Eurographics Conference on Visualization (EuroVis) (2014)
R. Borgo, R. Maciejewski, and I. Viola (Editors)

State of the Art of Performance Visualization

Katherine E. Isaacs[1], Alfredo Giménez[1], Ilir Jusufi[1], Todd Gamblin[2], Abhinav Bhatele[2],

Martin Schulz[2], Bernd Hamann[1], and Peer-Timo Bremer[2]

[1]Department of Computer Science, University of California, Davis
[2]Lawrence Livermore National Laboratory

Abstract

Performance visualization comprises techniques that aid developers and analysts in improving the time and energy efficiency of their software. In this work, we discuss performance as it relates to visualization and survey existing approaches in performance visualization. We present an overview of what types of performance data can be collected and a categorization of the types of goals that performance visualization techniques can address. We develop a taxonomy for the contexts in which different performance visualizations reside and describe the state of the art research pertaining to each. Finally, we discuss unaddressed and future challenges in performance visualization.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

1. Introduction

High performance computing (HPC) simulations drive innovation across a wide range of scientific fields, including astrophysics, climate simulation, material science, combustion, and energy production. Numerical problems in these disciplines would take hundreds of years to compute without massively parallel machines. To shape the development of future fast, power-efficient architectures, and to accelerate the pace of computational science, it is critical to gain a comprehensive understanding of the factors that affect performance and power consumption on HPC systems.

Optimizing the performance of parallel applications is not straightforward, and performance analysis has become increasingly complex. Programs now must take advantage of multicore processors, programmable Graphics Processing Units (GPUs), and multi-level non-uniform memory hierarchies. On-node performance counters and instrumentation tools allow detailed performance measurements, but the profusion of data they generate when applied to parallel programs makes exploring and understanding the data difficult.

This highlights the need for performance visualization techniques. We present an overview of performance visualization and survey existing work. Our contributions are:

- An overview of the available performance data.
- A classification of the goals of developers and analysts who use performance visualization.
- A context-based classification and survey of existing performance visualizations.
- A discussion of challenge areas in developing new and more powerful performance visualizations.

Others [MMC02, KS93] have reviewed software visualization techniques, but tended to lump visualizations focusing on performance into a single category. We focus solely on performance, avoiding other software visualization areas such as software evolution, programming environments, visual programming, and software design. Performance tends to overlap with debugging and general program comprehension, so we include work from those areas as appropriate.

2. Performance Data

We detail methods for acquiring performance data and the types of performance data that can be generated. Many tools can be used to record performance measurements [GKM04, NS07, Rei05, MBDH99, SM06, BM11], allowing visualization developers to gather their own datasets.

2.1. Methods for Acquiring Performance Data

2.1.1. Instrumentation

Instrumentation is the act of modifying a program for an alternative purpose: in this case, for acquiring performance data. At the most basic level, an instrumentation tool inserts extra code into a program's control flow. The instrumentation code may record timer values, or it may perform more complex analysis, such as writing out program variable values and recording variable accesses and conditions met. Instrumentation can be applied to source code before compiling, or it can be applied at runtime using binary modification [BM11] or sampling [ABF*10]. Care must be taken to ensure that the instrumentation does not change normal program behavior or add excessive overhead.

2.1.2. Interception

Interception is a form of instrumentation that leverages function calls already present in program source code. Interceptor functions are typically grouped together into a library, which is then linked with a program, either dynamically or statically. The program's original function calls are linked to the interception library, which executes special measurement code, then delegates to the original implementation of the intercepted function. Interception is useful for profiling libraries because it can record the dynamic values of parameters passed to library calls. This can give more semantic context to measurements. For example, communication performance of many parallel programs is often measured by intercepting calls to the Message Passing Interface (MPI), and interceptor calls can differentiate between send and receive operations based on the size of data passed to them. As with other types of instrumentation, interception must be used sparingly to avoid incurring overhead.

2.1.3. Profiling and Tracing

Profiling and tracing are measurement techniques that determine where a single execution of a program spends its time. Profiling tools, such as gprof [GKM04] and VTune [Rei05], pause execution of a program repeatedly over a specified sampling period and record the contents of either the instruction pointer or the entire call stack. At the end of a profiling run, samples are analyzed to determine the percentage of time spent in each part of the code. Profiles lose temporal information but quickly identify key bottlenecks in a program. Tracing is similar to profiling in that it measures a program's execution, but it records a detailed time line of when events occurred. For example, a trace might record function entry and exit times for an entire run. Because it does not aggregate over time, recorded traces can require large amounts of memory, which can cause excessive overhead compared to profiling. TAU [SM06], Vampir [NAW*96], and EPILOG [WMfAM04] provide resources for creating and dumping traces of programs that use MPI.

Profiling trades off comprehensive data for low overhead, while tracing provides complete data on runtime events at the cost of much higher overhead.

2.2. System Monitoring

The measurements discussed so far are application-level measurements, in that they measure the performance of a single application process. We can also acquire system-wide performance information during a program's execution by executing external processes, enabling system-wide counters (described in Section 2.3.1), or gathering other metadata at runtime. Such methods require no modification to the code or executables involved, and as such are simple to use. However, this kind of data is generally very coarse and semantically low-level. Further, because data is collected outside the measured application process, it can be difficult to attribute measurements to the target program's source code.

2.3. Types of Performance Data

2.3.1. Counters

A counter is a special hardware register that accumulates the number occurrences of a specified event over time. These can be either software events, such as system calls, or hardware events, such as floating-point operations, cache misses, and packets received over a network link. The complete set of countable events is specific to the platform being used but is generally quite extensive. Commonly, counters are either instrumented to initialize and terminate around a block of code designated for analysis or run system-wide during program execution. PAPI [MBDH99] provides a portable interface to specify, initialize, terminate, and read out counters.

The overhead and precision of performance counter measurements depends on how frequently they are sampled. Sampling performance counters too frequently gives high overhead, which can limit precision and make attributing counted events to particular instructions difficult.

Counter data most directly benefits visualizations in the hardware (Section 5) and software (Section 6) contexts. For example, in a network visualization, packet counters can be recorded per-link to visualize network traffic. Other hardware counters can be mapped directly to the resource (CPU, memory, etc.) responsible for generating the event. Counters measured within instrumentation can also be attributed indirectly to the instrumented code, giving software context.

2.3.2. Hardware Samples

Traditional accumulative hardware counters have been extended to provide more precise and detailed information about particular instructions. Instead of simply incrementing a counter, modern hardware performance units can write detailed information about an instruction's execution, including its precise instruction pointer, progress within the processor's pipeline, total latency, and more. Intel and AMD

processor architectures both include hardware capabilities to measure memory loads and stores, and Intel in addition provides a capability to sample branching events [Int07, DC07].

Hardware sampling provides finer granularity with low overhead because it is implemented as part of a microprocessor. Tools still need to conduct detailed analysis to attribute such samples to program source code.

2.3.3. Traces and Call Paths

Trace files contain lists of timestamped point events recorded during program execution. These events can include procedure entry and exit, message sends and receives, and object acquisitions and releases. By following function entries and exits, the call stack at any point in time can be derived. These events may also be associated with certain hardware elements like memory addresses or particular CPUs.

In some parallel environments, there may be one trace file generated for each process or thread, so trace data size typically scales with the number of concurrent tasks. Parallel systems may not guarantee high resolution clock synchronization, resulting in some inaccuracy in event timestamps.

Depending on the features in the tracing tool and the options selected by the user, more or less information can be included – for example, message sizes with the sends and receives or parameters with the procedures. Some tools can also record counter values with each event.

3. Performance Goals

The main goal of performance analysis and thus in performance visualization is to make the application execute faster or use less power. There are several sub-goals on the road to efficiency that have utilized visualization. In this section we discuss these goals, dividing them into three main categories: global comprehension, problem detection, and diagnosis and attribution.

3.1. Global Comprehension

Often the first step in optimizing an application is understanding the big picture regarding what occurred during an execution. When specific targets for optimization are unknown, analysts must narrow down regions of interest from the whole application. Global comprehension goals also exist so users can get a sense of normal behavior as well as compare predicted and achieved performance. Visualizations that present a strong overview or allow for pattern matching may be particularly useful here.

The tasks involve understanding program structures and resource utilization. Program structures include phases of execution, algorithms, data structures, communication patterns, data motion, access patterns, and data dependences. Resource utilization includes the magnitude and distribution of demands on processors, memory, and the network. Understanding the intricate relationshpis between these different aspects of program behavior forms the necessary foundation for identifying and understanding the performance of an execution.

3.2. Problem Detection

Visualization can help developers detect performance problems such as anomalous behavior, performance bottlenecks, load imbalance, and resource usage issues. Outlier detection, pattern detection, focus+context features, and dependency tracking can aid in finding problems.

Anomalous behavior includes deadlocks, livelocks, data race issues, or unexpected behavior. Bottlenecks and imbalance may exhibit similar symptoms like outlier computation or message durations and significant idle times. These problems could also be detected by recognizing network congestion or memory contention or by characteristics of the critical path through the execution.

Resource misuse includes low parallelism and false parallelism, where many threads are created unecessarily. Synchronization may also be unnecessary and impede performance. Another resource issue could be poor locality in data accesses.

3.3. Diagnosis and Attribution

Diagnosis of a problem may follow directly from detection or be more subtle. Problems may be attributed to software, relationships with lines of code, variables, data structures, or third party libraries. This may be a step in the process of recognizing poor distribution or division of work, a sub-optimal algorithm or data structure, or a better overlap of messaging and computation. In distributed and parallel systems, the mapping of tasks to the system can likewise be an issue.

Problems may also be attributed to the system on which the code is run. Operating system effects, memory or scheduling policies, and network routing algorithms may contribute to poor performance. Finding these effects gives developers the information necessary to take any steps they can to ameliorate them.

Highlighting the true sources of inefficiency can be difficult. Linking and correlation, pattern detection, dependency tracking, and ensemble comparison features may aid in achieving these goals.

4. Taxonomy

We organize our survey by the main *context* represented by the visualization. By context, we refer to the concepts onto which the data is mapped and of which the visualization is constructed. In some cases, this context can be derived directly from recorded data, such as a visualization focusing

on a specific data column. In other cases, the context may require some form of additional input about the environment from which the data was collected, such as structure mined from the source code or the graph of a distributed system. Sometimes this information is assumed and hard-coded into the visualization. We define four major contexts in performance visualization: hardware, software, tasks, and application.

Hardware is the natural context for data collected from performance counters, as these are associated with individual hardware elements like nodes, cores, or links. Additional context in hardware includes the hierarchical grouping of the elements, the network topology of these elements, and queues, scheduling and interfaces associated with these elements (even if they may be implemented by low-level software).

Software covers contexts related to a program's source code. This includes static information such as the class structure of the program and individual variables as well as dynamic data associated with executions, such as call graphs, developed data structures, and program flow.

Tasks contexts involve the individual tasks performing the computation. Tasks contexts exist at many levels of granularity: Processing elements, threads, and processes are fractions of a single program. Jobs and commands represent entire programs that may share a system. Note that processing elements form a tasks context when viewed as mostly anonymous actors performing work, but they form a software context when considered as specific (and largely different) objects interacting in an object-oriented program.

