Observability, Monitoring, and In Situ Analytics in Exascale Applications

Dewi Yokelson[1], Oskar Lappi[3], Srinivasan Ramesh[2], Miikka Väisälä[4], Kevin Huck[1] Touko Puro[3]
Boyana Norris[1], Maarit Korpi-Lagg[5], Keijo Heljanko[3], Allen D. Malony[1]

[1]Department of Computer Science
University of Oregon

[2]NVIDIA
Corporation

[3]Department of Computer Science
University of Helsinki

[4]Institute of Astronomy and Astrophysics
Academia Sinica

[5]Department of Computer Science
Aalto University

***Abstract*—With the rise of exascale systems and large, data-centric workflows, the need to observe and analyze high performance computing (HPC) applications during their execution is becoming increasingly important. HPC applications are typically not designed with online monitoring in mind, therefore, the observability challenge lies in being able to access and analyze interesting events with low overhead while seamlessly integrating such capabilities into existing and new applications. We explore how our service-based observation, monitoring, and analytics (SOMA) approach to collecting and aggregating both application-specific diagnostic data and performance data addresses these needs. We present our SOMA framework and demonstrate its viability with LULESH, a hydrodynamics proxy application. Then we focus on Astaroth, a multi-GPU library for stencil computations, highlighting the integration of the TAU and APEX performance tools and SOMA for application and performance data monitoring.**

I. Introduction

There is a growing interest in enabling greater "observability" of HPC applications. While HPC tools for performance measurement, visualization and data checkpointing can be regarded as "observing" an application during its execution, traditional HPC applications are not generally designed with observability in mind. Most tools are tightly integrated with the application's implementation, reducing their flexibility, extendibility, and portability. When new functionality, such as runtime performance monitoring or in situ data analytics is required, approaches are often severely constrained by how the HPC application is implemented and executed. Furthermore, the rise of scalable heterogeneous HPC systems is increasing execution dynamics and the need for improved observational awareness. In particular, the ability to track, analyze, and interrogate interesting events and phenomena about the application and system is important to support the next generation of exascale solutions.

Our interest in HPC observability is motivated by the problem of large-scale performance monitoring and analysis. Specifically, given robust technology for heterogeneous performance measurement (e.g., *TAU Performance System* ® [27], HPCToolkit [2], and CALIPER [6]), how can real-time access to performance data and its in situ processing (e.g., to identify runtime performance issues and possibly feedback actionable results) be realized, with minimal impact on the application and efficiency? While building performance monitoring and analysis technology that can seamlessly integrate with an HPC application is a challenge, the objectives are not unique intrinsically, as they are shared with other domains such as simulation data analytics and visualization.

Currently, we are investigating a ***S**ervice-based **O**bservability, **M**onitoring, and **A**nalytics (**SOMA**)* strategy to meet this challenge. The research work follows on successful outcomes of our SYMBIOMON [35] and SERVIZ [34] projects where we gained experience with the high-performance Mochi [37] microservice ecosystem. A SOMA framework has been under development with initial objectives of demonstrating flexible configuration, portable data representation, access to performance measurement infrastructure, and easy-to-use interfaces for application integration. Coincidentally, an opportunity presented itself to test an initial prototype of SOMA with an astrophysical hydrodynamics application based on a multi-GPU magnetohydrodynamics solver and the Astaroth stencil library [32]. The application is targeting the EU's flagship supercomputer in Finland, the *Large Unified Modern Infrastructure (LUMI)*, an HPE Cray EX supercomputer that is the #3 machine on the Top500 [9] at this time. Given our interest in using TAU [27] and APEX [17] performance tools with LUMI applications, the opportunity was accepted. This paper reports on the progress of our efforts.

While aspects of our work are still in flux and results are preliminary, the SOMA design approach is validated and prototype functionality demonstrated with a real application on an HPC platform. The paper makes the following research contributions:

- Describing SOMA design and development methodology.
- Validation study of SOMA prototype with LULESH application demonstrating different functional features and configurations on CSC supercomputer Mahti.
- Integration of TAU and APEX with Astaroth and example use cases.

- Initial experiments of the SOMA prototype with Astaroth on the CSC supercomputer LUMI.

II. Astaroth

Astaroth is a multi-GPU library designed for high-order stencil computations on modern HPC systems [32]. Recently, it has been applied to build a simulation framework for magnetized astrophysical plasmas in the magnetohydrodynamics (MHD) regime, for details of the physics and first production runs, see [43]. The framework solves the standard set of partial differential equations for MHD, namely the continuity, angular momentum, entropy, and induction equations, under conditions that usually occur in astrophysical plasmas. In such plasmas, the densities and temperatures usually range several orders of magnitude, in which case non-conservative formulation of the equations is numerically advantageous over flux conserving schemes. For example, a formulation in terms of logarithmic density, albeit non-conservative, can be numerically more accurate and faster to compute. In this case, however, there is no guarantee that the conserved quantities are accurate to the machine precision, but rather conserved up to the discretisation error of the scheme. Therefore, constant monitoring of the conserved quantities is necessary, and simulations that do not adequately conserve them should be disregarded. Magnetic fields are implemented in terms of the magnetic vector potential to ensure that the field remains divergence-free.

A full-fledged multi-node implementation was taken into production during the LUMI-G pilot phase to study a setup intended for investigating the solar fluctuation dynamo, the physics and highest-resolution CPU simulations so far are described in [44]. Astaroth allows for scenarios of unprecedented resolution, but this comes with the cost of several hundred terabytes per system state. The analysis and movement of such data has become a major bottleneck of performing large-scale computations. This motivates the idea to do data analysis in situ, thus reducing the amount of data that eventually needs to be stored.

Astaroth operates in a *single-program, multiple data* (SPMD) manner, with one MPI rank per GPU device. LUMI-G nodes house four AMD MI250x GPUs each, with each GPU consisting of two Graphics Compute Dies (GCDs). HIP considers each GCD a separate device, and so Astaroth maximally uses eight MPI ranks per node, one per GCD. [1] With Astaroth only using eight processes per node, many CPU cores remain idle during the computation. These CPU cores could be used for data analysis *in situ*.

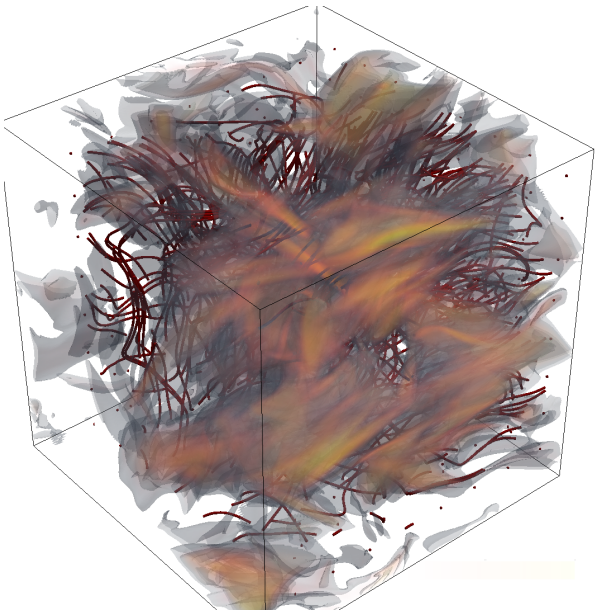

Fig. 1. Visualization of magnetic field lines (dark red streamlines) and its intensity (volume-rendered colours) in a dynamo-active Astaroth simulation from [43].

III. Service-based Performance Observability

A. Status Quo

The dominant computational model for scalable HPC applications is Single Program, Multiple Data (SPMD). SPMD programs combine distributed-memory parallelism with some form of shared-memory parallelism. MPI is the leading library for message passing [40], OpenMP [7] is popular for multi-threading on multi-core CPUs and CUDA [30] or HIP [18] support accelerator programming. Most HPC performance tools have been developed to support SPMD applications. Measurement libraries are primarily implemented to execute with the application code and run on the same processes and computational resources. Performance measurements are made in the context of processes/threads and stored in the application memory, making them low-overhead and highly efficient. For the most part, measurements are node-local and performance data is collected and written at the end of the execution.

Unfortunately, the SPMD model and its implementation with MPI can constrain the development of new HPC tool functionality. Consider a simple case of wanting to compute performance statistics from performance data across all MPI processes while the application is executing. While MPI does offer support for both synchronous and asynchronous methods of computing these statistics, running MPI in threading mode is notably more difficult than its traditional synchronous counterpart, either because of the limited support for threaded MPI, or because it contorts the MPI programming, making it awkward to integrate more advanced analytics solutions and forcing the "shoe-horning" of arbitrary analytics functionality into an MPI program. Further, there is another practical problem encountered on many leadership class clusters — some large HPC platforms disallow the running of multiple binaries (programs) on the nodes used by an application. In other cases, MPI environments do not allow the number of application ranks on a single node to exceed the number of available cores (known as *oversubscription*).

[1] During the LUMI pilot, Astaroth ran on 1024 nodes and 8192 GCDs.

B. Special Case of Idle Cores

Heterogeneous HPC places an emphasis on the use of accelerated devices, primarily GPUs, for achieving performance gains. In some heterogeneous applications that take advantage of GPUs it can be the case that the CPUs (cores) on accelerated nodes are under-utilized during the execution. One motivation for our monitoring research is to take advantage of such situations by finding techniques to locate monitoring processes on the same nodes as the application. Suppose we consider a special case where an MPI application does not use all of the available cores on each node allocated to it. Instead, the cores remain idle for the entirety of the execution.

