KNOWLEDGE SUPPORT FOR PARALLEL PERFORMANCE DATA MINING

by

KEVIN A. HUCK

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

March 2009

University of Oregon Graduate School

Confirmation of Approval and Acceptance of Dissertation prepared by:

Kevin Huck

Title:

"Knowledge Support for Parallel Performance Data Mining"

This dissertation has been accepted and approved in partial fulfillment of the requirements for the degree in the Department of Computer & Information Science by:

Allen Malony, Chairperson, Computer & Information Science
Sarah Douglas, Member, Computer & Information Science
Michal Young, Member, Computer & Information Science
Marina Guenza, Outside Member, Chemistry

and Richard Linton, Vice President for Research and Graduate Studies/Dean of the Graduate School for the University of Oregon.

March 20, 2009

Original approval signatures are on file with the Graduate School and the University of Oregon Libraries.

An Abstract of the Thesis of

Kevin A. Huck for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken March 2009

Title: KNOWLEDGE SUPPORT FOR PARALLEL PERFORMANCE DATA MINING

Approved: ____________________________________
Dr. Allen D. Malony, Chair

Parallel applications running on high-end computer systems manifest a complex combination of performance phenomena, such as communication patterns, work distributions, and computational inefficiencies. Current performance tools compute results that help to describe performance behavior, as well as to understand performance problems and how they came about. Unfortunately, parallel performance tool research has been limited in its contributions to large-scale performance data management and analysis, automated performance investigation, and knowledge-based performance problem reasoning.

This dissertation discusses the design of a performance analysis methodology and framework which integrates scalable data management, dimension reduction, clustering, classification and correlation analysis of individual trials of large dimensions, and comparative analysis between multiple application executions. Analysis process workflows can be captured, automating what would otherwise be time-consuming and possibly error prone tasks. More importantly, process automation provides an extensible interface to the analysis process. The methods also integrate context metadata and a rule-based system in order to capture expert performance analysis knowledge about known anomalous behavior patterns. Applying this knowledge to performance analysis results and associated metadata provides a mechanism for diagnosing the causes of performance problems, rather than just summarizing results. Our prototype implementation of our data mining framework, PerfExplorer, and our data management framework, PerfDMF, are applied in large-scale performance studies to demonstrate each thesis contribution. The dissertation concludes with a discussion of future research directions.

CURRICULUM VITAE

NAME OF AUTHOR: Kevin A. Huck

PLACE OF BIRTH: Portland, Oregon

DATE OF BIRTH: May 2, 1972

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
University of Cincinnati

DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science,
2009, University of Oregon

Master of Science in Computer and Information Science,
2004, University of Oregon

Bachelor of Science in Computer Science,
1995, University of Cincinnati

AREAS OF SPECIAL INTEREST:

Parallel Performance Analysis
Data Mining
Knowledge Engineering

PROFESSIONAL EXPERIENCE:

Graduate Research Fellow, Neuroinformatic Center, University of Oregon, 2003 - present

Teaching Assistant, Department of Computer and Information Science, University of Oregon, 2002 - 2003

Senior Software Engineer, Southwest Financial Services, Ltd, Cincinnati, OH, 2001 - 2002

Senior Systems Engineer, Triple-I Systems, Inc., Cincinnati, OH, 1997 - 2001

Software Engineer, International TechneGroup, Inc., Milford, OH, 1992 - 1997

AWARDS AND HONORS:

Chuan-lin Wu Best Paper Award, International Conference on Parallel Computing (ICPP2005), June, 2005.

Upsilon Pi Epsilon Honor Society for the Computing Sciences, October, 2006

Distinguished Service Award, Computer and Information Sciences Department, University of Oregon, June 2008.

PUBLICATIONS:

K. Huck, A. Malony, R. Bell, and A. Morris, "Design and Implementation of a Parallel Performance Data Management Framework," in *Proceedings of the International Conference on Parallel Computing (ICPP2005)*, (Oslo, Norway), pp. 473–482, 2005. *(Chuan-lin Wu Best Paper Award).*

K. A. Huck and A. D. Malony, "PerfExplorer: A Performance Data Mining Framework for Large-Scale Parallel Computing," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'05)*, (Washington, DC, USA), IEEE Computer Society, 2005.

K. Karavanic, J. May, K. Mohror, B. Miller, K. Huck, R. Knapp, and B. Pugh, "Integrating Database Technology with Comparison-Based Parallel Performance Diagnosis: The Perftrack Performance Experiment Management Tool," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'05)*, (Washington, DC, USA), IEEE Computer Society, 2005.

P. Worley, J. Candy, L. Carrington, K. Huck, T. Kaiser, G. Mahinthakumar, A. Malony, S. Moore, D. Reed, P. Roth, H. Shan, S. Shende, A. Snavely, S. Sreepathi, F. Wolf, and Y. Zhang, "Performance Analysis of Gyro: A Tool Evaluation," *Journal of Physics: Conference Series*, vol. 16, pp. 551–555, 2005.

K. A. Huck, A. D. Malony, S. Shende, and A. Morris, "TAUg: Runtime Global Performance Data Access Using MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI)*, vol. 4192/2006 of *Lecture Notes in Computer Science*, (Bonn, Germany), pp. 313–321, Springer Berlin / Heidelberg, 2006.

L. Li, A. D. Malony, and K. Huck, "Model-Based Relative Performance Diagnosis of Wavefront Parallel Computations," in *International Conference on High Performance Computing and Communications (HPCC2006)*, (Munich, Germany), 2006.

K. A. Huck, A. D. Malony, S. Shende, and A. Morris, "Scalable, Automated Performance Analysis with TAU and PerfExplorer," in *Parallel Computing (ParCo2007)*, (Aachen, Germany), 2007.

D. Gunter, K. Huck, K. Karavanic, J. May, A. Malony, K. Mohror, S. Moore, A. Morris, S. Shende, V. Taylor, X. Wu, and Y. Zhang, "Performance database technology for SciDAC applications," *Journal of Physics: Conference Series*, vol. 78, June 2007.

Y. Zhang, R. Fowler, K. Huck, A. Malony, A. Porterfield, D. Reed, S. Shende, V. Taylor, and X. Wu, "US QCD Computational Performance Studies with PERI," *Journal of Physics: Conference Series*, vol. 78, pp. 24–28, June 2007.

V. Bui, B. Norris, K. Huck, L. C. McInnes, L. Li, O. Hernandez, and B. Chapman, "A Component Infrastructure for Performance and Power Modeling of Parallel Scientific Applications," in *Component-Based High Performance Computing (CBHPC 2008)*, 2008.

K. A. Huck, O. Hernandez, V. Bui, S. Chandrasekaran, B. Chapman, A. D. Malony, L. C. McInnes, and B. Norris, "Capturing Performance Knowledge for Automated Analysis," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*, 2008.

K. A. Huck, A. D. Malony, S. Shende, and A. Morris, "Knowledge Support and Automation for Performance Analysis with PerfExplorer 2.0," *Large-Scale Programming Tools and Environments*, special issue of *Scientific Programming*, vol. 16, no. 2-3, pp. 123–134. 2008.

K. A. Huck, W. Spear, A. D. Malony, S. Shende, and A. Morris, "Parametric Studies in Eclipse with TAU and PerfExplorer," in *Proceedings of Workshop on Productivity and Performance (PROPER 2008) at EuroPar 2008*, (Las Palmas de Gran Canaria, Spain), 2008.

A. Malony, S. Shende, A. Morris, S. Biersdorff, W. Spear, K. Huck, and A. Nataraj, "Evolution of a Parallel Performance System," in *2nd International Workshop on Tools for High Performance Computing* (M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, eds.), pp. 169–190, Springer-Verlag, July 2008.

ACKNOWLEDGMENTS

This research was sponsored by the U.S. Department of Energy Office of Science and by the National Science Foundation. I would like to thank my advisor, Prof. Allen D. Malony, for all his guidance and assistance. I would like to thank the other members of my dissertation committee, Prof. Sarah Douglas, Prof. Michal Young, and Prof. Marina Guenza. I would like to acknowledge my many collaborators, without whom this research would not be possible: Sameer Shende, Alan Morris, Wyatt Spear, Scott Biersdorff, Robert Bell, Karen Karavanic, John May, Brian Miller, Kathryn Mohror, Pat Worley, Shirley Moore, Boyana Norris, Van Bui, Oscar Hernandez, Barbara Chapman, Sunita Chandrasekaran, Lois Curfmann McInnes, and Meng-Shiou Wu. Finally, I would also like to acknowledge Li Li for her work on the initial design and implementation of PerfDMF.

DEDICATION

To my wife Laurel, my parents Lawrence and Paula, my brother Brian and my sister Amy. Thank you for your inspiration and your support.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

Parallel Computing And Scientific Discovery

Computational science, and indeed scientific discovery itself, is being driven by massive parallel computing. A current list of projects funded by The United States Department of Energy (DOE), Office of Science SciDAC projects include the areas of Physics (Computational Astrophysics, Quantum Chromodynamics (QCD), High Energy Physics and Nuclear Physics with Petabytes, and Turbulence), Climate Modeling and Simulation, Computational Biology, Fusion Science, Groundwater Reactive Transport Modeling and Simulation, Materials Science, and Chemistry.[139] A recent study by the DOE Office of Science stated "...six of the top ten recent scientific advancements in computational science [...] provide unprecedented insight into supernovas, combustion, fusion, superconductivity, dark matter and mathematics."

However, scientific advancement of these fields through the use of computational resources comes at a great cost. 'The Federal Plan for High-End Computing' report published by the DOE stated that High End Computing (HEC) resources are difficult to program efficiently:

> The current class of HEC systems is extremely difficult to program for or port applications to, such that without a heroic effort, applications rarely achieve more than a few percent of the peak capability of the system. The development of better programming tools (porting, debugging, scaling, optimization) would substantially improve this situation, enabling true performance portability to new architectures. These tools must become vastly easier to use, totally seamless (integrated into an IDE), completely cross-platform, and highly efficient when applied to applications running at the full scale of the system. [140]

The first issue to address is the size of the systems used for large scale parallel computing. The systems can have thousands of processors, and have complex designs. As shown in Table 1, the current top 5 supercomputers in the world [137] have over 100,000 computational cores (the smallest hardware unit capable of running one parallel process or thread). In addition, despite their hardware differences, the #2 and #5 machines (the XT4 and XT5 partitions of the Jaguar system) can actually be allocated as one larger combined machine. The large scale of these systems creates great challenges for both application developers who hope to execute their

TABLE 1: Top 5 supercomputers as of November, 2008. The computers are located at Los Alamos National Lab (LANL), Oakridge National Lab (ORNL), Lawrence Livermore National Lab (LLNL), and Argonne National Lab (ANL).

Rank	Location	Name	Architecture	Cores
1	LANL	Roadrunner	2 dual-core, AMD Opterons and 4 PowerXCell 8i per node	129,600
2	ORNL	JaguarXT5	quad-core 2.3 GHz Opteron per node	150,152
3	LLNL	BlueGene/L	dual-CPU PowerPC 440	212,992
4	ANL	Intrepid	quad-CPU PowerPC 450	163,840
5	ORNL	JaguarXT4	quad-core 2.1 GHz Opteron	30,976

software on large parallel computers, and for tool developers who are working to create measurement and analysis tools to aid in application development.

The second problem to address is the complexity involved in writing and maintaining parallel software. In addition to the many choices with regard to software development which any architect needs to make (such as programming language, compiler choice, system design, and others), there are many options which are unique to parallel programming, which all need to be made, but not necessarily in the order presented here. First, there is the choice of parallel implementation, such as whether to use threads, processes, some combination of the two, or use a modern programming language which postpones that decision until runtime. The hardware which the application will execute on will invalidate some choices, such as the option to use thread-only parallelism on a distributed machine without a single system image. If threads are used, the programmer needs to decide between thread implementations, such as pthreads or OpenMP. If processes are used, the programmer needs to integrate a communication and synchronization library such as MPI, PVM, or some other solution. These choices will have consequences with regard to the eventual performance of the application on current and future architectures.

Another software design decision which will have performance consequences is the choice of parallel model, such as master-worker, bag-of-tasks, divide-and-conquer (Single Program, Multiple Data, or SPMD), pipeline, some composition of these models, or some other model. In many cases, the algorithm used in the simulation may predetermine the model choice, but in others the decision may be left up to the developer. A choice which works well on one architecture or with one data set may not perform as efficiently when executed on another platform or with a distinctly different data set. Finally, the inclusion of scientific solver libraries may preclude one or more of the options above. For example, the differential equation library PETSc[100] requires the inclusion of an MPI implementation for parallel computation, and a linear algebra package such as BLAS-LAPACK[142].

In summary, the performance challenge is that of achieving efficient use of large scale HPC platforms. It is a challenge due to several factors: hardware, system software, parallel algorithms, parallel programming technologies, numerical methods, and application development and requirements. High efficiencies (relative to peak capabilities of the hardware) are difficult without comprehensive understanding of performance factors and behavior. There are limitations on parallel performance for each application, due to its inherent parallelism. Performance tools can help maximize the parallel performance potential of the application given the available parallel resources offered by the HPC system.

When this research was started in 2004, how did the high performance computing landscape look with regard to the current complexity challenge? A review of the Top 500 supercomputers in 2004 [137] shows that scaling was taking off with the construction of the first system with over 30,000 processors, the IBM BlueGene/L [82], and the 10,000 processor Columbia system at NASA [96]. These systems were a brute-force response to what had been the top supercomputer in the world, the Earth Simulator [39] which had "only" 5120 processors. These computers are complex, highly integrated systems with specially designed interconnects. The performance disparity between the speed of the processor and the time required to process memory operations (known as the "cpu-memory wall"[156]) was intensifying, and multilevel memory hierarchies with three or more levels of cache was adding to the complexity. Multicore, the fabrication of two or more central processing units (CPU) in one integrated circuit (die), had started with the introduction of the POWER4 processor from IBM. The POWER4 had been used in several systems in the top 500, including the ninth ranked IBM pSeries at the US Naval Oceanographic Office. Because two or more CPUs were competing for the same memory in a multicore processor, the cpu-memory wall on these systems became worse.

Unfortunately, as we will see in Chapter II, parallel performance tools were not dealing with the additional complexity of these systems. Tools did not offer a solution with regard to managing large scale performance data, and they could not adequately analyze that data. There was a high reliance on tool users to manage the analysis process, and there was no support for automatic performance diagnosis or reasoning. There was no support for identifying the system characteristics, such as whether multicore was present or not, so there was no way to explicitly and semantically relate performance characteristics to hardware properties or software design choices.

In summary, high performance computing can be used to make scientific progress, but the machines are large, complex and difficult to use efficiently. Parallel performance tools need to be aware of, and handle, all this complexity. Tool research has been limited in its contributions in large-scale performance data management, large-scale performance data analysis, automated performance investigation, and knowledge-based performance problem reasoning. A lack of automation ultimately leads to loss of knowledge, as successful analysis workflows are difficult to reproduce. Performance analysis without speculation requires the experiment context to be include in the analysis. Finally, individual performance experts do not scale - their expertise should be encoded in a form that performance tools can use.

Motivation

Parallel applications running on high-end computer systems manifest a complex combination of performance phenomena, such as communication patterns, work distributions, and computational inefficiencies. Tools that analyze parallel performance attempt to observe these phenomena in measurement datasets captured by instrumentation of the source code with timers, or by periodically sampling the program counter during runtime. The resulting datasets are rich with information, relating multiple performance metrics to performance variations and parameters specific to the application-system experiment. Performance analysis tools process these datasets to extract results that help to describe performance behaviors, such as hot spots and load imbalances, as well as to understand performance problems and how they came about. Unfortunately, parallel performance tool research has been limited in its contributions to large-scale performance data management and analysis, automated performance investigation, and knowledge-based performance problem reasoning. With the technology innovations in high-performance computing, it is critical now to move the field forward in these areas.

This dissertation presents a performance data mining framework which supports both advanced analysis techniques and extensible *meta analysis* of performance results. Parallel performance tools can be more effective when they are aware of specific aspects and properties of applications, parallel software, and high performance computing systems. Effective tools should provide a context-aware performance explanation, rather than just a descriptive performance summary. Thus, it will be important to integrate into a performance data mining process two key features: *metadata*, which describes experiment context, and *expert knowledge*, for reasoning about relationships between performance characteristics and behavior, related by the metadata.

Any performance data set has *context metadata* relating to the application, platform, algorithm, and related parallel performance problems that would be helpful in the analysis process. This context metadata is as yet not fully integrated into existing tools. Encoding this knowledge in some form that a performance tool can use would be a first step in developing new analysis techniques that include more information about the experiment than simply the measured performance data. Once this metadata is available, analysis methods which use it can be integrated.

In many parallel scientific applications, intimate knowledge of the computational semantics and systems environment is necessary to accurately reason and make conclusions about performance data. In order to develop intelligent analysis heuristics, this *expert knowledge* has to be encoded in a form that an analysis expert or engine can apply to the problem in order to identify correlations, locate causality, and otherwise make conclusions about application performance. Collecting and integrating this knowledge into an analysis tool is a complex and perpetual challenge.

In addition to the key inclusion of context metadata and expert knowledge, there are other data mining infrastructure components required. The use of *process control* for analysis scripting, *persistence* and *provenance* mechanisms for retaining analysis results and history are all important for productive performance analytics. However, the framework must also be concerned about how to interface with application developers in the performance analysis process. The ability to engage in process programming, knowledge engineering (metadata and inference rules), and results management are key to creating data mining environments specific to the developer's concerns.

Contribution Overview

There are five conceptual contributions to this work, along with prototype implementations, and application studies which demonstrate our work.

- **Parallel performance data and metadata definition and management.** We have designed a common schema for mapping parallel performance data from over a dozen formats, and designed portable, scalable data management strategies for managing large scale profiles. [51] We also developed a flexible schema for capturing context metadata from various sources to be used in the analysis process, and designed multiple avenues for the aggregation and storage of metadata. [56]

- **Data mining algorithms applied to parallel performance analysis.** In order to manage the analysis of very large scale performance profiles, we explored the use of data mining methods. [53] To reduce the data across metrics and across threads and discover classes of threads and processes, we used clustering, and we used random linear projection, Principal Components

Analysis and thresholding to reduce data across measured events. By computing regression correlation between parameters, we also found correlations within the performance data, and between the performance data and the context metadata.

- **Design of a systems framework to support performance data mining.** We designed a framework to provide flexible access to performance data for the purpose of applying data mining methods, performing comparative analysis, and constructing portable, lightweight views in order to provide selective access into a performance data repository. [53] This framework can and has been used to compare performance from multiple experiments in application and hardware benchmarking studies.

- **Techniques to automate data analysis and mining.** Recognizing the need to extend and automate the performance analysis process, we implemented a method for automating our data mining framework. [56, 55, 52, 11] We designed an extensible operation and data object interface for specifying and combining analysis methods in a workflow, and designed a mechanism for results persistence. We also integrated automatic storage and retrieval of analysis provenance data. This automation represents an extensible operation and data object interface that demonstrates a general approach to combining analysis methods in a workflow.

- **Techniques to incorporate knowledge rules and inferencing.** We explored the use of rule-based inference reasoning in order to interpret analysis results, and provide expert knowledge to the diagnostic process. [56] We integrated context metadata into the automated analysis process, and developed a rule base to diagnose parallel performance problems. [52] We also applied machine learning techniques to design a general-purpose recommender system for application parameters in order to optimize performance.

- **Development of prototype tools for performance data and metadata management (PerfDMF) and performance data mining (PerfExplorer v1/v2).** In order to perform analysis on large collections of performance experiment data, we developed PerfExplorer[53], a framework for parallel performance data mining and knowledge discovery. The framework architecture enables the development and integration of data mining operations that can be applied to large-scale parallel performance profiles. PerfExplorer is built on a performance data management framework called PerfDMF[51], which provides a library to access the parallel profiles and save analysis results in a relational database. The application is integrated with existing analysis toolkits, and allows for extensions using those toolkits. A redesign of PerfExplorer[56] integrated a script interpreter for analysis automation, and an inference rule engine for expert analysis of performance results and context metadata. We extended our use of the data mining tools to include classification of application behavior, and constructed a general purpose recommender framework for runtime selection of application parameters.

