Eurographics Conference on Visualization (EuroVis) 2013
B. Preim, P. Rheingans, and H. Theisel
(Guest Editors)

Volume 32 (2013), Number 3

Visualizing Large-scale Parallel Communication Traces Using a Particle Animation Technique

Carmen Sigovan[1], Chris W. Muelder[1], and Kwan-Liu Ma[1]

[1]University of California, Davis

Abstract

Large-scale scientific simulations require execution on parallel computing systems in order to yield useful results in a reasonable time frame. But parallel execution adds communication overhead. The impact that this overhead has on performance may be difficult to gauge, as parallel application behaviors are typically harder to understand than the sequential types. We introduce an animation-based interactive visualization technique for the analysis of communication patterns occurring in parallel application execution. Our method has the advantages of illustrating the dynamic communication patterns in the system as well as a static image of MPI (Message Passing Interface) utilization history. We also devise a data streaming mechanism that allows for the exploration of very large data sets. We demonstrate the effectiveness of our approach scaling up to 16 thousand processes using a series of trace data sets of ScaLAPACK matrix operations functions.

Categories and Subject Descriptors (according to ACM CCS): Multimedia Information Systems [H.5.1]: Animations—;

1. Introduction

High performance computing (HPC) systems are one of today's most valuable resources available to the scientific community. Parallel numerical simulations are the method of choice for scientists who are unable to conduct their experiments in a laboratory environment or in the field, for example supernova studies or global climate predictions. But parallel computation introduces the need for communication between processes, and the subsequent overhead has a significant impact on the performance of parallel applications. Furthermore, as the number of processors used in a parallel computation increases, it becomes more difficult to evaluate the communication overhead and to determine if data distribution and communication requirements are optimized for that particular computation.

Most large-scale scientific codes use the standard Message Passing Interface (MPI) to transfer data between compute nodes. Their implementations, however, rarely make direct calls to MPI functions and instead utilize intermediary libraries, such as ScaLAPACK [Sca] linear algebra package. These intermediary libraries obscure the message passing activity from the user, making it difficult to understand or evaluate communication patterns.

Visualization is a commonly used method for analyzing the communication patterns of parallel programs. Many tools developed for this purpose display individual process activity using Gantt charts or Kiviat diagrams. While effective at small to medium scales, such approaches are cumbersome with data sets containing many thousands of processes or more. Other proposed methods have bypassed this scalability problem by concentrating on event patterns and disregarding per-process representations altogether [MGM09].

We present a visual analysis method for parallel communication traces, by which we aim to address some of these issues. First, we collect the communication patterns of some representative numerical calculations by tracing ScaLAPACK parallel matrix operations at the MPI level using the MPE (MPI Parallel Environment) library. We then convert these traces into a database format which is suitable for out-of-core visualization. We then stream data into an animation that renders individual communication events while also grouping processes, such that system-wide trends become visible even when rendering large traces with numer-

© 2013 The Author(s)
Computer Graphics Forum © 2013 The Eurographics Association and Blackwell Publishing Ltd. Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

ous processes. Being user-controlled, the animation provides both snapshots of the communication state at any given time and provides an overview of the entire execution over time. A history of past events is accumulated as a background texture, and if the trace is streamed in its entirety, becomes a static visualization of the entire execution.

The presence of the time dimension is a salient feature of parallel execution and communication traces. Mapping execution time to real time is natural and intuitive, which is why we chose to implement an animation technique for our visualization of MPI. By displaying concurrent function calls as particles moving on trajectories parallel to each other, we aim to support the analyst's mental model of parallelism and make it easier to detect potential communication slowdowns, when particles that should be aligned fall out of synch.

We demonstrate our method with case studies of MPI activity traces on two separate supercomputers. We have designed this system with the aim of helping parallel library and application developers easily understand the communication patterns of their software, identify performance issues, and subsequently optimize their parallel codes.

2. Related Work

Parallel execution trace analysis has nescessitated the development of numerous analytic tools. Some of these tools use numerical methods to detect performance issues within trace data. For instance KOJAK [MW, WM00, WM03] (Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks), and its successor, Scalasca [GWW*10] perform automated analysis on parallel execution traces and can detect performance problems, such as detrimental wait states, even in very large traces. But numerical analysis is not always enough. Visualization is often employed to analyze parallel execution traces when the data is too large or complex for pure numerical analysis methods to suffice. Additionally, visual representations are often easier to understand than the results of numerical analyses.

Many performance visualizations [ZLGS99, SM06, NAW*96] use techniques that show either the timings and durations of parallel events or application executions, or represent system resource utilization. This includes well-known representations such as Gantt charts, histograms and Kiviat diagrams. Other visualizations [SML*12, LLB*12] focus on the communication topology, and hence use network representations such as node-link diagrams or matrices. Since communication has a direct impact on performance in parallel environments, many visualization toolkits for parallel traces combine both types of displays. And since the data is inherently temporal, many toolkits utilize some form of animation to convey activity over time.

ParaGraph [Hea93, HE91], features a multitude of both performance/utilization and communication displays. ParaGraph also incorporates an animation display, in which the system is represented by a graph, with the nodes corresponding to processors and indicating their status, and edges depicting communication between processors.

Jumpshot [ZLGS99] (and its predecessors, Upshot [HL91] and Nupshot [KL94]) are MPI communication analysis visualizations based primarily on Gantt charts, where each row shows which MPI function calls are being executed by each process, while arrows between the chart's segments indicate the processes involved in each communication. The Jumpshot toolkits are also responsible for creating the MPE tracing libraries and some scalable binary trace formats, such as CLOG2 and SLOG2 [CGL08].

VAMPIR (Visualization and Analysis of MPI Resources) [NAW*96] is an integrated tracing and analysis suite that combines overviews of aggregate system activity or processor utilization with more in-detail Gantt chart based execution representations and communication pattern displays. VAMPIR's visualization application is also capable of displaying snapshots of system state at any given time, state changes (stepping) and animations of program activity throughout an entire run.

