A Runtime Monitoring Framework for the TAU Profiling System

T. Sheehan, A. Malony, and S. Shende

Computational Science Institute, Department of Computer and Information Science,
University of Oregon, Eugene OR 97403
{sheehan, malony, sameer}@cs.uoregon.edu

Abstract. Applications executing on complex computational systems provide a challenge for the development of runtime performance monitoring software. We discuss a computational model, application monitoring, data access models, and profiler functionality. We define data consistency within and across threads as well as across contexts and nodes. We describe the TAU runtime monitoring framework which enables on-demand, low-interference data access to TAU profile data and provides the flexibility to enforce data consistency at the thread, context or node level. We present an example of a Java-based runtime performance monitor utilizing the framework.

Keywords: monitor, runtime data access, performance monitoring, parallel execution, performance tools, runtime interaction, Java, TAU, multi-threaded.

1 Introduction

The building of complex software systems is increasingly targeted to high- performance computing architectures that support thread-based parallel execution within a shared memory context and process-based (component-based) distributed execution across multiple physical nodes [4, 7, 10]. Object-oriented computing is particularly suited to such architectures because it can naturally capture thread abstractions and extend the object interaction paradigm to distributed environments [4, 2, 9]. While rich programming frameworks and layered middleware systems help tame the complexity of software built for next-generation high-performance environments [16, 11], the observation of program operation and performance is critical to understanding and improving code efficiency.

Most monitoring systems built for parallel system environments are created for a specific purpose (e.g., performance measurement [8], debugging [13], or computational steering [3]), and with a specific computational model in mind. As a result, the monitoring system design and implementation tends to reflect decisions concerning the requirements for system observation, the type and level of instrumentation, the online accessibility to monitored information, the degree of consistency with respect to system state, and the acceptability of monitoring costs. It is difficult to construct a *monitoring framework* that is flexible enough to apply in different computational contexts and in which monitoring parameters can be selectively controlled. It is not simply an issue of monitor portability. The framework should support the construction of monitor software that operates in a manner consistent with the parallel software paradigm and its execution without excessively imposing it own constraints.

In this paper, we describe the TAU monitoring framework and its use in observing the performance of parallel applications built for complex, high- performance computing architectures. The framework instantiates a monitoring system that augments the application and system software with monitor interaction components and distributed monitor communication and control mechanisms. This monitor infrastructure is intended to interface with different online instrumentation facilities, extending the range of monitor applications. Integrating monitoring capabilities with complex software requires knowledge of the monitor's influence on program execution, and the monitoring framework should support a variety of alternative methods of data access. We discuss monitor interaction models and certain problems that can arise in the context of online access to performance data provided by the TAU profile library.

2 Model of Computation

A monitoring model must be based on an underlying computational model which must, in turn, reflect the real computing environment. The computational model we are targeting is illustrated in figure 1. A *node* is a physically distinct entity with one or more processors. 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 (e.g. the Internet). 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 single context. Threads within a context share the same address space.

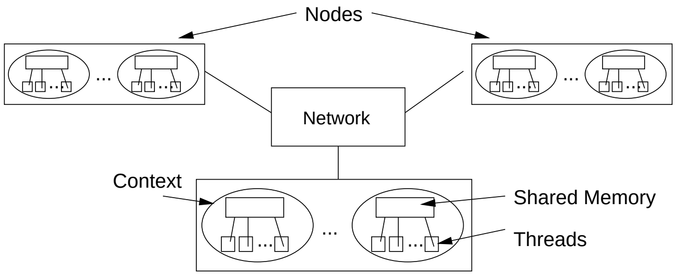

Fig. 1. The HPC++ computational model.

This computational model is general enough to apply to many high- performance architectures as well as to different parallel programming paradigms.

This enables us to consider a monitoring framework with respect to different computational model views. In addition, the model is the basis for HPC++ [4], a high-performance parallel and distributed object-oriented system that supports both task- and data-parallel programming paradigms.

3 Monitoring Systems

Monitors can be used for a wide variety of functions including data visualization, performance analysis, computational steering, and debugging. All of these activities utilize at least one of two basic operations on a executing application: the observation of the program's data and interaction with the executing code. Implicit in this description is a computational entity that provides access to the program's context(s) and mechanisms to involve itself with the executing code. We term such an entity a *monitor agent*. Also implicit is a *monitor client* which directs the observation and interaction.

