Online Monitoring for High-Performance Computing Systems

Chad Wood
Department of Computer and Information Science
University of Oregon
Eugene, OR, United States
cdw@cs.uoregon.edu

ABSTRACT

In this work we explore the area of online monitoring systems in high-performance computing. This area of research is increasingly important as software and machines grow in scale and architectural complexity. We begin by outlining the terms of the art and scope of the area being considered. We provide a high-level overview of online monitoring within the context of high-performance computing, including various subtopics. Significant features of each subtopic are discussed, as well as the reasoning behind the integration of these topics into a holistic area of research. This leads into a deeper discussion of the special constraints imposed by high- performance computing, and how various solutions have evolved along with this unique computational landscape. We then provide a survey of the current and prior tools and techniques for online monitoring. Finally, we end this work with a brief discussion of open research areas for significant future efforts in this domain.

KEYWORDS

online, runtime, monitoring, observability, introspection, high-performance computing, HPC, in situ, scalability, exporting, storage, provenance, logging, ensembles, code-coupling, workflows, interactivity, optimization, tuning

Contents

1 INTRODUCTION

The general theme of this paper is that of gaining insights that facilitate greater *productivity* in a high-performance computing context. This is *why* we are interested in online monitoring, analysis, and feedback systems. We will be considering both low-level and high-level aspects of insight and productivity. Note that these terms are intentionally used loosely and relatively in this document, merely to lend a rough sense of scope. The term *low-level* is taken to mean closer to the machine or software engineering. We use *high-level* to indicate something is closer to the application behavior, the science purpose of an application, or the goals of human users or managers.

Insights might be gained by investigating something as low-level as hardware counters and source code performance hotspots, or as high-level as application data dependency graphs, simulation state, or human-in-the-loop evaluation of scientific visualizations. Productivity also refers to a plurality of possible goals. It can indicate low-level enhancements to the use of network resources, application runtimes, communication slack, power utilization, machine temperatures, etc. Productivity can just as well mean making improvements to the correctness of scientific results, the quality and timeliness of reports and images, the speed at which software can be developed and debugged, or the portability of source codes and optimizations between various machines.

High-performance computing (**HPC**) refers to a specialized branch of computing traditionally used to tackle problems too large to be effectively solved using commodity computational resources. HPC architectures often couple powerful integrated compute nodes together using a high-speed interconnect. The HPC systems we are concerned with in our area of research almost exclusively run a variant of the POSIX-compliant Linux operating systems. The operating system of each compute node runs various services and specialized hardware drivers that allow applications to take advantage the resources which are distributed across several nodes. Such services usually include:

- Networking and shared memory region APIs: Allows for applications, libraries, and services to communicate with each other. This communication can take place within a single physical resources, or between processes running on different devices.
- Batch Manager (ex. *IBM® Job Step Manager* [76] or the *Slurm Workload Manager* [96]): Queues, allocates, launches, and manages user's jobs in a batch scheduling environment, breaking apart a parallel task into ranks and establishing a shared runtime environment that may span one or more nodes.
- Network Filesystem (ex. *Lustre* [14] or *IBM®'s General Parallel Filesystem* [69]): Provides a coherent filesystem view across many nodes in parallel, where reads and write to the filesystem from multiple ranks are eventually synchronized and available to all nodes within a parallel job's allocation.
- Message Passing Interface (*MPI* [31]): Allows ranks of applications to communicate amongst themselves and coordinate their activity via point-to-point messaging and safe synchronous collective data operations.

We will look at how these common HPC software resources, among others, can be exploited for our monitoring, analysis, and feedback purposes.

This paper explores an intersection of three different topics: *monitoring*, *analysis*, and *feedback*. We use the term feedback to imply interacting with applications and execution environments, based on analysis of monitored information, potentially within the same job being monitored.

Modern HPC has introduced extreme scale parallelism, large and complex codes, interactivity between coupled software components, and an unprecedented velocity of data creation, consumption, and displacement. The introduction of these changes has given rise to new computational models, performance paradigms, design challenges, and research possibilities. These recent developments in HPC have both *created new roles for* and also *expanded the prior roles* of monitoring, analysis and feedback.

It is important to understanding the structure of this effort that we are ultimately building towards the current state of the art where these three topics are able (and desired) to be integrated. At times we will discuss monitoring, analysis, or feedback as a standalone topic, but will attempt to explain why the coverage is only partial in a those specific moments as a way of giving insight into the computing landscape at the time of that prior work. Always bear in mind that we are building to what exists in the present features of HPC research in this area.

2 OBSERVABILITY

Before something can be monitored, analyzed, or utilized online, it needs to be *observable*.

Computation involves applying operations to a set of data inputs in order to transform that data according to those operation's stable rules, resulting in reliable and reproducible output. The output produced by a piece of software can be used to validate limited but crucial properties of that software, such as its mathematical precision or the correctness of the computed results compared to a trusted independent measure.

But what of the behavior of the software itself?

By the time an application has completed its work and generated its output, information about the execution of an application that is not observed and stored is lost. Observations such as the basic behavior of the software and the efficiency of its algorithms, and the interactions of its internal components and external execution environment. Information relevant to the performance of an application may include a variety of data sources, both within and external to the application. In order to make informed decisions regarding the behavior of an application, this behavior needs to be observed, annotated, and stored for later use.

Observability is a critical first step into online monitoring, but it is worth noting that points where something is made available for monitoring are often also points were feedback from analysis can be applied.

The depth and significance of observation will vary based on the method, completeness, and invasiveness of the techniques employed. Observability can be achieved or enhanced by a variety of techniques, principally including:

- Application Source Instrumentation
- Shared Library, Runtime, or Service Instrumentation
- Sampling and Tracing
- Probing and Inferences from Indirect Sources

We will now discuss each of the preceding techniques in turn.

2.1 Application Source Instrumentation

Instructions to capture observations, compiled directly into the executable code of an application.

Software source code can be instrumented to self-report its progress from state to state. Such instrumentation takes the form of function calls (or macros) that are embedded in-line between normal application code. Once instrumentation is in place, it is encountered and evaluated during the normal course of application execution. This placement can be done by hand, embedding direct calls to some annotation API, or it can be done programmatically by an automatic code instrumentation tool. Hand-instrumentation is more invasive and labor-intensive than tool-based instrumentation, requiring developer time and expertise. In exchange for the extra work to emplace and maintain it, there are some added benefits to using hand-instrumentation over tool-based solutions. By selectively instrumenting specific code regions, a developer can minimize the cost of observing a piece of software. Since no code can be observed without the computer doing a little bit of extra work, doing too much of this extra work will mask off the underlying application behaviors of interest. A developer can use their judgement to skip the observation of areas that are not of interest, or that are executed so frequently that the overhead of making observations would dominate any application performance that could be observed.

Hand-instrumentation is also able to introduce *high-level annotations* to the observed low-level execution features. High-level annotations are essentially labels which identify the nature or purpose of the region of code being observed. They allow for that data to be quickly individuated from other observations, to allow for efficient categorization and analysis (to be discussed in later sections). Developers do not always know what regions of their code are important, or the code that is having the most significant impacts on the applications' behavior will change as the codebase evolves or new inputs are fed into the program. To remedy this, it is useful to have a variety of mechanisms available to observe and explore the performance of an application. Common instrumentation interfaces [13] being embedded in codes show promise in this regard. They provide a point at which many different performance tools can be attached and activated to provide observation of the software, without needing to edit code or recompile applications. When not in use, these instrumentation interfaces would not impose any significant overhead. It will be interesting to see whether this idea gains broad support going forward.

Because instrumentation involves inserting extra instructions into code, regardless of the kind of instrumentation that is in place, it is sometimes desirable to temporarily disable it. Instrumentation is typically disabled when code moves from being actively developed and optimized into a "production" scenario where maximum efficiency is desired and introspection of application behavior is less important. To this end, it is important for instrumentation to have an "off switch" of some kind. One option is to excise the instrumentation from the application code at compile time, so that the source code remains instrumented, but those blocks of instructions are skipped over by the compiler and do not appear in the application binary at all. Recompilation can be costly, but will yield the most

efficient application binary. Another option is to disable code with a setting that can be checked by a program in execution. This means leaving the instrumentation in the code, but while the code is running, whenever some instrumentation is encountered, first have it check to see if it has been disabled, and then if so skip over the block of instrumentation code and resume normal execution. With this method, it is important to be able to "do nothing, quickly", so that the impact of the extant (disabled) instrumentation is minimized. This would be an appropriate method to use if the instrumentation were not directly embedded in an application code, but emplaced in the code of a shared library or service that an application makes use of. When the application is recompiled, that library or service might not be, so runtime enabling or disabling of instrumentation is required if a system is to be observable in this way. We will now discuss that scenario.

2.2 Shared Library, Runtime, or Service Instrumentation

Making observations within the code which is executed when an application makes API calls to an external library or service, or when a runtime platform is evaluating collections of instructions (code or queries).

HPC software is often built up out of multiple libraries being interacted with by the core logic of an application. When an application is launched by the operating system, any libraries that it has been linked into will also be loaded into memory and initialized. `C` and `C++`, for example, offer standard libraries which provide many features essential to applications written in those languages, from collections of optimized data structures to network and multithreading routines. Higher-level libraries exist to provide domain-specific features, such as *SAMRAI: Structured Adaptive Mesh Refinement Application Infrastructure* [91], an implementation of optimized data structures and algorithms of general utility for adaptive mesh refinement codes, common in physics simulations. In the absence of direct application source code instrumentation, shared libraries can be good targets for observing application behavior. Through calls into the API of that library, execution will pass into the code compiled within it. Any instrumentation within will then be executed.

As will be discussed in later sections, especially "Exposing Data" (§ 3.4), the information that is generated or observed by instrumentation needs to either be made available for use, or stored to be used elsewhere or at another time. Control flow through the execution of a program binary is typically fixed at compile time, where the operating system will establish the basic execution environment and protected memory, initialize the stack, and begin execution at the designated starting function of a program, `main` in the `C` family of languages, for example. This process does not automatically provide hooks for tools to be initialized or optional accessory services to be started. In the case that memory needs to be allocated for storage, or services need to be invoked that can capture and operate on monitored information, the shared library instrumentation path offers another useful engineering options: *static singletons*. This refers to a `C++` language convention where code objects can be can be marked as `static` and be executed at program initialization, and through clever means cause the initialization of a class that uses the *singleton* design pattern, where only one instance of an object is allowed to exist. This combination of techniques allows a shared library to execute some initialization routines at the beginning of a program, merely by being linked into the program and loaded when that program starts.

Two common ways to instrument libraries by adding code are to pre-load a surrogate (or *wrapper*) library, or provide customized header files that implement some instrumentation. Wrapper library instrumentation can also be achieved without needing to recompile an application. `LD_PRELOAD` is a special environment variable supported by the Linux operating system. When paths to shared libraries are set into that variable, those libraries will also be loaded by the operating system when an application is launched. This can be used to to flexibly provide instrumentation around existing libraries without needing to access their source or recompile them. The wrapper library should expose all of the same function signatures to the invoking application, such that normal API calls to the shared library will instead invoke the same function in the wrapper. On its first invocation, the wrapper can then manually load the normal shared library and populate a table of function pointers all of the normally-exposed functions within. As the normal library's functions are called and return back, the wrapper is able to track these timings and perform any other desired instrumentation or monitoring-layer interactions desired.

If recompilation of the application is not a burden, customized header files are also an option, and can impose a slightly lower performance impact as less runtime activity is required to resolve API calls. Customized header files will require the calling application to be recompiled, and its source code or build scripts updated to point to that custom header file. Within the header file, functions can be implemented instead of merely defined, and these functions can embed instrumentation around calls to the normal library. The header file technique usually requires an API to be expressed in two layers, with a public-facing API wrapping calls to an internal implementation API. This avoids the problem of *namespace collision*, what occurs when an object has two different definitions within the same callable scope, and the compiler does not know which of the two is being referenced as it attempts to build or link the software.

A successful example of header file instrumentation in the real world is provided by the MPI codebase. MPI's public API calls all begin with `MPI_` and every such call jumps into a tiny wrapper function that immediately and only calls its implementation function, which is prefixed `PMPI_` and which provides the actual implementation code. Developers can add code to the wrapper functions in that header file, as a way of intercepting calls to the MPI routines.

Both of these technique are able to facilitate a variety of advanced interactivity, such as making adjustments to the parameters being passed through the wrapper into the normal library, or changing the behavior of the normal library based on some performance observations or goals.

2.3 Runtimes and Services

Many HPC applications take advantage of standardized libraries and packages designed to grant traditionally-engineered software access to the unique advantages enabled by HPC hardware, without requiring wholesale re-writes.

One such library is OpenMP [17], which presents a standard for annotating, or "decorating", the parallel regions of a block of code, and then compiler extensions which can intelligently and safely adapt the code according to those notes for it to be automatically parallelized. The primary mechanism for parallelizing codes that OpenMP uses is the spawning of multiple threads, distribution of data between those threads, and the gathering of the results produced in parallel back into a unified memory location for processing by the serial portions of the program. In addition to the injection of inline codes, OpenMP provided a flyweight runtime within the process, to manage the creation and destruction of threads, or teams of threads, achieving safe management of the memory regions those threads were operating over.

Automatic code generation, especially in this case where it profoundly altered the characteristics of the code's execution, introduced some complexity to the various source-instrumentation-based means of observing codes, though techniques were developed [46] to address this. Iterating over the years, as more modern generations of the OpenMP frameworks were designed, a tools interface named OMPT [19] was added to OpenMP to provide an organized and flexible means of interacting with the OpenMP runtime and observing the application, providing hooks into the normal semantics of the program as well as events unique to the internal activity of the OpenMP runtime. One especially useful feature of the OMPT interface is that tools can be enabled or disabled at runtime, not requiring an application to be recompiled from source. For the many HPC applications which make use of OpenMP, this interface can be a useful source of information for online monitoring frameworks.

Elaborated further under "Distributed Computing" (§ 5.1.1), the Message Passing Interface (MPI) runtime can be an indispensible resource when monitoring HPC applications. In addition to posessing a number of valuable datum related to the execution of a single distributed task, the runtime is also potentially managing many other processes distributed across the machine concurrently. MPI is aware not only of some performance measures of the application, but of its own configuration and performance. With higher-level permissions and a common observational infrastructure, it is possible to observe complex interactions between parallel jobs of parallel processes in situ and online [10], observations which by necessity require shared service-level instrumentation and online monitoring.

Task-based runtimes or applications written using *partitioned global address space* (PGAS) languages [95] [18], like HPX [35] or Charm++ [36], or even distributed workflow managers such as Swift/T [93], can make it difficult to cleanly separate out the workings of the runtime service layer from the program that the service layer is facilitating the execution of. That is, a program can be broken up into so many different asynchronous parts that traditional monitoring patterns do not effectively capture coherent or developer-relevant performance data. The ratio of monitoring overhead to the overall productive work performed by the application can quickly become undesirable, especially if tasks are dispatched and retired at a very fine-grain, and have short lifespans. Tools such as the Autonomic Performance Environment for Exascale (APEX) [60] have been developed specifically to address [48] many of these challenges, but it remains an open area of research.