TAU (Tuning and Analysis Utilities) [SM06] is a comprehensive set of analysis tools for parallel programs, with both tracing and visualization capabilities. TAU's tracing tools can convert data into many different formats used by other existing tools, while its visualization utilities include a Gantt chart, communication matrix, and call graph views.

Virtue [SRWS99] is unique among the cited parallel performance visualization tools in that it can monitor execution in real time and supports use in a CAVE Automatic Virtual Environment, as well as collaborative visualization through annotations. Virtue is graphically elaborate, featuring 3D displays such as a wide-area geographic view, a time tunnel display, and an interactive call graph with a specialized lens tools for exploration. Another 3D approach to visualizing parallel communication data is the work of Schnorr et al. [SHN], which shows the topology and communication of geographically distributed Grid applications.

While Gantt charts and node-link diagrams are natural representations for this kind of data, they run into severe scalability issues as the number of processes or parallel function calls increases, because the number of elements to display can easily exceed the number of available pixels on the screen. The works of Muelder et al. [MGM09, MSM*11] address this problem by arranging events to form larger patterns and using overplot techniques such as high-precision alpha blending and opacity scaling techniques similar to those proposed by Johansson et al [JLJC05].

In our approach, we aim for an intermediate point between depicting activity per individual process and showing patterns while ignoring process IDs. Our visualization displays every event in a trace data set individually over time. We also link events to the process which generated them. By aggregating individual processes into process groups when necessary, we maintain an overview of system activity while remaining in the bounds of allotted screen space.

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

2.1. The Case for Animation

Some studies have suggested that animation is not well-suited to certain tasks or that in some cases it can even be detrimental. Tversky et al. [TMB02] question the validity of several studies that had flawed experiment methodology. A study by Robertson et. al [RFF*08] indicates that animation is particularly ill-suited to trend visualization and has found that participants interpreted trend data with higher accuracy when they were shown static depictions of the data rather than animations. But there are other tasks for which animation has been shown to be useful.

One of the most prevalent uses of animated visualizations is to describe the functioning of algorithms. A study by Kehoe et. al. [KST01] shows that animation helped students better understand algorithms and made this difficult task more approachable. We believe that animation methods can be successfully employed to illustrate changes in a system over time. For example, the code_swarm visualization [OM09] was well-received due to its appealing visual design and its capability of illustrating the evolution of the complex interactions between developers working on large software projects. Code_swarm also visualizes time-varying activity, with actors operating in parallel. However, the somewhat unpredictable particle trajectories exhibited by code_swarm were not well-suited to our analysis tasks. A further particle animation method was proposed by Choudhury and Rosen [CR11]. They particles to visualize the flow of data through cache and main memory as programs execute. This method differs from ours in that it is meant for use on a single machine and is not designed to scale up and show communication between multiple processes.

When applied to parallel system performance analysis or parallel software visualizations, animation is often applied to existing visualizations, such as Gantt charts or communication matrices, to show changes in the system during execution. VAMPIR [NAW*96] and ParaGraph's [Hea93] time stepping animations function in this way. Kraemer and Stasko proposed an animation-based visualization methodology for the analysis of parallel algorithms and software [KS94, SK93]. They devised a system for user-defined visualization of parallel programs which included support for application-specific visualizations and for redefining the temporal ordering of events. In order to support this temporal reordering option, they also developed a dual timestamping methodology for tracing events in the parallel system [TSS95]. PVaniM also employed some animated traditional views, such as a Gantt chart and a communication view in the form of a temporally evolving node-link diagram. While these methods were successful at the time, their representations of individual processes no longer seem appropriate for today's parallel system scales, and development of PVaniM appears to have ceased. We have designed our visualization method with the intention of scaling up to today's HPC systems, where applications are likely to utilize thousands or tens of thousands of nodes. We also chose not to animate previously static performance or communication views, but instead developed our technique with the goal of showing the evolution of communication patterns in a parallel application execution.

Complex motion is detrimental to animation techniques, as it tends to confuse the viewer [RFF*08]. However, we employ 1D, predictable motion along the vertical axis for the particles in our visualization, making the animation easy to follow. Our method is not designed to show trends or rates of change, but to support the tasks of comprehending the complex communication required by large parallel applications and identifying slowdowns due to communication overhead.

3. Methodology

Our work incorporates two major components: collection and processing of data and visualization of the resulting trace. This section presents the design aspects behind each of these components.

3.1. Data Collection and Processing

This study is based on data collected from two of the world's largest supercomputers: Franklin and Kraken [Fra, Kra].

Many scientific computations utilize the ScaLAPACK library. Thus, the case studies we have performed are executions of various ScaLAPACK matrix operations, where the underlying MPI calls were traced using the MPE library. MPE records each instance of MPI functions being called, along with start and end times of the function call and the process that initiated it. The resulting trace file is written in the Jumpshot CLOG2 format. CLOG2 files store start and stop times of MPI events as separate entries, which we match up when we convert the data to our database format. We use a database because it enables us to stream events more easily than the CLOG2 format and because it allows random access and searching for data values.

Here we define some of the terms by which we refer to our data in the following sections.

- A **trace** is the data output by MPE at the end of a parallel application run. It records all MPI function calls along with their timing and process ID.
- An **event** is an instance of an MPI function call. The event's **type** is the actual MPI function called (such as *MPI_Send* or *MPI_Broadcast*).
- A **process** is an initiator of MPI function calls. Processes are identified in the trace data by their unique MPI ranks.
- **Time** in our data is the actual execution time of the parallel application being traced, although slowed down to a more reasonable speed for animation.

3.2. Visualization Approach

When the goal is to represent the data at a high level of detail in the limited screen space available, one method is to render each event as a point. However, even mid-sized traces often contain far too many events to simultaneously display on-screen without causing clutter and losing the ability to show patterns and meaningful information. We advocate an animation approach because it allows us to render data sets

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

in their entirety, while reducing clutter. We achieve this by displaying only currently active data at any given time.

