PERFORMANCE TECHNOLOGY FOR COMPLEX PARALLEL AND DISTRIBUTED SYSTEMS

Allen D. Malony, Sameer Shende
Department of Computer and Information Science
University of Oregon, Eugene, OR, USA
{malony,sameer}@cs.uoregon.edu

Abstract The ability of performance technology to keep pace with the growing complexity of parallel and distributed systems will depend on robust performance frameworks that can at once provide system-specific performance capabilities and support high-level performance problem solving. The TAU system is offered as an example framework that meets these requirements. With a flexible, modular instrumentation and measurement system, and an open performance data and analysis environment, TAU can target a range of complex performance scenarios. Examples are given showing the diversity of TAU application.

Keywords: performance tools, complex systems, instrumentation, measurement, analysis

1. INTRODUCTION

Modern parallel and distributed computing systems present both a complex execution environment and a complex software environment that target a broad set of applications with a range of requirements and goals, including high-performance, scalability, heterogeneous resource access, component interoperability, and responsive interaction. The execution environment complexity is being fueled by advances in processor technology, shared memory integration, clustering architectures, and high-speed inter-machine communication. At the same time, sophisticated software systems are being developed to manage the execution complexity in a way that makes available the potential power of parallel and distributed platforms to the different application needs.

Fundamental to the development and use of parallel and distributed systems is the ability to observe, analyze, and understand their performance at different levels of system implementation, with different performance data and detail,

for different application types, and across alternative system and software environments [5]. However, the growing complexity of parallel and distributed systems challenge the ability of performance technologists to produce tools and methods that are at once robust and ubiquitous. On the one hand, the sophistication of the computing environment demands a tight integration of performance observation (instrumentation and measurement) technology optimized to capture the requisite information about the system under performance access, accuracy, and granularity constraints. Different systems will require different observation capabilities and technology implementations specific to system features. Otherwise restricting technology to only a few performance observation modes severely limits performance problem solving in these complex environments. On the other hand, application development environments present programming abstractions that hide the complexity of the underlying computing system, and are mapped onto layered, hierarchical runtime software optimized for different system platforms. While providing application portability, a programming paradigm also defines an implicit model of performance that is made explicit in a particular system context. System-specific performance data must be mapped to abstract, high-level views appropriate to the performance model. The difficult problem is to provide such a performance abstraction uniformly across the different computing systems where the programming paradigm may be applied. This requires not only a rich set of observation capabilities that can provide consistent relevant performance information, but a high degree of flexibility in how tools are configured and integrated to access and analyze this information. Without this ability, common performance problem solving methodologies and tools that support them will not be available.

In this paper, we propose an approach to performance technology development for complex parallel and distributed systems based on a general complex systems computation model and a modular performance observation and analysis framework. The computation model, discussed in Section 2, defines a hierarchical execution architecture reflecting dominant features of modern systems and the layers of software available. In Section 3, we present the TAU performance framework as an example of a flexible, configurable, and extensible performance tool system for instrumentation, measurement, and analysis. TAU's ability to address complex system performance requirements is demonstrated in Section 4 using examples drawn from MPI, multi-threading, and combined task/data parallelism performance studies. We conclude the paper with an outlook towards open performance technology as a plan for developing next-generation performance tools.

2. A GENERAL COMPUTATION MODEL

To address the dual goals of performance technology for complex systems – robust performance capabilities and widely available performance problem solving methodologies – we need to contend with problems of system diversity while providing flexibility in tool composition, configuration, and integration. One approach to address these issues is to focus attention on a sub-class of computation models and performance problems as a way to restrict the performance technology requirements. The obvious consequence of this approach is limited tool coverage. Instead, our idea is to define an abstract computation model that captures general architecture and software execution features and can be mapped straightforwardly to existing complex system types. For this model, we can target performance capabilities and create a tool framework that can adapt and be optimized for particular complex system cases.

Our choice of general computation model must reflect real computing environments. The computational model we target was initially proposed by the HPC++ consortium [3]. In this model, a *node* is defined as a physically distinct machine with one or more processors sharing a physical memory system (i.e., a shared memory multiprocessor). A node may link to other nodes via a protocol-based interconnect, ranging from proprietary networks, as found in traditional MPPs, to local- or global-area networks. A *context* is a distinct virtual address space residing within a node. Multiple contexts may exist on a single node. Multiple *threads* of execution, both user and system level, may exist within a context; threads within a context share the same virtual address space.