Another way to observe processes in vivo is by stepping outside of their execution environment entirely, and then turning around to look back in. This is most often observed in cases of commercial "cloud computing", where a system image is hosted by a virtual machine hypervisor, and applications are run within that virtualized environment. Through extensions to the hypervisor agent, such as with the ongoing work with Xen introspection extensions [82], the performance, progress, or various other information can be observed from outside the runtime instance with only relatively small increases in overhead compared to running unobserved within the virtual machine. These increases in overhead would be proportional to the overhead of monitoring the same application running natively on physical hardware.

2.4 Sampling and Tracing

Exploiting binary formats, memory layout conventions, and explicit operating system APIs to apply instrumentation to a compiled application without modifying its source code.

Sampling means inspecting the state of an application and reading the available performance counters provided by the operating system. It is usually performed at some regular interval of time, so inferences can be made about the activity that transpired between those intervals, and the impact those activities had on the sampled parameters. Sampling is by far the most efficient method for gathering observations to use when monitoring an HPC system, and as such is favored for online monitoring systems. Sampling can be done directly by a tool, by making calls to the Linux operating system's `perf_events` API, reading counters from the `/proc/stats` virtual filesystem furnished by the operating system kernel, or through registering counters of interest and making inquiry into a pre-packaged introspection tool like PAPI [51].

Tracing deviates from sampling in that *every single action* an application takes has the opportunity to be counted as a significant event and measured, though each action might not be of interest. In order to achieve the extremely fine-grained analysis afforded by tracing, many additional instructions are inserted around those of the application, able to capture and count the application's instructions and follow the control flow's branching paths through the application logic based on the input data and the evolving results of computation at runtime.

Traces often have orders of magnitude higher overhead to gather than performance measures arrived at through sampling. Hand-annotated source code has the added benefit (and developer overhead) of an expert identifying the significant regions of an application, so that uninteresting information does not need to be collected or analyzed. This lowers the overhead of performing a trace, in both time spent gathering measurements, and by reducing the space required to store any performance measurements. It also allows for code regions to be intelligently named for quick identification, for cases where a person is utilizing a monitoring system, and such insights can be exploited for code tuning, etc.

Tracing can be performed over specific domains of application events, such as tracing only the I/O of an application, the loading and storing of regions of system memory, or just generally serarching for latency [65]. One could choose to trace only the interactions

between an application and the operating system kernel, or even to trace only the activity within the kernel.

While outside the scope of this survey, note that there are many types and implementations of tracers [27] available to trace both kernel and user-space activity, including ftrace [11] [26], perf [53], LTTng [25] [16], and some commercial offerings such as Intel PT ("Processor Trace"), etc. Much of the lower level tracing infrastructure, such as the Linux `perf_events` subsystem, is available to be used in other user-space tools like Valgrind [55] or PAPI [51] to provide aspects of their overall performance information set, including data associated with branches and traces.

2.5 Probing and Inference from Indirect Sources

Combining multiple external sources and epochs of information to form intuitions about the behavior of a system and its components.

There are a variety of questions an interested party may wish for their monitoring system to answer that, while requiring online monitoring, are not well-suited to the mechanics, scope, or frequency of events which are revealed by directly observing a single application, or even a single instance of a complex workflow, as the source of information. Some examples:

- On average, how long are jobs waiting in dispatch queues before being launched, including as ratios of their actual and requested runtime?
- What portion of a job's occupancy is spent waiting on shared resources to become available (i.e. physical tape archives of large data sets that need to be fetched and brought online by an automated robot)?
- How much do the power requirements of the entire facility deviate through the day, and is there a correlation with specific jobs, or machine workloads, or the exterior environment's temperature and humidity?
- How often do the processors on the nodes slow their clock rate in order to stay within their configured thermal envelope?
- What portion of the energy budget of the total machine (and its enclave within the broader HPC facility) is spent on controlling temperature, vs. on providing compute capability?
- Which applications, and at what allocation sizes, result in the greatest amount of contention for shared resources like the network interconnect?
- When job occupancy is high and network congestion is low, but CPU or GPU utilization is also low, what are the jobs that are running at that time, to inspect for some bottleneck which is preventing codes from fully exploiting the available hardware?
- How much do identical measurements vary across nodes, and how much do identical measurements vary for each node across time?
- Are there any deviations from normal performance measures that can be accurate predictors of pending hardware failure?
- How often are job walltime limits reached, and how often do jobs terminate (successfully) without using some significant portion of the walltime that they had requested?
- What are the most used system libraries, compiler versions, and versions of applications?
- What are the most frequent causes of a program being terminated by the operating system?
- What are the least-utilized components of the total machine architecture?

These are just a handful of such questions, by no means a comprehensive list. What may stand out in that list is the frequency with which the word "job" appears.

Often some measure of interest will not be observable without increasing the sample size beyond the one application or workflow that a user may have enabled instrumentation for. Observations of multiple programs and also observations of sources outside of the scope of applications are needed to answer most such questions. Some of these observations can be accessed within its context using open-source toolkits or system APIs, while other data points may get emitted from a vendor's proprietary drivers, and one must write tools to get access to and appropriately contextualize this information. Moreover, all information needs to be gathered continually, online, and retained over various epochs, in order to be interrogated later on to yield answers not anticipated by the developer who originally made some information observable in the first place.

Looking at online monitoring for HPC from a holistic perspective like this allows for interesting questions to be asked and answered, but the necessary software and sensor infrastructure gets complicated, invasive, and expensive, very quickly.

Here we can see yet again that monitoring systems serve a variety of purposes, and so their deployment and use will have a diversity of motivations. An application developer is unlikely to be personally concerned with the thermal consequences of using a high-speed solver library that activates additional circuitry and causes more heat to be dumped by a compute core over the duration of their job. The types of jobs which coorellate with an increased load on the cooling infrastructure, and the peaks and valleys and averages of such thermal readings, likely *will* be of interest to someone who is tasked with managing a machine, budgeting for power, or maximizing the longevity of machine parts.

Gathering and making use of these data sets means taking on a wide array of engineering and design challenges, many of which are discussed in the next section and later areas. One such challenge has to do with the diversity of epochs and frequencies of measurements, and the need to capture and compose information efficiently. Thermal readings and power settings can be measured from a compute core in tiny fractions of a second, whereas some facility-wide sensors may have significant hystersis in reporting and only be updated every several minutes. This means that short-lived events can be more difficult to make accurate judgements about, for example. When composed against and considering their influence on the longer timelines described by coarse-grained measurements like power draw readings for a row of server racks, average ambient air temperature around a row of servers, or the power draw of the HVAC system responsible for cooling the entire building, etc., such short-lived events are difficult to render judgements about. They are also typically not able to be efficiently stored in any detail over many jobs or longer periods of time, to allow for sophisticated meta-analysis, though there are some serious efforts to do just this,

such as the Sonar [39] [29] project at Lawrence Livermore National Laboratory.

When integrating observations made at different system layers and produced by different development teams, the semantics of what is being reported can vary widely, and must be carefully considered when composing data. One sensor might be reporting *absolute temperature* in Kelvin, and another may be reporting the *delta between two temperatures* in Farenheit. One must record units of measure at some datum's origin, or have a brokered ingestion of information into the monitoring system, such that various sources are processed by bespoke aggregation functions to be made available as normalized statistical metrics.

The complexities inherent to online monitoring systems quickly become apparent, especially *as the monitoring need grows beyond a single point of measurement or is desired to fulfill more than a single purpose.* From this understanding, it becomes relevant to more deeply explore the topic of capturing and using information.

3 CAPTURING AND USING DATA

Once an event or some state in an HPC system has been observed, it must be represented in a stable format to be useful. Our practical research interest is in the type of data that can be accumulated or streamed through algorithms to discover and react to trends and patterns. This sort of data can usually be stored for reference or data-mining as a member of a set of data that can span multiple scales or epochs, being combinable to reveal facts beyond what is locally available during the immediate execution of an isolated process.

3.1 Overview

> *Representing, disclosing, aggregating, storing, and accessing observed facts about processes, configurations, input data, activity, and the HPC execution environment.*

This section focuses on the mechanics of making and using data out of something that has previously been rendered observable, in one or several of the ways outlined in the prior section. In order to adequately characterize the sort of data we are interested in, something will need to be said about each of the aspects listed here:

- Representation and Meaning
- Patterns Within HPC
- Exposing or Exporting Data
- Introspection, Opacity, and Interface Standardization

Still, it is worth pointing out that not all observations have the same complexity or purpose. For example, some observations do not need to be retained or even exported from a process to be useful. Those observations may be temporarily fixed and used to inform a process-local or immediate decision-making process, and then discarded or overwritten. Such transient observations still must be encoded and accessible in a coherent format to be utilized – even by logic within the same process. A discussion of the fullest life-cycle of data sets from observations will also serve to inform the treatment of data sets with more limited purposes and characterization requirements.

So what do we do, once something is observable?

3.2 Representation and Meaning

It is important in all journeys to start off in the direction of one's goal. Any eventual application of observations will be counting on the observations being correct, consistent, and precise. Further, information must include not only the measurements, but some standard notion of interpreting the measurements. Mistakes or omissions here in this fundamental consideration can invalidate or undermine the entire purpose of monitoring HPC phenomena.

An engineering specification that only included the numeric component of measurements for its dimensions might give a clue about the proportions of the design in reference to itself, but would not be helpful to understand its overall scale in relationship to its environment or other engineered objects. This would lead to the design object being difficult or impossible to accurately reproduce, or for people unfamiliar with it to have useful intuitions about its purpose or place simply by looking at that partially-annotated specification. Perhaps the simplest notion of a standard for interpreting measurements is the expression of *units of measure*, noting what "1" means in terms of units of length, volume, temperature, chronological element, etc. Once the standard for one unit is expressed, all measurements of that type can be scaled off of that unit. Time can pass in seconds, or in milliseconds, or in days, or even be denoted by abstract and unscaled CPU "ticks" within the context of a single architecture.

Units of measure can themselves be complex entities. Knowing that some numerical representation refers to a temperature in Celsius may only tell you half the story. A measurement may be referring to:

- an observation of an event or state at one point in time
- rate of change between two points in time
- result of a function relating multiple observations across time or domains

When considering an application for performance data like *constructing performance models* using machine learning methods, it is worth noting that some forms of machine learning are designed to function well over completely opaque or unannotated data sets, such as deep learning using neural networks. Typically the overall data set this type of learning is applied to has at least been pre-filtered and organized into a regular structure by some domain expert to contain distinctions of likely relevance presented in a consistent layout, to allow for the learning algorithm to recognize and adapt to some notion of concepts or categories within this unannotated data. There are always trade/offs to be made regarding the selection of machine learning algorithms, such as speed, overhead, accuracy, timeliness, consistency of input data layout, and the amount of data needed to make good decisions. For now let us assume that information is needed for purposes beyond training deep learning models, and so correct annotations will have importance across a variety of purposes, and look at what is entailed by that idea.

3.2.1 Encoding the Data and Metadata. The simplest things can go unnoticed but be deeply important. One of these is the way in which information is encoded. In addition to storing a value for the measurement of a temperature, and having some way of knowing it refers to a change in temperature for some epoch of time, it matters

how that floating point value is encoded. For example, floating point values can be stored in the condensed IEEE 754 formatting, where there are special meanings for subsequences of bits in the byte words of a 16, 32, or 64-bit encoding. This format strikes a balanced trade/off between storage and representational accuracy, and is how most floating point numbers are stored and operated over from the perspective of a CPU. If the number were to be pulled up in an ASCII text editor and reviewed by a human, it is unlikely that they would be able to determine the precise floating point value with their manual review. Numerical values can be projected out into a character string, which is much easier for a human to understand, but consumes much more storage space, and cannot be processed for mathematical operations by a CPU without converting back into the IEEE 754 encoding, potentially decaying the accuracy in the process.

In addition to the *encoding* of observations, the formatting of *multiple observations* bundled together is a significant factor in that information's openness to exploitation. Opaque file formats, or undocumented network protocols, bearing messages or observations, can be difficult or impossible to exploit, if the information has not been orchestrated into some consistent arrangement. There do exist remedies for this, with formatting standards like CSV, YAML, XML, or JSON. These standards make no assumption about the meaning or semantics of the data they contain, but they do impose rules on how data generally will be encoded, so that at a minimum the raw values can be parsed from the collection into its individual components through consistent mechanisms.

There are higher-level standards which emerge from more fundamental encoding standards, such as the Open Trace Format (OTF) [38] [41] [21], which is purpose-built to store the performance measurements of HPC applications.

3.2.2 Encoding the Expertise. There are many kinds of expertise in the HPC field. For our purposes, we will focus on three:

- Users
- Developers
- Optimizers

The users of HPC applications, especially in the scientific community, such as the Dept. of Energy (DOE), are often domain experts. Users will have a deep understanding of the purpose of an application, what it is that the software system is helping them to explore, understand, or control. A user can also be thought of as a stakeholder, or someone who approves funding for projects, manages a budget that covers a machine, an entire premise, or who needs to make decisions balancing the purpose of the software with the overhead and mechanisms of building and maintaining that software. Developers need to be experts in the mechanics of software architecture generally, from design to deployment to long-term software integrations and standards. They understand the process of designing codes, connecting components together, interpreting compiler error messages, etc. Developers are sometimes also domain experts, and users of codes, and they are usually motivated to write code that runs reasonably optimally, though it is not as incentivized as correctness. Optimizers are developers whose role is less about building software to meet the needs of a user, but about maintaining the effectiveness of codes over time. This means porting codes to new architectures, tuning adjustable parameters to best exploit the hardware to achieve the computational task.

No person can perfectly prognosticate about future architectural evolution and its specific optimizations for any given algorithm, so there is always a role for personnel who specialize in the tasks of porting and tuning codes. Generally, it is a fuzzy distinction, but it can be said that application developers are primarily rated on their application running correctly, consistently, and are not primarily tasked with maximizing the performance of codes, where the role of an optimizer of codes is to facilitate maximum performance, as well as code lifespan through portability to novel architectures.

The roles of user, developer, and optimizer will each have intersecting but distinct domains of expertise. What people consider important, when it comes to observations made about HPC systems and software, will be strongly influenced by a person's various responsibilities and their areas of expertise. Some examples of motives:

Users or system stakeholders might be more interested in minimizing the time their jobs sit in batch queues, or in the failure rate of parts, or in network congestion or other metrics related to shared resources. They may want to know things like machine temperature, or what versions of codes are being run the most. *Application developers* might be interested in using local and remote system state or application progress to make better decisions about task assignments or dynamic simulation domain refinement within a simulation step. They may be interested in using in situ runtime services to couple together workflows out of legacy components that are not engineered by themselves to be coupled together asynchrously, and concepts of direct in situ monitoring at the application level come into play.

Optimizers can be interested in data at all sorts of levels of detail. They may wish to observe the frequency with which a function is called during a run, or its average evaluation time. They may wish to see how much time is spent in application logic vs. in the system libraries the application makes use of, seeking places where optimization can be found. They may need to observe the behaviors of a complex workflow in situ and at scale, to find performance bottlenecks that only emerge online, during the course of a run, and are not readily apparent through offline static analysis or manual code review. Drilling into deep and invasive observations, optimizers may need to record high-frequency samples of measurements, pathtrace data to map the flow of execution, or correlate full sets of application data with batteries of performance observations generated when those inputs were being processed. This can be especially important when porting codes to a novel architecture that may offer general compatibility with the previous, but have significantly different resources types and capacities, such that an optimizer must observe how the detailed internal components of a large complex application are occupying the machine and how [in]efficiently it is running as a whole. Discovering optimal compilation options is sometimes as significant a contributor to performance gains as is learning the optimal runtime tuning parameters.