3.1 Monitor Agent

In the HPC++ model of computation, the monitor agent must reside within the executing application's context(s) in order to access its memory. This agent may be a *direct (active) agent* or an *indirect (passive) agent*. A direct agent is invoked from code instrumentation to provide data access to the monitor client. An indirect agent, on the other hand, runs in a separate thread of execution, with full access to the context memory. Additional software resources are not needed to use a direct agent, but execution of the computation is directly impacted. An indirect agent can observe passively with less interference on the application's thread(s) of execution.

3.2 Monitor Client

The monitor client communicates with the monitor agent, directing the agent's actions and interpreting the monitored information. A client may reside within the same context as the agent or in a separate context. Normally, to avoid competing for the application's computational resources, the client resides outside of the context of the monitor agent and communicates with it in the same manner that other threads communicate across contexts.

4 Profiler Structure

A *profiler* keeps summary statistics of application execution based on the occurence of events[12]. Profiling systems generally track two different types of entities: user defined events and performance of blocks of code such as a routine or a group of related statements. For purposes of this paper we term such code blocks *functions*. The profiler maintains a database of user defined event counters. Whenever a user defined event is triggered, its counter is incremented. Count and execution time values are profiled for functions. Time is measured by elapsed wallclock time, or can be substituted by hardware performance counters to measure low-level CPU activity such as secondary data cache misses, or instructions issued. The profiler maintains a database of function information and an image of the callstack. The database includes the time consumed by each profiled function. The callstack image includes the starting time of each profiled function called. When a function completes and its entry is popped from the top of the callstack image, its starting time is subtracted from the current system time and this is added to the cumulative time for that function's database entry. At the end of a run, when the callstack image is emptied and the database has been updated, the data is written to a file where it can be accessed for post mortem processing and visualization. The profiler data structures are maintained on a per thread basis, and executing threads update their profiling data completely independent of one another.

5 System Snapshot

The post mortem profile analyses are derived from the data monitored during execution. A runtime monitor should provide similar views of the system while the application is executing. Such a view can be derived from the callstack image and count and time databases maintained by the profiler. We term this a *snapshot*, meaning a view of the state of the system at a given point in time.

5.1 A Consistent Snapshot

The concept of consistency is important in the representation of the state of the system. In post mortem analysis, the state of the system is simply the static state at program termination. At runtime, however, the system state is constantly evolving and its consistency must be taken into consideration when obtaining profile data.

If the monitor attempts to read the profile database or callstack while it is being updated by a thread, the information may not be consistent and the snapshot of the system may be erroneous. Thus, for a thread, a consistent snapshot requires that the acquisition of all the profile data be an atomic operation. This is accomplished by locking all of a thread's profile data before reading it.

The definition of a consistent snapshot can easily be extended to a context. Within a context a consistent snapshot is a set of images, one per thread, which represent the state of all the threads in the context at the same point in time. Acquiring a consistent snapshot within a context requires that the profile data for each thread be simultaneously locked and that a lock only be released once its thread's data is read. Likewise, a consistent snapshot of a multi-context system requires the simultaneous locking of data for all threads in all contexts, and the releasing of any lock only after its data is read.

5.2 Practical Considerations

As mentioned above, a truly consistent snapshot of multiple threads requires simulataneously locking the access of all threads to profiling data until the data is read. This could have a profound effect on application performance. While this doesn't stop the execution of the application, subsequent function entry and exit operation and event counter operation is blocked until the monitor agent has obtained the performance data. The possible impact on performance is serious, especially when one of the goals of the monitoring system is to minimize the extent of intrusion.

A less costly alternative to an absolutely consistent snapshot is to obtain an approximately consistent snapshot. This can be done by effectively looping over each thread, locking its profile data, reading its profile data, then releasing the lock. What is produced is analogous to a radial sweep radar image. The only portion of the display that is absolutely current is where the sweep is right now. The whole image, however, is approximately current. The advantage of this method is that it only has the potential to affect the execution of one process at a given time, and the maximum effect on a process is the time it takes to read that process's callstack and database images.

6 Runtime Data Access

Accessing data at runtime involves trade-offs between how current the data is, the timing of access, and the impact of the access on the application. We consider three passive agent models for runtime monitoring of application profile state: *push*, *push-pull*, and *pull*.

6.1 Push Model

Figure 2(A) shows the push model of data access to profile data [5, 6]. During execution, the callstack and profile database are updated. Periodically, the application pushes current profile data into a block of memory accessible to the monitor agent, a separately executing thread. The application signals the monitor agent that the data is available. *Synchronous* access by the monitor requires the application thread signalling the monitor to block until the monitor agent signals that data access is complete. If access to the data is *asynchronous*, the application thread continues while the monitor operates, later blocking until data access is finished. The application controls where and when state can be observed.

