Chapter 10
Advances in the TAU Performance System

Allen Malony, Sameer Shende, Wyatt Spear, Chee Wai Lee, and Scott Biersdorff

Abstract Evolution and growth of parallel systems requires continued advances in the tools to measure, characterize, and understand parallel performance. Five recent developments in the TAU Performance System are reported. First, an update is given on support for heterogeneous systems with GPUs. Second, event-based sampling is being integrated in TAU to add new capabilities for performance observation. New wrapping technology has been incorporated in TAU's instrumentation harness, increasing observation scope. The fourth advance is in the area of performance visualization. Lastly, we discuss our work in Eclipse Parallel Tools Platform.

10.1 Introduction

The TAU performance system [12] has been in development and use in the high-performance computing (HPC) community for over 20 years. During this time, it was the changes in the parallel architectures, systems, software development, and applications that make up HPC technologies that generated requirements for continual improvement in the TAU toolset and the methodology it supports. Today's HPC evolution is no different in the demands it places on next-generation performance tools. The benefit of an established system like TAU and others, is in the ability to leverage existing capabilities to create new and more powerful features. Of course, it can also be more challenging if advances force re-engineering of the toolset's architecture and implementation. The following paper highlights recent advances in the TAU performance system in five areas that cover different aspects of the changing needs of HPC. First, we discuss updates on our work to integrate into TAU support for measurement of heterogeneous systems using GPUs. Second,

A. Malony (✉) · S. Shende · W. Spear · C.W. Lee · S. Biersdorff
University of Oregon, Eugene, OR, USA
e-mail: malony@cs.uoregon.edu; sameer@cs.uoregon.edu; wspear@cs.uoregon.edu; cheelee@cs.uoregon.edu; scottb@cs.uoregon.edu

H. Brunst et al. (eds.), *Tools for High Performance Computing 2011*,
DOI 10.1007/978-3-642-31476-6_10, © Springer-Verlag Berlin Heidelberg 2012

we describe the addition of event-based sampling in TAU to address limitations of a purely probe-based approach for performance observation. Instrumentation has always been an important technology for enabling performance measurement. The third area covers new wrapping technology that has been incorporated in TAU's instrumentation harness. The need to makes sense of high-dimensionality performance data motivates better methods for data analysis and presentation. The fourth advance we will discuss is in the area of performance visualization. Lastly, we discuss our work in Eclipse Parallel Tools Platform.

10.2 Instrumentation Of GPU Accelerated Code

Understanding the performance of scalable heterogeneous parallel systems and applications will depend on addressing new challenges of instrumentation, measurement, and analysis of heterogeneous components, and integrating performance perspectives in a unified manner. Here we will cover an approach to addressing these requirements undertaken by the TAU, PAPI [3] and Vampir-Trace [4] development teams. We will examine the approach in terms of computation and measurement models in order to ground the discussion on tool implementation. The measurement techniques supported by the tools are intended to be practical solutions for these approaches with respect to present technology.

10.2.1 Synchronous Method

The validity of the time measurement is predicated on when the kernel issued to the accelerator begins execution and when it ends. We use the term synchronous to indicate that it is the CPU (host) who is observing the begin and end events, as denoted by the diamonds in Fig. 10.1. Measurements are made on the CPU by recording events before kernel launch and after synchronization. If the host immediately waits on kernel termination after launch, its kernel measurement is, in effect, synchronized with the kernel execution. In this case, the measurement method is equivalent to measuring a subroutine call. In essence, a synchronous approach assumes the kernel will execute immediately, and the interval of time between the begin and end events will accurately reflect kernel performance. Unfortunately, this assumption is overly restrictive and leads to inaccuracies when more flexible modes of kernel execution are used. As the figure suggests, the host need not block after kernel launch and it can be a long time before it is synchronized with the kernel, resulting in poor estimates of actual kernel execution time. Moreover, multiple kernels can be launched into a stream or multiple streams before a synchronization point is encountered. The benefit of a synchronous approach is that it does not require any additional performance measurement mechanisms beyond what is presently available.