3. TAU FRAMEWORK

The computation model above is general enough to apply to many high-performance architectures as well as to different parallel programming paradigms. Particular instances of the model and how it is programmed defines requirements for performance tool technology. For any performance problem, a performance framework to address the problem should incorporate:

- an *instrumentation model* defining how and when performance information is made available;
- a *performance measurement model* defining what performance information is captured and in what form;
- an *execution model* that relates measured events with each other;
- a *data analysis model* specifying how data is to be processed;
- a *presentation model* for performance viewing; and
- an *integration model* describing how performance tool components are configured and integrated.

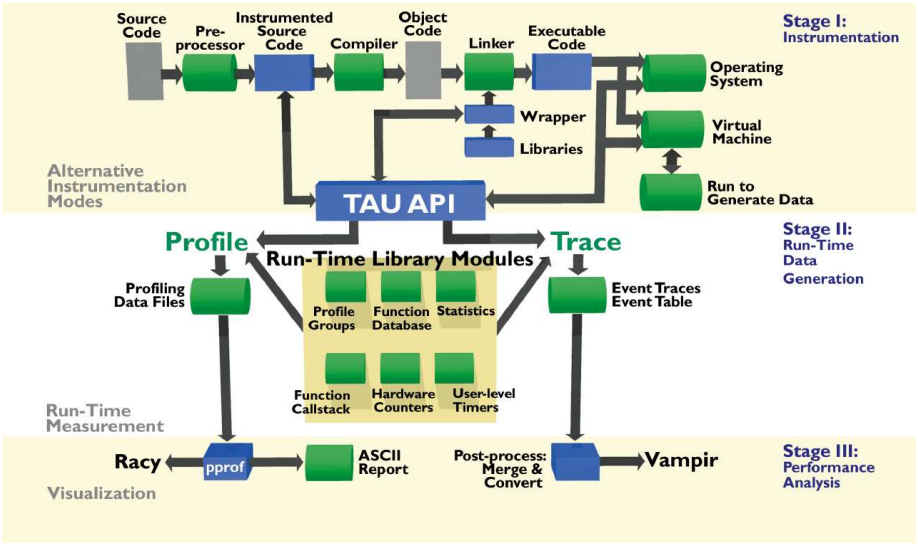

Figure 5.1 Architecture of TAU

We have developed the TAU performance framework as an integrated toolkit for performance instrumentation, measurement, and analysis for parallel, multithreaded programs that attempts to target the general complex system computation model while allowing flexible customization for system-specific needs.

The TAU performance framework [10] is shown in Figure 5.1. It is composed of instrumentation, measurement, and visualization phases. TAU supports a flexible instrumentation model that allows the user to insert performance instrumentation calling the TAU measurement API at several levels of program compilation and execution stages. The instrumentation identifies code segments, provides mapping abstractions, and supports multi-threaded and message passing parallel execution models. Instrumentation can be inserted manually, or automatically with a source-to-source translation tool [13]. When the instrumented application is compiled and executed, profiles or event traces are produced. TAU can use wrapper libraries to perform instrumentation when source code is unavailable for instrumentation. Instrumentation can also be inserted at runtime, using the dynamic instrumentation system DynInst [7], or at the virtual machine level, using language supplied interfaces such as the Java Virtual Machine Profiler interface [11].

The instrumentation model interfaces with the measurement model. TAU's measurement model is sub-divided into a high-level performance model that determines how events are processed and a low-level measurement model that determines what system attributes are measured. The measurement captures data for functions, methods, basic blocks, and statement execution. Profiling

and tracing are the two measurement choices that TAU provides. The API lets measurement *groups* be defined for organizing and controlling instrumentation. The measurement library also supports the mapping of low-level execution measurements to high-level execution entities (e.g., data parallel statements) so that performance data can be properly assigned. Performance experiments can be composed from different measurement modules, including ones that can measure the wall-clock time, the cpu time, or processor specific activity using non-intrusive hardware performance monitors available on most modern processors; TAU can access both PCL [9] and PAPI [14] portable hardware counter interfaces. Based on the composition of modules, an experiment could easily be configured to measure the profile that shows the inclusive and exclusive counts of secondary data cache misses associated with basic blocks such as routines, or a group of statements. By providing a flexible measurement infrastructure, a user can experiment with different attributes of the system and iteratively refine the performance of a parallel application.

The TAU data analysis and presentation models are open. Although TAU comes with both text-based and graphical tools to visualize the performance data collected in the previous stage, it provides bridges to other third-party tools such as Vampir [8] for more sophisticated analysis and visualization. The performance data format is documented and TAU provides tools that illustrate how this data can be converted to other formats.

An important component of the performance model presented in a tool is how its integration model provides composition and integration of its different components. The modules must provide well defined interfaces that are easy to extend. The nature and extent of co-operation between modules that may be vertically and horizontally integrated in the distinct layers defines the degree of flexibility of the measurement system.