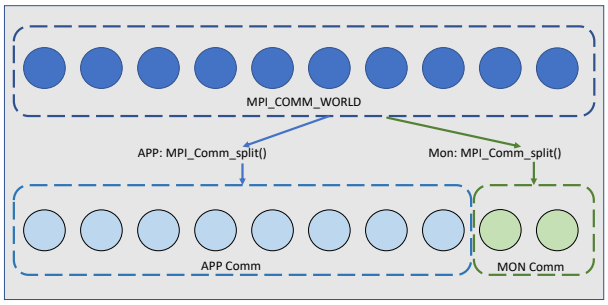

Fig. 2. Example of how `MPI_Comm_split` works to run two applications on the same node.

Without loss of generality, consider an MPI-based application configured to run with R total ranks on N nodes where each node has $r = R/N$ ranks per nodes (assume N evenly divides R). Let C be the number of CPU cores on a node and c be the number of cores not used by the application at all during execution. These unused CPU cores could be available to run monitoring processes. Suppose M total monitoring processes are to run with $m = M/N$ monitoring processes on each node (assume N evenly divides M). The question is then how to create the monitoring processes and configure them to run with the application such that m are assigned to each node.

One way of doing this is with the *Multiple Program, Multiple Data* (MPMD) approach can be done entirely in MPI. The idea is to run the application with $R + M$ ranks and split `MPI_COMM_WORLD` into two communicators: one for the application (call it `APP`) and one for the monitor (call it `MON`). The application code must be modified to do the split and to use the `APP` communicator in place of `MPI_COMM_WORLD` throughout. When launching the computation, $r + m$ ranks must be allocated to each node. There are two potential downsides to this approach. First, it might be problematic to modify the application code, for several valid reasons. Second, it might be difficult to integrate the monitor code with the application code. This is necessary because a single binary that includes the application and monitor code is being run by the MPI processes. A simple example of this approach is shown in Figure 2.

A big advantage of the above approach is that it does not require anything special to be done by the job submission system. Another approach that does not involve changes to the application or monitor MPI communicators is to launch the application (with R ranks) and the monitor (with M ranks) as separate binaries at the same time on the same nodes where r application ranks and m monitor ranks are running on each node. This can be done using a script that is launched on N nodes and then executes the monitor and application. Each would be able to use `MPI_COMM_WORLD` for MPI operations. Each would be its own binary. A different technique might be needed for the application to discover the monitor above. For this approach to be viable, the environment must allow two different binaries to run at the same time on the same resource set. Unfortunately, for some platforms, this is not the case.

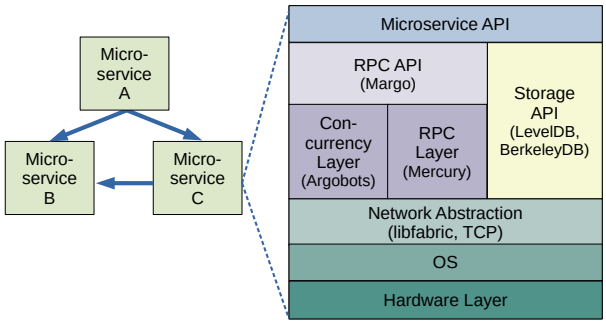

Fig. 3. Mochi microservice stack and example microservice.

C. Microservices

Being able to create monitoring processes alongside the HPC application and place them within the resource allocation (even on additional resources) is a first step. The question then becomes what code is being run on them and how do they interact with the application. We can think of a performance monitor as a coupled data service to an application for purposes of capturing and processing performance information. HPC data services have emerged as an essential component of coupled HPC workflow architectures. Mochi [37] is a software stack for developing data services built by composing individual microservices through the remote procedure call (RPC) as the communication mechanism. A *client* instance is the *origin* of the RPC and the *service provider* instance is the *target*. By providing a set of microservice building blocks, necessary tools, and a development environment, the Mochi framework enables the rapid development of customized functionality.

Figure 3 depicts three microservices (A, B, and C) interacting through RPC calls to generate different *call paths* through the network. These microservices can be located on the same process, on different processes within the same computing node, or on completely different nodes depending on how they are configured. The Mochi microservice software stack enabling this interaction consists of five core components: Mercury RPC library, Argobots, Margo, Thallium, and Scalable Service Groups. We focus on the first four:

- Mercury [41] is a high-performance RPC library that can utilize remote data memory access (RDMA) capabilities to transfer large RPC arguments efficiently.

- Argobots [38] is a lightweight user-level threading library that enables the development of highly concurrent software components.
- The Margo library provides a convenient abstraction that hides the complexities of programming the callback-driven Mercury library.
- Thallium is a header-only, C++ interface to Margo and is provided as a convenient wrapper to ease programming with Mochi.

D. SOMA Data Model

Inspired by the Mochi project [37] for composing HPC data services, we investigated the use of Mochi microservices for developing improved performance observability solutions. Our SYMBIOMON monitoring framework [35] demonstrates how a more flexible, configurable, and extensible monitor is possible with advanced distributed middleware that leverages high-performance interconnection hardware. We soon realized that other runtime HPC tools could benefit from this approach. Our SERVIZ [34] system for in situ visualization demonstrates another example of how service-based methods can help to overcome challenges of HPC observability.

SYMBIOMON is not flexible or general enough to serve as a starting point for our SOMA framework. In particular, it does not offer a solution for a shared representation of data between different workflow components. The time-series data model exposed by SYMBIOMON is not general enough to be applicable for monitoring any type of scientific or performance data. On the other hand, while SERVIZ is not a monitoring solution, it does utilize the Conduit [16] technology for representing and transferring simulation data between the application client and a visualization server.

Data representation and coupling between scientific code bases is a key challenge to building a vibrant ecosystem of HPC simulation tools. It requires agreeing on or adapting between data representations. This is also true for HPC application monitoring. It is not enough to set up SOMA services using Mochi on processes and execute RPC operations, since what data is sent and how it is represented matters. Conduit is an open-source project from Lawrence Livermore National Laboratory (LLNL) designed to simplify data description and sharing across HPC simulation tools. It provides an intuitive API for in-memory data description that enables human-friendly hierarchical data organization. There are commonly shared conventions for exchanging complex data and modular interfaces (in C++, C, Python, and Fortran) for use across software libraries and simulation applications. Conduit provides easy-to-use I/O interfaces for moving and storing data, including support for moving complex data with MPI (serialization). At the heart of Conduit is a hierarchical variant type called a *Node*. A Node can be used to capture and represent arbitrarily nested numeric data. Further, Conduit also provides convenient interfaces to serialize Conduit Nodes — we rely on this capability to store and transport monitoring data within the SOMA environment.

We apply the Conduit techniques to create a shared performance data representation that becomes the basis for data sharing across a SOMA environment. The Conduit data model was chosen for its ability to capture arbitrary hierarchical data. One clear example of hierarchical data would be TAU or APEX profiles. Listing 1 is an example of a TAU profile data capture in a Conduit representation for the LULESH application. The profile represents the data for MPI rank 0, which forms the second level of the Conduit::Node hierarchy, the first being the TAU namespace. Lower levels in the hierarchy represent the event classes, while the Conduit::Leaf nodes hold the actual interval timer or counter data.

```
{
 TAU:
    Rank 0:
        Uninstrumented:
            .TAU application_Calls: 1.0
            .TAU application_Inclusive: 13871247.0
        MPI_Routines:
            MPI_Init_thread()_Calls: 1.0
            MPI_Init_thread()_Inclusive: 992535.0
            MPI_Comm_rank()_Calls: 4507.0
            MPI_Comm_rank()_Inclusive: 3196.0
            MPI_Comm_split()_Calls: 1.0
            MPI_Comm_split()_Inclusive: 21108.0
            MPI_Comm_dup()_Calls: 1.0
            MPI_Comm_dup()_Inclusive: 31.0
            MPI_Comm_size()_Calls: 2.0
            MPI_Comm_size()_Inclusive: 1.0
            MPI_Barrier()_Calls: 2.0
            MPI_Barrier()_Inclusive: 321011.0
            MPI_Irecv()_Calls: 8507.0
            MPI_Irecv()_Inclusive: 5445.0
            MPI_Isend()_Calls: 5007.0
            MPI_Isend()_Inclusive: 16987.0
            MPI_Waitall()_Calls: 1501.0
            MPI_Waitall()_Inclusive: 143129.0
            MPI_Wait()_Calls: 8507.0
            MPI_Wait()_Inclusive: 32587.0
            MPI_Allreduce()_Calls: 499.0
            MPI_Allreduce()_Calls: 499.0
            MPI_Allreduce()_Inclusive: 5400.0
        TAU:
            Tau_plugin_mochi_dump_Calls: 500.0
            Tau_plugin_mochi_dump_Inclusive: 808.0
        MPI_Counters:
            Message size for all-reduce_Mean: 8.0
            Message size for all-reduce_Min: 8.0
            Message size for all-reduce_Max: 8.0
}
```

Listing 1. Conduit::Node of the TAU performance data model

However, SOMA does not have to be limited to only performance data. Monitoring of application execution state can be insightful to identify anomalous behavior and other artifacts. In this case, the data is application-specific, and the knowledge of how to best represent it lies with the application developer. Conduit is also gaining support in the scientific computing community as a data model to exchange scientific data within components in a workflow. SOMA, like SERVIZ and SYMBIOMON, is assembled out of robust, high-performance Mochi services. The use of a well-supported API and data model in Conduit is in line with our strategy of building a high-performance monitoring service using "off-the-shelf" components. This strategy promotes a high degree