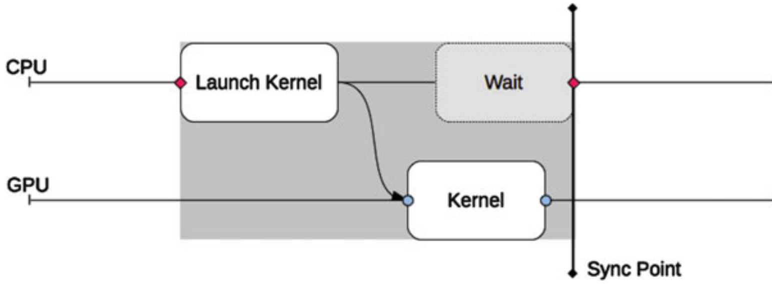

Fig. 10.1 Synchronous method timeline. *Shaded area* represents the execution time

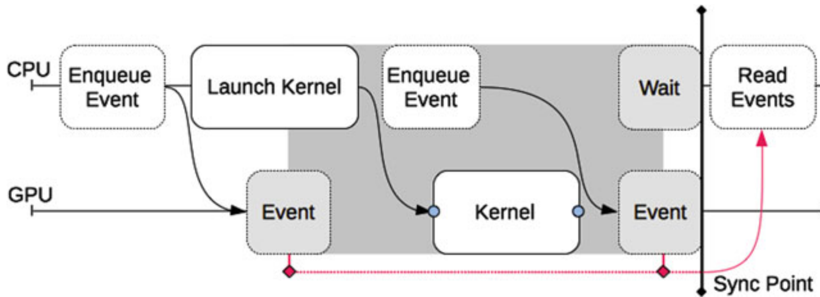

Fig. 10.2 Event queue method timeline. *Shaded area* represents the measured execution time on the host device

10.2.2 Event Queue Method

The main problem with the synchronous approach is that the kernel execution is measured indirectly, not by the GPU. Consider a special type of kernel called an event kernel which will record the state of the GPU when it is executed. If we could inject an event kernel into the stream immediately before and after the computational kernel, it would be possible to obtain performance data more closely linked with kernel begin and end. While it is the responsibility of the host to generate the event kernels, queue them into the stream, and read the results, it is the underlying GPU device layer that will take responsibility for making the measurement. Measurements are made on the event kernels placed in the same stream as the computational kernel. In theory this method, shown in Fig. 10.2, works well. It adequately addresses the case where multiple kernels are launched in a stream, if each is wrapped by an associated event kernel they are all accounted for even if the synchronization point is not until much later. However, there are a few practical downsides. First, it relies entirely on the device manufacture to provide support for the event kernel concept. The notion of events is a part of both the CUDA and OpenCL specification. However, restrictions on how events can be used and what performance information is returned is implementation dependent.

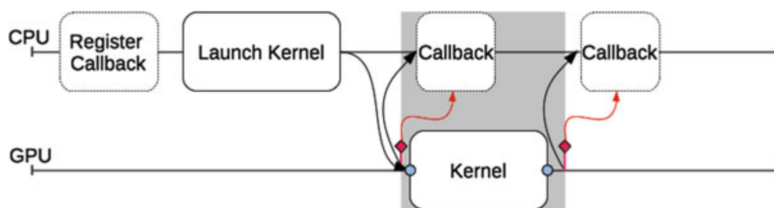

Fig. 10.3 Callback method timeline

10.2.3 Callback Method

A third method relies on a mechanism in the device layer that triggers callbacks on the host for registered actions, like the begin and end of kernel execution. Registered callbacks are triggered by GPU actions, allowing more tightly coupled measurements to take place. The "callback" method portrayed in Fig. 10.3 suggests that more immediate kernel performance measurement is possible since control can be given directly to a CPU process via the callback. It is also more flexible since a wider range of callbacks might be provided, and performance measurement can be specific to callback type. The process of callback registration makes it possible to avoid code modification at locations of kernel launch in the application. Clearly, a callback method is dependent on the device manufacturer to provide support in the device layer and even the GPU hardware. The research presented here demonstrates support for GPU performance measurement with CUDA and OpenCL in three well-known performance tools PAPI, VampirTrace, and the TAU Performance System. The tools targeted in this paper support performance counter measurement, profiling, and tracing for scalable parallel applications, and are generally representative of probe-based measurement systems.

