Prototype Visualization Tools For Multi-Experiment Performance Analysis

Roberto Araiza, Jaime Nava, Alan Taylor, and Patricia Teller
University of Texas-El Paso, El Paso, TX
{raraiza, jenava, amtaylor, pteller}@utep.edu

David Cronk and Shirley Moore
University of Tennessee-Knoxville, Knoxville, TN
{cronk, shirley}@cs.utk.ed

Abstract

The analysis of modern, parallelized applications, such as scientific modeling, is of interest to a variety of people within the computing community of the Department of Defense (DoD). Persons desiring insight into the performance of these large programs include application users, application programmers/developers, portfolio and center managers, and others. The analysis needed requires the examination of large data sets obtained from various performance analysis sources including, but not limited to, hardware counters, software event counters, communications event counters, and unrelated instrumentation code inserted into programs.

The PCAT (PerformanCe Analysis Team) at the University of Texas-El Paso (UTEP) has developed a suite of tools consisting of a performance database access tool and four different visualization methods to aid diverse DoD users in analyzing certain performance issues associated with serial and, especially, parallel programs. The tools are written in Java and provide multiple views of different aspects of performance metrics associated with a performance database. Preliminary analysis of two different codes resulted in PCAT users identifying possible sources of performance degradation solely from examination of performance metrics, without access to the source code.

1. Introduction

The demand for High Performance Computing (HPC) within the Department of Defense (DoD) for modeling physical events and processes continues to increase. Programmers, program managers, computer center managers, and other interested people need more tools to assist them in increasing the productivity of HPC centers. The increased complexity of programs demands tools that can be used to quickly identify subtle performance problems.

Computationally-intensive parallel programs executed on multiprocessors not only encounter performance degradation at the individual CPU level, but also at the system intercommunication level. Thus, the HPC community requires the means to not only find low-level performance bottlenecks, such as functional unit stalls, but higher-level ones as well, such as inter-processor communication stalls. The results of running such applications are of interest to a variety of people, but not for the same reasons. The people that are interested in the execution of these applications are:

- application users,
- application developers,
- portfolio managers,
- center managers/directors, as well as
- other interested parties.

Application users, on the one hand, are concerned with finding a machine or configuration of machines that will run their codes the fastest, so they may get results quickly. Center directors, on the other hand, are more concerned with having all of the machines in the center fully utilized all the time, so supercomputer cycles are not wasted. Data captured from running applications may answer questions of interest to all the types of people interested in the execution of applications on supercomputers (hereinafter called *users*); however, the data must be presented in an appropriate format in order for it to be useful to the intended audience.

Scientific applications instrumented to record events, such as hardware metrics from microprocessors and communications events (e.g., barriers), to evaluate the performance of the applications, generate large amounts of data. Although storing this data in a database facilitates its analysis, it is essential that tools that access the data and display it to users do so in a way that is easily understood.

Application developers and users of modern scientific, parallel applications need easy-to-use, well-engineered tools to determine how well their applications are performing and to analyze and improve application performance. Developers/programmers may collect performance data about the state of the system and the program at runtime by instrumenting applications with

HPCMP Users Group Conference (HPCMP-UGC'06)
0-7695-2797-3/06 $20.00 © 2006 IEEE

tools such as the Performance Application Programmer Interface (PAPI, http://icl.cs.utk.edu/papi/) and the Tuning and Analysis Utilities (TAU, www.cs.uoregon.edu/research/tau/). The obtained data sets help developers identify potential performance problems in applications. Instrumentation of contemporary applications to collect performance data yields huge multi-dimensional data sets, the sizes of which depend on the number of processors involved in the execution, the number of instrumentation points, and the number and type of monitored events. Discovering performance insights in such massive data sets remains a challenge in application performance analysis.

Application developers and users frequently have access to a variety of configurations for compiling and running their applications. Having accessibility to different compilers and optimization levels, MPI implementations, operating systems, and system architectures (including those with the same general architecture but different numbers of processors, and those with the same number of processors but different general architectures) makes it difficult for developers and users to determine which configuration is the best for their particular applications.

