In Situ Visualization of Performance Data for High-Performance Computing Applications

Dewi Yokelson
Computer Science, University of Oregon

***Abstract*—Performance visualization of high-performance computing (HPC) codes, while complex, can be one of the most useful tools in analysis. Moving the visualization earlier in a workflow can save scientists enormous amounts of time and money when they are optimizing and running their experiments on expensive and in-demand, HPC resources. This paper surveys current performance visualization capabilities and challenges, and analyzes the possibilities of taking a more *in situ* approach. This approach would boost efficiency and enable visualizations that might otherwise require prohibitively large amounts of data, thus incurring too much I/O overhead. One potential enabler for this approach is the application of machine learning, such as, using trained models to predict results, or generate visualizations from smaller samples of performance data, before the application has finished running. We look at current work in this area, as well as the neighboring fields of scientific visualization, information visualization, and and relevant subsets of machine learning.**

I. Introduction

HPC systems have been driving scientific discovery in many domains, yet utilizing them as efficiently as possible still poses a major challenge. The benefits of running simulations on HPC systems are countless. They have made it possible to model extremely intricate scientific systems that require a large volume of data, complex calculations, and high precision. It is necessary to expose, and use the parallelism in HPC applications in order to process these volumes of data quickly. Scaling these simulations across multiple nodes, each of which contain many cores can become difficult quickly, even for experts in the HPC domain.

Effectively using the necessary tools and programming models such as CUDA [1], OpenMP [2], and MPI [3], presents developers with additional challenges. This is especially true when these are used in conjunction with highly optimized for parallel performance math and modeling libraries, e.g., BLAS [4], LAPACK [5], and FFTW [6]. The steps to compile and marry together different versions and implementations of these libraries with the best thread and rank count can create an prohibitively large search space for optimal parameters.

In order to achieve optimal wall clock time for the application, it is important to monitor and analyze the performance, which adds yet another layer of data complexity. For an HPC application, this performance is a crucial consideration. Without sufficient performance of the application, both time and money are wasted on expensive resources. However, it can be difficult to understand the performance of such applications and systems. There are many aspects to this, including: understanding what the current application performance is, figuring out a realistic performance benchmark to achieve, the optimal parameters, and diagnosing reasons for poor performance so as to improve it.

One such way to help communicate these concepts and metrics is via data visualizations of performance data gathered from many of the different measurement applications. There are many tools that profile, measure, and can describe the performance of an application or of specific hardware. However, performance data in HPC is notoriously difficult to manage and make sense of because it is complex, and high in volume. Despite these challenges, performance visualization has come a long way in the last decade, including many new tools and novel graphs that help scientists on this path.

Performance visualization can be organized into a taxonomy comprising four *contexts*, as shown by K.E. Isaacs et al.: hardware, software, tasks, and application [7]. Hardware covers physical structures of the hardware and their performance. The software category encompasses the source code, or application that is being run. Tasks are similar to software but with the source code context removed, still actions that are happening, but without any relation to how the code was written. The application context contains things like linear algebra calculations or the computational aspects of the program. Isaacs points out that some visualizations are characterized by more than one of these contexts.

The contexts provided by Isaacs et al. help us to understand what kinds of performance visualizations already exist, what they do well, and where there are opportunities for improvement. As a general example, some of these visualizations, especially of the hardware nature, rely on reducing the dimensionality of the data in order to present something easily digested by the reader. As problem spaces get larger, with increasingly complex code on larger clusters with more cores per node, reducing the dimensionality proves to be a bigger challenge [7]. We center our work on these premises in order to discuss new methods for analyzing performance data and generating hardware-specific visualizations.

Visualizations are critical in presenting performance data in an understandable format. Yet knowing what data to present, and how is a challenge in and of itself. As the volume of performance data grows, there is an increasing need to be able to do more with less data. As one example, if a full simulation takes days to run, it can be unreasonable to do many runs to tune parameters, i.e., different libraries, number of ranks, threads. The ability to know if we have better performance earlier in the simulation would speed up productivity immensely. Another example is when a simulation is run on hundreds of nodes, with hundreds of processes and

threads, and per rank, per thread metrics are gathered. This can actually create I/O or data storage problems, especially if analysis is done *post hoc* (after completion). Thus being able to conduct *in situ* (during execution) performance analysis becomes necessary.

In this paper, we contribute discussion on the following recent advances in the field of performance visualization:

1) Successful performance visualization concepts, common challenges, and how they have been managed.
2) An analysis of many of the current tools and their "readiness" for *in situ* performance visualization.
3) Current *in situ* and *post hoc* performance prediction and problem diagnosis techniques and what opportunities have arisen there.
4) A look into the fields of scientific visualization and information visualization for inspiration.
5) Suggested opportunities for applying machine learning techniques to solve some of the common issues and better enable *in situ* performance visualizations.

Further research opportunities are also discussed in terms of the following challenges that are related to implementing *in situ* performance analysis in table I.

TABLE I
FOUR OF THE UNIVERSAL CHALLENGES ASSOCIATED WITH IMPLEMENTING IN SITU PERFORMANCE VISUALIZATION

Challenge	Description
(A) Superfluous Data	A direct effect of the high-dimensionality problem, managing large amounts of data and visualizing only items of interest
(B) Incomplete Data	Because a program has not completed, not all the possible data will have been collected, could lead to incorrect conclusions
(C) Distributed Data	The parallel nature of applications means that performance data could be distributed across resources and require synchronization
(D) Limited Resources	Collecting performance metrics already increases overhead, adding visualization generation can exacerbate this issue

Here we list the layout of this paper for ease of reading. Section II begins with background information on performance data. In section III we discuss the state of the art of performance visualization. This includes the current performance visualization capabilities and concepts, and which have proved most effective. This is followed by a discussion of the biggest challenges and opportunities in performance visualization, especially in relation to moving towards increasingly *in situ* methods. Section V takes a look into the neighboring domains of scientific visualization and information visualization and how they approach *in situ*, or exploratory, data visualization.

Finally, in section VI we take a look at the different machine learning projects that are, or can be, applied to the HPC domain. This includes current research in the HPC domain for performance diagnostics and improvement using machine learning techniques. It also includes discussion of using machine learning for generating big data visualizations. We conclude with discussion about reasons for success and failure of certain projects, and where we see the most potential for future work. Included here is an analysis of likely challenges and some mitigation strategies.

II. Performance Data and Visualization Background

The following sections provide some background on how performance data of HPC codes is gathered and structured for use. The structure of the data and it's many possible dimensions is one of the largest contributing factors to the challenge of visualization.

A. Dimensions

The high-dimensional nature of performance data is one of the challenges to creating 2d, 3d or even 4d (showing a change over time) visualizations. When thinking about the environment that HPC codes run in, we can quickly see why. First, there can be the code logging events and gathering timing data for specific functions to complete. Then this code could be multi-threaded or multi-processed, with numerous threads or processes executing simultaneously. It could be a hybrid application with multiple processes and multiple threads per process. This could then be running across many compute nodes. Parallel-enabling programs such as MPI and CUDA have their own overhead and sometimes functions that should be tracked for timing data. Then there are the hardware metrics that can be tracked, such as the memory bandwidth during an application run. The data can quickly become unwieldy, for a program running on multiple compute nodes, for multiple days, with multiple ranks and threads.

B. Profiles

A large subset of performance analysis of HPC codes is done with the use of profile data. Profiles are essentially metadata about the code running that gets written out during runtime (dynamic). This metadata usually includes information like the function name, current line in the code, and performance information such as thread ID that can be used to determine how long certain sections of code take to run, among other things. These profiles are multi-dimensional data files, often containing information per thread, or node, etc. They can be tricky to parse into a format that cooperates with non HPC specific visualization tools.

Profile data can be gathered either through instrumentation or sampling. Instrumentation of the code is a more involved method that requires recompiling the code - the tool used to instrument will insert code that writes out performance data periodically. This is the most comprehensive method, but it can often be time-consuming to implement and may not be necessary if sample data is sufficient. Sampling is when the tool interrupts the HPC application periodically to query for the metadata mentioned above. Sampling usually requires less set up but offers the programmer less control over when and how the application is monitored. For a large application that runs for a significant period of time, sampling can often be sufficient as it averages out to being quite accurate over many timesteps.

C. Trace Data

The second category of performance data that is widely used in analysis and visualization tools is trace data. Traces can

include the same data as a profile, but contain more detail, most notably with the addition of a temporal dimension. It can be thought of as a detailed log of events happening while the codes run. Trace data can be generated at many different levels including: the application, the compute node, the rank or process, and the thread. While this level of detail can be extremely useful, it can also quickly become unwieldy.

Because of this complexity and potential volume of trace data as well as the wide array of tools that use it. A standardization project has been in place for some time. The current standard format for trace data is give by Open Trace Format 2 (OTF2) [8]. This built on the predecessor Open Trace Format (OTF) [9], [10] and the EPILOG format [11]. The main improvements were adding more flexibility between different use cases and additional scalability.

D. Important Performance Metrics

One of the most common metrics that is used when analyzing performance of an application is the wall clock time. This is simply the full length of time that it takes for the application to run from start to finish. While this measurement is an excellent indicator of the performance of the system, and is often the main target for decreasing, it lacks the nuance that may be required to gain additional performance. To identify these further opportunities, function or kernel specific data, or hardware specific data is more helpful. For example, knowing how long a program spent executing one specific loop can give insight into how it could be optimized.

Hardware performance counters are registers that keep track of the count of certain events (e.g., number of cache misses). These events imply how efficiently code is executed on the hardware, and can be gathered by accessing API's like PAPI [12], [13]. Efforts to standardize this access across tools have been taken on with projects such as Score-P [14]. Examples of tools used to gather hardware counter information include: TAU [15], HPCToolkit [16], Caliper [17], Survey [18], [19], and Likwid [20].

III. Performance Visualizations and Their Effectiveness

This section encompasses the current work and effectiveness within the field of performance visualization. First, we discuss what is currently being visualized, and at what granularity. Second, what makes these visualizations effective, including examples of some such successful ones. Following this is a deep dive into the cache-aware roofline model, one of the most successful performance visualization concepts.

A. Current Performance Visualization

Current capabilities for performance visualization span across many scopes, including, the actual code performance, single-node hardware performance (FLOP rates and memory bandwidth), distributed memory system performance (multiple node), and the relationship between some of these. Table II categorizes the different capabilities of many of the existing tools. From an application/software perspective, we can visualize something as high level as the differences between wall clock times, or as granular as how long a loop takes to execute. Other concepts that are often visualized are a full call graph of the application, and time spent in each function over the course of an application run, denoted in the "Profile" column of table II. Table II also specifies in the tools that do trace visualization, which is simultaneously the most useful data due to its detail, but the least user-friendly due to its volume. This is discussed in much further detail within the next section, section III-B.

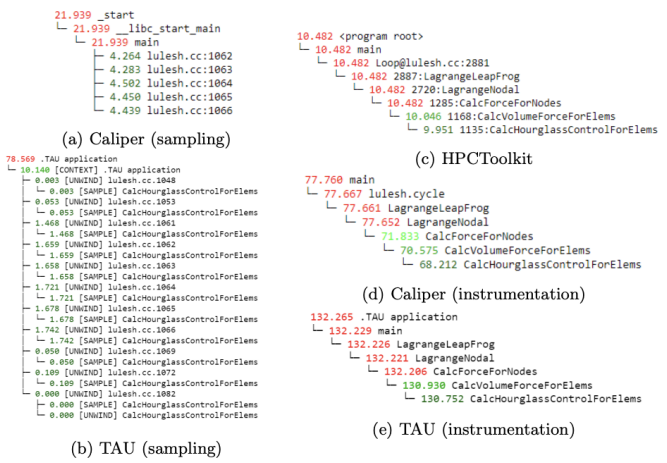

Fig. 1. An example of four different call graphs generated by the Hatchet tool for data gathered for the same code, CalcHourglassControlForElems, by three heavily used analysis tools, Caliper, HPCToolkit, and TAU [21]. There are notable differences between the clarity of the call graphs with data obtained through instrumentation.