Application refers to the context of what is actually being computed. In scientific simulations, this is often bounded physical space. Another common application context is the set of matrices used in matrix libraries.

Some visualizations draw contexts from multiple categories. For example, a tasks layout may be influenced by the underlying topology of the processors on which they run. In these situations, we classify the visualization by the dominant context, but make mention of the additional contexts used. In the specific case of contexts related to the operating system (OS), we generally classify them under hardware with the justification that the OS is typically not programmable to the extent that the software is.

Figure 1 provides a alternative picture, organizing the most recent visualizations by complete tool rather than by individual view as we discuss in the following sections. This shows which tools cover multiple contexts, as well as which goals they address and what sizes of problems they handle.

5. Hardware Visualization

Visualizations in the hardware context create a visual representation of the hardware on which an application code is run. Often these visualizations map data from performance counters onto a depiction of the hardware from which the data originated. Building representations of different hardware requires developing an intuitive metaphor for the topology of the hardware. An effective hardware metaphor decomposes the hardware into its basic elements while retaining its unique characteristics necessary for performance analysis. These techniques aim to illustrate complex hardware topology, identify hardware-based performance problems, and show the relationship between software and hardware.

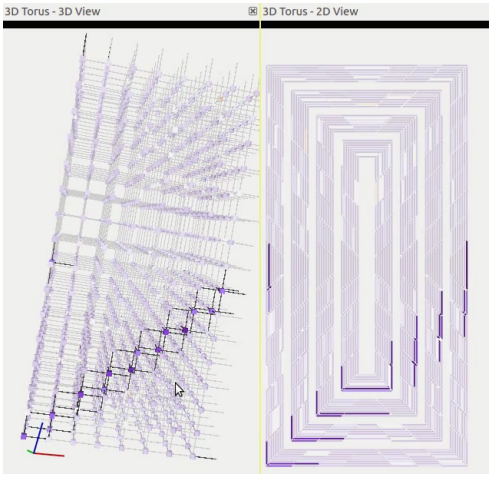


Figure 2: A screenshot of Boxfish [LLB*12, ILG*12] showing a selected set of nodes in a 3-dimensional torus network. Nodes are arranged in a 3-dimensional graph with links denoted by edges (left) and an orthogonal projection shows a subset of the edges without occlusion (right).

We categorize visualizations in the hardware context into those depicting the computing network and those depicting individual compute nodes.

5.1. Network

Supercomputing nodes are connected via a network. The system can be interpreted as a graph where vertices are nodes and edges are network links. The performance visualizations therefore often take the form of graph visualizations. Because network topologies vary so widely in their structures, a challenge in creating network visualizations is that each must be highly tailored to a specific topology. Tree-based networks, such as fat-trees, lend themselves to hierarchical visualizations, while others, such as torus and hypercube networks, lend themselves to complex graph layouts and dimension-reducing projections.

A common general representation of the network graph is an adjacency matrix with computation nodes as the x and y axes sometimes referred to as a communication ma-

		Taxonomy				Demonstrated Scale*		Global Compre.		Problem Detection			Diagnosis/ Attribution	
Visualization Techniques	Papers	H	S	T	A	Data	Parallel	Program Structure	Resource Usage	Anomalies	Bottlenecks and Imbalance	Resource Misuse	Software	System
Radial Tree	Bhatele et al. [BGI*12]			X		NR	10^4 processes	X	X		X			
Node-Link Graph	Boxfish [LLB*12, ILG*12]	X				NR	10^4 nodes		X		X			X
Radial Tree, Animation	Choudhury and Rosen [CR11]	X	X			10^7 transactions	N/A	X	X		X	X	X	
Layered Node-Link	DOTS [BKS05]	X		X		NR	NR	X	X		X	X	X	
Clustered Node-Link, Animation	Frishman et al. [FT05]	X		X		NR	10^2 objects	X	X	X				
Node-Link Graph	Heapvis [AKG*10]		X			10^3 nodes		X	X				X	
Radial Tree	Kim et al. [KLJ07]		X	X		NR	10^3	X		X			X	
Node-Link Trees, Indented trees	Lin et al. [LTOB10]		X			NR	NR	X	X				X	
Node-Link trees	Rose et al. [RHJ07]		X	X		NR	10^2 cores				X		X	
Node-Link graph, Animation	Sambasivan et al. [SSMG13]		X			NR	NR	X					X	
Radial Tree	Sigovan et al. [SMM*13a]	X				10^1 resources	10^3 processes		X	X		X		
Node-Link trees	STAT [AdSL*09]		X	X		NR	10^5 tasks	X			X		X	
Clustered Node-Link, Animation/Real Time	Streamsight [DPA09]		X	X		streaming	10^3 tasks	X	X	X	X			
Layered None-Link	Threadscope [WT10]	X		X		10^3 events	10^1 threads	X	X	X	X			
Node-Link Graph, Treemap	Weidendorfer et al. [WKT04]		X			NR	1	X					X	
Timeline, Stacked Graph, Small Multiples	de Pauw et al. [DPWB13]	X		X		streaming	10^3 tasks	X	X		X		X	
Shared Timeline	Muelder et al. [MGM09]			X		NR	10^4 processes	X			X			
Gantt Charts, Timeline, Matrix, Scatterplot	Muelder et al. [MSM*11]	X	X	X		NR	10^3 cores	X	X		X			
3D Parallel Gantt Chart, Treemap/Force-directed layouts	Triva [SHN10]	X		X		NR	10^3 processes	X	X		X			
Parallel Gantt Chart, Node-Link Tree, Bar Charts	Zinsight [DPH10]		X	X		10^5 events	10^2 processes	X	X		X		X	
1D Color-Coded Array, Histograms	Cheadle and Field [CFA*06]	X	X			10^1 memory groups	N/A	X					X	X
1D Color-Coded Array Stacked By Time	Moreta and Telea [MT07]		X			10^5 allocations	N/A	X				X	X	
Edge Bundling, Gantt Charts, Hierarchies	Extravis [CHZ*07]		X			10^5 events	NR	X					X	
Parallel Gantt Chart, Indented Trees, Code view	HPCToolkit [ABF*10, TMCF*11, LMC13]	X	X	X		10^1 gigabytes	10^4 processes	X	X		X	X	X	X
Stacked Barcharts, Stacked Timelines	Lumière [BBH08]	X	X	X		10^6 decisions	NR	X	X	X	X		X	
Parallel Gantt Chart, Small multiples, Plots, Ensemble	Projections [KZKL06, LMK08]		X	X		gigabytes	10^4 processes	X	X		X	X	X	
Stacked Barcharts, Scatterplot, Histograms, Code Coloring	TraceVis [RZ05]	X	X			10^7 instructions	NR	X	X		X	X	X	
Icicle Timelines, Coordinated views	Trumper et al. [TBD10]		X	X		10^4 events	10^1 threads	X			X	X	X	
Parallel Gantt Chart, Icicle Timeline, Adjacency, Indented Trees, Ensemble Timeline, Plots	Vampir [NAW*96, BW12, ISC*12, VMa13]	X	X	X		terabytes	10^5 processes	X	X		X			
Abstract Diagram	Choudhury et al. [CPP]	X	X			10^1 buffers	N/A				X	X	X	
Dot Plot, Bar Charts	Iviz [WYH10]		X	X		10^6 events	2 jobs	X			X		X	
Scriptable	ParaProf [SML*12]			X		NR	10^4 processes		X		X			
Indented Trees, Matrix	Scalasca [GWW*10, WG11]		X		X	terabytes	10^5 cores	X	X		X		X	
Color-coded 2D matrix, histograms, 3D graph layout	Schulz et al. [SLB*11]	X		X	X	NR	10^4 cores		X	X	X	X		
Bubble Chart, Animation	Sigovan et al. [SMM13]			X		NR	10^4	X	X		X			
City Metaphor	SynchroVis [WWF*13]		X	X		10^2 objects	10^1 threads	X		X			X	
Icicle Timeline, Bundles	SyncTrace [KTD13]		X	X		10^7 events	10^2 threads	X	X		X	X	X	
Sunburst, Matrix, Dendrogram	Trevis [AH10]		X			10^3 nodes	NR	X					X	

Figure 1: Classification of recent visualizations by context, scale, and goal. We limit the scale and goal to what was reported (rounded); in practice the visualization may exceed what is listed in the chart. Any value that was not clear or missing entirely in the publications are marked not reported (NR). We focus on works published in the last 10 years.

trix [HE91]. ParaGraph [HE91] depicts communication matrices and color-codes the elements to indicate areas of heavy link traffic. Zhou and Summers [ZSC03] use an adjacency matrix to show quaternary fat-trees and depict transactions by animating 3-dimensional glyphs on matrix element locations. While communication matrices effectively show all links, they are very ineffective in showing the shape of the network and the distance between non-neighboring nodes. Furthermore, because they show all possible links, and existing links are shown twice (once for each direction), they contain much visual redundancy.

Haynes et al. [HCR01] makes another general representation by depicting all nodes in a 2-dimensional grid and color-coding all network links. As such, the user can follow links by finding matching colors, with no redundant links or wasted space. However, this technique still suffers in showing the network shape and paths with multiple links. Another issue is that humans are visually limited in discerning multiple unique colors with accuracy, which limits the size of the network this visualization can usefully represent.

Zhou and Summers [ZSC03] also use a modified 2-dimensional H-tree layout to demonstrate the network topology of the quaternary fat-tree used in a variety of HPC systems. They aggregate histograms and arcs in a third dimension to show messages passed between nodes. Muelder et al. [MSM*11] demonstrate another hierarchical style graph visualization of the I/O network for Blue Gene/P. They depict the network in a radial layout with storage nodes in the center, compute nodes on the outside, and I/O nodes in between. The performance data is aggregated on the drawn links between the different types of nodes. Both hierarchical visualizations demonstrate the ability to discover areas of heavy communication traffic within their respective network topology. Muelder et al. [MSM*11] depict the entirety of the performance data in a single 2-dimensional view, while Zhou and Summers [ZSC03] take advantage of encoding data in the third dimension and using animation, at the cost of occlusion and complexity. Purely hierarchical approaches are only possible for specific network topologies, but effectively reduce the visual complexity of the network graph by utilizing well-known hierarchical metaphors. We also note that the aforementioned visualizations deal with network trees of relatively shallow depth, but this has not yet been an issue because existing HPC interconnects typically do not use much deeper hierarchies.

Many visualizations lay out the network graph of specific network topologies as 2- or 3-dimensional meshes, with nodes as vertices and links as edges. Boxfish [ILG*12] provides an interface for displaying performance data on a mesh representing a 3-dimensional torus network. Landge et al. [LLB*12] create 2-dimensional projections of the 3-dimensional torus network in Boxfish with no occlusion (Fig. 2). Haynes et al. [HCR01] depicts, in addition to the general 2-dimensional visualizations, another 3-dimensional mesh layout of a 3-dimensional torus network. The mesh layouts create more intuitive depictions of the network, but there does not always exist an intuitive mesh projection for a network topology type. As topologies increase in dimension, such as the 5-dimensional and 6-dimensional torus, it becomes much more difficult to create a low-dimensional mesh that is easily understood.