3.2.1. Interface

Our system has a simple and intuitive interface which consists of an animation window, a timeline window, and a color legend for the MPI event types selected to study. Since there are a finite number of event types (MPI function calls), we have created a color mapping that attempts to assign unique hues while minimizing repetition. While the entire MPI framework consists of over 100 different functions, we found that most traces use a much smaller subset of these. Studies have shown that the upper limit on the number of colors readily distinguishable is around 25 [GA10], so we also employ a strategy to assign similar colors to operations which are similar in nature or which can be easily associated, such as the *MPI_Comm* set of functions or the blocking and non-blocking variants of MPI *Send* and *Receive*.

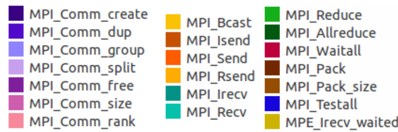


Figure 1: *Color mapping of the MPI function calls found in the traces. Some traces only contain a subset of these.*

The time line display is a stacked graph that indicates how many events of each type start at a given time and serves as a navigation interface for the animation. This display is very similar to the navigation timelines of [MGM09] and [MSM*11]. A vertical bar indicates the current temporal position in the data set, which can be set by dragging the bar across the time line. The main window also provides an interface for the user to control the speed of the animation, which is described in detail in Section 3.2.2.

The **animation window**, shown in Figure 2, displays the events in a data set over time as an “animated scatterplot”. At the bottom of this display is a sequence of segments of alternating shades of grey. Each segment corresponds to a set of processes, thus the particles generated above each segment correspond to events recorded on the segment’s processes. Processes are aggregated into these segments due to their potentially high numbers and to give local context to the particles. Processes within each segment are allocated equal space on the x-axis, in order, and particles are generated with randomized x-positions within the space allotted to their process ID. This ensures that particles will be sorted by their associated process ID, but alleviates overlap and aliasing issues. With very large numbers of processes, the space per process will be very small, unavoidably leading to large particle overlap. But this occurs primarily when the system is operating in unison, which the resulting plot still conveys. The number of processes per segment is calculated based on the total width available to the animation window and on a specified segment minimum length. The label under each segment indicates the ID of the first process in the set and since these IDs are in sequential order, it is very easy to deduce how many processes correspond to each segment. In our particular traces, the processes are naturally divided into groups, according to the process block size defined for each run. We use the term “process grid" to describe the MPI virtual topology used by the matrix operations under study, which is the Cartesian (grid) topology. Since Franklin has 8 cores per node, we found it best to use grids sized by powers of 2, and define the groups accordingly. Similarly, Kraken has 12 cores per node, so we used groups of multiples of 6.

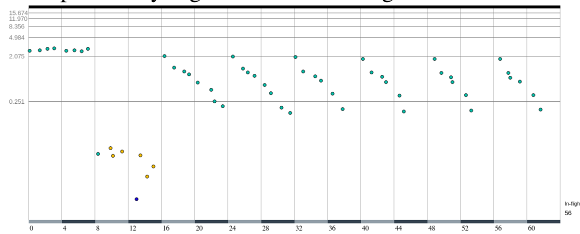


Figure 2: *A snapshot of the animation window. The particles are events in execution at a selected point in time. Processes 8-16 are mostly executing calls to MPI_Bcast, while the rest of the processes are waiting on long MPI_Recv operations.*

The background of the animation window employs a grid, with the horizontal lines delimiting event durations on a logarithmic scale and the vertical lines serving to separate the segments, which makes clarifies which segment a particle belongs to. The horizontal lines are placed on a logarithmic scale and each line has a corresponding label displaying its associated duration in milliseconds. We use a logarithmic scale since the duration of events can span many orders of magnitude, and previous work [MGM09, MSM*11] has shown it to work well for this class of parallel trace data, as the vast majority of these events are relatively small compared to the rarer long duration events. The lines are spaced such that the smallest distance between them is enough to allow for a label to be displayed with no overlap with its neighbors and the larger distances from top to bottom illustrate the logarithmic duration scale. The result of this is that the large number of short events are allotted a large amount of space, and the longer events persist at the top of the plot.

At any given time, a particle’s y-position on the grid corresponds to the amount of time that particular event has been in execution (i.e. the event’s age). The top horizontal line acts as a ceiling for duration within the data set, that is, it corresponds to the maximum event duration found in the data. Since we use a logarithmic scale for duration, if we designate d_{max} to be the maximum event duration found in the data, then an event particle’s position y_p at the current datatime t_c is, in pixels

$$y_p = y_{max} \cdot \ln(t_c - t_{start}) / \ln(d_{max})$$

where y_{max} is the distance in pixels between the bottom and top bold lines and t_{start} is the start time of the event.

3.2.2. Animation

So far, we have only described the static appearance of the animation window. But in the course of a data set explo-

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

ration, the contents of this window are rarely static. Here, we present the design behind the animation dynamics and explain the functionality behind the timing control variables available for editing on the main control window.

To generate animation, we use a sliding time window on the data set, which moves at a constant speed along the time axis. The *data time step* variable defines the width of the time window, while the *animation interval* controls the real time intervals at which the sliding window changes position. Both variables may be adjusted by the user.

In the animation window, particles travel upwards such that their position on the y axis always corresponds to the amount of time their associated events have been in-flight. When a particle reaches the height corresponding to its event's duration, the particle stops and fades away, as illustrated in Figure 3. All particles start at the level of the process segments (on the bottom) and, since the y axis is logarithmic, their speed decreases significantly as the event executes. This means that the longest events will be on-screen for the longest time, which will draw the observer's attention.We place this emphasis on longer communication events because they are more likely to be indicative of performance issues. Conversely, short events will reach their maximum height very quickly. Or, if the data time step is higher than their event duration, they would be skipped altogether. To counteract this problem, particles fade for a constant time of two seconds. This makes even very short events noticeable, which would otherwise vanish immediately.

