Enhanced Productivity Using the Cray Performance Analysis Toolset

Heidi Poxon
Cray Inc.

Luiz DeRose
Cray Inc.

Abstract

The purpose of an application performance analysis tool is to help the user identify whether or not their application is running efficiently on the computing resources available. However, the scale of current and future high end systems, as well as increasing system software and architecture complexity, brings a new set of challenges to todays performance tools. In order to achieve high performance on these peta-scale computing systems, users need a new infrastructure for performance analysis that can handle the challenges associated with multiple levels of parallelism, hundreds of thousands of computing elements, and novel programming paradigms that result in the collection of massive sets of performance data. In this paper we present the Cray Performance Analysis Toolset, which is set on an evolutionary path to address the application performance analysis challenges associated with these massive computing systems by highlighting relevant data and by bringing Cray optimization knowledge to a wider set of users.

1 Introduction

In order to be able to achieve high-productivity with peta-scale systems, users will need a new infrastructure for performance analysis that can handle the challenges associated with heterogeneous architectures with multiple levels of parallelism, hundreds of thousands of computing elements, and novel programming paradigms. The current state of the art in performance tools can barely support the needs of current architectures and will certainly not be adequate for the future high-performance computing systems.

The new generation of performance tools will have to be able to automate the process of performance analysis and will require a monitoring infrastructure able to handle the massive volume of performance data with low overhead. In order to accomplish these goals, we designed a new generation of performance analysis tools for the Cray Cascade peta-scale system. This advanced performance analysis infrastructure will enhance productivity by providing innovative techniques that use a knowledge base and set of performance models to support automatic performance analysis. In order to provide users with early access to this technology, we are phasing in features of this design in phases into our performance tools framework.

In this paper we outline the key aspects of this design and highlight some features from this design that were already integrated into our performance tools framework. The remainder of this paper is organized as follows: In Section 2 we discuss some of the performance tools challenges and requirements to support peta-scale systems. Section 3 provides an overview of the key components of the Cray analysis engine for Automatic Performance Analysis and their interaction. Section 4 presents some features of this approach that have been incorporated into CrayPat and Cray Apprentice^2. Finally, Section 5 presents our conclusions and future work.

2 Peta-scale Challenges

For peta-scale systems, the core components of the performance analysis and tuning cycle: instrumentation, measurement, analysis, and presentation can in principle remain the same. However, the overall control of this approach needs a fundamental redesign and generalization for a number of reasons, including the following:

- due to the power and processing speed of a peta-scale system, the user can no longer be involved in all aspects of performance analysis and tuning,
- the amount of performance data produced by program monitoring for trace files can exceed the capacity of the file system and overburden the workstations used for analyzing these data,
- most traditional tools for the visual presentation of performance data do not scale in an obvious way to the number of components in peta-scale systems,

In order to be able to obtain sustained performance on system architectures with tens or even hundreds of thousands of computing elements and multiple levels of parallelism, users will need performance tools that use automation to enhance the performance analysis process. Although a great deal of progress has been achieved in the area of performance measurement for high performance computing systems [2, 6, 4, 10, 12, 8, 7, 5], very little progress has been made in the area of *Performance Analysis*, where the most notable work includes Cray's ATExpert [9], Paradyn [11] from the University of Wisconsin, and KOJAK [13] from the Research Center Juelich. The main distinction here is that performance measurement tools provide the data needed for users to tune and optimize applications. However, in order to understand the performance behavior of the application, users still need to understand the intricate details of processor architecture, system software, and compilation systems, to correlate the observations from the performance measurements with the system in use. Contrary to this, performance analysis tools attempt to perform this correlation automatically by applying a combination of a knowledge base with a set of performance models. Cray pioneered this concept with ATExpert, which was the first and only vendor tool to incorporate performance modeling and analysis into performance tools for insight into performance problems and identification of possible solutions.

For peta-scale systems, performance analysis tools will require an infrastructure for monitoring the execution of hardware as well as software components, and the ability to gather the required information with low overhead. In addition, they will need to handle the large amount of performance data needed for the analysis process. Otherwise, the productivity of these systems will be compromised.