Differences in expertise lead to different priorities and values, and this means that every system will involve trade/offs in terms of implementation, integration, runtime overhead and application perturbation, since there is no free lunch. This justifies the important design priority for online systems that can be selectively enabled

and/or invasive, and that offer some general utility across multiple domains of expertise.

3.2.3 Time, Change, Identity, and Consistency. The continuous interactions of discrete elements, and the ability to reason about observations of change over time, is central to the purpose of online monitoring systems. A brief detour to discuss these concepts and terminology is warranted, given their constant presence in background of this entire area of study. It is not the purpose of this paper to give a thorough examination of these delicate and important conceptual underpinnings. Rather it can be said that since we will make use of the concepts mentioned in this section without completely justifying them, we wish to be reasonably clear about what is understood.

Time is a fundamental dimension of analysis for the study of computational performance. This is true in both obvious and subtle ways. One obvious way time in itself is a factor is in the definition of the objective function that directs performance tuning choices: When a region of code executes in less time, it could be considered to be more optimal than code which takes longer to execute. Normally the distinctions discussed here are implicit to the examination of code performance or the design of systems that monitor and evaluate it. It is worth taking a moment to pause and consider the subtler manner in which time is fundamental to observability, because of the profound ramifications that it has on short-term measurement obligations and the feasibility of long-term objectives.

For change (or similarity) to be observed, there must be something to compare an immediate observation to. A subtle and contingent way then that time is a fundamental consideration can emerge when one reflects on the semantic nature of observed phenomena, and then also on the identity of phenomena as both a type and a token. At this point in the discussion the level of detail or relevance to performance is not significant, merely the formal precondition for consistency.

Synchronic consistency refers to the *stability of meaning* for phenomena that are fixed within a single epoch, regardless of that epoch's unit of measure. These are coocurrent entities, that is to say, distinct observational artifacts that are claims about states or events that were extant within an interval of measure. The priority of synchronic consistency is the semantic load, or the meaning, of these observations. Synchronically consistent datum that are denoted as being the same type of event would connote a consistent meaning, or use a compatible scale of measurements, etc.

Diachronic consistency on the other hand speaks to the sense that as some phenomena are observed across epochs of time, *the identity of the observed* thing, state, or event, is apparent and conserved. In order to establish this kind of consistency, some provenance needs to be included as a component of the observations. This requires the observed phenomena to endure long enough to be named uniquely and be distinguishable from other similar entities. In simple terms: The story that is being observed and recorded may be changing, but the character that this story is about is the same character at every point in the story.

Consistency is generally assumed by designers and developers when working within their own projects. That is to say, most observational systems both assume and combine synchronic and diachronic consistency, because they were built for a singular purpose or by a team driven by a shared motivation. As such, consistency is often silently imported, and exists as a mere assumption. One reason to be aware of these assumptions is that without attention, the assumptions can become false assertions, and the observations then fail to reflect the truth or support comparison with other observations thought to be of the same nature. No system of monitoring can capture every aspect of everything that is true at every time, the system itself would come to dominate its own observations and lead to a nonsensical infinite regress. Trade/offs have to be made about the amount of specificity that is tracked in a system in order to provide safeguards to ensure synchronic and diachronic consistency. These specifiers, or meta-observations, should be chosen to maximize their value in lending stability to the observed performance phenomena they are correlated with, in order to justify the overhead of capturing and retaining them.

Meta-observations can be thought of as *qualifiers* that say things about an observation, as well as the observed thing, helping to distinguish both the reference and referent. Qualifiers help to establish and refine functional categories, prevent contradictions, and give hand-holds to grasp and utilize the observations for practical purposes. These qualifiers need something stable to be tagged to, and this is the object of the consistency described in the above paragraphs. Here of course we encounter a regress of rigors, where qualifiers need something attached to in order to enunciate for that thing its stability of identity and meaning, and yet *the qualifiers themselves seem as though they would need qualifiers in order to be stable in the meaning or identity they are capable of conferring*. This regress of observations and meta-data qualifiers is potentially infinite in theory, but in practice is rarely deeper than one or two layers of abstraction, and so does not emerge as problematic in most systems.

One way in which synchronic and diachronic consistency can be thought of is in the language of ontology, in the differentiation of *types and tokens*. There emerge many differences in the kinds of knowledge we might have, and the kinds of claims that our systems of observation might make about the observed. Types are used to enunciate *what something is*, while tokens are used to enunciate *that something is*. The distinction between knowledge-of and knowledge-that is useful for further unpacking the distinct between the synchronic and diachronic.

The establishing of *types* is a way of encoding knowledge synchronically. We might know it is true that `2 + 2 = 4`, but that does not tell us that `"4"` exists somewhere in the world, or that `4` of something exists or that something happened `4` times. For an easy to grasp example, let's think of how one knows *what a tiger is*. We can have some idea of what a tiger is, for example, being a big cat with striped fur, without there needing to be a tiger nearby to point to as a means of providing a more robust ostensive definition. In this sense, we can construct some kinds of knowledge about what a tiger is by assembling other concepts productively to establish a new *type*. This type of thing is a cat, it is a big cat, it is a striped cat, etc. Types can be as simple or as specific as needed for practical purposes.

When once we observe something, we may identify it and wish to make note of it. What we are identifying at that point is a *token* instance of that type of thing. We can say that "*A tiger is over there*

drinking water, on the other side of the river..." This form of tiger-knowledge entails an existential claim, that not only does *a* tiger possibly exist, but *that* specific tiger exists, at a certain point in time and space. Further, it should be mentioned explicitly here that one does not need to know the type of a thing in order to know that the thing exists, but in order to reason or communicate about that existential knowlege,some type will necessarily be applied to it. When this happens, this observation of an un-typed thing, observers will often reach for more general types and begin an ad-hoc construction of compound type: "*Big stripey animal thing over there...*" Particular to our topic of performance observations within HPC, this automatic attribution of types and identities to "existential knowledge" can be a major factor in general limiting our ability to utilized those observations for analysis, optimization, and feedback.

Ending this pedantic excursion into the realm of the abstract, and returning to the applied topic of online monitoring for HPC, this discussion of time, types, and identities must remain incomplete. Hopefully the distinctions called out in this section will lend some clarity to the reasoning that is done elsewhere in this text, regarding the formal requirements for systems of observation. When an online monitoring system makes observations, an essential aspect of what gets observed and recorded must be these qualifiers that establish some stability in the type of thing, and in the identity of things over time.

3.2.4 *Combination and Unit Semantics.*

Domains, complexity, incompatibility, and a brief look at one approach to the challenge by the Scrubjay project.

As has been mentioned several times so far, there is a wide dissaray of data in the HPC universe. Data can represent activity or state from differing domains, sourced at different intervals, from various tools, encoded in unique formats, with varying degrees of online accessibility. Development tools like TAU [72] or HPCToolkit [6] will describe the application domain, where other technologies like Ravel [87] [63] and Multipath Internet Protocol (MPIP) [81] can both describe and adapt activity in the networking or interconnect domain. Other information can be drawn from facility monitoring sensors, vendor introspection APIs for racks, power, and thermal management, etc.

There are various *Operational Data Analytics* (ODA) platforms that exist, some of which will be discussed later in "Monitoring for HPC: Dedicated Frameworks" (§ 4). For the most part, these integrated monitoring solutions are targeted to serve the needs of users from a particular domain, such as machine administrators, and do not offer value to users in other domains than that for which the ODA was not designed. Having a specific purpose, these monitoring solutions will often have built-in data processing routines, visualizations, logs, and reports which apply aggregation, transformation, and presentation of a priori designated domain-appropriate information.

It is interesting to conceive of a scenario where all monitored information could be retained and made available for many kinds of online and offline purposes, not limited in utility to a single domain, and where the set of data had not been transformed or aggregated in ways that would restrict its ability to be used to answer questions not anticipated at the time the system was deployed or metrics were gathered. While the computational and data storage resources necessary for such a system are themselves not trivial and would represent a significant investment of time and capital to field in production, another concern emerges regarding the dissaray of information. This is especially true when all of the data is not being prepackaged for an ODA infrastructure.

Solving the challenge of complex combinations of units and data semantics remains an open research area within HPC. A notworthy contribution to this topic was made by the ScrubJay [28] project. Scrubjay provided a constellation of tools to gather, store, annotate, and process queries over precisely the complex types of data we've just described. Effectively, it decouples the collection, representation, and semantics of data.

Scrubjay allows for data gathered from any source to be placed in a *wrapper* which represents it in a common format that can be transported, stored, and queried. Data is useless without meaning being ascribed to it, so Scrubjay also provides a framework for applying *semantics* to the wrapped data. These semantics are reusable, and can be applied automatically to all data arriving from various sources, after they are first annotated manually by a user with some system expertise. These stable semantics provide the basis for composable *derivation* functions, which define rules for inferring information from or computing relationships between various data sets. Because the number of derivations is potentially vast, the final contribution of Scrubjay is a *derivation engine* that navigates this space to efficiently find sequences of derivations that are appropriate to resolve queries over the wrapped and semantically-annotated data.

When queries are processed over this data, results are constructed combining both natural joins as well as *interpolative joins*, which are translations and projections of compatible kinds of data into the semantic categories or units of measure that a user has requesting in their query. When results are delivered, the rules the derivation engine used when making any interpolative joins are also presented, so that the derived results are open to verification and the resulting data set is reprodicible even if additional data or semantics are added to the system in the future which would cause the derivation engine to resolve the same query differently. Unlike a traditional query where a user will specify tables and columns of data, and apply specific join rules and aggregation clauses, Scrubjay provides its own query format. In this novel format, a performance analyst needs only identify a set of data sources, and then an expression of the measurements of interest, specifying the dimensions of the domains, and the dimensions and units of the measurements of interest. Scrubjay then determines whether this request can be satisfied, and if so assembles the resulting data, allowing for it to be passed through additional filtering stages which facilitate classic relational database query semantics.

While the support requirements for the Scrubjay platform are non-trivial, as it relies on a dedicated cluster of HPC servers to store and process continuously streaming site-wide monitoring data, the approach and the tools provided by Scrubjay represent a meaningful step forward in this open research area.

3.3 Patterns Within HPC

Qualifiers and considerations common within HPC scenarios, such as versioning, configuration of operating environments, hardware variability, communication hysterysis, and undifferentiated noise in observations, etc.

Application codes evolve over time, as well as the characteristics of the input data that codes operate over. Underlying the application's code, the operating system code, its version and configuration, the versions of system libraries and vendor drivers, the versions of linked libraries, and the general machine environment can be significant to the performance of codes in execution. Performance observations are often relevantly connected to various combinations of these factors, in addition to direct choices made by application developers and resulting from the algorithms they implement.

Furthermore, user priority level and permissions may have a direct impact on the performance characteristics. Some users may have their codes run transparently on assets that are shared between multiple users, and experience wild performance fluctuations that are completely beyond their ability to influence. Higher-priority users on a machine may delay or even evict lower-priority users, leading to variability in observations that would require this condition to be known in order for a user or automated system to make sense of.

Observations of software in *highly-variable environments* can make it difficult to gain insights into the actual performance characteristics of the software. If the code configuration or input set is always changing, it can be hard to know the ground truth about general system performance. If the relative priority of a user or utilization of shared resources are always in flux, it can mask-off the behavior of particular versions of software. Without some "stable middle" of observations made about any given configuration, and a sequence of observations showing that a centroid of observed values has shifted in one direction of another, it is a challenge to know whether performance was gained or lost by any given change to the system configuration or an application's code.

When the density or distribution of noisey sets of observations are not regular, a case can be made for simply throwing away the irregular or outlying information and using what lies more towards the median. In such cases, using less of what is observed can actually be beneficial to the cause of general performance understanding. It is important to point out though that in order to detect that there is noise, and that the noise is irregularly dense and centered around a stable middle, all of the observations of some epoch under consideration will have to have been made and analyzed. Just because some observation is later deemed to be the result of noise doesn't decrease the importance of it being either exported into a monitoring system, or exposed to inspection.

On that point, let's unpack what it means for something to be exposed or exported.

3.4 Exposing Data

Observability, even in an online sense, does not necessarily require information to be actively moving around within the operating environment or monitoring system. It may be sufficient to the needs of the online monitoring system that various components are available to be interrogated as needed, that is, that relevant performance metrics are merely *exposed* to a monitoring system.

Many sources of information in traditional HPC operating environments are regular system components that exhibit this "inspectability" property. A common way for performance data to be gathered is to inspect the statistics of a running process via operating system API calls, or by interrogating a filesystem abstraction such as `/proc/stats` which makes this data available through the form of memory-mapped files which are continuously updated with new statistics for processes.

In-memory logs and filesystem storage cannot be continuously populated by performance observations, this could consume all available resources over a long enough period of time. Because of the need to not burden the system with it own introspection, information that is made continuously available via exposure to inspectability is also often transient in nature. For example, the *history of values* in `/proc/stats` is not retained forever, it is continuously updated in place, obliterating the prior observations as new observations are made.

This update-in-place behavior imposes limitations to the types of understanding that can be gained, such as preventing the chance to identify that some average performance degradation was due to the interaction of two independent processes both simultaneously bursting with abnormal amounts of activity. Another point is that these systems usually do not retain performance measurements for processes which are no longer actively running on the machine, though they may be contributing in essential ways to the performance of processes which are still running, such as the case of complex scientific workflows that integrate the inputs and outputs of many independent processes, and where metrics are reflecting the behavior of the workflow in its aggregate performance from beginning to end.

While there may be counters and averages that track activity over arbitrary spans of time, the precise update interval, event density, or general distribution of events that are accounted for are not features that can be seen without this exposed information being retained in some way.

3.5 Exporting Data

Given the simple and limited nature of exposing information, in order to do more sophisticated things with data in our monitoring system we must retain it, and this will involve *exporting, or recording and migrating that information between components* of the system. This movement of information can happen in a number of ways. Information could be copied immediately, and in full, over to the receiving component. Or perhaps a lightweight reference to that information might be dispatched, taking the form of an event record or data pointer, to make some other element in the system aware that this information now exists and may be consumed.

In cases of lightweight dispatch of event records or pointers to data, other structures are implied, such as reference management, caching, or queing of records, transaction management services, etc. so that the information those records point to does not get "garbage collected" and vanish unaccountably. There are trade/offs here, like everywhere else, in the balance of retention policies for this information, and the needs of the monitoring system to not

consume too large a share of the resources which are meant to be dedicated to productive computation.

Any time information is moved from one context to another, leaving the boundaries of a process, a shared library, a tool, or a machine, it can be said to be *exported*, for our purposes in describing this research area. Here are some of the primary techniques or models for exporting information:

- logging
- checkpoint
- cacheing
- polling and pulling
- broadcast or push
- hybrid push/pull
- publish/subscribe

We will now discuss each of these techniques in turn.