The advantage of the push model is that instrumentation in the profiler controls where and how profile data can be accessed, and when it is made available to the monitor agent. It is possible to maintain better consistency in this case, but can be intrusive on execution.

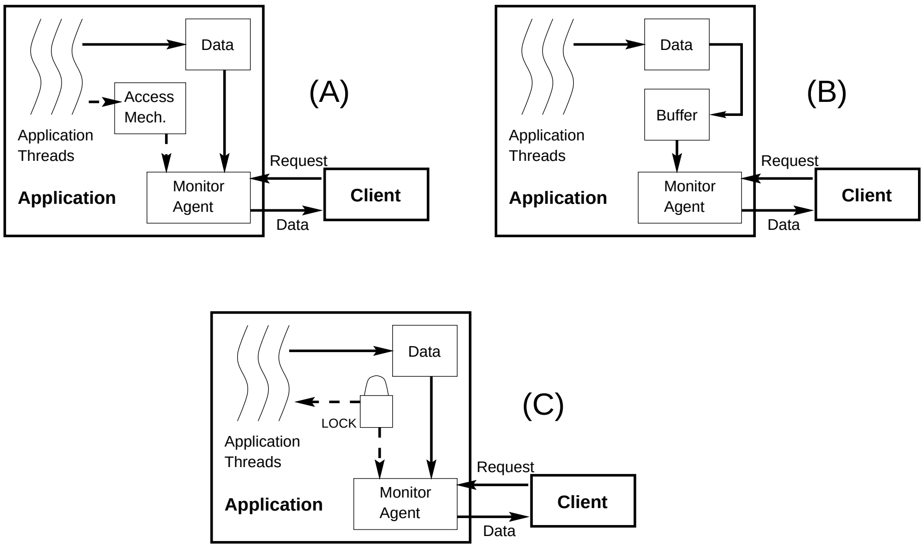

Fig. 2. Models of data access: (A) push model; (B) push-pull model; (C) pull model.

6.2 Push-Pull Model

The push-pull model is shown in figure 2(B) [15]. In this model, profile data is periodically pushed into a profile data buffer. The monitor agent has constant access to the the buffer and can pull data from it any time the application is not writing to the buffer. This solves two of the problems with the simple push model. First, the application thread does not have to block while the monitor agent accesses data. Second, the monitor agent can access the data in the buffer any time the application is not writing to the buffer. However, the data in the buffer does not reflect changes in the data as the program runs.

6.3 Pull Model

The final model is the pull model as shown in figure 2(C). In this model both the application thread and the monitor agent have access to the profile data at all times. When the application thread writes, or the monitor agent reads, a lock is used, insuring that the data does not change during the read. The disadvantage of this method is that while the monitor agent has the lock during a read, the application may block on a write to the database, impacting program performance.

7 TAU Runtime Monitoring Framework

We have have designed and implemented the *TAU runtime monitoring framework* that interfaces with the TAU profiling package [14, 1]. The goals of this framework are threefold: provide a consistent snapshot of performance data within an executing application; allow access to performance data at any point during execution; and to impose the smallest possible penalty on application performance.

7.1 Architecture

This system is an implementation of that depicted in figure 2(C). Each monitor agent in an application context provides access to a monitor client running in another context. For applications running across multiple contexts, an agent is present in each context. Hence, a monitoring program can create and manage multiple monitor clients as a means to interact with agents in multiple contexts. Each client object can access its agent's data on demand.

HPC++ [4] is used for both the monitor agent and client; monitoring clients are implemented as objects. It provides remote function invocation (RMFI) as well as data transport through global pointers across contexts. RMFI allows the client to remotely lock and unlock the application's data and direct the monitor agent to gather data while global pointers provide the client with remote data access.

7.2 Functionality

The HPC++ monitor agent is not limited to communication with only one client object. This allows for multiple client applications in multiple contexts to access the data from a single application. This allows for *collaborative monitoring* in which users on several different systems can use the framework to simultaneously observe the performance behavior of an application executing on a separate system.

Also, because the client is an object, any number of them can be created by a single application, so a single monitoring application can spawn a client object for each context in an executing application and simultaneously display data from all contexts as illustrated in figure 3.

7.3 Consistent Snapshots

This framework has the ability to lock the databases and callstack images in a context simultaneously and independently of when it reads the data. This provides the user with the ability to enforce consistency within a thread or among threads. In addition, all threads in all contexts can be locked at the same time within the limit of how long the system takes to issue one lock command for each context. Data can be read from each context before its thread is released, or from all contexts before any thread is released. The user can choose the tradeoff between the level of consistency of performance data and the potential performance penalty imposed on the executing application.