10.2.4 TAU Performance System Implementation

TAU [12] has tools for source instrumentation, compiler instrumentation, and library wrapping that allows CPU events to be easily observed. In particular, they allow library wrapping of the CUDA runtime/driver API and preloading of the wrapped library prior to execution. Then, each call made to a runtime or driver routine is intercepted by TAU for measurement before/after calling the actual CUDA routine. TAU library interposition happens dynamically with the Linux LD_PRELOAD mechanism and can be used on an un-instrumented executable. Such features are commonly used in other performance tools. For instance, VampirTrace also applies the LD_PRELOAD mechanism with the CUDA libraries [5]. Clock synchronization is needed to correct any time lag between the CPU and the GPU, just as when tracing events across multiple nodes. This is accomplished by measuring simultaneously

(or as nearly as possible) at a synchronization point the time on CPU and the time on the GPU. The GPU time can be obtained at the synchronization event which has just occurred. By measuring the difference between these two times we can measure the clock lag between the CPU and GPU.

10.3 Event Based Sampling (EBS)

Our integration of the TAU performance system with event-based sampling measurement, called TAUebs, represents our approach to a hybrid measurement system which includes both probe-based and sampling-based components. A hybrid measurement system attempts to marry the strengths of both probe- and sampling-based measurements while mitigating their weaknesses.

Sampling-based measurements determine an application's performance by statistical observation via some interrupt mechanism. Probe-based measurement systems instruments the application code at specific points to collect performance data at the time the instrumented code is executed. Morris et. al. [10] detailed the design and implementation of a prototype for such a hybrid measurement system. The initial prototype focused on the generation of sampling traces followed by a post-mortem merge operation with probe-based TAU profiles generated from the same application run. Building on this prototype, we have now implemented support for tighter integration of sampling-based information into TAU's profile data structures at application runtime. The tool employs the same sampling technology based on work by HPCToolkit [1], Perfsuite [11] and PAPI [3].

When a TAU-instrumented application encounters an interrupt to take a sample, TAU's event path data structures are consulted to discover the current TAU event context. A basic information-gathering approach is simply to capture the number of times a program counter value is encountered. A histogram of sample counts against encountered program counter values is maintained for each unique TAU event path context at runtime. At the end of the application, before TAU profiles are written to disk, TAU's event paths are augmented by performance information derived from sampling-based measurements as follows:

1. An intermediate event representing the sum of all sampled events that were encountered in the context of some TAU event path P is created as a new leaf node to path P.
2. For each program counter value found in the histogram maintained for path P, attempt to resolve the source file name, function name and line number. Multiple program counter values mapping to the same line number in the code are treated as the same sample event. Metric values are then assigned by multiplying the known sample period with the number of samples. Finally, each sample event are added to TAU's event path as leaf nodes.

The sample information collected for each TAU event path is flat within the context of that TAU event path. This design is illustrated in Fig. 10.4. We are

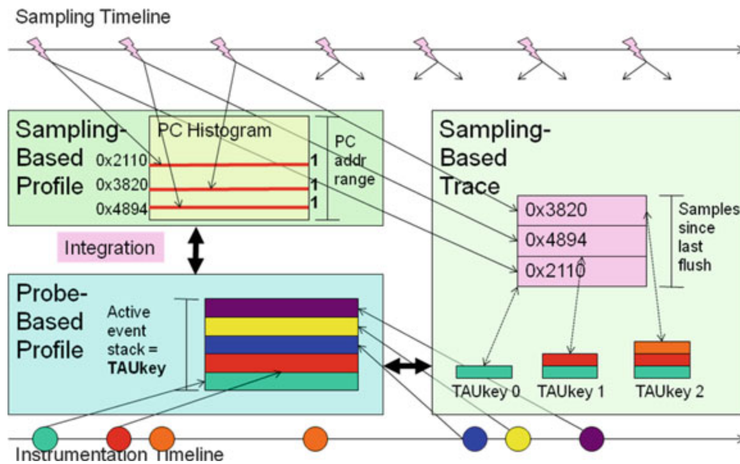

Fig. 10.4 An illustration of the complementary sampling and instrumentation methodologies. Instrumentation provides sampling context via an event stack