When the animation is paused, the user may hover over particles to receive information about the MPI function call, its exact start and end times and the process (rank) executing it. This feature, also shown in Figure 3, is useful in distinguishing particles with similar colors or for determining which process is responsible for abnormally large events.

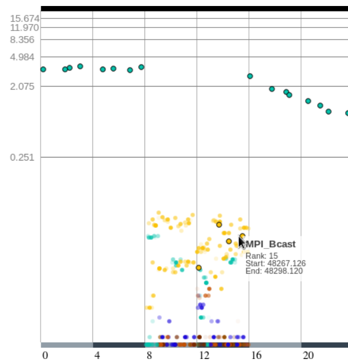


Figure 3: *A section of the animation window showing both running and ended events. The transparency of a fading particle indicates the time passed since the end of the event.*

When a particle's fading time expires, it will not completely disappear from the display. Instead, completely faded particles are rendered with very low opacity to off-screen density and color textures, which are then applied to the background of the animation window. Because large amounts of over-plot are likely to happen in this background buffer, the opacity is tone mapped on a logarithmic scale, which both keeps single particles visible and prevents areas with high over-plot from becoming too opaque and thus obscuring currently running event particles. A background pixel $P_{x,y}$ is assigned color and opacity based on the formula

$$P_{x,y} = C_{x,y} \times \left(O_{min} + (1 - O_{min}) \cdot \ln D_{x,y} / \ln D_{max}\right)$$

where (x,y) denote the pixel position, D is the density texture, D_{max} is the maximum overplot (density) that occurred, C is the color texture, and O_{min} is a constant, user-defined minimum opacity value.

With this method, we create a visual history of all the events up to the current point in the animation in an unobtrusive manner. At any point, the user can also manually clear the background buffer or the currently fading particles, allowing for the reset of the history if it gets too cluttered. One disadvantage of keeping track of history in this way is that it only tracks the history of what is animated; if the user moves to a new time position via the time line, events skipped in this manner will not appear in the history. A final history image from the end of a trace animation is shown in Figure 4.

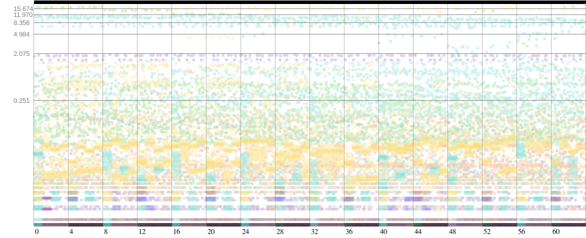


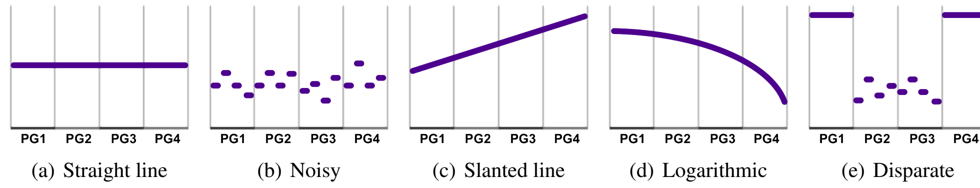
Figure 4: *The history background image at the end of the matrix invert operation. The striations are due to the float precision limitation present in the traces. There are noticeable differences in the duration patterns of MPI Broadcast function calls across the processes.*

4. Results

Most scientific applications require partitioning of their problem space in order to effectively run in parallel. But scientists usually prefer to think of the problem in terms of linear algebra operations, rather than manually partitioning and manipulating individual data blocks. ScaLAPACK is a widely used parallel linear algebra package, as it utilizes libraries that parallelize and abstract away these basic operations, allowing the developers to focus on their own algorithms. We chose to analyze a set of ScaLAPACK distributed matrix operations for our case studies primarily because it is a building block for many scientific simulations. Due to the way ScaLAPACK abstracts matrix operations, when we analyzed complex traces with multiple operations, the visualization clearly shows separate stages, each performing an individual matrix operation. Thus, to simplify our process and aid understanding of communication patterns, we here analyze traces of individual parallel matrix operations.

We collected data on the Franklin and Kraken supercomputers. As it is the smaller of the two systems, the operations

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.



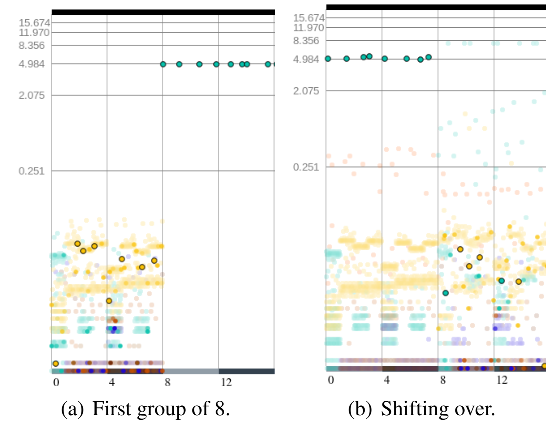
(a) Straight line (b) Noisy (c) Slanted line (d) Logarithmic (e) Disparate

Figure 5: *Common patterns revealed by our visualization. The four bottom segments labeled PGx encode four process groups. These patterns may occur both during the animation and in the background image.*

we ran on Franklin had smaller matrix sizes, to yield reasonable execution times and yet still observe communication patterns. For the Kraken tests, we scaled the input data size to the number of processes, resulting in larger trace files and longer communication events with more temporal gaps.

