Performance Evaluation of Adaptive Scientific Applications using TAU

Sameer Shende[a*], Allen D. Malony[a], Alan Morris[a], Steven Parker[b], J. Davison de St. Germain[b]

[a]Performance Research Laboratory, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, USA

[b]Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT 84112, USA

1. Introduction

Fueled by increasing processor speeds and high speed interconnection networks, advances in high performance computer architectures have allowed the development of increasingly complex large scale parallel systems. For computational scientists, programming these systems efficiently is a challenging task. Understanding the performance of their parallel applications is equally daunting. To observe and comprehend the performance of parallel applications that run on these systems, we need performance evaluation tools that can map the performance abstractions to the user's mental models of application execution. For instance, most parallel scientific applications are iterative in nature. In the case CFD applications, they may also dynamically adapt to changes in the simulation model. A performance measurement and analysis system that can differentiate the phases of each iteration and characterize performance changes as the application adapts will enable developers to better relate performance to their application behavior. In this paper, we present new performance measurement techniques to meet these needs. In section 2, we describe our parallel performance system, TAU. Section 3 discusses how new TAU profiling techniques can be applied to CFD applications with iterative and adaptive characteristics. In section 4, we present a case study featuring the Uintah computational framework and explain how adaptive computational fluid dynamics simulations are observed using TAU. Finally, we conclude with a discussion of how the TAU performance system can be broadly applied to other CFD frameworks and present a few examples of its usage in this field.

2. TAU Performance System

Given the diversity of performance problems, evaluation methods, and types of events and metrics, the instrumentation and measurement mechanisms needed to support performance observation must be flexible, to give maximum opportunity for configuring performance experiments, and portable, to allow consistent cross-platform performance problem solving. The TAU performance system [1,4], is composed of instrumentation, measurement, and analysis

*This research was supported by the U.S. Department of Energy, Office of Science, under contracts DE-FG03-01ER25501 and DE-FG02-03ER25561, and University of Utah and LLNL DOE contract 2705056.

parts. It supports both profiling and tracing forms of measurements. TAU implements a flexible instrumentation model that permits a user to insert performance instrumentation hooks into the application at several levels of program compilation and execution. The C, C++, and Fortran languages are supported, as well as standard message passing (e.g., MPI) and multi-threading (e.g., Pthreads) libraries.

For instrumentation we recommend a dual instrumentation approach. Source code is instrumented automatically using a source-to-source translation tool, $tau\_instrumentor$, that acts as a pre-processor prior to compilation. The MPI library is instrumented using TAU's wrapper interposition library that intercepts calls to the MPI calls and internally invokes the TAU timing calls before and after. TAU source instrumentor can take a selective instrumentation file that lists the name of routines or files that should be excluded or included during instrumentation. The instrumented source code is then compiled and linked with the TAU MPI wrapper interposition library to produce an executable.

TAU provides a variety of measurement options that are chosen when TAU is installed. Each configuration of TAU is represented in a set of measurement libraries and a stub makefile to be used in the user application makefile. Profiling and tracing are the two performance evaluation techniques that TAU supports. Profiling presents aggregate statistics of performance metrics for different events and tracing captures performance information in timestamped event logs for analysis. In tracing, we can observe along a global timeline when events take place in different processes. Events tracked by both profiling and tracing include entry and exit from routines, interprocess message communication events, and other user-defined atomic events. Tracing has the advantage of capturing temporal relationships between event records, but at the expense of generating large trace files. The choice to profile trades the loss of temporal information with gains in profile data efficiency.

3. CFD Application Performance Mapping

Observing the behavior of an adaptive CFD application shows us several interesting aspects of its execution. Such applications typically involve a domain decomposition of the simulation model across processors and an interaction of execution phases as the simulation proceeds in time. Each iteration may involve a repartitioning or adaption of the underlying computational structure to better address numerical or load balance properties. For example, a mesh refinement might be done at iteration boundaries and information about convergence or divergence of numerical algorithms is detailed. Also, domain specific information such as the number of cells refined at each stage gives a user valuable feedback on the progress of the computation.

Performance evaluation tools must capture and present key application specific data and corelate this information to performance metrics to provide a useful feedback to the user. Presenting performance information that relates to application specific abstractions is a challenging task. Typically, profilers present performance metrics in the form of a group of tables, one for each MPI task. Each row in a table represents a given routine. Each column specifies a metric such as the exclusive or inclusive time spent in the given routine or the number of calls executed. This information is typically presented for all invocations of the routine. While such information is useful in identifying the routines that contribute most to the overall execution time, it does not explain the performance of the routines with respect to key application phases. To address this shortcoming, we provide several profiling schemes in TAU.