5.2. Node

The topologies of CPU nodes are often relatively small compared to network topologies; these are on the order of tens and hundreds of processors and memory resources. For this reason, parallel programs usually employ a mapping from N tasks to M processors, with $N \gg M$. Techniques for visualizing on-node computation space often take the form of task-based visualizations [TBD10, KLJ07, dKSB00, Rei05, ABF*10]. However, such a mapping typically does not exist between tasks and memory resources, especially in the context of multi-level memory hierarchies where multiple processors share resources simultaneously. As a result, on-node hardware visualizations have mostly targeted memory address space and resource usage.

5.2.1. Processor Topology

Processor-based visualizations typically visually encode cumulative performance data per-processor. Often, the layout of processors is based on the hardware numbering and data is represented with histograms or stacked bar charts [BD01, ABF*10]. Schulz et al. [SLB*11] arranged processors based on the 2D layout of the application (see Section 8) and displayed values using color.

Processor topologies are often embedded within larger network visualizations to show processor resources within individual nodes. Haynes et al. [HCR01] depicted nodes with aggregated glyphs representing multiple processors on each node. Similarly, Zhou and Summers [ZSC03] showed each node as a subdivided grid with cells representing individual processors.

5.2.2. Memory Topology

Several on-node memory visualizations represent the virtual address space of memory as an infinite one-dimensional space. A program is allocated a finite subset of that space by the operating system, and all program variables lie within it. As such, many techniques [GT89,MT07,CFA*06] depict the space of a single program with variables as finite contiguous blocks within the program's memory. Griswold et al. [GT89] color-code different variables and their datatypes on a line which wraps down multiple rows to more effectively utilize screen space. Moreta and Telea [MT07] expands the 1-dimensional layout to depict allocations and deallocations over time, with the address space on the vertical axis and time on the horizontal axis. They also include an overview

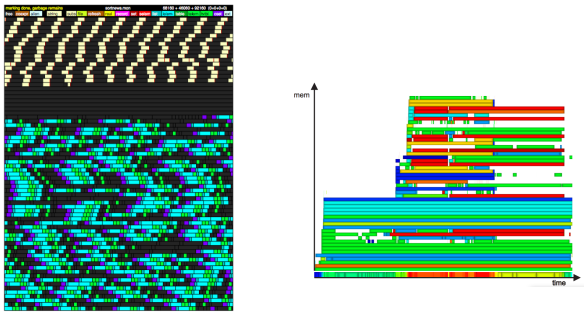


Figure 3: Left: Memory visualization by [GT89]. Memory address space depicted as a 1-dimensional array wrapping down several rows. Blocks indicate allocations along address space, color legend indicates variable data types. Printed with author's permission. Right: Memory visualization by [MT07]. Allocations depicted as 2D blocks with address space on the vertical axis and time on the horizontal axis. Image courtesy of A. Telea.

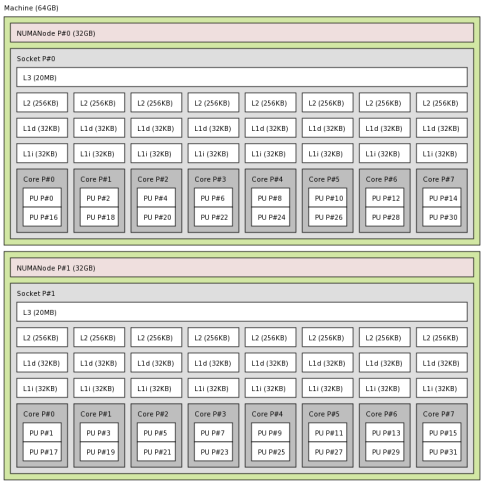


Figure 4: The output of `lstopo`, a tool provided by hwloc [BCOM*10] to visualize hardware topology. The memory hierarchy is shown as a hierarchical space-filling layout, with larger resources at the root and smaller caches and processors at the leaves.

visualization with the address space collapsed, which more clearly shows the order of allocations over time.

All the aforementioned address space-based visualizations target an understanding of allocator behavior and depict fragmentation issues effectively. However, the visualized datasets are often small, especially compared to common HPC applications. As programs utilize more and more of the growing 1-dimensional memory space, we believe the visualizations will not scale well enough to continue being useful.

Other memory visualization techniques have focused on depicting properties of the memory hierarchy, e.g. multi-level caches, RAM, and disk, rather than the address space.

Alpern [ACS90] created an early visualization showing the memory hierarchy of various hardware for the purpose of observing data migration between disk, memory, translational lookaside buffer, and registers. The visualization showed the different memory resources as boxes connected by drawn links and also drew subsets of the data within the memory resource they resided. While it did not embed performance data, it created a model for understanding what occurs in hardware and how cache-optimized algorithms more efficiently utilize memory resources. hwloc [BCOM*10], a software package for the analysis of system attributes, provides a tool called `lstopo` that detects and displays the topology of different architectures in a hierarchical space-filling layout (Figure 4) but like the work of Alpern, also does not encode performance data.

Choudhury et al. [CPP08] created an interactive visualization depicting simulated memory access data embedded within diagrams of the caches, address space, and iterations. This visualization shows accesses and misses from individual cache lines and addresses. While highly detailed, it would be unfeasible to scale beyond the demonstrated number of memory resources. Rivet [BD01] displays another diagram-like visualization of different caches with per-processor memory performance data mapped to cache resources.

Choudhury and Rosen [CR11] created a more abstract representation using a radial space-filling layout, also for simulated data. They represent different levels of cache as rings around a central processor, with lower levels closer to the processor, and depict data migration between levels of cache as lines between ring segments, as seen in Figure 5. Mu et al. [MTSM03] created a visualization targeting NUMA effects on multi-socket nodes by depicting different NUMA domains and transactions between them. The visualization also includes sufficient information to map areas of NUMA transactions to source lines of code. Because they focus on a specific performance issue, the visualization is able to depict a small amount of information in a way that is directly useful in optimizing code for NUMA efficiency.

Rosen [Ros13] specifically targets the memory topology of NVIDIA graphics processors and create a visual model depicting both processor and memory layout. The visualization decomposes performance data of multiple processing units (warps) to find representative subsets with which to compare. The visualized warps include information about memory banks used by each warp for the purpose of identifying bank conflicts, which represent major memory access bottlenecks. The idea to use representative subsets is an effective way to handle the plethora of performance data and the extensively large processor topologies while retaining information about average behavior and outliers.

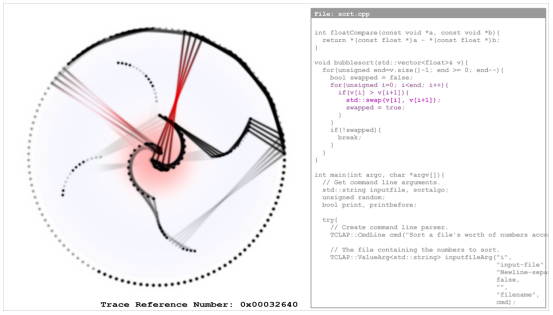


Figure 5: Memory hierarchy visualization by [CR11]. Left: Addresses are represented as points, and different sets of points represent different memory resources. The outer ring represents main memory, the outer four arcs of points represent L2 cache, and the inner two arcs represent L1 cache. Lines between points denote migration of data between resources. Right: Simulated transactions are associated with the lines of code which caused them. Image courtesy of P. Rosen.

6. Software Visualization

We survey software visualization only as it related to performance visualization. Therefore software visualization techniques applied for other purposes, such as education or software maintenance, belong outside of the scope of this work. As mentioned earlier, we define the software context as visualizations related to a program's source code. This includes visualization of the software structures in terms of classes and packages, visualization of the code itself, serial traces of events related to method invocation, and call graphs of specific execution. Data-structure visualization tools are mainly used for education and debugging, although Heapvis [AKG*10] offers features that could be for performance visualization.

6.1. Serial Trace Visualization

Serial trace visualization shows a sequence of events. Several different visualization metaphors have been used for the visualization of traces. One of them is a variation of icicle plots, where width encodes duration [RR99, TBD10, KTD13]. Figure 10 shows one such example; the icicle plot is located at the top of the screenshot.

A common practice in trace visualizations is to assign one of the axes to the time variable while the other axis is used to represent different processes, classes, instructions or methods [JSB97, DPH10, CHZ*07, MHJ91, MSM*11, RZ05]. In essence, most of these approaches represent different variations of Gantt charts. Some trace visualization tools animate the execution of events and even provide additional views for visualizing algorithms [JSB97, BBH08].

In contrast to the presented approaches, Cornelissen et al. [CHZ*07] place the methods in a circular layout while the edges (events) that represent method calls are bundled to avoid clutter (cf. Figure 6). They provide an additional linked view where different methods are placed on top of the view, calls between them are shown with horizontal lines, and time is shown in the y axis. In somewhat similar fashion, De Pauw and Heisig [DPH10] use the vertical axis for encoding time, while the horizontal view encodes different processes. Within each process column, events are represented as blocks color-coded according to different software components. The horizontal position within a process column denotes the load module where the calls were generated.

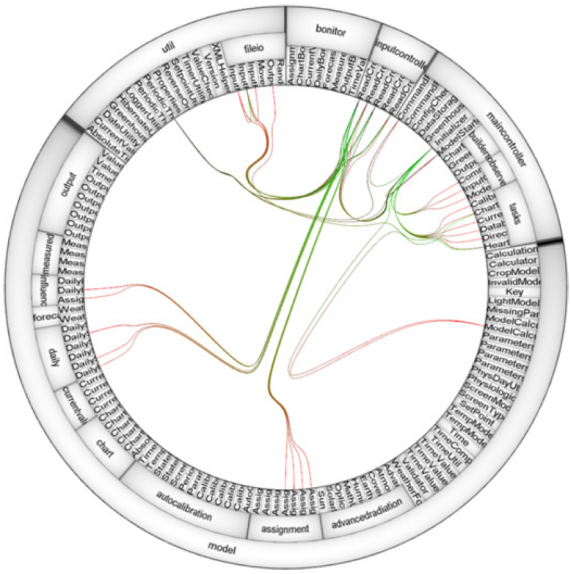


Figure 6: Inner nodes represent different methods. Edges show caller-callee relationships. The edge color gradient can be used to denote the call direction or to show the call time–less recent (light) to most recent (dark). Radial icicle plots show the software structure, which provides insight into how different packages are linked to each other. Image ©2007 IEEE. Reprinted, with permission, from [CHZ*07].

The trace visualizations thus far are usually viewed in fractions of the total duration. Wu et al. [WYH10] creates dot plots of the entire trace versus itself, marking where the events are the same. Additional information is encoded as "bar codes" along the axes. The dot plot shows global patterns along the full timespan of the trace. The same method can be used to compare two different traces. Sambasivan et al. [SSMG13] focus specifically on comparing two request-flow traces with a side-by-side view, difference view, and animation between them.

In order to facilitate the size of data or to gain insight into the possible branching of events, traces are aggregated into call graphs as described in [JSB97].

6.2. Call Graph Visualization

Together with serial trace views, call graph views appear to be one of the most common visualizations in the software context for performance data analysis. In most cases, call graphs are tree structures, such as context call trees that are usually produced by profilers to help understand caller-callee relationships. Here, one should keep in mind not to confuse the the debugging goal with the performance optimization goal. Call graph visualization for performance analysis purposes usually encodes additional performance data in itself or is shown together with other contexts such as tasks or hardware.