- **Application of PerfDMF and PerfExplorer to real parallel performance analysis studies and evaluation.** We have applied our tools to analyze the performance of over a dozen applications. At the end of each contribution chapter, we will overview an example performance study demonstrating an application of the concepts presented. In a separate results chapter, we will also present six analysis examples. The data mining capabilities were used to analyze large scale performance profiles of Miranda, a hydrodynamics simulation. Automation of phase analysis is used to analyze GTC, a large scale simulation of plasma particle interaction in a fusion reactor. Data mining, automation and knowledge engineering are used to analyze S3D, another plasma turbulence flow simulation. Two OpenUH related studies demonstrate automation and knowledge engineering to analyze a multiple sequence alignment in ClustalW and GenIDLEST, a flow modeling simulation. Finally, we demonstrate a recommendation system for linear solver and preconditioner selection in a PETSc simulation of driven cavity flow.

Thesis Outline

The presentation of this research is laid out as follows. A discussion of the previous research in related fields is in Chapter II. The following three chapters of contributions represent the foundation of our knowledge supported performance analysis framework. We will describe our data management framework in Chapter III. This framework consists of the database schema, programming API and related tools which are used as the foundation of the performance data repository. Chapter IV describes the original PerfExplorer performance data mining application, and its subsequent re-design as an extensible, programmable framework. Chapter V describes the data mining techniques used to analyze large scale performance data. The next two chapters outline the primary contributions of this work. Chapter VI will discuss the process control available in PerfExplorer, and describe the types of automated analysis available. Chapter VII will describe the use of a rule-based system and associated inference engine to help diagnose performance problems, and the use of machine learning to select optimal performance parameters. While each of the contribution chapters will include simple examples as a demonstration, Chapter VIII will provide more in-depth examples using the complete PerfExplorer analysis framework as applied to a number of performance analysis challenges. Finally, we will present our final discussion and conclusions in Chapter X.

CHAPTER II

RELATED WORK

The research work presented in the following chapters draws from a rich history of parallel performance evaluation. While correctness and accuracy are the primary goals of scientific software, the efficient and timely computation of results is also critical. Indeed, the field of parallel computing would not exist if the "time to solution" was not meaningful. Regrettably, perfect speedup from parallel solutions is very difficult to obtain. As a result, many tools and techniques have been developed to instrument, measure, and analyze the performance of parallel software. The following section will introduce previous work which is most relevant to our approach.

Performance Analysis Frameworks

A vast collection of analysis tools have been constructed, and techniques have been developed for the purpose of improving parallel performance . Entire books [60, 33] have been written on the subject. To construct a valuable analysis tool, successful current analysis methods should be supported, and new methods explored. The following discussion of these tools and their capabilities should give further insight into what new types of analysis should be considered.

The paper by Fisher and Gross [38] is not a tool paper *per se*, but a very basic introduction to teaching empirical parallel performance analysis, and a good guide as to what a performance tool should contribute to facilitate the analysis process. Before making a measurement and evaluation effort, the analyst should consider the goals of the computation. Time to solution is one measure of performance - computation efficiency is another. For many parallel programs, real time to solution is the most important measure. But when throughput and cost-effectiveness are important, they should be considered as well. The authors suggest using more than one input data set if it affects the application performance, and watch out for interference from the operating system or from other applications running on a shared system. To that end, we are proposing a framework which enables multi-experiment comparisons, and includes the application of hardware specific inference rules to interpret performance results.

HPM Toolkit [24] (now known as HPCT, the High Performance Computing Toolkit [1]) is a vertical solution for performance tuning applications written in C, C++ or Fortran, and executing on IBM Power3 and Power4 systems. The toolkit includes an instrumentation library, measurement libraries, and a performance visualizer. The visualizer shows source code linked to the performance results. Clicking on the performance result brings up a window which shows all relevant metrics. The results are colored red or green, indicating the possibility of a performance problem, or efficient use of resources, respectively. As is a common theme in motivating parallel performance tools papers, the author describes the classic complaint of how difficult it is to do accurate performance measurement without being an expert. It is difficult to achieve a high fraction of the theoretical peak performance. In addition, accessing hardware counters is a requirement for getting insight into how a code segment performs. Our research extends this work, creating a general purpose framework for all systems, with expert rules to interpret analysis results and increase the productivity of the application developer. Rather than focus one family of systems, we have developed rules to interpret parallel performance results from as broad a range of systems as possible.

The EMPS (Environment for Memory Performance Studies) [50] project is primarily concerned with making accurate predictions from empirical data. The overall goal is to do so without consuming a lot of time and effort, and without expertise. This work is based on efficiently stored, sampled memory traces, collected with the Dyninst API [103]. The authors contend that the primary performance limitation of an application is interaction with and contention for the memory subsystem.

Using a component called MetaSim, the authors characterize each subsection of an application's memory behavior by an extraction and summarization of the access patterns in each basic block of an instrumented program. This characterization is fed into the *Convolver* component, along with a profile of the machine from PMaC [102] MAPS benchmark, or is synthetically created (for non-existent machines). The final result is not a cycle-accurate simulation, but a statistical simulator.

The framework supports multi-processor memory hierarchies, and enables the examination of "what if" scenarios dealing with hardware changes. The simulator can predict changes in application behavior based on data redistribution. Event ordering is necessary in order to examine multiprocessor systems' cache behaviors, including mechanisms to detect false sharing. Due to the large size of performance traces, summarization is necessary to manage the data. Recurring data access patterns are detected as *signatures*. According to the authors, these are the only things necessary to characterize a program.

In contrast, our research includes reference rules to analyze memory performance, and evaluate the effect of excessive memory stalls. When dealing with parallel applications, examining memory system performance is not adequate in evaluating the performance of the parallel implementation. There are also factors including synchronization, communication, and other forms of data sharing. However, memory system performance is still an important factor in parallel performance analysis, as our research with the OpenUH compiler will show in Chapter VIII.

The PerfSuite collection of performance tools are described in [93]. The authors correctly argue that there is a dizzying array of complexity when evaluating parallel application performance. Options at the hardware level, compiler level, library level, application parameters and application algorithms all contribute to the total system configuration space. To help cut through this complex environment, PerfSuite is a suite of tools that will help in the performance analysis process. The PerfSuite tools try to provide analysis that is easy to use, comprehensible, interoperable, robust and simple. There are three command line utilities in PerfSuite. *Psinv* gets the information about a machine, *psprpcoess* helps with pre- and post-processing, and *psrun* collects application profile information without code instrumentation or recompiling. Psrun can show the performance data for the entire run, or broken down by process. PerfSuite uses DynaProf [92] to dynamically instrument binary code at runtime. The performance visualization tool, CUBE, is used to display the profile output. CUBE can also perform difference operations to show the performance improvement or degradation between one execution and another.

This collection of tools does not necessarily correlate metadata differences in experiments with performance. Metadata is collected, but not explicitly used in the analysis. In contrast, we propose including the metadata into the analysis, and using a rule base to explain the performance differences with regard to metadata differences. Our proposed methods support PerfSuite profiles, and have the capability of storing in a data repository.

The EXPERT tool [155] examines trace files, and searches for known performance problems, such as late sender, late receiver, barrier synchronization, idle threads, and so on. It creates a 3 dimensional tree structure, containing source location, problem type, and node or processor location. EXPERT can be used to analyze trace files from MPI and OpenMP applications. This is a similar analysis which we propose, but performed with profile data. In addition, the profiles can be archived, along with the analysis results for later study and comparison in performance regression testing.

SvPablo [111, 23] provides an instrumentation library which can be used to annotate software by hand, and also supports automatic instrumentation using the SvPablo Parser. The SvPablo library captures performance profiles and computes performance metrics. Full hardware counter support is provided by the PAPI interface. The SvPablo GUI displays the collected performance data linked with the application source code. The SvPablo browser provides a hierarchy of color-coded performance displays, including a high-level routine profile and source code windows. SvPablo also provides infrastructure support for users to conduct load balancing analysis and scalability studies. SvPablo was a successful early performance analysis tool, but has not kept pace with the leadership class systems in use today. The analysis available in the GUI is not scalable to thousands of threads of execution. In contrast, our proposed framework supports large scale profiles, and includes support for hardware counter analysis.

TAU (Tuning and Analysis Utilities) [117] is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, Java, Python. TAU is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements. All C++

language features are supported including templates and namespaces. The API also provides selection of profiling groups for organizing and controlling instrumentation. The instrumentation can be inserted in the source code using an automatic instrumentor tool based on the Program Database Toolkit (PDT) [80], dynamically using the Dyninst API, at runtime in the Java virtual machine, or manually using the instrumentation API.

TAU's profile visualization tool, ParaProf, provides graphical displays of performance profiles, in aggregate and single node/context/thread forms. The user can identify sources of performance bottlenecks in the application using the graphical interface. In addition, TAU can generate event traces that can be displayed with the Vampir [145], Paraver [31] or JumpShot [72] trace visualization tools. TAU is the measurement tool which is used to collect a majority of our profile data. ParaProf is a useful visualization tool, but is not capable of automated analysis, nor effective visualization of large scale profiles containing thousands of threads or more. Our performance data management framework has been integrated into ParaProf, providing an initial client consumer of our prototype framework. However, multiple profile comparison support is limited in contrast to our proposed parallel profile analysis framework.

Truong and Fahringer [138] describe SCALEA, which is very similar to TAU, but with less portability, and it only supports Fortran applications. Temporal overheads are organized as Data Movement, Synchronization, Control of Parallelism, Additional

Computation, Loss of Parallelism, and "Unidentified". Like TAU, SCALEA has targeted database support from the aforementioned ZENTURIO project, described later in this chapter.

The Paradyn parallel performance measurement tool [89] uses dynamic instrumentation to attach to a running application and start collecting performance data. The performance consultant in Paradyn uses the W^3 search model (why, where, when) to locate bottlenecks. Paradyn uses resource hierarchies (hardware, code locations, synchronization) to define performance locations. Currently, the resource hierarchy is defined as a tree structure, but it has been described as a matrix. Performance measurements are taken on intersections of the resource hierarchies (i.e. MPI process 14 running on compute15.neuronic.nic.uoregon.edu, line 345 of method foo(), send event from MPI process 14 to process 18), and the measurements are interpreted by the consultant to determine where additional instrumentation should be added. Paradyn can be used for real time monitoring of an application and can also be used for offline, post mortem analysis.

Until recent modifications, Paradyn failed to scale beyond a few hundred threads of execution, due to the collective reduction of performance data to a master process which controlled the runtime instrumentation and measurement of the application. The infrastructure also requires additional hardware resources to provide the data collection without excessive perturbation of the application, and modifications to the runtime environment of the application. While Paradyn is capable of runtime performance analysis, we propose a scalable, offline analysis approach.

Performance Databases

The use of performance measurement tools will inevitably lead to the obvious situation of needing a data management solution. Even the simplest parametric studies need something to manage, archive, and provide access to performance data. The need to manage data is a basic requirement of performance benchmarking activities, but is often accomplished in ad hoc ways. In this section, we will consider previous work related to managing and comparing parallel profiles, and also discuss some of the tools which suggest that they could benefit from data management solutions. These approaches also do not provide data persistence support, which we are proposing.

Both the Graphical Benchmark Information Service (GBIS) [141] and the more general Performance Database server (PDS) system [74] demonstrate the utility of a high-level access to a performance experiment repository that allows for meaningful queries without user-level knowledge of performance data storage details. These services are limited to providing web-accessible performance benchmark archives various architectures and benchmark applications. We intend to provide a common data management substrate as a robust part of the framework for the development of performance analysis tools, rather than as an online repository or archive of performance data.

Directly relevant to our work are the projects that utilize a performance database as a component of a performance analysis system, particularly for multi-experiment performance analysis. The SIEVE (Spreadsheet-based Interactive Event Visualization Environment) system [114] showed the benefit of a simple table-based structuring of performance data coupled with a programmable analysis engine. More sophisticated performance data models, such as found in Paradyn [89] and CUBE [122], allow a richer analysis algebra to be applied to multi-experiment performance information, although without explicit data repository support. Unfortunately, the dated database approach in SIEVE focused on supporting trace data. Storing trace data from thousands of processors and long running programs is not a scalable approach to parallel performance data management.

Similar to our work, the HPCToolkit [112] targets profile-based performance analysis. It is able to merge data from multiple performance experiments from XML files which are correlated with the program source and hyperlinked for analysis and viewing with the HPCView [87] tool. Performance data manipulated by HPCView can come from any source, as long as the profile data can be translated or saved directly to a standard, profile-like input format. To date, the principal sources of input data for HPCView have been sample-based hardware performance counter profiles. In addition to measured performance metrics, HPCView allows the user to define expressions to compute derived metrics as functions of the measured data and of previously-computed derived metrics. Our goals also include the ability to derive simple metrics, but also to provide a programmable interface for building analysis packages, such as those to compute complex metrics and archive the results. We propose to provide data management tools for profile data from HPCToolkit, and support the storage of both performance data and the derived metric analysis data. Performance data could also be exchanged with HPCToolkit through common partnership in the Performance Engineering Research Institute (PERI) project (see page 64).

The Prophesy system [129] uses a performance database to manage multi-dimensional performance information for parallel analysis and modeling. The database is a core component of the system, implemented using relational Database Management System (DBMS) technology and storing detailed information from the Prophesy measurement system and performance modeling processes. The Prophesy analysis has been applied to both parallel and grid applications. Data submitted to the Prophesy server is accessible from a web interface, and models of the application can be constructed from the data using methods such as curve fitting, parametric modeling, and kernel coupling. The performance data is stored in a database schema in the form of four information trees: application, executable, run and performance statistic information. Basic performance data is stored, similar to other performance databases. Prophesy is a targeted solution, and does not make the database solution available as a separate component. Prophesy is also engaged in the PERI project. Using the exchange, our data management solution could enable open access to the archived performance data and provide a programming interface for building multiple analysis components. This could allow Prophesy's modeling algorithms to be captured as part of a broader analysis library. In this way, several performance tools could benefit from the advanced modeling analysis Prophesy provides. Like the differences between our proposed work and the GBIS and PDS systems, we propose a data management solution which can be integrated into analysis tools, rather than a web accessible data archive.

The PPerfDB project [48] comes closest to sharing the broader objectives of our data management work. PPerfDB was developing methods for diagnosing the performance of large-scale applications using data from multiple executions over an application's lifetime. It supports the import of performance data produced from multiple sources and allow performance results to be exchanged and compared across geographically disperse sites. Performance information is related through hierarchical property, resource, and event mappings that enable PPerfDB to support powerful comparison and analysis operations. The PPerfXchange component enables distributed PPerfDB-enabled performance repositories to interoperate (see PPerfGrid [18]). The thesis by Colgrove [17] describes the implementation of PPerfXchange, an application for performing queries against data from geographically dispersed data sources.

The PPerfDB architecture provides the opportunity for other data management systems to co-exist in a performance analysis environment. Rather than focus on a single schema for storing parallel profile data, the authors propose a solution in which disparate data stores are linked through web services using ontologies. We propose that the community is better served by supporting one common storage format, but these proposed solutions are complimentary. Our solution might be used, for instance, to store high-volume TAU performance results for an application suite, while supporting PPerfXchange-compatible interfaces that tie the performance data to a global PPerfDB system.

Karavanic [66] describes how multi-experiment performance data is managed to encompass executions from all stages of the lifespan of an application. Here all experiment information is gathered in a *program space* which can be explored with a simple naming mechanisms to answer performance questions that span multiple program instances. With this interface, it is possible to describe differences between two runs of a program, both the structural differences (differences in program source code and the resources used at runtime), and the performance variation (how were the resources used and how did this change from one run to the next). As our work also demonstrates, the ability to easily access performance data history and comparatively process the data has high payoff for automating performance diagnosis, although this work does not propose a data management solution.

PerfTrack [65] is a newer extension and application of the technology described Karavanic to include database support. The performance context data is primarily concerned with the resource hierarchy. A resource is any named element of an application or its compile-time or runtime environment. Resources can have attributes. Performance results are tied to a resource hierarchy focus (i.e. the context) and are a measurement of a specific metric. The PerfTrack work was started after publication of our initial data management solution, and shares many of the same goals, but with an emphasis on exhaustive collection of metadata fields. Like HPCToolkit and Prophesy, PerfTrack is participating in the PERI effort to exchange performance data.

ZENTURIO [108] is a performance experiment management system that incorporates an experiment data repository at the core of its architecture. While ZENTURIO shares features of Prophesy, PPerfDB, and HPCToolkit, it is remarkable for its implementation as a set of services for experimentation and analysis, with a graphical portal for user interaction. In comparison to all of these efforts, we are specifically advocating a performance data management system component and analysis programming interface that is flexible for a broad range of applications and is based on open, standard implementations and reusable, pluggable toolkits. We believe such an open framework could address much of the data management functionality present in these tools, and provide common benefit.

Performance Analysis Using Data Mining

Just as the need for data management flows naturally from the collection of performance data, the need for scalable analysis support for the data quickly becomes apparent. Terascale, and soon petascale, computers have over 100,000 computational cores, and large scale data collection results in hundreds of thousands of performance profiles. Extensive parametric studies also can generate overwhelming amounts of data. The field of data mining has strategies for both guided and exploratory data analysis and data reduction. The techniques are certainly applicable to performance data, and a number of previous authors have explored the use of data mining strategies.

The SimPoint project [119] is not used for parallel application simulation, but for long running applications in general. The focus of the project is in building compact, accurate simulations, but there is much to apply to parallel applications. This paper represents application behavior as *basic block vectors*. In a basic block vector, each dimension represents the percentage of time spent in a region of code. The application is split into slices along the time axis, and the slices are compared to each other, and summarized using clustering. They calculate the distance between the points using both Euclidean and Manhattan distance. Manhattan is used to calculate the distance matrix, and Euclidean is used for the clustering. Random linear projection is used to project the data down to 15 dimensions. To choose k for the k-means clustering, they use the Bayesian Information Criterion (BIC) [67], after clustering from 1 to 10 clusters. In contrast, we have taken the idea of the basic block vector, and modified it to fit our needs. In our case, each individual in the cluster data set is one thread of execution, and the attributes of the individuals are made up of the time spent in each measured region of code. Because there is no known method for choosing the optimal k (BIC does not guarantee an optimal solution), we perform the k-means clustering clustering using k values of 2-10, and then visualize the results for the user to interpret, rather than rely on a heuristic to choose k. We have also examined using the Gap statistic [136] for evaluating cluster results, but with each of these methods, the data is assumed to be a mixture of Gaussians, which is often not the case with parallel profile data.

Vetter and Reed [146] discussed their use of Projection Pursuit to reduce parallel performance trace data. They do the projection to reduce dimensionality within a sliding window of a trace. That is, profiles are generated from data within a window of the trace, and the window is moved along the time axis of the trace data, and a time-series profile is generated. Various methods of data preprocessing used include shooting, trimming, centering, sphering, then the projection pursuit.

Extending this work, Ahn and Vetter [4] observed that current data collection and analysis techniques implement the extensive use of hardware counters, but the sheer amount of data is overwhelming, and new methods of analysis are needed. Automated performance analysis tools are needed to sort through the massive data sets, recognize important features, identify under-performing parts of the application, and prescribe solutions. The authors introduce the use statistical clustering of parallel performance data to reduce the amount of data collected. Data is organized as multidimensional vectors, where each vector represents one processor, and each dimension represents a hardware metric, such as floating point operations, wall clock time, or instructions issued. Principle Components Analysis (PCA) is used to show that several hardware counters capture redundant data, and shouldn't be collected due to the increased overhead. The authors also mention the use of F-ratio and factor analysis to help describe the types of clustering seen from the data. Again, our work differs in that we are clustering using detailed profile data, so the attributes which make up our instances are the measured regions of code, rather than the metrics captured for the entire run. We have also extended this work to use PCA on the attributes as a form of dimension reduction. However, we found that PCA was not ideally suited to this application (see page 101).