In Figure 5 we present illustrations of some of the particle alignment patterns we observed in our case studies. A **horizontal line** (Fig. 5(a)) means that the events all started at the same time (animation), or that they all had the same duration (history image). This pattern indicates full synchronization across processes which implies a balanced systems. A **noisy** pattern (Fig. 5(b)) appears when there are differences in the start times (animation) or durations (history) of events across processes. It indicates potential performance problems, as it is usually caused by de-synchronization, load imbalance or wait conditions. The **slanted line** (Fig. 5(c)) is a degenerate case between the noisy and logarithmic patterns. It occurs when events start later or take longer to complete as process ID increases or decreases and can only be seen in the lower section of the animation window, where event particles travel at high speed. If the slope is low, the slanted line does not indicate major performance problems, but could be attributed to fluctuations in the network. An example of a slanted line with low slope formed by *MPI_Bcast* events (yellow) can be seen in Figure 6 between processes 0 and 8 and below the first horizontal grid line. The **logarithmic** pattern (Fig. 5(d)) means that there is a linear delay in event start times or a linear progression of event durations as process ID increases. It may indicate serialization in the propagation of data, which is detrimental to performance. It may also relate to the network topology, with processes further from the data source taking longer to receive messages. Finally, the **disparate** pattern (Fig. 5(e)) appears when a group of processes exhibit significantly different communication activity from the rest. This may be due to them performing different computational tasks than the other processes and does not necessarily indicate a performance problem.

Examples from the Franklin **Matrix Inversion** execution trace have been presented in Section 3. This operation exhibits an interesting behavior in that as it runs, 8 of the 64 processes appear to be working on computation and execute very short communication operations, while the others are waiting on a long *MPI_Recv* operation. This group of 8 shifts to the left throughout execution, starting with ranks 0 to 8 and ending with 56-63. The first transition is illustrated in Figure 6.



(a) First group of 8. (b) Shifting over.

Figure 6: *The shifting "group of 8" pattern. In 6(a), the first 8 processes are performing short MPI_Bcast and MPI_Recv operations, while the other processes appear to be idle. In 6(b), the pattern shifts, with the next 8 processes working and the first waiting on long MPI_Recv operations.*

This pattern indicates that the ScaLAPACK matrix inversion algorithm divides the matrix into blocks and assigns each block to a group of processors. Excepting the very first one, each processor group must wait for the previous results to become available before it can start processing its block.

The **Cholesky Decomposition** operation on Franklin does not exhibit the same group patterns as Matrix Inversion. For the most part, communication activity is regular across all the processes, as can be seen from the time line in Figure 7. One interesting feature present in this trace is the appearance of logarithmic curve patterns in the positions of the particles as they travel upwards, which can be observed in

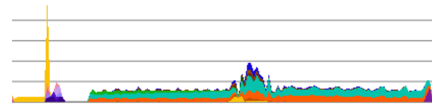


Figure 7: *Time line for the Cholesky Decomposition operation on Franklin. Total duration is approximately 0.24 seconds. After the initialization stage, which is dominated by a spike in the number of calls to MPI_Bcast, communication activity drops to a relatively low level and remains constant for the duration of the execution. The slight increase in activity in the middle indicates that this is a two-stage operation.*

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

Figure 8. These patterns are prevalent among the first half of the processes, during the second half of the execution. Due to their regularity, they are unlikely to be caused by clock skews across the computation nodes. Since the particles are ordered by process ID and their position on the y axis is on a logarithmic scale, this means that, within a process group of 8 processes, events start with a linear delay as process ID increases. These events all have approximately the same duration, indicating possible network delays or the necessity for processor P_i to wait for information from processor $P_i - 1$ in order to carry on its computation. The latter explanation is likely, because the ScaLAPACK Cholesky decomposition function uses a right-looking algorithm [CDO*96] which requires updating all columns to the right of the currently computed column i which are modified by it to be updated when column i is updated.

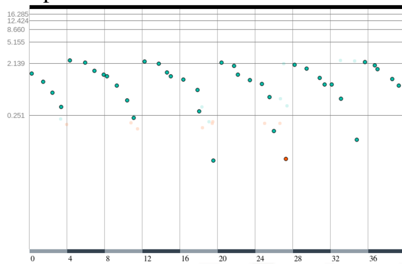


Figure 8: *Linear delay patterns present in the Cholesky Decomposion trace. Similar patterns can be observed in the Matrix Inversion operation, in Figure 2.*

We collected traces of parallel **Eigenproblem** solving operations on both the Franklin and Kraken systems. On Franklin, this operation is relatively long, running for 0.54 seconds on 64 processes with a matrix of size 2000x2000. Its time line (Figure 9) shows that the ScaLAPACK Eigensolver function (*"pdsyevd"*) executes in stages and that the total number of MPI function calls tends to decrease with each stage. Upon watching the animation, we discover that the

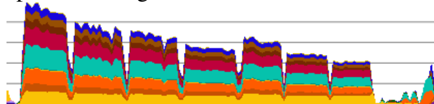


Figure 9: *Time line of the ScalaPACK parallel Eigenproblem solving operation running on the Franklin system.*

stages show similar characteristics to the stages of the Matrix Inversion operation, with a group of 8 processes exhibiting a different communication pattern from the rest at each stage; this group also shifts to the right on every stage. However, in the case of the Eigensolver, the rest of the processes are not idle, but still involved in communication. There is a less significant difference between the event durations of the processes in the group and the rest, as seen in Figure 10. We also found the reason for the decrease in number of MPI calls throughout execution. When the pattern shifts to the next group of 8 processes, a number of processes from the previous groups begin a very long *MPI_Bcast* operation,

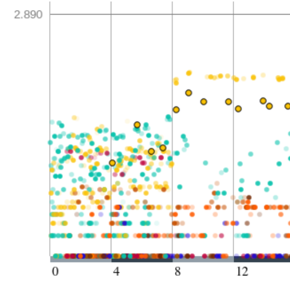


Figure 10: *The first "group of 8" in the Eigenproblem solving operation. Only currently running or recently ended (fading) events are shown here. The background history was cleared prior to taking this snapshot.*

which lasts until the very end of the run, shown in Figure 11. These processes have now become idle: they do not execute any more MPI function calls for the rest of the execution. This may mean that they have completed their allotted work, in which case their prolonged idling would not be a cause for major concern. In fact, these processes do become active for the clean-up phase at the end of the run, indicating that they were likely not locked in a contention for resources. However, this could potentially be an area for performance improvement, as idle processors may be put back in use, thus increasing the efficiency of the computation.