The most common representation of call graphs uses the node-link metaphor, where the node is usually a function (method) and the link represents a function call. In this regard, there are several tools that use an indented tree layout for visualizing the call trees [ABF*10, MW03, WG11]. Some of these tools integrate performance data directly into the nodes by color-coding them [MW03]. Others use the horizontal space provided by the indented layout to add tabular data or even small barcharts or histograms. They may also employ computational methods to find hot-paths, at which point the corresponding branch would expand and direct users' attention to the relevant portion of the tree [ABF*10]. However, due to the size and complexity of the call graphs, performance and statistical data is usually visualized using multiple coordinated views.

Other node-link layouts mainly use the conventional tree drawing algorithms [SM06,DPH10,Rei90,DPH10,LTOB10, WKT04, AdSL*09]. Usually some data is visualized using the color, shape or size of the nodes. For instance, DeRose et al. [DHJ07] managed to integrate load balancing data inside the nodes of the call tree by using the width and the height of the nodes as well as by integrating small barcharts inside the nodes.

Space-filling approaches such as treemaps [WKT04] and sunbursts [AH10] have also been used to represent call trees. Additionally, Adamoli et al. [AH10] present a view where a dissimilarity matrix is used to compare several calling context trees.

6.3. Code and Code Structure Visualization

Sometimes it is important to invoke specific lines of code where a potential performance problem is detected. Many tools that employ call graph visualization show the code as well, so that when users click on the specific node in the graph, the corresponding line of code is shown or highlighted in the code view [ABF*10, SG93].

However, there are also approaches that visualize the code and the performance data together. One of the first such examples is the Seesoft tool [ES92]. Here, each line of code is represented by a line of pixels and color-coded according to the number of executions, providing the user with an easy way to notice "hot spots." A similar idea is presented by Liao et al. [LDB*99] where each code character has been encoded into a pixel and the color denotes various cycles in the code. The aim of this visualization is improving parallelization.

There are approaches to show specific parts of code in other contexts as well. TraceVis offers functionality where the user can specify regions of static code [RZ05]. It will then color-code the background regions of the trace view, showing which static code elements map to the dynamic trace data. It is also possible to select a specific region in the dynamic trace view and automatically color-code all static instructions.

In some cases certain structural features of the code, such as class hierarchy, should be analyzed in context of the performance data in order to understand if a potential problem is originating from the application code or an external library. One approach is to visualize software modules or class hierarchy. Icicle plots could be used in this case as well [CHZ*07]. Figure 6 shows the use of icicle plots in a radial layout. SynchroVis shows program traces in the static structure of the program, visualized using a three-dimensional city metaphor [WWF*13]. Here different features of the city are mapped to code structures. For example, districts represent packages while buildings represent classes. This work is conceptually similar to the previous two-dimensional representation approach [JSB97]. One of the most straightforward methods to map different software components or modules in other contexts is color. For instance, different parts of a call tree can be color-coded according to the component they belong to [AH10, LTOB10].

7. Tasks Visualization

The fundamental context required by tasks visualizations are the attribution of the performance data to the abstract actors that generated it. These actors include processes, threads, and jobs. Further context in this area includes the hierarchical structure of the actors (e.g. what threads belong to what process). Some tasks visualizations are able to take advantage of other contexts, such as the specific nodes or sites where the process is being run.

Execution traces and system logs are often recorded with tasks context. These documents capture timestamped events such as function entry, message receives, and job initiation. Traces and logs offer analysts a full record of what occurred, but this increases the difficulty of making sense of them. Ordering of events can unveil bottlenecks, delays, and anomalies. Patterns in utilization and communication can be found over the duration of the data collection. The time component of this data is essential in this analysis, so there are many visualizations that attempt to display the time streams per task. We discuss these in Section 7.1.

However, time is not a necessary component in tasks vi-

sualizations. Sometimes aggregating information over a duration, either from the trace or through profiling, can yield insight into program behavior as well. Non-time tasks visualizations create a snapshot of task interactions over time. These are discussed in Section 7.2.

7.1. Time in Tasks Visualizations

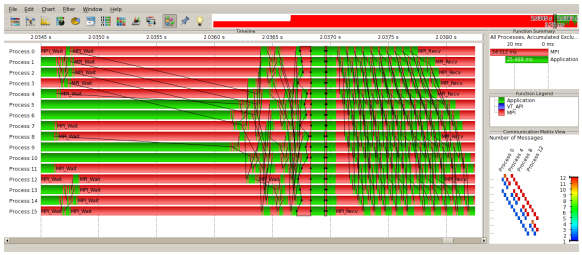


Figure 7: Vampir [NAW*96]. The main visualization is the timeline view. Bar color indicates whether application or MPI functions are active. The black lines indicate messages between processes.

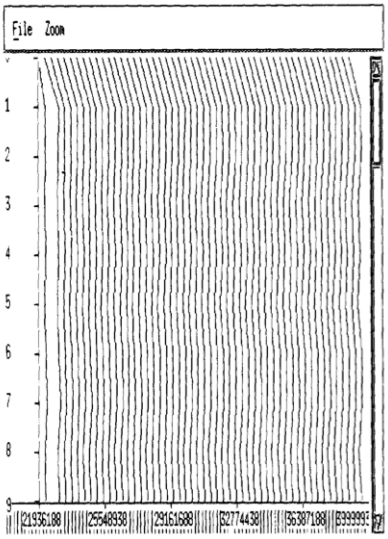


Figure 8: SIEVE contour plot shows event wavefront. Reprinted from [SG93], with permission from Elsevier.

The majority of time-based tasks visualizations assign time to the x or y axis and then constrain the events of each task to bars in a row or column respectively, similar to a Gantt chart. These visualizations generally omit the call stack information found in single task trace visualizations (Section 6) as available space for the parallel tasks is already a challenge. A typical visualization of this type would be Vampir [NAW*96] as shown in Figure 7. We refer to these as timelines and classify them and closely related ones in Section 7.1.1. Other techniques such as animation and sonification are discussed in Section 7.1.2.

7.1.1. Task Timelines

We classify task timeline visualizations by their representation of both time and the relationships between individual timelines. Most visualizations use *physical time* (e.g. wall-clock time, system time, and cycle counts), which is generally what is recorded in traces and logs. However, some visualizations support *logical time*, a partial ordering based on dependency information, often Lamport clocks [Lam78]. Cuny et al. [CHK92] claim that logical time is needed for debugging the correctness of parallel programs, while physical time is more important for performance where the ultimate goal is decreasing the total time required by the program. PARADE [KS98] supports *phase time* which is a partial ordering of phases of an execution rather than individual events. However, computing phases from trace data without extra information is difficult.

Some visualizations show no relationships between timelines [DPH10, SG93, LSV*89, TBD10, Sha90, Rei90]. Zinsight [DPH10] recognizes a hierarchy of tasks and allows users to select which granularity to plot events. While Vampir's default view shows messages between timelines, it also provides cluster timelines which show aggregated events over the cluster with no messages [VMa13].

SIEVE [SG93] draws contour lines across the tasks where the events are equivalent as shown in Figure 8. Muelder et al. [MGM09] show log-scale duration versus time, rather than placing tasks or groups of tasks on the y-axis. Instead, events from all tasks were drawn over the same area and overplotting and blending techniques were used to show consensus (or lack their of) among tasks.

In some cases, intertimeline relationship data may not exist. HPCToolkit [ABF*10] visualizes sampled data rather than full traces. It shows all tasks in an information mural style display, sampling each pixel for its task and sample contributors. Individual tasks can be selected for a detailed single timeline display.

De Pauw et al. [DPWB13], SeeLog [EL96], and `lviz` [WYH10] show separate program instances, which unlike processes do not interact directly. The De Pauw et al. visualization (Figure 9) displays job lifetimes on a shared system in an online stacked graph-like visualization that groups jobs by user. Rather than assigning rows to jobs permanently, De Pauw et al. changes the y value over time so the clusters remain continguous and separate from each other. SeeLog shows classes of applications per row with glyphs indicating how many are active rather than bars. `lviz` visualizes job logs on Microsoft Windows in a dot plot, revealing repeated patterns of jobs over time. The dot plot can be used to compare two separate logs by assigning one log to rows and another to columns.

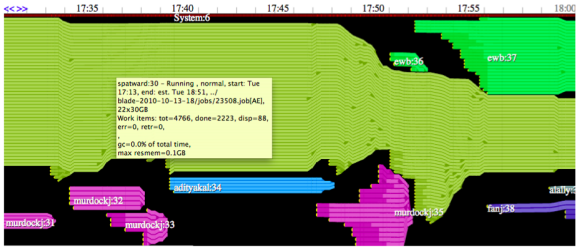


Figure 9: De Pauw et al. timeline of jobs on a shared system. Image ©2013 IEEE. Reprinted, with permission, from [DPWB13].

Timelines may affect each other through dependency constraints such as message sends and receives or access to shared objects. Message dependencies are often shown as a line or arrow from the send on one timeline to the receive on another [YSM95,SKV03,ZLGS99,LMCF90,FB89, dKSB00, KS98, HE91, PLCG95, KZLK06, TSS98, KTD13, SHN10, SRWS99, KTM97, KG96]. Including these types of dependencies makes it possible to highlight critical paths.

Virtue [SRWS99] draws timelines in 3-dimensional space, using a ring layout for tasks rather than an axis. The visualization is also compatiable with a CAVE environment. VisuaLinda [KTM97] and Triva [SHN10] use 3D in order to cluster tasks by their location on physical processors in two of the dimensions.

SyncTrace [KTD13], shown in Figure 10, draws a serial timeline overview for a selected thread and a focus view which draws multiple threads as sectors of a circle. The call stack is maintained for these threads, resulting in a sunburst-like design. Relationships between the focus thread and other threads are drawn as aggregated edges, similar to a chord diagram.

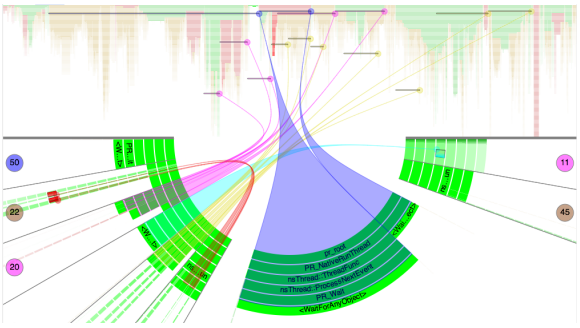


Figure 10: SyncTrace [KTD13] visualization shows thread call stacks and relationships between them. Image courtesy of B. Karran.

In addition to message and shared object dependencies, there may be relationships regarding the lifetime of threads. While these are also dependencies, we separate them because they involve the addition or removal of tasks in the visualization. Many of the visualizations supporting lifetime relationships also support message or shared object dependencies.

DOTS [BKS05] uses a Sugiyama-style layout algorithm to assign threads to columns and route dependency lines. The threads are grouped by processor. ThreadScope [WT10] also uses a layered node-link diagram with line styles representing different relationships and node styles representing both threads and memory. The graph can be condensed through grouping by malloc calls or classes.