Post-mortem application performance optimization will require measurement and tuning at many levels. Performance data will be correlated across many dimensions. Hence, the visualization system should provide a *drill-down* mechanism for hierarchical performance analysis, so users can understand and explore the immediate cause of poor performance, as well as its interactions and the underlying causes, beginning with high-level abstractions and drilling down for additional details where needed. Moreover, the visualization system should be able to not only identify performance bottlenecks, but also provide information to the user describing possible reasons for such bottlenecks and suggestions for improvements, basically, *where* are the performance bottlenecks, *why* these are performance bottlenecks, and *what* should be done to correct the problem.

3 The Cray Automatic Performance Analysis Framework

Cray is developing a new generation of performance analysis system that will address the issues discussed above. An innovative component of this infrastructure is an *analysis engine* whose actions are triggered by hardware or software events that occur during execution of the application. This analysis engine, referred to as *Pat_analyze*, will be guided by a performance model that combines knowledge of the underlying system with data correlated during application execution and Cray optimization expertise. The core competence of the system comprises all aspects related to performance monitoring, analysis, and tuning. It supports automatic *on-line* as well as *off-line* operation, involving the user mainly in high-level strategic decisions. The distributed nature of the system supports local analysis without the need to always refer to a centralized entity providing full global knowledge. This results in a number of important advantages. For example, local processing and filtering of data provided by a (hardware or software) *sensor* can reduce the volume of data to be transferred to higher levels of the analysis engine, thus lowering the bandwidth requirements of the system.

3.1 The Structure of PAT_analyze

The goal of PAT_analyze is to provide high-level expertise as an aid to solving the problem of inefficiently executing applications on the system. It will contain basic knowledge about the underlying hardware and software system, the primary programming models (MPI and OpenMP) used for writing scientific applications, and properties of key application classes. It will collect data characterizing the behavior of programs executing on the system, will analyze and process these data, and will make *recommendations* regarding changes of parameters, software components or resource allocation, with the goal of improving performance. All these actions will be guided by a *performance model*, which in addition to system knowledge includes user-provided input on specific performance problems and their relative importance in view of the objective function.

The main components of PAT_analyze are illustrated in Figure 1. At the highest level of abstraction, PAT_analyze consists of a *knowledge base* and a *framework for introspection*. The knowledge base contains domain knowledge, which can be organized into knowledge about the underlying system, applications, performance, executions, and methods for acquisition of new knowledge. The framework for introspection provides a general system of autonomous asynchronous agents that implement the functionality of the analysis engine.