Call graphs and calling context trees are the two most common formats for structured profile data acquired through instrumentation or sampling. Call graphs provide a more static view of any interrelated functions (that call each other), disregarding how the application currently got to the specific function sampled, see Figure 1. Calling context trees show a more dynamic view by using the stack trace to expose to the programmer what functions called each other to get to the current state in this run of the application. Both are extremely important ways to structure the data as the call graphs are more straightforward but the calling context tree provides more detail and potentially important context about the application.

Another concept that researchers have been visualizing is memory bandwidth, as the majority of applications are memory bound [22]. Two concepts to visualize here are the hardware capacity (maximum), and how well the application is utilizing the full bandwidth available. MemAxes is a tool developed to target specifically the memory domain of performance data [23]. Their approach takes into account source code, data structures, and hardware. They create interactive visualizations to communicate potential bottlenecks or issues with this complex relationship. Their sunburst-like visualizations are quite unique, which can be helpful if it can communicate well, yet require the consumer to adapt to a new visualization type.

While they may represent the complex structures of memory, they are not that intuitive since they are an uncommon

TABLE II
VISUALIZATION CAPABILITIES OF SOME PERFORMANCE VISUALIZATION TOOLS

Tool	Trace		Profile		Architecture	Distributed	CPU		GPU		Portable
	Thread	Rank	Instrumentation	Sampling	(Roofline)	Memory	Loop	Function	Loop	Function	
Intel Advisor					X		X	X			
Intel VTune	X	X	X	X							
NSight Compute					X				X	X	
NSight Systems	X	X		X	X						
Apprentice2	X	X		X							
Paraprof	X	X	X	X				X			X
Perfetto	X	X	X	X			X	X			X
Chrome Tracing	X	X	X	X							X
Hpcviewer	X	X		X			X	X			X
Traveler/Ravel	X	X						X			X
ERT					X						X
Boxfish						X		X			X
Vampir	X	X	X	X		X		X			X
ParaGraph	X	X				X					X
Jumpshot	X	X				X					X
TRIVA	X	X				X					X
Hatchet			X	X				X			X
CallFlow			X	X				X			X
VIPACT			X	X				X			X
Grafana	X	X	X								X

type of chart. With extended exposure to this tool, we can see how it could be quite useful. However, this is another example of a performance visualization project that did not have a significant impact in the industry. While out-of-the-box thinking like this is a good step in the right direction and demonstrates a need for a vast overhaul of how these concepts are communicated, we see that it must be balanced with ease of utility.

Hardware performance visualization is often encompassed by showing the peak performance of the compute node or cluster. The cache-aware roofline model, which is discussed in detail in section III-C displays both the theoretical maximum memory bandwidth of the hardware system (per cache level), and the theoretical maximum FLOP rates. It also allows for incorporating a visual of the actual bandwidth and FLOP rates (arithmetic intensity) achieved by the software for a full application, function, or loop. Another method of hardware performance visualization that is common to see is the stacked bar chart displaying the Top-Down metrics in conjunction with Top-Down Analysis. This method allows users to visualize at a high level and dive into more detail on hotspots [24]. This method was developed at Intel and has been incorporated into Intel VTune [25].

B. Effective Performance Visualization

For a performance visualization to be effective, it needs to communicate some result about the software or hardware to the viewer. The less time the viewer needs to confirm a hypothesis, spot a trend, or identify a problem, the better the visualization. This is, of course, subjective to a degree, as different humans find different graphs easier to understand. However, the most successful and popular approaches are that way for a reason, heavily adopted for their usefulness.

For the complex nature of performance data, the ability to flatten the data into something human digestible is a necessity. Heatmaps that display data on a per-thread or per-process basis are one effective approach. With a different color per procedure, a user can immediately see which procedures are taking the longest, and if there is inconsitency between processes or threads. Many tools employ this technique, including but not limited to HPCToolkit [16], TAU [15], and Grafana [26].

One of the major issues facing the visualization of software trace data today is that they do not scale well with the amount of data collected, and for HPC applications, visualizations need to scale to be useful. To address this issue, *Ravel*, structures the typical trace data, into that of *logical time* [27] based on the happened-before relation intorduced by Lamport [28]. *Logical time* in this context means looking at relationships between events, and the order in which they occur, rather than when events occur on a physical timeline. It takes the code structure into account, as well as making grouping of events easier. More details about the specific implementation, and topics like *lateness* can be read in section 3 of their paper [27]. Through preserving important physical timing data and different applications of clustering data and color coding, they create visualizations that are more easily understood, even at a higher processor count.

Ravel presents four views for the user, *logical time*, clustered *logical time*, classic physical time, and metric overview, see figure 2. The case studies of *Ravel* show improvement in visualizing high processor counts over the similar case examples generated by other tools. Some issues were also found by using *Ravel* that had previously been unknown by the developers [27]. This new concept proves promising, it is clear that visualizing performance data requires a paradigm shift such as this. More recent projects reference this work [29] and look into the problem of scaling performance visualizations with processor count. However, whether these visualizations are understandable enough to become mainstream, and widely adopted by modern tools is yet to be seen.

Another tool Isaacs has been working on is Traveler, as the performance visualization slice of the JetLag interactive com-

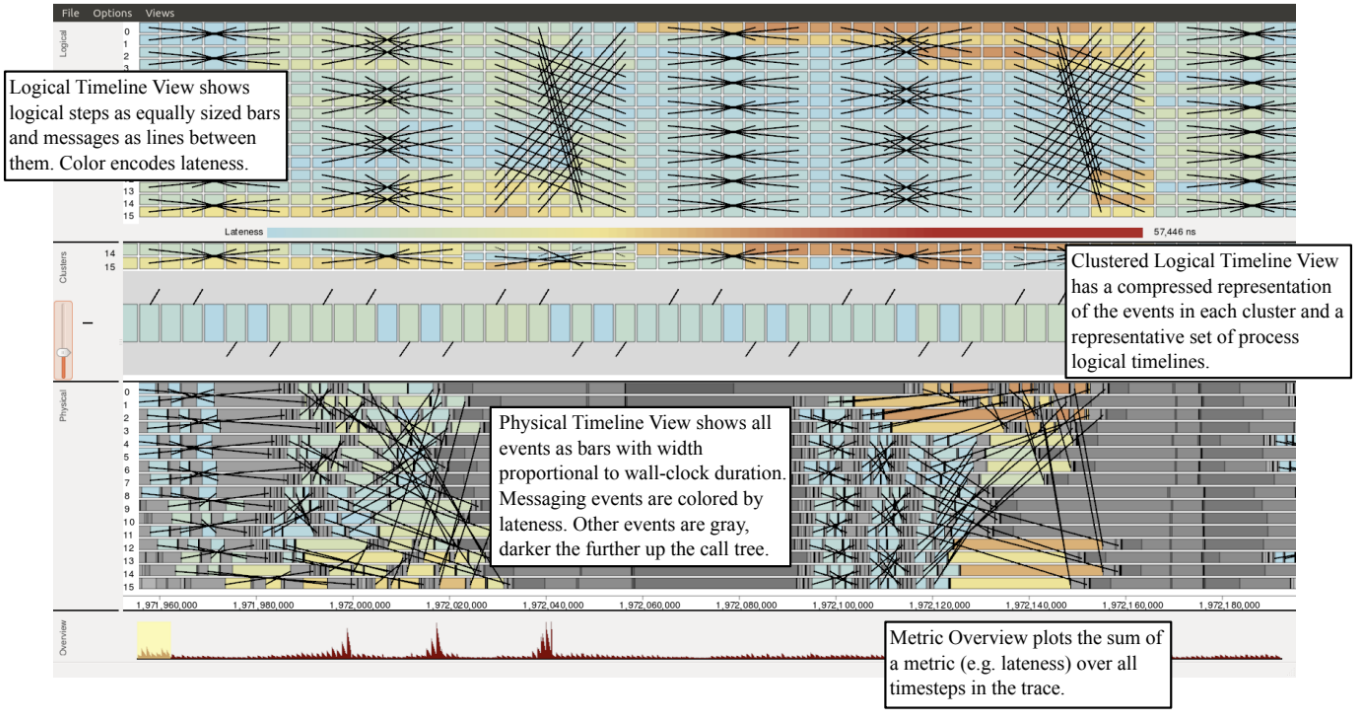

Fig. 2. An example of the trace visualizations generated by Ravel, each row represents a thread and the color represents a procedure. [27]. By nature, trace data can get unwieldy quickly, and Ravel was built specifically to tackle the challenge A (scalability) in table I could be a particular challenge when visualizing trace data like this.

puting system [30]. This environment contains a python asynchronous array library, the APEX performance measurement capabilities, and a jupyter notebook interface, among other components. The goal of the JetLag system is to provide a development environment where the user only completes tasks on their local computer, and heavier computations are sent to HPC resources an back as needed, but in the background. The result is more supportive of the exploratory nature of data analysis, and allows for a more seamless experience.

HPCToolkit is a popular and relatively widely used across domains and platforms. HPCToolkit gathers performance metrics with only a few percent overhead [16]. The HPCToolkit workflow includes measurement, analysis, correlation, and presentation. Presentation in HPCToolkit is comprised of two tools, hpcviewer and hpctraceview. Hpcviewer is simply a user interface for displaying profile data, it allows the programmer to view their callpath data. It can help with digging into problem areas in the code, and navigating through the call path in a few different views. Hpctraceview creates visualizations of how a parallel execution unfolds over time using asynchronous sampling. An example visualization they create is where each line in the chart represents a thread, and each color represents a different procedure. This view helps a programmer identify patterns based on color and size and helps communicate said patterns for this data which varies in both time and space.

TAU and TAU Commander, are widely-used tools built by ParaTools that can generate profile and trace data for applications on many different platforms. TAU can be complicated to build and configure correctly for an application and on specific hardware, TAU Commander was introduced as the solution for these challenges. It works much faster out of the box by adding some user friendly features to set up experiments [15], [31]. Recent updates allow output into a SQLite database as opposed to a profile file, this allows for easier access and flexibility manipulating the output.

Tau tools provide insight via the five modes: interval events, atomic events, query, control, and sampling. The output profile text data is then fed into Paraprof for analysis and visualization. Paraprof focuses on four domains: the Data Source System, the Data Management System, the Event System, and the Visualization System [15]. Resulting visualizations can even be interactive, so that a user can dig deeper into specific issues by clicking on areas of interest, e.g., a particularly large exclusive time value for a specific function.

Trace visualizations are supported by TAU as well, data is collected and formatted in a way that is compatible with Vampir for visualization. Vampir supports many different views for analysis beyond just tracing, including summarizing and clustering [32], [33]. Other trace visualization and analysis tools are also supported by converting the TAU trace data with one of their supported conversion tools i.e., EPILOG converter and the Expert tool. Grafana is a visualization dashboard that incorporates trace/time series data in more of a monitoring approach which will be discussed further in section IV-D [26].