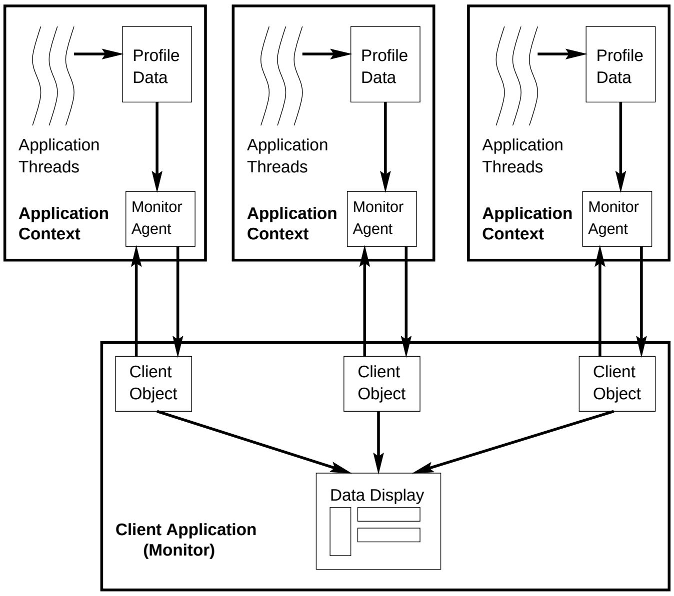

Fig. 3. Multiple clients accessing data from multiple application contexts.

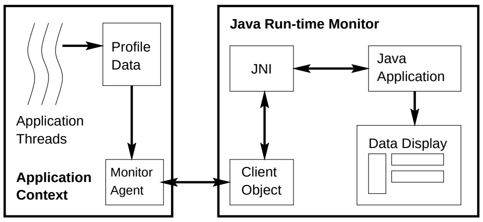

Fig. 4. Architecture of Java-based runtime monitor

8 Example Monitor Application

We have implemented a runtime TAU profile monitor based on the TAU monitoring framework. Shown in figure 4, the monitor uses a Java-based front end that interfaces with C++ through the Java Native Interface (JNI). This allows it to create a client object and attach to an executing application instrumented with TAU. At user-defined time intervals, the runtime monitor accesses application profile data through the monitor agent and displays profile data to the user.

Three different data displays are available. The *Exclusive Time* display (figure 5) provides a per thread overview of the exclusive time spent in each routine. A bar with contrasting colors graphically depicts the data, while a scrollable color key shows routines, exclusive times, and percentages. In the *Function Detail* display (figure 5), the functions from the database are shown for the selected

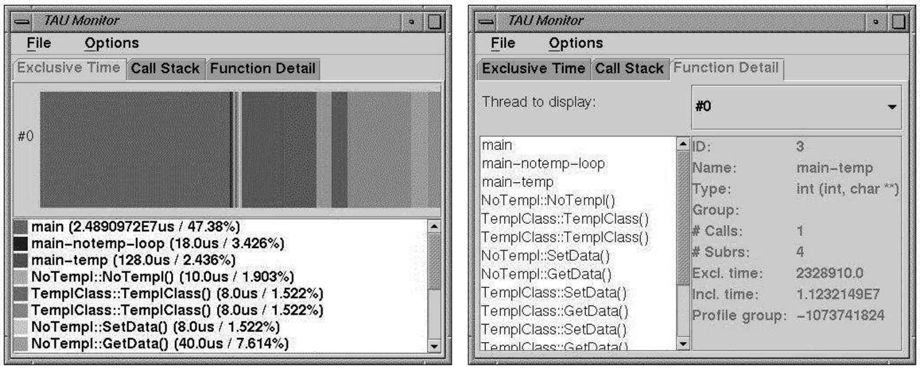

Fig. 5. TAU Runtime Monitor Exclusive Time and Function Detail displays.

thread. The function data is shown for the selected function. Also available is a callstack image.

9 Conclusions

Of the three monitoring models discussed, the TAU runtime monitoring framework is designed for the less-constraining, but more difficult to implement, pull model. Based on the general computation model in HPC++ and the portability of the TAU profiling system, a runtime monitor built from the framework can target complex monitoring needs for a diversity of parallel and distributed computing platforms.