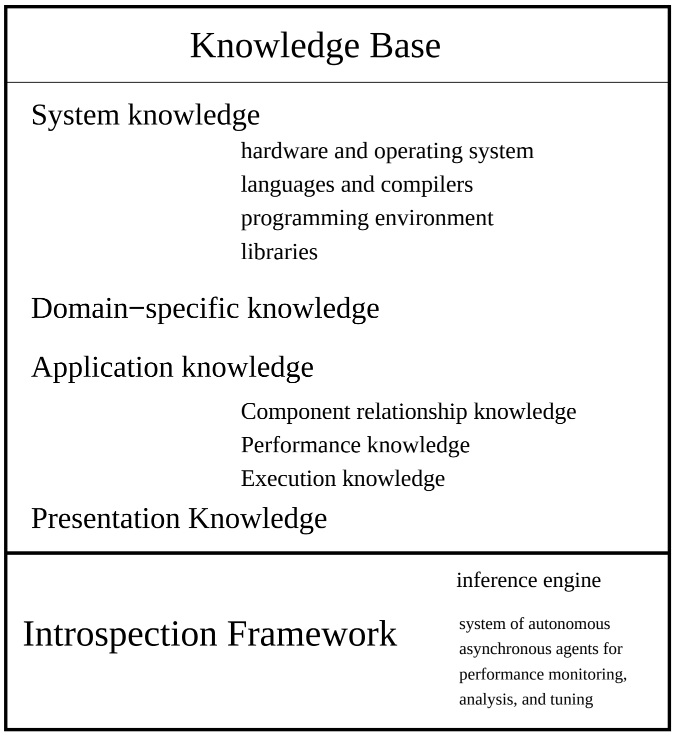

Figure 1: Main components of PAT_analyze

3.2 The Knowledge Base

Each component of the *Knowledge Base (KB)* contains facts and rules. Facts can be of a "semi-persistent" nature – such as information about the number of nodes in the architecture and their internal structure, but can also include assertions about the execution time of a certain routine, or pre- and post conditions specifying the functional transformation of performance parameters as a result of executing a function.

3.2.1 System Knowledge

The contents of the *System Knowledge Base (SKB)* provide a "(semi-) persistent" view[1] of the system underlying the execution of an application. It includes properties of:

- the target architecture, such as number of nodes, number of processing cores per node, cache characteristics, memory sizes, bandwidth and latency, network topology

[1]This information is not strictly persistent, since over time components of the hardware as well as the software may change. For example, the hardware configuration may be modified, libraries may be added or removed, and new tools or compiler-related functionality (such as new analysis functions) may be included.

- languages and language versions that can be handled by the system,
- the associated compilers and interpreters,
- restructuring tools,
- libraries,
- operating system,
- the tool environment, and
- the PAT_analyze system itself.

The *self knowledge* contained in the SKB, i.e., the knowledge about the capabilities of PAT_analyze itself includes not only structural, interface, and functional specifications, but also information about the degree of confidence in various system strategies, and the limits of its knowledge. In a performance state where the system does not possess a firm guideline how to proceed, the user will be solicited to either provide additional input or make a strategic decision.

3.2.2 Domain-Specific Knowledge

Domain-specific knowledge relates to *classes* of closely related applications or program systems, such as, say, for automatic differentiation or for the solution of partial differential equations.

With respect to such a class, domain-specific knowledge may include expert knowledge about existing approaches and their applicability to problems arising in practice, major solution strategies and their supporting libraries, key coding patterns and data structures. An important part of such information is the performance characterization of the various approaches, and in particular support for an efficient mapping to the system.

3.2.3 Application Knowledge

In the context of this section we discuss the information maintained by the system related to a *single* user application. This will include any domain-specific knowledge that may be available in the system. Additional knowlege can be classified into three categories: application components, performance characteristics, and execution knowledge.

Application Components

Many modern large-scale applications can be characterized as modular, multi-lingual, and multi-paradigm. Modules may contain legacy codes, and their number and structure may change dynamically.

One of the main concerns of PAT_analyze is the relationship between source programs and their compiled target programs, the optimization strategy used during compilation, and the auxiliary information characterizing various properties of source and target programs as well as their relationship. The relevant information about a specific source program may include:

- an interface specification,
- input and output assertions,
- the compilation environment,
- the target program,
- auxiliary information for the program and its components (functions), such as:
  - annotated syntax tree
  - symbol table
  - variable/type bindings
  - component interfaces
  - call graph
  - unit flow graphs
  - data flow graphs
  - dependence graphs
  - distribution, alignment, and affinity information
  - instrumentation

Performance Knowledge

Performance knowledge related to a specific application includes all of the following components:

- the performance properties of the system components used in the application
- domain-specific knowledge existing in the system that is relevant for the application
- user-provided specification of performance properties, problems, and related actions, and
- execution knowledge derived from application executions.

The *Execution Knowledge Base (EKB)* stores information about performance properties regarding one or more executions of an application under one or more given input data sets. This covers the complete range of performance information monitored, analyzed, and presented in the system. For a given program, parameter studies may be performed, which provide information about a controlled set of experiments based on varying sets of input data and environmental conditions.