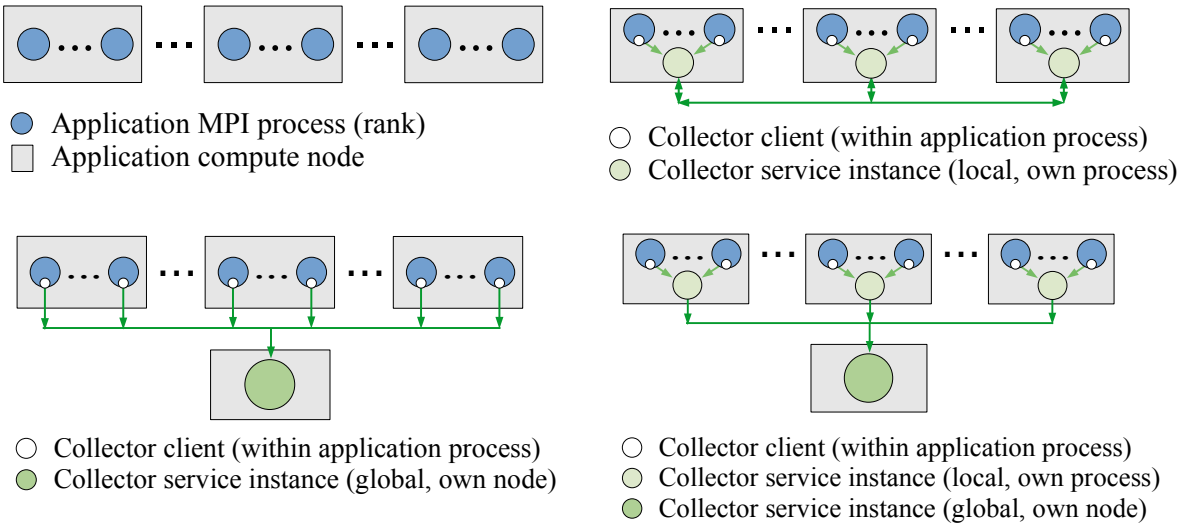

Fig. 4. **(Top-Left)**: Standard MPI application. **(Bottom-Left)**: A SOMA *client* is embedded within each MPI rank (i.e., process). The *client* is called via a TAU plug-in. The application is launched with an additional node on which a SOMA *service* instance executes. Each *client* will interact with the *service*. **(Top-Right)**: The *client* is the same as Bottom-Left. However, the application is launched with another process on each application node where an *service* executes. Each *client* will interact with the local *service* instance. The *services* will interact with each other for distributed processing. **(Bottom-Right)**: The *client* and local *service* are the same as the Top-Right. An additional node is allocated for a global *service* instance. The *service* instances will interact with each other and with the global *service* instance.

of code reuse, resulting in a monitoring service that is (1) easier to maintain and (2) whose functionality is easier to extend compared to ad-hoc implementations.

E. SOMA API

Table I depicts the entirety of the SOMA Collector API. The core API revolves around the idea of a *monitoring namespace*, borrowed from our earlier SYMBIOMON implementation. The creation of the namespace requires the user to supply a `string` argument representing the namespace name, following which an empty Conduit::Node is created inside the collector client's memory. Following creation, the namespace can be updated by providing a `key:value` pair, wherein the `key` represents the hierarchy level of the numeric data (e.g., "TAU/MPI/MPI_Allreduce"), and the value is the numeric data to be stored. Note that the top level in the hierarchy is always the namespace name — this name is automatically prefixed to the `key` argument and is not required to be supplied by the calling code. If the key exists, the value is either updated or appended to an existing list depending on the `operation_type` passed to the `soma_update_namespace` API call. If the key does not exist, a new Conduit::Leaf object is created.

A namespace that is updated is left in an *uncommitted* or *open* state until the user explicitly invokes the `soma_commit_namespace` API call. Committing a namespace decrements a *frequency counter* associated with the namespace. When this frequency counter reaches zero, `soma_publish_namespace` is triggered internally, resulting in an RPC call to the collector service instance carrying the payload of the namespace — a Conduit::Node object representing the data being monitored. The frequency counter can be set using the `soma_set_publish_frequency`

TABLE I
SOMA API DESCRIPTION

`soma_create_namespace`
Creates a SOMA namespace and returns a handle to the namespace
`soma_update_namespace`
Updates the SOMA namespace with hierarchical data in a key:value pair
`soma_publish`
Publishes a raw Conduit::Node to the SOMA collector service instance
`soma_publish_namespace`
Publishes the Conduit::Node underlying the namespace
`soma_commit_namespace`
Commits a namespace — akin to closing a file
`soma_set_publish_frequency`
Sets the monitoring frequency associated with a namespace
`soma_write`
Instructs the collector service to write Conduit::Node data to a file

API. Association of a monitoring frequency with a namespace in SOMA is an improvement over SYMBIOMON — the latter only allowed monitoring frequencies to be set on a per-metric basis, resulting in a flurry of RPCs in the system and a tendency for the monitoring system to be more network-latency-sensitive than necessary. SOMA also exposes an API to publish a raw Conduit::Node object directly to the collector instance. In summary, the SOMA API encompasses all of SYMBIOMON's features, while simultaneously being simpler, offering better performance, and being more generally applicable for scientific and performance data monitoring alike.

F. SOMA Implementation

SOMA is a Mochi microservice component implementing the API depicted in Table I. The collector client is stub that is linked against an application requiring SOMA monitoring capabilities. The collector service instance typically resides on a separate process and is contacted by means of an RPC call. The client-server abstraction allows SOMA to be configured

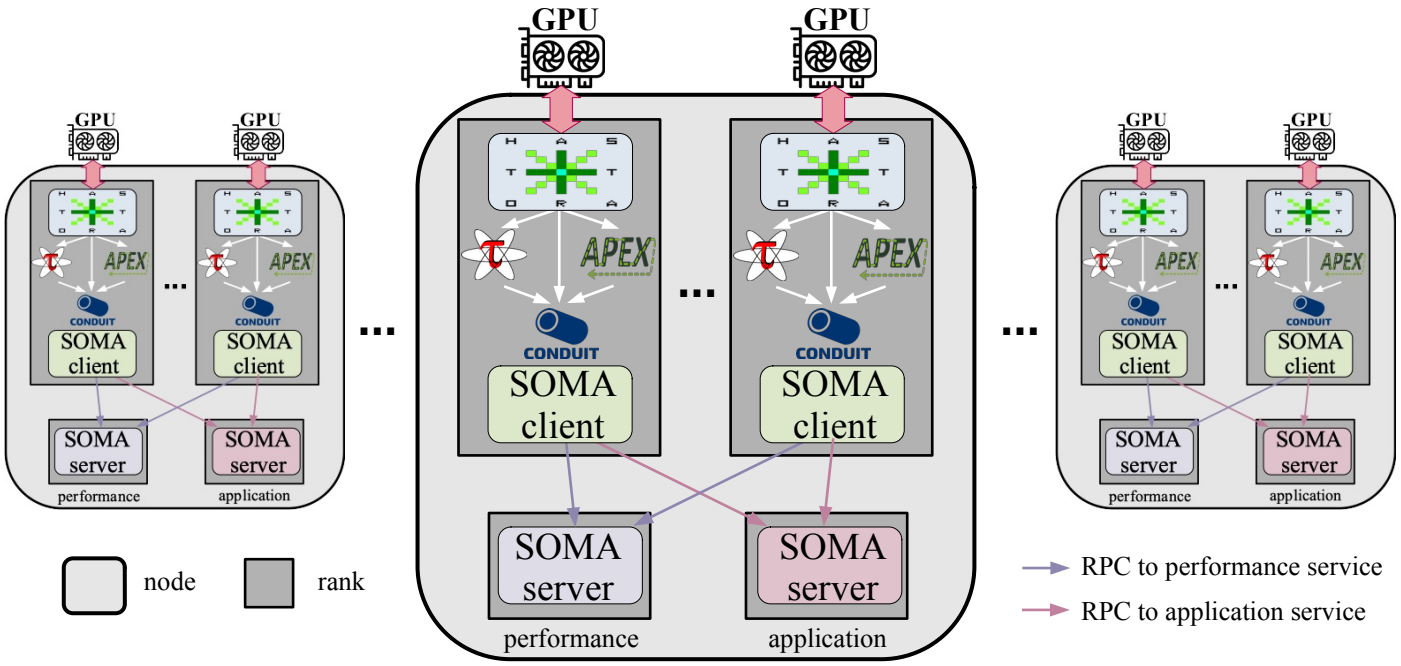

Fig. 5. An example configuration of SOMA with Astaroth. While Astaroth runs on the GPUs, we allocate extra ranks to run SOMA servers on the idle CPU. Data is collected from Astaroth in numerous ways, packaged as Conduit nodes and sent via RPC to the SOMA servers.

in a variety of ways without modifying the client stub code. Figure 5 depicts the SOMA stack in the context of an Astaroth application. Shown here is a scenario where each node is executing a SOMA performance data service instance and a SOMA application data service instance *alongside* the application, i.e., SOMA service instances share the computing node resources with Astaroth. SOMA can also be configured to run on a different node than Astaroth without *any* changes to the application or client stub code. The only difference between these two scenarios would be the datapath for the RPC calls — the former would involve shared-memory copies on the node (supported by most operating systems) while the latter would traverse the system network links. It is worth mentioning that Mochi microservices automatically support a variety of different communication "plugins" via Mercury — a few of which are `shared-memory`, `TCP`, and `verbs`. This plugin model allows us to switch between communication protocols through a simple configuration change, thereby circumventing the need for any code changes to the application client stub or SOMA service instance. Here we enlist the steps necessary for an application to establish a connection to SOMA and begin the monitoring workflow:

- **Generation of Server Addresses**: The SOMA MPI program is launched first, following which the division of the processes into the configured number of server instances takes place. Each process instantiates its SOMA service provider and makes its unique RPC address public through a file. SOMA implements *server instances* to support the scenario depicted in Figure 5. One instance can monitor scientific application data from Astaroth, while another instance can monitor TAU performance data. Once the RPC addresses are written out to an *address file*, the service is now ready to accept client requests.
- **Service Discovery**: Astaroth and TAU connect to SOMA by reading in the RPC address through the *address file* and creating a client object to manage the connection. This logic is housed inside the initialization routines of the client software. If there are N SOMA service instance ranks and M Astaroth (or TAU) client ranks such that M>N, our current implementation assigns the N SOMA ranks in a round-robin fashion to the M client ranks. Other client-server mapping strategies are also possible.
- **RPC Invocation**: Most SOMA API operations depicted in Table I are *local* operations, i.e., they execute directly inside the client stub memory. However, the `soma_publish` API results in a Mochi RPC call. When this RPC is invoked, the Conduit::Node underlying the SOMA namespace is serialized to a string representation using native Conduit routines. The resulting string representation is passed to Mercury, which serializes the string on the client, manages the data transfer through the chosen communication plugin, and de-serializes the data back into a string representation on the collector service instance. Serializing the Conduit::Node object to a string representation can be expensive if the Conduit::Node is large. In the future, we plan to explore binary representations as a way to reduce the RPC payload size, thereby improving performance.
- **RPC Execution**: The collector service instances can be effectively modeled as workers executing RPCs from their local work queue. When an RPC for

`soma_publish` is executed on the server, the string representation is converted back into a Conduit::Node and stored inside an in-memory queue. This queue is emptied upon receipt of a `soma_write` call on the collector service instance.