working toward incorporating stack unwinding for the runtime integration of sample information to TAU profiles. We are also working on resolving issues associated with the correct mapping of program counters to code structure in the face of code optimization. We have implemented several solutions to mitigate issues associated with asynchronous signal safety and thread safety. In particular, we did not wish to pre-allocate memory for entire code address spaces for program counter histograms. As a result, we had to make use of pool storage allocators in C++ to reduce the chances of making our own memory allocation calls when our signal handle happened to interrupt some memory allocation event in application code.

Szebenyi et. al. [15] presented results on a similar hybrid measurement effort developed on top of the Scalasca set of tools. They specifically insert probe-based measurements on MPI calls while capturing the performance of the rest of the application through sampling-based measurements. They have described their approach for addressing some of the common issues we both face in our efforts.

10.4 Automatic Wrapper Library Generation

Many parallel applications are constructed using software library packages with interfaces callable from standard programming languages. Packages are often layered, internally calling other libraries to implement underlying functionality, which can be hidden to the user. Having an ability to intercept package calls at library routine interfaces enables performance tools to gather both semantic (contextual) and performance data for analysis purposes. Shende et al. [13] discusses this wrapper feature in detail in the context of tracking IO events. Below is a summary of this feature.

TAU can automate the creation of wrapper libraries using its `tau_gen_wrapper` tool. It parses the interface of a library (as represented in a header file) and creates an wrapped interface for each routine, as may be specified in a selective instrumentation file. This interface starts and stops TAU timers around the original function call. There are three ways to generate the wrapper library:

1. Redefining the function using a pre-processor macro
2. Preloading the wrapper library at runtime
3. Using linker-based substitution of a routine with an alternative implementation.

This tool supports all three cases. It internally uses the Program Database Toolkit (PDT) to parse the header file.

This can be done by defining a header file that internally redefines the name of a routine as a macro that redirects all references to the given call with another. The compiler's pre-processor then replaces all references to the original call at the callsite in the source code with the corresponding call defined by the tool (e.g., read replaced by tau_read).

The above approach works well for C and C++ programs where library calls are replaced explicitly during compilation. Unfortunately, this approach does not extend well to Fortran programs. Moreover, the instrumentation technique is limited to application code regions where the source code is available for recompiling.

This method also relies explicit re-compilation but includes support for Fortran. TAUs instrumentation tool (`tau_instrumentor`) examines the source code, its PDB file as generated by the Program Database Toolkit (PDT), and re-writes the Fortran calls in the instrumented source code. In this method, performance measurement code is inserted directly in the source code.

Many HPC operating systems such as Linux, Cray Compute Node Linux (CNL), IBM BlueGene Compute Node Kernel (CNK), Solaris permit pre-loading of a library in the address space of an executing application specifying a dynamic shared object (DSO) in an environment variable (LD_PRELOAD). It is possible to create a tool based on this technique that can intercept library operations by means of a wrapper-library where the all calls are redefined to call the global routine (identified using the dlsym system call) internally. This method only supports dynamic executables—IBM BlueGene and Cray XE6 and XK6 systems default to static executables but dynamic executables can be requested.

10.5 3D Visualization

It has always been challenging to create new performance visualizations, for three reasons. First, it requires a design process that integrates properties of the performance data (as understood by the user) with the graphical aspects for good visual form. This is not easy, if one wants effective outcomes. Second, unlike visualization of physical phenomena, performance information does not have a natural semantic visual basis. It could utilize a variety of graphical forms and

visualization types (e.g., statistical, informational, physical, abstract). Third, with increasing application concurrency, performance visualization must deal with the problem of scale. The use of interactive three-dimensional (3D) graphics clearly helps, but the visualization design challenge is still present.

In addition to these challenges, there are also practical considerations. Because of the richness of parallel performance information and the different relationships to the underlying application semantics, it is unreasonable to expect just a few performance visualizations to satisfy all needs. Where visualization of large performance information does exist, it is generally embedded as "canned" displays in a profile or trace analysis tool. If a user has a different concept in mind, they have very limited ability to make changes.

