Visual Analysis of Inter-Process Communication for Large-Scale Parallel Computing

Chris Muelder, Francois Gygi, and Kwan-Liu Ma, *Senior Member, IEEE*

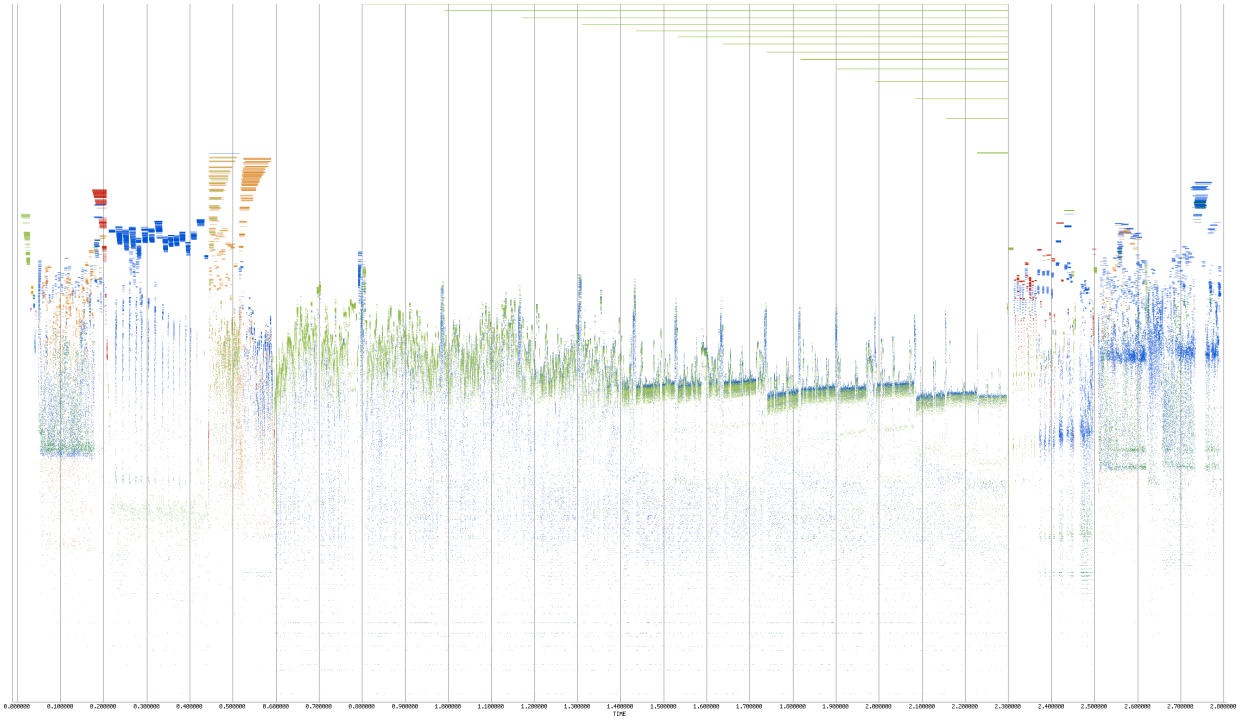


Fig. 1. *A scalable MPI visualization.* Rather than plotting MPI calls per process, this view plots the duration of the call on the y-axis with a log scale versus time on the x-axis. Patterns such as simultaneous start/end times, clusters and trends of similar calls, and particularly long communications can be seen. The data was collected from running matrix operations from the ScaLAPACK library on 256 nodes of NERSC's Franklin supercomputer.

Abstract— In serial computation, program profiling is often helpful for optimization of key sections of code. When moving to parallel computation, not only does the code execution need to be considered but also communication between the different processes which can induce delays that are detrimental to performance. As the number of processes increases, so does the impact of the communication delays on performance. For large-scale parallel applications, it is critical to understand how the communication impacts performance in order to make the code more efficient. There are several tools available for visualizing program execution and communications on parallel systems. These tools generally provide either views which statistically summarize the entire program execution or process-centric views. However, process-centric visualizations do not scale well as the number of processes gets very large. In particular, the most common representation of parallel processes is a Gantt chart with a row for each process. As the number of processes increases, these charts can become difficult to work with and can even exceed screen resolution. We propose a new visualization approach that affords more scalability and then demonstrate it on systems running with up to 16,384 processes.

Index Terms— Information Visualization, MPI Profiling, Scalability.

1 Introduction

Many phenomena are difficult or simply impossible to fully study experimentally due to a lack of reliable methods for measuring or controling the experiment. For example, very large-scale phenomena such as supernovae cannot be tested experimentally, and very small-scale phenomena such as interaction of particles at the quantum level cannot be precisely measured. In order to study such phenomena, scientists often use numerical simulations. These simulations can supplement existing partial results or lead to new insights which can guide scientists to specific experiments which would confirm the results. However, to obtain useful results, these simulations usually involve large and complicated calculations, which cannot be processed in a reasonable time on a serial computer. Therefore, a parallel supercomputer must be harnessed for such large-scale high-resolution modeling. For decades, significant investments have been made to advance areas of scientific research by creating larger and more powerful parallel su-

- *Chris Muelder, Francois Gygi, and Kwan-Liu Ma are with University of California, Davis.*
- *E-mails: muelder@cs.ucdavis.edu,fgygi@ucdavis.edu, and ma@cs.ucdavis.edu*

Manuscript received 31 March 2009; accepted 27 July 2009; posted online 11 October 2009; mailed on 5 October 2009.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org .

percomputers. In recent years, large-scale systems have gone from tera-scale to peta-scale, and are continuing to exa-scale systems, enabling scientists to study complex problems which were previously intractable.

Many such systems are being operated by both U.S. Department of Energy (DOE) and National Science Foundation (NSF). DOE's "Scientific Discovery through Advanced Computing" (SciDAC) is researching ways to optimize and utilize large-scale systems, and maintains and funds several of them at different DOE sites, such as the Oak Ridge National Laboratory, Sandia National Laboratories, Lawrence Berkley Laboratory, and Lawrence Livermore National Laboratory [21]. NSF sponsors several systems of its own though its PetaApps [18] program, including those at University of Texas, Austin [26] and University of Illinois at Urbana-Champaign [16]. These systems involve many thousands of processors networked together to allow for large-scale computational simulations.