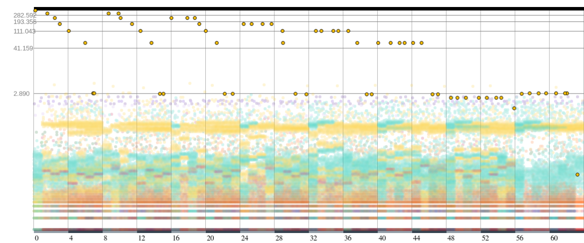
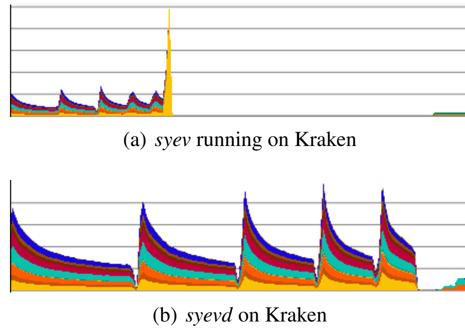


Figure 11: *The end of the Eigenproblem solving operation on Franklin, with the pattern of very long MPI_Bcast calls visible. The background history image shows slightly noisy patterns in event durations across the processes. Most notable are the synchronization operations (calls to MPI_Waitall), which appear to have occurred at each stage within the 8-process groups.*

For tracing Eigensolver operations on Kraken, we increased the matrix size to 6,000x6,000 on 36 processes, with a process grid of 6x6. The resulting time lines are shown in Figures 12(a) and 12(b). This configuration provides a better balance of input data size to number of processes, and we can see this in our visualization: there are larger temporal gaps between bursts of communication, indicating that the system spends time performing actual calculations, which we cannot capture using the MPE tracing system. However, larger data sizes also cause the communication events to become longer, with their durations increasing steadily over the course of the execution. These tests also had a much longer overall execution time than the smaller test on Franklin.

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.



(a) *syev* running on Kraken

(b) *syevd* on Kraken

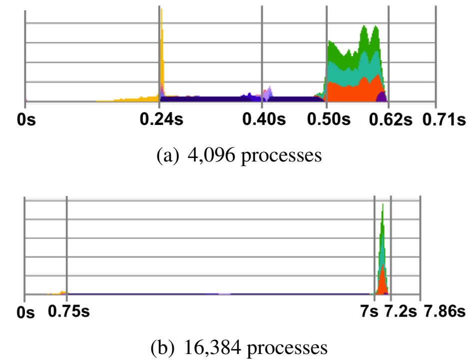
Figure 12: *Both functions execute in five stages, with a logarithmic decrease in the number of MPI calls during each phase. This logarithmic falloff indicates that, as processes finish their computation for a particular stage, they send their data to the next processes and no longer communicate until the next phase starts. Each phase is shorter than the previous one because there is less data to perform computations on, as is the nature of divide and conquer algorithms. syevd also appears to have a large temporal gap between the end of computation and communication and the final clean-up and conclusion of the run. Because Kraken is a system available for use, this is likely caused by other running applications interfering with inter-node communication.*

We collected traces from two ScaLAPACK Eigensolver functions (*"pdsyevd"* and *"pdsyev"*) and compared their behaviors both between each other and against the execution of *"pdsyevd"* on Franklin. *"pdsyevd"* is an eigensolver based on a QR algorithm [GKK10], while *"pdsyev"* is a parallel implementation of a divide and conquer algorithm [BS].

From our tests, the two functions do not appear to differ significantly in behavior from the point of view of their communication patterns. They both exhibit a similar pattern to our test on Franklin, where certain processes become idle and wait on a very long *MPI_Bcast* operation as the activity shifts through the process groups. We can clearly conclude that this is a feature of the ScaLAPACK eigensolvers' implementation and this supports our hypothesis that these processes have in fact completed their computational tasks. However, unlike our Franklin test, event durations seem to increase slightly over the course of execution. This shows that as computation of matrix blocks finishes, processes need to be aware of the current partial solution, so more data must be transferred through the system.

Matrix Multiplication is the most basic and widely used of the operations we have studied. It was also the basis for our larger-scale tests: we will present two examples which were run on 4,096 and 16,384 respectively, each performing a multiplication operation on "skinny" matrices of 10,000 × 1,000 elements. We show the time lines for these executions in Figure 13. At first glance, we notice that this operation is dominated by setup time at these scales. For instance, the 16k-process run had a total duration of 7.86 seconds, out of which the first 7.0 seconds were spent executing a few, very long *MPI_Comm* functions, i.e. setting up the communication groups. In comparison to the time spent in setup and communication, the computation time appears to be negligible. With such long setup times, it is clear that matrix size needs to be quite large, such that a significant amount of the execution time is spent performing useful computation and thus justifies resource utilization.

In both examples, logarithmic curve trends in particle y-positions become apparent. MPI events tend to start later as process ID increases, but they also tend to have the same duration by function call. There is an abundance of communication events, indicating that communication overhead is significant at these scales. Images from these animations can be found on the next page (Figures 14, 15, 16).



(a) 4,096 processes

(b) 16,384 processes

Figure 13: *Time lines of the matrix multiplication operation. The communication spikes are where the computation starts.*

This case study shows that the ScalaPACK matrix multiplication functions appear to be optimized at smaller scales, but the number of processes involved in such a computation must be carefully calibrated to matrix size, or vice-versa, in order to avoid wasting computational resources.

5. Conclusions and Future Work

We have demonstrated how effective data streaming and animation designs can facilitate the visualization of large data sets. In the case of parallel execution traces, these methods simplify the process of ensuring that events are shown in their relative order and that events executing in parallel are shown as such. With the tendency of parallel systems and applications to grow in size and complexity, such methods are becoming a valuable tool for analyzing the data these systems generate.