To move towards a more general method of creating 3D visualizations, we considered the salient components of the existing versions and the need to specify aspects of a 3D design in a more flexible manner. Two ideas resulted: (1) separate the visualization layout design from visualization user interface (UI) design, and (2) allow the properties of each to be specified by the user. We have added these capabilities to the ParaProf [2] profile analysis tool.

Visualization layout design is concerned with how the visualization will appear. Our approach allows the visual presentation to be specified with respect to the parallel profile data model (events, metrics, metadata) and possible analysis of this information [14]. Two basic layout approaches we support are mapping to Cartesian coordinates provided by MPI and filling a space of user-defined dimensions in order of MPI rank. We have also worked to develop a specification language for describing more complex layouts of thread performance in a 3D space. In our initial implementation of these custom layouts, mathematical formulae define the coordinates and color value of each thread in the layout. The formulae are based on variables provided by the profile data model. These input variables include event and metric values for the current thread being processed as well as global values such as the total number of threads in the profile. The specification is applied successively to each thread in the profile to determine X, Y and Z coordinate values and color values which are used to generate the visualization graphics. Our initial implementation for expression analysis uses the MESP expression parser library [9]. MESP provides a simple syntax for expressing mathematical formulae but is powerful enough to allow visualization layouts based on architecturally relevant geometries or the mathematical relationship of multiple performance variables.

The layout of points in a visualization which conforms to a useful pattern, reflecting the physical layout of the system or the data decomposition of the application, may be densely packed or otherwise obscured. Therefore it is also necessary to provide the user with a means to selectively display salient features of a performance topology. For example, as shown in Fig. 10.5, we can exclude the points with middling values and display only the outliers. In addition to clarifying the structure of performance behavior with respect to machine topology, this also helps to highlight potential topologically sensitive performance problems such as oversubscribed or underutilized nodes.

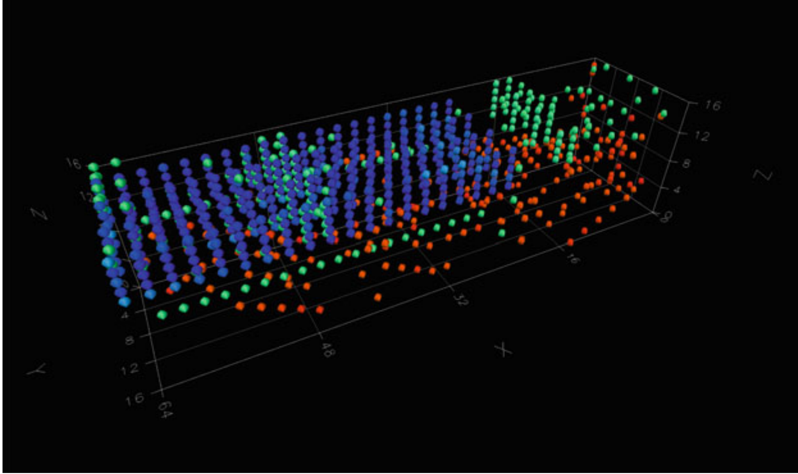

Fig. 10.5 16k-core 3D topology map of Sweep function in the Sweep 3d application on BG/L. Only nodes with high (*red*) and low (*blue or green*) values are shown

Visualization UI design is concerned with how the visualization will be controlled. The key insight here is to have the UI play a role in "binding" data model variables used in the layout specification. This approach implements the functionality present in the current ParaProf views, where the user is free to select events and metrics to be applied in the visualization as inputs to layout formulae. However, for large performance proles of many threads/processes, the specied layout can result in a dense visualization that obscures internal structures. The current ability to zoom and rotate the topology in the UI partially ameliorates this issue. Our model for visualization UI further allows more sophisticated ltering techniques.

10.6 Eclipse Integration

Tau has supported integration with the Eclipse IDE [6] for some time now. We wrap the standard command-line based TAU analysis workflow in the graphical Eclipse interface. This simplifies the process of performance analysis, ties it more closely with the overall development cycle in Eclipse and exposes the capabilities of the TAU performance system through the UI. By integrating with the Eclipse Parallel Tools Platform (PTP) [7] we also take advantage of new IDE-based parallel and HPC development capabilities.