Two papers by Clement and Quinn [15, 16] describe the use of linear regression models for parallel performance modeling and prediction. The original data is parallel performance trace data. The authors stated that the advantages of multivariate statistical analysis include standard error values and confidence intervals, automated fit, and correlation of model to empirical data. The authors state that parallel models should allow a user to extrapolate performance for larger problem sizes and larger numbers of processors, quantitatively measure the sensitivity to system parameters as the problem and processor sizes vary, and should account for paging memory behavior. In contrast, we propose that our solution can directly analyze large scale profiles, rather than estimate their behavior using smaller profiles.

The authors' method is to build a call graph for the application, including all loops and subroutines. They then try to determine if the numbers of loops can be ascertained from a) symbolic information (such as program constants), b) from the size of the problem, c) from the size of the processor count or d) if the number of loops is constant. The authors are seeking equations that are linear with respect to the speed of the hardware. The types of hardware counters collected include floating point operations, emulation loops, cache misses, messages sent and bytes transmitted. The author's assumption is that both the variables and the model errors are statistically independent. They also conclude that outliers are trouble, so they remove them before analysis. Having a hardware focus, the authors claim to be able to characterize important performance indicators equally well from application to application. In contrast, we believe that outliers should not be eliminated, but rather they are the deviant case with respect to performance, and should be identified and measured with regard to their effect on overall performance. We acknowledge the authors' insight in linking symbolic information (metadata) with performance data, identifying the effect that those parameters have on overall performance.

Analysis Process Automation

From a software engineering perspective, an analysis solution to aid the user in exploratory and guided data analysis requires the ability to extend the capabilities of a data mining framework, and reuse beneficial process workflows. The ultimate goal is to capture the performance analysis process, and intelligently automate it as much as possible, while still providing manual control where desired. In this way, analysis processes which proved useful for one data set could be reused for new data sets, and the user would be presented with a truly customizable and integrated suite of performance tools. In this section, we will describe work related to providing custom control and automation over analysis workflows.

The myGrid project [3, 123] links bioinformatics analysis processes together using web services. The authors started with Web Services Flow Language (WSFL), and then went on to write their own workflow language, SCUFL.The language is processed by their own engine, known as Freefluo.Overall, they provide a graphical development environment which links web services and data sources together into a workflow. In addition, when elements of the pipeline or the input data changes, the services are re-run to reflect the changes to the input. The system stores annotation, metadata, ontologies and provenance with the input data.

The Hyperthesis project [10] uses a visual programming environment to connect bioinformatics resources which have web services interfaces. The framework links these services, and combines the output from one service as the input to another service in order to chain research. This framework provides manageable, efficient reuse of services and data. they integrate geographically distributed, heterogeneous biological data sources and bioinformatics tools. The project is divided into two pieces, the workflow components and the execution engine.

The Pegasys project [116] is a Java integration of bioinformatics services, with a QT user interface for visual programming of analysis pipelines. The key principles of the project are modularity, flexibility and data integration. The analysis workflow is modeled as a directed, acyclic graph, and stored as an XML document. At the completion of each step, the results are stored.

The myGrid, Hyperthesis, and Pegasys projects are frameworks which link web services together, and are quite different from what we are attempting with regard to application automation. However, we are interested in the idea of a visual programming environment, and hope to eventually provide that layer to our scripted automation. We are also considering changing our automation model into a data-flow model, in which the performance analysis results would be regenerated when either the input data is changed, or the analysis components are changed. When a parametric study is constructed in our framework, and new data is collected, an update process would be initiated which would re-run analysis in order to incorporate the new data.

The Weka [154] data mining toolkit has a useful pipeline development user interface called the *Knowledge Flow* interface. With the Knowledge Flow interface, users select Weka components, place them on a layout canvas, and connect them in a directed graph that defines the processing and analysis of data. All of the classifiers, filters, clusters and visualization tools available in Weka can be linked together in a process flow. Again, this is an example of the type of data flow model which we are considering for the framework.

Several papers [124, 81, 90, 120, 43, 49, 2] describe workflow in the context of business logic and process automation. Linthicum [81] defines *process automation* as "the passing of information from participating system to participating system and the application of appropriate rules in order to achieve a business objective." While seemingly similar, process automation is not really relevant to our research, as it deals with static workflows which do not often change from iteration to iteration. This differs from our focus on pipeline management, as the input data, analysis methods, and desired output often change from one experiment to the next.

Cunningham et al. [19] describe a programming interface between Java and Python, similar to Jython. The authors point out that Rapid prototyping is easier with scripting languages, and an iterative process is easier in the interpreter. Once the control logic is determined, the code can be re-implemented in a permanent language, if desired. Also, the two-language design allows for a scripting interface to the Java application. A similar interpreter is described in [58]. We agree with this stance, and as described in Chapter VI, we have integrated a script interpreter into our framework. This does not prevent us from adding a semantic layer on top, in the form of a visual programming environment.

A trio of articles by Laddad and Kearns [73, 68, 69] introduce suggestions for integrating scripting support into applications. [73] is an excellent overview of integrating script interpreters into an existing Java application. The article includes a design for providing a common scripting interface to the user, so that any script interpreter can be added. The interpreters supported include JPython for Python, Rhino for JavaScript, and Jacl for Tcl. The author measured the performance of each of the interpreters tested (Jacl, Jython, Rhino, BeanShell), and determines that Jython is the fastest. All of the interpreters are easy to integrate and are equally suitable. This work guided our integration of scripting into our framework, although we decided to support just one scripting language. If desired, the script infrastructure could be replaced with another language.

Knowledge Engineering

With respect to our research, there are two areas of knowledge engineering which we will be exploring in this section. First, work has been done in the application of software systems which attempt to automatically diagnose performance problems. Some approaches include the use of rule-based systems and the use of inference rules. Second, we will explore previous work in parametric optimization, where software recommender systems have been built for targeted purpose. We will discuss these systems within the context of our work to build a general purpose parametric recommender system.

KappaPi [28, 29] (Knowledge-based Automatic Parallel Program Analyzer for Performance Improvement) and KappaPI2 [64] are tools which use trace files from PVM and MPI applications, detect known performance bottlenecks, and determine causes by applying inference rules. The causes are then related back to the source code and suggest recommendations to the user. In contrast, we propose to focus on profile data, and to analysis a broader range of parallel applications, not just distributed applications.

As described earlier in this chapter, Paradyn [89] utilizes the Performance Consultant [66] and Distributed Performance Consultant [113] for run-time and offline discovery of known performance problems. The latest version of the Performance Consultant uses historical performance data to help guide bottleneck detection. While the Performance Consultant does include contextual information about the runtime environment to help explain performance differences, there doesn't appear to be a mechanism for including additional expert knowledge about the application, such as data or event relationships. And like the aforementioned tools, the Performance Consultant's strength is in diagnosing known performance problems, rather than general performance characterization which we are proposing.

JavaPSL [34] is a Java Performance Specification Language, designed to be used to specify techniques for searching for known performance problems such as poor scaling, load imbalance, and communication overhead. The specification language could be useful in the application of search heuristics in a particular diagnosis process, and represents a good example of the type of low-level analysis whose results could be used in conjunction with expert knowledge and context metadata to suggest the causes of performance phenomena. Rather than perform diagnoses using a procedural or imperative programming design, we have decided to use a declarative programming paradigm. As described in Chapter X, we eventually plan on encoding the PSL diagnostic logic as rules in our rule base.

Performance Assertions [147] have been developed to confirm that the empirical performance data of an application or code region meets or exceeds that of the expected performance. By using the assertions, the programmer can relate expected performance results to variables in the application, the execution configuration (i.e. number of processors), and pre-evaluated variables (i.e. peak FLOPS for this machine). This technique allows users to encode their performance expectations for regions of code, confirm these expectations with empirical data, and even make runtime decisions about component selection based on this data. The use of performance assertions requires extensive annotation of source code, and requires the application developer's experience and intuition in knowing where to insert the assertions, and what kind of performance result to expect. In contrast, we propose to build the performance analysis and diagnostic logic in an analysis tool, rather than in the runtime measurement system. We also hope to avoid the excessive manual instrumentation necessary with this technique.

EXPERT [122], from the KOJAK [70] project, is an automatic event-trace analysis tool for MPI and OpenMP applications. It searches the traces for execution patterns indicating low performance and quantifies them according to their severity. The patterns target both problems resulting from inefficient communication and synchronization as well as from low CPU and memory performance. EXPERT searches for known problems, rather than focusing on characterization and new problem discovery. Also, the performance data analyzed is trace data.

CUBE [155] is a graphical browser suitable for displaying a wide variety of performance measurements for parallel programs including MPI and OpenMP applications, and is the primary analysis viewer for SCALASCA [42], a parallel implementation of the EXPERT trace analysis methods. CUBE implements Performance Algebra [66], a technique for performing difference, merge and aggregation operations on parallel performance profile data. While CUBE provides a powerful interface for visualization and exploratory analysis of the differences between two performance data sets, there is no mechanism for linking the performance behavior to the performance context, and providing the user with a meaningful explanation of why the performance differs between the two profiles. And as mentioned earlier, we propose an extensible analysis framework which uses profile data and context metadata.

Hercule [78, 77, 75, 76] is a parallel performance diagnosis tool which uses the expert system CLIPS [95] to process computational model-centric rules which can diagnose common performance problems. Hercule operates in four stages. In the first stage, *behavioral modeling*, the parallel computational algorithm in the application is an abstracted to major event transitions, and a simple model is constructed. In the second stage, the behavioral model is used as a basis to construct a *performance model*, which identifies performance properties in the behavioral model. The third stage, *model-specific definition*, is the process of defining hardware counter metrics within the context of the defined model, rather than as raw metrics. Finally, *inference modeling* is the mapping of performance data to the abstracted behavioral model. Hercule's rule base include the set of abstract events, performance metrics, and performance factors for individual parallel models.

Beginning at the root of the application algorithm, the inference system evaluates performance metrics at the highest granularity, and if a performance anomaly is detected, effectively "drills down" to the next level of algorithm abstraction, and evaluates each of the major steps of the algorithm. At each step, the worst behaving step in the algorithm is identified, and the process continues, breaking that step of the application into its composite sub-steps, and evaluates the performance metrics at that level. Using this process, Hercule can define symptoms of known parallel application problems, such as *load imbalance*, *insufficient parallelization*, and *communication overhead*, and offers possible solutions for correcting these known problems. In contrast to our research, Hercule operates on performance traces, rather than profiles, and requires both the identification of the parallel model used by the application and the construction of the algorithmic model. Hercule also requires specific hand instrumentation of the source code, in order to identify regions of the code which represent aspects of the parallel model. Our goal is to apply similar inference rule processing to performance profiles, and provide it at a more general use level, searching for known parallel inefficiencies throughout an application.

Vuduc et al. [148] describe a technique for finding optimal parameters for BLAS-like [25] libraries. In general, the parametric search space is searched and pruned in order to find the optimal solution. The authors acknowledge that finding the globally optimal solution is difficult, and could require an exhaustive search. The authors discuss an *early stopping criterion* based on the idea that even when the complete performance space cannot be otherwise modeled, there is access to some of the information characterized by the statistical distribution of the performance space. There are three methods considered, a cost-minimization approach, a regression model approach and a support vector method, something akin to generating equivalence classes to eliminate poorly performing parameter combinations. The support vector method was most successful in finding *an* optimal solution, but it takes much longer than the other two methods, regression and cost-minimization. In contrast to our research, this approach is designed to perform offline tuning of a solver library for given hardware. We propose to use a classifier to provide runtime recommendations for scientific components using application parameter data and hardware characteristics. However, this work provides insight in a training set generator, which would reduce the amount of data collected in order to construct an accurate classifier.

Bhowmick et al. [9] used machine learning to construct a recommender system for linear system solvers of sparse matrices in PETSc applications. PETSc, the Portable Extensible Toolkit for Scientific Computing, is a large parallel library with data structures and methods for constructing scientific applications. There are many linear solvers (and their respective options) in PETSc, and depending on the characteristics of the input matrix, some methods will solve faster or more accurately than others, and some may not converge to a solution at all. A recommender system would help eliminate the guesswork when choosing a solver for a given situation. The system was constructed as a binary classifier. A representative sample of sparse matrices were exhaustively solved with a set of the solvers available in PETSc, and this data was used to train the classifier. Each solution was rated as “good” or “bad”, based on whether it was “better” (faster, more accurate, and so on) than the default solution. Using three-fold cross validation, the authors validated the accuracy of their classifier.

Eijkhout and Fuentes [27] expanded the discussion of this work as it was included in Salsa, the Self-Adapting large-scale Solver Architecture. In that paper, the authors formally described their method, which is to build binary classifiers which classify solvers as “good” or “bad”, so sets of solvers are selected from multiple classifiers (convergence, time, accuracy, preconditioners, and so on), and the results are convolved. The proposed solution is limited to matrix solvers.

We are extending this work in order to construct general purpose component recommender systems for the PETSc library. Our classifier is constructed differently, as described in Chapter VIII, but the concept is the same. Training data is used to identify an "optimal" non-linear solver component which will converge without failure. In addition, we are capturing per-iteration performance data, so that linear solver components can be replaced at runtime, when convergence slows down, or when the state of the problem changes enough that another solver component will complete the convergence in less time, or more efficiently.

CHAPTER III

PERFDMF

Overview

Performance evaluation of parallel programs and systems, whether for purposes of benchmarking or application tuning, requires the analysis of performance data taken from multiple experiments [35, 66, 122]. While sophisticated tools exist for parallel performance profiling and tracing, allowing in-depth analysis of a single execution run, there was significantly less support for the processing and storage of multiple performance datasets generated from a variety of experimentation and evaluation scenarios. It might be expected that each performance tool solve the problem of multi-experiment performance data and results management individually. One can argue this is neither a reasonable expectation, since resources may be unavailable to build such support for some tool projects, nor a desired one, given the potential for building incompatible solutions. Instead, to promote performance tool integration and analysis portability, and to foster a multi-experiment performance evaluation methodology in general, we were motivated to develop open performance data management approach that can provide a common, reusable foundation for performance results storage, access, and sharing. Such an approach could offer standard solutions for how to represent types of performance data, how to store performance information in a manageable way, how to interface with the performance storage system in a portable manner, and how to provide performance information services to a broad set of analysis tools and users. A performance data management system built on this approach could serve both as a core module in a performance measurement and analysis system, as well as a central repository of performance information contributed to and shared by several groups.

This chapter presents the design and implementation of the parallel Performance Data Management Framework (PerfDMF). PerfDMF addresses critical requirements in the TAU project for the parsing, storage, and processing of multi-experiment performance measurements and results. However, our broader goal with the PerfDMF project is to provide an open, flexible framework that can support common performance management tasks and be extended and re-targeted to enhance performance data integration as well as reuse across performance tools used in the parallel computing community.

Design Issues

There were a number of issues which we tried to address with the PerfDMF solution. The first issue is that of *portability*. We wanted our solution to support multiple platforms, different computing models and languages, alternative profile formats, and relevant database management systems (DBMS). Prior to our work in designing a data management framework, performance analysis tools were designed to work in isolation, and not as candidates for integration with each other. In addition, prospective analysts could be constrained by hardware and software policy decisions at their respective research centers. Our data management framework is intended to broadly support many profile formats and tools, so we sought to eliminate hardware and operating system requirements for adaptation of the framework.

For starters, we decided to write PerfDMF in Java [125]. The Java runtime system is widely ported to nearly all workstation and server architectures, and is widely supported. Java provides the Java Database Connectivity (JDBC) interface, which is a common database interface for interacting with most widely used DBMS. Supporting a new DBMS ideally requires only linking with a new Java Archive (JAR) file, but sometimes requires minor changes to the code.

Supporting many different profile formats in the database was a rather easy to do, as profiles from different data collection tools have pretty consistent formats, as we will describe in this chapter. However, in order to store the data from different performance measurement and analysis tools, we had to write data parsers for those formats. Supporting many computing models and languages was also automatic, as there are mainly three classes of parallel applications: multiple processes which communicate, multithreaded applications with a shared memory space, and a third class, which is a mix of those two. As long as these three classes are supported, then all current parallel computing models and languages will be supported.

Another key issue for PerfDMF is that of *scalability.* Some parallel computers today have over 100,000 processors, and each of those processors can execute applications with hundreds or thousands of functions, each of which can be measured with multiple hardware counters. Any database solution should be capable of handling data sets with hundreds of millions of individual data points. In addition, comparing many trials in a parametric study will require querying and analyzing data from each of these potentially large data sets. To address the scalability concerns, we chose to only store profile data, which is in itself a reduced data format (as compared to full event traces). A discussion of the difference between profiles and traces is found later in this chapter. In addition, to speed up query response times, we stored aggregations of the data in the database itself, so that commonly used aggregations would be precomputed in separate tables.

The final issue we tried to address was that of *reusability.* From a tool engineering perspective, if the important profile data management features and functions could be captured in a common framework, performance tools could incorporate the framework in their design and interoperate with other tools that use the framework, resulting in several advantages for both enhancing performance tool capabilities and improving tool deployment. By providing an open source solution, we made the code available for others to use and extend as they see fit. By writing data parsers for commonly used profile formats, we eliminated a barrier of entry for potential users who do not use TAU as a data measurement tool. We designed the data management tables to be extensible, with as many or as few columns as the user requires. And as we describe on page 60, our flexible XML format ensures forward compatibility for potential future metadata fields.

Approach

Empirical performance evaluation of parallel and distributed systems or applications often generates significant amounts of performance data and analysis results from multiple experiments and trials as performance is investigated and problems diagnosed. However, the management of performance data from multiple experiments can be logistically difficult, impeding the effective analysis and understanding of performance outcomes. PerfDMF provides a common foundation for parsing, storing, querying, and analyzing performance data from multiple experiments, application versions, profiling tools and/or platforms. The PerfDMF design architecture is presented in this section. We describe the main components and their interoperation. Attention is also given to the profile database schema as the core of the PerfDMF database support.

Parallel Profiles

When collecting performance data, there are two measurement methods used, *tracing* and *profiling*. Tracing collects application events, including communication events, in an "event trace". Each thread of execution is responsible for collecting its relevant event data, usually the event start and end timestamps, and in the case of communication events, the sender(s) and receiver(s) for the event. At the end of the application execution, the event data is dumped to disk. After the dump is completed, all of the trace data files are merged together in an time-ordered manner, and a full application trace is available for analysis. The great benefit of tracing is its temporal and spatial resolution – every event that occurs is recorded in a trace record. However, this fine granularity comes at the price of large data volume. In most cases, tracing is not scalable for long-running applications, or for applications with more than a trivial amount of processors.

In contrast, profiling aggregates the individual measurements along the time axis into a summary of all the time (or some other metric) spent in a measured region of code, or event. For example, if the event is a function, then the profile measurement includes the number of times that function was called, and the total amount of time spent in that function. More formally, each profile measurement is the sum of the total amount of time (or some other metric) spent in each application event, or:

$$\forall e \in \mathbf{E} : \sum_{i=1}^{numcalls} e_{end_i} - e_{start_i}$$

where e is an event and $\mathbf{E}$ is the set of calls to that event executed on that thread of execution. Like tracing, each thread is responsible for capturing its own time spent in each measured event. At the end of the application execution, the profile data is dumped to disk. The data is sometimes merged into one file for convenience. Profiling is a far more scalable solution in comparison to profiling. Because profile data is aggregated along the time axis, the amount of data collected does not increase as the application executes. However, there is still the problem of comparing the performance of hundreds or thousands of threads of execution without tool support.

In order to collect this performance data, whether traces or profiles, some interruption of the application is necessary. There are two primary methods for doing this, either *direct measurement* or *sampling*. The first requires the use of instrumentation where measurement code is placed directly in the code. For instance, to measure the time spent in a code region, instrumentation at the beginning and end of the region measures the time of the begin and end events, respectively, and computes the difference. The instrumentation can be added at the source code level prior to compilation, or it can be added to the compiled binary either before runtime or during runtime, using binary rewriting. There are advantages and disadvantages to each method [118]. Source level instrumentation is completely portable, allows the insertion of timers anywhere in the program, and provides the semantic connection between the measurement and the source code location. The problem with source code instrumentation is that it requires access to the source code and the time and ability to recompile the code. Binary instrumentation does not require recompiling and therefore also does not interfere with compiler optimizations, but it is not widely available on all platforms.