In order to fully utilize such large parallel systems, the algorithms and calculations within the simulation must be carefully parallelized. Most implementations of parallel scientific computing use a message passing paradigm, such as that used by the standard Message Passing Interface (MPI). While some tasks can be embarrassingly parallel, many of the operations necessary for these complex simulations are not trivial to parallelize. For instance, many of the matrix algorithms in First Principle Molecular Dynamics (FPMD) simulations take $O(n^3)$ operations to work with $O(n^2)$ data [13]. The result of this is that the more parallel this kind of algorithm is made, the more sparsely data is divided among the processors and hence the more communication is necessary between processes to exchange data. As the number of processors increase, it becomes more effective for optimization to analyze and reduce the communication than to use metrics such as numbers of operations.

One common way to analyze the communications of such programs is through visualization. Several libraries and tools have been developed to capture MPI events and visualize the captured communication patterns. These tools, while effective at analyzing small systems, often do not scale well to large, massively parallel systems. For instance, one common visualization in most existing tools is a Gantt chart, which lines up the processes vertically and plots the MPI events versus time on the horizontal axis. This technique runs into problems once the number of processes exceeds the number of pixels available on the display. We propose an alternate visual analysis strategy for understanding MPI communications at extreme scales.

Once large-scale computation involves tens-of-thousands to millions of processes, it becomes less useful to consider every process individually; it makes more sense to consider groups of processes or groups of MPI calls before drilling down to individual processes or MPI events. At the highest level, we consider the system as a whole and see how the overall communications are impacting performance over time. Next, we consider the communications at the level of groups of processes by plotting related communications together regardless of the participating processes. This way, MPI calls can be represented at an abstract level regardless of the number of processes. Finally, individual calls and processes can be singled out from this view. We present a scalable approach to MPI visualization that does this by using a timeline overview in combination with focused views which are abstracted from individual processes. The focused view achieves this by directly mapping the MPI events in a temporal space regardless of process rank and using modulated opacity to show process density, as shown in Figure 1. We also show that with our visualization strategy it becomes possible to understand communication behaviors at a large scale and identify room for performance optimization.

1.1 Related work

Software visualization is a fairly broad field. Many visualizations focus on managing software development and repositories [24]. StarGate [17] is a tool that visualizes both the evolution of the software repository and the communication patterns of the developers involved. Other visualizations focus on visualizing the code itself and aid in the analysis of code dependencies in larger projects. Some use visualization to analyze and reverse engineer compiled binary code [25]. Using visualization to optimize performance has been approached in several ways by existing work. For example, TraceVis [19] visualizes the execution times of individual CPU instructions, and Bootchart [1] visualizes the performance of programs involved in the boot process of an operating system. Both of these examples use variants of Gantt charts to present the information. However, these tools focus on serial programming, where parallel issues such as communication delays do not come up.

The problem of characterizing communication has been studied by many researchers. Network monitoring tools such as EtherApe [4] and EZEL [28] show communication patterns well, but they focus on pure network activity and do not incorporate properties particular to distributed communication. The communications between parallel processes and data storage servers has also been researched through analysis of access patterns [20, 31, 32]. Visualization of communication between software modules such as client-server relationships have also been analyzed through the use of graph based visualizations [33]. These visual approaches are effective at analyzing network traffic, but by focusing singly on network information, the impact on computation efficiency in a massively parallel computation environment would be difficult to deduce

One common set of visualization tools for MPI data is Jumpshot [2, 30] and its predecessors (Nupshot [10] and Upshot [7]). These tools use the MPI Parallel Environment (MPE) library to intercept the MPI calls in a parallel program. Then they visualize the collected trace with a Gantt chart by plotting process rank versus time using color to represent the MPI calls. ParaGraph [6] is another, older program that visualizes MPI traces collected with the MPICL library which also uses Gantt charts, among other metrics such as overall summaries and communication graphs. Vampir [15] is another visual tool which combines Gantt charts and summary views. The Tuning and Analysis Utilities (TAU) [23] suite of tools is one of the more comprehensive tools. The logging facilities included with it allow for conversion to many of the formats used by other existing tools, such as Jumpshot or Vampir. Its own visualizations include Gantt charts, a communication matrix view, and a call graph, among others. Virtue [22] is the most unique of the related works listed here in that it is a real-time visualization. This allows the user to monitor the performance of an application while it is running and potentially tune it or interact with it. It also incorporates VR techniques such as a CAVE (Cave Automatic Virtual Environment) to provide a more immersive visualization than most other tools. For other parallel environments, GVUs PVaniM tool [27] and ATEMPT [11, 12] present some detailed views of communication events in a PVM (Parallel Virtual Machine) system.

Some software visualizations address the scalability issues of plots such as Gantt charts. The works of Jerding et al. [8], Moreta and Telea [14], and Cornelissen et al. [3] use plots similar to Gantt charts to profile program execution traces. However, these works maintain the strict ordering of the charts, and use sub-pixel techniques to handle the scalability and allow for visibility of both large trends and outliers. In contrast, our approach sacrifices the ordering to spatially separate large trends from individual outliers.

Our approach draws upon several existing visualization techniques. The timeline view consists of a stacked graph representation, and the detailed view is based on techniques such as scatterplots and arc diagrams [29]. In order to plot a large number of calls simultaneously, we also incorporate existing techniques such as high precision alpha blending and opacity scaling similar to the work by Johansson et al. [9].

2 A Scalable Approach

As the number of processes increases, the usefulness of keeping track of individual processes lessens, and it becomes more helpful and more useful to consider the system as a whole or in part before looking at individual processes. However, it is still useful to be able to drill down into the details of the data, so we implemented an interactive focus+context visualization which presents a high level abstraction, a focused view, and details on demand. The high level view consists of

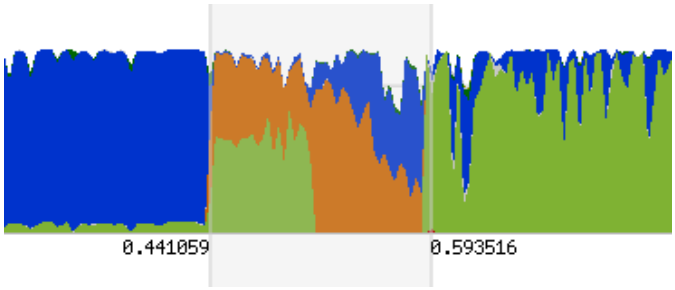


Fig. 2. *Section of a timeline of MPI calls.* The timeline provides an overview of the activity of the entire system on the y-axis versus time on the x-axis. Each layer corresponds to an MPI function, and the height is the fraction of processes calling that function. Distinct homogenous sections of the timeline correspond to the various matrix operations performed in the program. From the timeline, ranges of data can be selected to view in more detail.