4. SELECTED SCENARIOS

Our premise is that TAU can provide a robust and widely applicable performance technology framework for complex parallel and distributed systems. This section presents brief selected performance scenarios that demonstrate that TAU can offer effective technology across complex systems types. We begin with an MPI example showing how communication events are instrumented and performance measured and visualized with respect to other application events. We then discuss multi-threaded applications and the techniques we have developed for Java. The main point to highlight is TAU's ability to support different high-level performance problem solving requirements via system specific instrumentation, measurement, and analysis.

4.1 Distributed Systems and MPI

A common approach to enabling instrumentation in libraries is to define a wrapper library that encapsulates the functionality of the underlying library by inserting instrumentation calls before and after calls to the native routines. The MPI Profiling Interface [6] is a good example of this approach. This interface allows a tool developer to interface with MPI calls without modifying the application source code, and in a portable manner that does not require a vendor to supply the proprietary source code of the library implementation. A performance tool can provide an interposition library layer that intercepts calls to the native MPI library by defining routines with the same name (such as *MPI_Send*). These routines can then call the name-shifted native library routines provided by the MPI profiling interface (such as *PMPI_Send*). Wrapped around the call is performance instrumentation. The exposure of routine arguments allows the tool developer to track the size of messages, identify message tags or invoke other native library routines, for example, to track the sender and the size of a received message, within a wild-card receive call.

Requiring that such profiling hooks be provided in the standardized library before an implementation is considered "compliant", forms the basis of an excellent model for developing portable performance profiling tools for the library. TAU and several other tools (e.g., Upshot [1] and Vampir [8]) use the MPI profiling interface for tracing. However, TAU can also utilize a rich set of measurement modules that allow profiles to be captured with various types of performance data, including system and hardware data. In addition, TAU's performance grouping capabilities allows MPI event to be presented with respect to high-level categories such as send and receive types.

4.2 Multi-Threaded Systems and Java

Multi-threaded systems and applications present a more complex environment for performance tools due to the different forms and levels of threading and the greater need for efficient instrumentation. How to determine thread identity, how to store per-thread performance data, and how to provide synchronized and consistent update and access to the data are some of the questions that must be addressed. TAU provides modules that interface with system-specific thread libraries and provide member functions for thread registration, thread identification, and mutual exclusion for locking and unlocking the performance data. This allows the measurement system to work with different thread packages such as pthreads, Windows threads, Java threads, as well as special-purpose thread libraries such as SMARTS [15] and Tulip [2], while maintaining a common measurement model. Because TAU targets a general threading model, it can extend its common thread layer to provide well-defined core functionality for each new thread system.

We chose the Java language to demonstrate TAU's application in multi-thread systems since it utilizes both user-level and system-level threads and involves the additional complexity of virtual machine execution. The Java 2 virtual machine provides event callback hooks in the form of the Java Virtual Machine Profiler Interface (JVMPI) [11]. TAU uses JVMPI for performance instrumentation and measurement. The TAU measurement library is compiled into a dynamic shared object which is loaded in the address space of the virtual machine. An initialization routine specifies a mapping of events that are of interest to the performance system and registers a TAU interface that will be called when the events occur. When an event is triggered, event specific information is passed to the TAU interface routine by the virtual machine. TAU identifies the thread in which the event takes place and uses the Java thread interface to maintain per-thread performance data. TAU classifies all method names and their signatures into higher level profile group names. In Figure 5.2 we see the profile of per-thread execution for different methods and groups. Notice that some of the threads (0-3) are performing system functions for the JVM while others (4, 5, and 9) are performing user tasks. Profile (as shown) and tracing performance measurements can be made and reported.

4.3 Hybrid Parallel Systems

Increasingly, scalable parallel systems are being designed as clusters of shared memory multi-processors (SMPs), with MPI or some other inter-process communication paradigm used for message passing between SMP nodes, and thread-based shared memory programming used within the SMP. Runtime systems are built to hide the intricacies of efficient communication, presenting a compiler backend or an application programmer with a set of well-defined, portable interfaces. Performance measurement and analysis tools must embed the hierarchical, hybrid execution model of the application within their performance model. Because TAU supports a general parallel computation model, it can configure the measurement system to capture both thread and communication performance information. However, this information must be mapped to the programming model. We have used TAU to investigate task and data parallel execution in the Opus/HPF programming system [4]. Figure 5.3 shows a Vampir display of TAU traces generated from an application written using HPF for data parallelism and Opus for task parallelism. The HPF compiler produces Fortran 90 data parallel modules which execute on multiple processes. The processes interoperate using the Opus runtime system built on MPI and pthreads. In systems of this type, it is important to be able to see the influence of different software levels. TAU is able to capture performance data at different parts of the Opus/HPF system exposing the bottlenecks within and between levels.