Google has also created a robust trace analysis and visualization tool called Perfetto. Perfetto can analyze and visualize data from a number of different sources. It runs completely inside of the web browser and has interactive zoom and select capabilities. This allows the user to control their view and dive deeper into areas of interest. This can even be done offline

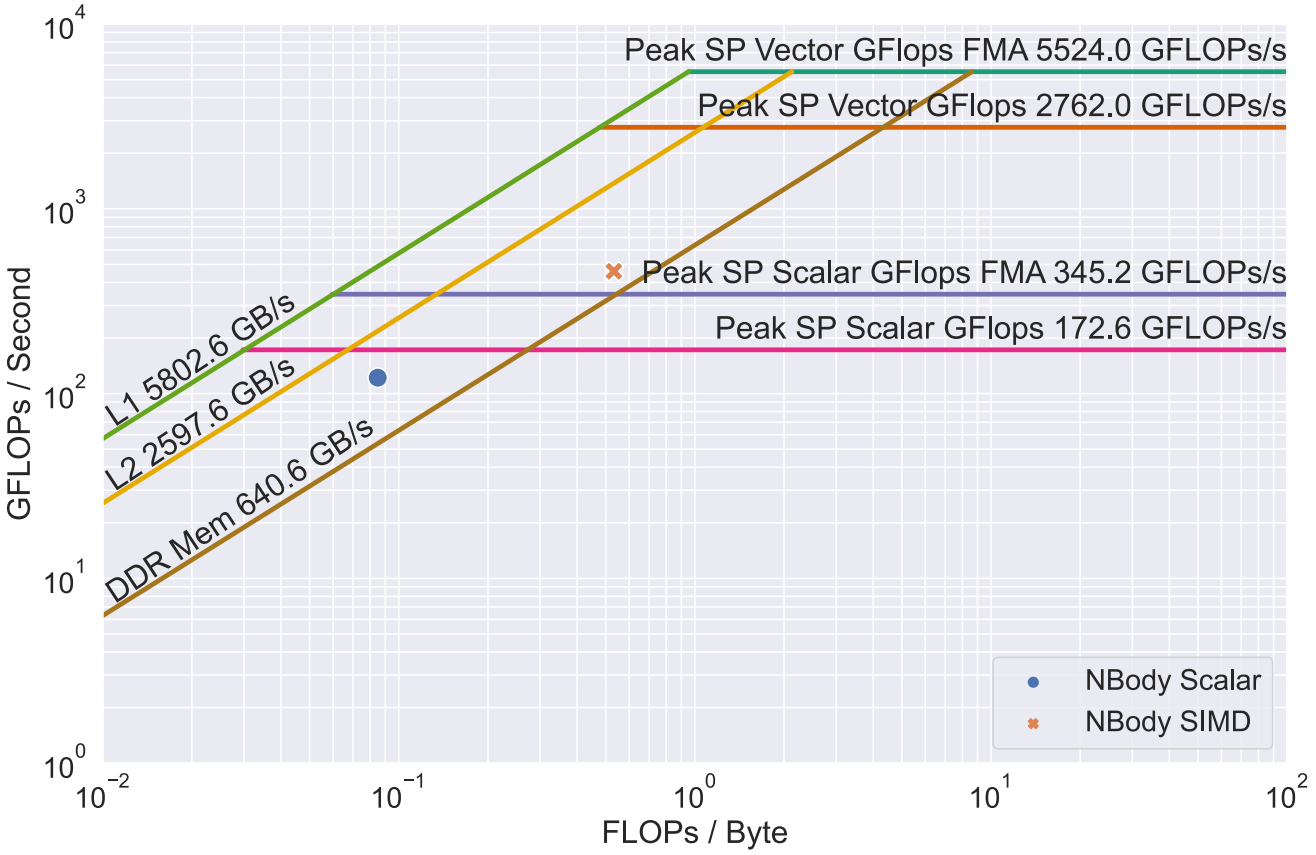

Fig. 3. An example roofline plot displaying multiple cache levels (diagonal lines), multiple peak flop rates (horizontal lines), and some example application data points (dots). Once an architectural roofline is created (lines), application points from functions or kernels could be plotted as soon data becomes available, which could integrate well into an *in situ* visualization workflow.

once the trace file is opened inside the browser window [34].

C. Rooflines: A Visualization Success Story

The graphical Roofline model is one of the most successful visualization concepts of hardware performance to come out of the literature in recent history. It has taken off like not many other concepts have, being incorporated into professional tools and used in the majority of papers discussing hardware performance. It's success is owed in part to it's simplicity, with a straightforward shape that is easily understood. Part of it's success is due to it's portable and flexible nature, one must only collect the correct values from their hardware and they are easily comparable across different architectures. The Cache-aware Roofline model (CARM) incorporates multiple cache levels [35]. Rooflines can also include multiple peak flop rates (roofs), for example, when looking at differences between scalar and vectorized peaks and/or enabling Fused Multiply Add (FMA). A Roofline model also lends itself well to customization, allowing additional plotting of measured application performance or kernel performance in regards to the theoretical peaks.

The original roofline model was outlined by Williams et al. out of Berkeley Labs in [36]. First emerging in 2009, it was instrumental in assisting scientists moving to parallel programming from serial programming. The basic concept behind the roofline graph is that it plots GFLOPs per second on the y-axis, and Arithmetic Intensity on the x-axis. Arithmetic intensity is defined as the total number of FLOPs computed, divided by the total number of bytes transferred between the processor and the specified memory level. Plotted on a log-log scale, this creates the visual of a sloped roof, connected to a flat roof, indicating the peak values achievable on this particular hardware. The ability to evaluate how a kernel is performing based on the limits of the hardware was a key for scientists porting their code to new hardware, or improving their code on existing hardware. It was now easier than ever to answer questions about performance and work towards greater optimization of codes.

Ilic et al. made some extremely useful updates to the original Roofline model when they introduced the Cache-aware Roofline model (CARM). They take into account the different cache levels, not just the DRAM, and thus the different bandwidth measurements for each of these cache levels. The bandwidth to the L1 cache is higher than the bandwidth to DRAM and thus there is now opportunity for understanding performance within the two bounds. The result is an improvement on the older model, and has been widely adopted, often replacing the original Roofline model.

Intel Advisor is one of the tools that has incorporated the CARM into it's suite of offerings [37], [38], see table II in the "Architecture (Roofline)" column. Expanding and building on the models proposed in [39] which focus on understanding the memory access can affect the energy efficiency. Intel Advisor's capabilities extend to plotting the kernels on the graphical CARM, and whether those kernels are vectorized. Intel Advsior has made it more straightforward to profile an application and generate a graphical CARM bringing the concept to the forefront. It is easier, especially for domain scientists who do not have the time or expertise to dig into the specifics of the machine and application, to create these visualizations and understand the performance of their code. Because of this adoption by professional tools and the ease of understanding, it is likely to see a roofline visualization of some variety in any performance analysis of an application now.

The Roofline Model tools have been extended to support GPU architectures as well, using the same Graphical Roofline concept [40]–[42]. This is another example of how flexible and portable the roofline concept is, that without changes to the visualization, it can be extended to different architectures. The extension to GPUs from CPUs allows for far more robust comparisons of application performance. With the increasing use of GPUs to accelerate code it is crucial to be able to characterize application performance on these architectures.

A recent extension includes the Instruction Roofline Model, which deviates from the traditional floating point metric approach, and instead focuses on representing integer operations. The Instruction Roofline Model incorporates the number of instructions issued, and can help to highlight bottlenecks, instead of just overall performance. A couple of Instruction Roofline Models have recently been proposed for both NVIDIA and AMD GPU architectures [43], [44]. In general the Instruction Roofline Model allows for deeper insight into the memory performance, enabling identification of issues with shared memory conflicts and memory access patterns.

IV. Challenges and Opportunities with Parallel Performance Visualization

This section discusses some of the major issues that researchers within the field of massively parallel performance visualization have to contend with today. First, it elaborates on the different approaches to different types of performance data. Then moves into the challenges of performance analysis and visualization on heterogeneous architectures and some vendor-specific solutions. This section also describes some current *in situ* performance visualization efforts, including performance monitoring.

A. Different Data, Different Views

While trace (time-series), and profile (summary) performance visualization, have been discussed in some detail already. it is worth mentioning some other approaches to the standards.

VIPACT is a visualization tool that is built to take in profile data from a tool such as HPCToolkit and generate a more intuitive visual representation. VIPACT differs from most other tools because it maintains per process performance data separately [45].

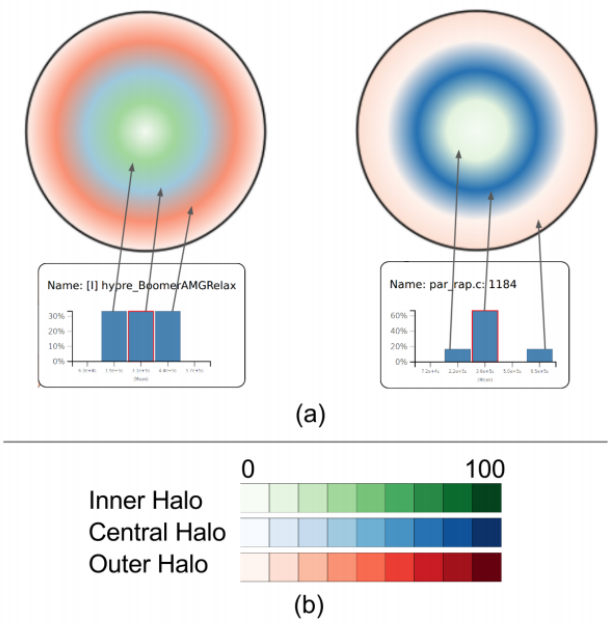

Fig. 4. An example of a the "halo node" visualization generated by VIPACT as an alternative to a traditional Calling Context Tree. This conceptualizes the distribution of work across processes, the halo on the left had unevenly distributed work and the halo on the right was well-distributed (functions executed close to the mean across all processes) [45].

VIPACT introduces the visualization concept of a "halo node", a new approach to contextualizing a normally flat calling context view. These show the distribution of performance data over per process calling context trees, see Figure 4. The other novel visualization that they have created is the directed acyclic graph (DAG) with interactive nodes. This halo node-DAG view allows a programmer to "zoom in" on specific graph nodes which represent a function. Zooming in on node can show the runtime compared to other functions in the graph [45]. This gives new perspective to programmers to identify which processes are slow.

Hatchet is a python library specific for handling structured profile data that contains information about the code the programmer is analyzing. It presents three options for viewing the same profile data, the traditional callgraph, a tree-based version, and a flamegraph, see figure 5. It builds on the python pandas data library concept of a dataframe to shape this performance data into something more easily manipulated. Bhatele et. al. created the main data structure called a graph frame which is made up of a graph and a dataframe that contains information about nodes in the graph [46], [47].

One of the concepts that is new with the release of Hatchet is the ability to use the graphframes in ways similar to a dataframe to easily shape the data for visualiztion. For example, you can add and subtract from each other, and quickly generate tree-like representations. It is also possible to load the data directly from some performance analysis tools

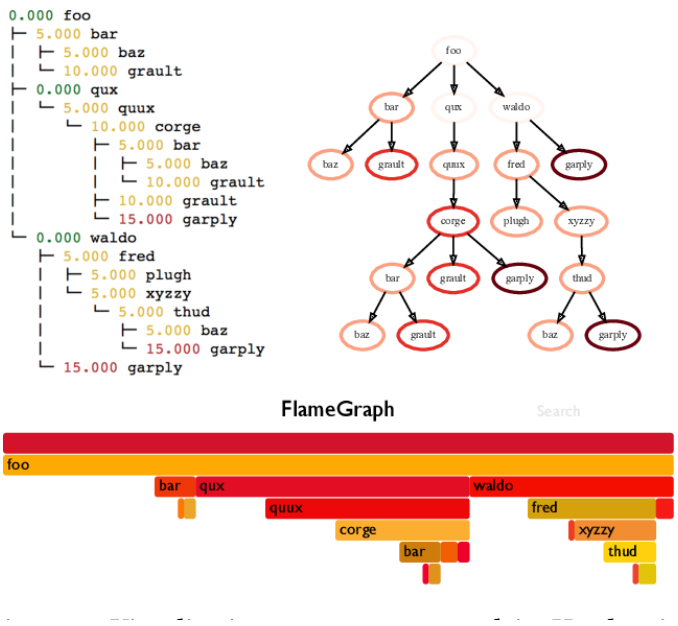

Fig. 5. An example of the three different profile visualizations of the same data, generated by Hatchet, including the traditional call graph, a tree-based call path, and a flame graph [46]. These are generated post-hoc, as that is the only time it is guaranteed that the full call stack is known. However, partial profiles could be generated *in situ*, especially in the case of a simulation that repeats function calls over each timestep, with the understanding that the stack could still be incomplete.

like HPCToolkit and Caliper [46]. Though for other tools that generate profile files it won't necessarily work the same way, and a custom parser may be needed.