During the development of the TAU plug-ins it became evident that much of the work being done was applicable to other performance analysis systems and similar command-line based tools. At a high level, such tools typically operate on some combination of compilation, execution, and analysis steps and their inputs are similar to those of TAU. To take advantage of this congruity, the workflow

logic and User Interface (UI) elements, which were initially hard-coded into the original TAU plug-ins, were converted to a generalized API. Additionally, to make the system more easily accessible and extensible, we developed an XML interface for defining both performance tool workflows and their UIs within Eclipse/PTP. The result is the general-purpose External Tools Framework (ETFw). ETFw allows both tool and application developers to integrate performance analysis systems into an Eclipse environment without the effort and expertise that are required to develop new Eclipse plug-ins. In fact, XML workflow definitions for external performance tools can be added or updated without restarting the Eclipse platform.

Although ETFw generalized much of the hard-coded behavior of the original TAU plug-ins, advanced TAU-specific functionality remains encapsulated within a plugin structure. This functionality includes PAPI hardware counter selection, as shown in the UI example in Fig. 10.6. However, the advanced API extension points used by the TAU-specific plug-ins are available to other tools that require logic or UI elements that are too application-specific for the ETFw to handle.

The ETFw's XML workflow format consists of three fundamental elements, which define the compilation, execution, and analysis steps of the workflow. The order, number, and presence of these steps may vary depending on the intent of the workflow and the employed analysis tools.

- The compilation step assigns compiler commands to be used for the relevant programming languages.
- The execution step defines commands to be composed with the target executable, if any. This covers tools such as Valgrind that take the target application as an input argument.
- The analysis step defines a series of commands that may be run on any data generated during program execution.

Each application or tool defined in an XML workflow may have its command and input parameters specified in the XML file. Alternatively, command-line options may be specified, which will appear in the Eclipse/PTP UI, where the user may enable, disable, and assign values to them dynamically. Once an XML workflow has been composed, it can be modified easily to suit different use cases. It also can be distributed to other users, who can easily load it into their Eclipse/PTP environments and run their applications with the performance analysis workflow, without concerning themselves with tool invocation details. In addition to adding support for arbitrary performance tools, the ETFw's abstraction of performance tool operations simplifies the implementation of more complicated workflows. This includes workflows that require multiple executions of the target application, such as parametric studies [8]. ETFw is now part of the PTP plug-in and is, thus, available to all users with a current Eclipse/PTP IDE configuration.

Formerly TAU and other external tools could only be invoked locally. Much of the power of the PTP lies in its ability to manage development on remote systems. We have extended the ETFw to take advantage of the PTPs remote capabilities. Now a user operating on a local workstation can use the Eclipse environment to build, execute and manipulate performance data from applications on remote,