Typical questions asked by developers and users include, but are not limited to, the following:

Developers:

DQ1. Are there any parts of my code that would benefit from performance tuning?
DQ2. Now that I have found out which routines are taking the most time, how do I tell if they are running well, i.e., with good performance?
DQ3. Which portions of my code are not scaling well?
DQ4. Is the performance I am getting portable – i.e., after developing the code for one platform, will it run well on other platforms?
DQ5. Are parts of my code memory bound? Communication bound? I/O bound?
DQ6. Where does my code have communication/synchronization deficiencies?
DQ7. Hardware counter data I have collected indicate one or more of the following:
  i. large number of L1/L2, instruction/data cache misses,
  ii. large number of translation look aside buffer (TLB) misses,
  iii. large number of stall cycles,
  iv. large number of misaligned loads, and
  v. large number of mispredicted branches.
DQ8. How do I determine if these factors are adversely affecting performance and, if so, how do I fix the problems?
DQ9 I have made some changes to the code and now it runs faster (slower)? Why?

Users:

UQ1. On which machine(s) will my code get the best performance?
UQ2. On a given platform, which compiler options will give the best performance for my code?
UQ3. How long will my code take to run on a given platform with a given input set?
UQ4. How do different input sets affect performance?
UQ5. What is the best number of processors on which to run my parallel code on a given input set?
UQ6. What performance problems does my code have that I may report to the developer?

2. Survey of HPC User Community

In order to best serve the DoD HPC user community with respect to tools that may aid them in performance data collection and analysis and could answer some of the questions noted above, a small survey of the community was conducted in Fall 2005. The results of the survey indicate a need for such tools and a need for tools that provide a means to simplify the data under analysis using methods such as thresholding and statistical analysis. Anecdotally, one user indicated that s/he used "*printf*" statements in pursuit of bugs and performance problems of multiprocessor programs, and welcomed any improvement in performance analysis tools.

3. Visualization Tools

Under a contract from the DoD, the Performance Analysis Team (PCAT) at the University of Texas at El Paso (UTEP) developed an integrated suite of visualization tools targeted at the DoD HPC community. All tools are written in Java in order to maximize portability and avoid platform dependence, and may be run on any desktop computer that has a Java Virtual Machine installed. In order to minimize learning time for the user, all tools within the suite are controlled via Graphical User Interfaces (GUI's). The input to the tool set consists of data that was previously obtained from instrumented HPC programs and then stored in a database according to an existing schema. The tool set consists of a:

- database query tool,
- colored tree viewer,
- two-dimensional (2-D) visualizer,
- comparator, and
- three-dimensional (3-D) visualizer.

The *database query tool* is used to access performance data stored in a database and to download data to a local desktop computer. The *colored tree viewer*

enables the user to quickly determine which parts of a program are associated with the greatest values of a given metric. The 2-D visualizer provides a means to examine results from multiple runs of an application on different computational platforms in terms of time. The comparator, on the other hand, gives users the ability to quickly compare two different runs of an application in terms of all available metrics. Finally, the 3-D visualizer offers the ability to analyze one or more functions (subroutines/code regions) of an application in terms of multiple metrics, across multiple processors.

The database query tool provides a means to access large data sets stored within a given database. It uses a Graphical User Interface (GUI) to display the call graph of the program under analysis to the user. If there is no call graph in the data, a flat tree is displayed; however, all other tools in the PCAT tool suite may not work properly without call graph information. The database query tool requires the address of the database, name of the database to be accessed, name of the port to use, a login, and password. Once the database is accessed, the user clicks on the desired program for analysis, and the database tool downloads all available data. This is necessary to build the various data structures used by the different tools in the PCAT tool suite. Large data sets may require some time to download; one data set analyzed, associated with the LAMMPS code, took over 45 minutes to download; of course, the time depends on network load. However, once the data are downloaded, the database does not need to be accessed again; the analysis is done locally given that the PCAT tool suite is installed locally. Figure 1 shows a view of the database query tool configured to obtain data from the SHAMRC code database.

Once all data are available for analysis on a local computer, the user may employ any of the several PCAT tools as appropriate. A typical analysis might center on the execution time of an application, leading the user to select the colored tree viewer to determine which functions (subroutines/code regions) of the application take the most time to execute. The tree viewer shows inclusive values of the metric under analysis to the right of a function name, and a colored arrow to the left of a function name provides a quick indication of the magnitude of the value of the metric that is associated with the function. The redder or "hotter" the arrow is, the larger the magnitude of the metric. The bluer or "cooler" the arrow is, the smaller the magnitude. Thus, by merely downloading data from the database and opening the tree viewer, the user, in pursuit of performance problems, can quickly locate functions that are consuming the most time, generating the most resource stalls, etc. Figure 2 shows the colored tree viewer displaying the call graph of the SHAMRC code; note the yellow arrow pointing to the function H2, indicating a larger amount of time consumed within that function.