Both Hatchet and VIPACT rely on existing tools to gather the performance data and then simply work with the output from these. As there are a significant number of tools that gather the desired metrics, the creators of Hatchet and VIPACT did not feel the need to rewrite this functionality, instead building on what is already available. This shows that what they felt was lacking most from the existing tools was the visualization aspect, especially in this multi-dimensional data domain. This also presents a potential approach for incorporating timing-agnostic visualization-specific tools with *in situ* metric gathering programs. However, updating only the output, and not restructuring the data in the first place is limiting in its own way. Both Hatchet and VIPACT are reliant on getting the correct information from existing tools, they cannot customize this beyond the measurement tool's capabilities if the programmer deems something is missing. These tools focus on how to best display the data we can already get. There is room in this exploration to look at how to restructure the source of the data, perhaps the profile files. Each of the earlier tools that gathers their own data takes advantage of this, by making sure to gather the data they they think a programmer will want to display. Clearly, though, they are still missing some important factors in visualization if tools like Hatchet and VIPACT need to be developed.

Recent work in the visualization of calling context trees includes an interactive approach built into jupyter notebooks that allow a user to manipulate the visualization and script in tandem so as to have a truly exploratory environment for analysis, see Figure 6. They used a node-link representation of the calling context tree and had users conduct both open-ended exploration as well as complete some pre-defined tasks [48]. While this specific project utilizes the summarized profile data from a complete application run, this is an approach that would lend itself well to an *in situ* workflow, as a user could interact with the data as it is being produced and make decisions based on their findings. CallFlow is another tool for visualization call graph and context data and make use of a Sankey diagram to indicate the flow of certain metrics, see figure 7. Recent developments include functionality to conduct analysis an ensemble of call graphs [49], [50].

B. Distributed Memory

This section discusses the class of tools that can visualize performance of distributed memory applications that run across multiple compute nodes. Understanding how each MPI rank is accessing memory and communicating between nodes gives insight into the effectiveness and scalability of an application on large compute clusters. Sometimes the ranks are consistent in their performance, and sometimes there In a general sense, collection information per process for these tools creates more overhead through instrumentation of the code. In one way, the creation of more detailed data suggests it may be harder to use in an *in situ* fashion. On the other hand, instrumenting the code also allows for a level of access to the code and any important metrics that could support a robust *in situ* performance visualization pipeline.

Perhaps one of the most well known in this category is Vampir [32], [51]. Since the initial release in the early 1990's there have been numerous new releases and updates. Vampir and VampirServer (a distributed version of Vampir) can perform analysis and visualization on trace data from VampirTrace or other collection tools such as TAU [33]. The Vampir tools enable data management through filtering and specific views. Reducing the amount of informational noise presented can sometimes allow performance bottlenecks to be discovered. Another helpful bottleneck identification feature is the ability to zoom in and out on a view, which allows for discovery at multiple different levels of granularity.

An interesting performance visualization concept was introduced by Isaacs et al. with *Boxfish* [52]. They capitalize on the idea of visualizing a *projection*, a projection refers to mapping data from one domain to another. For example OpenMP threads and hardware processing cores could be linked to provide insight about how an application is performing. The entire tool is built around supporting these types of projections. Their mapping concepts including many-to-one relationships which helps enable some of the more complex mappings. The visualizations that it can generate include scatter plots, histograms, 2D and 3D Meshes, and communication graphs [52]. Although in concept this seemed promising, Boxfish did not seem to gain very much traction after its initial release.

Jumpshot is a tool for conducting "post-mortem" analysis through a graphical interface. This includes multiple views, i.e. a timeline per process, or a "mountain range" (histogram) view

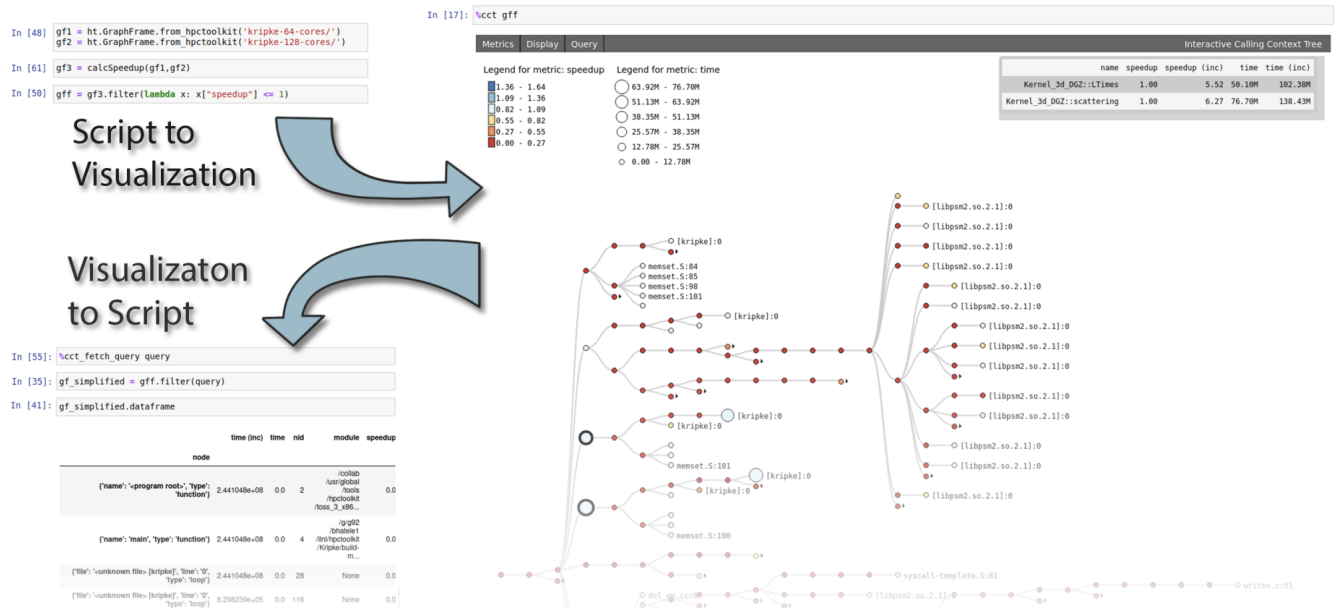

Fig. 6. The interactive jupyter notebook approach for visualizing calling context trees in [48]. While the data is currently being analyzed *post hoc*, the interactive nature of code to visualization would integrate well in an *in situ* workflow where the data is still changing.

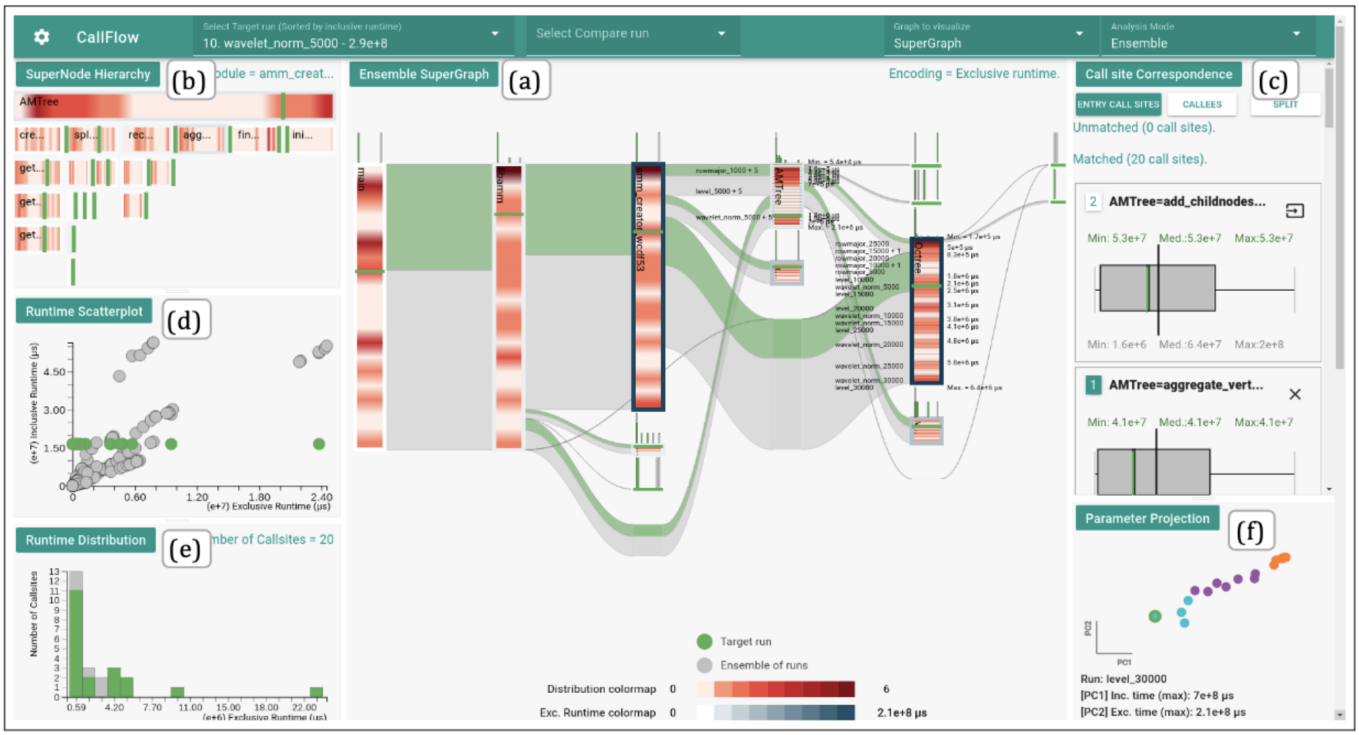

Fig. 7. The *ensemble-sankey* visualization for calling context trees from CallFlow [50] is the only tool utilizing a sankey diagram to denote the flow of certain metrics. In addition, this was targeted for comparative analysis of multiple calling context trees, which can be seen especially in the runtime comparison sections, (d) and (e), and the ensemble supergraph (a).

of processes and their states [53]. It relies on fully formed timestamped data files (logfiles) as input, and was not meant for any *in situ* capabilities. While it has specific integration to work well with MPI, it seems that the choice to use Java to implement it led to major portability issues. ParaGraph [54] is another tool that provides uniquely animated visualizations of distributed memory parallel systems.

C. Portability

Another crucial difficulty facing all creators of performance visualizations is the ability to use tools or techniques across heterogeneous architectures. A scientist will find themselves faced with the dilemma of running their codes on a new architecture anytime they gain access to upgraded equipment, or explore other options for acceleration. It is very useful to performance across different CPU architectures, or analyze their application's performance with the addition of GPU acceleration. Vendors will often build a visualization tool to support their own architecture, but Tools like TAU, that use standard formats such as OTF2 attempt to bridge this gap, yet many vendor-specific tools remain that

Intel OneAPI is a suite of tools built for HPC programming which includes the three performance analysis tools, VTune, GDB (a debugging tool), and Advisor which is referenced earlier when discussing roofline models. The focus for this study is on VTune and Advisor as they are more focused on displaying aggregated results in an intelligent manner. While some functions of these tools work on multiple hardware platforms and with compilers such as gcc, some features are only supported - or easier to implement - when using Intel hardware and an Intel compiler. Advisor makes heavy use of the roofline concept and plotting application data with respect to the hardware roofline. VTune on the other hand relies more on presenting the data in an almost spreadsheet like format, with some color coding to indicate outliers etc. [25], [55]

Intel Advisor can plot the roofline graph for the hardware environment as well as the performance of the application (kernels, functions, loops) as points on the graph. Advisor determines whether the kernels are compute or memory bound and can even offer suggestions for ways to improve performance [37]. Kernels that take up a higher percentage of the runtime are larger in size, making it easy to visually identify problem areas. Kernels can also be filtered out to focus on and compare fewer at a time. Custom color coding can also be applied to the kernels to classify them beyond the default format. Intel Advisor can also load different multiple views (into different windows), to compare versions of the application. For example, performing one of the suggested tasks for optimization, i.e., optimizing a loop, and comparing the before and after performance of the loop and application can be quite insightful. The loop could change from compute bound to memory bound, or could see a significant increase in performance, depending on the change made. Using this technique an application can be tuned over and over, continuously making improvements and visualizing the results. While CPU systems have historically been the focus, GPU systems are also supported, including showing the GTI bandwidth as one of the cache levels, to indicate external memory performance.