IV. Performance Tools

The TAU project began in the early 1990s with the goal of creating a performance instrumentation, measurement, and analysis framework that could produce robust, portable, and scalable performance tools for use in all parallel programs and systems over several technology generations. Today, the TAU Performance System® [39] is a ubiquitous performance tool suite for shared-memory, message passing, and task-based parallel applications written in multiple programming languages that can scale to the largest parallel machines available. It is installed on many HPC systems around the world and is used on a daily basis for performance analysis and tuning of applications across multiple domains.

The TAU Performance System consists of two toolkits: the tuning and analysis utilities (*TAU*) and the autonomic performance environment for exascale (*APEX*). The TAU model of performance measurement is based on a "worker" (first-person) perspective. Essentially, each thread of execution in a program will make performance measurements with respect to its operation. A measurement could occur as a results of an instrumentation probe the thread executes or an event-based sample interrupt that occurs on that thread. All performance data (e.g., time, HW counters) are stored within the thread context and retained during execution. All threads output their performance information when the program terminates. Many HPC performance tools are like TAU, including HPC-Toolkit [2], Score-P [23], Scalasca [45], and Caliper [6].

In contrast, APEX [17] is based on a "task" (third-person) perspective, with event-based and sample-based measurements. APEX uses an event API and event listeners to observe when a task is created, started, yielded or stopped, and updating timers for measurement. (Note, this is with respect to what constitutes a task, not necessarily its thread of execution.) Dependencies between tasks are also tracked, using globally unique identifiers (GUID). APEX periodically and on-demand interrogates (samples) OS, hardware, or runtime states (e.g., CPU utilization, resident set size, or memory "high water mark"). This also occurs in TAU, but in a different manner. APEX measurement includes background buffer processing to record GPU kernel execution and memory transfers to and from GPUs. Available runtime counters (e.g., idle rate, queue lengths) are also captured on-demand or on a periodic basis.

Both TAU and APEX can produce profiles and/or traces. With this in mind, the TAU profile data model provides for a type of analysis that can look at individual thread operation and the performance of particular events across multiple threads. It is possible to compute statistics for specific events to get a sense of aggregated performance. APEX is particularly appropriate for task-based runtime environments. Existing and emerging programming models present technical challenges that first-person measurement systems had not considered: untied task execution and migration, runtime thread control and execution, state sampling, and runtime performance tuning. APEX can address these issues while being lightweight enough to be present in an application for continuous performance introspection and adaptation.

Several programming systems and communication libraries implement a performance interface that allows tools to observe events and associated data associated with those components (e.g., OMPT [31] for OpenMP and PMPI [11] for MPI). Some adopt a plugin design that enables tool connection at runtime. However, user code instrumentation generally lacks support for tool interfaces. The PerfStubs library [5] is a thin, stubbed-out, "adapter" interface for instrumenting library or application code. The PerfStubs library itself does not do any measurement, it merely provides access to an API that performance tools can implement.

The Astaroth computational model benefits from both TAU and APEX performance measurement and analysis. Both provide process-level measurement, MPI, and GPUs, including NVIDIA and AMD. Because TAU and APEX implement the PerfStubs API, Astaroth can use PerfStubs to instrument high-level computation events. In addition, it is possible to instrument the task scheduler used in Astaroth [24] in order for APEX to observe asynchronous GPU kernel operations.

A. Performance Integration in Astaroth

While TAU includes an instrumentation API, adding it to Astaroth would significantly change the build configuration for the application and make TAU a build dependency. Rather than add TAU calls directly to capture timer data, TAU provides an instrumentation library called PerfStubs [5]. PerfStubs is self-contained, and creates a "stub" implementation of a timer library, with support for counters and metadata. At runtime, the PerfStubs library will check if specific symbols are implemented by any tool libraries in the environment, and if they are it will initialize the tool and pass any timer start/stop events to the tool. If no tool is present, PerfStubs will just return without any action.

Astaroth was instrumented in three locations: *pre-update*, *simulation*, and *post-update*. The *simulation* phase is the core simulation loop and consists of GPU kernel calls and MPI point-to-point communication calls, which are scheduled asynchronously to avoid global synchronization. The *pre-update* and *post-update* phases consist of control logic, global reductions, and output procedures, they contain calls to `MPI_Allreduce` and `MPI_Barrier` which force global synchronizations. These global MPI calls do not occur every timestep, but at configurable intervals. These three phases represent the computationally interesting portions of the code - all other time is attributed to initialization and finalization. Figure 6 demonstrates the mean time spent per function as measured by TAU profiling of Astaroth, including the instrumented functions, across 128 GPUs. Figure 7 shows the *simulation* phase of a benchmark when executed with 64

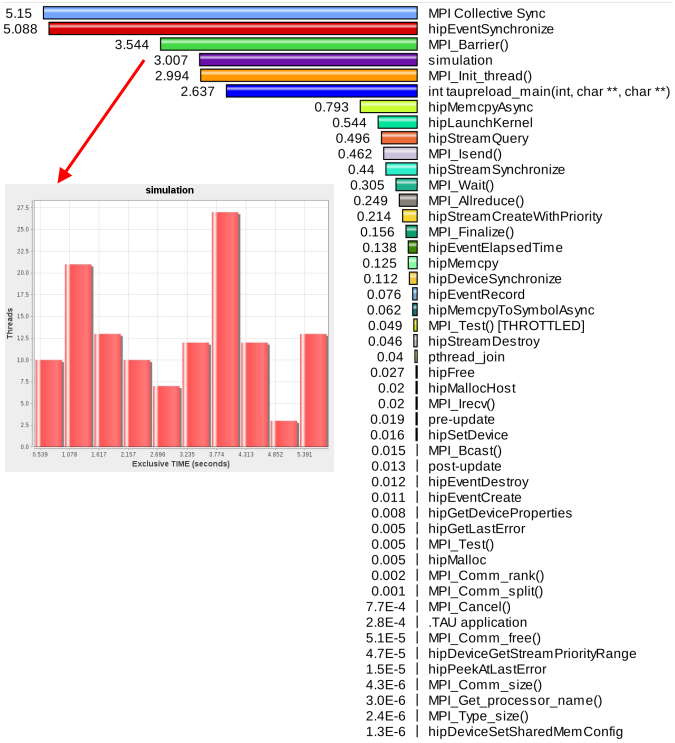

Fig. 6. The mean time spent per Astaroth function across all ranks for an Astaroth execution on LUMI-G (16-node, 128-GPU, 128 ranks). Inset shows the distribution of the `simulation` timer across ranks.

ranks and the APEX performance tool enabled. Even though only three timers were added to the application source code, APEX automatically captures MPI, HIP and GPU activity on both the CPU and the GPU. APEX is added to the application at run time with the `apex_exec` script, which preloads `libapex.so` and enables requested support through command line options. Each node in the graph is colored relative to the intensity of how much time is contributed to the overall execution time of the sub-tree.

B. Astaroth Performance Experiments

We benchmarked Astaroth with a grid resolution of 512^3, the total resident set size of the simulation is about 21 GB. We ran this benchmark in a strong scaling study, from 8 to 256 MPI ranks. The scaling behavior of the `pre-update`, `simulation`, and `post-update` timers are shown in Figure 8. While `simulation` has near perfect scaling, the `pre-update` and `post-update` phases both actually become more expensive with more MPI ranks, because they contain global synchronizations. The overhead starts to become noticeable at 64 MPI ranks. In Figure 9, we plotted the total time recorded in the `post-update` timer against the total time taken by `MPI_Allreduce`. The time taken by `MPI_Allreduce` starts to grow after 256 MPI ranks, and it seems to be the main contributor to the runtime of `post-update` at higher rank counts.

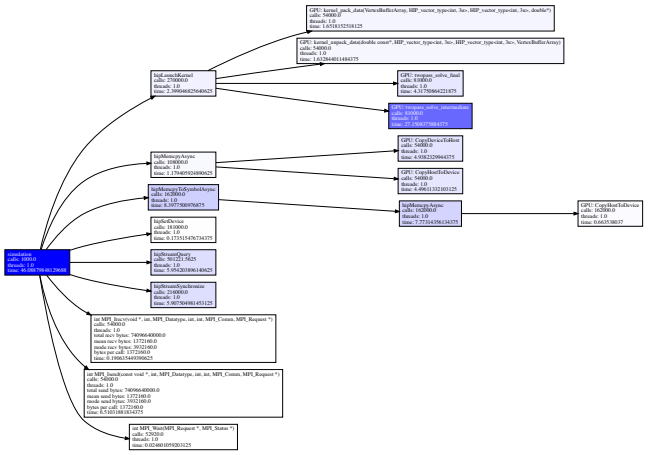

Fig. 7. Mean taskgraph from an Astaroth benchmark executed with 64 ranks. The graph represents a pruned sub-tree rooted at the *simulation* timer (other data is not shown), and shows both MPI and HIP API calls as well as the GPU activity that is launched from the HIP calls like memory transfers and kernel executions.

