Preparing the TAU Performance System for Exascale and Beyond

Journal Title
XX(X):1–16

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Kevin A. Huck[1] and Sameer Shende[1] and Allen D. Malony[1] and Camille Coti[2] and Wyatt Spear[1] and Jordi Alcaraz[1] and Dewi Yokelson[1] and Srinivasan Ramesh[5] and Monil Mohammad Alaul Haque[4] and Chad Wood[7] and Nick Chaimov[3] and Cameron Durbin[3] and Alister Johnson[1] and Jacob Lambert[6] and Izaak Beekman[3]

Abstract

The TAU Performance System® is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, UPC, Java, Python. TAU (Tuning and Analysis Utilities) is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling. All C++ language features are supported including templates and namespaces. The API also provides selection of profiling groups for organizing and controlling instrumentation. The instrumentation can be inserted in the source code using an automatic instrumentation tool based on the Program Database Toolkit (PDT), dynamically using binary modification, at runtime in the Java Virtual Machine, or manually using the instrumentation API.

Under the Exascale Computing Program (ECP), the TAU project was funded to prepare the software for exascale systems and beyond. Many new features and optimizations were added to TAU, including support for the new exascale system architectures and their preferred programming models. The new features include OpenMP Tools support, updated or newly implemented CUDA, HIP, and SYCL support, updated OpenACC and Clacc support, MPI updates, a new plugin API and several plugins, instrumentation updates, support for the Kokkos and Raja profiling interfaces, updated support for Python, PyTorch, TensorFlow, and Horovod, and removed threading limitations. In this paper, we will discuss these updates and more, and demonstrate the features with ECP Proxy Applications and full ECP applications.

Keywords

exascale computing, high performance computing, parallel computing, performance measurement tools, performance analysis

Introduction

Throughout the history of parallel computing, there has been a need to measure, analyze, and understand the performance of next-generation computer systems and the applications that run on them. However, designing, developing, and implementing parallel performance tools is a challenging research enterprise. Continuous advances in high-performance computing (HPC) architecture, hardware, system software, and programming environments challenge performance tool methodologies and technologies to keep pace with the ever-growing complexity of the parallel execution and the performance problems that can arise. The importance of heterogeneous parallelism to achieve extreme-scale performance, coupled with performance portable parallel programming systems, places even further demands on performance tools to be integrated seamlessly and ubiquitously in all aspects of the HPC ecosystem. Such was the case with the US Department of Energy (DOE) *Exascale Computing Program* (ECP)[2] and the *TAU Performance System®*[3].

The TAU project at the University of Oregon was funded to prepare the TAU (Tuning and Analysis Utilities) technology for exascale systems and beyond. TAU maintenance and development for these new systems, libraries, and applications was directly provided by the PROTEAS-TUNE Software Technology sub-project[4]. In addition to TAU development, the TAU team was also engaged with several ECP Application sub-projects[5] and Software Technology sub-projects[6] where TAU was being used and integrated. The Application sub-projects included WDMApp, CODAR, CANDLE, NWChemEx, ExaLearn, CoPA, ExaWind, and Combustion-PELE. The Software Technology sub-projects included ADIOS, Alpine, Exa-PAPI, SOLLVE, PETSc/TAO, SUNDIALS/Hypre, Exascale MPI/MPICH, UPC++/GASNet, Kokkos, Argo, and E4S. In the course of our activities, many new features and optimizations were added to TAU, including support for the delivered exascale systems and their preferred programming models. The new features include (but certainly not limited to):

[1]University of Oregon, USA
[2]École de Technologie Supérieure, Canada
[3]ParaTools, Inc., USA
[4]Oak Ridge National Laboratory, USA
[5]NVIDIA, USA
[6]AMD, USA
[7]Stability AI

Corresponding author:
Kevin A. Huck, OACISS Institute, University of Oregon, Eugene, Oregon, 97403, USA.
Email: khuck@cs.uoregon.edu

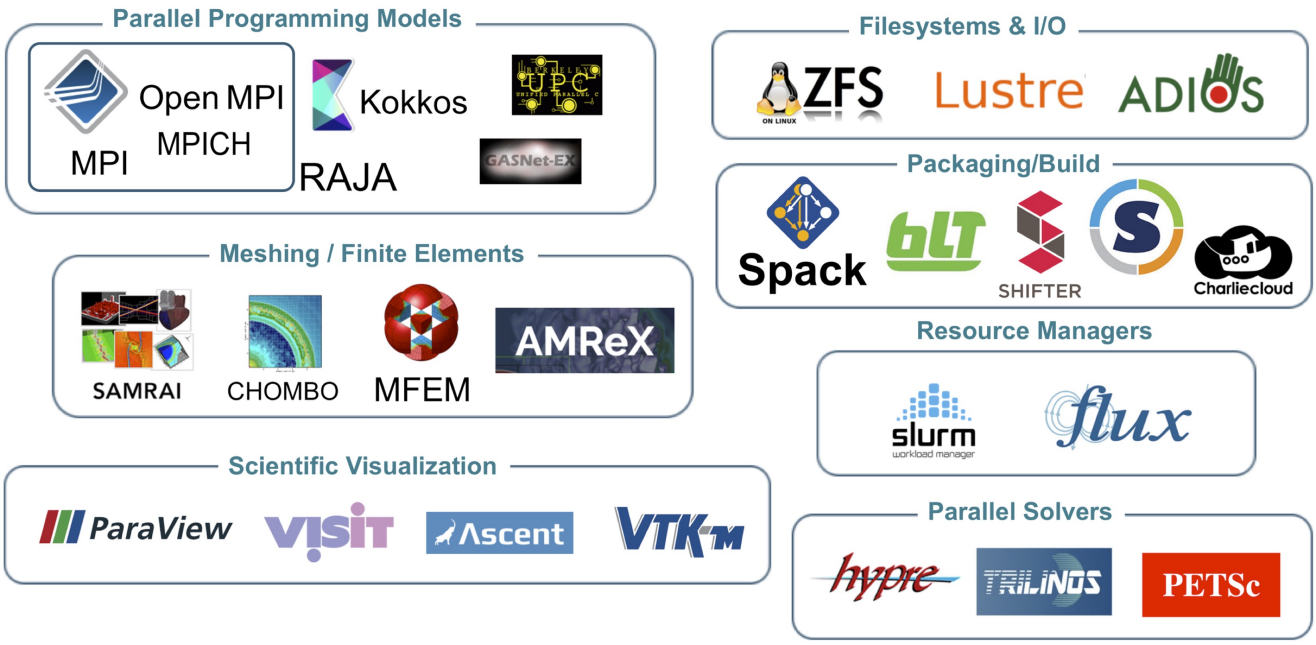

Figure 1. TAU works with all of these Open Source and ECP models, libraries and technologies (source: OSTI 1646052[1])

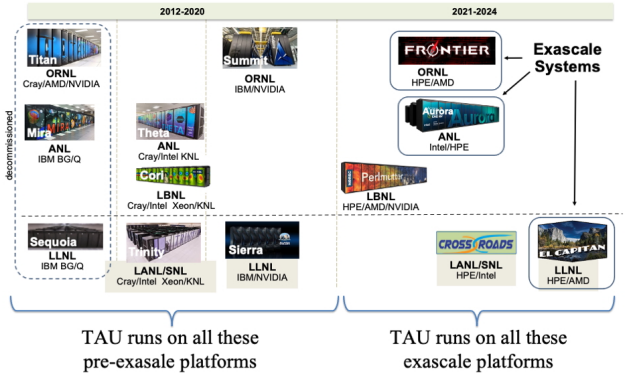

Figure 2. Due to its highly configurable and portable nature, TAU supports all existing US Department of Energy HPC systems, both pre- and post-exascale, and plans to do so for the years to come.

- OpenMP Tools support
- Updated/new CUDA, HIP, and SYCL support
- Updated OpenACC and Clacc support
- Message Passing Interface (MPI) updates
- New TAU plugin API and several plugins types
- Instrumentation updates
- Support for the Kokkos and Raja profiling interfaces
- Python, PyTorch, TensorFlow, and Horovod udpates
- Removal of threading limitations

Figure 1 shows a selection of the open source models, libraries, and container technology that have been tested or integrated with TAU.

One of the challenges of researching, developing, and maintaining software on state of the art HPC systems is that the systems are constantly evolving and are frequently replaced by new architectures. Figure 2 shows one small window into the evolution of leading machines at DOE Office of Science and National Nuclear Security Administration (NNSA) research laboratories. During the last 12 years, the systems have evolved in significant ways. The IBM BlueGene/Q[7] and Intel Knights Landing[8] systems were challenging due to dense, many-core processors. The heterogeneous systems using NVIDIA GPUs have evolved and have been complemented by exascale-class GPU accelerated systems using new AMD[9] and Intel[10] architectures. In addition, evolutionary and revolutionary advances in networking, filesystems, batch control systems, and all of the other system software have required complementary advances in performance measurement tools like TAU.

In this paper, we will discuss some of the updates to TAU, and demonstrate the features with ECP Proxy Applications and a few full ECP applications.

Background: TAU Design

The core of the TAU library is built around the seemingly simple task of observing performance events registered via API calls as an application executes. These events represent some action occurring in the computation and an associated performance measurement either at the time of the event (called an *atomic* event) or at the begin and end of the event (called an *interval* event). An atomic event could be used to capture the size of an MPI message, for instance. An interval event could be used to determine the duration of time spent between the entry and exit of a function. Events also be collected periodically, such as high-level monitoring counters that are collected once every 10 seconds, or low-level sampling support for profiling that is captured one hundred times per second.

The API calls that engage these events can be inserted into application code manually or using various automated mechanisms that TAU supports. TAU's Performance Database Toolkit (PDT) provides source code instrumentation for different languages. TAU uses code insertion features common

to modern compilers, to insert instrumentation during compilation. Different binary editing tools (e.g., DynInstAPI[11], MAQAO[12], PEBIL[13]) can be used to insert TAU API calls.

In addition to instrumentation of the target application TAU is also capable of registering performance events by library wrapping. A wrapped library engages the instrumentation API when calls to the library's functions occur. This greatly simplifies collection of performance data pertaining to functions commonly relevant to performance tuning such as MPI communications and POSIX I/O. Wrapped libraries can be preloaded at runtime for dynamic linking or injected at link time for static linking.