An important component of the EKB is the information dealing with the *semantic gap* between source programs and their associated objects executing on the machine [1]. This is similar to the problem arising in the context of debugging, except that performance analysis has to deal with all degrees of compiler and runtime optimization, whereas a debugger can usually assume to work with an un-optimized program. With the emphasis on high-level programming languages this means that the relationships between source-level entities and their associated symbols, and the corresponding target entities needs to be maintained, under consideration of the restructuring performed by the compiler and runtime system.

3.3 The Introspection Subsystem

Sensors yield information about the execution of the application, which can be used by PAT_analyze. We interpret sensors in a very general way, covering the range from hardware counters at the lowest level all the way to application-level information about program behavior. Here are a few examples for sensors and the information provided by them:

- *Hardware Monitors* use a set of registers for low overhead access to hardware performance data that record information about events in processors, possibly together with the instruction addresses involved. Pat_analyze will support three classes of performance monitoring events: *accumulators* for simple counting of standard events, like cache misses, loads, and floating point operations; *timers* for analysis of latencies and stalls, such as cycles stalled waiting on a resource or cycles that a particular resource is idle; and programmable "*watch events*" for special conditions, which will provide the program counter (PC) and memory address when a particular event happens (these events can be programmable for sampling at pre-defined rates). Here are a few examples for the information that can be provided by sensors:
  - number of cache misses
  - number of floating-point operations
  - occurence of special arithmetic conditions
  - occurrence of synchronization events
  - waiting times at synchronization points
  - instruction execution pipeline or memory latencies
- *Low-Level Software Monitoring* at the message-passing level includes
  - waiting times for blocking send and receive
  - communication transfer times
  - barrier synchronization times
- *High-Level Software Monitoring* includes
  - timing for a function invocation, loop, or program region
  - the degree of parallelism exploited in a function invocation, loop, or program region
  - the time for computing a communication schedule at the beginning and end of an Independentparallel loop

In addition, summary and profiling information can be provided for any type of individual measurement. The above discussion shows that sensors cannot be simply interpreted as an array of independent entities but that they are organized in a hierarchy that extends to all levels of the system. Furthermore, higher-level sensors may support local analysis that can contribute to the reduction of the volume of information in the system, for example by filtering data.

4 Current Status

We are implementing the design described above in steps,such that we can provide productivity enhancements in the current offering of the Cray performance tools. These productivity enhancements target to direct the user to performance bottlenecks, providing hints to the user as to how to address them, instead of just reporting raw data. In addition, we are also automating parts of the data collection and analysis phases. The features described in this Section were already implemented in the Cray Performance analysis infrastructure and available for use today.

4.1 Automatic Load Imbalance detection

Scientific applications should be well balanced in order to achieve high scalability on current and future high end massively parallel systems. However, the identification of sources of load imbalance in such applications is not a trivial exercise. The current state of the art in performance analysis tools does not provide an efficient mechanism to help users identify the main areas of load imbalance in an application. To address this problem, we defined a new set of metrics to identify and measure application load imbalance [3]. These metrics present the user with information about how imbalanced their program is at various levels, and how much potential execution time can be saved if the imbalance is corrected. This information is presented both in text form, and through the visualization tool, Cray Apprentice[2]. The call graph of the application is presented to the user. Load imbalance is easily identified with the supplied color scheme, which allows the user to get an overall feel for how much, and where imbalance lies within their application.

4.2 Profile Guided MPI Rank Placement Suggestions

Grouping MPI ranks either together on a node, or spread across nodes can offer significant improvements in execution time. Communication within a node can take advantage of shared memory, while off-node communication must use the network. With the introduction of multiple cores per node, some applications benefit from balancing their computation and communication within a node so that memory bandwidth is not over-utilized. To assist the user with logical placement of MPI ranks, an enhancement was added to the Cray performance tools to provide placement suggestions based on profile information obtained from the application. The user is presented with information based on message size, or other possible metrics. The user can then determine if a non-default placement is better for their program based on data provided. The tool automatically creates the placement file that can then be used for successive runs.