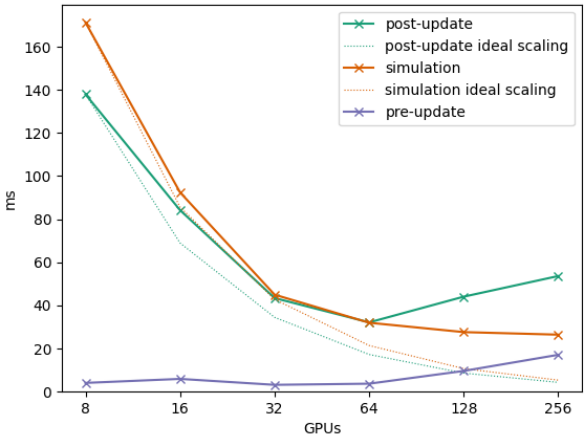

Fig. 8. Scaling behavior of the simulation phases, mean time per phase.

V. Astaroth and Performance Integrations

The type of data analysis that is most valuable for large-scale Astaroth runs is determining the numerical health of the simulation. Simulations sometimes fail due to misconfiguration of numerical parameters — the simulation becomes unstable or starts behaving nonphysically. For well-tested schemes this is less likely to occur, but during the development of Astaroth, new numerical methods are constantly being added. Without the experimental knowledge of how these new methods interact with Astaroth's existing system components, it is very difficult to create an intuitive understanding of the mechanisms that cause simulation failures. Being able to observe and analyze the numerical state of the simulation improves our ability to build an intuitive understanding of the numerical processes and how they should be tuned to avoid

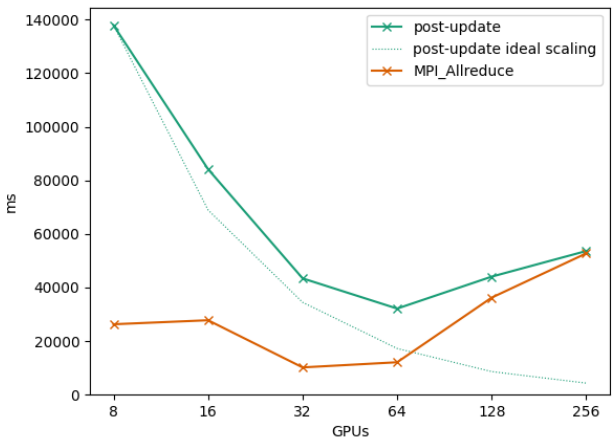

Fig. 9. Scaling behavior of the post-update phase and MPI_Allreduce, mean over ranks of the total time per timer.

failures. With increasing number of nodes and processing elements, in addition, the probability of bit flips and node failures increases, in which case monitoring the health of the system becomes of high importance, to avoid unnecessary crashes, corruption of data, and loss of computing time.

The most common symptom of a numerical failure in Astaroth is that the simulation becomes numerically unstable. Due to the non-conservative scheme, such an instability will manifest itself as a loss/gain in the quantity that is required to be conserved. Usually such events occur very localized in the simulation grid, where locally high gradients or extrema occur, and a numerical quantity will begin to grow uncontrollably. There are two possible actions once an instability starts developing: 1) the simulation is scrapped and reconfigured or reseeded to avoid the instability, 2) the simulation is recovered by reconfiguring the simulations parameters on the fly to counteract the instability. In both cases, instabilities should preferably be caught early. As the instability grows, it becomes harder and harder to recover any useful data, and if the simulation has to be scrapped, then one can iterate through simulation parameters more quickly if instabilities are caught early. Recovery is made even more difficult at high resolutions, where it is not feasible to keep more than two snapshots on disk. The snapshot frequency then determines the limit of how far back a simulation can be rewound. The period between snapshots forms a deadline within which Astaroth should detect instabilities.

In order to diagnose numerical instabilities, each Astaroth rank calculates certain reductions on the computational mesh that act as signals of a developing local instability. There are two kinds of data that we are interested in: 1) the numerical loss or gain of conserved quantities, 2) extreme values and their locations. The most important conserved quantity is the mass, on which we concentrate for this proof-of-concept study. For extrema, we record the minimum and maximum values of each field, and their corresponding location. Astaroth considers NaN values to be extremum and detects these as well. It is almost certainly too late to recover once the simulation has produced a NaN value, as they provide a clear indication that the simulation has failed. The mass is obtained as an integral over density, for this Astaroth uses a handcrafted reduction operation. The extrema and NaN locations are obtained using the thrust [4] library, which supports reductions on GPU buffers, both through CUDA and HIP.

Tracking extrema is useful not only for the detection of numerical instabilities, but also for identifying points or regions of interest within the simulation domain. By following the locations of extrema, we could carve out slices or volumes in the simulation domain for visualization and analysis, which would reduce the total amount of data needed to be stored on disk for post-processing. If this was done in situ, the observability of high-resolution Astaroth simulations would improve considerably.

Astaroth logs these signals, the mass conservation and extrema, but a human needs to interpret them in order to take action. The end goal in monitoring the signals is for Astaroth to be able to configure a policy in Astaroth that, e.g., pauses a simulation when one of these signals indicate a certain likelihood of an instability, or adjusts the simulation parameters on the fly, and continues computations in a numerically safer regime. Methods for calculating the likelihood of instabilities will be easier to develop in a separate component outside of Astaroth. A step towards this goal is to stream the diagnostics from Astaroth through a channel to which analysis components can subscribe. As examples, the diagnostics stream could be fed into a machine learning model doing inference, or a signal processing component that detects spikes in the extrema or fluctuations in the mass conservation.

SOMA provides an API through which such structured data can be published. This API is provided by a client library that Astaroth is linked against. We have set a publishing interval in Astaroth that determines how often Astaroth publishes diagnostics to SOMA. As a proof of concept, we write the data to a file and visualize this data in section VI-A. It is not difficult to imagine how SOMA could provide on-line analysis of the data through a plugin system, or forward the data to a completely separate analysis engine.

In this proof-of-concept study we use a low magnetic Prandtl number setup, meaning that the molecular magnetic resistivity in the induction equation is much higher than the viscosity in the Navier-Stokes equation. This setup is similar to [44], mimicking small-scale dynamo action in the solar and stellar convection zones. We initiate a non-zero but very small in magnitude seed magnetic field in the domain, and the crucial physics question is whether the flow field can act as a dynamo and exponentially amplify its seed? In such a setup, the plasma flow is highly turbulent, and Reynolds numbers, measuring the vigor of fluid turbulence, are high, while the magnetic Reynolds number can only be kept slightly above the threshold for dynamo action due to numerical reasons. In this kind of a system, the dynamo will grow, but very slowly, and the stability properties of the system are mainly determined by

the hydrodynamical part. Hence, in this paper, we concentrate on monitoring the conservation of mass and extrema of the flow field. In the study of [43], systems with magnetic Prandtl numbers of unity were investigated, see Fig. 1. In such a setup, the magnetic field can grow to a significant strength, and also participate in the dynamical evolution. In this case, monitoring also the magnetic field diagnostics becomes important. This is not the case in the setups presented in this paper, however.

VI. SOMA and Experiments

A. Monitoring Overhead Experiments

Our project objective was to build a SOMA framework for Astaroth that enables online performance monitoring, application-specific diagnostics, and simulation state data analytics. The use of microservice architectures allows us significant flexibility concerning how and where to run SOMA in relation to the application. The most straightforward configuration for monitoring *in general* would be to launch a job with extra compute nodes and run the SOMA monitoring service there. However, in the case of Astaroth, since there were idle CPU cores we implemented a scenario where the SOMA monitoring service ran on those idle CPUs.

Running two different binaries (applications) on the same compute node required splitting the MPI ranks appropriately between the different applications. Figure 2 shows how we split the `MPI_COMM_WORLD` communicator between our two applications. In this process, SLURM passes certain ranks to each application, and it is the responsibility of the application to assign these ranks to it's own communicator group. This functionality works without modifications in the case where all ranks on a node are assigned to one application as well – in this scenario the application just copies all ranks to its new communicator group. All experiments were carried out on either the LUMI-G or MAHTI systems detailed in Table II.

	LUMI-G	**MAHTI**
System	HPE Cray EX	Atos BullSequana XH2000
CPU	AMD EPYC 7653	AMD Rome 7H12
Total CPU Cores	64	128
Memory (GB)	512	256
# GPUs	8	4
GPU Arch	AMD MI250X	NVIDIA A100

TABLE II
Architectures used for the simulations and experiments.

1) LULESH: It was interesting to us to show the use of microservices in an application that could be configured similarly to Astaroth. For this purpose, we installed LULESH hydrodynamics proxy application [20], [21] on the CSC Mahti supercomputer and ran it successfully with SOMA in both the node-local and remote scenarios shown in Figure 4. In particular, we used an MPI-only version and allocated up to 64 ranks on each node, leaving 64 or more unused cores of the 128 cores available. We modified LULESH to launch with additional ranks per node and split the `MPI_COMM_WORLD` communicator to use certain ranks for LULESH and the rest for SOMA (see the top-right configuration in Figure 4).