Another fact that became apparent is that MPI function calls tend to have a very wide distribution of durations, for which a logarithmic scale is only partially appropriate. A better solution might be to treat long events differently from very short events in terms of visualization. Furthermore, rendering every individual event runs into limitations, even when events are not all on-screen at the same time. In our 16 thousand process example, we already ran into over-plot issues even though there were only as many particles as processes on screen at any given time. Thus, we plan to imple-

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

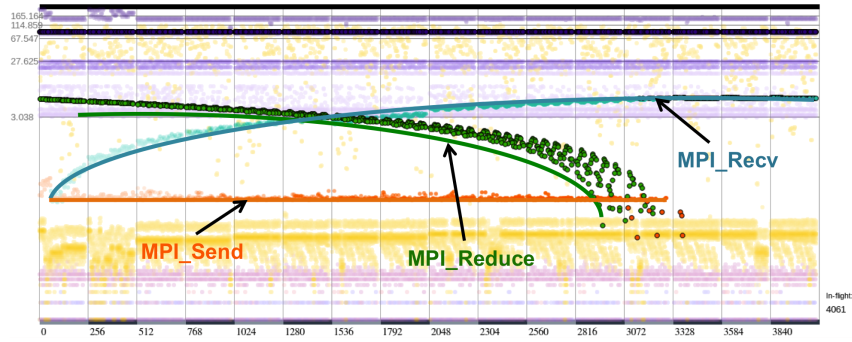


Figure 14: *Matrix multiplication on 4,096 processes, at the end of the communication groups' setup phase. There is an interesting pattern visible here: processes appear to take turns executing MPI_Send, MPI_Recv and MPI_Reduce operations. In particular, the MPI_Recv operations take longer to execute the higher the process ID. Globally, the operation appears to have been initiated by process #0 and each process had to wait for the previous one to receive its data.*

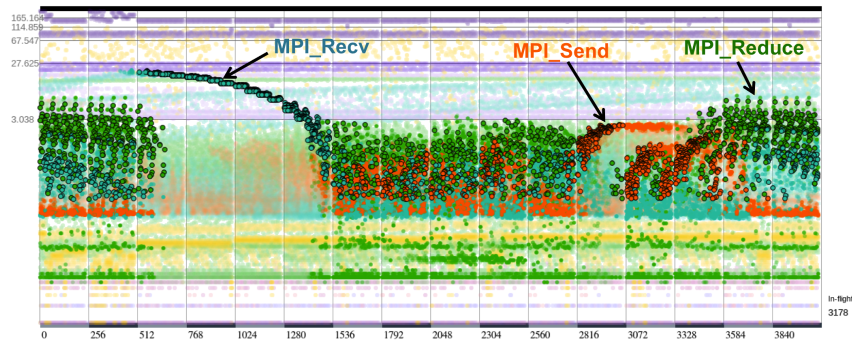


Figure 15: *The same trace as Figure 14 during a communication spike. The log curve of MPI_Recv operations continues to shift toward the right until the end of the execution. Unlike the smaller runs where events of the same types had roughly the same durations, event durations are more spread out and irregular here. This could be due to load imbalances or, to a lack of sufficient data for the number of processes used, or to the increased communication overhead present at these larger scales.*

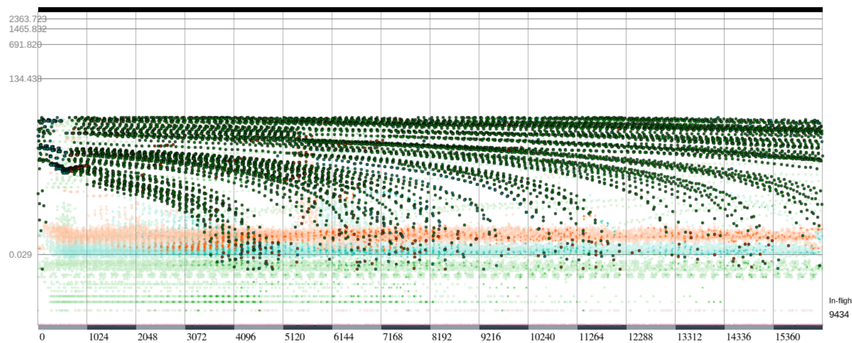


Figure 16: *For the 16,384 process trace, we reduced the size of the particles to minimize the over-plotting of active events. Large-scale logarithmic trends in particle positions are present, indicating delays in event start time across processes.*

ment zooming or filtering features in the current visualization. It may also be necessary to move away from per-event visualizations and towards aggregation methods.

Finally, although we have used communication data collected from ScaLAPACK computations, our tool is capable of producing animations from any MPI trace data and thus our study does not lose generality. Further studies can also instrument and trace the computational activities of other libraries or actual simulations to evaluate their performance.

Acknowledgements

This research has been supported in part by the National Science Foundation through grants CCF-0938114 and OCI-0850566.

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

References

[BS] BREITMOSER E., SUNDERLAND A.: *A Performance Study of the PLAPACK and ScaLAPACK Eigensolvers on HPCx for the Standard Problem*. Tech. rep. 8

[CDO*96] CHOI J., DONGARRA J. J., OSTROUCHOV L. S., PETITET A. P., WALKER D. W., WHALEY R. C.: Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Sci. Program. 5*, 3 (Aug. 1996), 173–184. 7

[CGL08] CHAN A., GROPP W., LUSK E.: An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Sci. Program. 16* (April 2008), 155–165. 2

[CR11] CHOUDHURY A., ROSEN P.: Abstract visualization of runtime memory behavior. In *6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)* (Sept. 2011), pp. 1 –8. 3

[Fra] Franklin. http://www.nersc.gov/users/computational-systems/franklin. 3

[GA10] GREEN-ARMYTAGE P.: A colour alphabet and the limits of colour coding. *JAIC-Journal of the International Colour Association 5* (2010). 4