4.3 Automatic Profiling Analysis

One of the first tasks that a user tackles when analyzing an application, is to determine where the application is spending most of its time. To do this, a profile is generated. Once this information is available, the user then wants to understand why time is being spent in certain routines. They typically set up a second experiment to collect further information. The Cray performance tools now offer a simple option to the instrumentation phase to instrument the application for creation of a profile. After application execution, this option causes the automatic creation of a template file which can then be used to gather further information. This template, which the user can view and edit, lists all functions for the application, with the functions that took the least time to execute commented out. It also includes options to request information such as HW counters. The user can use this template directly, edit it to refine the next collection of data, or use it to create new experiments for that application.

4.4 Discrete Units of Help

When performance measurement tools present data, it is often difficult for the user to find which data to focus on. Discrete Units of Help are intended to direct the user to meaningful data, such as the routine that exhibits the highest load imbalance. This enhancement, identified by a marker containing a '?', is introduced in the call graph of Cray Apprentice2 to quickly draw the user to the routine with the highest load imbalance. If the user clicks on the question mark, they are presented with the relevant data, an explanation as to what this data may mean and what the user could do to correct a potential problem. This enhancement is directed at the novice user who is either new to the tool, or is new to performance analysis in general.

5 Conclusions

Peta-scale systems will present significant challenges for performance tuning of applications with the current state of the art in performance analysis tools technology. In order to obtain high productivity on peta-scale systems, users will need a performance tools infrastructure that can automate the process of performance analysis, support heterogeneous architectures with multiple levels of parallelism, and scale to tens or hundreds of thousands of computing elements.

In order to address these issues, Cray has designed an automatic performance analysis framework, which will provide extensions to the existing performance measurement and visualization tools, as well as innovative techniques, based on performance models for automatic performance analysis. The key components of the Cray automatic performance analysis framework include automatic program instrumentation, a knowledge based system, and a infrastructure for introspection. The toolkit will provide high-level expertise as an aid to efficiently execute applications on the current and future Cray systems.

References

[1] Vikram Adve, John Mellor-Crummey, Jhy-Chun Wang, and Daniel Reed. Integrating Compilation and Performance Analysis for Data-Parallel Programs. In *Proceedings of Supercomputing'95*, November 1995.

[2] R. Bell, A. D. Malony, and S. Shende. A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Proceedings of Euro-Par 2003*, pages 17–26, 2003.

[3] L. DeRose, B. Homer, and D. Johnson. Detecting Application Load Imbalance on High End Massively Parallel Systems. In *Proceedings of Euro-Par 2007*, pages 150–159, September 2007.

[4] Luiz DeRose. The Hardware Performance Monitor Toolkit. In *Proceedings of Euro-Par 2001*, pages 122–131, August 2001.

[5] Luiz DeRose, K. Ekanadham, Jeffrey K. Hollingsworth, and Simone Sbaraglia. SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In *Proceedings of SC2002*, Baltimore, Maryland, November 2002.

[6] Luiz DeRose and Daniel Reed. Svpablo: A Multi-Language Architecture-Independent Performance Analysis System. In *Proceedings of the International Conference on Parallel Processing*, pages 311–318, August 1999.

[7] European Center for Parallelism of Barcelona (CEPBA). *Paraver - Parallel Program Visualization and Analysis Tool - Reference Manual*, November 2000. http://www.cepba.upc.es/paraver.

[8] S. Kim, B. Kuhn, M. Voss, H.-C. Hoppe, and W. Nagel. VGV: Supporting Performance Analysis of Object-Oriented Mixed MPI/OpenMP Parallel Applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2002.

[9] James Kohn and Winifred Williams. ATExpert. *Journal of Parallel and Distributed Computing*, 18(2):205–222, 1993.

[10] John Mellor-Crummey, Robert Fowler, Gabriel Marin, and Nathan Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–104, April 2002.

[11] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, November 1995.

[12] W. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. Vampir: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, January 1996.

[13] Felix Wolf and Bernd Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture, Special Issue 'Evolutions in parallel distributed and network-based processing'*, 49(10–11):421–439, November 2003.