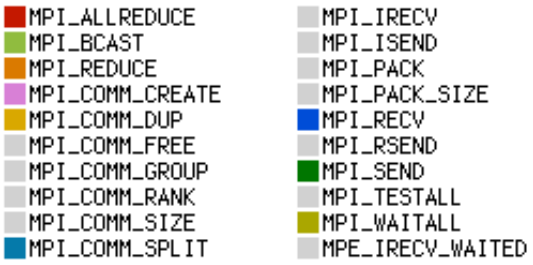


Fig. 3. *Color Legend.* The colors used in all figures. For clarity, the most prominent functions have been colored while the less commonly visible functions have been grayed out.

a timeline view which shows the status of the entire system over the entire run by depicting what fraction of the system is performing what MPI calls over time. From the timeline, a range of time can be selected to focus on. The MPI call view plots MPI calls within this range directly with respect to time, using opacity to handle overplotting issues due to the scale of the data. From the MPI call view, the individual processes can be highlighted to provide specific details to the user.

2.1 Timeline View

The timeline view depicts a stacked graph of the overall process activity over time. Each stacked area of the graph is associated with an MPI function, and its height represents the fraction of the processes that were calling that function at that time. One result of this is that the height of the remaining space which is empty corresponds to the efficiency of the system as a whole, as that is the fraction of processes not involved in communication at that time. Figure 2 shows a small portion of the timeline view blown up for clarity, with the MPI functions colored according to the legend in Figure 3. The more common functions are colored in unique colors while the less common functions are all grey. The timeline view is also used as an interface to select smaller time ranges to view in more detail, and the selected range is indicated by the semi-transparent box shown in Figure 2.

2.2 MPI Call View

The most direct representation of the MPI calls is to render each call from each process with respect to time. Gantt charts do this, but they restrict the y-axis to represent the MPI calls' originating processes. While we retain the use of the x-axis as time, we chose to use the y-axis to represent other properties of the MPI calls. In particular, we found it effective to use the y-axis to represent duration of the MPI calls, particularly on a log scale as the durations vary over several orders of magnitude. The advantage to using duration on the y-axis is that large delays due to communication will be prominently seen at the top of the plot. Since this and other y-axis mappings allow the MPI calls to overlap, we modulate the opacity of the calls, which makes the overall intensity of the visualization represent the density of the MPI calls. The color is mapped to the MPI function being called as in the timeline. For the representation of the calls themselves, we explored several options, including arches, lines, and individual points, examples of which are shown in Figure 4.

Of the representations we use, the arch representation, shown in Figure 4(a), is the least scalable, but is probably also the most intuitive. The beginning point of each arch corresponds to the start time of the MPI call, and the end point of the arch corresponds to the end time. The y-position of the apex is proportional to the duration of the call on some scale. Figure 4(a) is on a linear scale, and depicts patterns that show dependency relationships such as when many processes are dependent on a previous synchronized MPI call or when one global communication is blocking. These are indicated by sets of MPI calls that either start or end nearly simultaneously.

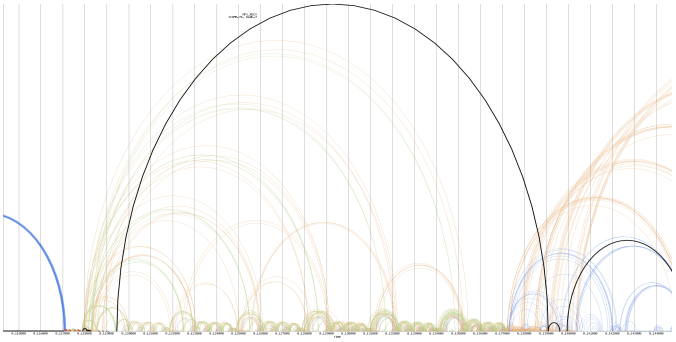
The line representation, shown in Figure 4(b), is the most similar to traditional Gantt charts. Each line goes from the start of the MPI call to the end of the MPI call. As in the other representations, the y-position is proportional to the duration and, in this example, is on a log scale. This representation is more scalable than the arc representation as it produces less clutter on the screen. This comes at the cost of being able to readily see dependencies, as dependent MPI calls no longer touch. However, patterns of simultaneous starting and stopping of MPI calls are readily visible as vertically linear and logarithmic trends. Also, clear groupings of MPI calls can be seen, corresponding to the originating MPI functions in the code.

As the duration of the MPI calls are already being encoded in the height, it is redundant to show duration on the x-axis as well. So the final and most scalable representation of MPI calls we implemented uses simple points to plot the duration of the MPI calls versus either the start or end times of the MPI calls, as shown in Figure 4(c). Similar to the line representation, dependency information is not easily visible. However, vertical and logarithmic trends clearly delimit simultaneous function calls and returns. When plotting start times versus duration, the vertical trends show simultaneous start times and the log curves to the left show simultaneous end times, and when plotting end time versus duration it is the other way around, with the log curves going to the right.

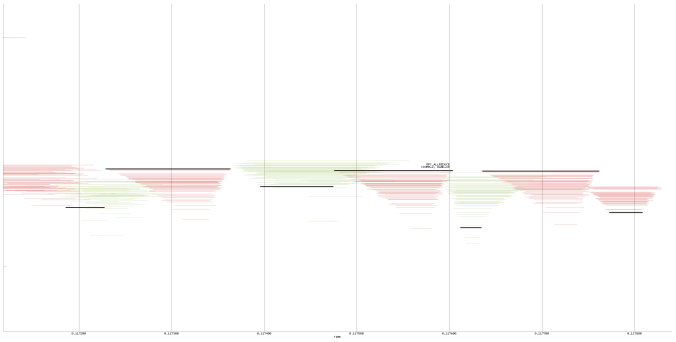
From any of these representations, details of any MPI call can be determined by selecting it with the mouse, at which point all the calls from the selected call's process are highlighted, and details about the selected MPI call are presented to the user textually, as is demonstrated in all three examples in Figure 4. MPI functions can also be highlighted by selecting them from the color legend, at which point all calls to that function get highlighted in the call view.