Having located one or more functions of interest, the user then has three tools that can be used for further analysis; the 2-D visualizer, the comparator, and the 3-D visualizer. Each tool has a distinct purpose, and may not work on some data sets.

The 2-D visualizer requires multiple runs of the same application on different platforms and is specifically intended to assist users in determining such things as which computing platform can execute the application in the least amount of time. Figure 3 shows a comparison among different SHAMRC runs, each executed on a different number of processors, in terms of execution time.

The comparator provides a rapid method for comparing metrics of two different executions (runs) of the same program. Metrics obtained from a program run are displayed in bar graph format with a unique bar for each program function. A separate view is generated for each collected metric. Metrics may be displayed either as raw or normalized counts, providing a quick comparison between two different runs of the same program. Figure 4 compares two different versions of the SHAMRC code run on the same computing platform in terms of resource stalls. The first version of the code is in blue, and the second is in red. The second version shows a lower number of resource stalls in all the functions (associated with available data), especially MOVEZ and H1.

The 3-D visualizer represents metric counts in the form of colored spheres, with each sphere uniquely associated with a metric/function/process tuple. The larger the sphere, the larger the count it represents. Color is used to differentiate among metrics in the Z direction, while the X and Y directions are used to differentiate among functions and processes. Navigation within the view is performed with mouse-controlled buttons and sliders, enabling the user to stretch and shrink all three axes, zoom in and out of the view, as well as view only one function in a bar-graph view. Figure 5 shows the 3-D visualizer displaying multiple metric counts, from multiple functions, on two processors that executed the SHAMRC code.

The PCAT visualization tools were used to analyze performance data from the SHAMRC and LAMMPS programs, two DoD codes. Comparing two different versions of SHAMRC via the comparator, a serial version and the original parallelized version executed on a single processor, it was found that the serial version executed faster than the parallel version and there were notable differences in functions MOVEZ and H1, especially in critical metrics such as floating-point stalls, number of clocks with no instructions completed, and number of clocks with no instructions issued. Analyzing an improved version of the parallel program, there was a reduction in execution time, as compared to the original parallel version, and a decrease in floating-point and

pipeline stalls, as indicated by "no instructions completed" and "no instructions issued". Also, the improved parallel version executed on a single processor had the same execution time as the serial version. Although the observation that the number of stalls increased with execution time provided a "clue", the actual reason for the difference in the performance of the two parallel versions of SHAMRC, which use the same MPI communication methods, was associated with array allocation. The original parallel version dynamically allocates a large array, which was allocated statically in the serial version, while the improved parallel version statically allocates the array. Note that in this case a performance bug in the Intel compiler prevents the use of dynamic allocation without performance degradation.

Although the LAMMPS program contains many more functions (code subsections/subroutines) than SHAMRC, the colored tree viewer was effective in analyzing its performance. Using the viewer, it was determined that the overall program required 1.683 E 10 time units, and a function called Verlet consumed 1.161 E 10 time units, which accounts for the majority of program execution time. Within Verlet, a function called Verlet:iterate consumed virtually all the time, 1.160 E 10 time units. Verlet::iterate called 16 other functions, many of which involved communication. One, in particular, Comm::exchange, consumed 4.685 E nine time units and, another one, Neighbor::build, required 1.311 E9 time units, making them the best candidates for performance analysis.

Using the 3-D viewer, it was found that a large number of floating-point stalls, branch mispredictions and resource stalls were located within these two functions, especially with respect to the MPI functions, such as MPI_Wait, MPI_Sendrecv, MPI_Send and MPI_Irecv, all of which are involved in interprocessor communications. Thus, the utility of the PCAT visualization tools to locate potential performance problems using only performance data and derived metrics, with no access to the source code, was demonstrated.

4. Conclusions and Future Work

The PCAT has created multiple visualization tools that, in conjunction with a database query tool, offer the DoD HPC community another means of locating performance problems within scientific applications. The tools have been demonstrated using performance data associated with DoD codes and stored in a database.

Providing the integrated PCAT visualization tools to a small number of DoD HPC programmers, as a prototype for evaluation, and re-surveying a larger portion of the DoD HPC community on the subject of visualization tools would be logical extensions to this work. A web site intended to assist users of the PCAT tools could be created in a fairly short period of time and at modest cost. Feedback from those programmers who evaluate the tools would provide direction for further development.