These LULESH experiments were performed with a problem size of 45 (per domain) which is the equivalent of 5,832,000 elements. LULESH simulations must be run with a number of ranks that is also the cube of an integer. Because we used up to 64 ranks per node, while we increased the node count, we also had to calculate the ratio to keep the total ranks equal to the cube of an integer. We measure SOMA overhead by varying the monitoring frequency (how often both application and performance data is published to the servers), with every 50 application iterations, every 5 iterations, and every single iteration. Results shown in Figure 10 demonstrate interesting results as compared to the baseline (Monitoring Frequency = 0). The network protocol used for all Mahti experiments was ofi+verbs. We see that as expected, using a minimal monitoring frequency of 50 iterations has the least overhead. The configuration of running the SOMA monitoring ranks on a remote node from the application does typically outperform the case where we run the SOMA monitoring ranks on the local node.

When only running on a single application node, we see a trend of greater overhead percentage than more nodes, but the total execution time remains within range. This is based on an average of 5 runs in each configuration. However, as we scale LULESH to run on an increasing number of compute nodes, we see the overhead begin to converge slightly. Running the SOMA servers on an extra allocated node likely utilizes completely free resources, even though there are free cores on the local node, perhaps more free cores would be better. Further tuning can be done when looking at message size, publishing frequency, number of SOMA server instances per rank to find the best performance in each scenario.

2) Astaroth: We analyze the overhead of collecting application data via SOMA for Astaroth. The description and analysis of the data are in Section VI-B. For the purpose of measuring overhead, we conducted, first, baseline execution times for Astaroth without any monitoring. This baseline is pictured in Figure 11 with Monitoring Frequency = 0.

We scale Astaroth from one node up to 16 nodes on LUMI-G, which contains 8 GPUs per CPU node. In all scenarios we run 8 Astaroth ranks per node to utilize all of the GPUs available. The grid size of Astaroth must be adjusted for each scaled run, beginning with x,y,z dimensions of 256,256,256 for one rank and multiplying each dimension by the number of ranks starting with z, e.g., for 8 ranks x,y,z was 512, 512, 512, for 16 ranks it was 1024,1024,2048. Application data is structured as a Conduit node and configured to publish to the SOMA service instance every five application iterations. Performance data monitoring is conducted via the SOMA TAU plugin which converts TAU profiles to conduit nodes for the SOMA RPC calls. Publishing frequency is varied between every 5 iterations and every 50 iterations for the performance data. The communication protocol used for publishing SOMA data between nodes used was ofi+tcp — we are investigating other high-performance communication protocols for Mochi on LUMI-G.

We launch the Astaroth jobs in the same configurations

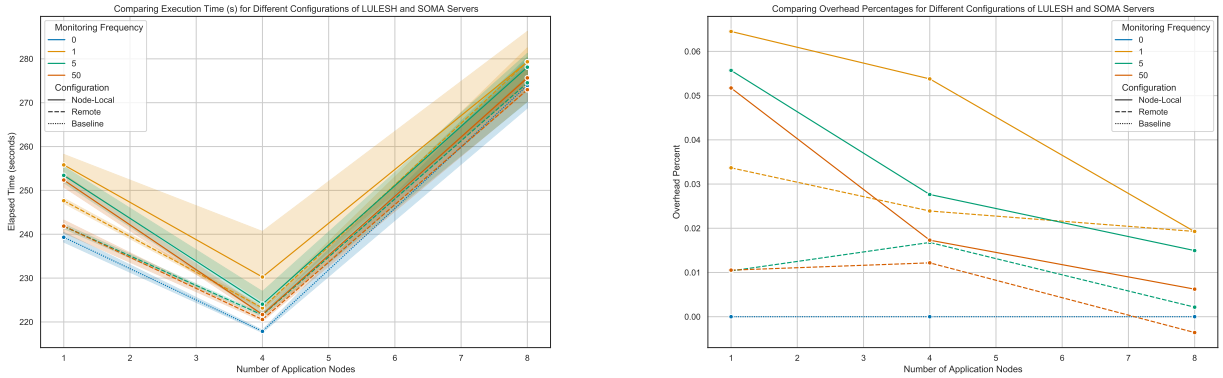

Fig. 10. Execution time (Left) and averaged overhead percent (Right) based on the configuration of our monitoring service with LULESH on Mahti. The baseline is indicated by the blue dotted line. In the case of the remote configuration, we run the indicated number of application nodes and an extra node solely for the SOMA servers. In cases where we publish more frequently, and run the SOMA servers on the same node we see increased overhead. The dashed lines demonstrate our performance when publishing data in the remote node configuration, which tend to outperforming the node-local solid lines.

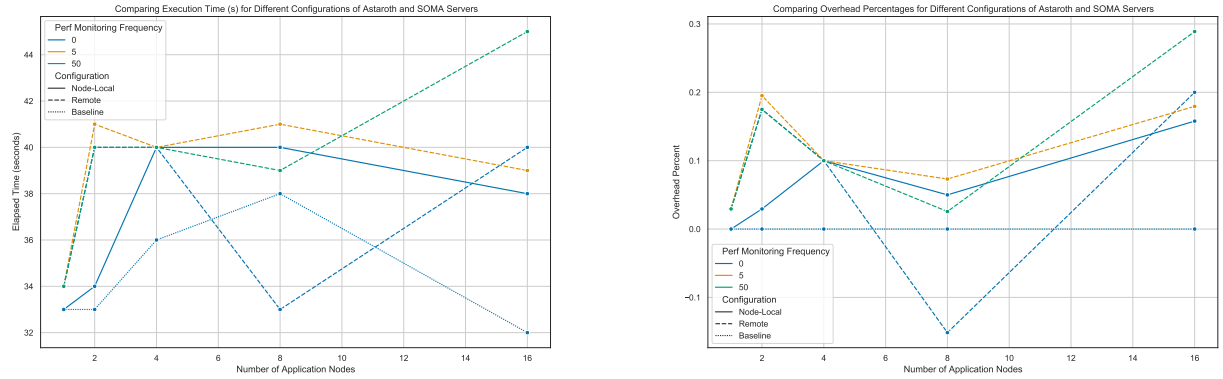

Fig. 11. Execution time (Left) and percent overhead (Right) based on the configuration of our monitoring service for Astaroth on LUMI-G. In the case of the remote configuration, we run the indicated number of application nodes, and an extra node solely for the SOMA servers. We ran a baseline with no monitoring, application only monitoring, and performance + application monitoring. We see only a small increase in overhead when we combine performance monitoring with the application monitoring (Perf Monitoring Frequency = 5,50).

as the baseline, but with SOMA ranks on either an extra ranks on an extra node, or extra ranks on each node already allocated. These ranks are reserved for the SOMA service instances, whether that be the application data or performance data service. On this additional node, we use the number of ranks equal to the number of Astaroth nodes, giving us an 8:1 ratio of Astaroth ranks to each type of SOMA service rank. This allows us to compare the results with the node-local version. For the node-local version, we run a SOMA service instance with an extra rank on each Astaroth node, also at an 8:1 process count ratio.

We can see in Figure 11 that for both configurations, there is a non-negligible amount of monitoring overhead. One potential reason for this is due to the use of the lower-performance ofi+tcp interconnect. Based on the significantly lower overheads we saw with the ofi+verbs network in our Mahti/LULESH experiments we anticipate significant performance improvement when we use the high-performance Cray network (ofi+cxi) on LUMI-G. On a positive note, we see very little increase in overhead when we conduct the performance data monitoring simultaneously with the application data monitoring, even at more frequent publication rates. Although these results may seem a bit underwhelming, we are early in our design, and there are many opportunities for optimization, beyond the network changes, that we believe will have a significant impact. For example, we are further scrutinizing the size of the data being sent, the serialization protocol used for the data, the frequency of publication, and the ratios of SOMA server instances per application rank, node count, and problem size. Analysis of the application telemetry data that we collect for Astaroth in these experiments with SOMA is described in the following section.

B. Application Diagnostics and Data Analytics

SOMA collects data from Astaroth live during a simulation. At the moment, for this proof-of-concept, we simply write the application data stream to a file at the end of the simulation,

which we postprocess using Astaroth's analysis tools. While the analysis in the proof-of-concept is not done live, there is no fundamental reason why these analysis tools cannot be attached to the data stream while the simulation is running. Live analysis would allow us to pause/stop the simulation or in the most ideal case to adjust the simulation parameters and carry on the integration with numerically safe parameters, if signs of numerical instability would be detected.

As discussed in section II, an Astaroth simulation can fail due to numerical instabilities, which result from the simulation encountering an extreme state, for which the chosen simulation parameters do no longer guarantee numerical stability. Typical cases are the viscosities being too small or the time integration step being too high to resolve the plasma flow and its evolution. These are the two example cases investigated here. As the most typical sign of an approaching numerical instability is the loss/gain of mass, which should be a conserved quantity in a healthy simulation, we will use this quantity for our proof-of-concept monitoring cases.

In the case of a healthy simulation, the mass as a whole and the isotropic turbulent fluid in each rank remains constant. Mass diagnostics collected from different ranks from such a simulation is shown in the leftmost panel of Fig. 12, and the evolution of density over time in each rank in the leftmost panel of Fig. 13, both showing healthy statistics with constant mass and nearly constant density extrema amongst all ranks.

```
{
           pid: uint32,
      timestep: uint64,
simulation_time: fp64,
    local_mass: fp64,
       FIELD_1: {
        min: { value: fp32, location: [uint16] x 3 },
        max: { value: fp32, location: [uint16] x 3 },
        nan: { value: bool, location: [uint16] x 3 }
       },
       FIELD_2: { min: ..., max: ..., nan: ... },
       ...
}
```

Listing 2. Conduit pseudoschema of the Astaroth diagnostics data model. The pid and timestep uniquely identify a node in a data stream. The simulation time and local mass provide

Next we demonstrate a case, where the integration over time, which is in this case performed with 3rd order Runge-Kutta scheme, is done too inaccurately. The simulation does use the Courant-Friedrichs-Levy (CFL) condition for calculating the maximum allowed time step length during each iteration, but in addition different physical setups require a safety prefactor around 0.1 ... 0.9 for stability. The required value of the prefactor is not known a priori for different types of physical setups, and hence it often happens that the user gives too high a value for it in the simulation parameter setup. Mass diagnostics collected from such a simulation are shown in Fig. 12 middle panel. We see that in such a case the mass is systematically lost. The mass loss is global, but happens isotropically in each rank, giving a clear imprint of the simulation parameters being globally wrong instead of indicating numerical instability developing due to an extreme condition localized to some rank/simulated region. Also the middle panel of Fig. 13 consistently agrees with this picture, the density extrema showing linearly decreasing trend with the same slope.