3.5.1 Logging. Generally this method involves a "fire and forget" or write-only approach to recording information into a monitoring system. Sources that generate data and spool it out into a log do not typically also read back from that log. This allows the logging mechanism to be optimized for low-latency intake of data, such that it does not pause the work of the application any longer than necessary.

The mechanics of logging systems are able to be much simpler than some of the alternatives below, which make this a popular choice for developers who do not have sophisticated observational needs. Logs can be as simple as appending output to an asymptotically growing file containing messages for a particular session. Being relatively passive systems, logging mechanisms are typically enabled (or disabled) via the use of environment variables or command-line options to applications.

Some logging systems allow for "levels" of logging to be enabled, so for example one can see only critical messages at a certain level, or could see all available log output at a different level. This log level control allows a user to control the amount of overhead that the logging mechanism imposes at runtime.

3.5.2 Checkpoint. Long-running HPC applications that operate over large datasets do not typically have enough time to continuously write their intermediate results out to the stable long-term storage. This is due to the relatively slower speed of I/O that addresses the long-term storage, in contrast to the high speed system bus and volatile memory. In order to provide some safeguards against losing all progress in the event of an application crash, or to be able to rewind a simulation and advance down a different search path, applications can choose to periodically write out their data at some user-defined intervals. These snapshots of data are called checkpoints.

One technique used for the export and storage of performance observations is to embed the performance measurements alongside the application data, and amortize the cost of measurement and I/O into the cost of creating and storing the checkpoints that the application is already producing. The Cheeta [45] codesign framework provides an excellent example of this, where an overall job management tool (Savanna) and a low-level performance monitoring tool (TAU) would emit their metadata and measurements into the streaming I/O layer (ADIOS) used by the application, embedding, contextualizing, and preserving performance observations.

This technique achieves two things. Firstly, the amount of overhead and additional I/O imposed by the performance metrics often dissapears into the large volume of work done to create and store an application checkpoint. Secondly, it provides a natural correlation between the productive work that an application performed, and the measurements of the performance metrics as it did that work. When "replaying" the checkpoint data, a developer has at the same moment a picture of the work performed, as well as the measurements of the execution environment and how efficiently the code was able to produce that work. This embedding of performance measures and metadata into an application's output, or into its checkpoint snapshots, gives a fair amount of additional provenance, including the scale of the job and the machine it was run on, so insights could be gained in future reviews by comparing similar jobs on the same or similar resources, to observe the impact of code changes on application efficiency over time.

3.5.3 Cacheing. In some systems, information is generated continually during an application's execution, and it is not overwritten in place but retained, and yet it is also not immediately exported fully into its final storage location. In such cases, information needs to be exported into a cache of some kind.

There are diverse reasons for caching information at various stages of a monitoring system. One simple and intuitive reason would be to avoid interrupting communication that is being done by the applications that are being monitored. Communications are typically orders of magnitude slower than computations. Overhead can be lowered and performance improved in many cases by retaining high-frequency events locally, and perhaps doing some compression, filtering, or other operations on the data, prior to its re-export and further transmittal.

Cacheing systems can range in sophistication from something as simple as a first-in-first-out (FIFO) queue, to complex event-processing layers with scriptable behaviors allowing custom logic that can react to the contents of things being stored in the cache. While it is not required, ideally all caching systems will have some mechanism in place to alert the user and perform appropriate fail-safe actions in the event that the cache grows beyond some reasonable size.

A cache can be implemented as a variety of different data structures, sometimes embedded within additional data structures. A ring buffer or unbouned queue is just as valid a means for retaining cached observations as a hash table. The data structure that is utilized should be selected based on the desired use-case of the system. A good example of this can be found in the Caliper [12] performance introspection tool. Caliper uses different data storage models depending on which services a user has activated at runtime. This allows it to record information in a manner optimized for low overhead, factoring in both the type of contextualization that is needed for observations, and the granularity of observations being requested.

Nearly all online monitoring systems employ some form of cacheing or another, especially if they offer support for network communication of observed data. A monitoring system typically pools or stages information before writing it out to disk into a

log file, or sending it out over the network in a publish/subscribe system. In such cases we'd describe it as using a cache, but generally the system would be a logging system or a publish/subscribe system, as that reflects the behavior of the system as a whole.

3.5.4 Polling and Pulling. When monitoring information is retained, but frequency or volume of information, or the operating environment's sensitivity to overhead is high, it may make sense to use a *pull*-based model for monitoring. In this context, pulling refers to the request and receipt of information being exported from one context into another. This usually takes places via interprocess communication methods, and can be within a single computing node, or coordinated remotely across the interconnect between nodes.

Oftentimes as well, due to the simplicity of its design, a *polling* mechanism is built-in that allows for remote processes to determine whether it is time to pull information, or perhaps what information they would like to pull. In this model, the monitoring system will provide a mechanism for remote components to interrogate the sources of information to determine whether new information exists, or the transmission of that information is warranted. Polling ranges in sophistication from full complex query languages where results are computed and returned, to simple call-and-response notifications where the polling message essentially says, "I'm ready, send what you have."

What is distinctive about polling and pulling models is that communication of the exported information is directed by the receiving end, and the sending side operates passively, cacheing its information and servicing the remote requests for it.

3.5.5 Broadcast or Push. Both the broadcast and push models are similar in that the sender of information is in charge of the content and frequency of what is exported. A system can be said to be broadcasting if it is indiscriminately dispatching information out to all other components of the system that are capable of receiving it, regardless of the content of the message or the capacity of the receiver to use it productively. This can be useful in cases where network interconnects or on-node IPC can very efficiently duplicate information out and provide it to multiple recipients within the same timeframe or with the same resource consumption as it would take to deliver it to a single recipient.

3.5.6 Hybrid Push/Pull. Push/pull systems [5] allow for disparate components to discover and engage with each other, but do not impose a particular coordination scheme or global state to be maintained. Each side of communication waits for incoming requests (or results from their previous outbound requests). Both sides are also freely permitted to fire off messages or push information out into the system at their own prerogative. This is useful for observing parallel applications that have ranks or components that operate independently of each other or that may finish out of synchronization with each other.

3.5.7 Publish/Subscribe. Monitoring systems that provide a publish/subscribe model are able to offer the finest-grain control over the movement of information of all the models discussed so far. These systems provide brokering services which connect receivers and senders together, and facilitate the orderly exporting of information through the system. In addition to the movement of monitoring information, these systems also must coordinate the state of the publishers and subscriber agents themselves, in order to provide notice about the availability and information sources, and the presence of information sinks to transmit to. These systems can make powerful contributions to the orderly operation on an online monitoring system at scale. Because of their capabilities, they are can also be difficult to implement, and can require greater configuration to effectively deploy.

3.6 Introspection, Opacity, and Interface Standardization

A particularly lamentable fact about extant online monitoring systems is that they generally have bespoke or opaque interfaces, protocols, and data formats. Since the beginning of the discipline of computer science, one of the running jokes amongst practicioners has been the sarcastic pronouncement, *"The great thing about standards is that there are so many of them to choose from!"* This applies to numerous subdisciplines in computing, but online monitoring no less. When one does not know that an information source exists, or when one does not know how to properly interact with it, it may as well not exist except for the overhead that is incurrent in its processing.

Many monitoring sources in HPC are produced for specific research experiments as one-off accessories, or for the utility of a single integrated workflow, or the operation of a specific physical compute resource. It can be difficult if not impossible to exploit the capabilities of these masked-off sources to make contributions to any more generalized online monitoring frameworks. Expert knowledge and specially-targeted and tailored code would need to be written in each instance of a deployed monitoring system in order to discover and then tap into these otherwise-observable subjects for online monitoring. That kind of knowledge and that amount of labor, for both initial implementation and for project maintenance, is obviously prohibitive in comparison to the commonly marginal value that can be discovered and extracted through online monitoring, as *the opportunities for large gains are assumed to be discovered and integrated into the layer of the project-specific internal introspection* that is already in place, having the property of being opaque that is being discussed here.

It may not be the case though that the introspection capabilities internal to a particular project or vendor-specific OS and hardware management were future-proofed or able to fully capture and exploits the opportunities for optimization that are available, where being integrated into a broader or more holistic monitoring framework potentially could.

Oftentimes capabilities have been available but the desire to utilize these abilities is newly emergent, and can be hamstrung by the lack of observability or accessible control points. One solution to the challenge of introspection is the use of generic performance annotation hooks, source-level instrumentation that is disabled by default and imposes no overhead, but can be activated to yield a rich set of information at runtime, with detailed contextualization. The PerfStubs [13] project proposes an API and toolkit for this. PerfStubs is not tied to any specific tools, but provides hooks for performance monitoring tools to tap into and observe programs in execution. This means some tools can engage with it in sparse flyweight ways

that avoid the overhead of something like full callpath tracing and context tracking, and take actions more suitable for always-on runtime monitoring or ocassional auto-tuning.

Understanding application-level context, such has having explicitly identified iteration boundaries, or having clearly defined divisions between communication and computation phases, allows a generic annotation framework to go beyond simple introspection tasks. Having performance-related annotations already baked into application codes can render them into both sources of information and targets of tuning within an online monitoring framework.

The more sophisticated the purpose-built or project-internal introspection system is, the more capabilities it is able to provide for configuration and efficient operation, there is an increasing likelihood that it is difficult to observe or interface with. Very simple models like logging, for example, where observable surface-area of noteworthy events are exported into plain-text log files with lightweight structure such as having tab or comma-delimited data fields, are relatively trivial to observe and integrate into a broader monitoring platform. More complex end-to-end workflow management systems with support for internal logic and dynamic behaviors and a robust online information flow implementing publish/subscribe capabilities, can from the outside be entirely opaque and mask off the events and status of internal components within the events and status of the management systems' data model and protocols.

It is unlikely that pure generality of observables will be achieved, given the multitudes of design influences and priorities that factor into fielding and operating even the simplest of HPC platforms in the modern era. Pure generality would mean that such that all observables can be heirarchically integrated from multiple perspectives, and with unrestricted visibility of any subsets of phenomena, and phenomena have stable identities and productively-combinable semantics. This would, of course, require incredible discipline and thoughtfulness in the design of the first-order lowest-level components of the system. Interfaces between ascending layers of integration would be required to conform to protocols that conserved the observability of any desired element operated over or participating in the computational task acting above it. There are many justifide reasons why data and activity live mostly in unobservable enclaves owned and arbitrarily operated over by so many processes. Not least among these reasons is efficient use of storage resources, and a desire to maximize the proportion of computation that produces output significant to users, compared to the amount of work done to facilitate that productive work.

What we may clearly percieve here is how complicated the trade/offs become, at multiple levels of understanding. There are broad and almost existential trade/offs of purpose between the partially-overlapping motives of stakeholders, developers, users, and optimizers of HPC systems. Then there are comparitavely microscopic trade/offs with enormous downstream implications that are made at the software and hardware design and integration levels of HPC. We can see then that embedding capababilities and increasing the sophistication within one design layer, or within a single component of a larger system, can have the consequence of *decreasing* the visibility of that layer or component to outside observers. At the same time, the activity observed by that introspection system will have high enough overhead to export elsewhere that it becomes increasingly likely it will not be considered worth the percieved benefits of doing so.

3.7 Case Study: The CDC 6600 Mainframe

HPC, or "supercomputing", has moved through several different eras from the single-core mainframes of the 1950s, to vector machines, into the era of distributed memory, heterogeneity, and extreme scales of devices. As time and technology progressed, the innovations of prior eras were integrated into the newer designs, often combined into unified components that were then multiplied in number and interconnected to provide expanded compute capabilities. These growing numbers and increasing reliance on complex communication patterns led to cyclical renewal to the challenges of understanding and fully utilizing the available resources of those machines. In addition to increasing performance of codes, having insight into the state and behavior of these complex machines could also increase the performance of developers and users. The easier some HPC resource is to understand, develop for, and debug, the more productive work can be achieved with it.

One of the earliest commercially available mainframe machines was known as the **CDC 6600** produced by the by the Control Data Corporation [83]. Principally designed by Seymour Cray [52], one of the legendary early innovators of HPC technology, the 6600 introduced a number of ideas which became fundamental to nearly every HPC system which followed. We mention it here not only because it was a conspicuously popular and important machine in the history of HPC development, but because it also shows two major challenges which are still with us today: Parallel complexity, and the trade off between opacity and operational efficiency.

The 6600 was designed with multiple functional units which were able to operate in parallel with each other, at the same time, reducing the gaps in productive work that are the natural result of operations stalling as data or new instructions get fetched from memory. In addition to a central processor (CP) which executed the majority of user code, there was an instruction cache put in place to facilitate pipelining, and a cohort of 10 different "peripheral processors" (PP). This queueing of instructions, and the processing of the different steps of an instruction (i.e. loading, evaluating, branching, storing, etc.) simultaneously, where new instructions can be introduced at the front end of the process as older instructions are partway through being evaluated and eventually retired, is typically referred to as *pipelineing*. Even naive implementations are capable of providing significant speedups, and these will typically be bounded by the depth of the execution pipeline and the frequency with which instructions cause unavoidable delays in the handling of a stage of execution, preventing that stage from vacating and the preceding stages from moving forward, delays which are known as *stalls*. The 6600's inclusion of an instruction cache was an early example of this pipelining idea, which has grown into much richer and incredibly sophisticated forms in modern HPC.

The overall performance of codes running on this hardware was in large part a factor of how efficiently a developer could take advantage of the parallelism the multiple PPs offered. At the time of the 6600's development, operating systems and compilers were also (relatively) new concepts, and were not able to provide many of the modern advancements of automated optimization or parallization

of code regions. The complexity of how the CP and the parallel PP components would cooperate to create a kind of pipelining, also created a novel burden for code developers targeting that platform, to design their implementations of algorithms around the optimal behaviors suitable for the 6600's specific internal coordination patterns. If a programmer did not take advantage of the parallelism on offer, the machine was hardly able to do it for them, but a higher degree of expertise and design overhead was thus introduced and imposed on HPC developers.

While the 6600 was not a true "multiprocessor" system in the modern sense of the term, it did support some forms of pure parallelism, where real work was being done concurrently, and not merely seeming concurrent through time/sharing techniques like context switching. The 10 PPs each operated independently of each other and the CP. Their primary task was to load and store information from the main memory of the 6600, freeing the CP to use its time more productively to perform complex multi-step operations on that data once it had been fetched from memory. The 0th PP was dedicated to running the operating system of the entire mainframe, including the CP. The 9th PP was dedicated to running the user terminal, managing the display and interactivity for users running programs and evaluating the results. Activity for memory accesses, the PPs, and the CPs, was coordinated by setting values for different states into registers. Importantly, these private registers were not addressable by user code running on the machine.

Here we see one of the first instances of an intentional trade/off which remains an interesting challenge all through HPC into the modern era: *Exchanging opacity for efficiency*. Private registers effectively created a communication channel for the operating system, nascent though it was at the time, to orchestrate the behavior of the machine in support of user software demands, without needing to impose the overhead of synchronization with the specific activity of user programs. This did, however, mean that the state of the machine itself could not be easily introspected on by any software that was running on it, leading to novel challenges when searching for optimal use patterns, or debugging a code that was behaving unexpectedly. Another way in which this opacity exchange could be seen was in the physical presence of the machine itself. Unlike many of the machines which preceded it, the 6600 did not have any integrated display panel of lights to represent the values of the different registers. Normally these had been used to inspect the machine state or to perform debugging, even if only for the initial startup of the machine, after which printouts or a cathode-ray tube (CRT) display could be used to check state.