The other data collection method is sampling. When sampling the application, the running application is periodically interrupted, and the application stack is evaluated, capturing the location of the program counter. After a number of such periodic data collections, a profile histogram develops, suggesting where the application is spending its time. It is important to note that the time (or other monotonically increasing metric) between interrupts is assumed to have occurred *entirely* at the halted location.

With either data collection method, data can be evaluated either *post mortem*, after the application has finished, or at runtime. Regardless of the method used, the application experiences execution intrusion as a consequence of measurement overhead, and intrusion can cause perturbation. This perturbation can be minimized through selective instrumentation or measurement, and can also be compensated for [85].

Components

PerfDMF consists of three main components: *profile input/output*, *profile database*, and a *database query and analysis API*. Figure 1 shows a representation of these components, and their relationships. The `DataSource` objects are responsible for parsing various performance profile formats, including retrieving data from the

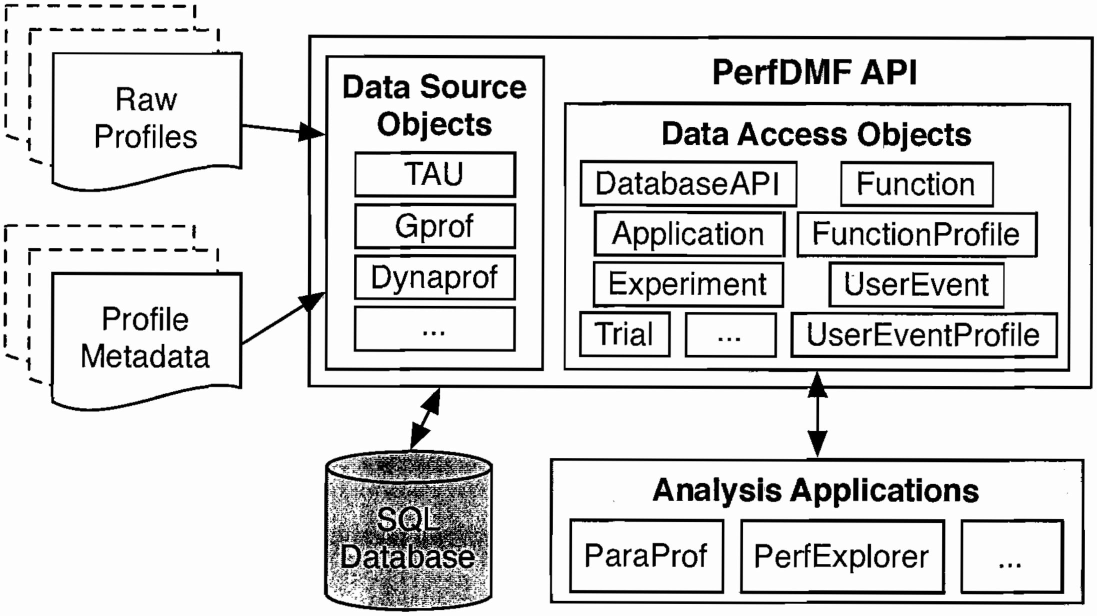


FIGURE 1: TAU PerfDMF Architecture.

database for some applications. The relational database schema is designed to store the profile data in various Structured Query Language (SQL) databases. The Data Access objects provide the query and analysis API for requesting performance data from the database, performing some filtering and aggregation operations, and for storing performance profile data to the database, including derived data from analysis applications.

Profile Input and Output

PerfDMF is designed to parse parallel profile data from multiple sources. This is done through the use of embedded parsers, built with PerfDMF's data utilities and targeting a common, extensible parallel profile representation. Currently supported profile formats include TAU profiles [117], gprof [44], Dynaprof [92], mpiP [13], HPC toolkit (IBM) [24], Perfsuite [98], CUBE [122], HPCToolkit (Rice) [87], Open SpeedShop [132], OMPP [40], GPTL [62], IPM [99], Paraver [31] and PERI-XML [45] The profile data is parsed into a common data format. In addition, parsers for a number of parallel application self-timer output formats have also been added where necessary. The format specifies profile data by *node*, *context*, *thread*, *metric* and *event*. A node is defined as one machine (or in simple profiles, one process), a context is defined as one process on a given machine, a thread is defined as one thread of execution in a given process, and an event is defined as a measured region of source code. Profile data is organized such that for each combination of these items, an aggregate measurement is recorded. The similarities in the profile performance data gathered by different tools allowed a common organization to be used. Export of profile data to TAU profiles is also supported.

The `DataSource` class is the base class for all the supported parsers. The `DataSource` handles all of the common functionality needed for each of the parsers. Each profile format has an implementation class which inherits from the `DataSource` class. Although automatic format detection is supported in some cases, the user usually has to specify the format of the profiles to be parsed when the data is loaded. The parsing utilities are used to load the profiles into the database, or just to load the profiles into data structures for use in an analysis application.

Profile Database

The profile database component is the center of PerfDMF's persistent data storage. It builds on robust SQL relational database engines, some of which are freely distributed. The currently supported Relational Database Management Systems (DBMS) are PostgreSQL [133], MySQL [94], Oracle [104], DB2 [59] and Derby [130]. The database component must be able to handle both large-scale performance profiles, consisting of many events and threads of execution, as well as many profiles from multiple performance experiments. Our tests with large profile data (113 events on 32,768 processors) showed the framework adequately handled the mass of data. Performance of the database will be discussed on page 68.

Query and Analysis API

To facilitate performance analysis development, the PerfDMF architecture includes a data management API. This API abstracts query and analysis operations into a more programmatic, non-SQL, form. This layer is intended to complement the SQL interface, which is directly accessible by analysis tools, with dynamic data management and higher-level query functions. It is expected that analysis programs will chose to use this API for implementation, rather than generating explicit SQL statements. Access to the SQL interface is provided using the Java Database Connectivity (JDBC) API. Because all supported databases are accessed through a common interface, the tool programmer should not need to worry about vendor-specific SQL syntax.

Schema

A relational database schema is used to organize the performance data. Figure 2 shows an Entity-Relationship diagram for a representative subset of the PerfDMF database schema. The top level table, `APPLICATION`, stores the data relevant to an application, such as name, version, and other descriptive values. The `EXPERIMENT` table contains a foreign key reference to the `APPLICATION` table, and stores all data relevant to an experiment, such as the system information, compiler information, and configuration information. The `TRIAL` table contains a foreign key reference to the `EXPERIMENT` table, and contains information relevant to a trial, such as the date/time, problem definition, node count, contexts per node, and max threads per context. PerfDMF provides a flexible schema for these three tables. The schema requires that the id, name and foreign key reference columns exist in each of these tables, but additional columns may be added to (or removed from) the tables without requiring changes to the Java source code. This ability is provided by the `getMetaData()` call in JDBC, and provides flexible access to the columns in the database. The schema is designed such that if capturing such data as compiler names and versions, operating system attributes, etc. is important for analysis, then those columns can be added to the database. In addition, the analysis team is free to organize the performance

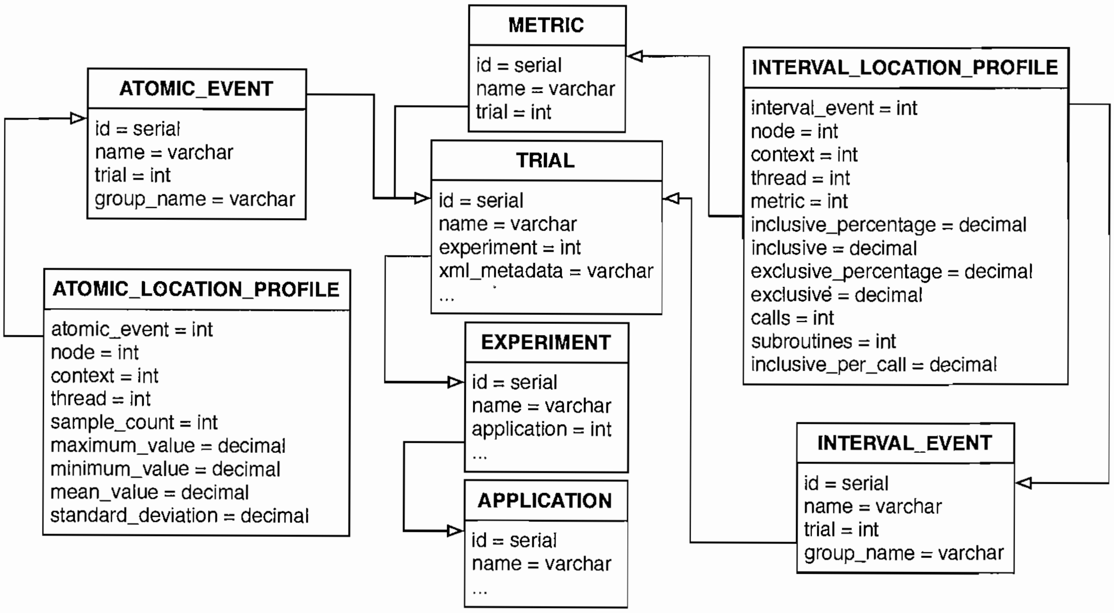


FIGURE 2: Entity-Relationship diagram of the PerfDMF schema.

attribute data in any way they like - the compiler information can be stored in the APPLICATION, EXPERIMENT, or TRIAL table or not at all. These features are important for the reusability of PerfDMF. Alternatively, the user can store metadata information in the XML_METADATA column of the TRIAL table, which is described on page 60.

Some profiling tools, including TAU, collect more than one metric when executing an experiment trial. These metrics can include measurements such as CPU time, data cache misses and floating point operations, as well as derived metrics such as floating point operations per second. Because there can be more than one metric per trial, the schema includes a METRIC table, which stores the name of the metric and a foreign key reference to the trial table. Because some analysis tools also generate derived data, derived metrics can be saved with the profile data in the database using the PerfDMF API.

Performance profile instrumentation normally organizes interval data from a profile run according to functions, or as blocks of code given a "function name". Profiling tools can also organize interval data in smaller logical blocks, such as loops, basic blocks or even individual lines of code. The top level interval data table within a trial is the INTERVAL_EVENT table. The INTERVAL_EVENT table contains the name of the event, an event group (i.e. computation, communication, etc.), and a foreign key reference to the TRIAL table, indicating the trial to which it belongs. The INTERVAL_LOCATION_PROFILE contains the cumulative data for each event, node, context, thread, metric combination. The data captured includes inclusive time, inclusive percentage, exclusive time, exclusive percentage, inclusive time per call, number of calls and number of subroutines. For some profiling tools, the value of one or more of these fields may be undefined. The INTERVAL_TOTAL_SUMMARY and INTERVAL_MEAN_SUMMARY tables (not pictured in Figure 2) contain the INTERVAL_LOCATION_PROFILE total and mean values, respectively, across all nodes, contexts and threads.

In addition to the regular instrumented profile data, data from atomic events can be captured in profiles. In TAU, for instance, users can define atomic events at code locations to collect data which varies for each instrumentation call, such as the current application size in memory, or the size of an MPI communication. The ATOMIC_EVENT table stores the atomic counter information, such as the name and group name for the counter. The ATOMIC_LOCATION_PROFILE table contains a foreign key reference to the ATOMIC_EVENT table, as well as the sample count, maximum value, minimum value, mean value and standard deviation for each ATOMIC_EVENT, node, context, thread combination.

Implementation

Our goals in developing PerfDMF are primarily integration, reusability, and portability. We also wanted an implementation based on robust and open software and protocols. We have decided to use Java, JDBC, XML and ANSI SQL, for portability, standard DBMS connectivity and profile data exchange. As described on page 50, there are three main components of PerfDMF, including profile input/output, profile database access, and profile management. All three are self-contained modules, but they share common profile data objects and API. This section discusses the PerfDMF implementation from the perspective of analysis code development. Particular attention is paid to the performance database and the management component.

As described on page 50, all data parsing and querying are handled through the DataSource class. Applications can choose to support database access, a subset of file parsing support, or some combination of the two. The nature of the analysis application will determine the support required. For example, if an analysis application will only be a database client application, and the application developer wants to selectively query the data without having to load entire (possibly large) trials, then a database-only interface should be used. If the analysis application needs to support profile data directly from profiling tools in the form of flat files, and/or does not need database support, then the second method should be used. The selection of one method does not preclude the use of the other, and the two are not mutually exclusive.

The two methods logically organize the profile data in the same way. Based on TAU's generalized performance data representation [117], PerfDMF structures its data in a *node*, *context*, and *thread* manner. Each thread then keeps track of a varying number of performance *events*, which associate singleton or aggregate data to named performance elements such as functions, loops or other blocks of code. In addition, for each node, context, thread, event, metric combination, there is an *event profile* object which stores the performance data for that particular combination. This event mapping approach allows an efficient and flexible method of performance data representation. Wrapped around this representation is PerfDMF's API for profile query and management. This API is implemented entirely in Java, and thus provides a completely portable and consistent method of accessing data.

The profile input component is responsible for obtaining performance data from a wide variety of sources, and converting it to PerfDMF's internal representation. It does so by creating a profile `DataSession` object specific to the profile format being imported. The `DataSession` object forms the core abstract object by which interactions with data sources take place. For example, the GprofDataSession provides an interface to parse gprof data. Support for other profiling tools is similar. Some profiling tools output multiple files, one for each process or thread of execution. In those cases, PerfDMF provides support for parsing a directory of files, or a subset of files in a directory that start with a particular prefix or end with a particular suffix. The profile input component manages the details of parsing the output from the supported profiling tools. There is also support for parsing and managing TAU user-defined events, as mentioned on page 47.

PerfDMF database access is provided through the use of interface functions that simplify the connection to the database. When building a client, the application developer need not concern herself with the details of database connectivity or with constructing SQL queries if she does not need or want to. It is relatively easy to get a list of APPLICATION rows from the database (returned as Java objects), and find an instance of interest. Iterating through the objects is similar to iterating through the tuples of a SQL query, but with a Java List interface. The profile database component is provided by the DBDataSource extension of the DataSession class. Once the session has been initialized, a call to getApplicationList() will return a list of Application objects, from which the desired application is selected and set as a filter for subsequent queries. The code is similar for listing and selecting Experiment, Trial, Function (interval events), FunctionProfile, UserEvent (atomic events), and UserEventProfile objects. Once an object is selected, all further query operations are filtered based on that particular context. For example, if a particular `Trial` has been selected, then any `Function` objects that are queried are only those from that particular trial. Alternatively, an application could load an entire performance profile from the database or import from a raw profile dataset into a `DataSession` object (as was mentioned earlier with the gprof example), and then apply selections with the PerfDMF API, setting node, context, and thread parameters. Saving data to the database is also easy, in that the `Application`, `Experiment` and `Trial` objects all have `Save()` methods, which will save the object and all of its related object references to the database. The `Trial` object also has support for adding new, possibly derived, metrics to an existing trial in the database.

Metadata

Performance instrumentation and measurement tools such as TAU collect context metadata along with the application performance data. This metadata contains potentially useful information about the build environment, runtime environment, configuration settings, input and output data, and hardware configuration. Metadata examples which are automatically collected by the profiling provided by TAU include fields such as processor speed, node hostname, and cache size. As we shall discuss in Chapter VII, performance metadata is essential in order to place the performance data within a *performance context*, which describes the conditions under which the data was collected. Comparisons between performance profiles without this data are not quite meaningless, but are not adequately defined without contextual boundaries.

The TAU instrumentation and measurement toolkit provides three ways to acquire metadata for analysis:

- The default behavior for the TAU measurement toolkit is to collect common hardware and software metadata from the runtime environment, such as processor speed, memory size, cache size, operating system version, etc. Table 2 shows examples of metadata fields which are automatically collected by the profiling provided by TAU. It should be easy to see how fields such as *CPU MHz*, *Cache Size* or *Memory Size* would be useful in explaining the differences between executions. In addition, on specialized hardware such as the IBM BlueGene/L or BlueGene/P systems, there are additional system calls which can provide detailed information about the hardware and the logical mapping of the processes to physical nodes.

- The TAU instrumentation API has a method, `TAU_METADATA()`, which the application analyst can insert into the code. This is the primary way for an end user to collect metadata about their application. The method takes two parameters, a name and a value. Any data of interest can be inserted into the metadata to be used later in the analysis. Input variables, runtime configuration settings, application arguments, and domain decompositions can be specified by the user.

TABLE 2: Default TAU metadata field examples.

Field	Example
CPU Cores	4
CPU MHz	2660.006
CPU Type	Intel(R) Xeon(R) CPU X5355 @ 2.66GHz
CPU Vendor	GenuineIntel
CWD	/home/joeuser/tau2/examples/NPB2.3/bin
Cache Size	4096 KB
Executable	/home/joeuser/tau2/examples/NPB2.3/bin/lu.C.16
Hostname	garuda.cs.uoregon.edu
Local Time	2007-03-29T16:06:08-07:00
Memory Size	8155912 kB
Node Name	garuda.cs.uoregon.edu
OS Machine	x86_64
OS Name	Linux
OS Release	2.6.18.1_ktau_1.7.9_pctr
OS Version	#2 SMP Mon Mar 26 17:36:14 PDT 2007
TAU Architecture	x86_64
TAU Config	-fortran=intel -cc=icc -c++=icpc -mpi ...
UTC Time	2007-03-29T23:06:08Z
username	joeuser

- The PerfDMF data importer can take an optional XML file with metadata fields which contain name/value pairs to be included in the performance metadata. The schema is very simple, and does not require special XML processing libraries to generate. Information relating to the build environment, compiler options, input files, batch system, allocated hardware, or anything else that might assist the performance analysis can be included in this XML file.

These additional metadata fields are stored in the `TRIAL` table, in the `XML_METADATA` and `XML_METADATA_GZ` columns, the former for metadata which is less than 64 kilobytes in length, the later for XML metadata which is more than that amount. The second column was added to support databases which do not support

```
<?xml version="1.0" encoding="UTF-8"?>
<tau:metadata xmlns:tau="http://www.cs.uoregon.edu/research/tau">
 <tau:CommonProfileAttributes>
  <tau:attribute>
   <tau:name>CPU Cores</tau:name>
   <tau:value>4</tau:value>
  </tau:attribute>
  <tau:attribute>
   <tau:name>CPU MHz</tau:name>
   <tau:value>2660.006</tau:value>
  </tau:attribute>
  <tau:attribute>
   <tau:name>CPU Type</tau:name>
   <tau:value>Intel(R) Xeon(R) CPU X5355 @ 2.66GHz</tau:value>
  </tau:attribute>
  <tau:attribute>
   <tau:name>CPU Vendor</tau:name>
   <tau:value>GenuineIntel</tau:value>
  </tau:attribute>
 </tau:CommonProfileAttributes>
</tau:metadata>
```

FIGURE 3: Sample XML metadata.

very long strings, but do support large binary data. The metadata is stored as an XML string, using a simple schema. An example of the XML data is shown in Figure 3 There can be unique metadata for each thread of execution, but the fields which are common across all threads and have the same value are aggregated into the `CommonProfileAttributes` node, to prevent data redundancy.

CCA/CQoS extensions

The Common Component Architecture (CCA) project [71] adopts a component-based approach for building large scale scientific applications. One of the sub-projects of CCA is the Computational Quality of Service (CQoS) project [101, 86], which has the stated goal of improving the runtime quality of service for CCA applications. In order to provide quality of service, the CQoS project needs to gather performance information for various components which perform similar functions, so that when an application has the option of using one of a number of components for a given task, it can use the component implementation which will provide the fastest execution, most accurate solution, or some other metric of quality. In order to store performance data for iterative non-linear solvers, the PerfDMF schema had to be extended to support metadata which is collected once per iteration, rather than just once per execution of the application (the current level of metadata support in PerfDMF). The metadata for each iteration was associated with a TAU phase event, which was captured once for each iteration of the outer main loop. This extension was supported by adding a table which contained the metadata fields, and a foreign key reference to the phase event in the profile data. A more thorough description of the motivation for this support is provided on page 213.

