A Scalable Approach to MPI Application Performance Analysis

Shirley Moore[1], Felix Wolf[1], Jack Dongarra[1],
Sameer Shende[2], Allen Malony[2], and Bernd Mohr[3]

[1] Innovative Computing Laboratory, University of Tennessee
Knoxville, TN 37996-3450 USA
{shirley,fwolf,dongarra}@cs.utk.edu
[2] Computer Science Department, University of Oregon
Eugene, OR 97403-1202 USA
{malony,sameer}@cs.uoregon.edu
[3] Forschungszentrum Jülich, ZAM, 52425 Jülich, Germany
b.mohr@fz-juelich.de

Abstract. A scalable approach to performance analysis of MPI applications is presented that includes automated source code instrumentation, low overhead generation of profile and trace data, and database management of performance data. In addition, tools are described that analyze large-scale parallel profile and trace data. Analysis of trace data is done using an automated pattern-matching approach. Examples of using the tools on large-scale MPI applications are presented.

1 Introduction

Parallel computing is playing an increasingly critical role in advanced scientific research as simulation and computation are becoming widely used to augment and/or replace physical experiments. However, the gap between peak and achieved performance for scientific applications running on large parallel systems has grown considerably in recent years. The most common parallel programming paradigm for these applications is to use Fortran or C with MPI message passing to implement parallel algorithms. The complex architectures of large parallel systems present difficult challenges for performance optimization of such applications. Tools are needed that collect and present relevant information on application performance in a scalable manner so as to enable developers to easily identify and determine the causes of performance bottlenecks.

Performance data encompasses both profile data and trace data. Profiling involves the collect of statistical summaries of various performance metrics broken down by program entities such as routines and nested loops. Performance metrics include time as well as hardware counter metrics such as operation counts and cache and memory event counts. Tracing involves collection of a timestamped sequence of events such as entering and exiting program regions and sending and receiving messages. Profiling can identify regions of a program that are consuming the most resources, while detailed tracing can help identify the causes of performance problems. On large parallel systems, both profiling and tracing present scalability challenges.

Collecting either profile or trace data requires the application program to be instrumented. Instrumentation can be inserted at various stages of the program build process, ranging from source code insertion to compile time to link time to run time options. Although many tools provide an application programmer interface (API), manual insertion of instrumentation library calls into application source code is too tedious to be practical for large-scale applications. Thus our tools support a range of automated instrumentation options. Once an instrumented version of the program has been built, only a few environment variables need to be set to control runtime collection of profile and/or trace data.

Collecting profile data for several different metrics on a per-process and per-routine basis, possibly for several runs with different numbers of processors and/or different test cases and/or on different platforms results in a data management problem as well as a presentation and analysis problem. Similar profile data may be collected by different tools but be incompatible because of different data formats. Our solution to the data management problems is a performance data management framework that sits on top of a relational database and provides a common profile data model as well as interfaces to various profile data collection and analysis tools. For presentation of profile data we have developed graphical tools that display the data in 2-dimensional and 3-dimensional graphs. Our tools also support multi-experiment analysis of performance data collected from different runs.

Event tracing is a powerful method for analyzing the performance behavior of parallel applications. Because event traces record the temporal and spatial relationships between individual runtime events, they allow application developers to analyze dependencies of performance phenomena across concurrent control flow. While event tracing enables the identification of performance problems on a high level of abstraction, it suffers from scalability problems associated with trace file size. Our approach to improving the scalability of event tracing uses call-path profiling to determine which routines are relevant to the analysis to be performed and then traces only those routines.

Graphical tools such as Vampir, Intel Trace Analyzer, and Jumpshot are available to view trace files collected for parallel executions. These tools typically show a time-line view of state changes and message passing events. However, analyzing these views for performance bottlenecks can be like searching for a needle in a haystack. Our approach searches the trace file using pattern-matching to automatically identify instances of inefficient behavior. The performance bottlenecks that are found and related to specific program call-paths and node/process/thread locations can then be focused on using one of the previously mentioned trace file viewing tools.

The remainder of this paper is organized as follows. Section 2 describes our automated approach to insertion of performance instrumentation and collection of performance data. Section 3 describes our performance data management framework. Section 4 describes our scalable approaches to analyzing profile data, including techniques for multi-experiment analysis. Section 5 describes our scalable automated approach to trace file analysis. Section 6 contains conclusions and directions for future research.