NVIDIA Visual Profiler automatically generates a plethora of performance data for the programmer to navigate through their UI. This tool is to be used with any CUDA-enabled application, and comes basically automatically for anything written with CUDA [57]. NVIDIA Visual Profiler provides many different "views" to the programmer, these"views" highlight different aspects of the performance, GPU details, CPU details, etc. All these views are inside a graphical user interface for ease of navigation. Nsight is another NVIDIA tool, meant to eventually completely replace NVIDIA Visual Profiler, however it does not contain all the overlapping features yet. One major update for Nsight is that NVIDIA wanted it to be less CUDA-centric than Visual Profiler [58].

For programmers who are used to using these kinds of tools, it may be quite intuitive to interpret the visualization. Yet often for those unfamiliar it can take some time to understand, much like many of the other tools mentioned thus far. Due to the complex nature of performance data it is hard to see a way around this fact. One other limitation of NVIDIA Visual Profiler is that it only works for CUDA-enabled applications, a problem they are trying to improve on with Nsight. Nsight also has updated usage methods and outputs, attempts to make both using it, and communicating results more intuitive [59].

RocProf is the profiling tool for AMD GPU hardware, with Chrome Tracing as the front end for visualization [60]. It utilizes the RocProfiler and RocTracer APIS to collect metrics and counter data. The output file is in JSON trace event format [61], meant to be used with the Chrome Tracing for viewing [62], [63]. Chrome Tracing was built for other purposes (visualizing web usage) and is thus quite limited in it's capabilities for performance data. Unfortunately, unlike the NVIDIA and Intel vendor tools, this data format and tool choice does not allow for visualizing a roofline or the architecture limits. However, Chrome Tracing is set to be replaced by Perfetto, which offers many more features.

The Cray architecture performance measurement suite includes CrayPat and MPP Apprentice2 for visualization [64], [65]. Cray performance measurement tools create the required data file that serves as input into Apprentice2. Apprentice2 can then generate a number of different tables and plots of report information, i.e. Call Stack, I/O. Plots can be zoomed in to show less data, or zoomed out to show a bigger picture of the data. Gathering the appropriate data for an effective performance visualization is a large task, most tools that generate visualizations gather their own in order to properly visualize it. These are powerful, delivering rich features and interesting views that can help a scientist understand the performance of their application. However, there is a subset of tools that rely on data gathered by other tools and focus on just making it visually interactive and appealing. There is an exploratory aspect to some of these tools that we like – allowing the user to create something that makes sense to them. Both of these categories of software visualization tools will be discussed in this section.

D. In Situ Performance Visualization Developments

The concept of performance monitoring is not new, there are numerous systems already that can help with gathering the data

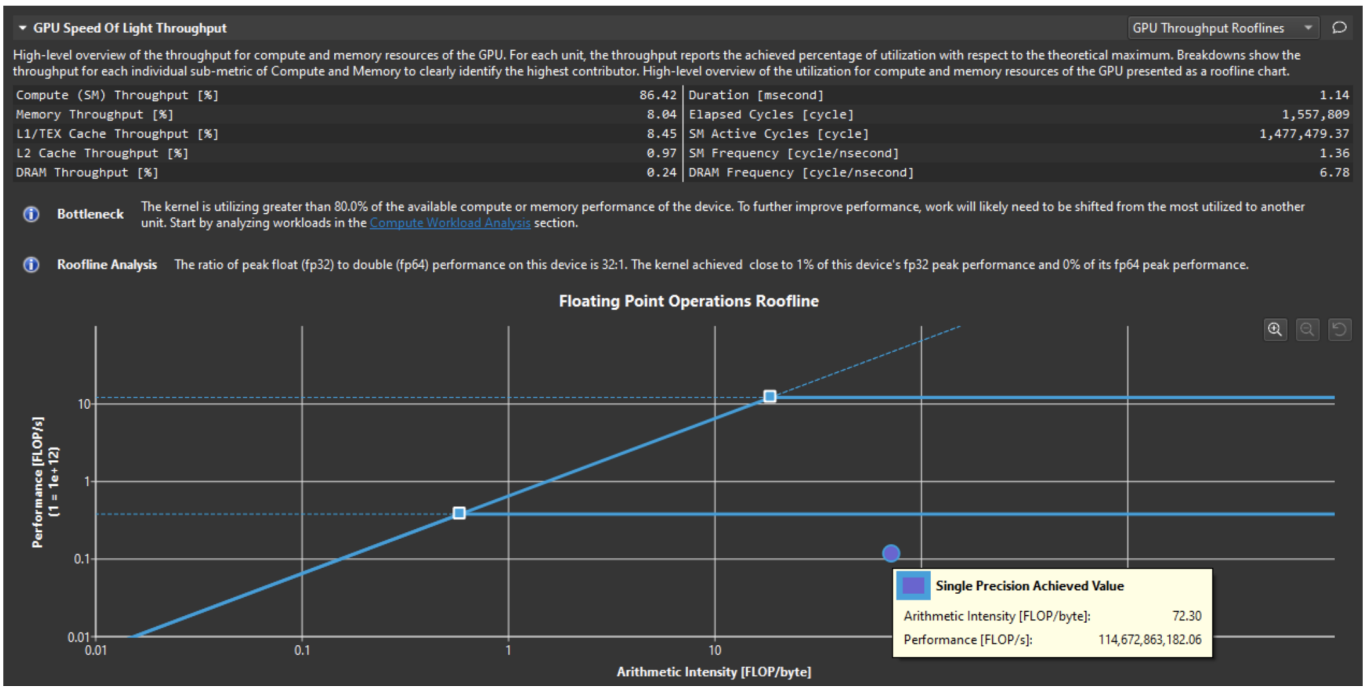

Fig. 8. An example of a GPU roofline visualization generated by NVIDIA's Nsight Compute [56], similar to Intel Advisor, it also provides suggestions for optimizing performance. When compared to the roofline in figure 3 they are both generated with *post hoc* data, however, Nsight's is interactive. Once the architectural roofline is generated, kernel profiles could theoretically be added *in situ*.

required to conduct performance analysis online. Wood provides an excellent survey in the area of online monitoring [66]. Visuzaliation of this data ranges in these systems from very basic text/table output to more comprehensive graphs. The term "monalytics" was used as a combination of the terms "monitoring" and "analysis", referencing the approach for detecting and managing system and applications behaviors in a data center. [67]

One such example is SosFlow [68]. SosFlow shows how the *Scalable Observation System* (SOS) can collect low-level data from instrumented software for use in analysis. They support complex scientific workflows running on clusters by implementing an integration with TAU (called TAUflow) which intermittently submits the regularly collected TAU data to SOSflow. This is a useful integration as it makes use of existing tools for collecting performance metrics and couples it with an in situ analysis framework. They evaluated the overhead for this extra processing in the general range of 1%-3% of the total walltime of the application. Visualization of this data was extended using Alpine, mapping the performance data to the geometry of simulation data [69].

The Falcon system provides application-specific monitoring, information about the overhead of the current monitoring, as well as graphical monitoring views [70]. This is useful in steering the application towards better performance. The visualization capabilities are relatively basic and two-dimensional, and some custom work was done for their evaluation to be able to create useful visualizations in a short enough period to enable on-line analysis and steering. While this is a promising start, realistically, scientists cannot be expected to do custom visualization work for each application they wish to monitor and analyze on-line.

DIMVisual Hierarchical Collection Model (DIMVHCM) is perhaps the most visualization focused of recent monitoring tools. Two major goals were to visualize the behavior of large scale parallel programs as well as collect this data in an on-line manner. DIMVHCM consists of three different types of data collectors and a push mechanism, e.g. data sent when certain conditions are met [71]. The system includes DimVisual [72], which aggregates the data for the visualization component TRIVA [73]. However, in order to actually run the graphical interface *in situ* they were required to implement a workaround client that integrated with TRIVA based on timestamps of the data. The ultimate effect was essentially *in situ* graphical monitoring, but with perhaps too many required steps and workarounds for mass adoption. While TRIVA is capable of some interesting distributed memory visualization, it is not clear what was taken advantage of in the DIMVHCM case studies.

Ganglia, Nagios, and Lightweight Distributed Metric Service [74] are primarily entire cluster monitoring systems [75], [76]. The focus here is on the behavior and health of the entire cluster, and not of a specific application performance and/or how to improve it. Ganglia has interesting visualization capabilities though, specifically, it integrates with RRDtool (Round Robin Database), a circular database, [77] to visualize the time series data that is collected. The final output is web-based and separated from the performance data, which allows for customization of the visualizations without accidentally manipulating the collected data. While this is interesting and necessary for understanding the full context of application performance, it is not the granularity of information needed

for on-line tuning of specific scientific codes.

TACC Stats is similarly focused on a full data center, and can offer insights such as when an application has idle nodes. [78] Some plotting functionality is included with some optional scripts/workflows that are designed to be run on a predetermined intermittent basis, not based on any conditional fulfillment. These plots are useful, but not able to be customized, two-dimensional, or interactive.

Grafana is an open-source and enterprise tool that offers a dashboard of customizable visualizations for a number of different data types, including trace data. This is often used as a monitoring dashboard for users to understand the overall health and performance of their system. Users can build visualizations that make sense for their system, including bar charts for categorical data and heatmaps. As their interface is an API, it is already used in an *in situ* fashion, with live updates as changes happen on the system [26].

V. Related Domains and Their Approach to In Situ or Exploratory Visualization

A. In Situ Scientific Visualization

The visualization of scientific simulation data has become increasingly challenging area as the volume of data grows. Scientists in this field face similar issues to those in performance data, with large amounts of complex data that needs to be visualized in a way that is digestible by the end user. Thus, the demand for *in situ* functionality grows, often in such a way that a user or program can "steer" the application based on these visualizations. One such example of a response to handling this data is with VisIt [79]–[81]. VisIt's focus is delivering scientific visualization capabilities for large datasets that have been generated on parallel clusters.

Numerous *in situ* scientific visualization frameworks have emerged in recent years. Paraview focuses on generating interactive and exploratory scientific visualizations for large scale datasets [82], [83]. Paraview Catalyst has been incorporated into *in situ* scientific work flows as the visualization component in [84]. Alpine is an *in situ* scientific visualization infrastructure built upon the Strawman prototype [85], [86]. The ALPINE API uses VTK-h, Flow, and Ascent to generate in situ data analysis and visualizations.

One key difference between the two fields that may be to the advantage of performance visualization is the fact that it may be more plausible to ignore, or lose data. Many scientific applications must be paused in order for the scientific visualizations to be generated, because if the simulation were allowed to continue on then important discoveries could be overlooked or missed completely. However, because of the nature of performance issues, it is much less likely that any such insight is only able to be made at a single point in the application. This opens the door to the possibility of reducing the overhead seen by some of these *in situ* scientific visualizations, and allowing the simulation to continue while performance metrics are analyzed and produce results.