Next we will study a case, where the viscosity is set to a value that turns out to be too small to guarantee numerical stability. In this case, some parts of the flow are well resolved, while more extreme conditions happen only as local fluctuations in the turbulent fluid. These extreme conditions cause a numerical instability on one individual rank, in this case rank 14, the instability grows rapidly, and causes the entire simulation to crash. In the mass distribution plot Fig. 12 rightmost panel the local rapid increase of mass in rank 14 is clearly visible.

Even more information can be retrieved by plotting the density evolution in each rank, Fig. 13 rightmost panel. From there one can see that the explosion of mass is clearly preceeded by abnormal behavior of the minimum density on Rank 14. It starts diminishing abnormally fast in comparison to the other ranks already 1500 timesteps before the actual crash happens. This gives us a clear hint of the nature of the numerical instability. In the location of the extreme conditions in the turbulent flow some mass is actually lost, pressure in the location is decreased, and that causes a rapid inflow of plasma towards that location. This results in the break-down of the numerical scheme and the seen mass explosion.

VII. Related Work

There are a wealth of robust performance measurement and analysis tools that have been developed for HPC systems and applications. These include HPCToolkit [2], Score-P [23], Scalasca [45], Extrae/Paraver [33], Caliper [6], Timemory [26], and others, as well as machine-specifc vendor offerings. For the most part, these tools were designed for offline performance analysis and tuning, with a focus on first-person performance measurement of tied task functions on a per-thread OS thread basis. In addition to capturing time and hardware counter data, some of the tools also support heterogeneous systems and are able to measure GPU performance. TAU provides a comprehensive set of performance measurement and analysis capabilities that covers practically all HPC environments and parallel computing models.

In contrast, there are fewer performance tools that address existing/emerging programming models and runtime systems where there exists untied task execution and migration, runtime thread control and execution, third-person observation, and runtime performance tuning. What also sets APEX apart from runtime-specific solutions is that it has been refactored from its HPX-centric design [19] to a more general purpose asynchronous multi-tasking runtime profiling library. TAU and APEX form a powerful combination (as the TAU Performance System) for HPC application performance analysis and engineering that is not replicated in other performance toolkits.

Concurrent with the significant research and development work in the performance tools community, there has been long-term interest in parallel performance monitoring. System-level

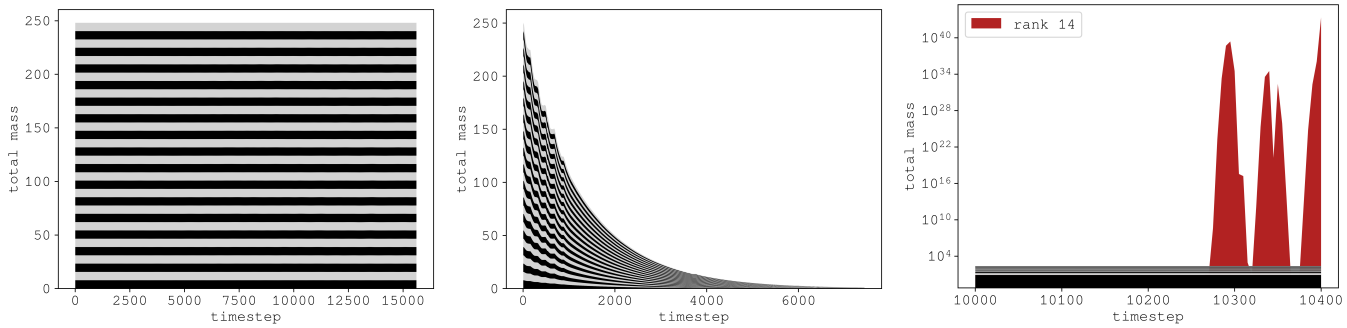

Fig. 12. Time series of the mass distribution from different types of simulation setups: healthy (left), too large time step prefactor (middle) and too small viscosity (right). Each shaded region in the stack plot is the local mass in a rank over the runtime of the simulation. In a healthy simulation the total mass in the system remains constant to the accuracy of the spatial discretization scheme. Note: in the rightmost series, for a better visualization of the instability, the y-axis is on a log scale and the x-axis starts from 10000 .

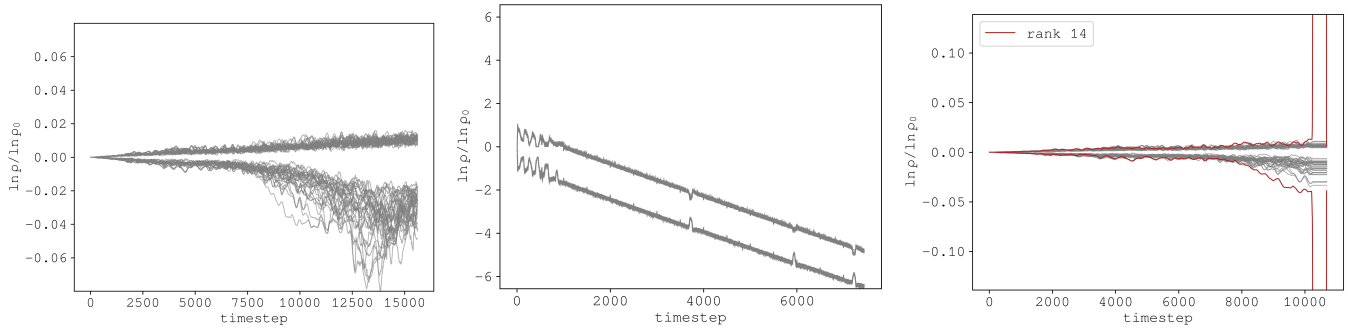

Fig. 13. Time series of the density range from different types of simulation setups: healthy (left), too large a time step prefactor (middle) and too small viscosity (right). Each gray curve is a minimum or maximum of one rank's density field over the runtime of the simulation.

monitors like LDMS [3] provide extensive online measurement of machine infrastructure and OS activities. Tools like Falcon [15], Autopilot [1], Periscope [12], ActiveHarmony [42], and WOWMON [47] utilized monitoring to provide online analysis and support for adapting and steering the application. Chimbuko [22], utilizing TAU as a performance measurement system, implemented in situ trace analysis to detect performance anomalies and generate provenance for root cause analysis.

Our own monitorng work has focused on the architecture design and high-performance implementation, specifically to access TAU performance data. TAUoverSupermon [29], TAUoverMRNet [28], TAUmon [25], and SOS [46] explored different approaches and technologies for scalable application-level monitoring. Our recent research on SYMBIOMON [36] instead chose to build upon the Mochi high-performance microservices framework [37] thereby adopting a existing development model with well-defined interfaces, available components, and active users. Seer [14], SERVIZ [34], Colza [10], and other projects have taken this route to build in situ analysis, visualization, and autotuning.

The SOMA research here continues in this direction with the additional incorporation of data models which are necessary for the semantic communication of application and performance data between microservices. Conduit [16] is an established project born out of the visualization community for the purpose of describing and sharing data in situ.

Harvesting unused computing cycles has been explored in the context of inline visualization and analytics. The TINS [8] package leverages work-stealing strategies to execute analytics tasks when there are no available simulation tasks scheduled. GoldRush [48] and Landrush [13] employ smart co-scheduling of analytics routines alongside MPI-OpenMP and GPU simulation tasks. They combine monitoring data with a scheduler to identify regions of idle time on the processor that can be used to run these routines demonstrating significant cost savings without perturbing the execution of the simulation. Our approach is not nearly as sophisticated as these research results, but nevertheless attempts to take advantage of a situation with free CPU resources via a straightforward MPI-based strategy.

VIII. Conclusion and Future Work

Our research work presented in this paper was in two main directions. First, we integrated the TAU and APEX tools with Astaroth to provide a robust, portable, and configurable environment for performance analysis and engineering. These tools work on heterogeneous HPC platforms, across different processor architectures, and at scale. Astaroth did not previously employ such powerful performance technology and the integration makes possible detailed analysis and optimization

going forward. Importantly, it helps to address the challenges that come with measurement of leading-edge CPU and GPU technologies present in supercomputer systems like LUMI. The key to success in this endeavor is for the Astaroth team to work with the features the tools offer. Instrumenting application-specific events with PerfStubs is a good, straightforward example of how developers can add semantic value to the tools. Furthermore, the Astaroth team should be empowered to develop more substantial performance engineering features built around the TAU Performance System ecosystem, including parametric performance studies, cross-architecture performance characterization, performance regresssion testing, and continuous integration. The TAUdb performance database and other performance analysis are relevant to this objective.