2 Automated Performance Instrumentation

TAU (Tuning and Analysis Utilities) is a portable profiling and tracing toolkit for parallel threaded and or message-passing programs written in Fortran, C, C++, or Java, or a combination of Fortran and C [3]. TAU can be configured to do either profiling or tracing or to do both simultaneously. Instrumentation can be added at various stages, from compile-time to link-time to run-time, with each stage imposing different constraints and opportunities for extracting program information. Moving from source code to binary instrumentation techniques shifts the focus from a language specific to a more platform specific approach.

Source code can be instrumented by manually inserting calls to the TAU instrumentation API, or by using the Program Database Toolkit (PDT) and/or the Opari OpenMP rewriting tool to insert instrumentation automatically. PDT is a code analysis framework for developing source-based tools. It includes commercial grade front end parsers for Fortran 77/90, C, and C++, as well as a portable intermediate language analyzer, database format, and access API. The TAU project has used PDT to implement a source-to-source instrumentor (`tau_instrumentor`) that supports automatic instrumentation of C, C++, and Fortran 77/90 programs.

The TAU MPI wrapper library uses the MPI profiling interface to generate profile and/or trace data for MPI operations. TAU MPI tracing produces individual node-context-thread event traces that can be merged to produce SLOG, SDDF, Paraver, VTF3, or EPILOG trace formats.

TAU has filtering and feedback mechanisms for reducing instrumentation overhead. The user can specify routines that should not be instrumented in a selective instrumentation file. The `tau_reduce` tool automates this specification using feedback from previously generated profiling data by allowing the user to specify a set of selection rules that are applied to the data.

3 Performance Data Management Framework

TAU includes a performance data management framework, called PerfDMF, that is capable of storing parallel profiles for multiple performance experiments. The performance database architecture consists of three components: performance data input, database storage, database query, and analysis. The performance profiles resident in the database are organized in a hierarchy of *applications*, *experiments*, and *trials*. Application performance studies are seen as constituting a set of experiments, each representing a set of associated performance measurements. A trial is a measurement instance of an experiment. Raw TAU profiles are read by a profile translator and stored in the database. The performance database is an object-relational DBMS specified to provide a standard SQL interface for performance information query. MySQL, PostgreSQL, or Oracle can be used for the database implementation. A performance database toolkit developed with Java provides commonly used query and analysis utilities for interfacing performance analysis tools. ParaProf (described in the next section) is one of the tools capable of using this high-level interface for performance database access. Other performance analysis tools that have been interfaced with PerfDMF include mpiP, Dynaprof, HPM, gprof, and KOJAK.

4 Scalable Display and Analysis of Profile Data

ParaProf is a graphical parallel profile analyzer that is part of the TAU toolkit. Figure 1 shows the ParaProf framework architecture. Analysis of performance data requires representations from a very fine granularity, perhaps of a single event on a single node, to displays of the performance characteristics of the entire application. ParaProf's current set of displays range from purely textual based to fully graphical. Many of the display types are hyper-linked enabled, allowing selections to be reflected across currently open windows.

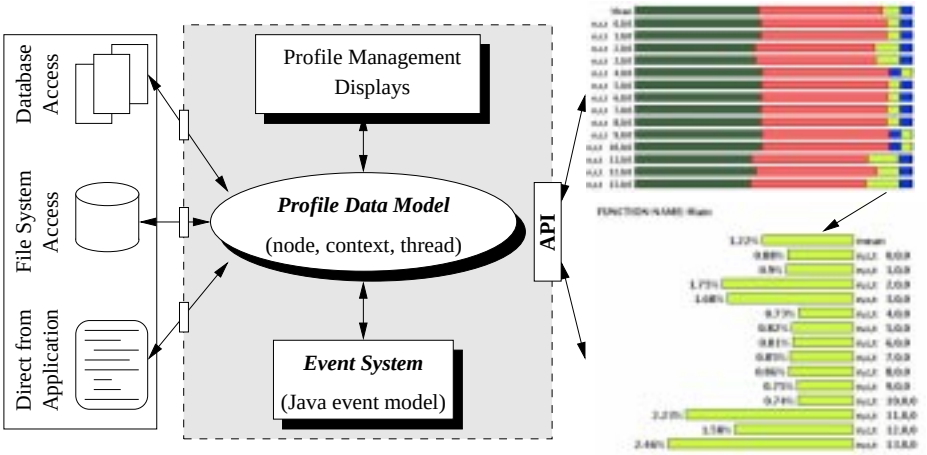

Fig. 1. ParaProf Architecture.