B. Exploratory Information Visualization

Exploratory information visualization or visual analysis is the ability for a user to interact with their data visualizations

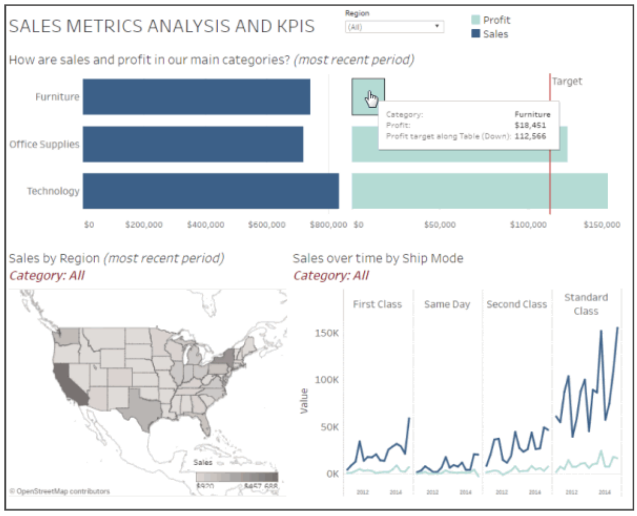

Fig. 9. An example of an interactive, clickable dashboard in Tableau, that enables a user to change the data they are viewing [88]. The data behind these visualizations could be considered *post hoc*, as it is gathered from a previously completed time period, yet is made interactive for exploratory purposes.

in order to guide what is being visualized and uncover new insights. A user may start with a general hypothesis, and refine the visualization until they have a solution, or uncovered a problem. Or a user may have an "open ended" approach, without a specific goal in mind. Either way, exploratory visual analysis is a crucial component of visualization research, yet because of these differing goals, can be difficult to implement well [87]. The domain of the data, and goals of the user may be key in producing an effective tool here.

The increase of big data collected by businesses has given rise to tools like Tableau which can provides insights into their data through exploratory visual analysis [88]–[90]. While business data is not the exact same as HPC data, there can be multi-dimensional and temporal qualities, i.e., number of units of each item sold, across different types of stores, in different regions, within certain time periods [88]. Tableau's main approach is to enable exploratory visual analysis via clickable dashboards that can change what data is viewed to answer different questions, and remove the need for programming analysis written in Python or R, see Figure 9 [89]. Their VizQL solution enables the majority of their interactive visualization functionality, allowing the user to learn with feedback from the visualization, and vice versa [90]. Applying a similar technology in the performance data domain would be groundbreaking.

Other research projects looking at visual analytics of high-dimensional data can provide other perspectives. For example, mapping multi-variate data to a concept that people already understand, in this case, interactive route planning [91]. This creates a visualization that the user can understand and interact with more readily from data that might otherwise be unmanageable. Another project explores the relationship between the data analyst and a machine learning model of the data, allowing the analyst to interact and refine the model. At the same time the model learns to identify relationships

and patterns in the data, which may be difficult for a human to identify. This project supports "both model-driven data exploration, as well as data-driven model evolution" [92].

VI. Machine Learning for Performance Improvement and Visualization

This section discusses current and potential applications of machine learning for diagnosing bottlenecks and identifying opportunities for performance improvement as well as through performance prediction or modeling. Performance modeling is determining how an application will perform on a different architecture, or with different configurations. This section also discusses how machine learning can be used to generate better, more effective performance data visualizations. Similar tactics are being used in many fields with large search spaces for their data already, i.e., scientific visualization. Finding the best visualization from an exponentially large number of options can be time-consuming and tedious, even if automated.

A. Machine Learning for Performance Analysis

1) Diagnostics: The following approach to understanding the I/O throughput is crucial in that every HPC system will have a significant amount of data I/O. This is often to write out intermediate solutions or calculations for validation or use. Gauge is a tool used to analyze logs from HPC applications to identify I/O bottlenecks [93]. They showed success wherein a Gauge user was able to successfully identify problem nodes in their cluster that were causing a slowdown in performance. This success, based on their analysis of groups of jobs with similar traits, shows that with the correct training data an effective diagnostic tool can be created. Similarly, in [94], they build a predictive model for the output on the supercomputer cluster Titan.

Tuncer et al. introduce [95] which looks at the frequent performance variations on systems that disrupt job runs and result in worse performance. Problems can range on clusters from software to hardware changes that can negatively affect performance and they are difficult to identify due to the large volumes of data collected. These performance changes can be caused by resource availability on shared clusters, or hardware fluctuations. This particular project utilizes performance counters and resource usage in their training data to train a model that can identify the reasons for the performance variations for an application. Their experiments cover both an HPC cluster and a public cloud service.

Examples of the node-level anomalies they identify are out-of-memory (typically caused by a memory leak), orphan processes, and hidden hardware problems. They use 7 different models, including two baseline models, ST-Lan and FP-Bodik. The five models evaluated were k-nearest neighbors, support vector classifier, AdaBoost, decision tree, and random forest. Tests of their models were conducted with the NAS Parallel Benchmarks. Their results were quite good, showing a low-overhead and performant framework of models. They are also able to extract some of the most important features to the models, which provides a good understanding of where and why the causes of performance variation tend to arise. This work in [95] shows significant promise for this area of using machine learning to sift through large volumes of HPC data and identify problems.

A later extension of this work by Tuncer et al. includes [96] where they transform time-series data into statistical features and train the model on this transformed data. Here they focus on the tree-based models used in earlier works, decision tree, AdaBoost, and random forests, because they were more successful than other model types. They also focus heavily on selecting the most useful features for training, using the Kolmogorov-Smirnov test with the Benjamini-Yakutieli procedure. Their experiments are run on a CRAY cluster, and they use over 500 metrics collected on the system by Lightweight Distributed Metric Service. They use the FP-Bodik as a baseline model again, but replace ST-Lan with ICA-Lan. Ultimately their method results in a 98% correct identification rate of the anomaly types, and is independent of the application. It is important for it to be independent of the application, because the real value of this model is that it could be applied to new applications/environments it has never been trained on before.

In another anomaly detection project, [97], an auto-encoder neural network is used to identify problems. A unique facet of their approach is to actually embed a monitoring board onto each computer, to gather the training data, and then later, housing the trained model. While this likely results in less overhead on the compute node, this requires their trained auto-encoder to be very lightweight, because the boards are not very powerful. Their results are highly accurate with up to 98% accuracy. Their results also show that they can run their model on the cluster computers without any significant overhead.

Targeting similar issues is Proctor [98], which aims to identify similar anomalies, but using a semi-supervised framework. One of the largest challenges facing machine learning models, and this domain is no exception, is the scarcity of labeled training data. Acquiring enough labeled training data to train a model can require countless hours of manual work if the data is not structured/categorized already in a way that makes sense for predicting the targets. Their solution to this is to use an unsupervised model initially combined with a supervised one for the final steps. They use the best random forest results from previous work in Tuncer et al [96].

2) Performance Prediction: Performance prediction or modeling is a related but somewhat different field to diagnostics. Performance modeling looks across different hardware or settings to predict where comparable or better performance can be achieved. This is useful when porting code to newer architecture, or when deciding what architecture to use in the beginning. Autotuning is an area within the performance prediction domain, relating specifically to automatically exploring the large search space of parameters into an application to optimize performance. This can be extremely useful as it can reduce the need to manually launch hundreds to thousands of different jobs in order to find the best combination of parameters.

WOWMON is a machine-learning based profiler, integrated with TAU, and provides on-line analysis, but there are no built-in visualization capabilities [99]. It's workflow manager MCo-

ordinator, uses some machine learning techniques to determine which metrics have strong correlation with latency. This is relayed back to communicate which metrics are important, and when to change them. Apollo and Artemis are two other machine learning based autotuners that are actually meant to be used for online tuning. Pretrained models that can quickly decide on new parameters during an application run make for minimal overhead [100], [101].

Other interesting approaches to autotuning include a couple of Bayesian Optimization models, one for selecting pragma parameters in a kernel [102], and HiPerBOt for exploring a larger parameter space of runtime settings, compiler flags, etc. [103]. Neural networks have also been applied towards performance prediction and tuning in [104] and [105]. There is also a tool specifically looking at OpenMP thread count and scheduling (binding) as a means of optimization [106]. Numerous other machine learning-based kernel tuning projects have been implemented across both CPUs and GPUs including [107], [108].

B. Machine Learning for Generating Visualizations

1) Learning from the Scientific Visualization Domain: The neighboring field of scientific visualization is currently grappling with many of the same issues as performance visualization. For scientific visualization, the data can also be multi-dimensional and complex, but unlike performance data, it is domain specific, concerned with displaying the scientific results that are being calculated by the application. For example, the data might be used to generate intermediate and final visualizations that model weather patterns or states in some other physical system. However, there are enough similarities to performance data, that looking to what they are doing as an example could prove useful. The first is that there can be a lot of data - and not all of it will be necessary for generating a meaningful visualization. Choosing the important features, and transforming the data in such a way that spotlights interesting findings are the two main goals in both cases. Another interesting similarity is that data can be collected and analyzed *in situ* or *post mortem*. *In situ* visualization has been an important field of scientific visualization for many years, and recently, some researchers are using machine learning to improve it. A recent project combines the Paraview scientific visualization tool with PyTorch. Allowing a user to select a visualization filter that is based on a trained deep learning model [109].

HPC codes are already heavily optimized to fully utilize fast computers and environments, when machine learning is added, sometimes additional HPC resources are needed to train or run a model to generate the needed visualizations. When running the models *in situ*, the problem then becomes how best to manage the available resources to support both the application code and the machine learning models at the ideal level. PAVE [110], is a framework that attempts to find the correct balance between the scientific application and the machine learning required to generate the visualizations.

2) Recommendation Engines: Ehsan et al. present a promising approach to searching for the *top-k* data visualizations and efficient pruning mechanisms to reduce data processing costs [111]. This method trains a model to recommend the *top-k* data visualizations after evaluating the usefulness of many more than a human could manually. They introduce three schemes that allow for more efficient pruning of poor visualizations, Multi-Objective View Recommendation for Data Exploration (MuVE), upper MuVE (uMuVE), and Memory-aware uMuVE (MuMuVE). These three schemes achieve comparable results and mostly differ in their tuning parameters and memory usage allowed.

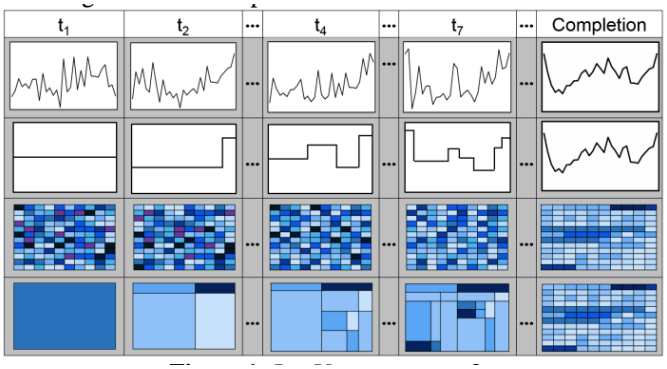

Fig. 10. An example of the incremental steps of visualization (rows 2 and 4) that IncVisage generates, detecting trends earlier, and with less fluctuation, than a traditional data sampling method (rows 1 and 3) [112].

Applying existing machine learning algorithms, from the MuVE family or others could be very effective in the performance data domain. The data domains used share some common traits, the first and foremost of which is that there can be so much data that it is next to impossible to manually sort through it all.

3) Other Visualization Models: Due to the high volume of data that can go into a performance visualization, looking at methods that seek to produce better visualizations with less information, or before processing all of the data are a promising endeavour. In [112] they do just this, by presenting a tool called IncVisage that can generate trend lines and heatmaps for large datasets in an incremental but quick fashion. This takes the approach of building out a visualization in stages, so as to identify areas of interest early on, allowing for faster understanding, decision making, and exploration by the viewer, see figure 10. This approach yields potential because it solves the major issue with most sampling methods for visualizations, which is that correctness can not typically be guaranteed when only using a subset of the data.