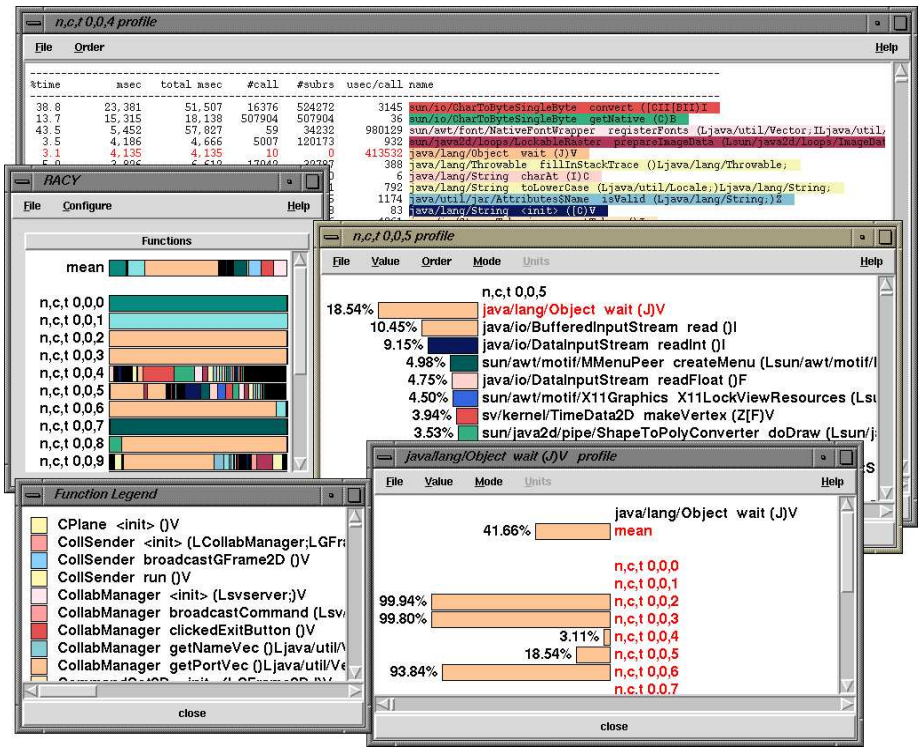

Figure 5.2 TAU profiles a multi-threaded Java visualization application using JVMPI

5. CONCLUSIONS

To be at once robust and ubiquitous, TAU attempts to solve performance technology problems at levels where performance analysis system solutions can be configured and integrated to target specific performance problem solving needs. TAU has been developed based on the principle that performance technology should be open, easy to extend, and able to leverage external functionality. The complex system case studies presented here is but a small sample of the range of TAU's potential application [12].

In rapidly evolving parallel and distributed systems, performance technology can ill-afford to stand still. A performance technologist always operates under a set of constraints as well as under a set of expectations. While performance evaluation of a system is directly affected by what constraints the system imposes on performance instrumentation and measurement capabilities, the desire for performance problem solving tools that are common and portable, now and into the future, suggests that performance tools hardened and customized for a particular system platform will be short-lived, with limited utility. Similarly, performance tools designed for constrained parallel execution models