2.3 Opacity Scaling

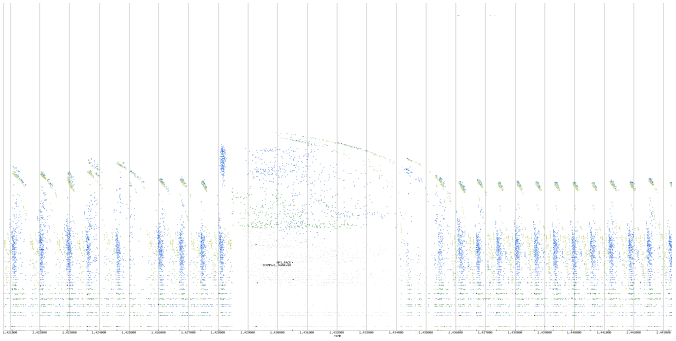
When plotting the MPI calls with our approach, many of them overlap, particularly when they start or end simultaneously. A simple way to resolve this overlap is to make the calls semitransparent and use alpha blending to combine them. However, this very quickly runs into limitations as the number of calls increases, as shown in Figure5(a). First, the standard 8-bit alpha buffer only allows for a maximum overplotting of 256. And second, in order to show overlap of large numbers of MPI calls, the opacity has to be set so low that outliers are nearly invisible. In order to keep both the opacity of outliers high and the combined opacity of dense overlap from overflowing the alpha buffer, we utilize the opacity scaling techniques of [9]. In our implementation of this technique, we first render to a high precision density buffer D which keeps track of the total amount of overplot and then to a high precision color buffer C which blends the input color information with opacity inversely proportional to the density information to result in an average color that is fully opaque. We then combine these buffers with a transfer function to render the final pixels P to the screen. We implemented two such functions: a linear map, and a logarithmic map.



(a) Arches

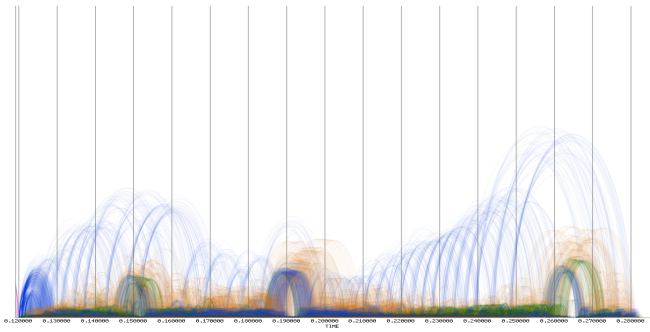


(b) Lines

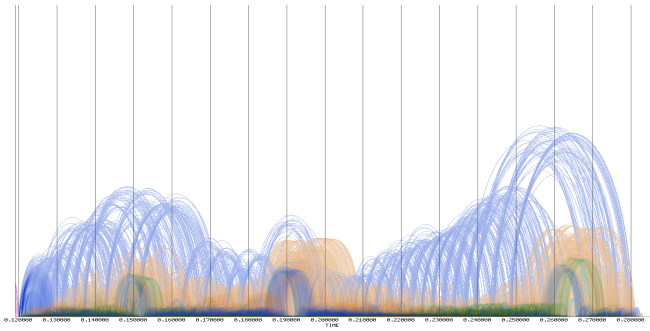


(c) Points (start points pictured)

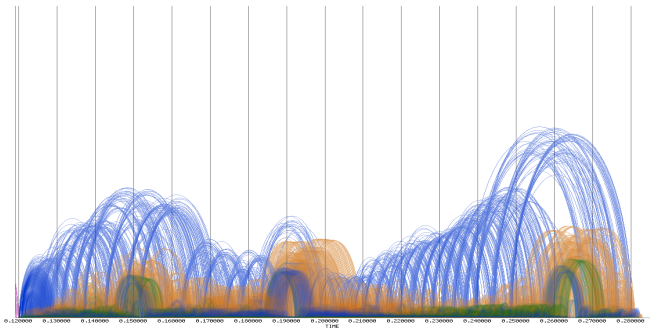
Fig. 4. *MPI Call Plots.* Different representations of MPI calls for direct visualization. Arches (a) are easier to visually follow, while lines (b) and start/end points (c) are more scalable with respect to screen space. In each case, one call is selected, with details presented in text form, and with all calls from the same process highlighted.



(a) Standard opacity accumulation



(b) Linear opacity map with minimum value



(c) Logarithmic opacity map with minimum value

Fig. 5. *Opacity Scaling.* Standard opacity accumulation (a) has trouble both with too much overplotting and with outliers being too transparent. Applying a minimum value and scaling the opacity (b) helps, and applying a logarithmic scale (c) helps even more.

The linear map (shown in Figure5(b)) is defined as:

$$P_{x,y} = C_{x,y} \times \left(o_{min} + (1 - o_{min}) \times \frac{(D_{x,y})}{(D_{max})} \right)$$

And the logarithmic map is defined as (shown in Figure5(c)):

$$P_{x,y} = C_{x,y} \times \left(o_{min} + (1 - o_{min}) \times \frac{log\,(D_{x,y})}{log\,(D_{max})} \right)$$

Where o_{min} is a user defined minimum opacity level and D_{max} is the maximum level of overplotting that occurred. By calculating the final opacity in this manner, we guarantee that any outliers will have at least opacity o_{min}, that no overplotting exceeds the maximum opacity and, in the case of the logarithmic map, that the system will be able to handle many orders of magnitude of overplotting.

3 Case Studies

Many simulations are performed through the use of large linear algebra calculations. One of the most common tools used to run these calculations is the ScaLAPACK library, which utilizes MPI communications to perform distributed linear algebra calculations. The Qbox FPMD simulation codes, for instance, utilize ScaLAPACK functions intensively. In order to demonstrate our approach of visualizing large parallel MPI traces, we use it to analyze matrix operations which use the ScaLAPACK library and its underlying libraries.

We captured the MPI communication using the Multi-Processing Environment (MPE) library. This generates a standardized log file in either clog2 or slog2 format, which we can then visualize. The examples shown here were run on NERSC's Franklin, which is a Cray XT4 massively parallel processing system with 38,128 Opteron compute cores and a peak performance of 356 TFlops/sec [5]. All tests were run with one process per processor, so that no extraneous context switching overhead would be incurred. While tracing adds overhead for writing the log file out at the end of the program, we found that the impact on performance of actual computation was negligible.

Common Matrix Operations Figure 6 shows the results of visualizing a series of common matrix operations. The operations chosen are commonly used in scientific calculations. In this example, the operations were run on 256 processes. From the timeline in Figure 6(a), the first thing that is plainly visible is that the program went through several visually distinct stages, each of which correspond to different matrix operations. We visualize each section in more detail, then compare and contrast them.

The first operation performed was a matrix multiplication, shown in Figure 6(b). As indicated by the colorings, the matrix multiplication's communication pattern mostly consists of MPI_Send, MPI_Recv, and MPI_Reduce, with the MPI_Recv calls generally taking the longest. The calls are staggered and generally quite short, indicating that the algorithm is already well optimized. It ends with a single large MPI_All_Reduce which resynchronizes the system.