As we will see later in the paper, several programming models and languages (OpenMP, OpenACC, Kokkos, Python) provide built-in mechanisms for allowing tools to register with the application at startup. These models and libraries will recognize that a performance tool has been loaded into the user's environment, and will register callback functions for entry and exit into key regions or phases of the model or library. In these cases the application does not need to be instrumented to provide timing insights. These integrations can either be built as a weak symbol replacement model, or using a plugin interface.

In contrast to direct instrumentation, TAU also supports periodic sampling. This requires no modification to the application and is achieved by periodically interrupting the application with a signal, which triggers a signal handler that interrogates the instruction counter and unwinds the call stack at runtime. The precise entry and exit times of functions are not recorded by sampling but with an adequate sampling rate observations of the time functions spend on the call stack before completion are comparable in accuracy to direct instrumentation. Integration of timers and samples is also supported[14].

All forms of performance data collection require careful design and testing to ensure compatibility with the parallel programming paradigms supported by TAU. MPI implementations, threading models, a variety of rapidly developing accelerator hardware and different compilers all have specific requirements and performance interfaces that must be taken into account in TAU's measurement architecture. In order to support portable hardware counter collection, TAU is integrated with PAPI[15] and LIKWID[16], thereby providing a common interface across many different processor and integration architectures. PAPI also provides access to hardware counters from devices such as NVIDIA and AMD GPUs.

Measurements of an application's performance behavior must be stored as performance data that can be recorded and analyzed. TAU generates profile data which summarizes the time spent in instrumented or sampled blocks of code for each rank and thread of execution. Profiles also provide statistical summaries of atomic events. TAU profiles can be read and filtered as text or visualized in the ParaProf profile viewer[17]. Multiple profiles can be stored in the TauDB database[18] and analyzed in scaling, regression, or other parametric studies via the PerfExplorer profile analysis tool[19]. TAU also provides tools to convert its profile output to other formats for analysis with other tools.

Profiles provide an efficient summary of an application's performance behavior. TAU can also generate performance traces which record every performance event on a timeline for each thread of execution. Traces can be viewed in tools such as Jumpshot (included with TAU) or the Vampir trace viewer[20]. A performance trace provides a visual representation of an application's execution behavior, allowing identification of communication patterns and the specific context of performance hotspots that may be obscured by the reduced resolution of a profile.

New TAU Features Developed Under ECP

Throughout the 30+ year history of the TAU Performance System, the evolution of the HPC hardware, software, and systems technologies has resulted in new requirements for HPC performance tools. Some of the challenges presented by these requirements to TAU's next-generation development could be met by improvements or extensions of existing functionality or better integration with other tools. TAU takes pride in the fact that much of its fundamental design and core engineering have remained robust over generations and allowed a growing set of capabilities that span generations of HPC platforms. However, other challenges introduced can have a more disruptive effect whereby a re-thinking and possible re-engineering of certain TAU mechanisms are necessary. ECP resulted in both cases (incremental, disruptive) happening in TAU's advancement as we highlight in the discussion that follows. (The new TAU features are in bold.) While some features might seem more of a continuation of what TAU has already been doing, it does not imply that updating TAU to support them was trivial. It is also the case that brand new features are designed to be integrated in TAU's overall architecture and not only for immediate release.

OpenMP Tools Support

OpenMP is a Fortran, C, and C++ language extension that provides portable parallel programming support for shared memory and heterogeneous computing resources. The specification includes source code annotation (pragmas) that are transformed by the compiler to generate API calls into an OpenMP runtime provided by the compiler. OpenMP provides a short runway for application and library developers to each provide portable multi-threaded parallelism, but at the cost of an opaque runtime implementation. This opacity was a challenge when parallel efficiencies are less than expected and users are unable to get insight into how the original source code was transformed and how they might eliminate bottlenecks.

In May 2013, the first Technical Report (TR) related to the OpenMP Tools standard was published by the OpenMP standards committee. Since then, the API specification for both OpenMP runtimes and tools has evolved to include the first integrated and accepted standard of OpenMP-Tools (OMPT) in the OpenMP 5.0 standard released in November, 2018. The beauty of the tool specification is that it provides tool access to the internal workings of what had previously been a *black box* OpenMP runtime. Prior to OMPT, TAU could only provide insight by instrumenting the source code with the OPARI API[21], a semi-automated and sometimes error-prone approach. **Since the early days of the first TR, TAU has evolved to both evaluate prototype**

implementations of the specification as well as provide support to utilize it when the first compilers provided a full 5.0 compliant implementation.

TAU provides support for measuring all of the OpenMP callback types, including parallel regions, worksharing regions, teams, threads, tasks, synchronization, and locks. With this support, TAU can provide the application developer insight into how OpenMP pragmas are transformed by the compiler into multi-threaded or heterogeneous computing code without adding additional instrumentation.

OpenMP Target Offload. In addition to CPU events, TAU can utilize the OMPT runtime support to collect performance data related to OpenMP target offload events, where computational kernels are offloaded to processing devices such as graphical processing units (GPUs).

Figure 3 shows an example trace collected of the miniQMC proxy application[22] executed on the HPE Cray EX Frontier supercomputer[23]. In the example shown, TAU collected an OTF2 trace[24] of the application, and the OMPT support provided measurement of not just the CPU events on the POSIX threads, but also the GPU memory transfer, synchronization, and kernel events.

OpenACC and Clacc Support

OpenACC (for *Open Accelerators*) is a parallel programming standard specifically targeting accelerators and programming heterogeneous systems providing GPUs and multicore CPUs. The standard is published by the *OpenACC Organization*, led by industrial partners and hardware vendors such as Cray, HPE, Nvidia and PGI. It provides an interface specification in C, C++ and Fortran. Similar to OpenMP, OpenACC provides pragmas that are transformed by the compiler to runtime API calls into an OpenACC runtime library.

Whereas OpenMP follows a *prescriptive* model, OpenACC follows a *descriptive* model: the programmer describes where parallelism can be extracted and which data needs to be moved to and from the accelerator, and the compiler generates the corresponding parallel loops and data movements. Therefore, OpenACC programs rely more on the compiler. In addition to loops, OpenACC provides the `kernel` directive that indicates that this region can be parallelized. The compiler needs to analyze data dependencies in the region, identify parallelism that can be extracted and data that needs to be transferred, and generate the corresponding parallel loops and data movements.

A couple of open source and commercial compilers implement the OpenACC specification. The Clacc compiler[25] supports OpenACC using LLVM's existing OpenMP compiler and runtime support. The OpenACC code is parsed, a corresponding (OpenACC) Abstract Syntax Tree (AST) is generated, and this AST is translated into an OpenMP AST. At this point, Clacc offers two options: either this OpenMP AST is translated into LLVM IR and an executable is generated, or from this OpenMP AST, an OpenMP source code is generated and compiled by the OpenMP compiler.

The resulting executable is executed using LLVM's OpenMP runtime environment. Both OpenACC and OpenMP provide a callback-based profiling interface: *callback functions* are called at specific points of the program, such as task creation and before and after data transfers. In Clacc, the application triggers OpenMP events. **An OpenACC runtime environment translates these events into OpenACC events[26], which are intercepted by TAU**.

With commercial compilers such as NVHPC, the OpenACC runtime adheres strictly to the OpenACC Profiling API. OpenACC's profiling interface uses callbacks, like the OMPT interface. However, unlike the OMPT interface, all the callback routines of the OpenACC profiling interface have the same prototype (OpenMP has 40+ different callback function signatures):

Listing 1: Prototype of the OpenACC callback routines.

```
typedef void (*acc_callback)
  (acc_callback_info*, acc_event_info*,
  acc_api_info*);
typedef acc_callback acc_prof_callback;
```

The first argument, of type `acc_prof_info` contains the event type in its first field of type `acc_event_t`. As a consequence, TAU implements a single callback routine and its behavior is determined based on the event type.

For instance, when the run-time environment enters a parallel construct, it triggers the `acc_ev_compute_construct_start` callback. Any function that respects the prototype can be registered on this callback; it can be a specific function, or the same as for other callbacks. This function interrogates the `event_type` field to determine which event was triggered:

Listing 2: Registration of a generic callback routine.

```
void acc_register_library( acc_prof_reg reg,
                  acc_prof_reg unreg,
                  acc_prof_lookup lookup ) {
  /* ... */
  reg( acc_ev_compute_construct_start,
       &Tau_openacc_callback, acc_reg );
  reg( acc_ev_compute_construct_end,
       &Tau_openacc_callback, acc_reg );
  /* ... */
}
Tau_openacc_callback( acc_prof_info* prof_info,
                  acc_event_info* event_info,
                  acc_api_info* api_info ) {
  switch (prof_info->event_type) {
    case acc_ev_compute_construct_start:
      /* ... */
      break;
    case acc_ev_compute_construct_end:
      /* .... */
      break;
    /* .... */
  }
  /* ... */
}
```

OpenACC's profiling interface was introduced in the 2.5 version of its specification. Similarly to OpenMP's tooling interface, the OpenACC profiling interface gives access to internals of the runtime system: for instance, the `acc_ev_enqueue_launch_start` event is triggered just before an accelerator computation is enqueued for execution on a device, and `acc_ev_enqueue_launch_end` is triggered just after the computation is enqueued. **TAU**