Several visualizations represent parent-child relationships among tasks but do not emphasize creation and destruction; instead the space is allocated to the task far past the extent of its lifetime. We do not consider these as showing lifetime relationships because it is not clear if the task is non-existant or just idle. Wang and Kunz [WK00] modify the usual timeline view to show the lifetimes of migratable objects as they move between individual timelines representing machines in distributed systems.

Table 1 organizes the task timeline visualizations by what relationships are present between the individual timelines and what type of time is displayed.

7.1.2. Other Time-based Tasks Visualizations

Several visualizations use animation to represent time, showing the state of the tasks at every instance. VISTOP [BB92] uses a mailbox metaphor to show messaging and semaphore activity and a directory to show thread spawning relationships. SynchroVis [WWF*13] uses a city metaphor to represent the static structure of the program with special buildings where added floors represent thread and shared object creation. Arrows connecting to the special buildings show the evolution of the system in time.

Belvedere [HC88] and its follow-up Ariadne [CFH*93] animate messages between processes in logical time for debugging. Streamsight [DPA09] creates a node-link diagram with processing elements as nodes and streams between them as links. Grouping by job or host makes aggregation and clutter reduction possible. This visualization allows for real-time monitoring but can also be recorded and replayed.

Sigovan et al. [SMM13b] animate events as rising bubbles per process which fade into the background at the end of their duration, creating a contextual history. Using overplotting and blending techniques, this animation is able to scale to 16K processes.

PARADE [KS98] and PVaniM [TSS98] place processes on a circle and animate messages moving between them. Growing Squares [ET03] similarly places processors when animating dependency relationships between processes in logical time. Process squares 'grow' outlines that incorporate the colors of other processes that have causally affected

Relationships	Time	Visualizations
None	Physical	ConcurrencyVisualizer [GN10], De Pauw et al. [DPWB13], Devise [KMLM97], Falcon [GEK*95], HPCToolkit [ABF*10], `lviz` [WYH10], Muelder et al. [MGM09], PIE [LSV*89], Reillly [Rei90], SeeLog [EL96], Sharma [Sha90], SIEVE [SG93], Trümper et al. [TBD10], Vampir [NAW*96], Zinsight [DPH10]
Dependency	Physical	AIMS [YSM95], Jumpshot [ZLGS99], Moviola [LMCF90, FB89], Pajé [dKSB00], PARADE [KS98], ParaGraph [HE91], PARAVER [PLCG95], Projections [KZLK06], PVaniM [TSS98], SyncTrace [KTD13], Triva [SHN10], Virtue [SRWS99], VisuaLinda [KTM97], XPVM [KG96]
	Logical	Concurrency Maps [Sto88], DeWiz [SKV03], Moviola [LMCF90, FB89]
Lifetime	Physical	Wang and Kunz [WK00]
	Logical	DOTS [BKS05], Threadscope [WT10], Zernik et al.' [ZR91, ZSM92]

Table 1: Classification of Tasks Timelines

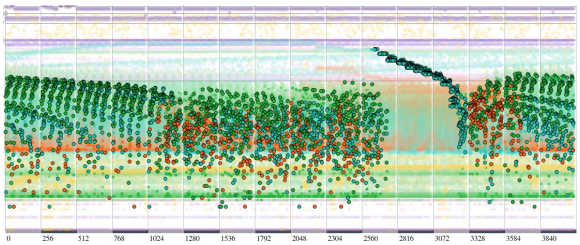


Figure 11: Snapshot from the animated trace visualization by Sigovan et al. [SMM13b]. As events persist, they rise upward logarithmically. Image courtesy of C. Sigovan.

them. Frishman and Tal [FT05] animate migrating object interactions using the Growing Squares technique to show object history. Clusters representing object locations lend their colors to objects that pass through. This visualization reduces clutter by more aggressively aggregating clusters further from the user's focus.

Yamaguchi and Itoh [YI03] animate the moving locations of tasks on hierarchical distributed systems. The hierarchy is shown using nested rectangles. Using the third dimension, they encode other metrics in the height of each process.

Sonification methods have also been used. Francioni et al. [FAJ91] and Pablo [RRA*93] map tasks to separate instruments or tones, having them sound for the duration of particular events or other states of interest (e.g. idleness).

7.2. Visualization of Non-Time Tasks

Several visualizations show the process communication graph, a summary of all messages among processes in some time frame. Adjacency matrices are frequently used [HE91, RRA*93, VMa13]. Bhatele et al. [BGI*12] modify a node-link diagram to aggregate processes with similar delay behavior into arcs.

Kim et al.'s [KLJ07] method represents threads as points on a cone, with the distance from the apex indicating creation depth. Threads can be aggregated to reduce clutter. ParaProf [SML*12] colors tasks by user-chosen metrics and provides a scripting language so the user can decide the task layout.

8. Application Visualization

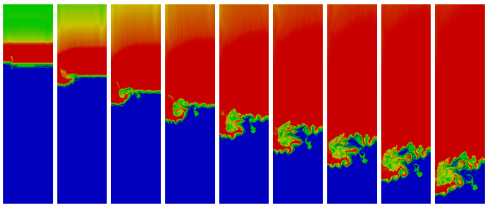
Application contexts are specific to the problems that their target programs are attempting to solve. This generally corresponds to the perceived space of the input and output of the program. For example, the application context of a matrix multiplication program is the space of the matrices involved in the operation.

ParaGraph [HE91] includes facilities to generate performance displays in the application context, noting that such displays could provide new detail and insight, but also mentioning that such displays are highly non-trivial and application specific. They show an example of data transaction counts of a matrix operation overlaid onto the input and output matrices.

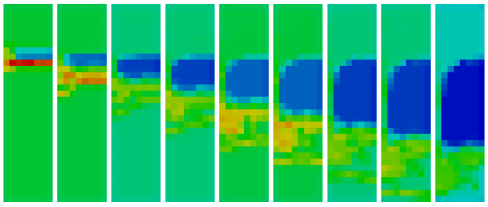
A similar visualization involving parallel prefix sums was shown by Stasko and Kraemer [SK93]. They created an animation showing different processors operating on different parts of the input dataset. This visualization proved useful in debugging parallelism issues in the prefix code and provided a stronger understanding of the utilized parallelism.

Schulz et al. [SLB*11] observed that application developers find the application context highly intuitive. They created visualizations which successfully uncover application-specific performance bottlenecks, by arranging processors and their generated counter data by the physical regions within the bounded fluid dynamics simulation they computed. The visualization showed that areas of high computational and bandwidth costs occurred in areas of high fluid turbulence, as seen in Figure 12. Schulz et al. also observed a need to complement the application context view with other contexts, such as hardware and communication, noting that problems become more obvious when projected into the context from which they originate.

Wylie and Geimer [WG11] likewise created a visualization with processor computation time attributed to physical



(a) Aluminum density visualization over several timesteps.



(b) Floating-point operations mapped to application context over several timesteps

Figure 12: Floating-point operations are highly correlated with aluminum density of their associated application areas. Images ©2011 IEEE. Reprinted, with permission, from [SLB*11].

regions in a large scale reservoir simulation. They also observed application-specific performance bottlenecks where certain areas of the dataset incurred larger computational overhead.

The aforementioned applications generally have intuitive contexts in 2 or 3-dimensional space, but it is a challenge to depict the application context of programs with high-dimensional or abstract output. Furthermore, application visualizations have required significant implementation effort by the analyst, rather than allowing them to leverage existing visualization tools associated with simulation output rather than performance.

9. Challenges

In this section we discuss challenges in performance visualization. Many of the challenges highlight a need for close and continued collaborations with domain experts. The challenges of parallel scale, system complexity, and attribution require expert input to craft useful and informative visualizations. Experts in HPC are not strangers to issues of data scale, and it is highly beneficial to harness this experience in an effort to create scalable and useful visualizations. Finally, the integration of new visualizations with data collection or performance workflows necessitates sustained partnerships between communities.

9.1. Scale

As the scale of computing resources continues to grow exponentially, and along with it, the scale of the collected performance data, it is becoming increasingly critical to create highly scalable performance visualizations. There are two major scale challenges facing performance visualization: parallel scale and data scale. Parallel scale refers to the number of elements required by the context that the visualization is attempting to represent simultaneously. This includes nodes, cores, and memory addresses in the hardware context and tasks, processes, threads, and jobs in the tasks context. Data scale refers to the amount of data collected that need not necessarily be displayed all at once, but must be processed by the visualization. There are several ways to think about data scale – file size, execution time, and total number of samples or events. Figure 1 shows the reported parallel and data scales of the most recent performance visualizations.

As Figure 1 shows, few of the visualizations methods cited demonstrated an ability to handle tens of thousands of simultaneous tasks, and some that do only do so for statistical plots, not for more sophisticated views. Others simply average across pixels, which may hide the insights users seek. At the same time, requiring users to pan extensively within detailed visualizations is not reasonable. While some tools may scale, the utility of the visualization does not. Creating sweet spots between full aggregation and largely unprocessed detail so that the necessary contexts are still shown remains a challenge.

As the size of acquired performance data increases, it has become necessary to scale not only the visualization, but the underlying data. Though this problem has been often neglected by the visualization community, tools developed within the performance community have begun to address the issue. HPCToolkit [TMCF*11] maintains interactivity of its views by sampling the data rather than reprocessing all of it during panning and zooming. It reduces the size of data during collection through sampling as well. Vampir [ISC*12] can utilize the same systems it is meant to analyze, handling terabytes of data via the parallel filesystem and an allotment of processors. It can also employ data reduction techniques during collection. As the data size increases, it may not be feasible to save the entirety of the collected data, so integrating more approaches like the parallel system usage of Vampir or the sampling-based functionality of HPCToolkit is crucial. The greater use of sampling and the effects of overhead and clock synchronization necessitate more techniques for handling uncertainty.

9.2. System Complexity

Many of the continuing challenges in performance visualization are the product of the ever-evolving technology in high performance computing systems. Network topologies

are constantly changing and increasing in dimensionality to improve parallelism and efficiency, and as a result existing techniques may quickly become obsolete. Previous networks had natural embeddings into 2- or 3-dimensional space but this is no longer the case for the largest systems. New layouts are needed that can leverage the inherent structure of these networks and improve developer's understanding of them. Furthermore, as hardware developers expose new ways to capture different performance events, visualizations must adapt to fully utilize the new and changing performance data and contexts.

9.3. Ensemble Runs

Analysts often must make comparisons among different executions of the same application to determine the most likely causes of performance differences or to validate the performance benefits of changes in algorithms or parameters. Few visualizations we surveyed had support for handling ensemble datasets. Those that did were limited to or demonstrated only a few at a time [BW12, WYH10, TMCF*11]. Instead, users generally compare two executions by examining visualizations of each one individually. This is an area where visualization can help reduce the cognitive load on the analyst. Showing differences can be tricky – even in the two-run case there are issues of normalizing metrics and resolving multiple corresponding entities.

9.4. Coordination

The current state of performance visualization software is scattered amongst a variety of tools and techniques serving different purposes. As different techniques are able to accomplish different subsets of the tasks delineated in section 3 and no individual technique accomplishes all of them, this chaotic state is largely unavoidable. While we presented the relevant research in four context categories, we have observed that the distinction between contexts is not always well-defined and as such visualizations need not be constrained to any single one. Many of the studies we have observed have highly validated the usefulness of combining multiple techniques, whether closely tied together in the form of linked views [VMa13], or simply applying different techniques to the same target program [KZLK06]. One of the main challenges in performance visualization therefore is the development of improved integrations of multiple views and performance data in intuitive ways.