3.1. Static timers

These are commonly used in most profilers where all invocations of a routine are recorded. The name and group registration takes place when the timer is created (typically the first time a routine is entered). A given timer is started and stopped at routine entry and exit points. A user defined timer can also measure the time spent in a group of statements. Timers may be nested but they may not overlap. The performance data generated can typically answer questions such as: *what is the total time spent in* $MPI\_Send()$ *across all invocations?*

3.2. Dynamic timers

To record the execution of each invocation of a routine, TAU provides dynamic timers where a unique name may be constructed for a dynamic timer for each iteration by embedding the iteration count in it. It uses the start/stop calls around the code to be examined, similar to static timers. The performance data generated can typically answer questions such as: *what is the time spent in the routine* $foo()$ *in iterations 24, 25, and 40?*

3.3. Static phases

An application typically goes through several phases in its execution. To track the performance of the application based on phases, TAU provides static and dynamic phase profiling. A profile based on phases highlights the context in which a routine is called. An application has a default phase within which other routines and phases are invoked. A phase based profile shows the time spent in a routine when it was in a given phase. So, if a set of instrumented routines are called directly or indirectly by a phase, we'd see the time spent in each of those routines under the given phase. Since phases may be nested, a routine may belong to only one phase. When more than one phase is active for a given routine, the closest ancestor phase of a routine along its callstack is its phase for that invocation. The performance data generated can answer questions such as: *what is the total time spent in* $MPI\_Send()$ *when it was invoked in all invocations of the* IO *(*$IO => MPI\_Send()$*) phase?*

3.4. Dynamic phases

Dynamic phases borrow from dynamic timers and static phases to create performance data for all routines that are invoked in a given invocation of a phase. If we instrument a routine as a dynamic phase, creating a unique name for each of its invocations (by embedding the invocation count in the name), we can examine the time spent in all routines and child phases invoked directly or indirectly from the given phase. The performance data generated can typically answer questions such as: *what is the total time spent in* $MPI\_Send()$ *when it was invoked directly or indirectly in iteration 24?* Dynamic phases are useful for tracking per-iteration profiles for an adaptive computation where iterations may differ in their execution times.

3.5. Callpaths

In phase-based profiles, we see the relationship between routines and parent phases. Phase profiles do not show the calling structure between different routines as is represented in a callgraph. To do so, TAU provides callpath profiling capabilities where the time spent in a routine along an edge of a callgraph is captured. Callpath profiles present the full flat profiles of routines (or nodes in the callgraph), as well as routines along a callpath. A callpath is represented syntactically as a list of routines separated by a delimiter. The maximum depth of a callpath is controlled by an environment variable.

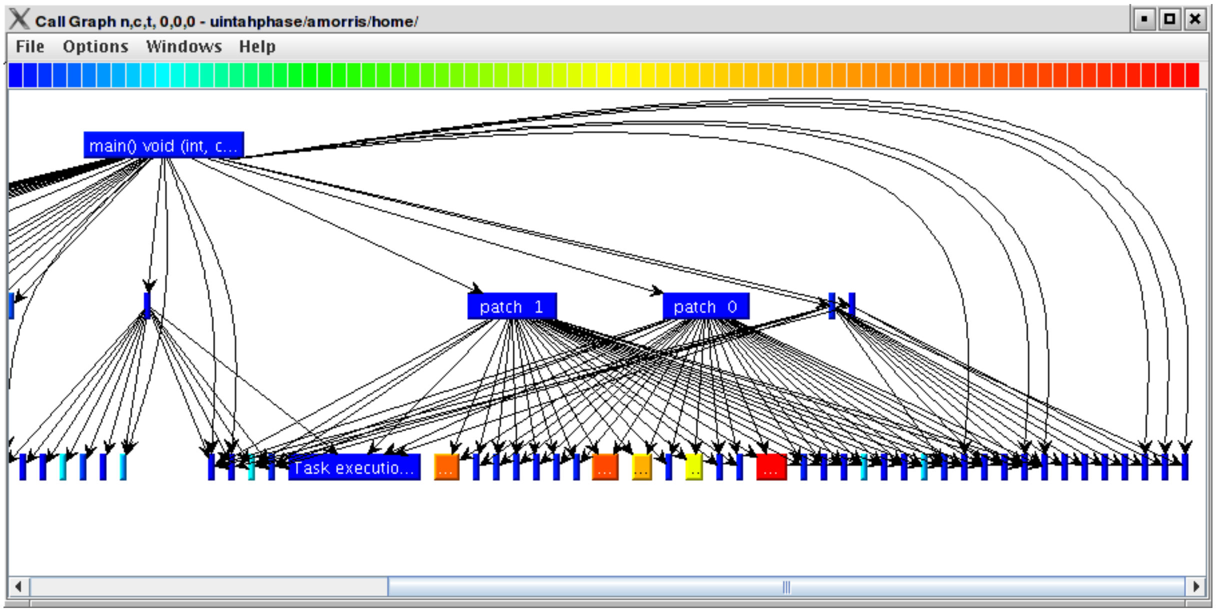

Figure 1. Callgraph view of the TAU's parallel profile of Uintah Computational Framework