It would not have been feasible for the 6600 to offer a lightbulb-array-based "live look" into the state of the machine registers, as the physical layout of the device was too dense, and the number of registers that would need to be displayed was too great to be practical. So even this early on in the field of HPC, it was understood that monitoring systems are not free, that there is a trade/off. By choosing to forgo a physical monitoring system for the 6600, Seymour Cray was able to lower the power requirements, reduce the operating temperature, and bring the components of the computer closer together. The physical locality of resources in HPC systems is significant, because at the cutting edge of design, having reduced wire length translates directly into performance gains. This informs the physical layout of the 6600, taking the form of a star-shape with the CP in the center of the device, to be as close as possible on average to each of the PPs and memory banks.

The 6600 is one machine, but a representative example, showing the origins to a couple of the enduring challenges for development and monitoring in HPC.

3.8 Observability: In Conclusion

Now having familiarized ourselves with both practical and theoretical aspects of making online observations in an HPC environment, we are equipped to proceed into the discussion of tools and techniques with a richer understanding of the means and meaning underlying what is being discussed.

4 MONITORING FOR HPC: DEDICATED FRAMEWORKS

As should be apparent from the discussion so far, the topic of online monitoring can take on many dimensions in the HPC context. Data sources may be as diverse as the version numbers of software being run, XML files emitted by proprietary commercial sensors and software that report the power of a building's HVAC systems every few minutes, or hundreds of temperature sensors scattered around the server room with data being aggregated every few seconds, to in situ (online) probes of scientific workflow components executing on massive clusters, high-resolution performance data being captured and aggregated by the millisecond.

The manner and means by which data sources, applications, tools, and system services conspire to produce the general outcome of online monitoring are as diverse as these examples. There are as many purposes for online monitoring as there are individual contributors or consumers to the monitoring infrastructure or the system that it is monitoring.

It is therefore worth making note of the basic fact: *Online monitoring for HPC is rarely the exclusive role of a single tool dedicated to monitoring a single aspect of the HPC system.* In Monitoring for HPC: General Topics (§ 5) we will discuss some of the common challenges of HPC that are closely related to online monitoring, analysis, and feedback, but are not necessarily centered on a particular monitoring concept or tool.

For now, in this section, we'll survey some of the past and present heavy-hitters amongst *purpose-built online monitoring systems* [33].

4.1 SuperMon

SuperMon [77] is a set of tools for cluster monitoring, engineered to be high-speed and to minimize overhead. Delivered during the terascale era of HPC in the early 2000's, one of SuperMon's design goals and achievements was to allow for low-impact monitoring high-frequency events, making previously invisible behaviors of the cluster open to observation.

The system operated online, and could gather data from all nodes and assemble it into a coherent single perspective of the cluster as a whole. SuperMon's developers described the state of the art as being extraordinarily primitive, being little more than shell scripts that would periodically run the *ping* command to test the responsiveness of nodes. If the ping attempt failed, or if a support notice arrived from a user saying their job failed or they could not log into a node,

machine administrators would then direct their attention to manually determining the cause of the failure. The monitoring sensors available to administrators were, in the scenario they describe, limited to the server daemon that handled user logins, and the daemon that would respond to ping commands. There were other solutions which might have worked on their newly constructed Linux-based terascale cluster, but that did not meet their design requirements for minimizing overhead and application performance perturbation. One such tool they mention is *rstatd* for "remote status" based on the SunRPC protocol. Again, it was deemed too slow, and also at 20 years old did not offer sufficient flexibility to describe the dynamism of events and hardware that were beginning to show up in HPC clusters. They considered this inadequate for terascale computing, and set about to construct their own solution.

The Supermon cluster monitoring system was built from three distinct parts:

- Linux kernel module to observe and emit performance data.
- *mon*: In situ data server to capture and cache data from the kernel module, and to service requests for that data.
- *Supermon*: To compose samples from any number of nodes into a single set of samples that represents the state of the cluster.

Supermon, like may monitoring systems, also utilized its own client-server protocol to exchange information between the three components. Its developers used a clever encoding of performance data into self-describing *s-expressions*, something like modern-day XML, but designed originally as a part of the LISP programming language in the 1950s. These recursively-defined self-describing packets in the Supermon protocol were advances in utility and flexibility over the existing RPC packets, which were strictly defined and packed into binary formats. S-expressions could vary in size and content, and could be easily processed and composed into various representations or aggregations as desired. They found that processing packets of this nature, even in plain text, was faster than what was needed to serialize and deserialize everything into rigidly defined structures, and had the added benefit of not requiring the use of special RPC compilers or inspection tools.

This system represented a major step forward for online monitoring of HPC clusters, being faster, more efficient, more flexible, and easier to use than the immediately outdated RPC-based monitoring state of the art.

4.2 MonALISA

With grid computing, often teams would be using computing systems that were connected over the internet, distributed across a nation and even around the world. The number of constituent systems in a computing grid, and the extreme heterogeneity of them, posed a challenge to administrators and users who wished to be able to observe the system in aggregate. Around 1998 the MonALISA project was born, looking to provide practical solutions to this problem.

Monitoring Agents in A Large Integrated Services Architecture (MonALISA) utilized forward-deployed "station servers" positioned at each of the major grid system locales. These station servers would run a variety of agent-based services, forming a dynamic distributed services architecturem, capable of deploying, starting, stopping, discovering, and utilizing arbitrary monitoring agents online. This system was not overly concerned with maximizing throughput of monitoring data, focusing rather on flexibility and self-organizing capabilities. It was considered acceptable and also useful for an agent to capture individual or summary measures of its grid location once a minute, and to aggregate on the order of hundreds of station server's data from around the world every several minutes. By modern standards this is not impressive, but at the time this was very useful, especially given the deep configurability and flexibility of the agent-based system. MonALISA was also not concerned about overhead and performance perturbation, since agents were running on their own server attached to the grid facility's network. In fact, much of the infrastructure of MonALISA was developed in JAVA, rather than the traditional high-performance languages like FORTRAN, C, and C++.

Much work was done in this project to facilitate the distributed nature of grid computing, or to model this monitoring solution around the features of distributed computing. Agents would place themselves in a common registry, report changes in their availability, and report what information they were able to provide. Monitoring clients could then subscribe to the information streams from those agents, and this subscription would propagate through the system, and that client would begin to receive streams of information from all active and available agents of that type. The MonALISA framework was built to be resilient to the vissictitudes of internet connectivity, and so all operations were asynchronous and all interacting components were loosely-coupled. Individual sites, or entire enclaves of sites, were capable of performing just as effectively in isolation as they would when completely joined and online together.

This project also included a client which could project the monitoring data over a global map, allowing for useful dashboard-style visualizations of a variety of topics, for example: system availability, load balance, and data link saturations. More than just collecting and presenting information, MonALISA could also be used to optimize grid-based workflows, based on the types of agents deployed and the sensitivity of an application to receiving directives and adjusting plans mid-run. MonALISA interfaced with a variety of other on-site monitoring tools we will discuss here, such as MRTG and Ganglia. Those interfaces is where MonALISA gathered much of the actual site data that was ingested and shared by agents. While we are in this survey mostly interested in the in situ (online) monitoring that is closer to the nodes, applications, and facility sensors, the ability to step out and up another layer and provide monitoring across vast distances, uniting multiple clusters into an aggregated perspective, is a noteworthy achievement by the MonALISA team. This project can serve as an example for how to think about and even implement some of the technologies that are required to perform those tasks.

4.3 MRTG

The Multi Router Traffic Grapher (MRTG) [57] [56] first emerged as a single-purpose tool, designed to monitor the inbound and outbound traffic on a internet gateway router. This perl script would read the octet counters of the router every 5 minutes, and then generate a graph which could be seen by visiting a web page hosted

on the same server where the script was running. After it became open source, people from all over the world began to use it and make code contributions, even porting parts of it to C for performance increases. MRTG quickly grew in sophistication, configurability, and monitoring capability.

The main bottleneck slowing MRTG's early adoption was the need for an external client library to interface with routers over the Simple Network Management Protocol (SNMP), since not all potential users had access to or the ability to build such libraries. Eventually, a perl-based SNMP implementation was integrated into MRTG and the project was then entirely self-contained and trivially easy to configure and use. Being implemented in perl, it was also automatically portable to every platform where perl code ran, which was just about everywhere. Since it was simple, self-contained, useful for a variety of tasks, free, and open-source, by the mid 1990s MRTG had become a very popular monitoring tool within the IT world. Usability cannot be underestimated, when considering the value and effectiveness of monitoring solutions.

Another way in which MRTG facilitated usability was by embracing a functional opinion about monitoring data: that it is less important the less recent it is. This allowed for a "*lossy data storage*" paradigm in MRTG's implementation, which would allow MRTG to compress expiring data into rolling averages of the prior measurement periods, preventing server storage from filling up with old monitoring data if MRTG was left running for an arbitrary amount of time. While also offering a boon to administrators who did not need to manually flush logs or purge databases, it also dovetailed with the automatic activity graphs that MRTG produced. By default it would offer a 5-minute resolution of the last 24 hours, every 30 minutes for the last week, and average values for every 2 hours for the last month. Two years worth of history are archived, but compressed further to where entries represent the average value over two day periods. Having the monitoring data constantly flatting and coarsening like this kept the service running smoothly, and provided handy reports to summarize both immediate events in detail, and longer-term trends from an overview perspective. Though only a simple approach, it was very practical, and had the effect of making this monitoring solution useful for both system administrators, and site resource managers who needed to keep an eye on system utilization in order to make purchasing decisions about new systems or increases in networking capacity.

4.4 RRDTool

The same creator of MRTG also produced a toolkit for rapidly developing one's own monitoring solutions, the Round Robin Database Tool (RRDtool), which was released back in 1999. The core functionality it provides is a time-series data model and a suite of utilities for accessing the monitoring data repository. RRDTool is the central data storage solution running beneath a number of popular monitoring solutions, such as Ganglia, Cacti, Collectd, etc. One modern incarnation of this tool is the SE-RRDTool [97], which extends the core features of RRD with the ability to provide semantic enhancements, that is to say semantic annotations, to data sources. These annotations improve the ability of tools to utilize information gathered within the system, especially for automated learning systems that do not accomodate "human in the loop" expert review of monitoring data. SE-RRDTool allows for the expression of data ontologies for values that are captured in a monitoring service that utilizes it, including units, quality of service metrics, system heirarchy such as cloud entities, and other custom user-defined typings. In addition to marking up the data, it also enhances queries, allowing for semantic-based retrieval of values with a cursory support for automatically generating derived or projected values based on the semantic rules built into user-defiend ontologies.

4.5 Ganglia

Ganglia [66] [42] is a popular distributed monitoring solution that targets both clusters and Grid computing environments. The Grid computing is especially supported by the heirarchical design of the Ganglia data model and services. Its implementation uses XML for encoding its data, and the previously discussed RRDTool for data storage and analysis / visualization. By 2004, Ganglia was in use at over 500 compute clusters worldwide.

Ganglia is built around a monitoring daemon that uses TCP/IP multicast listen/announce protocols to monitor activity within a cluster, gathering a set of built-in metrics as well as allowing plugins to capture arbitrary user-defined metrics. It leans into the idea of federations of clusters very heavily, supporting this through the ability to pull in collections of child data sources from various clusters periodically, and aggregate this information into a unified data store.

Generally Ganglia has operating overhead below 0.1%, since it is focused on coarse-grained sampling of metrics like hardware counters, temperatures, general system activity, network traffic, etc., and does not need to engage with or interrupt application processes, and its use of RRDTool for data management means it does not need to retain large data sets indefinetly. Though it has local services that run in situ, and it aggregates its information online, its focus on collecting samples of metrics at a coarser-grain than the individual processes or components of a workflow lends it more value to system administration types than to developers or even users of HPC systems. As discussed above, Ganglia isn't intended to be used in all scenarios, and can be complemented or even potentially replaced for certain production environments or user sets by other online metric collection services such as LDMS.

4.6 Nagios

Nagios [47] [37] is another online monitoring tool with a strong emphasis on monitoring of network devices and their service statuses, to provide automatic notice to administrators when there are service failures or capacity is being approached. Like Ganglia and many other services described here, it has an in situ server that runs in the background local to the nodes of a cluster. This service periodically probes the state of the machine and services, and can fire off triggered behaviors depending on what is observed. Nagios offers a very flexible plugin system, and over the years has gained hundreds of plugins and been used as the core component of many different commercial monitoring solutions, where the commercial product contributes their added value features in the form of proprietary plugins which run on the basic Nagios software stack.

There are some limitations to Nagios, including being user-unfriendly to configure (perhaps why it is often wrapped up into

a commercial product), and also not having its own data storage solution built in. However, despite these limitations, it's being lightweight and, when configured, a stable and reliable system monitoring tool, and one that can be infinitely extended through plugins, Nagios endures as a commonly available monitoring tool for making observations to support the management of HPC clusters.

4.7 TACC stats

In 2013, Texas Advanced Computing Center (TACC) fielded a set of sweeping updates and enhancements to the monitoring solution for their Linux-based HPC clusters, though keeping the rather straightforward name for their project: *TACC stats* [22]. The central premise to TACC stats is that users and developers and adminstrators do not need to do anything in order for it to be enabled and functioning. TACC stats is constantly enabled and accumulates performance and utilization metrics for every single job that runs on the cluster. It utilizes a variety of sources for information, from the filesystem to the messaging services to the job scheduler, to operating system performance introspection APIs. All metrics gathered into TACC stats are resolved to the job and hardware device, so individual jobs and applications can be analysed separately. Many kinds of metrics are gathered by this system, from core-level CPU usage, socket-level memory usage, swapping and paging statistics, system and block device counters, interprocess communication, interconnect fabric traffic, memory controller cache, NUMA coherence agents, and the power control units on servers. TACC stats is built to be modular, and can be extended to track arbitrary additional data points based on user interest and data availability.

TACC stats is a fine example of what can be achieved with an always-on monitoring solution. The overhead of collecting the monitioring data is simply amortized into the operational overhead of the cluster itself. Because it has records of every single job going back to 2013, long-term trends can be observed in use patterns, so stakeholders can get clear and detailed reports about how their machines are being used, and what users needs may be for the design and purchase of future resources, or the targeting of talent and funding to support the improvement of software packages which are seeing the most use. System administrators are also able to take a more proactive approach to the detection and diagnoses of hardware failures or configuration issues, since the system is continuously collecting and integrating the monitoring data, and constantly reviewing that data for anomalies or events which were able to be correlated with problems that had previously been discovered and resolved.

4.8 ProMon

Observing that the vast majority of performance tooling in HPC systems is targeted at heavyweight program introspection during development, the ProMon [71] system was developed and fielded in 2015 to offer another approach to online introspection in HPC. The defining design principle for ProMon captured by it's full name: *Production Monitoring*. Like TACC stats, ProMon is aligned with the vision of always-on monitoring, so that developers, administrators, and users do not need to take any additional actions in order to have access to runtime introspection data, and the potential benefits that it might enable.