9.5. Attribution

While complex visualizations can elucidate novel or interesting performance data, it is important to keep in mind that the visualization has to aid the developer in accomplishing some set of performance goals. Many of the examples in Figure 1 reflect case studies that rely heavily on expertise or insight from the user. Most solutions that handle attribution directly do so at the function call or line-of-code level. The area of performance goals targeted least by the surveyed papers was attributing performance problems to semantically high-level reasons and determining possible avenues of improving the code.

9.6. Evaluation

When dealing with more complex systems and programs, especially in the high-performance computing field, the number of domain experts capable of participating in user studies is small. Therefore, full-fledged usability evaluations or controlled experiments with large numbers of participants are rare (for example [SSMG13]). However, variations of expert evaluations can be performed [TM05] as was done in De Pauw and Heisig [DPH10]. These require a small number of domain and visualization experts, making them more feasible to conduct. The surveyed papers have in general not studied the usability of performance visualization methods and interfaces. Expert evaluations with the inclusion of usability would further help to fill this gap in knowledge.

10. Conclusions

Performance visualization is a growing field which continues to adapt to the growing ecosystem of high performance computing. As supercomputers become more powerful, increasing effort is required to understand how different software is run on such machines and optimize their performance. Rising complexity of systems and performance data collected on them invites the utilization of visualization and analysis tools. Largely driven by necessity, performance visualization presents new and challenging research questions, many of which remain to be answered.

We have presented a survey of existing approaches in performance visualization. The current work has been organized based on the primary contexts in which the data has been visualized. Moreover, we have presented and categorized the goals that domain experts seek to address through visualization. Finally, we have discussed the existing challenges in this domain. This survey should act as an introduction to the state of the art for information visualization experts seeking to apply their knowledge to new domains. It also may aid HPC professionals in exploring new tools and methods to analyze their data.

11. Acknowledgements

We thank the participants of the Dagstuhl Perspectives Workshop 14022 "Connecting Performance Analysis and Visualization to Advance Extreme Scale Computing" for their constructive discussions that inspired parts of this paper.

This work supported in part by the University of California Laboratory Fees Research Grant Program and the

Department of Energy Office of Science Graduate Fellowship Program (DOE SCGF) administered by ORISE-ORAU under contract no. DE-AC05-06OR23100. This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-652873.

References

[ABF*10] ADHIANTO L., BANERJEE S., FAGAN M., KRENTEL M., MARIN G., MELLOR-CRUMMEY J., TALLENT N. R.: Hpctoolkit: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience 22*, 6 (2010), 685–701. doi:10.1002/cpe.1553. 2, 6, 9, 10, 12

[ACS90] ALPERN B., CARTER L., SELKER T.: Visualizing computer memory architectures. In *Proceedings of the 1st Conference on Visualization '90* (Los Alamitos, CA, USA, 1990), VIS '90, IEEE Computer Society Press, pp. 107–113. doi:10.1109/VISUAL.1990.146371. 7

[AdSL*09] AHN D. H., DE SUPINSKI B. R., LAGUNA I., LEE G. L., LIBLIT B., MILLER B. P., SCHULZ M.: Scalable temporal order analysis for large scale debugging. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 44:1–44:11. doi:10.1145/1654059.1654104. 9

[AH10] ADAMOLI A., HAUSWIRTH M.: Trevis: A context tree visualization & analysis framework and its use for classifying performance failure reports. In *Proceedings of the 5th International Symposium on Software Visualization* (New York, NY, USA, 2010), SOFTVIS '10, ACM, pp. 73–82. doi:10.1145/1879211.1879224. 9

[AKG*10] AFTANDILIAN E. E., KELLEY S., GRAMAZIO C., RICCI N., SU S. L., GUYER S. Z.: Heapviz: Interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th International Symposium on Software Visualization* (New York, NY, USA, 2010), SOFTVIS '10, ACM, pp. 53–62. doi:10.1145/1879211.1879222. 8

[BB92] BEMMERL T., BRAUN P.: Visualization of message passing parallel programs. In *Parallel Processing: CONPAR 92-VAPP V*, Bougé L., Cosnard M., Robert Y., Trystram D., (Eds.), vol. 634 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1992, pp. 79–90. doi:10.1007/3-540-55895-0_400. 11

[BBH08] BERNARDIN T., BUDGE B. C., HAMANN B.: Stacked-widget visualization of scheduling-based algorithms. In *Proceedings of the 4th ACM Symposium on Software Visualization* (New York, NY, USA, 2008), SoftVis '08, ACM, pp. 165–174. doi:10.1145/1409720.1409746. 8

[BCOM*10] BROQUEDIS F., CLET-ORTEGA J., MOREAUD S., FURMENTO N., GOGLIN B., MERCIER G., THIBAULT S., NAMYST R.: hwloc: A generic framework for managing hardware affinities in hpc applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on* (Feb 2010), pp. 180–186. doi:10.1109/PDP.2010.67. 7

[BD01] BOSCH R., DEPT S. U. C. S.: *Using visualization to understand the behavior of computer systems*. Stanford University, 2001. 6, 7, 14

[BGI*12] BHATELE A., GAMBLIN T., ISAACS K. E., GUNNEY B. T. N., SCHULZ M., BREMER P.-T., HAMANN B.: Novel views of performance data to analyze large-scale adaptive applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 31:1–31:11. doi:10.1109/SC.2012.80. 12

[BKS05] BLOCHINGER W., KAUFMANN M., SIEBENHALLER M.: Visualizing structural properties of irregular parallel computations. In *Proceedings of the 2005 ACM Symposium on Software Visualization* (New York, NY, USA, 2005), SoftVis '05, ACM, pp. 125–134. doi:10.1145/1056018.1056036. 11, 12

[BM11] BERNAT A. R., MILLER B. P.: Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools* (New York, NY, USA, 2011), PASTE '11, ACM, pp. 9–16. doi:10.1145/2024569.2024572. 1, 2

[BW12] BRUNST H., WEBER M.: Custom hot spot analysis of hpc software with the vampir performance tool suite. In *Parallel Tools Workshop* (2012), pp. 95–114. doi:10.1007/978-3-642-37349-7_7. 14

[CFA*06] CHEADLE A. M., FIELD A. J., AYRES J. W., DUNN N., HAYDEN R. A., NYSTROM-PERSSON J.: Visualising dynamic memory allocators. In *Proceedings of the 5th International Symposium on Memory Management* (New York, NY, USA, 2006), ISMM '06, ACM, pp. 115–125. doi:10.1145/1133956.1133972. 6

[CFH*93] CUNY J., FORMAN G., HOUGH A., KUNDU J., LIN C., SNYDER L., STEMPLE D.: The Ariadne debugger: Scalable application of event-based abstraction. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging* (New York, NY, USA, 1993), PADD '93, ACM, pp. 85–95. doi:10.1145/174266.174276. 11

[CHK92] CUNY J., HOUGH A., KUNDU J.: Logical time in visualizations produced by parallel programs. In *Visualization, 1992. Visualization '92, Proceedings., IEEE Conference on* (1992), pp. 186–193. doi:10.1109/VISUAL.1992.235209. 10

[CHZ*07] CORNELISSEN B., HOLTEN D., ZAIDMAN A., MOONEN L., VAN WIJK J. J., VAN DEURSEN A.: Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th IEEE International Conference on Program Comprehension* (Washington, DC, USA, 2007), ICPC '07, IEEE Computer Society, pp. 49–58. doi:10.1109/ICPC.2007.39. 8, 9

[CPP08] CHOUDHURY A. I., POTTER K. C., PARKER S. G.: Interactive visualization for memory reference traces. *Computer Graphics Forum 27*, 3 (May 2008), 815–822. doi:10.1111/j.1467-8659.2008.01212.x. 7, 14

[CR11] CHOUDHURY A. N. M. I., ROSEN P.: Abstract visualization of runtime memory behavior. In *Proc. 6th IEEE Int. Workshop on Visualizing Software for Understanding and Analysis* (2011). doi:10.1109/VISSOF.2011.6069452. 7, 8

[DC07] DRONGOWSKI P. J., CENTER B. D.: Instruction-based sampling: A new performance analysis technique for amd family 10h processors. *Advanced Micro Devices, Inc* (2007). 3

[DHJ07] DEROSE L., HOMER B., JOHNSON D.: Detecting application load imbalance on high end massively parallel systems. In *Euro-Par* (2007), Kermarrec A.-M., Bougé L., Priol T., (Eds.), vol. 4641 of *Lecture Notes in Computer Science*, Springer, pp. 150–159. doi:10.1007/978-3-540-74466-5_17. 9

[dKSB00] DE KERGOMMEAUX J. C., STEIN B., BERNARD P.: Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing 26*, 10 (2000), 1253 – 1274. doi:10.1016/S0167-8191(00)00010-7. 6, 11, 12

[DPA09] DE PAUW W., ANDRADE H.: Visualizing large-scale streaming applications. *Information Visualization 8*, 2 (2009), 87–106. doi:10.1057/ivs.2009.5. 11

[DPH10] DE PAUW W., HEISIG S.: Zinsight: A visual and analytic environment for exploring large event traces. In *Proceedings of the 5th International Symposium on Software Visualization* (New York, NY, USA, 2010), SOFTVIS '10, ACM, pp. 143–152. doi:10.1145/1879211.1879233. 8, 9, 10, 12, 14

[DPWB13] DE PAUW W., WOLF J. L., BALMIN A.: Visualizing jobs with shared resources in distributed environments. In *VISSOFT* (2013), Telea A., Kerren A., Marcus A., (Eds.), IEEE, pp. 1–10. doi:10.1109/VISSOFT.2013.6650535. 10, 11, 12

[EL96] EICK S. G., LUCAS P. J.: Displaying trace files. *Softw. Pract. Exper. 26*, 4 (Apr. 1996), 399–409. doi:10.1002/(SICI)1097-024X(199604)26:4<399::AID-SPE8>3.0.CO;2-J. 10, 12

[ES92] EICK S. G., STEFFEN J. L.: Visualizing code profiling line oriented statistics. In *Proceedings of the 3rd Conference on Visualization '92* (Los Alamitos, CA, USA, 1992), VIS '92, IEEE Computer Society Press, pp. 210–217. doi:10.1109/VISUAL.1992.235206. 9

[ET03] ELMQVIST N., TSIGAS P.: Growing squares: Animated visualization of causal relations. In *Proceedings of the 2003 ACM Symposium on Software Visualization* (New York, NY, USA, 2003), SoftVis '03, ACM, pp. 17–ff. doi:10.1145/774833.774836. 12

[FAJ91] FRANCIONI J. M., ALBRIGHT L., JACKSON J. A.: Debugging parallel programs using sound. *SIGPLAN Not. 26*, 12 (Dec. 1991), 68–75. doi:10.1145/127695.122765. 12

[FB89] FOWLER R., BELLA I.: *The programmer's guide to Moviola: An interactive execution history browser*. Tech. rep., DTIC Document, 1989. 11, 12

