Berlin Heidelberg, 2008.

[44] K. Wongsuphasawat, J. A. Guerra Gómez, C. Plaisant, T. D. Wang, M. Taieb-Maimon, and B. Shneiderman. LifeFlow: Visualizing an overview of event sequences. In *Proc. SIGCHI Conf. on Human Factors in Computing Systems*, CHI '11, pages 1747–1756, 2011.