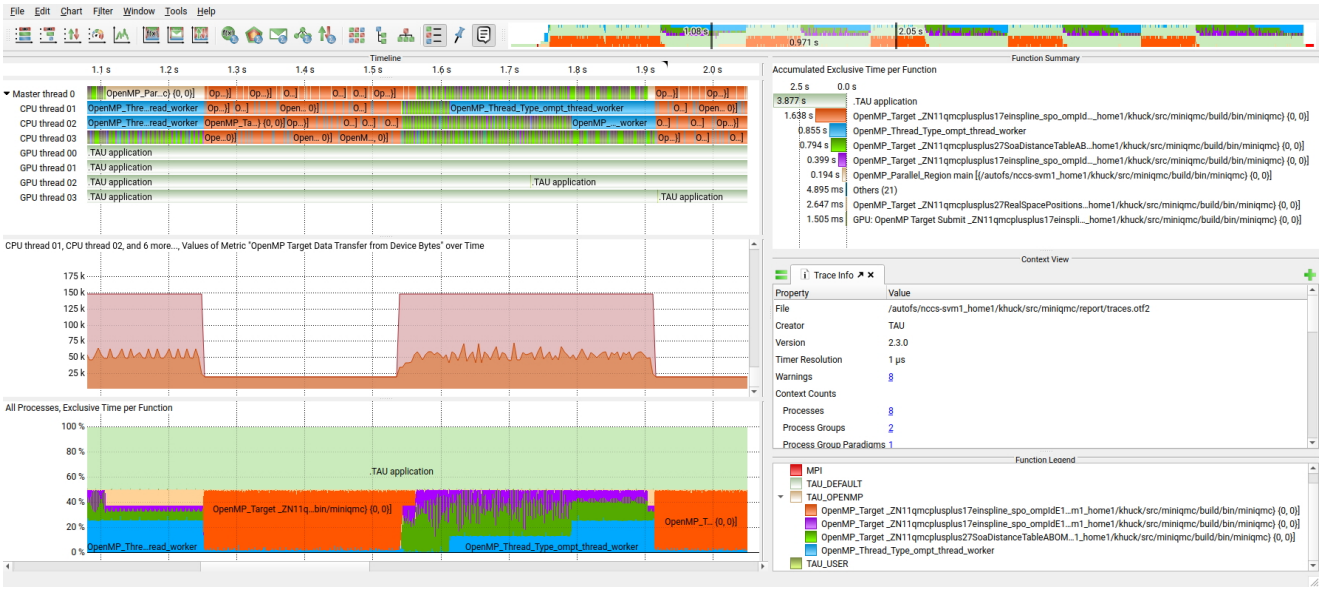

Figure 3. TAU was used to collect profiles and traces of OpenMP target offload benchmarks (miniQMC on Frontier shown), observing OpenMP regions and device offload events without application instrumentation

has implemented full support for both the OpenACC profiling interface and the profiling technique used by Clacc. The initial implementation supported the PGI compiler starting in 2015, and has subsequently been tested with the Clacc and NVHPC compilers on exascale systems.

GPU Measurement Support

NVIDIA CUDA. CUDA is a parallel compute platform for general purpose computing with NVIDIA GPUs[27]. The CUDA Toolkit is available for C, C++, and Fortran with the compilers shipped with the toolkit. Additionally, there is support for other popular languages, such as Python (PyCUDA), and multiple libraries have been optimized and implemented directly on top of CUDA, for instance: BLAS, FFT, SPARSE, and SOLVER.

CUDA does not require directives nor code annotations (pragmas). However, the developer needs to include CUDA API calls into the application for the GPU memory allocations, data transfers between GPU and host (or use unified memory if available) and design how the offloaded code should be executed in the underlying GPU architecture. The CUDA code is then compiled by the provided compilers (or compiler wrappers).

The toolkit also includes its own performance analysis tools and an API called CUDA Profiling Tools Interface (CUPTI)[28] that can be integrated into performance analysis tools to profile applications offloading work to NVIDIA GPUs. CUPTI reports performance data related to the execution of kernels, memory transfers between GPU and CPU, memory allocations, different CUDA API calls and also NVIDIA Tools Extension (NVTX) events[29]. NVTX provides a library to insert annotations into programs and there are two types of annotations included in this library: a) markers: markers insert a message at a certain point of the code. Multiple markers can be included in an application and each marker can have a different category, color and payload values; and b) ranges: between two points of the application two markers can be inserted to establish where a certain event occurs in the code and obtain profiling information of one or multiple ranges.

TAU began providing CUPTI support in 2010. **As the NVIDIA technology has evolved, TAU has evolved along with it to provide performance measurement support for kernel execution, memory transfers, NVTX events, and OpenMP/OpenACC kernel execution**. TAU provides trace support for the CUDA runtime and driver APIs, which allows for both high and low level measurement, respectively. In addition, **TAU integrates with the PAPI[30] portable hardware counter library to provide access to thousands of low level hardware counters available on the latest NVIDIA hardware**. For monitoring support, **TAU now integrates the NVIDIA Monitoring Library (NVML)[31] to provide periodic interrogation of GPU counters indicating utilization, ECC error counts, power management, the process being executed, temperatures, fan speed and clock frequencies**.

AMD ROCm/HIP. As mentioned in the introduction, the first exascale system deployed in the US is the Frontier system at Oak Ridge National Laboratory. Frontier is designed around AMD processors and CDNA™2[32], a new architecture of GPU accelerator also designed by AMD.

ROCm is AMD's software stack for GPU programming, it has support for both NVIDIA and AMD GPUs. There are three parallel programming models in the ROCm software stack: HIP, OpenMP and OpenCL (Open Computing Language – broadly supported in TAU since 2010, shortly after the initial release of the standard). HIP is the new programming model developed by AMD to write C++ kernels for the two main GPU vendors. Additionally, ROCm includes a tool called *HIPIFY*, which is used to automatically convert CUDA code to HIP C++ code.

Multiple supercomputers are installed with AMD GPUs, such as Frontier (the first official exascale supercomputer), El Capitan (another exascale computer, which was deployed in late 2024) and other supercomputers in the TOP 500.

Therefore, as the number of computers using AMD GPUs increases, the need for performance analysis tools to improve performance of applications also increases. **Hence, we implemented the ROCm profiling interface into TAU to be able to access performance metrics of applications using the ROCm software stack**.

TAU's interface to profile OpenCL and OMPT based applications can be used with AMD GPUs as both parallel paradigms are included in the ROCm software stack. Furthermore, ROCm includes its own profiling tool, called rocprof which reports multiple metrics related to kernel execution, memory usage, memory transfers, ROCm related api calls, code annotations and additional metrics.

The API used to integrate ROCm profiling into performance analysis tools is divided into two parts, each of which operate independently. The two parts are: a) ROCprofiler, a basic profiling interface with detailed information that reports kernel execution times and parameters of the kernels, such as grid and work group size, number of barriers and register usage; b) ROCtracer, which provides limited profiling of the kernels but also reports ROCTX code annotations, HIP API calls (both runtime and driver level) and memory operations. **TAU has integrated support for both libraries**.

As having two different profiling interfaces was redundant, an initial second version of the ROCm profiler tools (called rocprofv2) was released with ROCm v6.0.0. This version incorporates both ROCprofiler and ROCtracer functionalities in one single interface, whereas in the previous version, the user had to choose between the profiler or the tracer when monitoring an application. Reflecting this change, **TAU was modified and the flag *-rocprofv2* can be used at configuration time to select the new version of the profiler**. With this change, TAU is able to obtain information about kernel execution time, time spent in memory operations, memory usage, grid and work group sizes, and other ROCm related metrics alongside the trace of HIP API calls.

AMD also provides a monitoring library, ROCm-SMI, similar to the CUDA NVML library. **TAU has integrated support for this library, and also provides access to low level GPU counters through the PAPI library**.

Intel oneAPI. The second exascale system to be deployed in the US is the Aurora system at Argonne National Laboratory. Aurora is designed around Intel Xeon processors and a new GPU accelerator architecture, X^e, also designed by Intel. The Aurora system is programmed using Intel's development system called oneAPI, Intel's approach to multi-architecture parallel programming, with support for CPU, GPU (there is support for Intel, AMD and NVIDIA), FPGA and other accelerators. oneAPI consists of DPC++, C++/C, and Fortran compilers including OpenMP target offload and OpenCL support for all languages.

The Intel DPC++ compiler provides support for the SYCL programming model, which is the recommended software stack for GPU programming on Intel X^e systems like Aurora. On Intel systems, the SYCL support is layered on top of Level Zero[33]. The objective of the oneAPI Level Zero API is to provide low-level interfaces to offload computation to accelerator devices.

TAU has added Level Zero integration for DPC++ runtime events, including DPC++ symbol demangling support for computational kernel names. **TAU now provides tracing and profiling support for Level Zero API calls, and TAU also provides OpenCL performance instrumentation and measurement support**. **TAU has also included integrated Level Zero support for measuring asynchronous kernel execution and data transfer times**. **In addition, TAU now includes support for Intel AI Toolkit for Tensorflow/PyTorch** (further described in the next section) to complement existing Intel Exascale Laboratory MAQAO binary instrumentation, Intel PIN integration, PAPI and LIKWID performance counter library integration for Intel CPUs.

Kokkos. Both the Kokkos[34] and Raja[35] performance portability models include a mechanism for profiling tool integration and support, and **TAU has added support for both of them.** Kokkos and Raja are both C++ abstraction libraries, providing a common programming interface for heterogeneous back end technologies. In the case of Kokkos, it provides a cross-platform suite of programming hooks that are consistent regardless of the selected back end. Kokkos currently provides back end support for CUDA, HIP, SYCL, HPX, OpenMP and C++ threads. Because TAU already supports most of those back ends, it was a straightforward exercise to provide Kokkos profiling support through TAU.

Performance tools register with Kokkos at startup, providing an environment variable with the name of a shared object library that contains definitions of Kokkos callback functions for key events in the Kokkos runtime. These events include kernel entry/exit for `parallel_for`, `parallel_reduce`, and `parallel_scan` events. Kokkos also has events for allocation/free events, and its own region push/pop calls for instrumenting user code. **TAU added support for these events automatically for users, with no modification to existing programs necessary**.

MPI Tools Information Interface (MPI_T)

For many years, TAU has provided MPI measurement using the standard weak symbol replacement approach, where each MPI function is a weakly defined wrapper around an equivalent PMPI function. At either link or runtime, TAU replaces those weak definitions with strongly defined symbols including TAU instrumentation around the PMPI calls. MPI 3.0 introduced the MPI Tools Information Interface (MPI_T)[36]. MPI_T provides an interface that allows users to access two types of variables, which are called performance variables (PVARs) and control variables (CVARs). PVARs represent internal counters or metrics which can be accessed to collect and analyze performance data of the MPI library by tools. CVARs represent properties or configurations of the MPI library that may change the behavior of the implementation, such as the maximum number of buffers to use in a given pool and the eager communication threshold. There are variables that can only be modified before the MPI runtime is started in the application but others can be modified more than once and while the runtime is being executed.