Recently, the TAU project has focused on how to measure and analyze larg-scale application performance data. A significant amount of performance data can be generated for large processor runs. We have been experimenting with three-dimensional displays of large-scale performance data. For instance, Figure 2 shows the entire parallel profile measurement for a 32K processor run. The performance events (i.e., functions) are along the x-axis, the threads are along the y-axis, and the performance metric (in this case, the exclusive execution time) is along the z-axis. This full performance view enables the user to quickly identify major performance contributors. Figure 3 is of the same dataset, but in this case each thread is shown as a sphere at a coordinate point determined by the relative exclusive execution time of three significant events. The visualization gives a way to see clustering relationships.

5 Automated Analysis of Trace Data

KOJAK is an automatic performance evaluation system for parallel applications that relieves the user from the burden of searching large amounts of trace data manually by automatically looking for inefficient communication patterns that force processes into

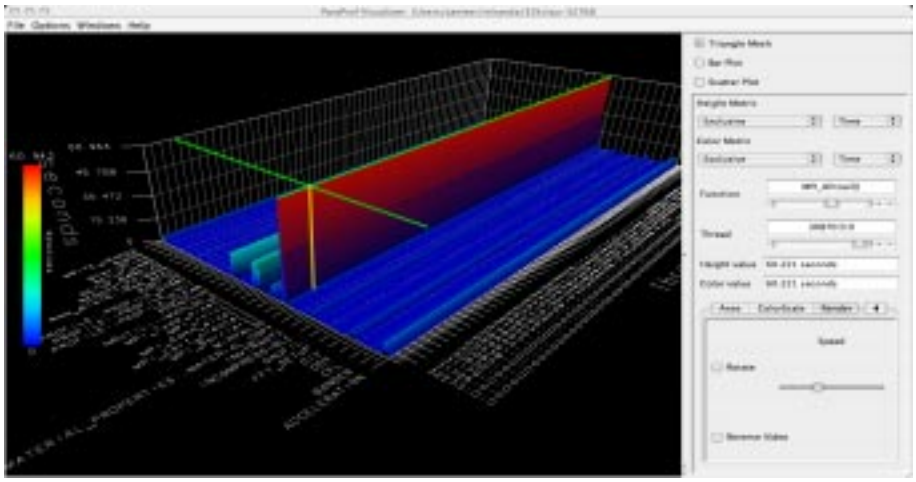

Fig. 2. Scalable Miranda Profile Display

undesired wait states. KOJAK can be used for MPI, OpenMP, and hybrid applications written in C/C++ or Fortran. It includes tools for instrumentation, event-trace generation, and post-processing of event traces plus a generic browser to display the analysis results. The instrumentation tools complement those supplied by TAU.

After program termination, the trace file is analyzed offline using EXPERT [5], which identifies execution patterns indicating low performance and quantifies them according to their severity. These patterns target problems resulting from inefficient communication and synchronization as well as from low CPU and memory performance. The analysis process automatically transforms the traces into a compact call-path profile that includes the time spent in different patterns.

Finally, the analysis results can be viewed in the CUBE performance browser [4], which is depicted in Figure 4. CUBE shows the distribution of performance problems across the call tree and the parallel system using tree browsers that can be collapsed and expanded to meet the desired level of granularity. TAU and KOJAK interoperate in that TAU profiles can be read by CUBE, and CUBE profiles can be read by ParaProf and exported to PerfDMF.

We recently used KOJAK to investigate scalability problems observed in running the GYRO MPI application [1] on the SGI Altix platform using a specific input data set. We used TAU in combination with PDT to automatically insert appropriate EPILOG API calls into the GYRO source code to record entries and exits of user functions. Unfortunately, non-discriminate instrumentation of user functions can easily lead to significant trace-file enlargement and perturbation: A TAU call path profile taken of a fully-instrumented run with 32 processes allowed us to estimate the trace file size above 100 GB.