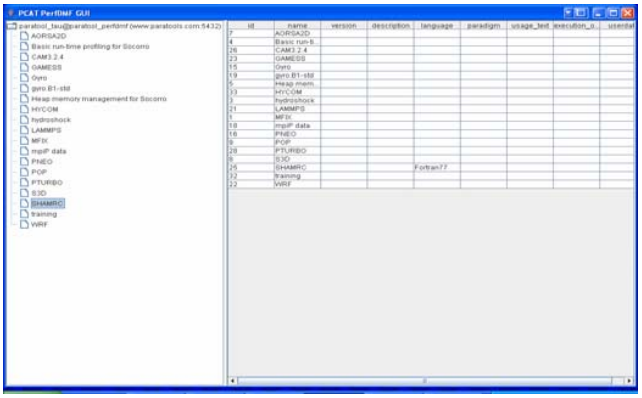

Figure 1. PCAT database query tool accessing SHAMRC database

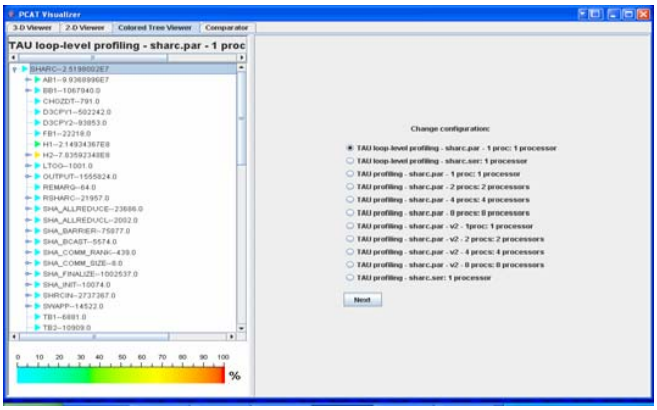

Figure 2. PCAT colored tree viewer displaying SHAMRC call graph

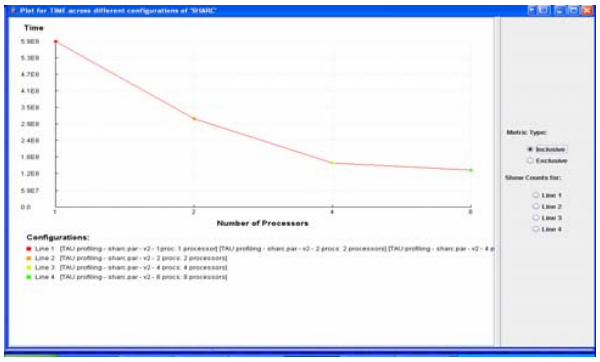

Figure 3. PCAT 2-D visualizer showing SHAMRC execution time for different numbers of processors

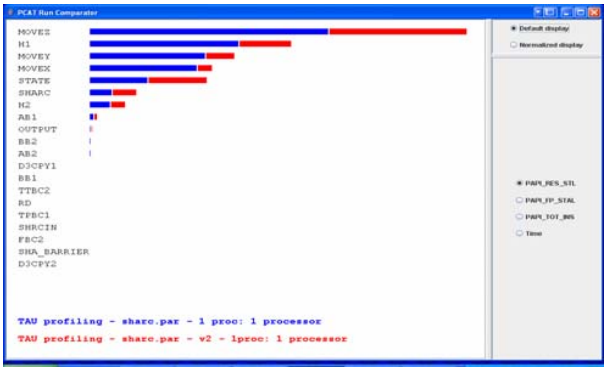

Figure 4. PCAT comparator displaying different versions of shamrc code in terms of resource stalls

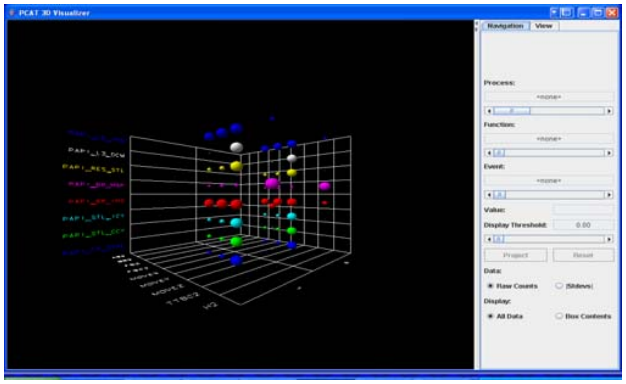

Figure 5. PCAT 3-D visualizer displaying multiple metrics across multiple functions, on two processors

0-7695-2797-3/06 $20.00