MPI_T is not limited to only access one variable and all available variables may be consulted. However, the correct

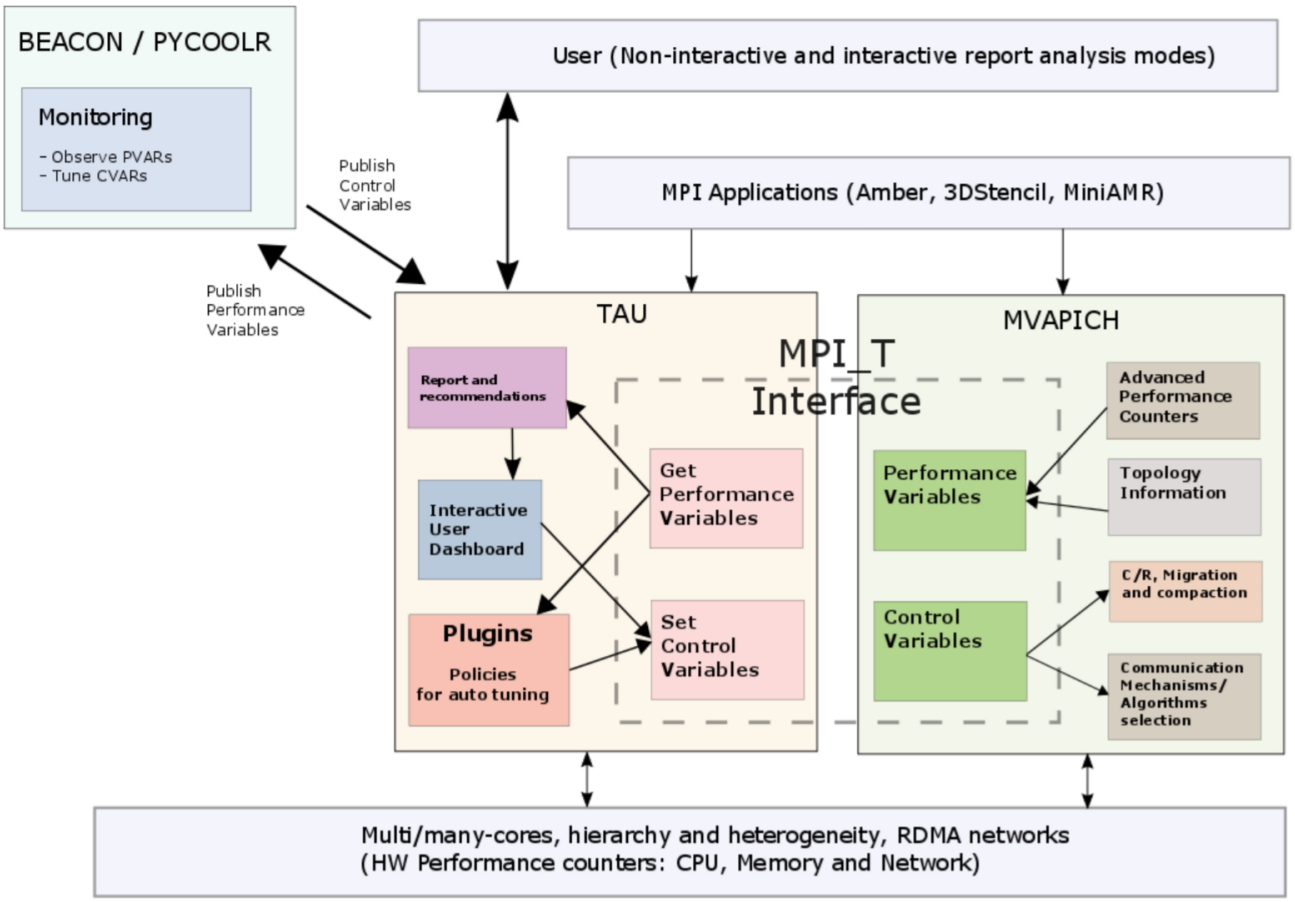

Figure 4. Implementation of TAU with the MPI_T interface.

modification of the performance variables is left to the user's discretion. Also, each MPI implementation may have different PVARs and CVARs.

Figure 4 shows how TAU is able to access the MPI_T interface to consult and also modify variables. We can interact with TAU at real-time using BEACON and PYCOOLR[37] and access the MPI_T interface to monitor CVARs and PVARs. Additionally, BEACON and PYCOOLR can also be used to apply manual tuning and dynamically tune CVARs.

Furthermore, TAU can also be used to obtain the value of all the CVARs without interaction with the user at the end of the application. TAU is able to use plug-ins for auto-tuning – analyze the behavior and/or performance of an application, and with a set of rules or functions analyze the available metrics to decide if a control variable, or a set of control variables, should be modified to increase the performance of an application or solve performance issues. Plug-ins can also be used to report CVARs which cannot be modified at runtime, so a report is generated after execution providing a message with the variables that should be modified and the recommended value.

Python Updates

Usability issues with traditional HPC programming languages, libraries, and frameworks are pushing users to newer, higher-level frameworks for specialized purposes, such as workflow management, application prototyping, deep learning, and data analytics. HPC systems, including leadership Department of Energy systems, are increasingly being called upon to support such workloads, often multi-language applications in which a Python frontend is used to call into C, Fortran and C++ libraries and to run code on GPU accelerators. **In order to support profiling such applications with TAU, we made several improvements to TAU's Python support and made wide-scale changes to data structures in TAU to better support applications with large numbers of potentially transient threads**.

Thread Scalability Improvements. Data analytics and deep learning frameworks making use of Python infrastructure tend to spawn a much larger number of threads than traditional HPC applications. In particular, they tend to spawn transient threads corresponding to a fine-grained task, and even when a thread persists as a worker across an entire program run, there tend to be more threads spawned per node than there are cores on the node. Historically, TAU had several limitations owing to its history being applied to programs with tens of threads rather than hundreds or thousands per process. These limitations reduced both scalability and ease of use. In cases where TAU maintains thread local data internally, this had been handled by fixed length arrays from 0 to a maximum number of threads per node. Because these arrays could not grow at runtime, the user was required to know in advance the maximum number of threads per node a given application will spawn. For deep learning frameworks, the number of threads used may depend on properties of the input data, so that there might be no setting guaranteed to work for any input.

To resolve these usability issues in TAU, we surveyed the existing TAU code, resulting in identification of 83 data structures in 39 source files in TAU that were of a fixed size at the time. **TAU was then refactored to replace such usages with dynamically sized data structures ("dynamization"), typically making use of C++ Standard Template Library data structures where possible**. A regression test suite was run after each data structure was updated in order to identify any issues caused by differing semantics between the old and new data structures. Once correctness had been achieved, we analyzed the performance consequences in terms of overhead of dynamization. Increased overheads can be incurred because data structures that might be resized must be locked on access to prevent access during resize. Where a performance regression is found, optimizations are applied, such as by using thread local caches.

With thread local caching, a thread first checks whether it already possesses a local copy requested data and, if so, uses it directly without locking. Only if a thread does not have an existing copy (typically only occurs early in the lifespan of a thread) must a lock be acquired so that the global, dynamically-sized data structure can be accessed. This optimization is permitted in cases where the data structures are append-only, implying that once an entry is inserted into the map, it will never be updated. Ultimately, all 83 instances of fixed-size data structures within TAU were replaced with dynamized data structures. Testing was performed using several example benchmarks, including the LULESH[38] code to provide testing of high thread counts in a distributed setting.

After optimization, no additional overhead is observed in a pthread and OpenMP matrix multiplication benchmark, nor in a pthread version of LULESH. Overhead of 9% was observed in OpenMP with LULESH. Similar overheads were observed with the TensorFlow iris example code. To further reduce overheads, the use of coarse-grained locks within TAU was reduced. Overhead with LULESH and with TensorFlow was reduced by changing the implementation of thread local caches to use a per-map lock instead of the global lock to protect access. After this, overheads were comparable between the original, fixed-thread-number implementation of TAU and the dynamic-thread-count version.

Deep Learning Framework Integration. The central difficulty in producing insightful and actionable performance data about high-level Machine Learning (ML) frameworks is the declarative nature of the language and the abstract, task-based nature of its runtime. TensorFlow[39], PyTorch[40], and the Keras[41] API provide callback hooks that enable tools such as TAU to receive information about the scheduling and execution of work by the runtimes. TensorFlow's built-in instrumentation uses a class `SessionRunHook` that enables registration of callbacks that are invoked at session start and session end. **We developed a `SessionRunHook` instance that inserts operations around each graph element that start and stop TAU timers and register metadata identifying the graph node being executed, whether on the CPU or GPU**.

For Keras, **we used call-path-based profiling to link the identity of high-level Keras operations responsible for a particular lower-level operation in the underlying framework by starting a timer representing the high-level operation when its graph node is executed**. In both TensorFlow and PyTorch, this is implemented using the Keras Callbacks API and is based on modifications to the `TensorBoardCallback` class built into Keras. Additionally, the `on_epoch_begin` and `on_epoch_end` callbacks are used to support phase-based profiling so that performance differences between epochs can be identified. These callbacks are used by specifying an additional parameter when training a model.

We compared the overheads of TAU instrumentation using technologies developed under this project with Kinteo, a first-party PyTorch profiling tool. We measured the overhead of TAU with combinations of Python profiling, CUPTI profiling for capturing CUDA kernel timings, and event-based sampling as applied to the FashionMNIST[42] example code included with PyTorch. As is shown in Table 1, TAU provides lower overheads than does the first-party Kineto profiler[43]. In this case, the Kineto profiler, even when configured to not store or process performance data in any way (instead discarding the data) incurred an overhead of 671% merely by being enabled. In contrast, a relatively lower overhead configuration of TAU (collecting GPU kernel timing data from an NVIDIA accelerator) introduced only 28.93% overhead, and a maximal configuration of TAU, collecting GPU kernel timings, Python function timings, and event-based samples incurred only 106% overhead. Note that these overheads are largely a result of the unavoidable CUPTI and Python profiling library support.