PERI-DB

The TAU project is participating in a larger effort known as the Performance Engineering Research Institute (PERI) [143]. The PERI project has the stated goal of focusing on delivering petascale performance to complex scientific applications running on leadership class computing systems. One of the sub-projects of PERI is the PERI-DB group, which is tasked with providing interoperability between performance tools. That interoperability is provided through the use of a common data interchange format, known as PERI-XML. The PERI exchange format definition has gone through many revisions, and is still in development, but PerfDMF is a key collaborator in the project, and is providing the ability to parse and format PerfDMF databases to and from the PERI-XML format. In addition, using draft versions of the PERI-XML schema, we were able to exchange data between the Open|SpeedShop [132] performance tool and a PerfDMF database. In its current form, the exchange format is mainly supporting metadata, but it will be extended to support profile data in the future.

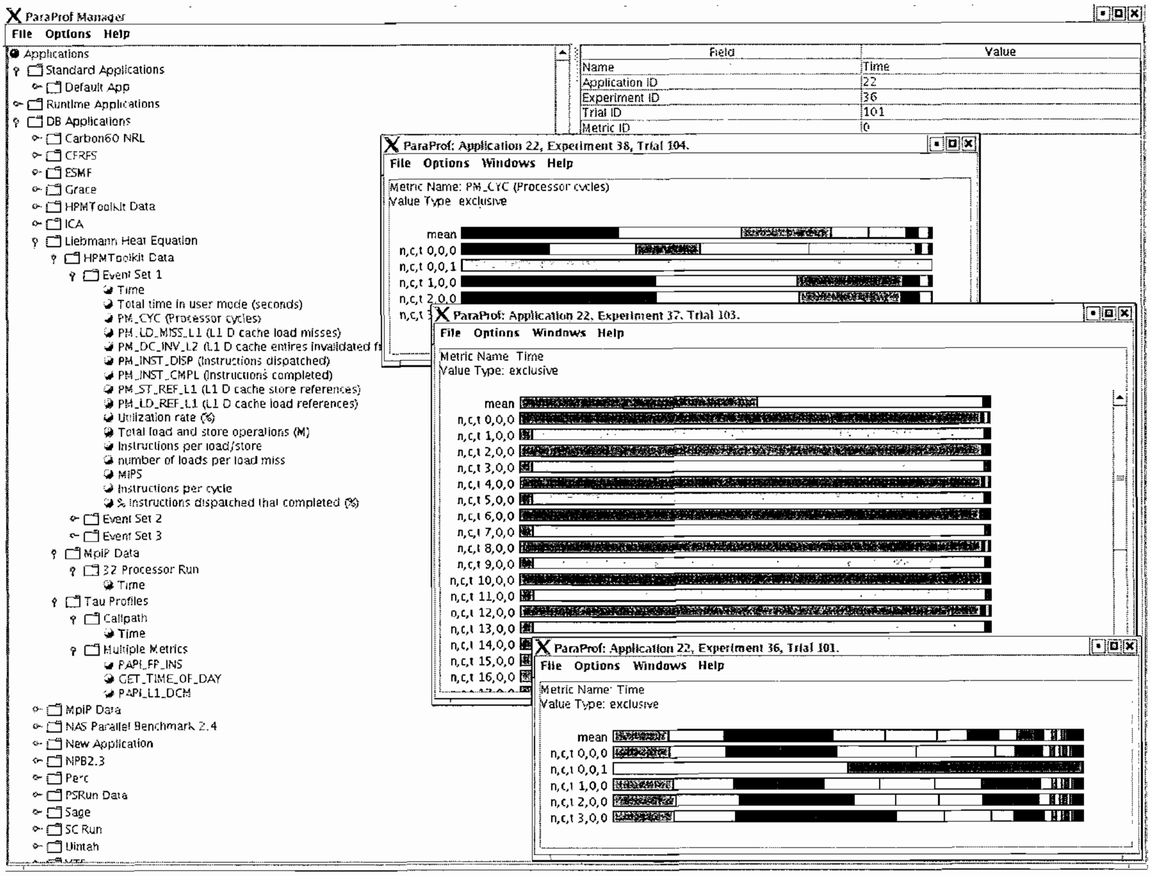
Applications

This section presents some applications of PerfDMF to existing performance tools. We shall consider one application of PerfDMF: parallel profile analysis and viewing in the ParaProf tool. The ParaProf profile analyzer is particularly enhanced by the ability to parse additional profile formats, and the ability to store data to a database. We will also present a performance study of the database management framework.

ParaProf

ParaProf [7] is TAU's main profile browser, and is a portable, extensible and scalable tool for parallel performance analysis. ParaProf provides a mature, reliable platform on which to graphically browse parallel performance profile data. It implements graphical displays of all performance analysis results in aggregate and single node/context/thread forms. ParaProf also provides the ability to compare the behavior of one instrumented event across all threads of execution, and offers summary text views of performance data, with various groupings and contextual highlighting. The initial release of ParaProf could only read TAU data from flat files, and though it could generate rudimentary derived data, it had limited methods by which that data could be saved for further analysis. With the addition of PerfDMF, ParaProf is now able to parse profile data from additional profile tools, and has database support for accessing archived profile data and saving derived metric data. ParaProf can also be used as the primary interface to the performance profile database, providing a graphical user interface which analysts can use to store and view performance profiles in a shared data repository. ParaProf supports connecting to multiple PerfDMF databases, and also has a user interface for creating and editing PerfDMF configurations.

Figure 4 shows an example of ParaProf using the PerfDMF API to interface with the database. On the left side of the application window is a tree view of the applications, experiments and trials which have been loaded into the database.



Field	Value
Name	Time
Application ID	22
Experiment ID	36
Trial ID	101
Metric ID	0

Metric Name: PM_CYC (Processor cycles)
Value Type: exclusive

Metric Name: Time
Value Type: exclusive

Metric Name: Time
Value Type: exclusive

FIGURE 4: ParaProf with PerfDMF support accessing HPMToolkit, mpiP, and TAU data from a database archive. The top graph window shows the HPMToolkit data, the middle window is mpiP data, and the bottom window is TAU data. ParaProf can also be used to input data into the database.

Three trials shown, all from the same application, have been loaded into the database using the PerfDMF API, and are expanded in the tree. The three trials come from three different profiling tools, specifically HPMToolkit, mpiP and TAU. Additional application profile data is loaded into the database, mostly from TAU data files. This figure is not intended to show comparative analysis between trials, but rather the use of PerfDMF to parse various profile formats and store them in a database archive.

This archive could be made available in one physical location for all analysts within an organization.

PerfDMF performance

The datasets used to test the performance of PerfDMF are from the production application Miranda [12] from the Lawrence Livermore National Laboratory (LLNL). The Miranda profile data was collected by scientists at LLNL, in the form of TAU profile data from test runs on the (at the time) in-development IBM BlueGene/L [82]. BlueGene/L currently has 106,496 dual-processor compute nodes. The test data we were provided was from runs of 4K, 8K, 16K and 32K processors (where 1K = 1024). Over one hundred events were instrumented for each trial, and only one metric was collected, wall clock time. The 32K processor run consists of over 3.7 million data points (tuples).

Table 3 shows the average time to load the data from a database for various profile sizes, using a Linux workstation with two dual-core 3GHz CPUs and 4GB of memory, using PostgreSQL version 7.4.19 and 64-Bit Java version 1.6.0-015. Each dataset was loaded 20 times to compute the average time to load the data, and the ± values represent 95% confidence intervals of the mean. The *Tuples* column shows how many data points were loaded ($number\ of\ processors * number\ of\ events * number\ of\ metrics$). The *Query Time* column shows how long it took to execute the query to select the data from the database, and transfer that data from the database

TABLE 3: Time (in seconds) to load Miranda data from the PerfDMF database.

Proc.	**Events**	**Tuples**	**Query Time**	**Process Time**	**Total**
4K	105	430080	3.06 ± 0.1	2.56 ± 0.57	6.04 ± 0.60
8K	105	860160	7.16 ± 0.30	5.56 ± 0.39	13.34 ± 0.33
16K	101	1654784	15.21 ± 0.51	9.01 ± 0.96	25.13 ± 0.88
32K	113	3702784	44.12 ± 0.59	18.18 ± 0.38	68.14 ± 1.62

TABLE 4: Time (in seconds) to load TAU profiles from the file system.

Proc.	**Events**	**Tuples**	**Query Time**	**Process Time**	**Total**
4K	105	430080	n/a	1.50 ± 0.13	1.85 ± 0.19
8K	105	860160	n/a	3.10 ± 0.20	3.85 ± 0.23
16K	101	1654784	n/a	6.32 ± 0.32	7.71 ± 0.36
32K	113	3702784	n/a	14.69 ± 0.73	17.90 ± 0.82

to the application, allocating JDBC objects along the way. The *Process Time* shows how long it took to iterate through the query results, and store the data in PerfDMF data structures. The data was loaded essentially as a pair of queries, first to select the events for the trial, then to select the performance data for all events across all threads of execution.

Table 4 shows how long it would take to load the same data from the file system, in the form of raw TAU profiles, using PerfDMF. Clearly, there is some overhead from using the database, as reflected in the *Query Time* column, and to a lesser extent, the *Process Time* column, which is reading lines from a file on disk rather than iterating through query results. Regardless, the scaling behavior is fairly linear, with the exception of the 32,768 processor data, which is an exceptionally large profile. However, the time to load the data is not unreasonable, considering the volume of data.

TABLE 5: Time (in seconds) to load selected data from the PerfDMF database.

Proc.	Events	One Event, All Threads	All Events, One Thread	One Event, One Thread
4K	105	0.0610 ± 0.0004	0.2609 ± 0.0001	0.0022 ± 0.0002
8K	105	0.1220 ± 0.0015	0.5153 ± 0.0004	0.0040 ± 0.0001
16K	101	0.2563 ± 0.0086	0.9916 ± 0.0008	0.0082 ± 0.0002
32K	113	0.5374 ± 0.0234	2.2056 ± 0.0014	0.0162 ± 0.0002

One final table shows another benefit of storing the data in a database. Table 5 shows the amount of time, in seconds, it takes to selectively query data from the database. Here we see that once the data is loaded into the database, optimized queries into the database provide very fast access to performance data. When querying for one event, all threads, the number of tuples returned equals the number of threads. When querying for all events, one thread, the number of tuples returned equals the number of events. Finally, querying for one event on one thread returns only one tuple.

While the user has the ability to write custom queries, it may not be recommended in some cases. For example, if the user wishes to return the top ten events for all threads, using the exclusive time spent in the event to rank the events, the user can create that query. However, not all users are that sophisticated at writing SQL queries. Any data management toolkit has to walk a fine line between providing every possible query combination, and limited access to data. The targeted access described above should be sufficient for scalable data access even with very large profiles.

Summary

In this chapter, we discussed the need for management of parallel performance profile data, and our data management framework. Our solution consists of a scalable, flexible database schema, access API and associated tools. With this framework, we have enhanced the ability of TAU performance analysis tools such as ParaProf to access data repositories, but also to parse and analyze many different profile formats through our data session interfaces. In the next chapter, we will discuss our efforts to use this framework as the foundation for a new performance analysis framework which can be used to perform scalable differential analysis of parallel profile data.

CHAPTER IV

PERFEXPLORER ANALYSIS FRAMEWORK

Overview

As high-end parallel computer systems scale in number of processors, their operation, programming, and performance evaluation grow more complex. Complexity, as it arises from the evolving nature of scalable machines and applications, is also a source of concern of parallel performance tools. The general goal of any performance tool is to provide the user with an understanding of performance phenomena, whether that be by interactive data analysis or by more automatic methods for performance investigation. However, when faced with systems and applications of greater sophistication, size, and integration, the requirements to address new performance complexity goals challenge tool design, engineering, and technology.

How do we build performance tools that can deliver high utility and productivity for the parallel computing community without being overwhelmed by high-end complexity demands? Large-scale parallel computing presents a complex face to performance tools. Tools that ignore the complexity are limited in power, either by their simplicity or by their scope. Similarly, tools that feature creative solutions but which are complicated and unusable in practice will go largely unnoticed. Parallel performance technology must acknowledge the complexity challenges of high-end systems and strive to deliver tool solutions of high value and productivity. Ultimately, the potential of performance tools will be realized by both addressing hard performance analysis problems and by developing and delivering tools with strong computer science contributions and high engineering standards.

In this chapter, we describe our research and development work in exploring methods for parallel performance comparative analysis and data mining. Our prototype framework, PerfExplorer, represents our efforts to explore the issues of managing complexity in this environment. Our research is motivated by our interest in automatic parallel performance analysis and by our concern for extensible and reusable performance tool technology. PerfExplorer is built on PerfDMF (see Chapter III), which provides a common, reusable foundation for performance results storage, access, and sharing. Our work targets large-scale performance analysis for single experiments on thousands of processors and for multiple experiments from parametric studies. PerfExplorer addresses the need to manage large-scale data complexity using techniques such as clustering and dimensionality reduction, and the need to perform automated discovery of relevant data relationships using comparative and correlation analysis techniques. Such data mining operations are engaged in the PerfExplorer framework via an open, flexible interface to existing statistical analysis and data mining applications, including the R Project [134] and Weka [154]. To our knowledge, PerfExplorer is the first large scale, integrated framework for mining parallel performance profile data. With the aforementioned programmable applications, PerfExplorer functionality can be extended by us and others in the future.

This chapter will focus on the design of the comparative framework and the graphical user interface. Primarily, this chapter will present the original design and implementation of PerfExplorer. Our experience with this prototype led us to examine the need for new strategies to handle even more complexity, such as large profiles, repetition, extensibility and knowledge engineering. In the next three chapters, we will discuss the data mining techniques used (see Chapter V), the need for automation support (Chapter VI), and the integration of metadata and expert knowledge (Chapter VII). Where appropriate, we will briefly mention those aspects of our work within the context of this framework design discussion.

Goals And Design

The overall goal of the PerfExplorer project is to create a software infrastructure to help conduct parallel performance analysis in a systematic, collaborative, and reusable manner. The infrastructure should exist to provide easy performance data access and to link analysis capabilities. It is also important to provide support to manage the analysis process. In particular, our objective is to integrate sophisticated data mining techniques in the analysis of large-scale parallel performance data. Given existing robust data mining tools, PerfExplorer's design motivation is to interface cleanly with these tools and make their functionality easily accessible to the user. The power and extensibility of the integrated analysis libraries, coupled with the data management of PerfDMF gives the PerfExplorer environment a strong set of capabilities for performance knowledge discovery.

Complexity Management

One important goal of PerfExplorer is to reduce the degree of complexity in large performance profiles and in their analysis. This is accomplished by more robust support for performance data and results management as well as management of analysis processes and automation. To discover characteristics of an application or parallel machine which may be hidden in the data, we need flexibility in a performance data mining tool to select features of interest to investigate and mining operations to perform. PerfExplorer manages data complexity through the use of PerfDMF and by making it easy for a user to select datasets and parameters in different combinations for analysis. PerfExplorer manages analysis complexity through the abstraction of data mining procedures, thereby reducing the expertise required of the user to develop these procedures or to efficiently access them via available statistical software.

The intended uses of PerfExplorer include, but are not limited to benchmarking, procurement evaluation, modeling, prediction and application optimization. In all these uses, the ability to quickly compare the results of several experiments, and summarize characteristics of large processor runs will replace the need for users to develop their own tools or manually integrate several tools in an analysis process. Our aim is to provide an application that does not require a performance expert to operate, and yet still provide meaningful performance analysis.

Comparative Analysis

Another goal of our framework is to provide convenient methods for performing comparative analysis. With the performance data available in a data management system, it makes sense to have the ability to easily generate multi-dimensional comparisons between trials. The framework should provide the ability to visually explore the data and begin the search for relationships between performance behavior and its context. The comparisons should support the ability to aggregate or separate the data among any of the dimensions, such as the processor count, metrics, events, or any other contextual field available.

In addition to the data mining operations to be described in Chapter V, the user can request comparative analysis. Table 6 lists the types of predefined scalability charts available. These charts include total execution time, time-steps per second, and various breakdowns of relative efficiency and speedup. In addition, when the

TABLE 6: Pre-defined scalability charts available in PerfExplorer.

Total Execution Time	Time-steps per Second
Relative Efficiency	Relative Speedup
Relative Efficiency by Event	Relative Speedup by Event
Relative Efficiency for One Event	Relative Speedup for One Event
Relative Efficiency by Phase	Relative Speedup by Phase
Group % of Total Runtime	Runtime Breakdown
Correlate Events with Total Runtime	Phase Fraction of Total Runtime

events are grouped together, such as in the case of communication routines, yet another chart shows the percentage of total runtime spent in that group of events. Finally, there is a chart which will correlate the scalability of the application overall with the scalability of each of the events in the application. These charts can be generated across different combinations of parallel profiles. Examples of these charts are shown later in this chapter.

While these charts are useful in scalability studies, we realized we would need a more flexible comparison interface in order to perform general purpose parametric studies. For that reason, we added an additional interface to the application to allow for custom charts, as shown in Figure 5. The custom chart interface allows for user selection of x-axis categories and series groupings of scatter plot and line charts. The selection can be made from columns of the database tables, or the metadata fields from the trials. The user also has the option of filtering out call path events, using a logarithmic y-axis, selecting individual events or groups of events, and filtering insignificant events. All charts can be output as vector or bitmap images to disk, for later use in publications or presentations.

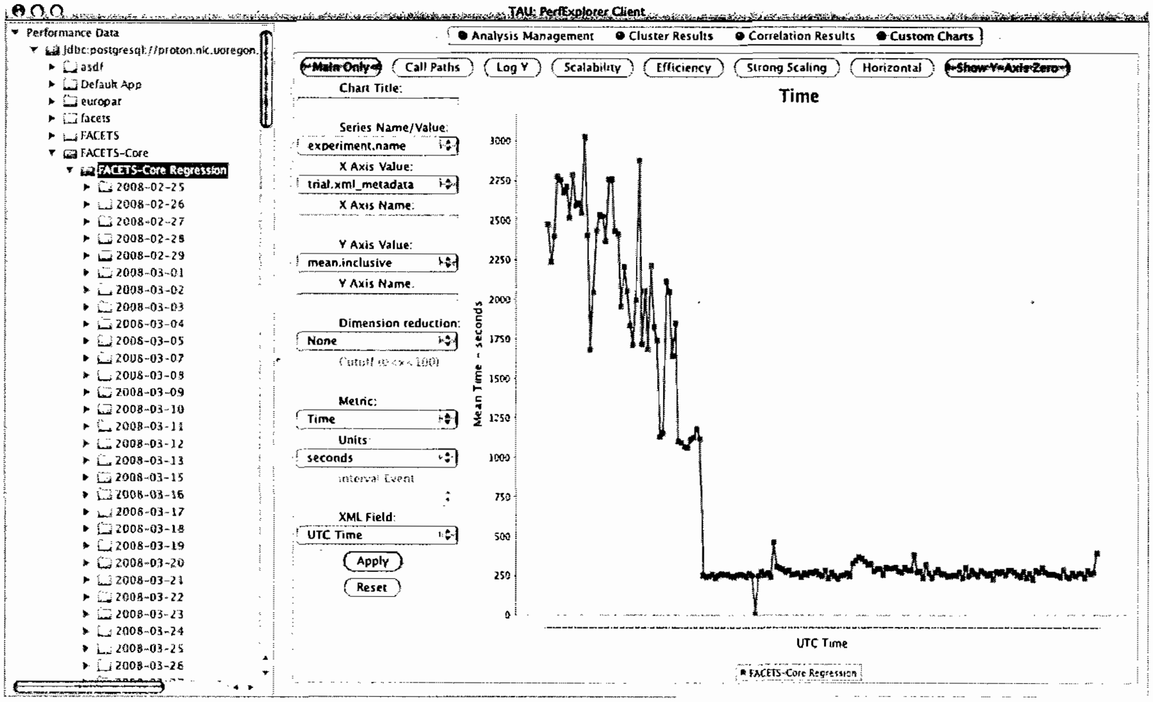


Chart Title:

Series Name/Value: experiment.name

X Axis Value: trial.xml_metadata

X Axis Name:

Y Axis Value: mean.inclusive

Y Axis Name:

Dimension reduction: None

Cutoff (0 < x < 100)

Metric: Time

Units: seconds

Interval Event

XML Field: UTC Time

FIGURE 5: An example of a custom chart in PerfExplorer. In this figure, daily performance data is compared to show the improvement in total runtime as the application develops over time.