When considering how best to help users interact with creating or understanding their visualization, this notion of fast, incremental updates is quite attractive. Imagine using an interactive *in situ* visualization tool, shifting to a new view or deep diving into a particular area and waiting minutes for a new visualization to appear. This would be unacceptable, tools need to be responsive to be usable. Most vendor tools on the market get around this by relying on collecting the performance data, and pre-processing it all at one time so preset views can be displayed once the user is ready to view. A user has access to a variety of views, but they are limited

to However, this new research opens up a world of flexibility and exploration that is not supported by current tools. A user could quit the program early if they have already discovered the information they were after. They could build visualization on-the-fly, with less limitations from the tool.

Projects such as DeepEye [113] provide a user interface that involves using a "Google-like" search to specify the type of visualization a user wants generated. The user then chooses among multiple highly ranked data visualizations that have been generated with their input data. They use a combination of binary classifiers and supervised learning to achieve this goal. Similarly to Ehsan's recommendation engine described above, the user can select "good" visualizations and be presented with additional visualizations in a somewhat exploratory manner. Their case studies provide interesting insights and stories about the user data, and generate but a more quantifiable analysis would be interesting to see.

VISER is another project that shows promising results for automatically generating data visualizations [114]. VISER is a multi-part algorithm that takes in a simple "sketch" and the initial dataset, and generates multiple visualizations using some of the most popular visualization libraries. They call this *visualization-by-example* because they start with the initial sketch and generate the necessary code to create the user's desired visualization. One of the biggest challenges they had to overcome was the difficulty in preparing the data correctly, as data needs to be reshaped, columns need to be added, etc. With performance data, this challenge would be exacerbated, beyond what they study with VISER, by the level of complexity. They create a search algorithm that transforms the data and selects the best transformations. Combining this technique with their other algorithms for generating the code, they have successful initial results, with 70% of their solved trials culminating with the desired visualization being among the top five generated by VISER.

VII. Conclusion

Current advances in effective performance visualization have trended towards more exploratory and interactive approaches, as evidenced by [37], [46], [48], and many more. These aid the user in performing visual analytics, diagnosing performance issues and finding opportunities for speedup. These also lend themselves to interfacing with *in situ* performance data in a way that could make the best use of real-time decisions by a user. Addressing the complications, i.e., superfluous data, lack of data, distributed data, and limited resources, listed in table I with a new perspective could warrant exciting results.

There is a large window of opportunity when it comes to using machine learning to enable *in situ* visualization and analysis in the performance data domain. Machine learning may be one of the best ways of dealing with the challenges listed in table I, as it can allow more analysis with less data, and faster results. We see from some of the other applications to complex data that these approaches can be not only interesting, but useful and even practical to implement in production. There is already the projects for analyzing performance data directly to diagnose bottlenecks or suggest ways to improve performance. There is also an opportunity to automatically generate insightful performance data visualizations from the raw data.

One of the challenges is structuring and labeling performance data in such a way as to be successful training a working model. However, there are many ways to address this challenge. Unsupervised methods can be implemented, or datasets can be curated for supervised strategies more easily if it is planned ahead. Automated application profiling approaches, such as in [115], could be built to collect labeled performance data. Automatic data visualization generation at this scale in the HPC domain is yet relatively unproven by consistent use in the wild. Although many of the models listed in this paper have shown potential, they are still mostly in the research phase.

Profile data is usually a summary over the entire application run, but now, using only a sample of the full data set becomes possible if we are able to feed it to a trained model. This sample could be taken from the application as it is running, or it could even be taken from a subset of nodes if collecting the data becomes too cumbersome. The model could be used to predict the performance once the codes have completed its run. This means that calling context trees could be generated earlier, even if some final data is missing. It could still be enough to get an idea of the performance.

Processing trace data *in situ* will be a huge win, as that means less data will need to be manipulated at the end of the application run. As an example, using a model similar to IncVisage [112], to generate heatmaps from trace data during execution, could allow a user to identify performance issues with specific functions or processes early on. Or they could see they are not achieving the performance they desire and kill the application early, make code changes, rebuild, rerun, and finally compare. Both of these scenarios will help save the user time and cut down on the cost of the HPC resources they utilize.

References

[1] J. R. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *2008 IEEE Hot Chips 20 Symposium (HCS)*, pp. 1–2, 2008.

[2] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," 1998.

[3] W. Gropp, E. L. Lusk, N. E. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Comput.*, vol. 22, pp. 789–828, 1996.

[4] B. Kågström, P. Ling, and C. V. Loan, "Gemm-based level 3 blas: high-performance model implementations and performance evaluation benchmark," *ACM Trans. Math. Softw.*, vol. 24, pp. 268–302, 1998.

[5] E. Anderson, "Lapack users' guide," 1987.

[6] M. Frigo and S. G. Johnson, "Fftw: an adaptive software architecture for the fft," *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, vol. 3, pp. 1381–1384 vol.3, 1998.

[7] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer, "State of the art of performance visualization," in *EuroVis*, 2014.

[8] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. A. Wolf, "Open trace format 2: The next generation of scalable trace formats and support libraries," in *PARCO*, 2011.

[9] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the open trace format (otf)," in *Computational Science – ICCS 2006*, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 526–533.

[10] A. Malony and W. Nagel, "Open trace—the open trace format (otf) and open tracing for hpc," 01 2006, p. 24.

[11] F. Wolf and B. Mohr, "Epilog binary trace-data format," 01 2004.

[12] S. Browne, C. Deane, G. Ho, and P. Mucci, "Papi: A portable interface to hardware performance counters," 1999.

[13] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[14] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. A. Wolf, "Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir," in *Parallel Tools Workshop*, 2011.

[15] S. Shende and A. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, pp. 287 – 311, 2006.

[16] L. Adhianto, S. Banerjee, M. W. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, 2010.

[17] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: Performance introspection for hpc software stacks," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 550–560.

[18] (2022) Overview survey. [Online]. Available: https://trenza.gitlab.io/survey.io/docs/introduction.html

[19] P. J. Nichols, "Nmsba: Continuous application benchmarking analysis – caba," 1 2021. [Online]. Available: https://www.osti.gov/biblio/1756775

[20] T. Röhl, J. Eitzinger, G. Hager, and G. Wellein, "Likwid monitoring stack: A flexible framework enabling job specific performance monitoring for the masses," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 781–784.

[21] O. Cankur and A. Bhatele, "Comparative evaluation of call graph generation by profiling tools," in *ISC*, 2022.

[22] R. Robey and Y. Zamora, *Parallel and High Performance Computing*. Shelter Island, NY: Manning Publications, 2021.

[23] A. Giménez, T. Gamblin, I. Jusufi, A. Bhatele, M. Schulz, P. Bremer, and B. Hamann, "Memaxes: Visualization and analytics for characterizing complex memory performance behaviors," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, pp. 2180–2193, 2018.

[24] A. Yasin, "A top-down method for performance analysis and counters architecture," *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 35–44, 2014.

[25] (2021) Intel vtune profiler performance analysis cookbook. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top.html

[26] (2022) Grafana documentation. [Online]. Available: https://grafana.com/docs/grafana/latest/

[27] K. E. Isaacs, P. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann, "Combing the communication hairball: Visualizing parallel execution traces using logical time," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, pp. 2349–2358, 2014.

[28] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *CACM*, 1978.

[29] C. Scully-Allison and K. E. Isaacs, "Design and evaluation of scalable representations of communication in gantt charts for large-scale execution traces," *ArXiv*, vol. abs/2107.00065, 2021.

[30] S. R. Brandt, A. Bigelow, S. A. Sakin, K. Williams, K. E. Isaacs, K. Huck, R. Tohid, B. Wagle, S. Shirzad, and H. Kaiser, "Jetlag: An interactive, asynchronous array computing environment," in *Practice and Experience in Advanced Research Computing*, ser. PEARC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 8–12. [Online]. Available: https://doi.org/10.1145/3311790.3396657

[31] A. D. Malony, S. Ramesh, K. A. Huck, C. Wood, and S. Shende, "Towards runtime analytics in a parallel performance system," *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 559–566, 2019.

[32] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis toolset," in *Parallel Tools Workshop*, 2008.

[33] H. Brunst, D. Hackenberg, G. Juckeland, and H. Rohling, "Comprehensive performance tracking with vampir 7," in *Parallel Tools Workshop*, 2009.

[34] (2022) Perfetto - system profiling, app tracing and trace analysis. [Online]. Available: https://perfetto.dev/docs/

[35] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, pp. 21–24, 2014.

[36] S. Williams, A. Waterman, and D. A. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, 2009.

[37] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. Matveev, "Performance analysis with cache-aware roofline model in intel advisor," *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 898–907, 2017.

[38] D. Marques, A. Ilic, Z. Matveev, and L. Sousa, "Application-driven cache-aware roofline model," *Future Gener. Comput. Syst.*, vol. 107, pp. 257–273, 2020.

[39] A. Ilic, F. Pratas, and L. Sousa, "Beyond the roofline: Cache-aware power and energy-efficiency modeling for multi-cores," *IEEE Transactions on Computers*, vol. 66, pp. 52–58, 2017.

[40] C. Yang, T. Kurth, and S. Williams, "Hierarchical roofline analysis for gpus: Accelerating performance optimization for the nersc-9 perlmutter system," *Concurrency and Computation: Practice and Experience*, vol. 32, 2020.

[41] C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. O. Adetokunbo, B. Friesen, B. Cook, D. Doerfler, L. Oliker, J. R. Deslippe, and S. Williams, "An empirical roofline methodology for quantitatively assessing performance portability," *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 14–23, 2018.

[42] Y. Wang, C. Yang, S. A. Farrell, T. Kurth, and S. Williams, "Hierarchical roofline performance analysis for deep learning applications," *ArXiv*, vol. abs/2009.05257, 2020.

[43] N. Ding and S. Williams, "An instruction roofline model for gpus," *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pp. 7–18, 2019.

[44] M. Leinhauser, R. Widera, S. Bastrakov, A. Debus, M. Bussmann, and S. Chandrasekaran, "Metrics and design of an instruction roofline model for amd gpus," *ArXiv*, vol. abs/2110.08221, 2021.

[45] H. T. Nguyen, L. Wei, A. Bhatele, T. Gamblin, D. Böhme, M. Schulz, K.-L. Ma, and P.-T. Bremer, "Vipact: A visualization interface for analyzing calling context trees," *2016 Third Workshop on Visual Performance Analysis (VPA)*, pp. 25–28, 2016.

[46] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: pruning the overgrowth in parallel profiles," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[47] S. Brink, I. Lumsden, C. Scully-Allison, K. Williams, O. Pearce, T. Gamblin, M. Taufer, K. E. Isaacs, and A. Bhatele, "Usability and performance improvements in hatchet," *2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools)*, pp. 49–58, 2020.

[48] C. Scully-Allison, I. G. Lumsden, K. Williams, J. Bartels, M. Taufer, S. Brink, A. Bhatele, O. Pearce, and K. E. Isaacs, "Designing an interactive, notebook-embedded, tree visualization to support exploratory performance analysis," *ArXiv*, vol. abs/2205.04557, 2022.

[49] H. T. Nguyen, A. Bhatele, N. Jain, S. P. Kesavan, H. Bhatia, T. Gamblin, K.-L. Ma, and P.-T. Bremer, "Visualizing hierarchical performance profiles of parallel codes using callflow," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, pp. 2455–2468, 2021.

[50] S. P. Kesavan, H. Bhatia, A. Bhatele, T. Gamblin, P.-T. Bremer, and K.-L. Ma, "Scalable comparative visualization of ensembles of call graphs," *IEEE transactions on visualization and computer graphics*, vol. PP, 2021.