Instrumentation Methods

Source Based Instrumentation. TAU has historically supported source instrumentation through a component developed alongside TAU, the Program Database Toolkit[44], or PDT. C89, C11, C++98, C++11, Fortran 77, Fortran 90, Fortran 95, and UPC source code files are parsed using a parser for the given language to form an intermediate representation. This intermediate representation is then converted into a series of Program Database files (PDB files) which provide a common representation of the semantic structure of the source files along with mappings into the source file showing the location of each semantic element. TAU includes a tool, TAU instrumentor, which takes these PDB files and, utilizing a user-specified instrumentation specification or selective instrumentation file to determine where instrumentation points are desired, generates modified versions of the source files with instrumentation points added.

However, PDT has a number of missing features, limitations, and usability issues owing to its design and longevity which limit its applicability to applications now being developed for exascale and which result in users who are not compiler experts having difficulty applying PDT to their codes. PDT, its parsers, and the PDB file format were originally designed for the versions of C and Fortran in use circa 2000. While support for C++11 has been retrofitted onto PDT, support for certain language features is awkward; for example, C++11 lambda expressions are represented in PDB as generic expressions and TAU therefore cannot place instrumentation points within a lambda function.

	Wallclock Time (s)	Overhead (s)	Overhead as percent of original runtime
No Profiling	47.12	—	—
TAU with CUPTI	60.75	13.63	28.93%
TAU with Python profiling	65.37	18.25	38.73%
TAU with CUPTI+Python	82.92	35.80	75.98%
TAU with CUPTI+Python+Sampling	97.17	50.05	106.22%
Kineto with empty handler	363.38	316.26	671.18%

Table 1. Overheads of various configurations of TAU compared to the first-party performance monitoring tool Kineto as applied to the FashionMNIST[42] example included with PyTorch.

Additionally, new versions of compilers have introduced vendor-specific language extensions used in header libraries, which are not supported by PDT. The CUDA language[27], an extension of C designed for programming kernels for execution on NVIDIA GPUs, permits mixing device and host code within the same file. The C and C++ parsers included with PDT do not support the CUDA extensions to the language used in specifying or invoking device functions, which prevents source instrumentation of the host-targeting functions within a mixed device/host CUDA source file. PDT's primary C and C++ parser is based on the Edison Design Group C/C++ compiler frontend[45], a closed-source, proprietary parser licensed by the University of Oregon for inclusion in PDT. The terms of this license prohibit the redistribution of source code. As a result, PDT must be distributed primarily as binary packages, which results in large binary sizes, inability to compile PDT optimized for particular microarchitectures, and the need for PDT developers to explicitly build versions of the library for each new architecture adopted by HPC systems.

To resolve these issues, we have built a new C and C++ source instrumentor, called SALT[46], based on LLVM. LLVM[47] is an open-source compiler infrastructure project producing a common intermediate representation, LLVM IR, for many languages and architectures; a set of compilers producing LLVM IR; a set of tools for processing and transforming LLVM IR; and a set of code generators which reduce LLVM IR to native code for a variety of architectures.

It is tempting to insert instrumentation into LLVM IR. That approach allows a single instrumentor to be used to instrument code written in any language supported by any LLVM compiler, and this is an approach which has been adopted by some tools, such as Score-P[48] and the XRay Instrumentation system[49]. However, this approach has the downside that the instrumented code *must* go through LLVM's code generation stage. This prevents any other compiler, such as IBM, Cray, PGI, or other vendor-specific compilers, from being used with code instrumented at the LLVM IR level.

Instead, SALT operates at the compiler frontend stage, prior to generation of LLVM IR. This enables SALT to be used to instrument a code even if it will ultimately be compiled by a non-LLVM compiler. SALT is built using LLVM's libTooling API[50], which enables SALT's code analysis tools to closely mimic the interface of the full compilers. This will enable the analyzers to be used as a drop-in replacement for real compilers, allowing use with CMake-based build systems.

SALT provides a configurable instrumentation framework for TAU and other tools which can be invoked from the command line, from C or C++ codes, or from Python codes. It accepts configuration in both a YAML-based format and in the legacy TAU selective instrumentation file format. A default configuration is included to insert TAU instrumentation. Other configurations are provided for the PerfStubs[51] interface which targets multiple performance tools, for NVIDIA's NVTX annotation library[29] which tags code regions for NVIDIA's performance analysis tools, and for for AMD's ROCTX annotation library[52] which serves an equivalent function for AMD's tools. A high-level overview of the components is shown in Figure 5.

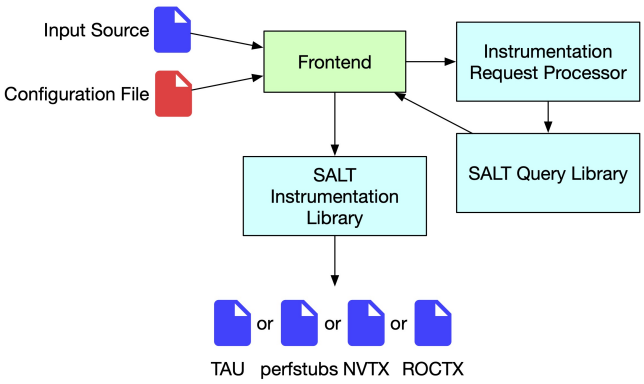

Figure 5. Flow of data through the SALT instrumentor. The frontend takes in input source files and a configuration file and processes the source file through `libtooling` up until the point where the Abstract Syntax Tree (AST) is generated. The configuration file is used to generate Instrumentation Requests. SALT queries the AST to identify AST nodes at which instrumentation calls should be placed. The instrumentation library then carries out such insertions.

SALT has been used for source instrumentation of DCA++[53], a code which simulates the physics of correlated electron systems using dynamical cluster approximation. It is written in modern C++ and uses the CMake build system, as well as HDF5, FFTW, BLAS, MPI, and CUDA. We configured DCA++ using the compiler wrapper as the compiler specified to the CMake build system. The SALT parser was able to successfully parse the source files within DCA++, and running DCA++ on a test input resulted in successful generation of TAU profiles.

We then extended SALT with an additional frontend for instrumentation of modern Fortran code. This project, called SALT-FM, adds a frontend plugin for LLVM's Fortran compiler, Flang. Combined with the C and C++ SALT instrumentors, the combined project can handle source instrumentation of mixed-language codes using all three languages.

Compiler Plugin Support. Most compilers provide some compiled-based instrumentation: for example, with GCC or Clang, when the option `-finstrument-functions` is passed to the compilation command, `cyg_profile` functions are inserted after each function entry and just before function exit. For GCC, a list of functions and files can be passed to be excluded from the instrumentation.

The TAU LLVM plugin gives more flexibility to specify which functions are to be instrumented. It can read an input file, the *selection file*, containing a list of functions to instrument, a list of functions to exclude from the instrumentation, a list of files to instrument functions in, and a list of files to exclude from the instrumentation. It supports wildcards both in file names and in function names. It supports C, C++ and Fortran. C++ support includes templates, polymorphism and overloaded operators.

The selection file takes a list of functions to instrument or to exclude from the instrumentation using `?` (one character) and `#` (any number of characters) as wildcards (the standard wildcard character, `*`, is valid in function signatures). It takes the function names for C and Fortran source code. In the example below, all functions starting with the substring `check` are instrumented *except* for `check_ortho`.

Listing 3: Sample selection showing detailed selection.

```
BEGIN_INCLUDE_LIST
check#
END_INCLUDE_LIST
BEGIN_EXCLUDE_LIST
check_ortho
END_EXCLUDE_LIST
```

Because of polymorphism, C++ requires the full prototype. The functions are demangled, and templates are supported using wildcards or explicit types. In the example below, all the functions specialized from the template `apply` are instrumented, and only the specialization for doubles of the template `add`.

Listing 4: Sample selection with C++ functions.

```
BEGIN_INCLUDE_LIST
void apply<#>(int, #**, #*, #, int)
void add<double>(int, double*, double*)
END_INCLUDE_LIST
```

File selection follows the same idea, using `?` and `*` as wildcards. In the example below, functions defined in all files whose names match `file*.c` are instrumented, except for `file4.c`. Functions in files that match `foo*.h` are not instrumented, and functions in `bar1.h` are.

Listing 5: Sample selection of file names.

```
BEGIN_FILE_INCLUDE_LIST
file*.c
bar1.h
END_FILE_INCLUDE_LIST
BEGIN_FILE_EXCLUDE_LIST
file4.c
foo*.h
END_FILE_EXCLUDE_LIST
```

Once the compiler has built the AST corresponding to the input source code, a function pass goes through all the functions of the program. At this point, we define two behaviors between which the user can choose using two environment variables: `TAU_COMPILER_CALLSITES` and `TAU_COMPILER_DEFINITIONS`. *Callsite instrumentation* inserts measurement points before and after each call to a given function. Therefore, for each function definition (including the *main* function), the plugin goes through each function call and decides whether or not to instrument it. *Definition instrumentation* inserts measurement points at the entry and at every exit of a function. The latter inserts instrumentation *inside* functions, the former inserts it *around* function calls.

If the selection file specifies source files that need to be included in or excluded from the instrumentation, call site instrumentation decides to instrument or not based on the file in which the function is called, whereas definition instrumentation decides based on the file in which the function is defined.

To avoid instrumenting very small functions, it is also possible to set a minimum number of instructions for a function to be instrumented using the environment variable `TAU_COMPILER_MIN_INSTRUCTION_COUNT`. We have found that a valid default value for the parameter is 50 instructions to prevent instrumentation of getter and setter methods in C++ classes.

DyninstAPI Support. DyninstAPI[54,55] can modify both processes (dynamic instrumentation) and binary files (static instrumentation), allowing the insertion of code snippets into functions, loops and basic blocks. This capability of the DyninstAPI is ideal for performance analysis tools, as it can be used to insert instrumentation into binaries, enabling the modification of applications without modifying the original code nor requiring the re-compilation of the application. The tool or part of the tool that employs Dyninst to modify binaries is called *mutator* and the modified binary, *mutatee*.