Important features of the monitoring framework are its ability to support multiple levels of monitoring to capture consistent snapshots (single thread, multi-thread, or multi-context) and multiple monitor client interactions. This gives monitor developers the flexibility to build runtime monitor solutions specific to the observational and platform constraints. Additionally, the framework allows the developer to augment the monitor's functionality, such as to choose where monitored data analysis takes place, in the monitor agent or monitor client.

Our implementation of a Java-based runtime monitor based on this framework demonstrates its use for online access to TAU profiling data. We are presently pursuing the integration of this monitor in large-scale ASCI applications.

10 Acknowledgments

This work was supported by the Department of Energy DOE 2000 program (#DEFC0398ER259986) . We would like to thank the Los Alamos National Laboratory for their support. Matthew Sottile, and Chad Busche at the Univ. of Oregon, contributed to the implementation of the system.

References

1. Advanced Computing Laboratory (LANL): TAU Portable Profiling URL:http://www.acl.lanl.gov/tau. (1998)
2. Chandy, K., Kesselman, C.: CC++: A Declarative Concurrent Object Oriented Programming Notation, In Agha, G., Wegner, P., and Yonesawa (Eds.), Research Directions in Concurrent Object Oriented Programming, Cambridge, MA, MIT Press (1993) pp. 218–313.
3. Cuny, J., Dunn, R., Hackstadt, S., Harrop, C., Hersey, H., Malony, A., Toomey, D.: Building Domain-Specific Environments for Computational Science: A Case Study in Seismic Tomography, Intl. Jour. of Supercomputing Applications and High Performance Computing. Vol. 11 (3). (1997).
4. Gannon, D., Beckman, P., Johnson, E., Green, T., Levine, M.: HPC++ and the HPC++LIB Toolkit, Technical Report Department of Computer Science, Indiana University (1998).
5. Hackstadt, S., Harrop, C., Malony, A.: A Framework for Interacting with Distributed Programs and Data, In: Proc. of the Seventh Int'l Symp. on High Performance Distributed Computing 1998 (HPDC-7). IEEE, July (1998).
6. Hackstadt, S., Malony, A.: DAQV: Distributed Array Query and Visualization Framework, Journal of Theoretical Computer Science, special issue on Parallel Computing Vol. 196, No. 1-2, April (1998) pp. 289–317.
7. Laure, E., Mehrotra, P., Zima, H.: Opus: Heterogeneous Computing With Data Parallel Tasks, Technical Report TR 99-04, Institute for Software Technology and Parallel Systems, University of Vienna URL:http://www.par.univie.ac.at. (1999).
8. Miller, B., Callaghan, M., Cargille, J., Hollingsworth, J., Irvin, R., Karavanic, K., Kunchithapadam, K., Newhall, T.: The Paradyne Parallel Performance Measurement Tools, IEEE Computer. Vol. 28(11). Nov (1995).
9. OMG: CORBA/IIOP 2.2 Specification, URL:http://www.omg.org (1998).
10. OpenMP: OpenMP Fortran Interpretations Versions 1.0, URL:http://www.openmp.org (1999).
11. Reynders, J. et. al.: Pooma: A Framework for Scientific Simulation on Parallel Architectures, In: Wilson, G., Lu, P. (Eds.): Parallel Programming using C++, M.I.T. Press (1996) pp. 553–594.
12. Shende, S.: Profiling and Tracing in Linux, In (to appear) Proc. Second Extreme Linux Workshop, USENIX '99. (1999)
13. Shende, S., Cuny, J., Hansen, L., Kundu, J., McLaughry, S., Wolf, O.: Event and State Based Debugging in TAU: A Prototype, Proc. of ACM SIGMETRICS Symp. on Parallel and Distributed Tools. May (1996) pp. 21–30.
14. Shende, S., Malony, A. D., Cuny, J., Lindlan, K., Beckman, P., Karmesin, S.: Portable Profiling and Tracing for Parallel, Scientific Applications using C++, Proc. of ACM SIGMETRICS Symp. on Parallel and Distributed Tools. Aug (1998) pp. 134–145.
15. Shende, S., Malony, A. D., Hackstadt, S.: Dynamic Performance Callstack Sampling: Merging TAU and DAQV. In Kågström, B. and Dongarra, J. and Elmroth, E. and Waśniewski, J. (editors). Applied Parallel Computing, 4th International Workshop, PARA'98, Lecture Notes in Computer Science, No. 1541, Springer-Verlag, Berlin, pp. 515-520, June (1998).
16. The Staff, Advanced Computing Laboratory, Los Alamos National Laboratory: Taming Complexity in High-Performance Computing. White Paper. Accessible from URL:http://www.acl.lanl.gov/software. Nov (1998)