Inversion is more complicated than multiplication, involving multiple steps. It starts with an LU decomposition (Figure 6(c)), then uses the resulting triangular matrices to calculate the actual inverse (Figure 6(d)). The LU decomposition consisted almost entirely of MPI_Recv calls, with the corresponding MPI_Send calls barely visible. Interestingly, there is a very cyclic pattern, alternating between short calls and long calls. The strong synchronicity of the communications in this section is also interesting, and it could indicate potential for optimization either through redistribution of the data or changing the communication methods to be more asynchronous. Figure 6(d) shows the completion of the matrix inversion and contains two sub-sections. These sections each start with large calls to MPI_Reduce, and there are many shorter calls to MPI_Bcast in the first half and MPI_Recv in the second half. While the calls to MPI_Bcast and MPI_Recv are quite numerous, they are generally short and staggered. The real expense here are the MPI_Reduce calls, which keeps many processes idle for a long time. The these MPI_Reduce calls form a very distinctive pattern where they start synchronously, but their ends follow a logarithmic trend. This pattern could be indicative of a network communication issue. For instance, this pattern could be induced by using a logical tree communication network when the underlying physical network topology is a actually a torus.

The eigenproblem is an eigenvector/eigenvalue solver and is the single largest matrix operation in this case study, so in Figure 6(e), we only show a representative part of it. Most of the MPI calls here also group together into distinct clusters, and they are a mix of MPI_Reduce and MPI_Recv calls, with the MPI_Reduce taking slightly longer. However, they are all very short compared to the single large MPI_Bcast which starts at the top of Figure 6(e), gradually locks more processes as the program progresses, and does not finish until the middle of Figure 6(f), at which point some processes have been idle for more than half of the total computation time. This can be seen in its entirety at the top of Figure 1. Figure 6(f) depicts more calculations that were involved in the eigenproblem after the long operation finished, such as MPI_All_Reduce calls, along with the MPI_Reduce and MPI_Recv calls which are running very synchronously.

The sixth section, shown in 6(g) shows a Gram-Schmidt orthogonalization, which is composed of 4 individual matrix operations, one of which is trivially parallelizable and takes nearly no time to complete. The communications in the other three operations look much like the matrix multiplication, with the exception that there are gaps between the operations where the processes synchronized and that there are some MPI_All_Reduce calls. However, there is one large cluster of calls to MPI_Send and MPI_Recv near the end which are substantially longer. We determined that this occurred in the middle of the pdtrsm() operation. If this were due to a straggling process or poor load balancing, the calls would end simultaneously. Since they do not, this could indicate a network bottleneck or other system interference.

Testing Scalability While Figure 6 demonstrates our approach on a series of matrix operations on a moderate size system, larger systems should also be considered. In order to investigate the effects of scaling on the visualization, we focus on matrix multiplication, as it is a commonly performed operation. Figure 7 demonstrates the effects of scaling up a matrix multiplication from a modestly small set of processes (64) up to large numbers of processes (16,384). As the number of processes is scaled up, so is the size of the data it is working on. This keeps the communication effects from completely overwhelming the execution, and vice versa. The first observation that can be made from these timelines is that the proportion of time spent doing the actual calculation decreases as the scale goes up. That is, as more processes are used, it takes longer to finish the initialization process to set up the communication channels and distribute the initial data. By 4,096 processes (Figure 7(d)), it already takes more time to initialize the program than to calculate the result. However, this effect would be offset on the more complex programs or larger datasets used in actual simulations, as the shorter computation offered by larger systems marginalizes the cost of initialization. Another observation that can be made is that within the matrix multiplication itself, the more processes there are, the greater the proportion of them that are in the middle of some form of communication at any given time, and thus the lower the efficiency of the system. In particular, by the time we reach 16,384 processes, almost all the time is spent in communication rather than actual computation. Finally, while the communication patterns were fairly cyclic at smaller scales, variances in the communications add up in the larger scales leading to acyclic patterns, as can be seen well in Figure 7(c). To understand what goes on within the matrix operation at large scales, we then focus on it in the detail view.

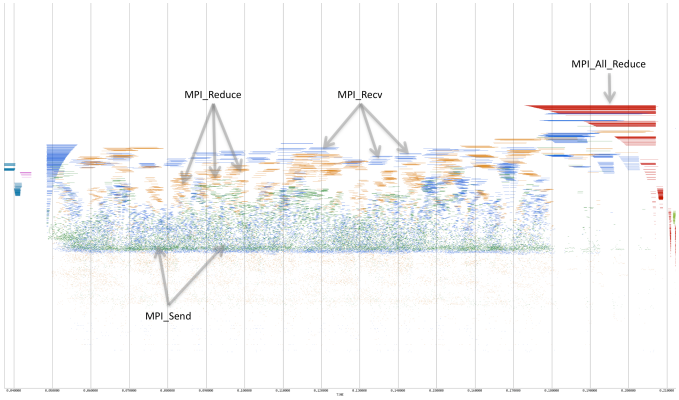
Figure 8 shows the detail of the matrix multiplication on 16,384 processes shown near the end of Figure 7(d). As this scale, the MPI calls are quite dense, so we use the point representations of plotting either start or end times versus duration separately. Figure 8(b) shows the end points of the communication calls. The first major trend visible in this view is that there are two stages in the operation. While the second half is much the same as in smaller scales such as in Figure 6(b), the first half is quite different. It contains MPI calls that took much longer than at the smaller scale. Namely, it begins with MPI_Recv calls that take a long time followed by some MPI_Reduce calls which took longer than normal. It can be seen that there were still unfinished MPI_Comm_Create calls, which would explain the perturbation of the matrix multiplication. Thus in this case, to improve the performance of the matrix multiplication itself it would help to optimize the initialization procedures. After that, the matrix multiplication is a fairly dense mix of MPI_Send, MPI_Recv, and MPI_Reduce calls with some of the MPI_Recv calls taking distinctly longer than the rest of the MPI calls.

In order to better understand the normal communication patterns at this scale, we zoom into a small region of the operation where the communication was fairly regular. Even at this scale, the number of communications and thus points on the screen is quite dense. However, some small trends and clusters of communication are visible. One point of interest is how the communication does split into several very distinct layers. The MPI_Send calls are still the shortest, as in Figure 6(b). Next is a layer of MPI_Recv calls which are also fairly short. Above that are the MPI_Reduce calls, which take longer to distribute data among all processes involved. Finally, there is the distinct layer of clusters of MPI_Recv calls above, which are clearly separated from the rest of the calls.