Taking advantage of this capability, we built a tool to instrument applications using the DyninstAPI. TAU's mutator utility (*tau_run*) makes use of the DyninstAPI and a pre-compiled TAU measurement library to instrument an application. The instrumentation can be done either dynamically or statically, according to the selected parameters when executing the mutator. If dynamic instrumentation is selected, the DyninstAPI is used to read the binary's image, create the application process (or processes if using MPI), and read the symbol tables to find the list of modules and routines from the application. Then, it inserts the TAU initialization and instrumentation of the different functions found in the modules of the application. Once the application is modified with all the required instrumentation, the modified application is executed. Instrumentation is inserted at the entry point of functions and also at the exit to measure time, or other metrics allowed by TAU, from the start to the end of a function.

In the case of static instrumentation, the image of a binary is read and TAU's initialization, finalization and instrumentation is inserted into a modified binary. This modified binary can be used in the same way as the binary of the original application, but its execution will generate profiling output from TAU. **Additionally, TAU can now also employ binary rewriting to modify shared**

libraries. The mutator reads the shared library and inserts instrumentation into the binary of the selected shared library. Then, the path of the modified library is included into the environmental variable *LD_LIBRARY_PATH*, which will load the instrumented library when the application is executed. However, in this case, the initialization of TAU is not inserted into the library. Therefore, the user must either execute a TAU instrumented application or execute with `tau_exec` to obtain the profiling information.

Rewriting of binaries is the preferred option if the objective is profiling an application multiple times, as the overhead of modifying the application to insert instrumentation will only appear once, when modifying the binary, whereas dynamic instrumentation requires the modification of an application at every execution.

TAU's mutator also accepts selective instrumentation, where a file is given as an input parameter. The selective instrumentation file can contain different functions, divided into two different labels: a) *INCLUDE_LIST*, where a list of functions to measure will be written and only those functions will be instrumented; b) *EXCLUDE_LIST*, as opposed to the include list, the functions that appear in the list do not require instrumentation, so if found, no instrumentation will be inserted into them.

Integration

Spack and E4S

The Extreme-scale Scientific Software Stack (E4S)[56] is a curated ecosystem of HPC tools and libraries developed under ECP. Software included in E4S provides a Spack[57] package which enables installation of the application and all of its dependencies via the Spack package manager. E4S provides binary repositories where Spack can obtain pre-built applications to accelerate installation, complete Spack environment definitions tailored for deployment both on general target platforms and at specific HPC centers, containerized Spack environments optimized for a variety of hardware platforms and accelerators, and Continuous Integration (CI) testing and validation for the software included in each E4S release.

TAU has been a member of the E4S project since its inception and has made a continuous effort to model full participation in E4S. TAU's developers have made it a priority to keep the TAU Spack package up to date, adding new versions and Spack variants for new TAU features as they are released. The TAU Spack package was also an early adopter both of Spack's internal package testing capabilities and the E4S test suite which performs validation on Spack-installed software. TAU is rigorously tested on the compilers, compiler architectures and hardware accelerators prioritized by E4S.

PerfStubs

The PerfStubs API[51] enables application developers to include generic timer stubs in their code base. This means that **application developers can now permanently incorporate specialized knowledge of their application's performance characteristics into the application rather than relying on automatic compile-time instrumentation** which may add superfluous timers or neglect critical code regions without additional input from a performance analyst. The PerfStubs API is activated at link time and connected to a performance profiling library such as TAU when performance data is desired, making full use of the chosen performance library's runtime features and output options. When inactive, PerfStubs has no effect on the timing or behavior of the instrumented application. PerfStubs provides this functionality through a plugin interface that detects that a tool has been loaded and has defined the expected function symbols to provide callback support. **TAU has added implementations of the PerfStubs symbols.**

PerfStubs timer hooks have been incorporated into Camtimers[58], PETSc[59], Ginkgo[60] and ADIOS2[61]. This has enabled the collection of TAU performance data from runs using these libraries without relying on compile time instrumentation.

Plugin support for CODAR: ADIOS2 and Chimbuko

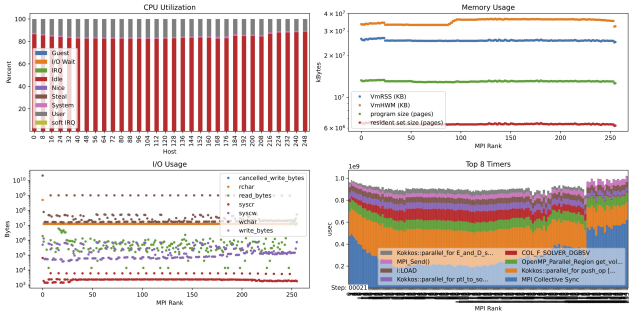

Figure 6. XGC Performance data in Python. The figures show real time updates of CPU utilization, memory usage, I/O usage and top timers in each MPI rank of the simulation.

One of the key developments for the TAU measurement library was a new feature added to allow custom plugins to integrate with the TAU profiling and tracing events. The plugin API allows for functional extensions to TAU, and several plugins have been written for event filtering, new output file and database formats, streaming performance data, and integration with other libraries. For example, TAU is now integrated with the Adaptable Input/Output (I/O) System (ADIOS2) in two different ways.

In addition to the PerfStubs integration so that TAU can measure ADIOS2 performance, TAU now can stage performance data using ADIOS2. This is valuable in several ways, one of which is that it allows for streaming real-time performance measurement of running applications using the ADIOS2 Python query API for in situ analysis. Figure 6 shows an example of visualizing TAU profile data using Python analysis. Each time that a major application iteration is completed, the ADIOS data output is updated, and the ADIOS2 Python reader API will read the next iteration of performance data, updating the display. Experiments of the WDMApp XGC fusion simulation were run with 256 MPI ranks on the Summit system and the performance data was routed through ADIOS2 and plotted with Python libraries, showing the iterative evolution of the performance of the simulation.

Another use of streamed TAU performance data is the Chimbuko trace analysis tool[62]. Chimbuko performs anomaly detection on TAU trace data streamed over ADIOS2, and feeds that analysis to a visualization framework.

SYMBIOMON and SOMA

In further support of online observability of long-running or performance data-intensive HPC applications, **two plugins were added to integrate with the SYMBIOMON**[63] **and SOMA**[64,65] **frameworks**. Both integrations make use of the MOCHI[66] underlying architecture to publish TAU performance data to the frameworks via remote procedure call (RPC). These highly configurable frameworks allow for online collection, aggregation, and analysis of the performance profiles gathered by TAU. SOMA, as a second generation version of SYMBIOMON, restructures TAU profiles into a canonical data model which improves upon two things: (1) it allows for flexible integration with other collection and analysis tools and (2) reduces the RPC overhead by enabling more control over how much data, and when, it is sent. This TAU data is then aggregated and analyzed by the framework and any subscribed analyzer tools in order to support online decision-making such as ending or changing the HPC simulation.

ECP Application Examples

As mentioned throughout the paper, TAU has been used with several of the ECP projects over the course of the project. This section presents a sample of the of applications and libraries that have utilized TAU for performance measurement and optimization while transitioning to the exascale systems.

Nalu-Wind Performance Analysis on Summit (ExaWind)

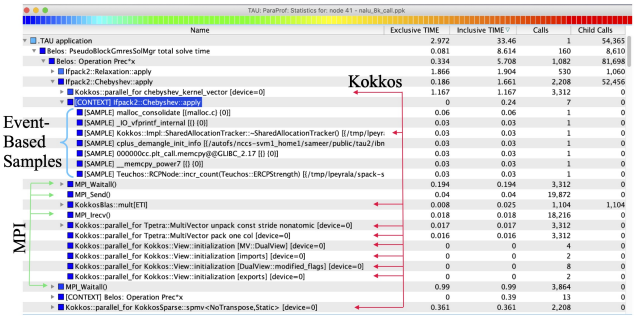

Figure 7. Instrumentation, MPI, Kokkos and sample profile data of Nalu-Wind visualized in ParaProf.

The ECP ExaWind project's scientific goal is to advance fundamental understanding of the flow physics that govern whole wind plant performance, including wake formation, complex terrain impacts, and turbine-turbine interaction effects[67]. Nalu-Wind[68] is a generalized, unstructured, incompressible flow solver for wind farm simulations at exascale.

Nalu-Wind was measured with TAU using multi-level instrumentation (MPI, Kokkos) on the Summit supercomputer. Figure 7 shows a view of the TAU profile in the ParaProf profile viewer, using the tree-table view. The tree-table is used to show call path hierarchies in the performance data, as well as inclusive and exclusive profile measurements. This example also uses sampling, showing how the sample data is integrated into the instrumented timer hierarchy.

ExaFEL Performance Analysis on Summit

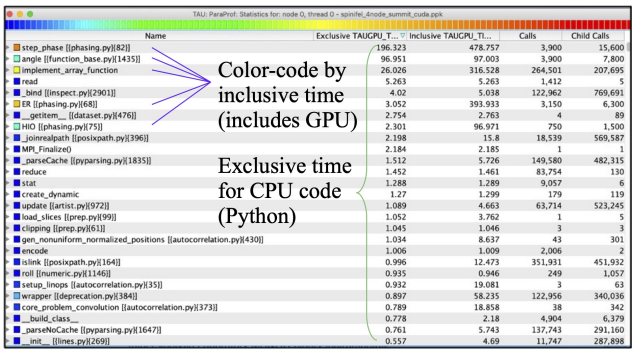

Figure 8. Python CPU measurement of Spinafel on Summit.

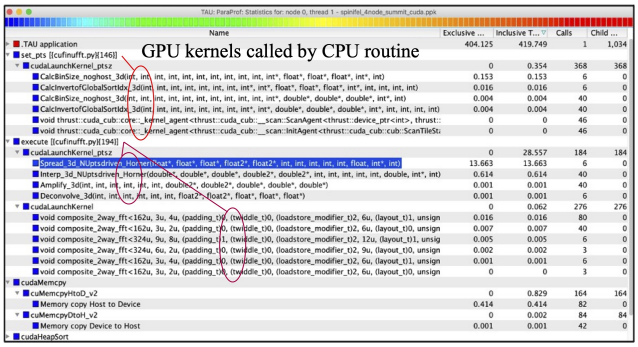

Figure 9. Measurement of Spinafel CUDA API calls on Summit.