ProMon's developers are motivated like many in the online monitoring community by the need for introspection into the runtime environment and into long-running jobs on HPC systems. Remarkable increases in system scale and heterogeniety, the integration of massive and complex software projects into campaigns of scientific workflows operating over in situ data stores, and the complex entailments of individual component failure or soft error accumulation over a long run, all add increasing motivation to the case for online in situ monitoring for HPC. The challenge then is to provide flexible low-overhead facilities to meet this monitoring need, without negatively impacting system stability or software usability. Only then will users and stakeholders of large and expensive HPC systems be willing to broadly introduce online monitoring to their production environment, and not only their development environment.

Since ProMon is a generic and programmable platform, it can be configured in ways which will cause large amounts of performance perturbation to applications. However, in realistic scenarios, its developers have claimed less than 1% overhead by the deployment and use of ProMon in a production environment. On the development side, the ProMon concept outlines how value can be gained by doing more heavyweight profiling of applications, which can be stored in performance databases and later integrated to enrich the more flyweight measurements taken on the production side at runtime. Given the focus on online monitoring in HPC, we will focus on the production aspects of the ProMon design. On the production side, ProMon consists of several components:

- Analyzer
- Injector
- Reporter
- Parser
- FlowGrapher

Other components can be added in the future, but these are the essential core of ProMon. The Injector inserts monitoring probes into applications using Dyninst [90] to perform binary instrumentation, using either static or dynamic instrouemtnation. These probes collect and organize some local data and then send them over to the Analyzer using TCP or UDP protocols.

The FlowGrapher is where users of ProMon can identify parts of their applications that they are interested in monitoring, to drive the selection of targets for the insertion of probes. Work on this component is ongoing, but it is able to provide textual output identifying loops within codes which the user can then select from by a numerical identifier. The Analyzer is a robust service capable of receiving information from a variety of processes from different applications simultaneously. Implemented as a daemon server, it also integrates the streaming probe data into a data store with provenance that can be used to disambiguate similar types of data, or data from different sources that was generated in parallel. The Analyzer operates on single or dual-event types, where single events represent milestones such as the end of a simulation step, and dual events reperesent beginning and end times, or other forms of encoding events in terms of their duration of overlap with other events.

Like the SuperMon system, ProMon utilizes its own plain-text data format to exchange information in a simple to use self-describing format. ProMon's format is named the Production Monitoring Language (PML) and is compliant with the XML standard to make it

very easy to parse, and open the use of countless extant libraries and commercial data processing tools. It comes bundled with a variety of tags for annotating performance events in rich ways, and these tags can be combined, embedded, or added to in order to extend the capabilities of ProMon to suit an arbitrary array of use cases.

ProMon is an actively developed project and in its design and implementation seems to be taking a very sensible and effective angle of attack on the more difficult aspects of in situ (online) monitoring at scale and in production HPC environments.

4.9 SOS and SOSflow

This author's own research work falls squarely within the domain of online monitoring for HPC, the initial contribution being the Scalable Observation System (SOS) model for online characterization and analysis of HPC applications, and its reference implementation in the SOSflow [92] project.

Three principles were core to the design and implementation of SOS when it was introduced in 2016: First, that an effective monitoring system needs to be deployed in situ and running online at the same time as and colocated with the subjects that it is monitoring. Secondly, the system needed to provide the ability for interactive exploration of monitoring data online, in order to support real-time analysis of metrics, as well as feedback and code-steering. Finally, the system needed to have a small footprint in terms of memory and CPU requirements, such that it did not perturb the environment that it was monitoring.

This also meant that interactions with the SOS system would need to be loosely-coupled and asynchronous, so that no steps in observing or communicating information into and out from or through the SOS system would require an application or operating environment to block and cease doing productive work. By co-locating the observation system's online processing and analysis of measurements with the workflow components, SOS could improve the fidelity of system performance data without requireing the costly delays of synchronization or congestion of shared network and filesystem resources. While SOS had various other motivating concepts and grew to enable a wider variety of purposes than simple observation and online analysis, those are its core tenets.

SOS comprises several components:

- Information Producers: APIs for bringing information into SOS.
- Information Management: Online and optionally persistant databases and caches.
- Introspection Support: Services to provide online access to the SOS databases and high-speed caches.
- In Situ Analytics: Components to perform online analysis, including APIs to additional languages condusive to analytics, such as Python.
- Feedback System: APIs for sending information to non-SOS entities, as well as providing feedback to sources of data such that control loops can be established for purposes such as code steering.

These components work together to provide SOS's core features:

- **Online**: Observations are gathered and available at runtime to capture and exploit features that may only emerge in that complex interactive moment, and may not be discoverable during development or with offline single-component analysis.
- **Scalable**: SOS is a distributed runtime platform, and as the scale of the deployment increases, so too does the amount of available resources for the operating of SOS adjacent a running HPC application. Because SOS uses loosely-coupled asynchronous protocols for all of its interactions with applications and within itself, communication bottlenecks can be avoided by adjusting settings to perform analysis in situ rather than migrating information online to centralized repositories which might become bottlenecks at extreme scales.
- **Global Information Space**: Information gathered from numerous sources, system layers, or actors within an execution environment, all are captured and stored within a common context, both on-node and across the entire allocation of nodes. This information is characterized by:
  - **Multiple Perspectives** - Queries over the observed data in SOS can isolate or aggregate the data in entirely arbitrary ways, so the system can service both fine-grained analysis as well as high-level dashboard views of the system state or an application's progress. Workflows or even campaigns can be observed in their entirety, and then individual components of those workflows can be selected and introspected on in greater detail.
  - **Time Alignment** - All values captured in SOS are time-stamped so that events which occured in chronological sequence but in different parts of the system can later be aligned and correlated.
  - **Reusable Collection** - Information gathered into SOS can be used for multiple purposes and be correlated in various ways without having to be gathered or transmitted multiple times.
  - **Unilateral Interactivity** - Sources and sinks of information need not coordinate with other workflow or SOS components about what to publishj, they can submit information and rely on the SOS runtime to decide how best to utilize it. The SOS framework will automatically migrate information where it is needed, or resolve online queries in a parallel distributed manner when that is superior to migrating all data online for central analysis. SOS is also capable of managing the retention of unused information, and allows users to control this selectively at runtime, as well.

SOSflow was implemented as a multithreaded Linux daemon and client library, both coded in the C language and designed to be nearly entirely self-contained so to be easy to integrate into existing applications, workflows, performance tools, or broader monitoring infrastructures. It is also coded at that lower level and without other runtime service dependencies to maximize its performance while minimizing its runtime footprint in the in situ environments it is distributed across. SOS runs in user-space, and is invoked at the beginning of a parallel job script, and brought down at the end of a user's job, with the option of exporting the database of observations to persistant storage for offline analysis.

Early versions of SOS were tested out to hundreds of nodes and the overhead of the system even in early development phases was typically below 2%, with the highest overhead as a percentage increase in walltime for jobs codeployed with SOS being extremely short-lived processes, where the presence of SOS increased the runtime by only 3%, likely to do with the distributed launching of the SOS runtime daemon within the user's allocation. The asynchronous and online nature of SOS, and the efficiency of its internal communication protocols, is one of its most robust aspects. While the distributed persistent data stores would sometimes increase in queue depth for transactional commits of batches of data during times of heavy traffic, queues would eventually drain out and the time between a value being published into SOS and it being available for querying or other uses would eventually fall back down to its initial baseline. Regardless of the system load from codeployed simulation software or the volume of traffic being processed into the persistent data stores on the backplane of the SOS runtime, the velocity of data capture, the time cost of API calls made to SOS in the client library, and the RTT for probe messages between the client and the daemon, all remained constant and exteremely high in all experiments.

SOSflow is being actively developed and has found a variety of uses in different experiments and projects in the years since its initial release. One such experiment, an integration with the ALPINE (Ascent) project for online projection of performance metrics into the domain of simulation geometry, will be discussed in a later section, as it is a neat example of the power of flexible online monitoring tools designed for the modern in situ HPC paradigm.

4.10 FogMon

While we've primarily focused on leadership-class massively parallel Linux clusters in our discussions of HPC, there is room here to talk about some rather cutting-edge and monitoring technologies that are looking ahead to possible futures and rather exotic dynamic computing topologies, though these works to have immediate significance to classical HPC concerns.

One such monitoring tool that has recently been developed is FogMon [24], a lightweight self-organizing distributued monitoring framework for Fog infrastructures. Cloud computing has introduced *utility computing* as a cost-effective way to ship software services to their final users by substantially reducing the operational effort required by service providers. Over the same epoch, the Internet of Things (IoT) has been constantly growing, from the rich compute and sensor capabilities of cellular devices, to the embedding of wifi and low-power general computational capability in nearly any device with a power cord or a battery. The number of connected IoT devices has caused the amount of data being generated to increase explosively, though it is noteworthy that these so-called *edge devices* are often much more resource contrained than traditional HPC machines or cloud servers. Consequently, deployments of IoT applications are typically broken into two categories: IoT+Cloud where the majority of computing is offloaded to cloud services, and IoT+Edge where data is processed locally on the device, and dependence on availability of cloud resources is minimized. IoT+Cloud gains the massive compute capacity of cloud resources, but can suffer latency, network congestion, or even service unavailability, while IoT+Edge allows for immediate interactivity, but can aggressively consume limited resources such as battery life or storage space, and can impose a great deal of complexity on IoT application developers to have safe and coordinated information synchronization between local and cloud resources.

Addressing these constraints, the paradigm of *Fog computing* is beginning to gain traction, where applications are split into microservices which can be, along with the appropriate data, migrated and executed at the location or service layer where it is most appropriate to. Fog-enabled designs make it possible to reduce network traffic by processing and filtering IoT data before sending it to the cloud, and to reduce application response times by suitibly placing latency-critical services in proximity to the information consumer at the point of interactivity. In the abstract, Fog computing relies on a common orchestration layer which delivers a Monitoring, Analysis, Planning, and Execution loop that can theoretically support the dynamic, adaptive life-cycle management of multi-service data-aware Fog applications. FogMon is an actively developed research project which aims to support that orchestration layer, with a strong emphasis on the monitoring component, and a design that takes the Fog environment with resources contraints and unstable connectivity as a first principle. In the FogMon paper cited above, the authors provide a robust technical account of their research accomplishments and experimental validation.

FogMon and projects like it have an interesting relationship with the history of online monitoring in HPC. In one sense, Fog computing is only the latest evolution of classic *Grid computing*, which also involved the loose coupling of powerful HPC resources over relatively slow or unstable Internet connections, and which also benefitted from and existence of an orchestration layer. For an example see the MonALISA project mentioned above. The self-organizing agents of MonALISA are almost entirely mappable onto the concepts inherent in the resource-aware microservices envisioned by Fog computing. Obviously there are differences, especially in terms of the complexity of the modern IoT and Cloud computing infrastructure, and in the vast asymmetry in compute capability between edge devices and Cloud servers compared to the more evenly distributed compute capability at the nodes of a Grid computing platform. Still, there is a clear line from Fog computing back to Grid computing, and perhaps developers in the Fog space would be well-served by surveying the research done in that era.

Looking ahead, the Fog computing concepts are likely to begin to show up in traditional HPC environments, especially at extreme scales. Gains made in this field, especially by the development and validation of fundamental service infrastructures designed for low-impact and extreme-scales, will have direct implication for classical HPC compute topologies. Designing systems to be resilient to component failure is an important paradigm when an individual job may be distributed to so many hardware components in parallel that the likelihood of a component failing during a job approaches 100% for jobs of non-trivial duration. Further, in complex integrated in situ environments with many interacting parts and irregular spikes in demand for shared resources, there is much to be gained through thoughtfully engineering data processing systems with the discipline to not rely on direct and synchronous communication for productivity and progress. As the FogMon researches clearly are aware, the design and development of these orchestration layers

and loosely-coupled application paradigms is extremely complex and sensitive task. There will be much more to say about these topics in the coming years, and the type of online monitoring that is required by and enabled by Fog computing is likely to be worth paying attention to for developers and researches interested in online monitong for HPC.

4.11 LDMS

One of the most important online monitoring frameworks for current petascale and future exascale HPC clusters is the Lightweight Distributed Metric Service (LDMS) [7] [8] [23] [34]. This service is widely deployed and in consistent use in both development and production environments. LDMS was designed to attempt to bridge the gap between coarse-grained system event monitoring, and fine-grained (function or message-level) application profiling tools. Because of the higher cost of collecting fine-grained performance profiling data, wrapping code and extracting detailed information at a high frequency, often impinging on the performance of the code being observed, profiling and application tuning have usually been deemed episodic activities and not a part of normal or production executions. This does leave the vast amount of time that applications are running on HPC clusters largely opaque to detailed introspection, including understanding codes' impact on overall system behavior and other applications running concurrently but in different allocations. There are inherent complexities to HPC machine architectures, both in hardware and in their software. This is including the complex Cray architectures targeted by LDMS's developers, featuring deeply customized hardware and proprietary operating system extensions and closed vendor-specific drivers. For such systems, ready-made monitoring frameworks such as Ganglia (discussed below) were unable to meet even the basic coarse-grained monitoring needs which were motivating the creation of LDMS.

Sandia National Laboratory and the Open Grid Computing Group began a collaboration on a set of HPC monitoring, analysis, and feedback tools to attempt to begin to fill in this observational gap, and in 2014 began publishing on the monitoring component of that project, which is LDMS.

LDMS is a distributed data collection, transport, and storage tool that is highly configurable, consisting of samplers, aggregators, and storage components to support a variety of formats. Samplers periodically sample data according to user-defined frequencies, defining and exposing a metric set, and running independently from any other deployed samplers. Memory allocated for a metric set is overwritten by each successive sample, no history is retained within a sampler. Aggregators pull data from samplers or other aggregators, again according to a user-defined frequency. Distinct metric sets can be collected and aggregated at different frequencies, but unlike samplers the aggregators cannot be altered once set without restarting the aggregator. Because of the strict behavior constraints dealing with both memory and sampling frequency, LDMS' samplers and aggregators can be very well-optimized to collect very high volumes and velocities of information with low-latency and nearly zero impact on overall system performance. Further, due to the engineering effort put into a low-level RDMA communication backplane for LDMS, individual aggregators are able to collect from an enormous number of distributed hosts, with initial experiments demonstrating successful aggreation of more than 15,000:1 for RDMA over Cray's Gemini transport. Storage can write to a variety of formats, including MySQL, flat files, and a proprietary structured file format called the Scalable Object Store (not to be confused with the "Scalable Observation System" mentioned above).

The base LDMS component is its multi-threaded server daemon *ldmsd* which is run in either sampler or aggregator mode, and can support the storage functionality when running in aggregator mode. The ldmsd server loads the sampler and aggregators dynamically in response to commands from the owner of the ldmsd process. All activity within ldmsd, including the activity of samplers and aggregators and storage modules, is processed by a common worker thread pool. In more recent iterations, LDMS has gained support for more sophisticated in situ processing of sample data, including the ability to apply complex operators to metric sets as they flow through various stages of aggregation, and including the ability to interact with other services or storage systems at intermediate stages of aggregation within the cluster. In order to retain its high-level of efficiency, LDMS does not support many of the dynamic interactivity features of other online monitoring solutions discussed here. It also does not support embedded or complex self-describing data types, nor the capture of arbitrary string-based values, rather LDMS samplers are only able to capture and encode numerica values in floating point representation, a strict discipline which allows for some deep optimization to its performance and to data movement.