The second research direction we pursued was the design and development of a microservices-based monitoring framework called SOMA and its deployment with HPC applications. Based on the robust Mochi infrastructure, SOMA is a new monitoring system that is targeting in situ performance and application observation, data collection, and analysis. We described the architecture and functionality of our SOMA prototype and demonstrated its operation with the well-known LULESH benchmark application. The high degree of configurability possible with SOMA allows it to be flexibly deployed to address in situ objectives. In the case of Astaroth, the opportunity exists to utilize idle CPU cores for SOMA operations. We demonstrated different SOMA configurations with Astaroth to show the collection of application and data diagnostics. Another unique research contribution was the design of a performance data model for TAU to enable data representation and sharing, in this case, between SOMA client/server processes. An application data model was also created for in situ transmission and processing within SOMA. Again, it was important to engage with the Astaroth team to define the application data of interest. The Conduit [16] technology was used to implement both data models. In this proof-of-concept study, the data harvested was tailored to monitor the numerical stability of the code, with great future potential to optimize the simulation workflow. Now that SOMA functionality has been validated with Astaroth, we hope to find optimal configurations for minimal overhead, test its performance in larger-scale scenarios, and build on its capabilities for specific Astaroth purposes. It is technically possible to build support in SOMA that allows feedback to an application and Astaroth could take advantage of this for runtime adaptation.

IX. Acknowledgements

The research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration), as well as the Scientific Discovery through Advanced Computing (SciDAC) program funded by DOE, Office of Science, Advanced Scientific Computing Research (ASCR) under contract DE-SC0021299. This project has also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Project UniSDyn, grant agreement no: 818665).

The simulations and experiments for this publication were run on the EuroHPC "LUMI" supercomputer in Kajaani, Finland and the "Mahti" supercomputer at the CSC Centre for Scientific Computing.

X. Code availability

TAU and APEX are open source under a BSD-style license and available at https://tau.uoregon.edu and https://uo-oaciss.github.io/apex.

References

[1] The autopilot performance-directed adaptive control system. *Future Generation Computer Systems*, 18(1):175–187, 2001.

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

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

[4] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, chapter 26, pages 359–371. Morgan Kaufmann, Boston, 2012.

[5] David Boehme, Kevin Huck, Jonathan Madsen, and Josef Weidendorfer. The case for a common instrumentation interface for HPC codes. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, pages 33–39, 2019.

[6] David Böhme, Todd Gamblin, D. A. Beckingsale, Peer-Timo Bremer, Alfredo Giménez, Matthew P. Legendre, Olga Pearce, and Martin Schulz. Caliper: Performance introspection for HPC software stacks. *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 550–560, 2016.

[7] Leonardo Dagum and Ram Menon. OpenMP: An industry standard API for shared-memory programming. 1998.

[8] Estelle Dirand, Laurent Colombet, and Bruno Raffin. Tins: A task-based dynamic helper core strategy for in situ analytics. In *Asian Conference on Supercomputing Frontiers*, pages 159–178. Springer, 2018.

[9] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. Top500 supercomputer sites. *Supercomputer*, 13:89–111, 1997.

[10] Matthieu Dorier, Zhe Wang, Utkarsh Ayachit, Shane Snyder, Rob Ross, and Manish Parashar. Colza: Enabling elastic in situ visualization for high-performance computing simulations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 538–548. IEEE, 2022.

[11] MPI Forum. Mpi specifications, 2023.

[12] Michael Gerndt, Karl Fürlinger, and Edmond Kereku. Periscope: Advanced techniques for performance analysis. In *International Conference on Parallel Computing (ParCo)*, pages 15–26, September 2005.

[13] A. Goswami, Y. Tian, K. Schwan, F. Zheng, J. Young, M. Wolf, G. Eisenhauer, and S. Klasky. Landrush: Rethinking in-situ analysis for GPGPU workflows. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 32–41, 2016.

[14] Pascal Grosset, Jesus Pulido, and James Ahrens. Personalized in situ steering for analysis and visualization. In *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 1–6. 2020.

[15] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 422–429, 1995.

[16] Cyrus Harrison, Matt Larsen, Brian Ryujin, Adam Kunen, Arlie Capps, Justin Privitera, et al. Conduit: A successful strategy for describing and sharing data in situ. Presented at SC22 ISAV Workshop, 2022.

[17] Kevin A. Huck, Sameer Shende, Allen D. Malony, Hartmut Kaiser, Allan Porterfield, Robert J. Fowler, and Ron Brightwell. An early prototype of an autonomic performance environment for Exascale. In *International Workshop on Runtime and Operating Systems for Supercomputers*, 2013.

[18] Advanced Micro Devices Inc. HIP API documentation, 2022.

[19] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 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*, PGAS '14, pages 6:1–6:11. ACM, 2014.

[20] Ian Karlin. LULESH programming model and performance ports overview. 2012.

[21] Ian Karlin, Jeff Keasler, and J. Robert Neely. LULESH 2.0 updates and changes. 2013.

[22] Christopher Kelly, Sungsoo Ha, Kevin Huck, Hubertus Van Dam, Line Pouchard, Gyorgy Matyasfalvi, Li Tang, Nicholas D'Imperio, Wei Xu, Shinjae Yoo, et al. Chimbuko: A workflow-level scalable performance trace analysis tool. In *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 15–19. 2020.

[23] Andreas Knüpfer, Christian Rössel, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E Nagel, et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

[24] Oskar Lappi. A task scheduler for astaroth, the astrophysics simulation framework, 2021. Masters Thesis.

[25] C.W. Lee, A. Malony, and A. Morris. TAUmon: Scalable Online Performance Data Analysis in TAU. In *Workshop on Productivity and Performance (PROPER 2010)*, September 2010.

[26] Jonathan R Madsen, Muaaz G Awan, Hugo Brunie, Jack Deslippe, Rahul Gayatri, Leonid Oliker, Yunsong Wang, Charlene Yang, and Samuel Williams. Timemory: modular performance analysis for hpc. In *International Conference on High Performance Computing*, pages 434–452. Springer, 2020.

[27] Allen D. Malony, Sameer Shende, Wyatt Spear, Chee Wai Lee, and Scott Biersdorff. Advances in the tau performance system. In *Parallel Tools Workshop*, 2011.

[28] A. Nataraj, A. Malony, A. Morris, D. Arnold, and B. Miller. TAUoverMRNet (ToM): A Framework for Scalable Parallel Performance Monitoring. In *International Workshop on Scalable Tools for High-End Computing (STHEC '08)*, 2008.

[29] Aroon Nataraj, Matthew Sottile, Alan Morris, Allen D Malony, and Sameer Shende. Tauoversupermon: low-overhead online parallel performance monitoring. In *Euro-Par 2007 Parallel Processing: 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007. Proceedings 13*, pages 85–96. Springer, 2007.

[30] John R. Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 1–2, 2008.

[31] OpenMP. Openmp specifications, 2023.

[32] Johannes Pekkilä, Miikka S. Väisälä, Maarit J. Käpylä, Matthias Rheinhardt, and Oskar Lappi. Scalable communication for high-order stencil computations using CUDA-aware MPI. *Parallel Computing*, 111:102904, 2022.

[33] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 17–31. mar, 1995.

[34] Srinivasan Ramesh, Hank Childs, and Allen Malony. Serviz: A shared in situ visualization service. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*, pages 277–290, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.

[35] Srinivasan Ramesh, Robert Ross, Matthieu Dorier, Allen Malony, Philip Carns, and Kevin Huck. SYMBIOMON: A high-performance, composable monitoring service. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 332–342, 2021.

[36] Srinivasan Ramesh, Robert Ross, Matthieu Dorier, Allen Malony, Philip Carns, and Kevin Huck. Symbiomon: A high-performance, composable monitoring service. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 332–342. IEEE, 2021.

[37] Robert B. Ross, George Amvrosiadis, Philip H. Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Gregory R. Ganger, Garth A. Gibson, Samuel Keith Gutierrez, Robert Latham, Robert W. Robey, Dana Robinson, Bradley W. Settlemyer, Galen M. Shipman, Shane Snyder, Jérome Soumagne, and Qing Zheng. Mochi: Composing data services for high-performance computing environments. *Journal of Computer Science and Technology*, 35:121–144, 2020.

[38] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, et al. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2017.

[39] S. Shende and A. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[40] Marc Snir, Steve W. Otto, David W. Walker, Jack J. Dongarra, and Steven Huss-Lederman. MPI: The complete reference. 1996.

[41] Jerome Soumagne, Dries Kimpe, Judicael A Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert B Ross. Mercury: Enabling remote procedure call for high-performance computing. In *CLUSTER*, pages 1–8, 2013.

[42] C. Tapus, I-H. Chung, and J. Hollingsworth. Active harmony: Towards automated performance tuning. In *ACM/IEEE Conference on Supercomputing*, pages 1–11, 2002.

[43] Miikka S. Väisälä, Johannes Pekkilä, Maarit J. Käpylä, Matthias Rheinhardt, Hsien Shang, and Ruben Krasnopolsky. Interaction of Large- and Small-scale Dynamos in Isotropic Turbulent Flows from GPU-accelerated Simulations. *The Astrophysical Journal*, 907(2):83, February 2021.

[44] Jorn Warnecke, Maarit J. Korpi-Lagg, Frederick A.Gent, and Matthias Rheinhardt. Numerical evidence for a small-scale dynamo approaching solar magnetic Prandtl numbers. *Research Square*, 2022.

[45] F. Wolf, B. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi. Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In *Proc. of the 2nd HLRS Parallel Tools Workshop*, pages 157–167, Stuttgart, Germany, July 2008. Springer.

[46] Chad Wood, Sudhanshu Sane, Daniel Ellsworth, Alfredo Gimenez, Kevin Huck, Todd Gamblin, and Allen Malony. A scalable observation system for introspection and in situ analytics. In *2016 5th workshop on extreme-Scale Programming Tools (ESPT)*, pages 42–49. IEEE, 2016.

[47] Xuechen Zhang, Hasan Abbasi, Kevin Huck, and Allen D Malony. Wowmon: A machine learning-based profiler for self-adaptive instrumentation of scientific workflows. *Procedia Computer Science*, 80:1507–1518, 2016.

[48] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference-aware execution. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.