[FT05] FRISHMAN Y., TAL A.: Visualization of mobile object environments. In *Proceedings of the 2005 ACM Symposium on Software Visualization* (New York, NY, USA, 2005), SoftVis '05, ACM, pp. 145–154. doi:10.1145/1056018.1056038. 12

[GEK*95] GU W., EISENHAUER G., KRAEMER E., SCHWAN K., STASKO J., VETTER J., MALLAVARUPU N.: Falcon: online monitoring and steering of large-scale parallel programs. In *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers '95., Fifth Symposium on the* (1995), pp. 422–429. doi:10.1109/FMPC.1995.380483. 12

[GKM04] GRAHAM S. L., KESSLER P. B., MCKUSICK M. K.: gprof: A call graph execution profiler. *SIGPLAN Not. 39*, 4 (Apr. 2004), 49–57. doi:10.1145/989393.989401. 1, 2

[GN10] GEORGE B., NAGPAL P.: *Optimizing Parallel Applications Using Concurrency Visualizer: A Case Study*. Tech. rep., 2010. 12

[GT89] GRISWOLD R., TOWNSEND G.: *The Visualization of Dynamic Memory Management in the Icon Programming Language*. Tech. Rep. TR 89-30, Department of Computer Science. University of Arizona, December 1989. 6, 7

[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*, 6 (Apr. 2010), 702–719. doi:10.1002/cpe.v22:6. 14

[HC88] HOUGH A. A., CUNY J. E.: Initial experiences with a pattern-oriented parallel debugger. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging* (New York, NY, USA, 1988), PADD '88, ACM, pp. 195–205. doi:10.1145/68210.69234. 11

[HCR01] HAYNES R., CROSSNO P., RUSSELL E.: A visualization tool for analyzing cluster performance data. In *Cluster Computing, 2001. Proceedings. 2001 IEEE International Conference on* (2001), pp. 295–302. doi:10.1109/CLUSTR.2001.959990. 6

[HE91] HEATH M., ETHERIDGE J.: Visualizing the performance of parallel programs. *Software, IEEE 8*, 5 (1991), 29–39. doi:10.1109/52.84214. 4, 6, 11, 12

[ILG*12] ISAACS K. E., LANDGE A. G., GAMBLIN T., BREMER P.-T., PASCUCCI V., HAMANN B.: Abstract: Exploring performance data with Boxfish. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis* (Washington, DC, USA, 2012), SCC '12, IEEE Computer Society, pp. 1380–1381. doi:10.1109/SC.Companion.2012.202. 4, 6

[Int07] INTEL: *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*. Intel Corporation, August 2007. 3

[ISC*12] ILSCHE T., SCHUCHART J., COPE J., KIMPE D., JONES T., KNÜPFER A., ISKRA K., ROSS R., NAGEL W. E., POOLE S.: Enabling event tracing at leadership-class scale through i/o forwarding middleware. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2012), HPDC '12, ACM, pp. 49–60. doi:10.1145/2287076.2287085. 13

[JSB97] JERDING D. F., STASKO J. T., BALL T.: Visualizing interactions in program executions. In *Proceedings of the 19th International Conference on Software Engineering* (New York, NY, USA, 1997), ICSE '97, ACM, pp. 360–370. doi:10.1145/253228.253356. 8, 9

[KG96] KOHL J. A., GEIST G.: The pvm 3.4 tracing facility and xpvm 1.1. In *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on,* (1996), vol. 1, IEEE, pp. 290–299. doi:10.1109/HICSS.1996.495474. 11, 12

[KLJ07] KIM Y.-J., LIM J.-S., JUN Y.-K.: Scalable thread visualization for debugging data races in openmp programs. In *Proceedings of the 2nd International Conference on Advances in Grid and Pervasive Computing* (Berlin, Heidelberg, 2007), GPC'07, Springer-Verlag, pp. 310–321. doi:10.1007/978-3-540-72360-8_27. 6, 12

[KMLM97] KARAVANIC K. L., MYLLYMAKI J., LIVNY M., MILLER B. P.: Integrated visualization of parallel program performance data. *Parallel Comput. 23*, 1-2 (Apr. 1997), 181–198. doi:10.1016/S0167-8191(96)00104-4. 12

[KS93] KRAEMER E., STASKO J. T.: The visualization of parallel systems: An overview. *J. Parallel Distrib. Comput. 18*, 2 (June 1993), 105–117. doi:10.1006/jpdc.1993.1050. 1

[KS98] KRAEMER E., STASKO J. T.: Creating an accurate portrayal of concurrent executions. *IEEE Concurrency 6*, 1 (Jan. 1998), 36–46. doi:10.1109/4434.656778. 10, 11, 12

[KTD13] KARRAN B., TRÃIJMPER J., DÃŰLLNER J.: Synctrace: Visual thread-interplay analysis. In *Proceedings (electronic) of the 1st Working Conference on Software Visualization (VISSOFT)* (2013), IEEE Computer Society, p. 10. doi:10.1109/VISSOFT.2013.6650534. 8, 11, 12

[KTM97] KOIKE H., TAKADA T., MASUI T.: Visualinda: a framework for visualizing parallel linda programs. In *Visual Languages, 1997. Proceedings. 1997 IEEE Symposium on* (1997), pp. 174–178. doi:10.1109/VL.1997.626578. 11, 12

[KZLK06] KALE L. V., ZHENG G., LEE C. W., KUMAR S.: Scaling applications to massively parallel machines using Projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance*

Modeling and Analysis (February 2006), vol. 22, pp. 347–358. doi:10.1016/j.future.2004.11.020. 11, 12, 14

[Lam78] LAMPORT L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565. doi:10.1145/359545.359563. 10

[LDB*99] LIAO S.-W., DIWAN A., BOSCH JR. R. P., GHULOUM A., LAM M. S.: Suif explorer: An interactive and interprocedural parallelizer. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 1999), PPoPP '99, ACM, pp. 37–48. doi:10.1145/301104.301108. 9

[LLB*12] LANDGE A., LEVINE J., BHATELE A., ISAACS K., GAMBLIN T., SCHULZ M., LANGER S., BREMER P.-T., PASCUCCI V.: Visualizing network traffic to understand the performance of massively parallel simulations. *Visualization and Computer Graphics, IEEE Transactions on 18*, 12 (2012), 2467–2476. doi:10.1109/TVCG.2012.286. 4, 6

[LMC13] LIU X., MELLOR-CRUMMEY J.: A data-centric profiler for parallel programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 28:1–28:12. doi:10.1145/2503210.2503297. 14

[LMCF90] LEBLANC T. J., MELLOR-CRUMMEY J. M., FOWLER R. J.: Analyzing parallel program executions using multiple views. *J. Parallel Distrib. Comput. 9*, 2 (June 1990), 203–217. doi:10.1016/0743-7315(90)90046-R. 11, 12

[LMK08] LEE C. W., MENDES C., KALÉ L. V.: Towards Scalable Performance Analysis and Visualization through Data Reduction. In *13th International Workshop on High-Level Parallel Programming Models and Supportive Environments* (Miami, Florida, USA, April 2008). 14

[LSV*89] LEHR T., SEGALL Z., VRSALOVIC D. F., CAPLAN E., CHUNG A. L., FINEMAN C. E.: Visualizing performance debugging. *Computer 22*, 10 (Oct. 1989), 38–51. doi:10.1109/2.42013. 10, 12

[LTOB10] LIN S., TAÏANI F., ORMEROD T. C., BALL L. J.: Towards anomaly comprehension: Using structural compression to navigate profiling call-trees. In *Proceedings of the 5th International Symposium on Software Visualization* (New York, NY, USA, 2010), SOFTVIS '10, ACM, pp. 103–112. doi:10.1145/1879211.1879228. 9

[MBDH99] MUCCI P. J., BROWNE S., DEANE C., HO G.: PAPI: A portable interface to hardware performance counters. In *Proc. Department of Defense HPCMP User Group Conference* (June 1999). 1, 2

[MGM09] MUELDER C., GYGI F., MA K.-L.: Visual analysis of inter-process communication for large-scale parallel computing. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1129–1136. doi:10.1109/TVCG.2009.196. 10, 12

[MHJ91] MALONY A., HAMMERSLAG D., JABLONOWSKI D.: Traceview: a trace visualization tool. *Software, IEEE 8*, 5 (1991), 19–28. doi:10.1109/52.84213. 8

[MMC02] MALETIC J., MARCUS A., COLLARD M.: A task oriented view of software visualization. In *Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on* (2002), pp. 32–40. doi:10.1109/VISSOF.2002.1019792. 1

[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* (New York, NY, USA, 2011), LSAP '11, ACM, pp. 19–26. doi:10.1145/1996029.1996036. 6, 8

[MT07] MORETA S., TELEA A.: Visualizing dynamic memory allocations. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on* (2007), pp. 31–38. doi:10.1109/VISSOF.2007.4290697. 6, 7

[MTSM03] MU T., TAO J., SCHULZ M., MCKEE S. A.: Interactive locality optimization on numa architectures. In *Proceedings of the 2003 ACM Symposium on Software Visualization* (New York, NY, USA, 2003), SoftVis '03, ACM, pp. 133–ff. doi:10.1145/774833.774853. 7

[MW03] MOHR B., WOLF F.: Kojak - a tool set for automatic performance analysis of parallel programs. In *Euro-Par 2003 Parallel Processing*, Kosch H., BÃűszÃűrmÃľnyi L., Hellwagner H., (Eds.), vol. 2790 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 1301–1304. doi:10.1007/978-3-540-45209-6_177. 9

[NAW*96] NAGEL W. E., ARNOLD A., WEBER M., HOPPE H. C., SOLCHENBACH K.: VAMPIR: Visualization and analysis of MPI resources. *Supercomputer 12*, 1 (1996), 69–80. 2, 10, 12

[NS07] NETHERCOTE N., SEWARD J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100. doi:10.1145/1250734.1250746. 1

[PLCG95] PILLET V., LABARTA J., CORTES T., GIRONA S.: Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments* (1995), vol. 44, pp. 17–31. 11, 12

[Rei90] REILLY M.: Presentation tools for performance visualization: the m31 instrumentation experience. In *System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on* (1990), vol. i, pp. 307–313 vol.1. doi:10.1109/HICSS.1990.205129. 9, 10, 12

[Rei05] REINDERS J.: *VTune Performance Analyzer Essentials: Measurement and Tuning Techniques for Software Developers*. Engineer to Engineer Series. Intel Press, 2005. 1, 2, 6

[Ros13] ROSEN P.: A visual approach to investigating shared and global memory behavior of CUDA kernels. In *Computer Graphics Forum (EuroVis)* (2013), vol. 32. doi:10.1111/cgf.12103. 7

[RR99] RENIERIS M., REISS S. P.: ALMOST: Exploring program traces. In *1999 Workshop on New Paradigms in Information Visualization adn Manipulation* (199), ACM, pp. 70–77. doi:10.1145/331770.331788. 8

[RRA*93] REED D. A., ROTH P., AYDT R. A., SHIELDS K., TAVERA L., NOE R., SCHWARTZ B.: Scalable performance analysis: The pablo performance analysis environment. In *Scalable Parallel Libraries Conference, 1993., Proceedings of the* (1993), IEEE, pp. 104–113. doi:10.1109/SPLC.1993.365577. 12