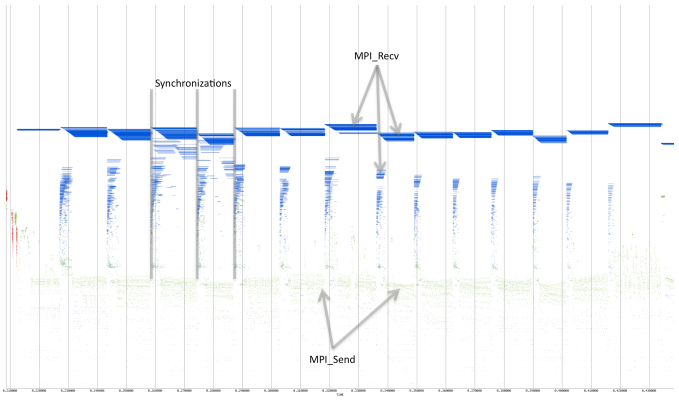
4 Conclusions

As massively parallel computer systems are constantly moving to larger scales, it is becoming ever more important to understand how to use these systems efficiently. Access to these systems is often limited, so scientists cannot usually afford to thoroughly analyze their codes during long term and computation intensive simulations. Our approach uses process independent visualization and focus+context techniques to offer more scalability than traditional parallel system visualizations. And by analyzing a common scientific computation library on a modern supercomputer, our results can aid in refining and optimizing the underlying library used by the scientists, which would allow for more efficient use of the limited access time the scientists are allotted on similar large-scale systems.

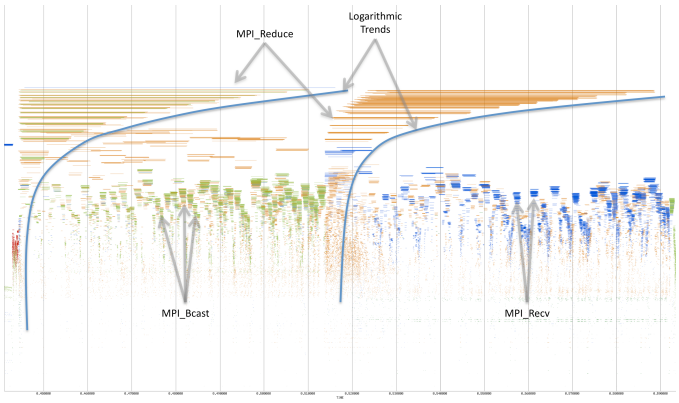
(a) Timeline



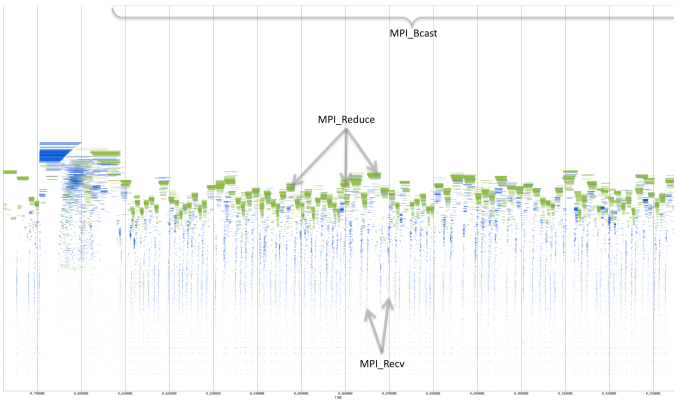
(b) *Matrix Multiplication ("pdgemm()").* This operation seems to be optimized fairly well. While there is much communication going on, the calls are staggered well, keeping overall communication lengths small.



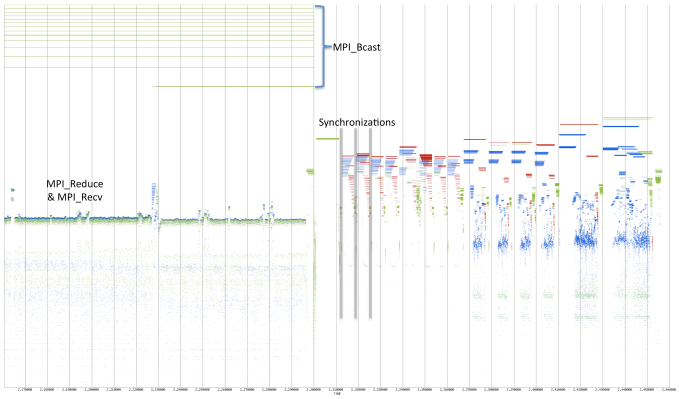
(c) *Matrix Inversion - LU decomposition ("pdgetrf()").* In the LU decomposition, communications alternate between distinct groups of short and medium length calls, indicating synchronizations.



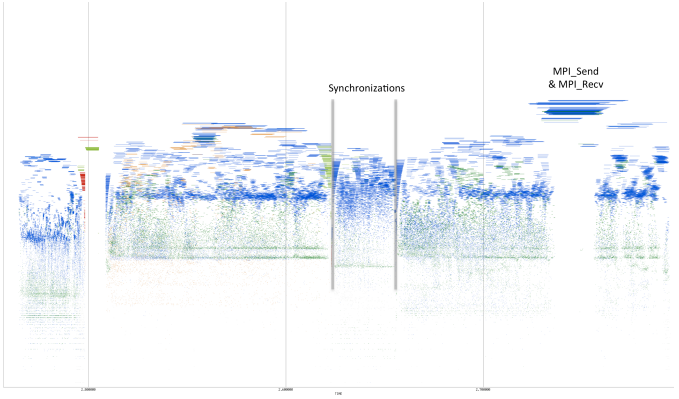
(d) *Matrix Inversion - Invert using LU ("pdgetri()").* This operation has two parts, each with a single large MPI_Reduce and many smaller, staggered communications. The MPI_Reduce calls are potentially inefficient.



(e) *Eigenproblem ("pdsyevd()").* The longest operation, so only a small section is shown here. Mostly, the communications are quite short, but gradually, processes get stuck in very long, inefficient call to MPI_Bcast.

