Optimization of Instrumentation in Parallel Performance Evaluation Tools

Sameer S. Shende, Allen D. Malony, and Alan Morris

Performance Research Laboratory,
Department of Computer and Information Science
University of Oregon, Eugene, OR, USA,
{sameer,malony,amorris}@cs.uoregon.edu

Abstract. Tools to observe the performance of parallel programs typically employ profiling and tracing as the two main forms of event-based measurement models. In both of these approaches, the volume of performance data generated and the corresponding perturbation encountered in the program depend upon the amount of instrumentation in the program. To produce accurate performance data, tools need to control the granularity of instrumentation. In this paper, we describe our experiences in the TAU performance system for improving the accuracy of performance data by limiting the amount of instrumentation. A range of options are provided to optimize instrumentation based on the structure of the program, event generation rates, and historical performance data gathered from prior executions.

Keywords: Performance measurement and analysis, parallel computing, profiling, tracing, instrumentation, optimization.

1 Introduction

The advent of large scale parallel supercomputers is challenging the ability of tools to observe application performance. As the complexity and size of these parallel systems continue to evolve, so must techniques for characterizing the performance of parallel programs. Profiling and tracing are two commonly used techniques for evaluating application performance. Tools based on profiling maintain summary statistics of performance metrics, such as inclusive and exclusive time or hardware performance monitor counts [3], for routines on each thread of execution. Performance evaluation tools either employ sampling of program state based on periodic interrupts or direct instrumentation. Sampling provides a fixed overhead, the accuracy of performance data generated depends on the inter-interrupt sampling interval. Here, we consider direct measurement based instrumentation techniques where instrumentation hooks are inserted at relevant program events. As the program executes, events, or actions that take place in the program are inspected to characterize the program execution. In this paper, we describe our experiences in optimizing program instrumentation in the TAU performance system [1]. Section §2 describes the motivation for the problem, Sections §3 and Sections §4 describe how we can limit the instrumentation based on selective instrumentation and measurement-based approaches.

2 Motivation

Optimizing program instrumentation can help us control the perturbation induced in the program for both tracing and profiling. Trace analysis of a parallel application is invaluable in improving our understanding of the temporal aspects of program execution. The ability to zoom into sections of trace files allows us to see the occurrence of events along a global timeline. Sometimes, trace-based displays of event streams from multiple processors can immediately highlight the causes of poor performance. It is not always sufficient to perform trace analysis on a small set of processors and extrapolate the cause of poor performance on a larger set, as performance properties may differ as the number of processors increases. Hence, to identify causes of poor performance, it is important to be able to observe the performance of a given application on large scale parallel systems. Balancing the volume of performance data produced and the accuracy of performance measurements is key to optimizing the instrumentation.

Techniques for improving performance observability fall into three broad categories:

– Instrumentation - Techniques that *reduce* the number of instrumentation points inserted in the program
– Measurement - Techniques that *limit* the amount of information emitted by the tool at the instrumentation points, and
– Analysis - Techniques that *scale* the number of processors involved in processing the performance data, and techniques that reduce and reclassify the performance information.

In this paper, we will limit our discussion to instrumentation and measurement based approaches.

3 Instrumentation

To reduce the volume of trace data, sometimes tracing tools limit program coverage by just focusing on a single library for instrumentation. Tools that highlight MPI performance typically fall into this category. An MPI interposition library tracks the time spent in all MPI routines. A manual instrumentation API often accompanies such tools where a user can insert annotations in the source code to record application events as well. For non-trivial applications, this becomes quite cumbersome. To aid this process, automated means for inserting program instrumentation can be utilized as well [5]. This is in the form of compiler flags or binary instrumentation techniques that instrument all routines within a given file. This requires specialized knowledge from a user's perspective as he/she must decide which files to exclude for instrumentation. Such choices are often a result of choosing just tracing for performance evaluation.

3.1 Selective Instrumentation

In the TAU project, we use a combination of profiling and tracing to effectively limit the program instrumentation. Using a variety of instrumentation techniques such as pre-processing using PDT [2], MPI wrapper interposition library, binary re-writing and dynamic instrumentation using DyninstAPI [6], a user can instrument all routines in a given program. When a fully instrumented program is executed, it produces profiles. TAU provides a tool, *tau_reduce*, to analyze the profiles and apply a set of rules for instrumentation control. The output is a list of routines that should be excluded from instrumentation.