LDMS is a rather straightforward project, employing simple designs to great effect. It does only a few things, but it does them very effectively, and is able to make larger or more sophisticated contributions through optional integrations with other projects or tools, serving either as an information source or a sink for them. This efficiency and simplicity has led to it being widely deployed in production environments, which in turn has led to it seeing a lot of activity with various tools seeking to exploit the fine-grained performance data it is capturing, or participate in dispatching information into LDMS for other tools to have access to it at runtime. LDMS will be playing a central role in online monitoring for HPC for many more years to come, as it has earned long-range funding, has deep developer buy-in, and offers multiple types of users or administrators a considerable monitoring capability and value at nearly no cost.

4.12 CluMon and ClOver

Based on the CluMon cluster monitoring project's plugin architecture, the Cluster Overseer (ClOver) [49] tool is designed to allow a high-level overview of the state of a cluster. ClOver came about around 2009, and utilized the Intelligent Platform Management Interface (IPMI) protocol, which by then was becoming a somewhat standard protocol for online management of large computing systems. Extending the "at a glance" monitoring overview capability of CluMon, ClOver's principle design goal was to more completely decouple the operation of the monitoring infrastructure from some of the legacy components that CluMon had employed, such as the PCP services for monitoring, to facilitate genuine extensibility and realize the flexibility of the plugin architecture model. It was also

desired for ClOver to be able to provide its monitoring features to a variety of outlets, including streaming databases or web-based dashboards rather than a traditional desktop GUI client. The ClOver project showed improved performance, flexibility, and ability to be integrated with a wider array of components from its predecessor.

Where there are monitoring needs and available developers, there will soon be a system implemented. These two projects in pairing are a good example of what can sometimes happen in the HPC research space: When one project begins to show its seams or has an unavoidable dependency that doesn't translate well into newer execution environments, rather than updating or extending the prior work it is often more fruitful to simply recreate the project anew but with new tools, techniques, and integrations. This type of perennial re-design and re-writing of projects is far from inefficient in many cases, and can lead to better tools with less baggage, and a more positive ongoing impact. Something to bear in mind considering our favorite HPC projects and tools, as the pace of innovation in the HPC world seems unlikely to slow down in the coming decades.

4.13 Additional Monitoring Solutions of Note

For continued exploration of dedicated monitoring solutions, the following frameworks may be of interest to the reader:

- Performance Co-Pilot (PCP) [4]
- PerSyst [32]
- LIKWID [85] [64]
- MPCDF [78]
- OpenNMS [73]
- Prometheus [79] [80] + Kubernetes [44]
- Pandora FMS [59]
- Telegraf [15] [62] + InfluxDB [54]
- Zabbix [1] [75] [88]
- collectd [3]
- Periscope [9]
- Ovis [43]
- XDMoD [89]

5 MONITORING FOR HPC: GENERAL TOPICS

As discussed in prior sections, online monitoring for HPC is rarely a simple as deploying a single service and satisfying a single user or stakeholder's interests. *Online monitoring for HPC represents a complex constellation of interests, tools, techniques, challenges, and possibilities.* Oftentimes what is desired from a system will require understanding and leveraging a variety of perspectives, talents, and technologies. These solutions can be esoteric and bespoke to individual HPC deployments or teams, but over the years and across many sites and projects some common themes emerge. Here we present a grab-bag of some of the more common challenges (or scenarios that offer opportunities) related to the online monitoring, analysis, and feedback dimensions of HPC.

All of this is to say that monitoring solutions are often deployed for one or two specific purposes, and so a discussion of some of those purposes is an important part of understanding the role of monitoring in HPC environments. In each case we will look at a selection of representative solutions.

5.1 Portability Frameworks as Monitoring Opportunities

Several underlying principles or themes are motivating this entire area of research into Online Monitoring, Analysis, and Feedback for HPC. One of them is productivity, and *portability is a key contributor to productivity.*

There have emerged many standards, toolkits, and techniques now to give HPC developers a consistent API to address, with consistent behavior, that will achieve the same effect portably on different systems, past, present, and as far as can be anticipated into future of HPC architectures. Portability frameworks make for good instrumentation targets for a variety of reasons, beyond their widespread adoption. Typically they have well-documented semantics, the way in which they are used is consistent across various projects, and in recent times they often offer generic plugin-like interfaces for instrumentation or tools to connect to at runtime and interact with applications. Let's take a look at several of these solutions and their relationship to monitoring in HPC.

5.1.1 Distributed Computing. Distributed computing refers to a program running in parallel across several logical or physical compute resources, where each of the "ranks" of the program is isolated from the other and must communicate via the computer network interfaces rather than by being able to inspect each other's memory directly. It is no simple feat to connect multiple compute resources together in a way that enables software to run in multiple locations, discover, and coordinate in parallel to solve tasks. One would need to write networking code, become aware of load-balancing concerns, learn about efficient transport algorithms, interact with low-level device drivers, and implement all of this infrastructure quickly, bug-free, with security in mind, and then maintain it along with the main HPC application one set off to implement in the first place. Coordinating multiple distributed processes and meeting all of the above criteria is a dauntingly complex task, and yet this represents a very common need within HPC software. This has been the case for decades, and this has led in that time to the development and widespread adoption of several important solutions for distributed computing. Here are some of the mechanisms that have emerged over the years to help HPC developers meet these common challenges with a high-degree of productivity:

Message Passing Interface (MPI): Arguably one of the most influential and essential pieces of software in HPC, no conversation could be complete without discussing MPI [2]. With its roots going back into the late 1970s, MPI refers severally to its abstract model for distributed computation, a community-driven standard with official guides produced for each version, a fully-functional reference implementation of the standard (MVAPICH [58]), any standards-compliant API and library to link applications to, and a collection of runtime services deployed over a cluster in order to launch and manage the interactions of processes using MPI to send and receive messages.

In addition to MVAPICH, there are various alternative implementations such as OpenMPI [30] or more recently ExaMPI [68]. These alternative implementations are often able to provide compatibility with new or more experimental features of MPI before they make their way into the standard reference implementation. Vendor-specific implementations are also common, as this allows

the vendor to optimize the MPI runtime environment to fit and exploit features of their chipsets or a cluster's interconnect technology that may be protected intellectual property. An example of this is Intel MPI, and they even issued product-specific optimized MPI libraries like DCFA-MPI [74], tailored for their Intel Xeon Phi architecture.

Three common techniques for monitoring and interacting with MPI applications are through the use of the *MPI Profiling Interface* (PMPI, and more recently QMPI), and via the *MPI Tools Information Interface* (MPI_T). PMPI works by allowing for any calls to MPI routines to be intercepted by a tool which implements a wrapper function with the same signature, and then internally calls the actual MPI routine. PMPI is rather rudimentary in its functionality, in that it uses the linking phase of compilation to embed the tool into the application, connecting the application to the tool's implementation of certain API calls, and then connecting any "uninstrumented" MPI calls directly to the MPI library. This has the benefit of being extremely efficient, as the entire interface step can be compiled out of the application, as it is for any routines which are not intercepted by a tool, in the event a tool is being included. A negative consequence of this design is that, without some careful tool-to-tool coordination, only one tool at a time can be observing MPI activity, since the linker will select only one library to link any give MPI call to, at the exclusion of alternatives. Further, swapping from one tool to another can in some cases require an application to be rebuilt entirely. Any software which latches into MPI using PMPI in order to provide extended functionality will then prevent other tools from successfully doing the same at runtime, though often in a way that does not appear to fail to the tools which are excluded, though their routines do not get called. Depending on the order in which shared libraries are loaded and resolved by the host operating system, software with multiple components making use of PMPI can have undefined behavior without ever emitting errors a priori, which is deeply undesirable.

The QMPI [20] represents the most cutting edge enhancements to the classic PMPI model, and it provides a more flexible remedy for multiple-tool integrations, overcoming PMPI's limits while adding additional features and enhancements. QMPI allows for multiple tools to be registered, and for the wrapper routines of those tools to be executed concurrently, when calls are made to the parts of the MPI API the various tools have implemented hooks for.

Introduced in the MPI 3.1 standard[2], MPI_T provides a general purpose API and enumerated set of tags that tool writers can use to interrogate any standards-compliant MPI runtimes and get consistently formatted and representative metrics describing the system and job's configuration or activity. MPI_T also provides a standard interface for providing hints or adjusting the settings of the MPI runtime online, with varying degrees of control based on the specific type of directive given and the state of the MPI application in execution. These routines can be a great opportunity to perform direct monitoring, analysis, and automated tuning feedback, as has been done recently using a module of TAU [72] that engages with MPI_T so that TAU coordinates a parameter sweep for settings, observes and analyzes performance, and is able to optimize [61] MPI runtime settings online. MPI_T does not supplant the need for PMPI or QMPI, in fact may of the routines that MPI_T supports are implemented internally using calls to PMPI or QMPI.

5.2 Monitoring and Multiple Domains

Oftentimes it is beneficial, if not necessary, to combine observations of multiple layers or domains of a system in order to understand the behavior of individual applications or system components. When general end-to-end performance results for an application can be influenced by factors outside of the selection of algorithms or quality of the source code, it is especially useful to be able to observe beyond the source code measurements and application behavior. In 2011, Schultz, et al. identified and discussed [70] three high-level domains of analysis: Hardware, Application, and Communication. These *intuitive domains* have some natural overlap, but a result of their work was observing an increase in understanding of performance data when observations from one domain could be overlayed or projected over observations made in another.

LBNL's NetLogger [84] monitoring tool from the late 1990s used source instrumentation to capture and log performance measurements, and included a suite of offline analysis scripts to assemble detailed graphs showing *flow of application data* through a process, including measurement of time data moved between processes over a network. Events within NetLogger refer to traces of the processing of individual chunks of data. Flowing from their source-annotated origin as a logical application event, timing data could be captured showing processing through the hardware stages of loading into cache, being operated on, being queued up, transmitted, and ultimately received for processing on the remote of a distributed parallel system. Tracing chunks of application data rather than logging flat application performance measurements on a per-code-block basis gave this tool the power to reveal the complex interactions and emergent processing bottlenecks which might occur, and express these observations in a way that was relevant and actionable to the application user or developer. It also allowed for the performance impacts with origins outside of the application to be revealed in terms of their influence on the specific behaviors of the application, which can be a great help when determining where to focus effort when attempting to improve application performance.

Uniting observations from sources of information across multiple domains can become a challenge in itself, with many factors impacting the feasibility of the task and the overhead of the mechanisms engaged to provide a solution. The Scalable Observation System (SOS) [92] was introduced in 2016 with the notion of facilitating this kind of cross-domain online monitoring, engineered to be optimized for interacting with HPC systems and applications without introducing excessive overhead or blocking the progress of components which might interact with the SOS runtime. Later in 2017, SOS was used along with the ALPINE [40] in situ scientific visualization infrastructure to automatically capture and project performance data over the geometry being simulated by the application. The SOS and ALPINE integration captured a number of different performance observations, aggregated them online, and allowed for the simulation to be observed in real-time as the geometry evolved. Users could then select among the available performance measurements and have that projected out over the geometry of the simulation. This projection of hardware performance measures into the application domain allowed for an application developer to observe the performance of their code not in terms of individual

code regions, but in terms of the complex behaviors that emerge dynamically as a simulation progresses.

An application algorithm might begin to drift away from optimality in certain conditions, and it is beneficial to easily identify those conditions, perhaps to then design and introduce an updated behavior in the application that can perform a test and then switch processing over to the most suitable algorithm. For example, as two elements in a system approach each other and begin to influence each other within the simulation, an approach like this could make visible as hotspots such conditions as cache misses, mapped out over the surface of those elements. Markers could be displayed over those elements in the full context of the scene, or animated as the simulation progresses, indicating such things as an increased number of messages between the two parallel application ranks each responsible for one of the two elements.

While it is possible to discover many origins of performance issues through direct analysis of tabulated measurements, or by using traditional performance measurement tools [94], the ability to watch a simulation evolve online and immediately see the relative performance disturbances in brightly-enunciated graphical forms, paired to the phenomena which trigger the degradation of performance, makes the task of finding and fixing input-dependent issues much more straightforward.

5.3 Online Monitoring for Large and Complex Codes

Tools that automatically pinpoint certain aspects of arbitrarily complex software stacks through online monitoring, facilitating discovery and correction of bugs or execution bottlenecks.

Diagnosing performance variation in an HPC environment, automatically, online, or otherwise, is a significant challenge. Experiments [86] [50] show that it *is a problem that indeed can be solved*, despite the numerous difficulties to overcome, and so the great work ever continues. There are at present no one-size-fits-all solutions, and solutions that are being designed and deployed [67] use parts of other solutions, or take inspiration from many other projects.

6 CONCLUDING REMARKS

Online monitoring for HPC is a vast and complex field with many different motivations and trade/offs. As long as HPC architectures are evolving, compute loads are changing, and scales are growing, there will be a need for innovative ideas and new research efforts.

As the complexity and dynamism and interactivity of HPC workloads continues to increase for the foreseeable future, taking everything presented so far into consideration, it is the online monitoring solutions that are suitable to production environments and not only development environments which seem to offer the most promise of research impact and long-term value to the community. Especially promising are the ideas and designs for in situ (online) solutions to support automated monitoring and feedback directly into applications, to help the runtime environment and the applications within it adapt and become better fitting to maximize performance without costly code interventions or human-in-the-loop supervision by developers and users.

REFERENCES

[1] [n.d.]. *Zabbix Distributed Monitoring Solution.* https://www.zabbix.com

[2] 2015. MPI 3.1 Report. https://www.mpi-forum.org/docs/mpi-3.1

[3] 2020. collectd: The system statistics collection daemon. https://collectd.org

[4] 2020. *Performance Co-Pilot: System-Wide Monitoring.* https://pcp.io

[5] Omar Aaziz, Jonathan Cook, and Hadi Sharifi. 2015. Push me pull you: Integrating opposing data transport modes for efficient hpc application monitoring. In *2015 IEEE International Conference on Cluster Computing*. IEEE, 674–681.

[6] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[7] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, et al. 2014. The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 154–165.

[8] Anthony Agelastos, Benjamin Allan, Jim Brandt, Ann Gentile, Sophia Lefantzi, Steve Monk, Jeff Ogden, Mahesh Rajan, and Joel Stevenson. 2015. Toward rapid understanding of production HPC applications and systems. In *2015 IEEE International Conference on Cluster Computing*. IEEE, 464–473.

[9] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. 2010. Periscope: An online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009*. Springer, 1–16.

[10] Abhinav Bhatele, Kathryn Mohror, Steven H Langer, and Katherine E Isaacs. 2013. There goes the neighborhood: performance degradation due to nearby jobs. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[11] Tim Bird. 2009. Measuring function duration with ftrace. In *Proceedings of the Linux Symposium*. Citeseer, 47–54.

[12] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: performance introspection for HPC software stacks. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 550–560.

[13] David Boehme, Kevin Huck, Jonathan Madsen, and Josef Weidendorfer. 2019. The Case for a Common Instrumentation Interface for HPC Codes. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. IEEE, 33–39.

[14] Peter Braam. 2019. The Lustre storage architecture. *arXiv preprint arXiv:1903.01955* (2019).