The ECP ExaFEL challenge problem is the creation of an exascale-based data analysis workflow for serial femtosecond crystallography[69]. One of the goals of the project is to perform analysis of molecular structure x-ray diffraction data generated by the Linac Coherent Light Source (LCLS).

Figure 8 and 9 show the CPU and GPU measurements of unmodified Spinafel launched with `tau_python` on the Summit supercomputer. **This example demonstrates support in TAU for measuring MPI, Python, CUPTI, and event based sampling even in the Thrust solver library**.

AMReX Performance Analysis with TAU

The AMReX (AMR for the Exascale) project supports the development of block-structured AMR algorithms for solving systems of partial differential equations on exascale architectures[70]. Several ECP application projects and software use AMReX, including WarpX, ExaStar, ExaSky, MFIX-Exa, and AMR-Wind.

TAU can capture all heterogeneous events used by AMReX, including support for MPI, pthreads, OpenACC, CUDA, and integrated sampling data.

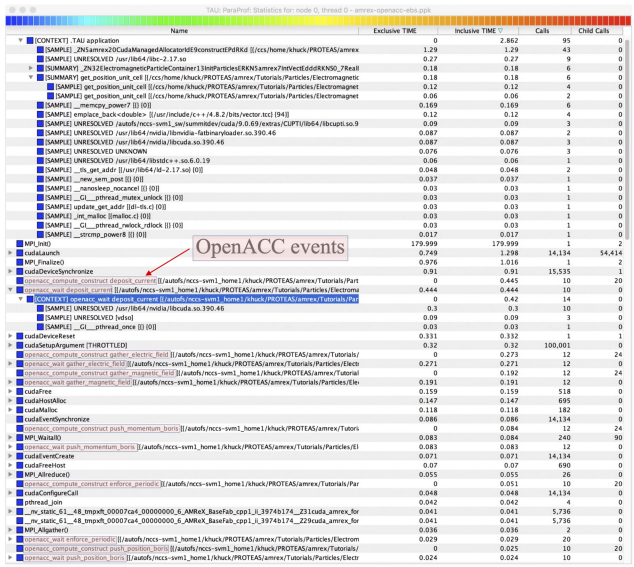

Figure 10. TAU measurement of an AMReX application using OpenACC.

Figure 10 shows a profile of an AMReX application that includes host-side OpenACC calls and sample data.

CANDLE Performance Measurement

The CANcer Distributed Learning Environment (CANDLE) Project focuses on large-scale machine learning problems for cancer applications. Specifically, the CANDLE challenge problem is to solve large-scale machine learning problems for three cancer-related pilot applications: the drug response problem, RAS pathway problem, and treatment strategy problem[71]. The project uses a Python-based runtime that launches many machine learning tasks.

TAU uses Python profiling as well as sampling to measure the TensorFlow applications on HPC systems. Figure 11 shows the now familiar tree table view along with a distribution of times spent in one function across all threads of execution.

ECP Codesign Center for Particle Applications (COPA)

The Co-design Center for Particle Applications (CoPA) focuses on co-design of several "sub-motifs," including short-range particle-particle interactions, long-range particle-particle interactions, particle-in-cell (PIC) methods, and O(N) complexity electronic structure and quantum MD (QMD) algorithms[72]. The CoPA project has developed a common library for particle methods called the CoPA Cabana Particle Tookit[73].

CabanaMD is a molecular dynamics proxy app built with Kokkos and the CoPa Cabana Particle Toolkit. Figure 12 shows an example of CabanaMD running on the Summit supercomputer with NVIDIA V100 GPUs. **This example demonstrates the use of Kokkos profiling API, MPI, CUPTI, and sampling**.

WDMApp

The Whole Device Model Application (WDMApp) project aims to develop a high-fidelity model of magnetically confined fusion plasmas, which is urgently needed to plan future physical experiments and optimize the design of future next-step fusion facilities[74].

The TAU team worked closely with the WDMApp project to help with overall performance analysis and porting of the XGC simulation to the Frontier supercomputer[75]. **PerfStubs integration with the Camtimers library, along with MPI, Kokkos, HIP, OpenMP, ADIOS2, and sampling support were used with XGC.** The HIP support in TAU was crucial in helping to identify computation kernels with high register pressure and testing code modifications that provided better performance. Finally, having a portable measurement system allowed us to compare across platforms, evaluating the performance on both Summit and Frontier.

Conclusion

For performance analysis tools to stay relevant and useful it is critical that they maintain compatibility with developing software and hardware capabilities. The bleeding edge exascale platforms targeted by the ECP and the complex software applications being developed there were challenging targets for TAU to support. In addition to achieving significant new functionality that will be beneficial to application developers at DOE sites, the effort advanced TAU's modernization as a robust, production-quality performance tool system for the wider HPC community.

The success of TAU's improvement and functional expansion under the ECP was achieved not just by following a mandate and making use of technological resources afforded by the effort. Access to and collaboration with the teams responsible for hardware development, application and tool development and facilities management was critical both for overcoming technical challenges and maintaining the direction of TAU's development over the course of the project. Facilitation of interaction between these groups afforded by ECP was incredibly valuable. As TAU continues to be improved to support the performance analysis requirements of future use cases it, and by extension its users, will benefit both from the technical achievements and the community development that occurred under the ECP.

Acknowledgements

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357. The material on Python updates and deep learning framework integration is based upon work supported by the U.S. Department of Energy, Office of Science, Office of SBIR & STTR Programs under Award Number DE-SC-0019700. The material on C and C++ source instrumentation is based upon work supported

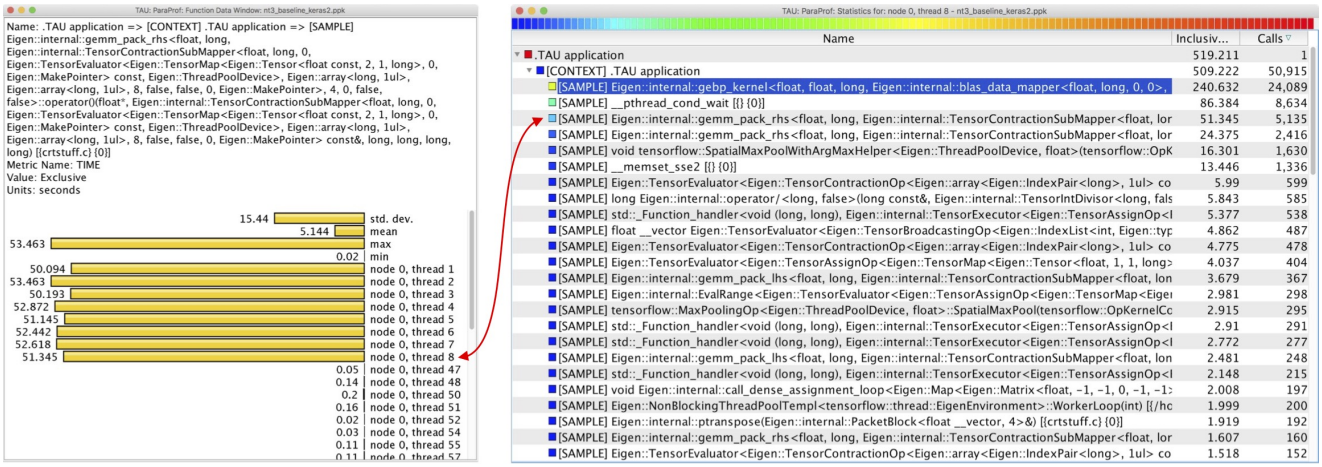

Figure 11. On the left, the profile of function `Eigen::internal::gemm_pack_rhs` across all threads of execution is shown. On the right side is a view of the sampling data collected in the application run.

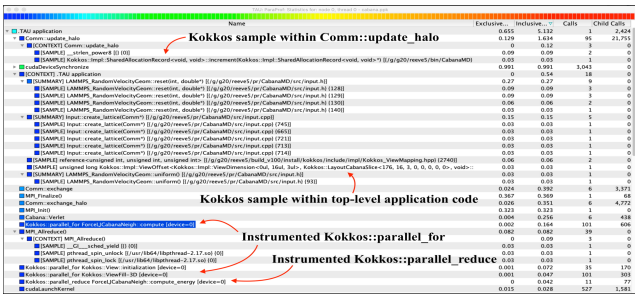

Figure 12. CabanaMD profile data from the Summit supercomputer, visualized in ParaProf.

by the U.S. Department of Energy, Office of Science, Office of SBIR & STTR Programs under Award Number DE-SC-0022511. The material on Fortran source instrumentation is based on work supported by the National Aeronautics and Space Administration under Contract Number 80NSSC24PB401.

References

1. Younge AJ and Gamblin T. ECP packaging technologies. 2019; URL https://www.osti.gov/biblio/1646052.
2. The Exascale Computing Project. The Exascale Computing Project. https://www.exascaleproject.org/.
3. Shende SS and Malony AD. The TAU parallel performance system. *The International Journal of High Performance Computing Applications* 2006; 20(2): 287–311. DOI:10.1177/1094342006064482. URL https://doi.org/10.1177/1094342006064482. https://doi.org/10.1177/1094342006064482.
4. The Exascale Computing Project. PROTEAS-TUNE. https://www.exascaleproject.org/research-project/proteas-tune/.
5. The Exascale Computing Project. The Exascale Computing Project Applications. https://www.exascaleproject.org/research/#application.
6. The Exascale Computing Project. The Exascale Computing Project Software Technology. https://www.exascaleproject.org/research/#software.
7. Bertran R, Sugawara Y, Jacobson HM et al. Application-level power and performance characterization and optimization on IBM Blue Gene/Q systems. *IBM Journal of Research and Development* 2013; 57(1/2): 4–1.
8. Rosales C, Cazes J, Milfeld K et al. A comparative study of application performance and scalability on the Intel Knights Landing processor. In *High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^ 3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers 31*. Springer, pp. 307–318.
9. Oak Ridge National Laboratory. The Frontier supercomputer. https://www.olcf.ornl.gov/frontier/.
10. Argonne National Laboratory. The Aurora supercomputer. https://www.anl.gov/aurora.
11. Buck B and Hollingsworth J. An API for runtime code patching. *The International Journal of High Performance Computing Applications* 2000; 14(4): 317–329. URL http://citeseer.ist.psu.edu/buck00api.html.
12. Charif-Rubial A, Barthou D, Valensi C et al. MIL: A language to build program analysis tools through static binary instrumentation. In *Proc. 20th Annual International Conference on High Performance Computing, HiPC 2013*. IEEE, pp. 206–215.
13. Laurenzano M, Tikir M, Carrington L et al. PEBIL: Efficient static binary instrumentation for Linux. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*. pp. 175–183. DOI:10.1109/ISPASS.2010.5452024.
14. Malony AD and Huck KA. General hybrid parallel profiling. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. pp. 204–212. DOI:10.1109/PDP.2014.38.
15. Terpstra D, Jagode H, You H et al. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 2010. pp. 157–173.
16. Treibig J, Hager G and Wellein G. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th international conference on parallel processing workshops*. IEEE, pp. 207–216.
17. Bell R, Malony AD and Shende S. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In *Euro-Par 2003 Parallel Processing: 9th International Euro-Par Conference Klagenfurt, Austria, August*