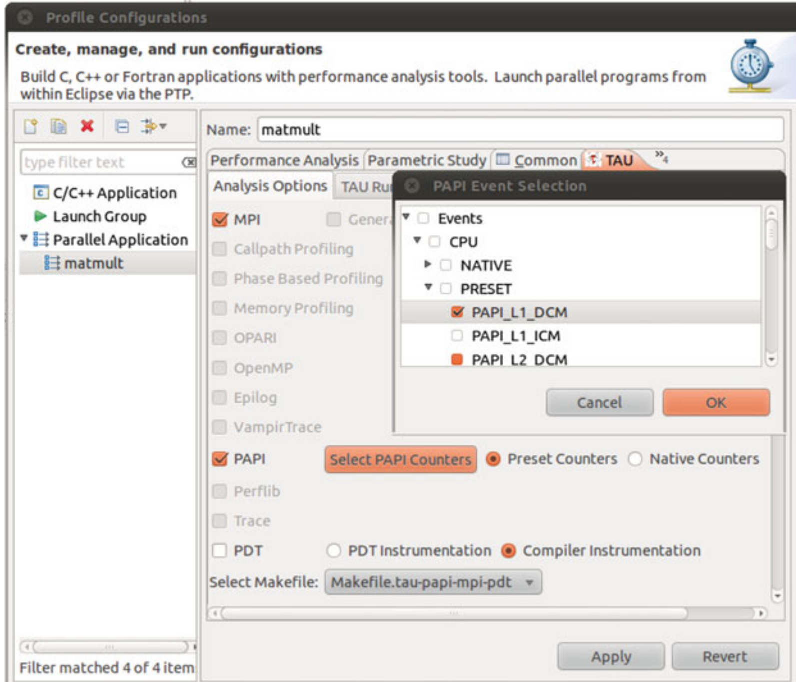

Fig. 10.6 Configuration selection UI for the TAU Eclipse plugin, including popup window for selecting PAPI hardware counters

production machines. This has the potential to make the typical HPC application development cycle much easier and more closely aligned with the standards of conventional software development.

10.7 Conclusion

Parallel performance toolsets must continue to improve to keep abreast of HPC innovations and technology evolution. Five recent advances in the TAU performance system were presented and discussed with respect to various HPC performance measurement and analysis requirements. All of them are or will be part of the TAU open source distribution.

Acknowledgements This research was conducted at the University of Oregon under grants DE-SC0001777, DEFG02-08ER25846, and DE-SC0006723 from the Department of Energy, Office of Science. Resources from NERSC were utilized in this work. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurr. Comput. Pract. Exp. **22**, 685–701 (2010)
2. Bell, R., Malony, A.D., Shende, S.: A portable, extensible, and scalable tool for parallel performance profile analysis. Proceedings of the EUROPAR 2003 Conference, Klagenfurt, Austria, pp. 17–26 (2003)
3. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**(3), 189–204 (2000)
4. Brunst, H., Hackenberg, D., Juckeland, G., Rohling, H.: Comprehensive performance tracking with vampir 7. In: Muller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Tools for High Performance Computing 2009, pp. 17–29. Springer, Berlin/Heidelberg (2010)
5. Dietrich, R., Ilsche, T., Juckeland, G.: Non-intrusive performance analysis of parallel hardware accelerated applications on hybrid architectures. In: First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2010), pp. 135–143. IEEE Computer Society, Los Alamitos, CA (2010)
6. Eclipse Foundation: Eclipse Integrated Development Environment. http://www.eclipse.org (2009)
7. Eclipse PTP Project: Eclipse parallel tools platform. http://www.eclipse.org/ptp (2009)
8. Huck, K., Spear, W., Malony, A., Shende, S., Morris, A.: Parametric studies in eclipse with TAU and perfExplorer. In: Proceedings of the Workshop on Productivity and Performance (PROPER08), EuroPar 2008 Workshops – Parallel Processing 5415/2009, Las Palmas de Gran Canaria, Spain, pp. 283–294 (2008)
9. Math Expression String Parser (MESP): http://expression-tree.sourceforge.net/ (2004)
10. Morris, A., Malony, A.D., Shende, S., Huck, K.: Design and implementation of a hybrid parallel performance measurement system. In: International Conference on Parallel Processing, San Diego, pp. 492–501 (2010)
11. National Center for Supercomputing Applications: University of Illinois at Urbana-Champaign, Perfsuite. http://perfsuite.ncsa.uiuc.edu/ (2011)
12. Shende, S., Malony, A.D.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)
13. Shende, S., Malony, A.D., Spear, W., Schuchardt, K.: Characterizing i/o performance using the tau performance system. In: International Conference on Parallel Proceedings, Parco Exascale Mini-symposium, Ghent, Belgium (2011)
14. Spear, W., Malony, A.D., Lee, C.W., Biersdorff, S., Shende, S.: An approach to creating performance visualizations in a parallel profile analysis tool. In: Workshop on Productivity and Performance (PROPER 2011), Bordeaux, France Aug, 2011
15. Szebenyi, Z., Gamblin, T., Martin, S., de Supinski, B.R., Wolf, F., Wylie, B.J.: Reconciling sampling and direct instrumentation for unintrusive call-path profiling of MPI programs. In: Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2011, pp. 637–648. IEEE Computer Society, Anchorage, AK (2011)