[GKK10] GRANAT R., KÅGSTRÖM B., KRESSNER D.: A novel parallel QR algorithm for hybrid distributed memory HPC systems. *SIAM Journal on Scientific Computing 32*, 4 (2010), 2345–2378. 8

[GWW*10] GEIMER M., WOLF F., WYLIE B. J. N., ÁBRAHÁM E., BECKER D., MOHR B.: The Scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper. 22* (April 2010), 702–719. 2

[HE91] HEATH M., ETHERIDGE J.: Visualizing the performance of parallel programs. *Software, IEEE 8*, 5 (sep 1991), 29 –39. 2

[Hea93] HEATH M.: *ParaGraph: A Tool for Visualizing Performance of Parallel Programs*. Tech. rep., 1993. 2, 3

[HL91] HERRARTE V., LUSK E.: *Studying Parallel Program Behavior with* `Upshot`. Tech. Rep. ANL–91/15, Argonne National Laboratory, 1991. 2

[JLJC05] JOHANSSON J., LJUNG P., JERN M., COOPER M.: Revealing structure within clustered parallel coordinates displays. In *IEEE Symposium on Information Visualization (INFOVIS), 2005* (oct. 2005), pp. 125 – 132. 2

[KL94] KARRELS E., LUSK E.: Performance analysis of MPI programs. In *Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing* (1994), Dongarra J., Tourancheau B., (Eds.), SIAM Publications, pp. 195–200. 2

[Kra] Kraken. http://www.nics.tennessee.edu/computing-resources/kraken. 3

[KS94] KRAEMER E., STASKO J. T.: Toward flexible control of the temporal mapping from concurrent program events to animations. In *Proceedings of the 8th International Symposium on Parallel Processing* (1994), pp. 902–908. 3

[KST01] KEHOE C., STASKO J., TAYLOR A.: Rethinking the evaluation of algorithm animations as learning aids: an observational study. *International Journal of Human-Computer Studies 54*, 2 (Feb. 2001), 265–284. 3

[LLB*12] LANDGE A. G., LEVINE J. A., BHATELE A., ISAACS K. E., GAMBLIN T., SCHULZ M., LANGER S. H., BREMER P.-T., PASCUCCI V.: Visualizing network traffic to understand the performance of massively parallel simulations. *IEEE Trans. on Visualization and Computer Graphics 18* (2012), 2467–2476. 2

[MGM09] MUELDER C., GYGI F., MA K.-L.: Visual analysis of inter-process communication for large-scale parallel computing. *IEEE Trans. on Visualization and Computer Graphics 15*, 6 (Nov.-Dec. 2009), 1129 –1136. 1, 2, 4

[MSM*11] MUELDER C., SIGOVAN C., MA K.-L., COPE J., LANG S., ISKRA K., BECKMAN P., ROSS R.: Visual analysis of I/O system behavior for high-end computing. In *Proceedings of the third international workshop on Large-scale system and application performance (LSAP)* (2011), pp. 19–26. 2, 4

[MW] MOHR B., WOLF F.: KOJAK - A tool set for automatic performance analysis of parallel programs. In *Euro-Par 2003 Parallel Processing*, Lecture Notes in Computer Science. 2

[NAW*96] NAGEL W. E., ARNOLD A., WEBER M., HOPPE H.-C., SOLCHENBACH K.: Vampir: Visualization and analysis of MPI resources. *Supercomputer 12* (1996), 69–80. 2, 3

[OM09] OGAWA M., MA K.-L.: code_swarm: A design study in organic software visualization. *IEEE Trans. on Visualization and Computer Graphics 15*, 6 (Nov. 2009), 1097–1104. 3

[RFF*08] ROBERTSON G., FERNANDEZ R., FISHER D., LEE B., STASKO J.: Effectiveness of animation in trend visualization. *IEEE Trans. on Visualization and Computer Graphics 14*, 6 (Nov.-Dec. 2008), 1325 –1332. 3

[Sca] ScaLAPACK http://www.netlib.org/scalapack/. 1

[SHN] SCHNORR L., HUARD G., NAVAUX P.: 3D approach to the visualization of parallel applications and grid monitoring information. In *9th IEEE/ACM International Conference on Grid Computing*, pp. 233 –241. 2

[SK93] STASKO J. T., KRAEMER E.: A methodology for building application-specific visualizations of parallel programs, 1993. 3

[SM06] SHENDE S. S., MALONY A. D.: The Tau parallel performance system. *Int. J. High Perform. Comput. Appl. 20* (May 2006), 287–311. 2

[SML*12] SPEAR W., MALONY A. D., LEE C. W., BIERSDORFF S., SHENDE S.: An approach to creating performance visualizations in a parallel profile analysis tool. In *Proceedings of the 2011 international conference on Parallel Processing - Volume 2* (2012), Euro-Par'11, pp. 156–165. 2

[SRWS99] SHAFFER E., REED D., WHITMORE S., SCHAEFFER B.: Virtue: performance visualization of parallel and distributed applications. *Computer 32*, 12 (dec 1999), 44 –51. 2

[TMB02] TVERSKY B., MORRISON J. B., BETRANCOURT M.: Animation: can it facilitate? *International Journal of Human-Computer Studies 57*, 4 (Oct. 2002), 247–262. 3

[TSS95] TOPOL B., STASKO J. T., SUNDERAM V.: *The Dual Timestamping Methodology for Visualizing Distributed Applications*. Tech. rep., IEEE Parallel & Distributed Technology, Systems & Applications, 1995. 3

[WM00] WOLF F., MOHR B.: Automatic performance analysis of mpi applications based on event traces. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing* (2000), pp. 123–132. 2

[WM03] WOLF F., MOHR B.: Automatic performance analysis of hybrid MPI/OpenMP applications. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003.* (feb. 2003), pp. 13 – 22. 2

[ZLGS99] ZAKI O., LUSK E., GROPP W., SWIDER D.: Toward scalable performance visualization with Jumpshot. *Inernational Journal of High Performance Computing Applications 13* (Aug. 1999), 277–288. 2

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.