[51] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel, "Developing scalable applications with vampir, vampirserver and vampirtrace," in *PARCO*, 2007.

[52] K. E. Isaacs, A. G. Landge, T. Gamblin, P.-T. Bremer, V. Pascucci, and B. Hamann, "Abstract: Exploring performance data with boxfish," *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 1380–1381, 2012.

[53] O. A. Zaki, E. L. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with jumpshot," *The International Journal of High Performance Computing Applications*, vol. 13, pp. 277 – 288, 1999.

[54] M. T. Heath and J. A. Etheridge, "Visualizing the performance of parallel programs," *IEEE Software*, vol. 8, pp. 29–39, 1991.

[55] A. R. Pandey, D. Tesfay, and E. Jarso, "Performance analysis of intel ivy bridge and intel broadwell microarchitectures using intel vtune amplifier software," *2018 2nd International Conference on Inventive Systems and Control (ICISC)*, pp. 423–426, 2018.

[56] (2021) Nsight compute. [Online]. Available: https://docs.nvidia.com/nsight-compute/2022.2/index.html

[57] (2021) Cuda toolkit documentation. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[58] (2019) Migrating to nvidia nsight tools from nvvp and nvprof. [Online]. Available: https://developer.nvidia.com/blog/migrating-nvidia-nsight-tools-nvvp-nvprof/

[59] (2021) Nvidia nsight systems user guide. [Online]. Available: https://docs.nvidia.com/nsight-systems/UserGuide/index.html

[60] (2022) Amd rocm profiler. [Online]. Available: https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCm-Tools.html

[61] (2022) Trace event format. [Online]. Available: https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/edit

[62] (2022) Chrome tracing as profiler frontend. [Online]. Available: https://aras-p.info/blog/2017/01/23/Chrome-Tracing-as-Profiler-Frontend/

[63] (2022) The trace event profiling tool (about:tracing). [Online]. Available: https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/

[64] (2017) Cray performance measurement and analysis tools user guide 7.0.0 s-2376. [Online]. Available: https://support.hpe.com/hpesc/public/docDisplay?docId=a00113917en_uspage=About_the_Cray_Performance_Measurement_and_Analysis_Tools_User_Guide.html

[65] W. Williams, T. W. Hoel, and D. M. Pase, "The mpp apprentice™ performance tool: Delivering the performance of the cray t3d®," 1994.

[66] C. Wood, "Online monitoring for high-performance computing systems," 2021.

[67] M. Kutare, G. Eisenhauer, C. Wang, K. Schwan, V. Talwar, and M. Wolf, "Monalytics: online monitoring and analytics for managing large scale data centers," in *ICAC '10*, 2010.

[68] C. Wood, S. Sane, D. A. Ellsworth, A. Giménez, K. A. Huck, T. Gamblin, and A. D. Malony, "A scalable observation system for introspection and in situ analytics," *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, pp. 42–49, 2016.

[69] C. Wood, M. Larsen, A. Giménez, K. A. Huck, C. Harrison, T. Gamblin, and A. D. Malony, "Projecting performance data over simulation geometry using sosflow and alpine," in *ESPT/VPA@SC*, 2017.

[70] W. Gu, G. B. Eisenhauer, E. T. Kraemer, K. Schwan, J. T. Stasko, J. Vetter, and N. Mallavarupu, "Falcon: on-line monitoring and steering of large-scale parallel programs," *Proceedings Frontiers '95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pp. 422–429, 1995.

[71] R. K. Tesser and P. O. A. Navaux, "Dimvhcm: An on-line distributed monitoring data collection model," *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 37–41, 2012.

[72] L. M. Schnorr, P. O. A. Navaux, and B. de Oliveira Stein, "Dimvisual: Data integration model for visualization of parallel programs behavior," *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, vol. 1, pp. 473–480, 2006.

[73] L. M. Schnorr, G. Huard, and P. O. A. Navaux, "Triva: Interactive 3d visualization for performance analysis of parallel applications," *Future Gener. Comput. Syst.*, vol. 26, pp. 348–358, 2010.

[74] A. Agelastos, B. A. Allan, J. M. Brandt, P. Cassella, J. Enos, J. Fullop, A. C. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. T. Showerman, J. Stevenson, N. Taerat, and T. W. Tucker, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 154–165, 2014.

[75] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Comput.*, vol. 30, pp. 817–840, 2004.

[76] G. Katsaros, R. Kübert, and G. Gallizo, "Building a service-oriented monitoring framework with rest and nagios," *2011 IEEE International Conference on Services Computing*, pp. 426–431, 2011.

[77] S. Dargad and M. Singh, "Rrdtool: A round robin database for network monitoring," *Journal of Computer Science*, 2017.

[78] R. T. Evans, W. L. Barth, J. C. Browne, R. L. DeLeon, T. R. Furlani, S. M. Gallo, M. D. Jones, and A. K. Patra, "Comprehensive resource use monitoring for hpc systems with tacc stats," *2014 First International Workshop on HPC User Support Tools*, pp. 13–21, 2014.

[79] H. Childs, "Visit: An end-user tool for visualizing and analyzing very large data," 2011.

[80] S. Ahern, K. S. Bonnell, E. Brugger, H. Childs, J. S. Meredith, and B. Whitlock, "Visit: a component based parallel visualization package," 2000.

[81] G. H. Weber, H. Childs, and J. S. Meredith, "Recent advances in visit: Parallel crack-free isosurface extraction," 2012.

[82] U. Ayachit, "The paraview guide: A parallel visualization application," 2015.

[83] J. P. Ahrens, B. Geveci, and C. C. Law, "Paraview: An end-user tool for large-data visualization," in *The Visualization Handbook*, 2005.

[84] J. P. Ahrens, S. Jourdain, P. O'Leary, J. M. Patchett, D. H. Rogers, and M. R. Petersen, "An image-based approach to extreme scale in situ visualization and analysis," *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 424–434, 2014.

[85] A. D. Malony, M. Larsen, K. A. Huck, C. Wood, S. Sane, and H. Childs, "When parallel performance measurement and analysis meets in situ analytics and visualization," in *PARCO*, 2019.

[86] M. Larsen, J. P. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison, "The alpine in situ infrastructure: Ascending from the ashes of strawman," *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, 2017.

[87] L. Battle and J. Heer, "Characterizing exploratory visual analysis: A literature review and evaluation of analytic provenance in tableau," *Computer Graphics Forum*, vol. 38, 2019.

[88] N. Matthew, "Why visual analytics?" [Online]. Available: https://www.tableau.com/learn/whitepapers/why-visual-analyticsform

[89] P. Stull-Lane, "Define analytics: The changing role of bi's favorite catch-all term." [Online]. Available: https://www.tableau.com/learn/whitepapers/define-analyticsform

[90] "Tableau and big data: An overview." [Online]. Available: https://www.tableau.com/learn/whitepapers/tableau-big-data-overview

[91] Z. Zhang, K. T. McDonnell, and K. Mueller, "A network-based interface for the exploration of high-dimensional data spaces," *2012 IEEE Pacific Visualization Symposium*, pp. 17–24, 2012.

[92] S. Garg, J. E. Nam, I. V. Ramakrishnan, and K. Mueller, "Model-driven visual analytics," *2008 IEEE Symposium on Visual Analytics Science and Technology*, pp. 19–26, 2008.

[93] M. Isakov, E. d. Rosario, S. Madireddy, P. Balaprakash, P. Carns, R. B. Ross, and M. A. Kinsy, "Hpc i/o throughput bottleneck analysis with explainable local models," ser. SC '20. IEEE Press, 2020.

[94] B. Xie, Y. Huang, J. S. Chase, J. Y. Choi, S. Klasky, J. Lofstead, and S. Oral, "Predicting output performance of a petascale supercomputer," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 181–192. [Online]. Available: https://doi.org/10.1145/3078597.3078614

[95] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. Leung, M. Egele, and K. Coşkun, "Diagnosing performance variations in hpc applications using machine learning," 05 2017, pp. 355–373.

[96] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. M. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Online diagnosis of performance variation in hpc systems using machine learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, pp. 883–896, 2019.

[97] A. Borghesi, A. Libri, L. Benini, and A. Bartolini, "Online anomaly detection in hpc systems," *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 229–233, 2019.

[98] B. Aksar, Y. Zhang, E. Ates, B. Schwaller, O. Aaziz, V. J. Leung, J. M. Brandt, M. Egele, and A. K. Coskun, "Proctor: A semi-supervised performance anomaly diagnosis framework for production hpc systems," in *ISC*, 2021.

[99] X. Zhang, H. Abbasi, K. A. Huck, and A. D. Malony, "Wowmon: A machine learning-based profiler for self-adaptive instrumentation of scientific workflows," in *ICCS*, 2016.

[100] D. A. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, "Apollo: Reusable models for fast, dynamic tuning of input-dependent code," *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 307–316, 2017.

[101] C. Wood, G. Georgakoudis, D. A. Beckingsale, D. Poliakoff, A. Giménez, K. A. Huck, A. D. Malony, and T. Gamblin, "Artemis: Automatic runtime tuning of parallel execution parameters using machine learning," in *ISC*, 2021.

[102] X. Wu, M. Kruse, P. Balaprakash, H. Finkel, P. Hovland, V. Taylor, and M. Hall, "Autotuning polybench benchmarks with llvm clang/polly loop optimization pragmas using bayesian optimization," in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2020, pp. 61–70.

[103] H. Menon, A. Bhatele, and T. Gamblin, "Auto-tuning parameter choices in hpc applications using bayesian optimization," *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 831–840, 2020.

[104] M. R. Wyatt, S. Herbein, T. Gamblin, A. T. Moody, D. H. Ahn, and M. Taufer, "Prionn: Predicting runtime and io using neural networks," *Proceedings of the 47th International Conference on Parallel Processing*, 2018.

[105] A. Marathe, R. Anirudh, N. Jain, A. Bhatele, J. J. Thiagarajan, B. Kailkhura, J.-S. Yeom, B. Rountree, and T. Gamblin, "Performance modeling under resource constraints using deep transfer learning," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[106] V. Sreenivasan, R. Javali, M. W. Hall, P. Balaprakash, T. R. W. Scogland, and B. R. de Supinski, "A framework for enabling openmp autotuning," in *IWOMP*, 2019.

[107] B. van Werkhoven, "Kernel tuner: A search-optimizing gpu code auto-tuner," *Future Gener. Comput. Syst.*, vol. 90, pp. 347–358, 2019.

[108] S. Muralidharan, M. Shantharam, M. W. Hall, M. Garland, and B. Catanzaro, "Nitro: A framework for adaptive code variant tuning," *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 501–512, 2014.

[109] D. Maharjan and P. Zaspel, "Towards data-driven filters in paraview," *ArXiv*, vol. abs/2108.05196, 2021.

[110] S. Leventhal, M. Kim, and D. Pugmire, "Pave: An in situ framework for scientific visualization and machine learning coupling," *2019 IEEE/ACM 5th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-5)*, pp. 8–15, 2019.

[111] H. Ehsan, M. Sharaf, and P. K. Chrysanthis, "Efficient recommendation of aggregate data visualizations," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, pp. 263–277, 2018.

[112] S. Rahman, M. Aliakbarpour, H. Kong, E. Blais, K. Karahalios, A. G. Parameswaran, and R. Rubinfeld, "I've seen "enough": Incrementally improving visualizations to support rapid decision making," *Proc. VLDB Endow.*, vol. 10, pp. 1262–1273, 2017.

[113] Y. Luo, X. Qin, N. Tang, G. Li, and X. Wang, "Deepeye: Creating good data visualizations by keyword search," *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[114] C. Wang, Y. Feng, R. Bodík, A. Cheung, and I. Dillig, "Visualization by example," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1 – 28, 2020.

[115] C. Yang, B. Friesen, T. Kurth, and B. Cook, "Toward automated application profiling on cray systems," 2018.