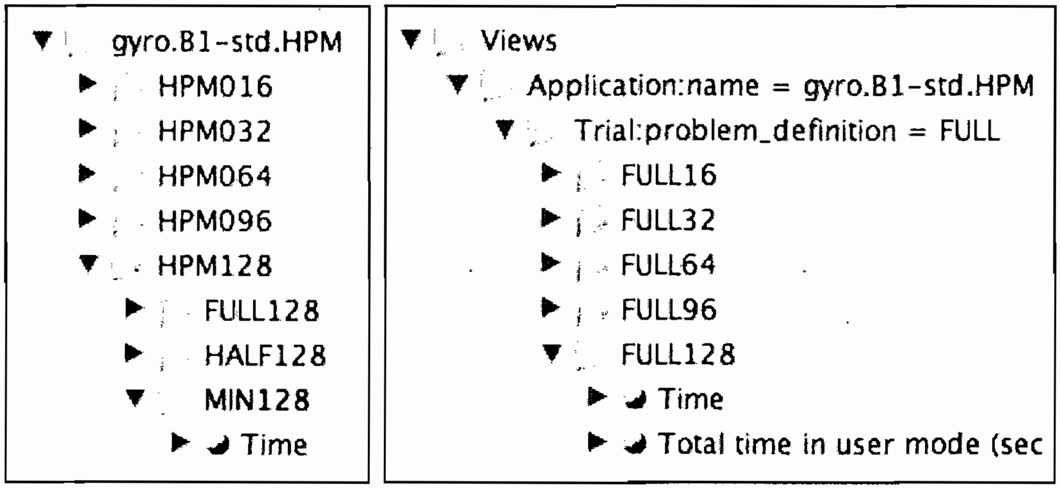
Views

Despite the flexibility provided by the application / experiment / trial hierarchy in PerfDMF, providing only one way to access data limits the ability to examine performance data from different perspectives, and within different contexts. In order to provide flexible comparative analysis support, some extensible mechanism is required to provide various cross-sections through the data in the database.

Relational databases have the concept of a *view*, which is essentially a virtual table which contains the result of a predefined query on the database. This result is often a subset of a table or a join of a number of related tables. Views can be used to filter table results, and to reduce complexity for the user. Rather than support view functionality with actual database views in the database schema (which may not be supported in all DBMS), we developed a simplified view mechanism to filter trials in our database. This view mechanism is used to create slices through the PerfDMF `TRIAL` table, and aggregate the data in custom ways.

For example, suppose the data for a particular experiment is organized to facilitate a scalability study. If the same data is needed for a parametric study related to the implementation of some user-tunable calculation, then it should not be necessary to re-load the data in order to re-organize it. Therefore, we have designed the comparative analysis to also support user defined *views* and *subviews*. The views are designed such that the user can select a subset of data from the database, and then further subdivide that data into different organizations based on arbitrary data columns. Figure 6 shows an example where the data has been loaded into the database organized by processor count, and then by input problem. By creating views and sub-views, the same data can be reused in a scalability analysis.

The views are supported by extending the PerfDMF database with a new table containing a description of the predefined query. The table contains a table, column, operator and value, as well as a parent view (for defining sub-views). As an example, the first view on the right of Figure 6 will result in a query where all trial records which belong to an application with the name equal to "gyro.B1-std.HPM" are selected and returned in the tree. When the sub-view is also added, the result is a compound



```
▼ gyro.B1-std.HPM
    ▶ HPM016
    ▶ HPM032
    ▶ HPM064
    ▶ HPM096
    ▼ HPM128
        ▶ FULL128
        ▶ HALF128
        ▼ MIN128
            ▶ Time
```

```
▼ Views
    ▼ Application:name = gyro.B1-std.HPM
        ▼ Trial:problem_definition = FULL
            ▶ FULL16
            ▶ FULL32
            ▶ FULL64
            ▶ FULL96
            ▼ FULL128
                ▶ Time
                ▶ Total time in user mode (sec
```

FIGURE 6: Views in PerfExplorer. In this example, the data for the B1 benchmark of the GYRO application has been loaded into the database organized by processor count, and then by input problem, as shown on the left. By creating views and sub-views, as shown on the right, the same data can be reused in a scalability analysis.

`WHERE` clause in the database query, in which all trials which meet the first view's criteria as well as the subview's criteria are selected. Then, when requesting analysis or charts for a view, the trials from the result of the query are used as input to the operation.

PerfExplorer Architecture Design

From the start, PerfExplorer was targeted to large-scale performance data analysis. The concept was one of an interactive environment from where analysis processes would be launched and results would be visually reviewed. Initial tests analyzing large data sets made it obvious that for an interactive application to be responsive to user events, the framework would need to be either multi-threaded or distributed. We decided to support both a client-server architecture and a multi-threaded, standalone option. In the client-server configuration, several analysis clients can share a single analysis server. Remote clients would request data mining operations and retrieve results. In a distributed environment, the architecture affords the potential to locate clients and servers where desired, and to leverage Internet and other technologies when implementing the components. For example, the PerfExplorer client and standalone configurations have been configured to be launched from a web browser, with no prior installation or configuration of the environment or client workstation required.

The architecture can be effectively run on a single machine, when workstation performance is not an issue, or where network security may be an obstacle to distributed applications. When PerfExplorer is executed as a standalone application, the behavior of the client application is exactly the same. The only difference is that rather than requesting remote objects from a server application, the analysis would be performed in a separate thread, allowing for responsive interactive queries while long-running analysis is performed by the server thread.

Figure 7 shows the PerfExplorer architecture. It consists of two main components, the *PerfExplorer Client* and the *PerfExplorer Server*. The PerfExplorer Client is a standard Java client application, with a graphical user interface developed in Swing [126]. The client application connects to the remote PerfExplorer server (also

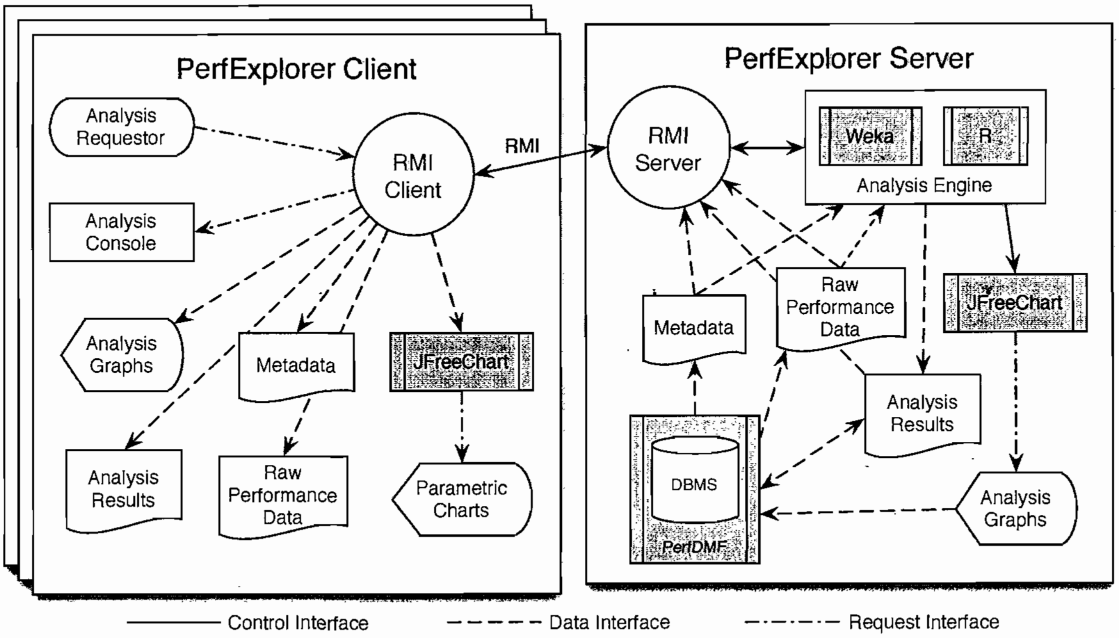


FIGURE 7: The PerfExplorer architecture.

written in Java) using Remote Method Invocation (RMI), and makes processing requests of the server. The process of performing the data mining analysis is straightforward. Using the PerfDMF API, the server application makes calls to the performance DBMS to get raw performance data. The server then passes the raw data to an analysis engine which performs the requested analysis. Once the analysis is complete, the PerfExplorer server saves the result data to the PerfDMF DBMS. Output graphics can also be requested at the server and images saved for later review. Because the analysis server is multi-threaded, it can continue to serve interactive requests to the client (or multiple clients) while performing analysis. One such type of interactive request is to perform comparative analysis. In that case, the user selects two or more data objects, and requests the data to be compared from the server. The server performs the database query, and returns the results to the client, which are rendered for the user. The types of interactive displays available include scalability charts, four dimensional correlation scatter plots, and data summarizations.

Approach

In this section, we will look in more detail at the technology involved in PerfExplorer's implementation, and take a look at the PerfExplorer user interface.

Components

The types of analysis described and their respective visualizations would take many man-years of development to implement. It makes much more sense to leverage the available tools in the open-source community, rather than struggle with the difficulty of implementing our own analysis routines. Several software components are needed for this research project, and we will discuss the major contributors.

PerfDMF

As mentioned before, PerfExplorer is built on PerfDMF. Queries to the database are constructed in standard SQL, to ensure compatibility with a large subset of DBMS implementations, assuming they provide a Java database connectivity (JDBC) interface. PerfDMF provides the foundation on top of which we have built the performance analysis toolkit, with extends the API to perform queries against the database to characterize parallel performance behavior. We have also extended the PerfDMF schema to support views and to queue analysis requests and store analysis results.

Java

The primary development language for the tools in the TAU project and PerfExplorer is Java 1.4, to ensure the maximum portability among systems today. The developers of the TAU project are focused primarily on integration, re-usability, and portability, based on robust and open software. Java was selected for its near ubiquitousness, its facility for extension, and for the large selection of class libraries which use Java as a code base. By using Java, we can leverage the software base already developed and in use in the PerfDMF project, as well as other supporting libraries and frameworks.

Data Mining and Statistical Packages

There are several techniques for statistical analysis and data mining on parallel performance data. Many of these techniques have already been implemented in other tools. It makes little sense to re-implement these capabilities in our own Java library. One of the goals of the PerfExplorer development is to leverage mature, open source software solutions where possible. In addition, if a performance analyst using PerfExplorer already has a library of analysis operations which she has developed, we would like to integrate that functionality within the PerfExplorer framework. Currently, we have designed PerfExplorer as a wrapper around two analysis applications: R and Weka. Other applications, such as Octave[63] or Matlab could also be wrapped with this interface. We will give a full discussion of the data mining methods we applied in Chapter V.

R is a language and environment for statistical computing and graphics. It is essentially an open source implementation of the S language and environment [6], which was developed at Bell Laboratories. R provides a wide variety of statistical and graphical techniques, and is highly extensible. R has been ported to a number of platforms. R is written in C and does not have a built-in Java interface. We used the Omegahat[131] interface to integrate the R analysis into the PerfExplorer Java application.

The second application we integrated is Weka, a collection of machine learning algorithms for data mining tasks. Unlike R, Weka is written entirely in Java. It contains tools for data preprocessing, classification, regression, clustering, association rules, and visualization. Several projects have used Weka and contributed new tools. PerfExplorer's wrapping approach allows R and Weka to be used separately or in combination.

JFreeChart

Each of the analysis applications mentioned have visualization functionality included, but they are dissimilar and non-uniform. Visualization is an important component of PerfExplorer and we desire consistent visualization results, regardless of the choice of application for analysis. For this reason, we chose to use a charting graphics library written in Java to integrate into the environment. JFreeChart [21] includes support for a large number of charts, including pie, bar, line, area, scatter, bubble, time series and combination charts. By loading the performance data and/or results in a common data format, we are able to visualize this data using one visualization call. JFreeChart can be used to generate visualizations on the server side which are stored in the database, awaiting request from the client. It can also be used to generate charts in the client, for interactive data display.

Java Packages

The Java classes in PerfExplorer are organized into nine Java packages, as shown in Figure 8. A Java *package* is essentially a namespace, and provides a mechanism for organizing classes. The simplest place to start is with the `constants` package, which defines common constants used in the application. The `common` package defines RMI interfaces, and other classes to be passed between the client and server. The `server` package manages the analysis queue, and processes the analysis requests. The `cluster` package defines the interfaces used to process analysis request. A factory

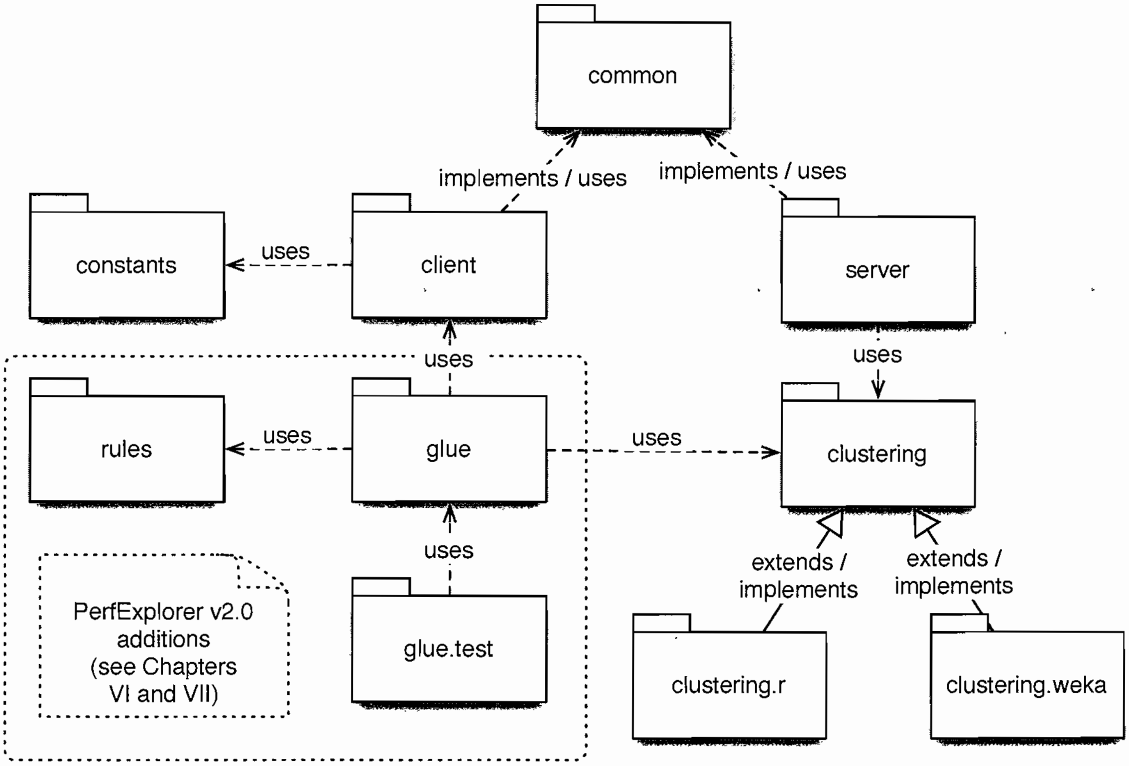


FIGURE 8: The PerfExplorer packages.

class in the cluster package is used to instantiate the objects to perform analysis. Those objects are concretely defined in the `cluster.r` and `cluster.weka` packages. At runtime, the user specifies which analysis application to use (Weka is the default). The two remaining packages, `glue` and `rules`, are used to define the component scripting interface described in Chapter VI, and the inference engine integration described in Chapter VII, respectively. The `glue.test` package is a collection of test objects to perform unit testing on the script interface objects.

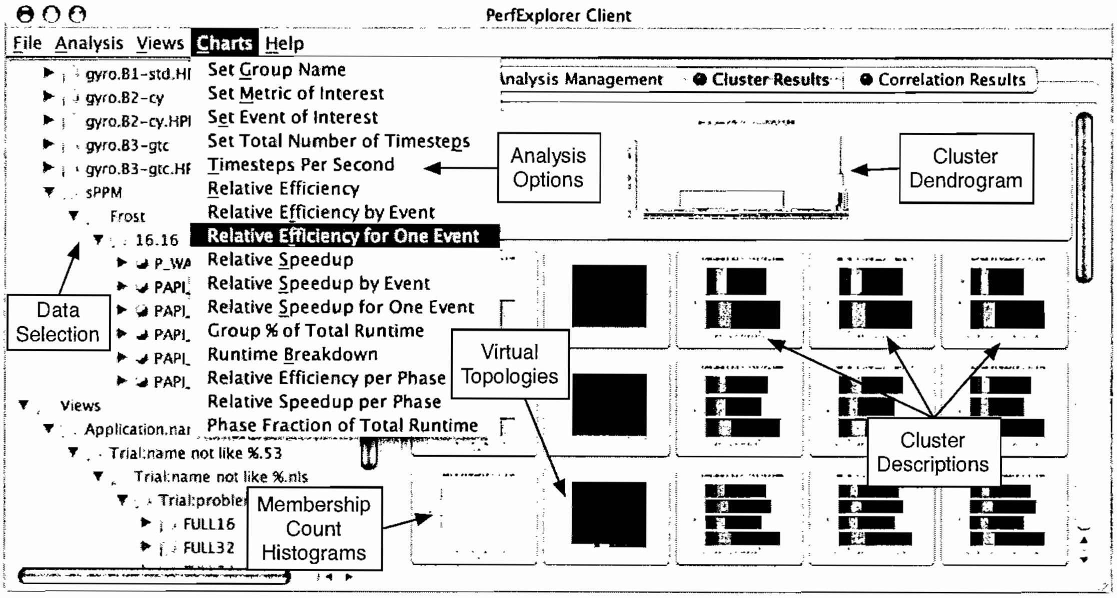


FIGURE 9: The PerfExplorer user interface. Performance data is organized in a tree view on the left side of the window, and the cluster analysis results are visible on the right side of the window. Various types of comparative analysis are available from the drop down menu selected.