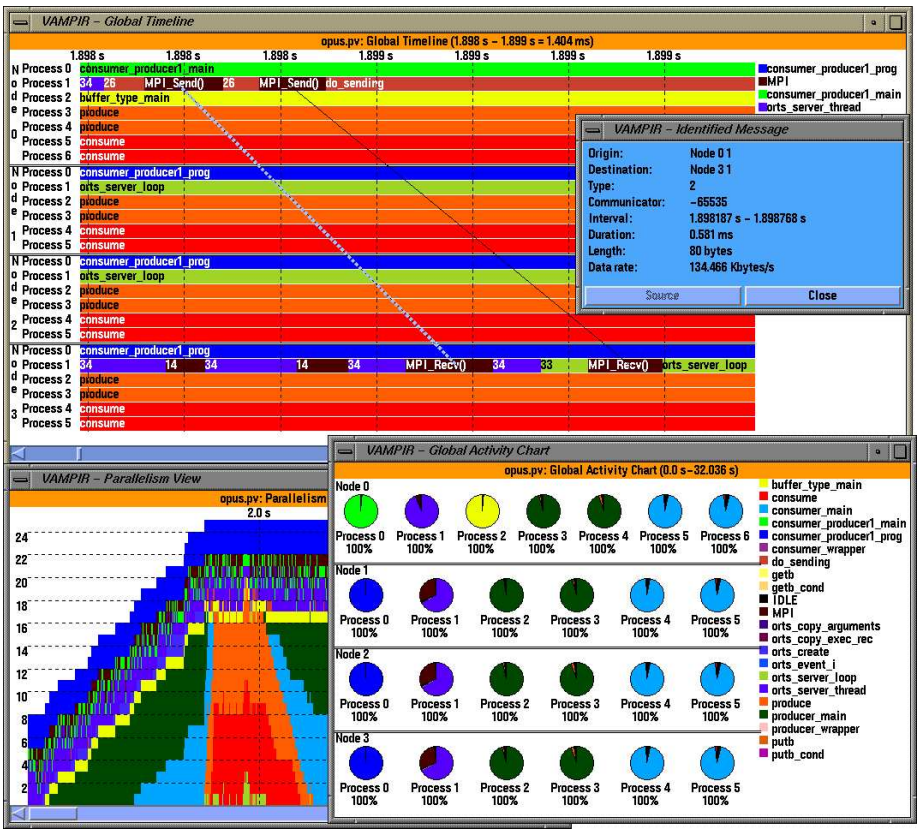

Figure 5.3 Vampir displays for TAU traces of an Opus/HPF application using MPI and pthread

will likely have little use to more general parallel and distributed computing paradigms. Unless performance technology evolves with system technology, a chasm will remain between the users expectations and the capabilities that performance tools provide. The challenge for the TAU system in the future is to maintain a highly configurable tool architecture while not arbitrarily enforcing constraining technology boundaries.

Acknowledgments

This work was supported in part by the U.S. Department of Energy, DOE2000 grant #DEFC0398ER259986.

References

[1] Argonne National Laboratory, "The Upshot program visualization system," URL: http://www-fp.mcs.anl.gov/l̃usk/upshot/.

[2] P. Beckman, D. Gannon, "Tulip: A Portable Run-Time System for Object Parallel Systems," Proceedings of the 10th International Parallel Processing Symposium, August 1996.

[3] HPC++ Working Group, "HPC++ White Papers," Technical Report TR 95633, Center for Research on Parallel Computation, 1995.

[4] E. Laure, P. Mehrotra, H. Zima, "Opus: Heterogeneous Computing With Data Parallel Tasks," Parallel Processing Letters, 9(2):275–289, June 1999.

[5] A. Malony, "Tools for Parallel Computing: A Performance Evaluation Perspective," Handbook on Parallel and Distributed Processing, (Eds. J. Blazewicz, K. Ecker, B. Plateau, D. Trystram); Springer, pp. 342–363, 2000.

[6] Message Passing Interface Forum, "MPI: A Message Passing Interface Standard," International Journal of Supercomputer Applications, Special issue on MPI. 8:3/4, 1994.

[7] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. "The Paradyn Parallel Performance Measurement Tools", IEEE Computer, 28(11):37–46, Nov. 1995.

[8] Pallas, "VAMPIR - Visualization and Analysis of MPI Resources," 1998. URL: http://www.pallas.de/pages/vampir.html.

[9] Research Center Juelich GmbH, "PCL - The Performance Counter Library," URL: http://www.fz-juelich.de/zam/PCL/.

[10] S. Shende, A. Malony, J. Cuny, K. Lindlan, P. Beckman, S. Karmesin, "Portable Profiling and Tracing for Parallel Scientific Applications using C++", Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 134–145, ACM, August 1998.

[11] Sun Microsystems, "Java Virtual Machine Profiler Interface (JVMPI)," URL: http://java.sun.com/products/jdk/1.3/docs/guide/jvmpi/jvmpi.html.

[12] University of Oregon, "Tuning and Analysis Utilities," URL: http://www.cs.uoregon.edu/research/paracomp/tau/.

[13] University of Oregon, "Program Database Toolkit," URL: http://www.cs.uoregon.edu/research/paracomp/pdtoolkit.

[14] University of Tennessee, "PerfAPI - Performance Data Standard and API," URL: http://icl.cs.utk.edu/projects/papi/.

[15] S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, S. Smith, "SMARTS: Exploiting Temporal Locality and Parallelism through Vertical Execution," Los Alamos National Laboratory, Technical Report LA-UR-99-16, 1999.

[16] Sun Microsystems Inc. "The JAVA HotSpot Performance Engine Architecture," Sun Microsystems White Paper, April 1999. http://java.sun.com/products/hotspot/whitepaper.html