[RZ05] ROBERTS J., ZILLES C.: Tracevis: An execution trace visualization tool. In *MoBS '05* (2005). 8, 9

[SG93] SARUKKAI S. R., GANNON D.: SIEVE: A performance debugging environment for parallel programs. *J. Parallel Distrib. Comput. 18*, 2 (June 1993), 147–168. doi:10.1006/jpdc.1993.1053. 9, 10, 12

[Sha90] SHARMA S.: *Real-time visualization of concurrent processes*. Springer, 1990. doi:10.1007/3-540-53065-7_160. 10, 12

[SHN10] SCHNORR L. M., HUARD G., NAVAUX P. O.: Triva: Interactive 3d visualization for performance analysis of parallel applications. *Future Generation Computer Systems 26*, 3 (2010), 348 – 358. doi:http://dx.doi.org/10.1016/j.future.2009.10.006. 11, 12

[SK93] STASKO J. T., KRAEMER E.: A methodology for building application-specific visualizations of parallel programs. *J. Parallel Distrib. Comput. 18*, 2 (June 1993), 258–264. doi:10.1006/jpdc.1993.1062. 12

[SKV03] SCHAUBSCHLÄGER C., KRANZLMÜLLER D., VOLKERT J.: Event-based program analysis with DeWiz. In *Proceedings of the Fifth International Workshop on Automated Debugging AADEBUG2003* (2003). 11, 12

[SLB*11] SCHULZ M., LEVINE J., BREMER P.-T., GAMBLIN T., PASCUCCI V.: Interpreting performance data across intuitive domains. In *Parallel Processing (ICPP), 2011 International Conference on* (2011), pp. 206–215. doi:10.1109/ICPP.2011.60. 6, 12, 13

[SM06] SHENDE S., MALONY A. D.: The TAU parallel performance system. *International Journal of High Performance Computing Applications 20*, 2 (2006), 287–311. doi:10.1177/1094342006064482. 1, 2, 9

[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* (Berlin, Heidelberg, 2012), Euro-Par '11, Springer-Verlag, pp. 156–165. doi:10.1007/978-3-642-29740-3_19. 12

[SMM*13a] SIGOVAN C., MUELDER C., MA K.-L., COPE J., ISKRA K., ROSS R.: A visual network analysis method for large-scale parallel i/o systems. *Parallel and Distributed Processing Symposium, International 0* (2013), 308–319. doi:http://doi.ieeecomputersociety.org/10.1109/IPDPS.2013.96. 14

[SMM13b] SIGOVAN C., MUELDER C. W., MA K.-L.: Visualizing large-scale parallel communication traces using a particle animation technique. *Computer Graphics Forum 32*, 3pt2 (2013), 141–150. doi:10.1111/cgf.12101. 11

[SRWS99] SHAFFER E., REED D., WHITMORE S., SCHAEFFER B.: Virtue: performance visualization of parallel and distributed applications. *Computer 32*, 12 (1999), 44–51. doi:10.1109/2.809250. 11, 12

[SSMG13] SAMBASIVAN R., SHAFER I., MAZUREK M., GANGER G.: Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *Visualization and Computer Graphics, IEEE Transactions on 19*, 12 (2013), 2466–2475. doi:10.1109/TVCG.2013.233. 8, 14

[Sto88] STONE J. M.: A graphical representation of concurrent processes. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging* (New York, NY, USA, 1988), PADD '88, ACM, pp. 226–235. doi:10.1145/68210.69237. 12

[TBD10] TRÜMPER J., BOHNET J., DÖLLNER J.: Understanding complex multithreaded software systems by using trace visualization. In *Proceedings of the 5th International Symposium on Software Visualization* (New York, NY, USA, 2010), SOFTVIS '10, ACM, pp. 133–142. doi:10.1145/1879211.1879232. 6, 8, 10, 12

[TKS01] TAO J., KARL W., SCHULZ M.: Visualizing the memory access behavior of shared memory applications on numa architectures. In *In Proceedings of the 2001 International Conference on Computational Science (ICCS), volume 2074 of LNCS* (2001), pp. 861–870. doi:10.1007/3-540-45718-6_91. 14

[TM05] TORY M., MOLLER T.: Evaluating visualizations: Do expert reviews work? *IEEE Comput. Graph. Appl. 25*, 5 (Sept. 2005), 8–11. doi:10.1109/MCG.2005.102. 14

[TMCF*11] TALLENT N. R., MELLOR-CRUMMEY J., FRANCO M., LANDRUM R., ADHIANTO L.: Scalable fine-grained call path tracing. In *Proceedings of the International Conference on Supercomputing* (New York, NY, USA, 2011), ICS '11, ACM, pp. 63–74. doi:10.1145/1995896.1995908. 13, 14

[TSS98] TOPOL B., STASKO J. T., SUNDERAM V.: Pvanim: a tool for visualization in network computing environments. *Concurrency: Practice and Experience 10*, 14 (1998), 1197–1222. doi:10.1002/(SICI)1096-9128(19981210)10:14<1197::AID-CPE364>3.0.CO;2-O. 11, 12

[VMa13] Manual - Vampir 8.2. http:www.vampir.eu, November 2013. 10, 12, 14

[WG11] WYLIE B. J. N., GEIMER M.: Large-scale performance analysis of PFLOTRAN with Scalasca. In *Proc. of the 53rd Cray User Group meeting, Fairbanks, AK, USA* (May 2011), Cray User Group Inc. 9, 12

[WK00] WANG Y., KUNZ T.: Visualizing mobile agent executions. In *Proceedings of the Second International Workshop on Mobile Agents for Telecommunication Applications* (London, UK, UK, 2000), MATA '00, Springer-Verlag, pp. 103–114. doi:10.1007/3-540-45391-1_8. 11, 12

[WKT04] WEIDENDORFER J., KOWARSCHIK M., TRINITIS C.: A tool suite for simulation based analysis of memory access behavior. In *Computational Science - ICCS 2004*, Bubak M., Albada G., Sloot P., Dongarra J., (Eds.), vol. 3038 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 440–447. doi:10.1007/978-3-540-24688-6_58. 9

[WMfAM04] WOLF F., MOHR B., FÜR ANGEWANDTE MATHEMATIK Z.: *EPILOG Binary Trace-data Format*. FZJ-ZAM, 2004. 2

[WT10] WHEELER K. B., THAIN D.: Visualizing massively multithreaded applications with threadscope. *Concurr. Comput. : Pract. Exper. 22*, 1 (Jan. 2010), 45–67. doi:10.1002/cpe.v22:1. 11, 12

[WWF*13] WALLER J., WULF C., FITTKAU F., DÖHRING P., HASSELBRING W.: Synchrovis: 3d visualization of monitoring traces in the city metaphor for analyzing concurrency. In *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)* (September 2013). doi:10.1109/VISSOFT.2013.6650520. 9, 11

[WYH10] WU Y., YAP R. H., HALIM F.: Visualizing windows system traces. In *Proceedings of the 5th International Symposium on Software Visualization* (New York, NY, USA, 2010), SOFTVIS '10, ACM, pp. 123–132. doi:10.1145/1879211.1879231. 8, 10, 12, 14

[YI03] YAMAGUCHI Y., ITOH T.: Visualization of distributed processes using "data jewelry box" algorithm. In *Computer Graphics International, 2003. Proceedings* (2003), pp. 162–169. doi:10.1109/CGI.2003.1214461. 12

[YSM95] YAN J., SARUKKAI S., MEHRA P.: Performance measurement, visualization and modeling of parallel and distributed programs using the aims toolkit. *Software: Practice and Experience 25*, 4 (1995), 429–461. doi:10.1002/spe.4380250406. 11, 12

[ZLGS99] ZAKI O., LUSK E., GROPP W., SWIDER D.: Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications 13*, 2 (Fall 1999), 277–288. doi:10.1109/ipdps.2008.4536187. 11, 12

[ZR91] ZERNIK D., RUDOLPH L.: Animating work and time for debugging parallel programs foundation and experience. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging* (New York, NY, USA, 1991), PADD '91, ACM, pp. 46–56. doi:10.1145/122759.122763. 12

[ZSC03] ZHOU C., SUMMERS K. L., CAUDELL T. P.: Graph visualization for the analysis of the structure and dynamics of extreme-scale supercomputers. In *Proceedings of the 2003 ACM Symposium on Software Visualization* (New York, NY, USA, 2003), SoftVis '03, ACM, pp. 143–149. doi:10.1145/774833.774854. 6

[ZSM92] ZERNIK D., SNIR M., MALKI D.: Using visualization tools to understand concurrency. *Software, IEEE 9*, 3 (1992), 87–92. doi:10.1109/52.136185. 12

Biography

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

Alfredo Giménez is a third year PhD student at the University of California at Davis. His research focus is in instrumentation and information visualization for performance analysis on HPC systems. He received a B.S. in Computer Science at the University of California at Davis in 2010 and worked for 2 years developing performance optimization tools for graphics hardware at Intel Corporation.

Ilir Jusufi is a Postdoctoral Scholar at the University of California, Davis. His research focuses on visualization of performance analysis data for HPC. He received a B.S. in Computer Science at the South East European University in Macedonia and a M.S. in Computer Science at the Växjö university in Sweden. He earned his Ph.D. degree at the Linnaeus University in Sweden focusing on the visualization and interaction techniques of multivariate networks.

Todd Gamblin is a computer scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research focuses mainly on scalable algorithms for measuring, analyzing, and visualizing performance data from massively parallel applications. He is also interested in fault tolerance, resilience, MPI, and parallel programming models. Todd has been at LLNL since 2008.

Todd works closely with researchers in CASC and with staff in the Development Environment Group in Livermore Computing. He is the team leader for the Performance Analysis and Visualization at Exascale (PAVE) project, and he also works on the Exascale Computing Technologies LDRD project, the SciDAC Sustained Performance, Energy, and Resilience (SUPER) project, and many other ASC projects at LLNL.

Todd received the Ph.D. and M.S. degrees in Computer Science from the University of North Carolina at Chapel Hill in 2009 and 2005. He received his B.A. in Computer Science and Japanese from Williams College in 2002. He has also worked as a software developer in Tokyo and held graduate research internships at the University of Tokyo and IBM Research.

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

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

Martin Schulz is a Computer Scientist at the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL). He earned his Doctorate in Computer Science in 2001 from the Technische UniversitÃd't MÃijnchen (Munich, Germany) and also holds a Master of Science in Computer Science from the University of Illinois at Urbana Champaign. He has published over 150 peer-reviewed papers. He is the PI for the Office of Science X-Stack project "Performance Insights for Programmers and Exascale Runtimes" (PIPER) and for the ASC/CCE project on OpenlSpeedShop. Further, he is the chair of the MPI forum, the standardization body for the Message Passing Interface, and is involved in the DOE/Office of Science Exascale Projects CESAR ExMatEx, and ARGO. Martin's research interests include parallel and distributed architectures and applications; performance monitoring, modeling and analysis; memory system optimization; parallel programming paradigms; tool support for parallel programming; power efficiency for parallel systems; optimizing parallel and distributed I/O; and fault tolerance at the application and system level. In his position at LLNL he especially focuses on the issue of scalability for parallel applications, code correctness tools, and parallel performance analyzers as well as scalable tool infrastructures to support these efforts.

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

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