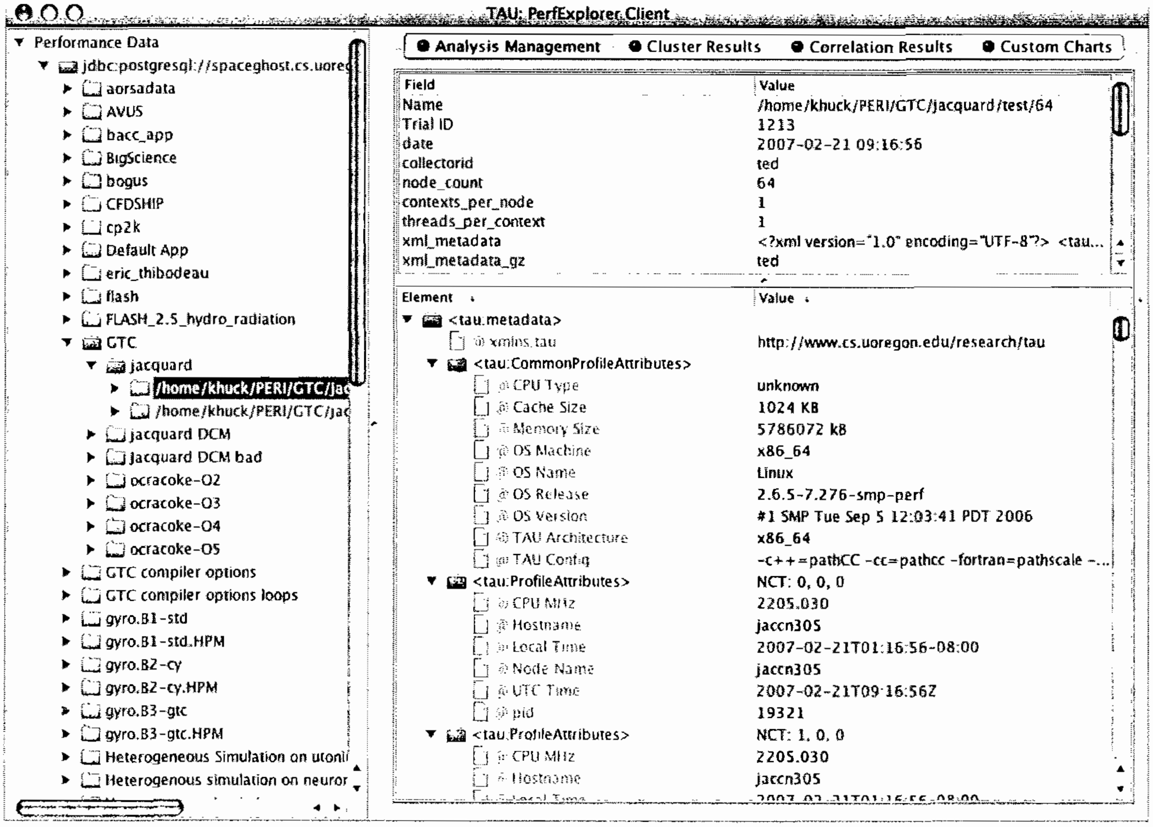
User Interface

As important as the data mining functionality provided, the user interface to PerfExplorer will determine how productively it is used. Figure 9 shows the user interface for PerfExplorer. The PerfExplorer client serves as a management console for requesting, checking the status of, and reviewing the results of analysis operations. The main client window is divided in two. The left side contains a navigation tree, representing the performance data as it is stored in the database. PerfDMF data is organized in an *Application / Experiment / Trial* hierarchy, and that hierarchy is represented in the top of the tree. In addition, the user has the ability to create views of the data, with arbitrary organization. The views are visible in the lower section of the tree view.

The tree navigation is primarily used for selecting the focus of the data analysis, requesting analysis operations, and querying the status of the analysis operations. In a sample user case, as shown in Figure 9, the user will browse to the trial(s) of interest. The user then selects the relevant datasets that will form the basis of analysis. After setting optional analysis parameters, the user will then request the analysis operation. While the operation is being performed, the user can use the console window (not shown) to monitor the status of the analysis. As soon as preliminary results of the analysis are available, or when the analysis is complete, the user can access the other two tabbed consoles to examine the results.

As shown in Figure 9, the *Cluster Results* console presents the user with a "thumbnail" view of the performance graphs generated by the analysis. If the user finds a graph that is particularly interesting, she can select the thumbnail. A larger view of the graph will be presented to the user.

The *Correlation Results* console (not shown) presents the user with thumbnail views of correlation analysis. Currently, the user can request correlation analysis of events or metrics, the result of which will help guide the selection of performance metrics and/or events of interest. Scatter plots are used to represent the results of the correlation analysis, along with the coefficient of correlation between the events and/or metrics, and regression curves through the data.



Field	Value
Name	/home/khuck/PERI/GTC/jacquard/test/64
Trial ID	1213
date	2007-02-21 09:16:56
collectorid	ted
node_count	64
contexts_per_node	1
threads_per_context	1
xml_metadata	<?xml version="1.0" encoding="UTF-8"?> <tau...
xml_metadata_gz	ted

Element	Value
<tau:metadata>	
xmlns.tau	http://www.cs.uoregon.edu/research/tau
<tau:CommonProfileAttributes>	
CPU Type	unknown
Cache Size	1024 KB
Memory Size	5786072 kB
OS Machine	x86_64
OS Name	Linux
OS Release	2.6.5-7.276-smp-perf
OS Version	#1 SMP Tue Sep 5 12:03:41 PDT 2006
TAU Architecture	x86_64
TAU Config	-c++=pathCC -cc=pathcc -fortran=pathscale -...
<tau:ProfileAttributes>	NCT: 0, 0, 0
CPU MHz	2205.030
Hostname	jaccn305
Local Time	2007-02-21T01:16:56-08:00
Node Name	jaccn305
UTC Time	2007-02-21T09:16:56Z
pid	19321
<tau:ProfileAttributes>	NCT: 1, 0, 0
CPU MHz	2205.030
Hostname	jaccn305
Local Time	2007-02-21T01:16:56-08:00

FIGURE 10: The PerfExplorer user interface, with the XML metadata tree display. Metadata for the trial is displayed as a tree of hierarchical data.

When performing comparative analysis, the user can browse the data in the database, and either select data from the existing application / experiment / trial structure, or build custom views of the data. The user can then select a number of experiments, where each experiment represents a machine and parameter combination. Then the user selects a comparative analysis to perform from the drop-down menu at the of the application, as demonstrated in Figure 9.

Figure 10 shows how XML metadata is displayed in PerfExplorer. The data is rendered as a tree table, and is organized in $N + 1$ subtrees. The first tree contains the common metadata across all processes and threads of execution. The other N subtrees contain the metadata which is unique to each of the respective processes and threads. With this interface, the user can browse the metadata in the trial, and visually compare the metadata fields which are different between threads.

Application

To demonstrate how PerfExplorer is used in practice, this section reviews our work with one large-scale parallel application, GYRO. As compared to generating textual performance summaries, loading them into a spreadsheet or plotting program, and then generating figures, the relative ease of using the PerfExplorer GUI should be mentioned. Much of the analysis process is handled behind a graphical user interface, and the user interfaces with the framework at a high level. It is straightforward to capture the analysis results graphically for reports, or have them recorded back into the performance database for later use.

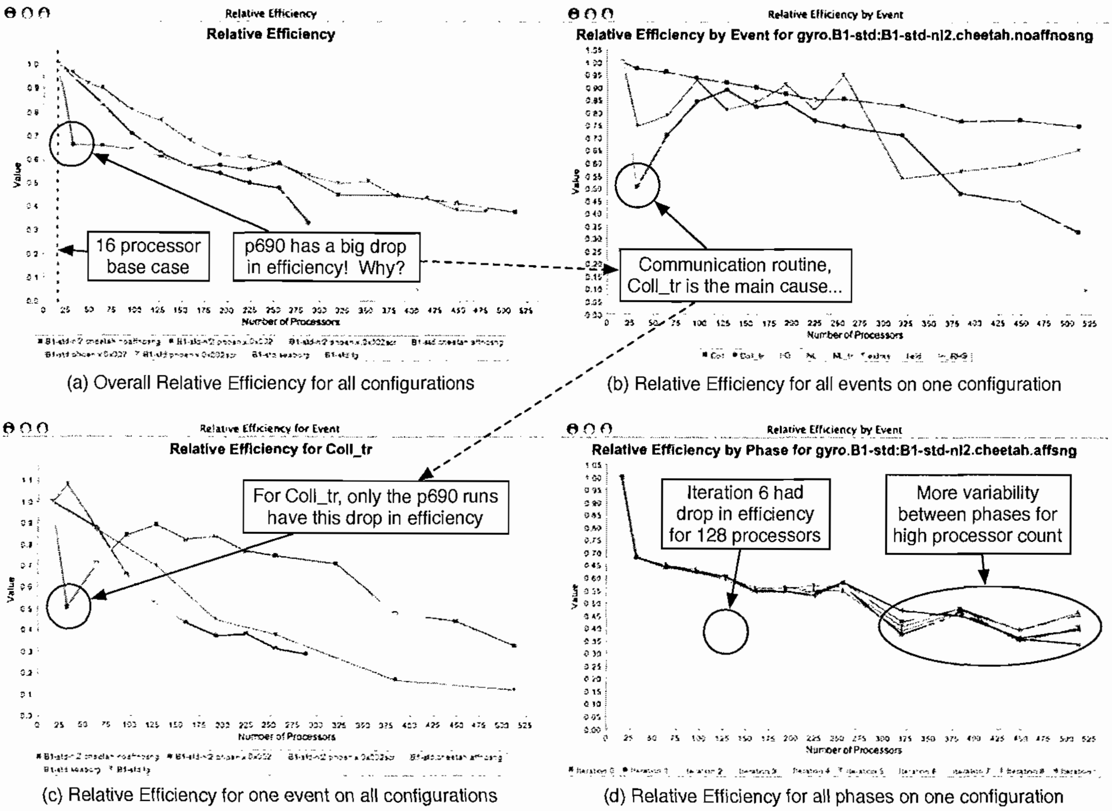
GYRO

GYRO[32] is a physics code that simulates tokamak turbulence by solving the time-dependent, nonlinear, gyrokinetic-Maxwell equations for both ions and electrons. It uses a five-dimensional grid and advances the system in time using a second-order, implicit-explicit (IMEX), Runge-Kutta (RK) integrator. The equations are solved in either a local (fluxtube) or global radial domain. GYRO has been ported to a variety of modern platforms, and the data which we have analyzed includes ports to the Cray X1, SGI Altix, TeraGrid, and the IBM p690 and SP3.

GYRO was the subject of a PERC [106] tool evaluation effort at Oak Ridge National Laboratory (ORNL). The scientists there executed the application many times on several machines with different configurations in an effort to perform an in-depth analysis of GYRO, and to evaluate current performance tools. The TAU project team participated in the effort, and we have access to the performance data available. This data comes from various sources: embedded timers, HPMToolkit, MPICL, mpiP, and TAU. Preliminary comparative analysis by ORNL of the embedded timers was done manually using Perl scripts [149] to process the data, and gnuplot scripts [152] to generate the scaling charts. Our interest was to see if we could eliminate the need for manually processing the data to construct the charts.

The PerfDMF parsers give the advantage of importing profile data from multiple sources. Parsers were already available for HPMToolkit, mpiP, and TAU. For our comparative analysis study, we decided to focus on the data collected by the hand-instrumented application timers. It was a straightforward matter to write a PerfDMF parser for this data. The instrumentation tracks seven events of interest, two of which are communication events, and the data is divided into execution phases. There are three benchmark data sets used (B1, B2, and B3), each with a different number of time-steps. The B1 benchmark runs for 500 time-steps, and outputs performance data every 50 time-steps, giving ten phases. This data was loaded into the PerfDMF database as call-path data, modeled after the phase-based analysis data structures available in TAU. Because the data given was aggregate data across all processors, cluster analysis would be of no use, as the data would form one cluster (the average behavior). So, we focused our attention to comparative evaluation across the combinations of benchmark type, machines, and platform configurations.

As mentioned above, there are several types of comparison available in PerfExplorer. The figures in Figure 11 show the relative efficiency comparison for the B1 benchmark. We start by comparing the total execution time for the application, across all machines and configurations in the study, using 16 processors as our base case. In Figure 11(a), it is obvious that the IBM p690 (cheetah) has a sharp dip in efficiency when going from 16 to 32 processors. PerfExplorer provides the ability to "drill-down" through the data, for example, 11(b) shows the relative efficiency by event for only one configuration of the p690. This view is, in effect, a event scalability view. The dip in efficiency is caused primarily by the `Coll_tr` event, which performs transpose communications before and after the main collision routine. By doing a total execution percentage breakdown (not shown), we learned that the other events which do not scale well do not contribute significantly to the overall runtime of the application. By looking at the data from another perspective in Figure 11(c), we can view only the `Coll_tr` event for all machines/configurations in the study, and see that only the p690 has this large dip in efficiency, for both configurations tested.


Relative Efficiency
Relative Efficiency
Value
Number of Processors
16 processor base case
p690 has a big drop in efficiency! Why?
Relative Efficiency by Event
Relative Efficiency by Event for gyro.B1-std:B1-std-nl2.cheetah.noaffnosng
Communication routine, Coll_tr is the main cause...
Relative Efficiency for Event
Relative Efficiency for Coll_tr
For Coll_tr, only the p690 runs have this drop in efficiency
Relative Efficiency by Event
Relative Efficiency by Phase for gyro.B1-std:B1-std-nl2.cheetah.affsng
Iteration 6 had drop in efficiency for 128 processors
More variability between phases for high processor count

(a) Overall Relative Efficiency for all configurations

(b) Relative Efficiency for all events on one configuration

(c) Relative Efficiency for one event on all configurations

(d) Relative Efficiency for all phases on one configuration

FIGURE 11: Relative efficiency comparisons in PerfExplorer for various machines and configurations when running the GYRO application. In (a), there is a noticeable drop in efficiency for the two IBM p690 runs (cheetah), when looking at total execution time. By comparing the scalability of all events for that particular execution, we see that the significant cause for the drop is due to poor performance for the Coll_tr event, shown in (b). The other three events which scale poorly are I/O related, and/or do not contribute to a large percentage of the total execution time. By comparing this event on all machines and configurations, we see that the p690 is the only machine in the study that has this drop in efficiency for this event, shown in (c). Figure 11(d) is an example of phased-based performance analysis, showing variability between time-steps in GYRO.

In contrast to hand-generated graphs, it is important to realize that PerfExplorer is generating graphs interactively with the user. It is certainly possible to have the production of the graphs seen here be fully automated through the use of scripting, which has been implemented, and will be described in Chapter VI. This could enable performance regression reports to be generated with little user intervention. Also, PerfExplorer's access via PerfDMF to performance data from multiple tools and perspectives provides the opportunity to do more integrative analysis.

Unrelated to the previous three figures, Figure 11(d) shows the breakdown of a single execution by phase (based on time-steps), and shows the variability in the time to solution (shown as relative efficiency) for each phase as the number of processors increase. The phases here are in 50 time-step intervals. One aspect of the data visible is that the variability appears to increase as the number of processors increases. In addition, in this particular execution, iteration 6 had some drop-off in performance, which was found to be a 3x increase in the amount of time spent in one communication routine (`NL_tr`) during that phase. This was likely an anomaly due to an unusual network load. In Chapter VII, we will examine how collecting additional metadata and integrating it into the analysis process can help eliminate the speculation when identifying the causes of performance anomalies.

On page 91, we stated that our goal was to improve upon the manual process of Perl and gnuplot scripts. This manual approach is problematic, as the scripts will only support parsing the embedded timers – not from other performance tools. By generating the scalability charts from repository data stored in a common format, we are able to reproduce the scalability charts without making modifications to any scripts or source code. Once the new data has been loaded into PerfDMF, it is just a matter of selecting the data to be included in the chart, and requesting the chart.

Data collected from HPM Toolkit and TAU was used as input data for the charts with no changes to source code scripts. Doing the same with the manual process using Perl and gnuplot would have required changes to the script. Scalability charts for other applications and other projects is also now available. In summary, while the manual process is inherently difficult to maintain, our approach is flexible to apply to new data sets, and has been used in other studies [54, 78, 157].

Summary

In this chapter, we discussed complexity as a source of concern for parallel performance tools. Faced with systems of greater sophistication, size and integration, performance tools need to address the problems of scalability and data access. We discussed the need for relative differential analysis of large parallel performance profiles, and our integrated comparative study framework. We discussed the comparative analysis charts, and the simplified views for creating slices through the database. Finally, we discussed an application example which was the motivation for much of the work described in this chapter. With this parallel performance analysis framework foundation, we can begin to explore other methods for managing complexity in the analysis process - the use of data mining to reduce the performance profile data. We will describe these methods in the next Chapter.

CHAPTER V

PARALLEL PERFORMANCE DATA MINING

Overview

In the previous chapter, we discussed the growing problem of complexity, with regards to comparative analysis and large scale profiles. Having explored solutions for handling comparative studies in the last chapter, we now need to address the complexity inherent in large scale performance profile analysis. Many of the same points about complexity and usable tools are even more acute when addressing the problems of large profiles than they are to comparative studies. Large profiles are difficult to analyze, simply due to the problem of characterizing and comparing the behavior of tens of thousands of individuals. For example, in the case of summarization, if you compute the mean from thousands of processes, outliers are hidden, and related performance phenomena are difficult to detect. In addition, large profiles are difficult to visualize. When visualizing instrumented events for thousands of processes, do you show the user thousands of pie charts or bar graphs? If the user is shown one bar graph, could it convey usable information with thousands of stacked bars? The traditional visualization solutions are not scalable. What is needed are methods for reducing the quantity of performance data in a meaningful way, without eliding details which are critical in understanding performance behavior. We have identified data mining methods, such as dimension reduction, correlation and clustering as potential solutions to the data reduction problem.

As mentioned in Chapter II, Ahn and Vetter demonstrated the efficacy of clustering large-scale performance profile data to aggregate the performance data across processes. With the steady march toward petascale applications, profiles with tens of thousands and even hundreds of thousands of processes are common. In order to effectively manage these large profiles, aggregation approaches are necessary, and clustering seems a natural fit. Methods of dimension reduction are also necessary, to determine parameters which contribute to variance in the data between processes, and to reduce noise. Finally, correlation analysis is helpful in determining relationships between measured events with regard to performance, and in determining correlations between collected metrics.

The overall goal of the PerfExplorer project is to create a software infrastructure to help conduct parallel performance analysis in a systematic, collaborative, and reusable manner. Our objective in this chapter is to describe how we can integrate sophisticated data mining techniques to analyze large-scale parallel performance data. Given existing robust data mining tools, we designed PerfExplorer to interface cleanly with these tools and make their functionality easily accessible to the user. Without this domain-specific support, doing this type of analysis is a convoluted process of data exportation, management and specialized analysis, which would have to be repeated for each subsequent study.

Approach

Data mining of large-scale parallel performance data seeks to discover features of the data automatically, using statistical techniques. Areas in which we are interested are clustering, summarization, association, regression, and correlation. Cluster analysis is the process of organizing data points into statistically similar groupings, called clusters, in order to discover classes in the data. Summarization is the process of describing the similarities within, and dissimilarities between, the discovered clusters. Association is the process of finding relationships in the data. One such method of association is regression analysis, the process of finding independent and dependent correlated variables in the data.

Clustering

Cluster analysis is a valuable tool for reducing large parallel profiles down to representative groups for investigation. There are two types of clustering analysis considered. Both *hierarchical* and *k-means* analysis can be used to group parallel profiles into common clusters, and then the clusters are summarized. Initially, we are interested in similarity measures computed on a single parallel profile as input to the clustering algorithms, although other forms of input are possible. Here, the performance data is organized into multi-dimensional vectors for analysis. Each vector represents one parallel thread (or process) of execution in the profile. Each dimension in the vector represents an event that was profiled in the application. Events can be any sub-region of code, including libraries, functions, loops, basic blocks or even individual lines of code. In simple clustering examples, each vector represents only one metric of measurement. For our purposes, some dissimilarity value, such as *Euclidean* or *Manhattan* distance, is computed on the vectors.

Hierarchical clustering is a form of clustering which starts with individuals, and works to organize them into clusters by merging the two closest members into a new cluster. Initially, each individual is assigned to a different cluster with size 1. Using a Manhattan distance calculation between the cluster centers, the two closest clusters are merged into one cluster, and the mean is calculated for the new cluster. The process is continued until there is only one cluster. The result is typically displayed as a tree dendrogram.

k-means clustering groups the individuals into k common groups, or clusters. The clustering is performed by selecting k initial cluster centers, and assigning the individuals to the cluster to which they are the closest, using a Euclidean distance calculation. New cluster centers are computed as the mean of the members of the cluster, and the process is repeated until convergence. k-means clustering results are typically displayed as scatter plots, using the two or three variables which represent the most variance in the data, intended to show separation between the identified clusters.