[15] Nicolas Chan. 2019. A Resource Utilization Analytics Platform Using Grafana and Telegraf for the Savio Supercluster. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*. 1–6.

[16] David Couturier and Michel R Dagenais. 2015. LTTng CLUST: a system-wide unified CPU and GPU tracing tool for OpenCL applications. *Advances in Software Engineering* 2015 (2015).

[17] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[18] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. 2015. Partitioned global address space languages. *ACM Computing Surveys (CSUR)* 47, 4 (2015), 1–27.

[19] Alexandre E Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. 2013. OMPT: An OpenMP tools application programming interface for performance analysis. In *International Workshop on OpenMP*. Springer, 171–185.

[20] Bengisu Elis, Dai Yang, Olga Pearce, Kathryn Mohror, and Martin Schulz. 2020. QMPI: a next generation MPI profiling interface for modern HPC platforms. *Parallel Comput.* (2020), 102635.

[21] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E Nagel, and Felix Wolf. 2011. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries.. In *PARCO*, Vol. 22. 481–490.

[22] Todd Evans, William L Barth, James C Browne, Robert L DeLeon, Thomas R Furlani, Steven M Gallo, Matthew D Jones, and Abani K Patra. 2014. Comprehensive resource use monitoring for HPC systems with TACC stats. In *2014 First International Workshop on HPC User Support Tools*. IEEE, 13–21.

[23] Steven Feldman, Deli Zhang, Damian Dechev, and James Brandt. 2015. Extending LDMS to enable performance monitoring in multi-core applications. In *2015 IEEE International Conference on Cluster Computing*. IEEE, 717–720.

[24] Stefano Forti, Marco Gaglianese, and Antonio Brogi. [n.d.]. Lightweight self-organising distributed monitoring of Fog infrastructures. *Future Generation Computer Systems* 114 ([n. d.]), 605–618.

[25] Pierre-Marc Fournier, Mathieu Desnoyers, and Michel R Dagenais. 2009. Combined tracing of the kernel and applications with LTTng. In *Proceedings of the 2009 linux symposium*. Citeseer, 87–93.

[26] Daichi Fukui, Mamoru Shimaoka, Hiroki Mikami, Dominic Hillenbrand, Hideo Yamamoto, Keiji Kimura, and Hironori Kasahara. 2015. Annotatable systrace: an extended Linux ftrace for tracing a parallelized program. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems*. 21–25.

[27] Mohamad Gebai and Michel R Dagenais. 2018. Survey and analysis of kernel and userspace tracers on Linux: Design, implementation, and overhead. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 1–33.

[28] Alfredo Giménez, Todd Gamblin, Abhinav Bhatele, Chad Wood, Kathleen Shoga, Aniruddha Marathe, Peer-Timo Bremer, Bernd Hamann, and Martin Schulz. 2017. ScrubJay: deriving knowledge from the disarray of HPC performance data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[29] Alfredo A. Gimenez and USDOE National Nuclear Security Administration. 2018. Sonar. https://doi.org/10.11578/dc.20190131.3

[30] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. 2005. Open MPI: A flexible high performance MPI. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 228–239.

[31] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing* 22, 6 (1996), 789–828.

[32] Carla Guillen, Wolfram Hesse, and Matthias Brehm. 2014. The PerSyst monitoring tool. In *European Conference on Parallel Processing*. Springer, 363–374.

[33] Ramin Izadpanah, Benjamin A Allan, Damian Dechev, and Jim Brandt. 2019. Production application performance data streaming for system monitoring. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 4, 2 (2019), 1–25.

[34] Ramin Izadpanah, Nichamon Naksinehaboon, Jim Brandt, Ann Gentile, and Damian Dechev. 2018. Integrating low-latency analysis into HPC system monitoring. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–10.

[35] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. 1–11.

[36] Laxmikant V Kale and Sanjeev Krishnan. 1993. Charm++ A portable concurrent object oriented system based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. 91–108.

[37] Gregory Katsaros, Roland Kübert, and Georgina Gallizo. 2011. Building a service-oriented monitoring framework with rest and nagios. In *2011 IEEE International Conference on Services Computing*. IEEE, 426–431.

[38] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E Nagel. 2006. Introducing the open trace format (OTF). In *International Conference on Computational Science*. Springer, 526–533.

[39] Steffen Lammel, Felix Zahn, and Holger Fröning. 2016. Sonar: Automated communication characterization for hpc applications. In *International Conference on High Performance Computing*. Springer, 98–114.

[40] Matthew Larsen, James Aherns, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. 2017. The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman. In *Proceedings of the In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization Workshop (ISAV2017)*. ACM, New York, NY, USA.

[41] Allen D Malony and Wolfgang E Nagel. 2006. The open trace format (OTF) and open tracing for HPC. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. 24–es.

[42] Matthew L Massie, Brent N Chun, and David E Culler. 2004. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Comput.* 30, 7 (2004), 817–840.

[43] Jackson R Mayo, Frank Xiaoxiao Chen, Philippe Pierre Pebay, Matthew H Wong, David Thompson, Ann C Gentile, Diana C Roe, Vincent De Sapio, and James M Brandt. 2010. *Understanding large scale HPC systems through scalable monitoring and analysis.* Technical Report. Sandia National Laboratories.

[44] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. 2016. Modelling performance & resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*. 257–262.

[45] Kshitij Mehta, Bryce Allen, Matthew Wolf, Jeremy Logan, Eric Suchyta, Jong Choi, Keichi Takahashi, Igor Yakushin, Todd Munson, Ian Foster, et al. 2019. A Codesign Framework for Online Data Analysis and Reduction. In *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 11–20.

[46] Bernd Mohr, Allen D Malony, Sameer Shende, Felix Wolf, et al. 2001. Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In *Proceedings of the Third Workshop on OpenMP (EWOMP'01)*.

[47] Sophon Mongkolluksamee, Panita Pongpaibool, and Chavee Issariyapat. 2010. Strengths and limitations of Nagios as a network monitoring solution. In *Proceedings of the 7th International Joint Conference on Computer Science and Software Engineering (JCSSE 2010). Bangkok, Thailand*. 96–101.

[48] Mohammad Alaul Haque Monil, Bibek Wagle, Kevin Huck, and Hartmut Kaiser. [n.d.]. Adaptive auto-tuning in HPX using APEX. ([n. d.]).

[49] D Montaldo, E Mocskos, and D Fernández Slezak. 2009. Clover: Efficient Monitoring of HPC Clusters. (2009).

[50] Adam Morrow, Elisabeth Baseman, and Sean Blanchard. 2016. Ranking anomalous high performance computing sensor data using unsupervised clustering. In *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 629–632.

[51] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710.

[52] Charles J Murray. 1997. *The supermen: the story of Seymour Cray and the technical wizards behind the supercomputer*. John Wiley & Sons, Inc.

[53] Akihiro Nagai. 2011. Introduce New Branch Tracer 'perf branch'. *Linux Technology Center, Yokohama Research Lab, Hitachi Ltd., Copyright* (2011).

[54] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. 2017. Time series databases and influxdb. *Studienarbeit, Université Libre de Bruxelles* (2017).

[55] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[56] Tobias Oetiker. 2001. Monitoring your IT gear: the MRTG story. *IT professional* 3, 6 (2001), 44–48.

[57] Tobias Oetiker and Dave Rand. 1998. MRTG: The Multi Router Traffic Grapher.. In *LISA*, Vol. 98. 141–148.

[58] Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohammadreza Bayatpour. 2020. The MVAPICH project: Transforming research into high-performance MPI library for HPC community. *Journal of Computational Science* (2020), 101208.

[59] Simon Patarin and Mesaac Makpangou. 1999. Pandora: A flexible network monitoring platform. (1999).

[60] Kevin A Huck PI, Allen D Malony, and Monil Mohammad Alaul Haque. 2019. APEX/HPX Integration Specification for Phylanx. (2019).

[61] Srinivasan Ramesh, Aurèle Mahéo, Sameer Shende, Allen D Malony, Hari Subramoni, Amit Ruhela, and Dhabaleswar K DK Panda. 2018. MPI performance engineering with the MPI tool interface: the integration of MVAPICH and TAU. *Parallel Comput.* 77 (2018), 19–37.

[62] Prapaporn Rattanatamrong, Yoottana Boonpalit, Siwakorn Suwanjinda, Ayuth Mangmeesap, Ken Subraties, Vahid Daneshmand, Shava Smallen, and Jason Haga. 2020. Overhead Study of Telegraf as a Real-Time Monitoring Agent. In *2020 17th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, 42–46.

[63] Laurynas Riliskis, James Hong, and Philip Levis. 2015. Ravel: Programming iot applications as distributed models, views, and controllers. In *Proceedings of the 2015 International Workshop on Internet of Things towards Applications*. 1–6.

[64] Thomas Röhl, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2017. LIKWID Monitoring Stack: A flexible framework enabling job specific performance monitoring for the masses. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 781–784.

[65] Steven Rostedt. 2009. Finding origins of latencies using ftrace. *Proc. RT Linux WS* (2009).

[66] Federico D Sacerdoti, Mason J Katz, Matthew L Massie, and David E Culler. 2003. Wide area cluster monitoring with ganglia. In *null*. IEEE, 289.

[67] Sam Sanchez, Amanda Bonnie, Graham Van Heule, Conor Robinson, Adam DeConinck, Kathleen Kelly, Quellyn Snead, and J Brandt. 2016. Design and Implementation of a Scalable HPC Monitoring System. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1721–1725.

[68] Derek Schafer, Ignacio Laguna, and Kathryn Mohror. 2020. ExaMPI: A Modern Design and Implementation to Accelerate Message Passing Interface Innovation. In *High Performance Computing: 6th Latin American Conference, CARLA 2019, Turrialba, Costa Rica, September 25–27, 2019, Revised Selected Papers*, Vol. 1087. Springer Nature, 153.

[69] Frank B Schmuck and Roger L Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters.. In *FAST*, Vol. 2.

[70] Martin Schulz, Joshua A Levine, Peer-Timo Bremer, Todd Gamblin, and Valerio Pascucci. 2011. Interpreting performance data across intuitive domains. In *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 206–215.

[71] Hadi Sharifi, Omar Aaziz, and Jonathan Cook. 2015. Monitoring HPC applications in the production environment. In *Proceedings of the 2nd Workshop on Parallel Programming for Analytics Applications*. 39–47.

[72] Sameer Shende, A Malony, G Allen, J Carver, Sct Choi, T Crick, and MR Crusoe. 2016. Using TAU for performance evaluation of scientific software. In *Workshop on Sustainable Software for Science: Practice and Experiences*.

[73] Basem Shihada. 2002. Conceptual & Concrete Architectures of Open Network Management System (OpenNMS).

[74] Min Si, Yutaka Ishikawa, and Masamichi Tatagi. 2013. Direct MPI library for Intel Xeon Phi co-processors. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 816–824.

[75] Ed Simmonds and Jason Harrington. 2009. SCF/FEF Evaluation of Nagios and Zabbix Monitoring Systems. *SCF/FEF* (2009), 1–9.

[76] David George Solt, Joshua Hursey, Austen Lauria, Dahai Guo, and Xin Guo. 2019. Scalable, Fault-Tolerant Job Step Management for High Performance Systems. *IBM Journal of Research and Development* (2019).

[77] Matthew J Sottile and Ronald G Minnich. 2002. Supermon: A high-speed cluster monitoring system. In *Proceedings. IEEE International Conference on Cluster Computing*. IEEE, 39–46.

[78] Luka Stanisic and Klaus Reuter. 2019. MPCDF HPC Performance Monitoring System: Enabling Insight via Job-Specific Analysis. In *European Conference on Parallel Processing*. Springer, 613–625.

[79] Nitin Sukhija and Elizabeth Bautista. 2019. Towards a Framework for Monitoring and Analyzing High Performance Computing Environments Using Kubernetes and Prometheus. In *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 257–262.

[80] Nitin Sukhija, Elizabeth Bautista, Owen James, Daniel Gens, Siqi Deng, Yulok Lam, Tony Quan, and Basil Lalli. 2020. Event Management and Monitoring Framework for HPC Environments using ServiceNow and Prometheus. In *Proceedings of the 12th International Conference on Management of Digital EcoSystems*. 149–156.

[81] Liyang Sun, Guibin Tian, Guanyu Zhu, Yong Liu, Hang Shi, and David Dai. 2018. Multipath IP Routing on End Devices: Motivation, Design, and Performance. In *2018 IFIP networking conference (IFIP networking) and workshops*. IEEE, 1–9.

[82] Benjamin Taubmann and Hans P Reiser. 2020. Towards Hypervisor Support for Enhancing the Performance of Virtual Machine Introspection. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 41–54.

[83] James E Thornton. 1980. The CDC 6600 Project. *Annals of the History of Computing* 2, 4 (1980), 338–348.

[84] Brian Tierney, William Johnston, Brian Crowley, Gary Hoo, Chris Brooks, and Dan Gunter. 1999. The NetLogger Methodology for High Performance Distributed Systems Performance Analysis. (12 1999). https://doi.org/10.2172/764331

[85] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 207–216.

[86] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J Leung, Manuel Egele, and Ayse K Coskun. 2017. Diagnosing performance variations in HPC applications using machine learning. In *International Supercomputing Conference*. Springer, 355–373.

[87] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. 2016. Ravel: A database-defined network. In *Proceedings of the Symposium on SDN Research*. 1–7.

[88] Tao Wang, Jiwei Xu, Wenbo Zhang, Zeyu Gu, and Hua Zhong. 2018. Self-adaptive cloud monitoring with online anomaly detection. *Future Generation Computer Systems* 80 (2018), 89–101.

[89] Joseph P White, Martins Innus, Robert L Deleon, Matthew D Jones, and Thomas R Furlani. 2020. Monitoring and Analysis of Power Consumption on HPC Clusters using XDMoD. In *Practice and Experience in Advanced Research Computing*. 112–119.

[90] William R Williams, Xiaozhu Meng, Benjamin Welton, and Barton P Miller. 2016. Dyninst and MRNet: Foundational infrastructure for parallel tools. In *Tools for High Performance Computing 2015*. Springer, 1–16.

[91] Andrew M Wissink, Richard D Hornung, Scott R Kohn, Steve S Smith, and Noah Elliott. 2001. Large scale parallel structured AMR calculations using the SAMRAI framework. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. 6–6.

[92] Chad Wood, Sudhanshu Sane, Daniel Ellsworth, Alfredo Gimenez, Kevin Huck, Todd Gamblin, and Allen Malony. 2016. A scalable observation system for introspection and in situ analytics. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*. IEEE Press, 42–49.

[93] Justin M Wozniak, Timothy G Armstrong, Michael Wilde, Daniel S Katz, Ewing Lusk, and Ian T Foster. 2013. Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 95–102.

[94] Cong Xie and Wei Xu. 2018. *Performance visualization for TAU instrumented scientific workflows*. Technical Report. Brookhaven National Lab.(BNL), Upton, NY (United States).

[95] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. 2007. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*. 24–32.

[96] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 44–60.

[97] Shuai Zhang, I-Ling Yen, and Farokh B Bastani. 2016. Toward semantic enhancement of monitoring data repository. In *2016 IEEE Tenth International Conference on Semantic Computing (ICSC)*. IEEE, 140–147.