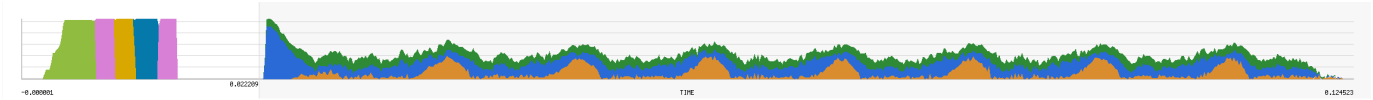


(f) *Eigenproblem cont.*. All the MPI calls from the previous section end synchronously, including the very long MPI_Bcast. Communications are sparse in this next section, indicating periods of heavier computation. The patterns themselves are quite synchronized, similar to the LU decomposition patterns in Figure6(c)

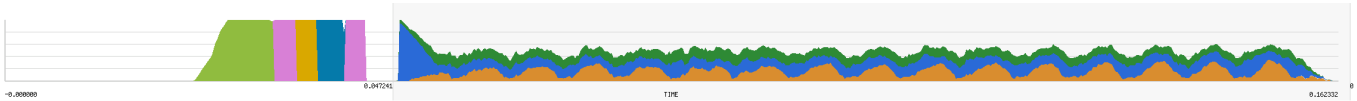


(g) *Gram-Schmidt orthogonalization cont. ("pdsyrk(),"* "*pdsyr(),"* "*pdpotrf(),"* *and "pdtrsm()").* The patterns here are similar to the matrix multiplication patterns in Figure6(b), with two exceptions: the two major synchronization points near the middle which delimit a small section, and abnormally long calls to MPI_Send and MPI_Recv near the end.

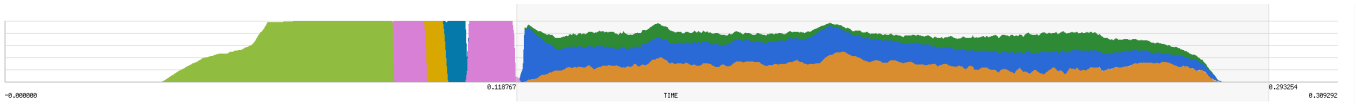
Fig. 6. *A series of matrix operations.* The timeline shows the entire run of program consisting of a series of matrix operations on 256 processes. The individual matrix operations are visible as distinct sections in the timeline (a). Each section is shown in more detail with call plots (b-g). The ScaLAPACK functions are listed in the parentheses.



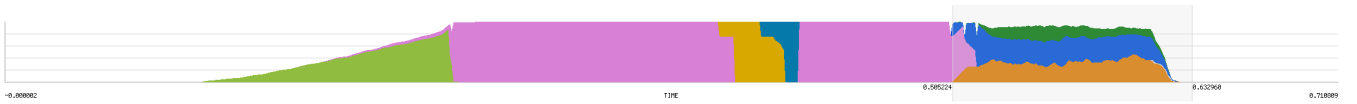
(a) 64 Processes, 10,000x1,000 matrices. Measured = 199 GFlop/s. Peak = 589 GFlop/s. Efficiency = 34%



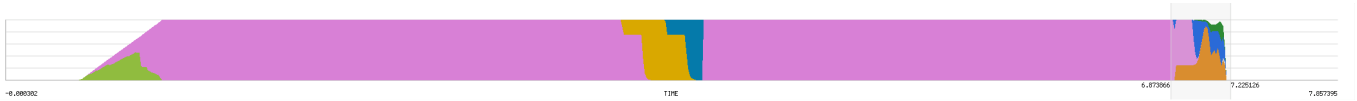
(b) 256 Processes, 10,000x2,000 matrices. Measured = 725 GFlop/s. Peak = 2355 GFlop/s. Efficiency = 31%



(c) 1,024 Processes, 20,000x2,048 matrices. Measured = 1,263 GFlop/s. Peak = 9,420 GFlop/s. Efficiency=13%



(d) 4,096 Processes, 20,480x2,048 matrices. Measured = 1,611 GFlop/s. Peak = 37,683 GFlop/s. Efficiency = 4.2%



(e) 16,384 Processes, 20,480x2,048 matrices. Measured = 1,438 GFlop/s. Peak = 150,733 GFlop/s. Efficiency = 0.9%

Fig. 7. *Effect of scale on matrix multiplication.* As the number of processes increases, so does the cost of setting up the communication structures before actually executing the matrix operation. Overall time for running the program increases with the number of processes and scale of the data, but the multiplication takes less time per element. The efficiency of the system drops substantially with large numbers of processes.

5 Future Work

While the results we achieved were quite effective at the scale we were dealing with, further extension of this work to greater scales could prove challenging. For instance, we currently load the entire log file into memory before visualizing it. Very large log files would need out-of-core access. Support for more log formats would be very beneficial to this end. We support clog and clog2 formats, but extending to slog2 format would aid out-of-core visualization, as it was designed with that intent. Extending the work to include profiling a real simulation would be useful, but the resulting log file would likely be much larger than the ones shown here. This would necessitate not only out-of-core data access, but also a higher level interface than the current timeline, such as one that abstracts the data to the matrix operation level. The data formats we use do not clearly identify the MPI call across processes. If we move to a data format that identifies the calls hierarchically from the function level, the MPI calls could be accurately clustered together, which would allow for a hierarchically based visualization. As our current approach only uses two views, it would be interesting to either add an intermediate level view or a more detailed view based on selections from the MPI call plot. Further understanding could also be achievable by taking into account the topology of the supercomputer itself, or by drilling down to the underlying network traffic. This would allow for detection of network bottlenecks, which our current system cannot explicitly show.

Acknowledgments

This work is supported in part by the National Science Foundation through grants CCF-0938114, CCF-0808896, OCI-0749227, OCI-0749217, CNS-0551727, and CCF-0811422, and the U.S. Department of Energy through the SciDAC program with Agreement No. DE-FC02-06ER25777. This research used resources of the National Energy Research Scientific Computing Center (NERSC) through the DOE SciDAC program.

References

[1] Bootchart, a visualization of system startup processes for optimizing boot times. `http://www.bootchart.org/`.

[2] A. Chan, W. Gropp, and E. Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16(2-3):155–165, 2008.

[3] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Syst. Softw.*, 81(12):2252–2268, 2008.

[4] EtherApe: A graphical network monitor. `http://etherape.sourceforge.net`.

[5] Franklin at NERSC: `http://www.nersc.gov/nusers/systems/franklin/`.

[6] M. T. Heath. Paragraph: a tool for visualizing performance of parallel programs. In *Second Workshop on Environments and Tools for Parallel Sci. Comput*, pages 221–230, 1994.

[7] V. Herrarte and E. Lusk. Studying parallel program behavior with `upshot`. Technical Report ANL–91/15, Argonne National Laboratory, 1991.