3.6. User-defined Events

Besides timers and phases that measure the time spent between a pair of start and stop calls in the code, TAU also provides support for user-defined atomic events. After an event is registered with a name, it may be triggered with a value at a given point in the source code. At the application level, we can use user-defined events to track the progress of the simulation by keeping track of application specific parameters that explain program dynamics, for example, the number of iterations required for convergence of a solver at each time step, or the number of cells in each iteration of an adaptive mesh refinement application.

4. Case Study: Uintah

We have applied TAU's profiling capabilities to evaluate the performance of the Uintah computational framework (UCF) [2]. The case study simulates a capped metal cylinder filled with high-energy material and suspended above a pool of hydrocarbon fuel burning with an open flame. Energy from the flame is transported through the metal cylinder to the high-energy material, causing it to undergo complex chemical changes. Solid deformations, deterioration and cracking occur in both the cylinder and the high-energy material as pressure within the cylinder builds, eventually leading to rupture and detonation. To implement this simulation, a CFD component that simulates hydrocarbon combustion and reactant transport is coupled with a component that uses the material point method (MPM) to simulate the mechanics of solid deformation and energy transport within the cylinder.

The TAU profiling strategy for Uintah is to observe the performance of the framework at the level of patches, the unit of domain partitioning. Thus, we instrument UCF with dynamic phases where the phase name contains the patch identifier. Figure 1 shows the performance data obtained from the simulation as displayed by ParaProf. In this profile, we can distinguish the time taken by a computation “task’ (such as MPM::InterpolateParticlesToGrid) when it was executed within a given patch. It shows the callgraph of this simulation where the width of each node is mapped to the inclusive time spent in that routine. The color of the box shows the exclusive time spent in that routine (blue represents a low cost, red a high cost). As seen in this ParaProf figure, we can partition the time spent in different routines based on the calling phase. The full paper will show other parallel profile displays that highlight features of TAU’s phase based profiling.

Besides the Uintah computational framework, TAU has been applied successfully to several frameworks that are used for computational fluid dynamics simulations. These include VTF from Caltech, MFIX from NETL, ESMF coupled flow application from UCAR, NASA and other institutions, SAMRAI from LLNL, Miranda from LLNL, GrACE from Rutgers University, SAGE from SAIC, and Flash2 from University of Chicago. Though the nature of instrumentation and the performance data presented from TAU differs in each framework, our full paper will show examples of how TAU is applied in these frameworks. Our work in performance evaluation of adaptive scientific computations can be broadly applied to other CFD codes. Thus, CFD frameworks can benefit from the integration of portable performance profiling and tracing support using TAU.

5. Conclusions

When studying the performance of scientific applications, especially on large-scale parallel systems, there is a strong preference among developers to view performance information with respect to their “mental” model of the application, formed from the structural, logical, and numerical models used in the program. If the developer can relate performance data measured during execution to what they know about the application, more effective program optimization may be achieved. In this paper, we present portable performance evaluation techniques in the context of the TAU performance system and its application to the Uintah computational framework. We illustrate how phase based profiling may be effectively used to bridge the semantic gap in comprehending the performance of parallel scientific applications using techniques that map program performance to higher level abstractions.

REFERENCES

1. A. D. Malony and S. Shende and R. Bell and K. Li and L. Li and N. Trebon, “Advances in the TAU Performance System,” Chapter, “Performance Analysis and Grid Computing,” (Eds. V. Getov, et. al.), Kluwer, Norwell, MA, pp. 129-144, 2003.
2. J. D. de St. Germain and J. McCorquodale and S.G. Parker and C.R. Johnson, “Uintah: A Massively Parallel Problem Solving Environment,” Ninth IEEE International Symposium on High Performance and Distributed Computing, IEEE, pp. 33–41. 2000.
3. R. Bell and A. D. Malony and S. Shende, “A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis,” Proc. EUROPAR 2003 conference, LNCS 2790, Springer, pp. 17-26, 2003.
4. TAU Portable Profiling. URL: http://www.cs.uoregon.edu/research/paracomp/tau, 2005.