Naive instrumentation of parallel programs can easily include lightweight routines that perturb the applcation significantly when measured. If the user does not specify the rules for removing instrumentation using *tau_reduce*, TAU applies a default set (e.g., the number of calls must exceed a million and the inclusive time per call for a given routine must be less than 10 microseconds to exclude the routine). The program is then re-instrumented using the *exclude list* emitted by *tau_reduce*. To ensure that other routines that were above the threshold for exclusion before do not qualify for exclusion after re-instrumentation (due to removal of instrumentation in child routines), the user may re-generate the exclude list by re-running the program against the same set of rules. When any two instrumented executions generate no new exclusions, we say that the instrumentation fixed-point is reached for a given set of execution parameters (processor size, input, etc.) and instrumentation rules. The instrumentation is sufficiently coarse-grained to be accurate and the user may generate traces after this optimization of program instrumentation. Using both profiling and tracing together improves the program coverage and reduces the amount of trace data generated. In the next subsection, we examine how other forms of co-operation may improve such program coverage while reducing the volume of trace data generated.

4 Measurement

During program execution, instrumentation may be disabled in the program based on spatial, context or location based constraints imposed. When an entire program is instrumented, it may be difficult to reach an instrumentation fixed-point by re-running the application. The amount of instrumentation may be too much and the initial run might take a significant amount of time. To optimize the instrumentation at runtime, TAU uses a scheme to throttle events at runtime.

TAU allows the user to disable the instrumentation at runtime based on rules similar to the ones employed by the offline analysis of profiles using *tau_reduce*. The number of calls to each event is examined and when it exceeds a given user specified threshold (e.g., 100000 calls), the inclusive per-call value is examined to compare it with another threshold (e.g., 10 microseconds per call). If it falls below the threshold, the event is disabled and added to a new profile group (TAU_DISABLE). Subsequent calls to start or stop that event incur a minimal overhead of masking two bitmaps and effectively reduce the overhead. This is useful for disabling events that have a high frequency and low cost of execution.

Other techniques for controlling instrumentation costs include compensation of instrumentation overhead, APIs for event grouping and control of instrumentation, full program instrumentation control, context based control based on callpath depths, and callstack based control. In callstack based control, the program structure is analyzed at runtime and trace records are emitted for those events that directly or indirectly call an MPI routine. This generates compact TAU traces that may be converted to the Epilog format for further analysis by the Expert [5] tool.

The full paper will highlight these and other techniques for controlling program instrumentation.

5 Conclusion

Tool developers attempt to build measurement systems as efficiently as possible, but to improve the accuracy of performance measurements, it is important to optimize the program instrumentation. In this paper, we describe the techniques for generating coarse grained instrumentation when the entire program is instrumented automatically because the accuracy of performance data is inversely correlated with the degree of performance instrumentation.

6 Acknowledgments

Research at the University of Oregon is sponsored by contracts (DE-FG02-05ER25663, DE-FG02-05ER25680) from the MICS program of the U.S. Dept. of Energy, Office of Science.

References

1. A. Malony, S. Shende, “Performance Technology for Complex Parallel and Distributed Systems,” In G. Kotsis, P. Kacsuk (eds.), *Distributed and Parallel Systems, From Instruction Parallelism to Cluster Computing, Third Workshop on Distributed and Parallel Systems (DAPSYS 2000)*, Kluwer, pp. 37–46, 2000.
2. K. Lindlan, J. Cuny, A. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen, “A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates,” SC 2000 conference, 2000.
3. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, “A Portable Programming Interface for Performance Evaluation on Modern Processors,” *International Journal of High Performance Computing Applications*, **14**(3):189–204, Fall 2000.
4. D. Reed, L. DeRose, and Y. Zhang, “SvPablo: A Multi-Language Performance Analysis System,” *International Conference on Performance Tools*, pp. 352–355, September 1998.
5. B. Mohr, F. Wolf, “KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Programs,” *Euro-Par 2003 conference*, August 2003.
6. B. Buck and J. Hollingsworth, “An API for Runtime Code Patching”, *Journal of High Performance Computing Applications*, pp. 317–329, 14(4), 2000.