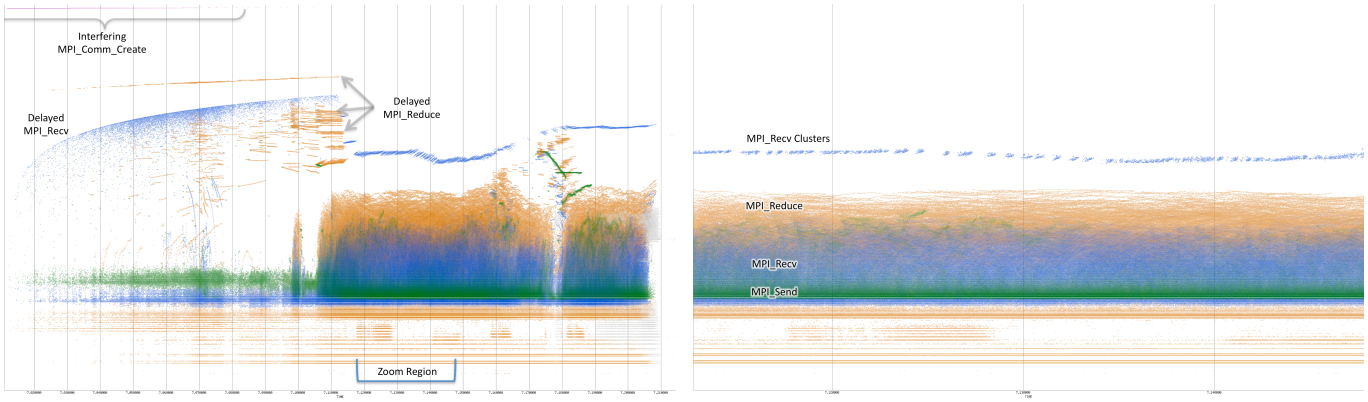
[8] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *ICSE '97: Proc. of the 19th Intl. Conf. on Software engineering*, pages 360–370. ACM, 1997.

[9] J. Johansson, P. Ljung, M. Jern, and M. Cooper. Revealing structure within clustered parallel coordinates displays. In *InfoVis '05: Proc. of the 2005 IEEE Symposium on Information Visualization*, pages 125–132. IEEE Computer Society, 2005.

[10] E. Karrels and E. Lusk. Performance analysis of MPI programs. In J. Dongarra and B. Tourancheau, editors, *Proc. of the Workshop on Environments and Tools For Parallel Scientific Computing*, pages 195–200. SIAM Publications, 1994.

[11] D. Kranzlmueller and J. Volkert. Debugging point-to-point communication in MPI and PVM. *Lecture Notes in Computer Science*, 1497:265–272, 1998.

[12] D. Kranzlmüller, S. Grabner, and J. Volkert. Debugging massively parallel programs with atempt. In *HPCN Europe 1996: Proc. of the Intl. Conf. and Exhibition on High-Performance Computing and Networking*, pages



(a) End points

(b) Zoomed in focus with start points

Fig. 8. *Running on 16,384 processes*: At this scale, the matrix multiplication looks similar to Figure 6(b), with some differences. There is a clear section early in the operation with longer MPI calls caused by interference from the previous operation, as seen in the upper left of (a). When zoomed in further, some detailed trends can be seen in the midst of the ocean of communication, but these clusters are generally small, indicating good division of processes.

806–811, London, UK, 1996. Springer-Verlag.

[13] R. M. Martin, "Electronic Structure. Basic Theory and Practical Methods in Physics" (Cambridge University Press, 2004) Ch. 23.

[14] S. Moreta and A. Telea. Multiscale visualization of dynamic software logs. In *EuroVis*, pages 11–18, 2007.

[15] W. E. Nagel. `http://www.vampir-ng.de/index.html`.

[16] National Center for Supercomputing Applications (NCSA) at University of Illinois at Urbana-Champaign: `http://www.ncsa.uiuc.edu/`.

[17] M. Ogawa and K.-L. Ma. Stargate: A unified, interactive visualization of software projects. In *Proc. of IEEE PacificVis 2008*, pages 191–198, March 2008.

[18] Accelerating Discovery in Science and Engineering through Petascale Simulations and Analysis (PetaApps), the National Science Foundation, `http://www.nsf.gov/pubs/2007/nsf07559/nsf07559.htm`.

[19] J. E. Roberts, "TraceVis: An execution visualization tool," Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, July 2004.

[20] R. Ross, T. Peterka, H.-W. Shen, Y. Hong, K.-L. Ma, H. Yu, and K. Moreland. Visualization and parallel i/o at extreme scale. *Journal of Physics*, July 2008. Proc. of DOE SciDAC 2008 Conf.

[21] Scientific Discovery through Advanced Computing (SciDAC): `http://www.scidac.gov/`.

[22] E. Shaffer, D. Reed, S. Whitmore, and B. Schaeffer. Virtue: performance visualization of parallel and distributed applications. *IEEE Computer*, 32(12):44–51, Dec 1999.

[23] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[24] M.-A. D. Storey, D. Čubranić, and D. M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *SoftVis '05: Proc. of the 2005 ACM symposium on Software visualization*, pages 193–202. ACM, 2005.

[25] A. Telea and L. Voinea. An interactive reverse engineering environment for large-scale c++ code. In *SoftVis '08: Proc. of the 4th ACM symposium on Software visualization*, pages 67–76. ACM, 2008.

[26] Texas Advanced Computing Center (TACC) at University of Texas, Austin: `http://www.tacc.utexas.edu/`.

[27] B. Topol, J. T. Stasko, and V. Sunderam. PVaniM: a tool for visualization in network computing environments. *j-CPE*, 10(14):1197–1222, Dec. 1998.

[28] L. Voinea, A. Telea, and J. J. van Wijk. Ezel: a visual tool for performance assessment of peer-to-peer file-sharing network. In *InfoVis '04: Proc. of the 2004 IEEE Symposium on Information Visualization*, pages 41–48, 2004.

[29] M. Wattenberg. Arc diagrams: Visualizing structure in strings. *InfoVis '02: Proc. of the 2002 IEEE Symposium on Information Visualization*, 0:110, 2002.

[30] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of ACM/IEEE Supercomputing (SC00)*, November 2000.

[31] H. Yu and K.-L. Ma. A study of I/O techniques for parallel visualization. *Journal of Parallel Computing*, 31(2):167–183, Feb 2005.

[32] H. Yu, K.-L. Ma, and J. Welling. A parallel visualization pipeline for terascale earthquake simulations. In *Proc. of ACM/IEEE Supercomputing (SC04)*, Nov 2004.

[33] D. Zeckzer, R. Kalcklösch, L. Schröder, H. Hagen, and T. Klein. Analyzing the reliability of communication between software entities using a 3d visualization of clustered graphs. In *SoftVis '08: Proc. of the 4th ACM symposium on Software visualization*, pages 37–46. ACM, 2008.