26-29, 2003 Proceedings 9. Springer, pp. 17–26.
18. Huck K, Malony A, Bell R et al. Design and implementation of a parallel performance data management framework. In *2005 International Conference on Parallel Processing (ICPP'05)*. pp. 473–482. DOI:10.1109/ICPP.2005.29.
19. Huck KA, Malony AD, Shende S et al. Knowledge support and automation for performance analysis with PerfExplorer 2.0. *Scientific programming* 2008; 16(2-3): 123–134.
20. Knüpfer A, Brunst H, Doleschal J et al. The Vampir performance analysis tool-set. In *Tools for High Performance Computing*. Springer, 2008. pp. 139–155.
21. Mohr B, Malony AD, Shende S et al. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing* 2002; 23: 105–128.
22. Exascale Computing Project. ECP proxy applications: miniQMC. https://proxyapps.exascaleproject.org/app/miniqmc/.
23. Oak Ridge National Laboratory. Frontier. https://www.olcf.ornl.gov/olcf-resources/compute-systems/frontier/.
24. Eschweiler D, Wagner M, Geimer M et al. Open Trace Format 2: The next generation of scalable trace formats and support libraries. In *International Conference on Parallel Computing*. URL https://api.semanticscholar.org/CorpusID:37062839.
25. Denny JE, Lee S and Vetter JS. Clacc: Translating OpenACC to OpenMP in Clang. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, pp. 18–29.
26. Coti C, Denny JE, Huck K et al. OpenACC profiling support for Clang and LLVM using Clacc and TAU. In *2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools)*. IEEE, pp. 38–48.
27. Kirk D et al. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, volume 7. pp. 103–104.
28. NVIDIA. Cuda profiling tools interface, 2020. https://docs.nvidia.com/cuda/cupti/index.html.
29. NVIDIA. NVIDIA Tools Extension (NVTX). https://docs.nvidia.com/nvtx/index.html.
30. Malony AD, Biersdorff S, Shende S et al. Parallel performance measurement of heterogeneous parallel systems with GPUs. In *2011 international conference on parallel processing*. IEEE, pp. 176–185.
31. NVIDIA. NVIDIA Management Library (NVML), 2020. https://developer.nvidia.com/nvidia-management-library-nvml.
32. AMD. AMD CDNA™2 Architecture. https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf.
33. Intel Corporation. Level Zero. https://dgpu-docs.intel.com/technologies/level-zero.html.
34. Trott CR, Lebrun-Grandié D, Arndt D et al. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems* 2022; 33(4): 805–817. DOI:10.1109/TPDS.2021.3097283.
35. Beckingsale DA, Burmark J, Hornung R et al. Raja: Portable performance for large-scale scientific applications. In *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE, pp. 71–81.
36. Ramesh S, Shende S, Malony A et al. MPI performance engineering with the MPI tool interface: The integration of MVAPICH and TAU. *Parallel Computing* 2018; 77.
37. Ramesh S, Mahéo A, Shende S et al. MPI performance engineering with the MPI tool interface: the integration of MVAPICH and TAU. In *Proceedings of the 24th European MPI Users' Group Meeting*. pp. 1–11.
38. Karlin I, Keasler J and Neely R. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, 2013.
39. Abadi M, Agarwal A, Barham P et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.
40. Paszke A, Gross S, Massa F et al. *PyTorch: an imperative style, high-performance deep learning library*. Red Hook, NY, USA: Curran Associates Inc., 2019.
41. Chollet F et al. Keras. https://keras.io, 2015.
42. Xiao H, Rasul K and Vollgraf R. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms, 2017. 1708.07747.
43. Kineto PyTorch Profiler. https://github.com/pytorch/kineto.
44. Lindlan KA, Cuny J, Malony AD et al. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. SC '00, IEEE Computer Society. ISBN 0780398025, p. 49.
45. Edison Design Group C++ Frontend. https://www.edg.com/c.
46. ParaTools, Inc. SALT: An LLVM-based Source Analysis Tookit for HPC, 2025. URL https://github.com/ParaToolsInc/salt.
47. Lattner C and Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, pp. 75–86.
48. Psallidas F. *Physical Plan Instrumentation in Databases: Mechanisms and Applications*. PhD Thesis, Columbia University, 2019.
49. Berris DM, Veitch A, Heintze N et al. Xray: A function call tracing system, 2016.
50. Duffy EB, Malloy BA and Schaub S. Exploiting the Clang AST for analysis of C++ applications. In *Proceedings of the 52nd annual ACM southeast conference*.
51. Boehme D, Huck K, Madsen J et al. The case for a common instrumentation interface for HPC codes. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 33–39. DOI:10.1109/ProTools49597.2019.00010. URL https://doi.ieeecomputersociety.org/10.1109/ProTools49597.2019.00010.
52. AMD. ROCTX: Application Code Annotation. https://docs.amd.com/bundle/ROCTracer-User-Guide-v5.0-/page/ROCTX_Application_Code_Annotation.html.
53. Hähner UR, Alvarez G, Maier TA et al. DCA++: A software framework to solve correlated electron problems with modern quantum cluster methods. *Computer Physics Communications* 2020; 246: 106709.

54. Williams WR, Meng X, Welton B et al. Dyninst and MRNet: Foundational infrastructure for parallel tools. In Knüpfer A, Hilbrich T, Niethammer C et al. (eds.) *Tools for High Performance Computing 2015*. Cham: Springer International Publishing. ISBN 978-3-319-39589-0, pp. 1–16.
55. Bernat AR and Miller BP. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. PASTE '11, New York, NY, USA: Association for Computing Machinery. ISBN 9781450308496, p. 9–16. DOI:10.1145/2024569.2024572. URL https://doi.org/10.1145/2024569.2024572.
56. E4S Project. The Extreme-Scale Scientific Software Stack. https://e4s-project.github.io/.
57. Gamblin T, LeGendre M, Collette MR et al. The Spack package manager: Bringing order to HPC software chaos. In *Supercomputing 2015 (SC'15)*. Supercomputing 2015 (SC'15), Austin, Texas, USA. DOI:10.1145/2807591.2807623. URL https://github.com/spack/spack. LLNL-CONF-669890.
58. WDMApp. Camtimers, 2024. URL https://github.com/wdmapp/camtimers.
59. Abhyankar S, Brown J, Constantinescu EM et al. PETSc/TS: A modern scalable ODE/DAE solver library. *arXiv preprint arXiv:180601437* 2018; .
60. Anzt H, Cojean T, Chen YC et al. Ginkgo: A high performance numerical linear algebra library. *Journal of Open Source Software* 2020; 5(52): 2260.
61. Godoy WF, Podhorszki N, Wang R et al. ADIOS 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX* 2020; 12: 100561.
62. Kelly C, Ha S, Huck K et al. Chimbuko: A workflow-level scalable performance trace analysis tool. In *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. 2020. pp. 15–19.
63. Ramesh S, Ross RB, Dorier M et al. SYMBIOMON: A high-performance, composable monitoring service. *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)* 2021; : 332–342.
64. Yokelson D, Lappi O, Ramesh S et al. SOMA: Observability, monitoring, and in situ analytics for exascale applications. *Concurr Comput Pract Exp* 2024; 36.
65. Yokelson D, Titov M, Ramesh S et al. Enabling performance observability for heterogeneous hpc workflows with SOMA. In *International Conference on Parallel Processing*.
66. Ross RB, Amvrosiadis G, Carns PH et al. Mochi: Composing data services for high-performance computing environments. *Journal of Computer Science and Technology* 2020; 35: 121–144. URL https://api.semanticscholar.org/CorpusID:210926325.
67. The Exascale Computing Project. ExaWind. https://www.exascaleproject.org/research-project/exawind/.
68. Sharma A, Brazell MJ, Vijayakumar G et al. Exawind: Open-source cfd for hybrid-rans/les geometry-resolved wind turbine simulations in atmospheric flows. *Wind Energy* 2024; 27(3): 225–257. DOI:https://doi.org/10.1002/we.2886. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/we.2886. https://onlinelibrary.wiley.com/doi/pdf/10.1002/we.2886.
69. The Exascale Computing Project. ExaFel. https://www.exascaleproject.org/research-project/exafel/.
70. The Exascale Computing Project. Adaptave Mesh Refinement. https://www.exascaleproject.org/research-project/adaptive-mesh-refinement/.
71. The Exascale Computing Project. CANDLE. https://www.exascaleproject.org/research-project/candle/.
72. The Exascale Computing Project. Particle Based Applications. https://www.exascaleproject.org/research-project/particle-based-applications/.
73. Slattery S, Reeve ST, Junghans C et al. Cabana: A performance portable library for particle-based simulations. *Journal of Open Source Software* 2022; 7(72): 4115.
74. The Exascale Computing Project. WDMApp. https://www.exascaleproject.org/research-project/wdmapp/.
75. Suchyta E, Klasky S, Podhorszki N et al. The Exascale Framework for High Fidelity coupled Simulations (EFFIS): Enabling whole device modeling in fusion science. *The International Journal of High Performance Computing Applications* 2022; 36(1): 106–128. DOI:10.1177/10943420211019119. URL https://doi.org/10.1177/10943420211019119. https://doi.org/10.1177/10943420211019119.