As a first result, we present an automated strategy to keep trace-file size within manageable bounds. It was notable that shortly-completed function calls without involving any communication accounted for more than 98 % of the total number of function-call events. Since the intended analysis focuses on communication behavior only, we auto-

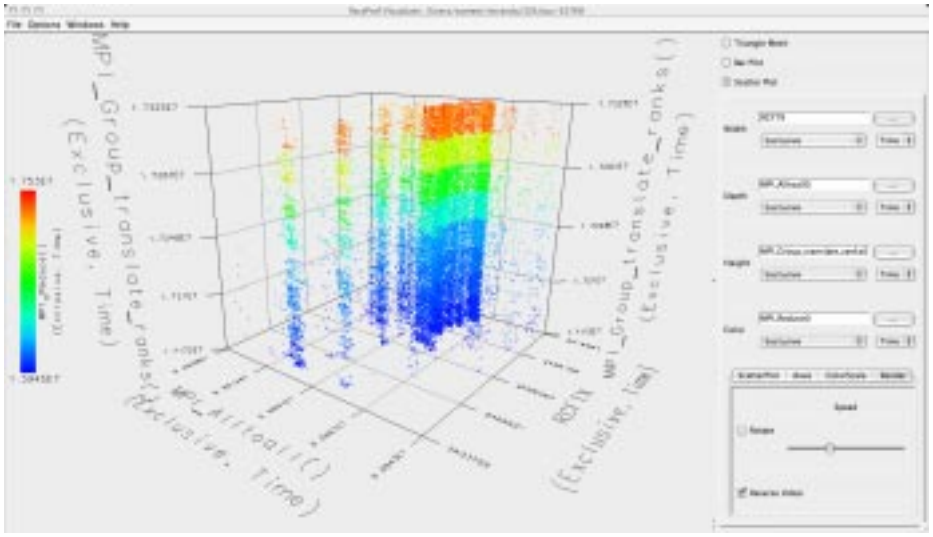

Fig. 3. 3-D Scatter Plot of Performance Metrics for 32K Miranda Processes

matically generated a so-called *TAU include list* from the call path profile using a script. The include list allowed us to instrument only user functions directly or indirectly performing MPI operations. Based on this include list we took trace files of with 32, 64, 128, and 192 processes. Trace file sizes varied between 94 and 562 MB and did not present any obstacles to our analysis.

GYRO's communication behavior is dominated by collective operations - in particular n-to-n operations, where every process sends to and receives from every other process. Due to their inherent synchronization, these operations often create wait states when processes have to wait for other processes to begin the operation. KOJAK defines a pattern called *Wait at N* $\times$ *N* that identifies this situation and calculates the time spent in the operation before the last participant has reached it.

Figure 4 shows the KOJAK result display for the 192-processor trace. All numbers shown represent percentages of the execution time accumulated across all processes representing the total CPU-allocation time consumed. The left tree contains the hierarchy of patterns used in the analysis. The numeric labels indicate that 21.3 % was spent waiting as a result of Wait at N $\times$ N as opposed to 27.7 % spent in actual collective communication. The middle tree shows the distribution of this pattern across the call tree. Thus, about 1/2 of the collective communication time is spent in wait states, a diagnosis that is hard to achieve without using our technique.

To better understand the evolution of these phenomena as processor counts increase, we compared the analysis output of the 128-processor run against the 192-processor run using KOJAK's performance algebra utilities [4]. The difference-operator utility computes the difference between the analysis results belonging to two different trace files. The difference can be viewed using the KOJAK GUI just in the same way the original results can be viewed. Figure 5 shows the 128-processor results subtracted from

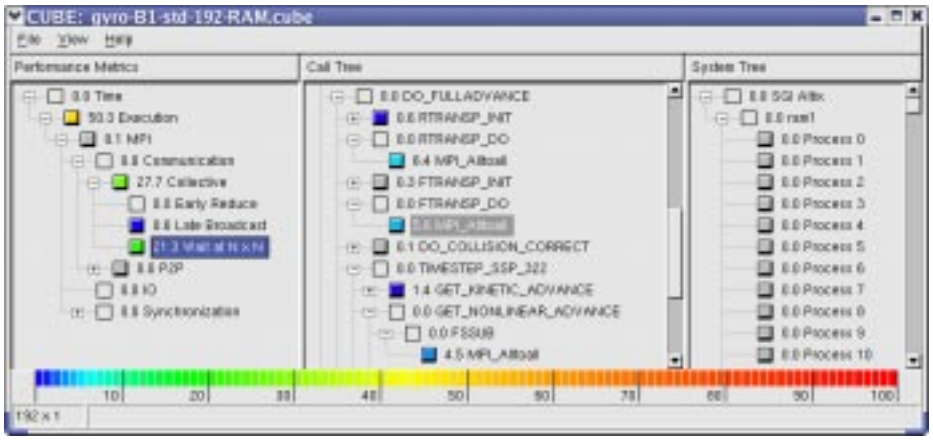

Fig. 4. KOJAK analysis results on 192 CPUs.

the 192-processor results. According to our measurements with KOJAK, the total accumulated execution time grows by 72.3 % when scaling from 128 to 192 processes, indicating a significant decrease in parallel efficiency. Figure 5 (left tree) shows the composition of the difference classified by performance behavior with all numbers being percentages of the total time difference. About 2/3 can be attributed to waiting in all-to-all operations, while about 1/3 can be attributed to the actual communication in collective operations. The increase in computation is negligible.

After writing short script for EARL [2], a high-level read interface to KOJAK event traces, we found that the performance problems observed when running with 192 CPUs happen in relatively small communicators not exceeding a size of 16. Although this information was not yet sufficient to remove the performance problem, it allowed us to pose the question about it more clearly by showing the evolution of hard-to-diagnose performance behavior (i.e., wait states) in a way that cannot be done using traditional tools.

Future work will investigate the reason for the increased waiting and communication times and try to clarify whether there is a relationship between both phenomena. Since the code scales well on other platforms, platform rather than application characteristics might play a role.

6 Conclusions and Future Work

TAU provides an extensible framework for performance instrumentation, measurement, and analysis. KOJAK provides an automated approach to analysis of large-scale event traces. The benefits of our research include automated trace-size reduction and automated analysis of hard-to-diagnose performance behavior. However, further work is needed on integrating profile and trace data analysis and on supporting additional tools such as multivariate statistical analysis tools. Further work is also needed to process the

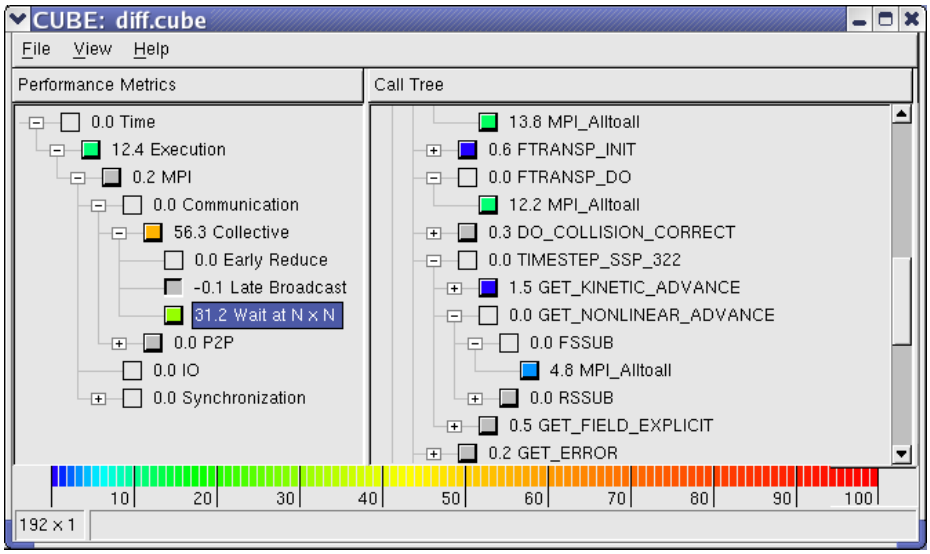

Fig. 5. Differences between the 128- and the 192-processor run.

trace files in a distributed parallel manner in order to scale to terascale and petascale systems of the future.

7 Acknowledgements

This research is supported at the University of Tennessee by the U.S. Department of Energy, Office of Science contract DE-FC02-01ER25490, and at the University of Oregon by the U.S. Department of Energy, Office of Science contract DE-FG02-05ER25680.

References

1. J. Candy and R. Waltz. An Eulerian gyrokinetic Maxwell solver. *J. Comput. Phys.*, 186:545, 2003.
2. N. Bhatia F. Wolf. EARL - API Documentation. Technical Report ICL-UT-04-03, University of Tennessee, Innovative Computing Laboratory, October 2004.
3. S. S. Shende. *The Role of Instrumentation and Mapping in Performan ce Measurement*. PhD thesis, University of Oregon, August 2001.
4. F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An Algebra for Cross-Experiment Performance Analys is. In *Proc. of the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August 2004.
5. F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient Pattern Search in Large Traces through Successive Refinement. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, Pisa, Italy, August - September 2004.