Dimension Reduction

In each of the clustering methods, a distance calculation is performed between individuals in the set. The dimensions of the n-dimensional space are defined as the attributes of each of the individuals. As described in the previous section, each individual in our data set is each thread of execution. Each dimension in the vector represents a measured metric for an instrumented region of code, or event, that was profiled in the application. Clustering algorithms perform reasonably well on datasets of low dimensions, with “low” defined as less than 15 [8]. Unfortunately, we have test datasets with dimensions over 100. As pointed out by several authors [8, 47, 88], locality-based clustering methods are not fully effective when clustering high dimensional data. In high dimensional data sets, it is very unlikely that data points are nearer to each other than the average distance between data points because of sparsely filled space. As the dimensionality increases, the difference between the nearest and farthest neighbors within a cluster approaches zero. As a result, a high dimensional data point is equally likely to belong to any cluster. Because of this, dimension reduction is necessary for accurate clustering.

There are three types of dimension reduction we will discuss. The first type of dimension reduction is to ignore dimensions which are less significant. That is, only consider dimensions which account for a large percentage of the overall runtime of the application. The user specifies a minimum percentage, and any events which, on average, constitute less than that percentage of the total execution are not included. For example, if time is the metric of interest and the user sets a minimum percentage of 3%, only events which, on average, constitute greater than or equal to 3% of the execution time will be included in the dimensions. For the sPPM application discussed further below, a setting of 1% reduced the number of dimensions from 105 down to 10.

Another method of dimension reduction is *random linear projection*. It has been demonstrated by Dasgupta[22] that data from a mixture of k Gaussians can be projected into just $O(\log k)$ dimensions while still retaining the approximate level of separation between the clusters. In addition, even if the original clusters are far from spherical, they are made more spherical by linear projection. Because of these two properties, random linear projection is a another alternative for dimension reduction. A drawback of random linear projection is that meaningful data could be discarded, resulting in a failure to discover natural clusters in the data.

The third dimension reduction method is *Principal Components Analysis* (PCA). PCA is designed to capture the variance in a particular dataset in terms of the dimensions which define the maximum amount of variation within the dataset. Because each resulting set of components are orthogonal to, and therefore uncorrelated with each other, this method helps to remove correlated variables in the data, leaving only the data which describes the maximum variance. Ahn and Vetter[4] used this technique to demonstrate that many hardware counters are often highly correlated.

Unfortunately, there is a drawback to PCA. Because PCA normalizes the data as a result of its calculation, all weighting of the input variables is lost. PCA assumes that all input dimensions are weighted equally, and it is not clear whether this should be the case when analyzing performance data.

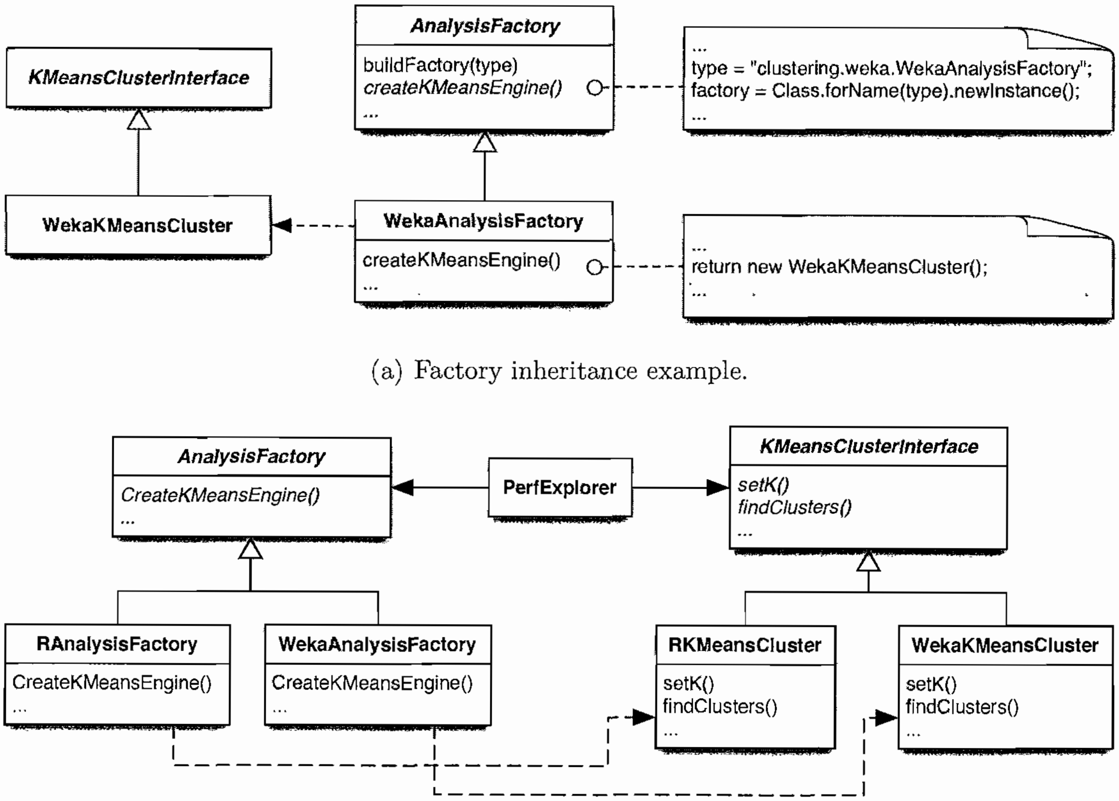
Correlation

Another analysis method useful for examining large parallel profiles is to compute the *Coefficient of Correlation.* This is a measure of the strength of the linear relationship between two variables, x and y. The coefficient value will lie between 1.0 and -1.0, indicating whether there is a linear relationship (close to 1.0), no relationship (close to 0.0) or a negative linear relationship (close to -1.0) between the two variables. The coefficient of correlation is useful in determining the relationships among different events and metrics in the data. For example, in parallel applications which implement message passing, there often are relationships between sending and receiving events. Also, hardware counters are often highly correlated, and this type of analysis can help differentiate between the metrics which will contribute to the understanding of the performance of the application and those which can be ignored or not collected at all.

Implementation

The methods described in the previous section are included in PerfExplorer by integrating the aforementioned libraries R and Weka. As shown in the Java package diagram in Figure 8, the analysis operations are implemented in the `clustering` package. The functionality in each of the libraries is abstracted to a higher level interface, to allow for common code execution, regardless of which library is used at runtime. The abstraction is defined in the `clustering` package in the form of factory objects [41], processing interfaces, and general purpose data objects. This way, support for another statistical or data mining package (such as Octave [63] or Matlab) can be integrated by adhering to the abstracted interface. The `clustering.AnalysisFactory` abstract class defines the requirements for generating a new analysis factory, which is implemented, for example, in the `clustering.r.RAnalysisFactory`. The application then refers to the instantiated factory through the higher level object, and implementation specific objects constructed by the factory are referenced through higher level objects, providing a consistent interface regardless of the runtime implementation.

Figure 12 shows the inheritance structure of the analysis factories, and a concrete example showing the `KMeansClusterInterface` implementation.



(a) Factory inheritance example.

(b) Factory inheritance behavior.

FIGURE 12: Data mining interface and implementation structure.

In 12, the `AnalysisFactory` abstract class is extended by the concrete class `WekaAnalysisFactory`, including a method for creating a `WekaKMeansCluster` class, which is a concrete implementation of the `KMeansClusterInterface` interface. The note shows that when the static method `buildFactory("Weka")` is called, Java reflection is used to call the constructor for the `WekaAnalysisFactory`. The other note shows that when the `createKMeansEngine()` method in the `WekaAnalysisFactory` object is called, the constructor for the `WekaKMeansCluster` class is called. As long as the factory is referenced through the abstract

AnalysisFactory methods, the rest of the code is abstracted from the choice of R or Weka. Figure 12(b) shows what happens when the Weka factory, referenced through the abstract factory methods, is asked to create a KMeansClusterInterface. The CreateKMeansEngine code in the concrete factory knows to create a WekaKMeansCluster object, which PerfExplorer handles as an abstracted KMeansClusterInterface. As long as this returned object is referenced through the interface methods, the rest of PerfExplorer is abstracted from the underlying implementation.

Application

sPPM

The sPPM [83] benchmark solves a 3D gas dynamics problem on a uniform Cartesian mesh using a simplified version of the PPM (Piecewise Parabolic Method) code. The code is written to simultaneously exploit explicit threads for shared memory parallelism and domain decomposition with message passing for distributed parallelism. sPPM has demonstrated good processor performance, excellent multi-threaded efficiency, and excellent message passing parallel speedups. The sPPM code is written in Fortran 77 with some C routines, OpenMP, and MPI. We compiled and executed sPPM on parallel computing resources at Lawrence Livermore National Lab (LLNL). Instrumentation and measurement was done with TAU. sPPM has been well studied, and much is known about its execution and performance behavior. Our interest was to see if PerfExplorer could uncover these well understood features. The program uses MPI to communicate between computer nodes and OpenMP to activate parallelism within a node. By clustering thread performance for different metrics, PerfExplorer should discover these relationships and which metrics best distinguish their differences.

sPPM was executed on LLNL's Frost machine. Frost is a cluster of 16-way IBM Power3 processors running AIX. sPPM used 16 processes (one per node), with 16 threads per process for a total of 256 threads of execution. Clustering on execution time, initial analysis showed that events executed by the master threads dominate. Worker threads sit idle for nearly all of the time, since they are only active during short OpenMP loops. Plus, the master threads perform all communication operations. The only metric which showed a balanced work load is that of floating point operations, where the work is nearly evenly distributed among the threads. For any choice of metrics, PerfExplorer will take the clustering results and display the representative performance for each cluster based on the average performance from the threads that are members of the cluster.

Consider the clustering analysis results for floating point instructions shown in Figure 13. Before the clustering is done, the data is reduced by setting a threshold of 2% - any events that are less than 2% of the total floating point operations are discarded. That leaves only four dimensions for the clustering: `DIFUZE`, `DINTERF`,

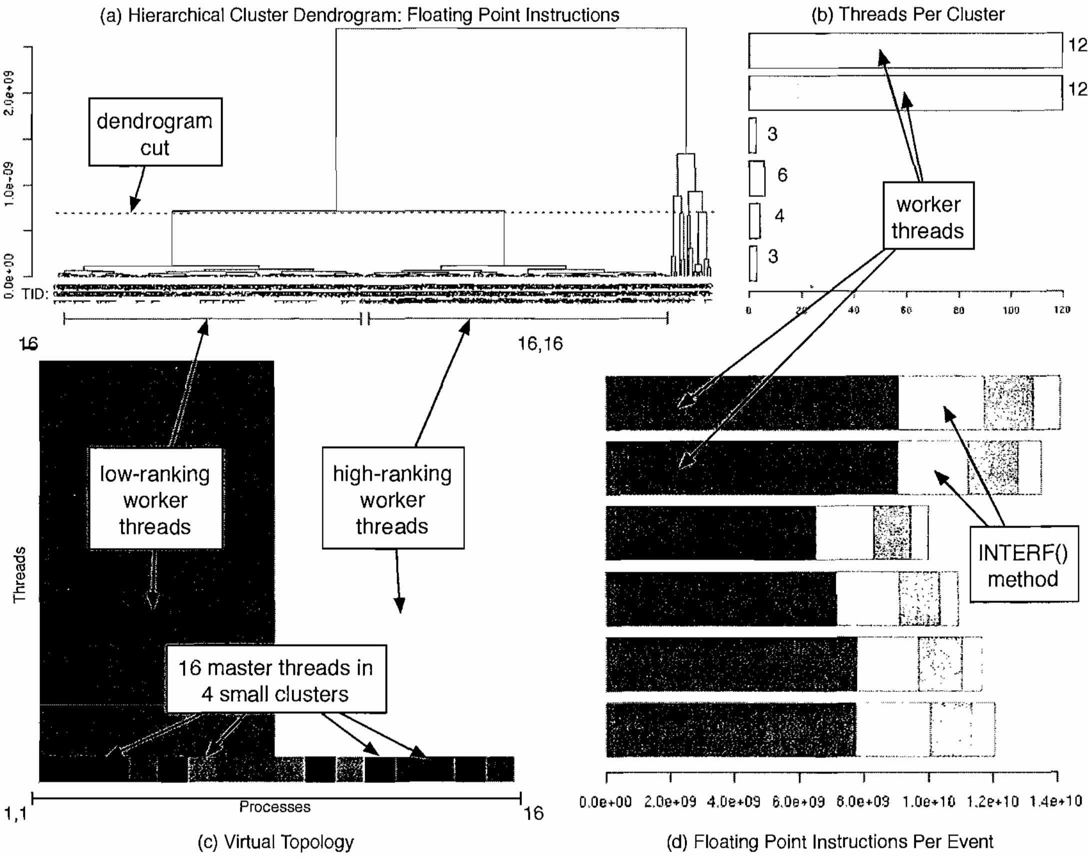
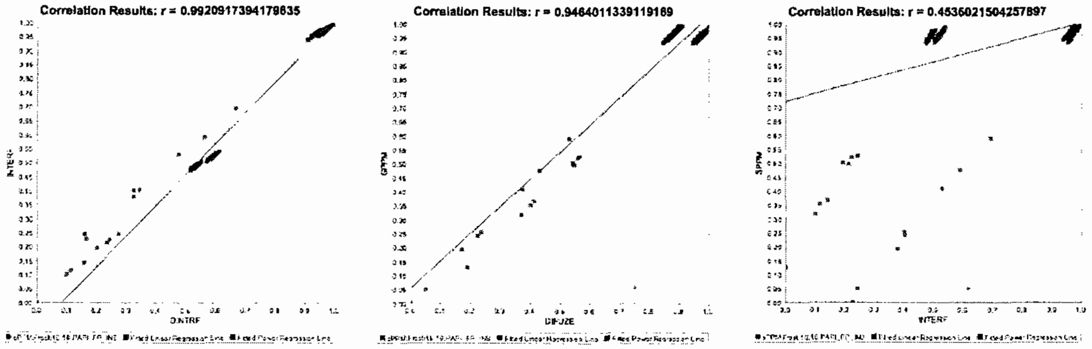


FIGURE 13: Cluster results in PerfExplorer. The dendrogram in (a) shows cluster relationship of floating point instructions when running sPPM on Frost. For clustering with $k = 6$, the histogram in (b) shows membership counts for each cluster and the graphic in (c) shows the virtual topology of the 16 processes (columns) and 16 threads (rows). Notice the worker threads are split into two distinct clusters. (d) shows the average behavior for each cluster. Notice that the higher ranked worker threads (second bar graph down) execute fewer floating point instructions in the INTERF() method.

INTERF, and SPPM. The hierarchical clustering is used in PerfExplorer to help the user select a logical number of clusters. The dendrogram shows the similarity ordering between the threads. k-means clustering is performed with k values of the integers 2 through 10, inclusive. The clustering results are examined to determine cluster grouping and representative performance. The virtual topology in Figure 13(a) gives a view of how clusters map to threads and processes. The floating point instructions in sPPM have an unusual, but not entirely unexpected, clustering. There is a clear distinction between the master threads and worker threads, as expected. There are also subtle differences between four groups of master threads. However, the worker threads do not have a uniform behavior. As noted by Ahn and Vetter[4] and reproduced here, the higher ranking threads seem to execute 3% fewer floating point instructions than lower ranking threads. This is appears to be true only for the worker threads. A selection of $k \geq 6$ is required to show this relationship in the cluster results. Further investigation shows that the difference is primarily in the `INTERF` method, used to construct the simplest possible monotone parabolas within zones of the grid. However, when the application is run on MCR, a cluster of dual-processor machines, the higher ranking threads execute *more* floating point instructions in the `INTERF` method than the lower ranking threads do, for both worker *and master* threads.

Without additional information about the differences between the lower and higher ranking threads, and how the work is distributed, it is difficult to explain the behavior without speculation. Our desire to explain results such as this is what motivated us to examine the use of additional metadata and inference rules, as described in Chapter VII. By correlating metadata values with performance results, and examining those correlation results in the context of an expert system, we can begin to eliminate the guesswork in describing performance phenomena.



(a) DINTERF and INTERF, $r = 0.992$.
(b) SPPM and DIFFUZE, $r = 0.946$.
(c) SPPM and INTERF, $r = 0.454$.

FIGURE 14: Correlation scatterplots between events on sPPM data. Each scatterplot plots two events, and the linear regression line is shown through the data.

Figure 14 shows the result of correlation analysis on the same data, reduced to the same four events. Figure 14(a) shows the correlation between DINTERF and INTERF, which has a correlation coefficient of 0.992, and Figure 14(b) shows the correlation between SPPM and DIFFUZE, which has a correlation coefficient of 0.946. These pairs of events are highly correlated, even without taking clustering into consideration.

Figure 14(c) shows the correlation between SPPM and INTERF, which has a correlation coefficient of 0.454. While these events appear correlated visually, they are only truly correlated within their cluster results. There are two classes of correlated values, each representing the difference in floating point operations between higher ranked threads and lower ranked threads.

What these correlation results tell us is that the floating point operations executed in each of these events are highly correlated, and that there are clear differences which are visible in the data. The primary difference between each instance is whether it is a master thread or not, and the secondary difference is the rank of the thread, which has an impact on the number of floating point operations in the DINTERF and INTERF methods. While we know that these differences exist, we are unclear as to what is affecting the floating point operations, other than the rank of the threads. However, the difference is small (less than 3%), and likely does not have a significant effect on the overall runtime.

Summary

In this chapter, we discussed the need to handle the volume of data inherent in large scale parallel profiles. With hundreds of thousands of potential data points, traditional analysis tools can be overwhelmed by the data, and subtle nuances in the analysis can be lost. Our approach consists of reducing the data, both with dimension reduction, and with cluster analysis. By reducing the large numbers of individuals into representative classes, we simplify the analysis process. Correlation analysis also gives us insight into which events are correlated, either through a cause-and-effect relationship, or because they are each affected by the same outside force. By applying these methods, we gain new insight into otherwise unmanageable data sets. However, as we saw in the application example, without additional context information, we are unable to fully explain the differences and similarities found in the performance data. In later chapters, we will approach this problem with new methods, and describe our subsequent redesign of the analysis framework.

CHAPTER VI

AUTOMATION AND COMPONENTIZATION

Overview

While our work with parametric analysis and data mining was useful when applied to parallel performance data, the analysis results were only a re-description of the performance information. There was little interpretation of performance behavior or guidance given about performance problems. In this way, our application of these methods did not significantly distinguish itself from other performance tools. In particular, there was no mechanism for meta analysis, or some way to evaluate the performance aggregations, correlations and clusters and *explain* or *diagnose*, based on symptomatic behavior, what might be happening in the application. As we shall see in Chapter VII, we need to integrate metadata into the analysis, and we need to encode and apply expert knowledge to the problem.

As we discuss our approach to analysis workflow automation, we will often consider two use cases. The first case focuses on understanding the performance of processes or threads of an execution by analyzing the performance data from a *single experiment*. The second case compares performance data from *multiple experiments* to understand differences between parallel executions. In both cases, our approach provides a means to explore the performance data, to analyze "good" and "bad" characteristics with respect to different performance metrics, and to identify code regions most affected. However, we realized that a better solution would also include support for explicit *process control*, a requirement for creating new analysis workflows and running repeated analysis procedures as non-interactive analysis automation. Also, there was an opportunity to add *higher-level reasoning* or *analysis* of the performance result in order to explain what caused the performance differences.

Because our initial approach was limited with respect to extensibility, there was no explicit mechanism for creating new analysis workflows or combining methods in new and unexpected ways. The modified framework must also be concerned about how to interface with application developers in the performance discovery process. A consistent interface to the analysis operations is necessary to simplify the creation of new analysis workflows. The ability to engage in process programming, knowledge engineering (metadata and inference rules), and results management are key to creating data mining environments specific to the developer's concerns.

In addition to process control, *persistence* and *provenance* mechanisms for retaining analysis results and history are also important for productive performance analytics. The extended data management schema included tables for storing clustering and correlation results, and the charts could all be output to disk for later reuse. If we were to provide process automation and extension support, we also saw the need to store intermediate results, and to annotate our analysis results with the methods which generated them.

As a first step to address these issues, we needed to redesign our approach to analysis, and move from a focus on an exploratory graphical user interface application to a programmable analysis framework, with better intermediate data storage support and process automation, extensibility and programmability.

Approach

Component Interfaces

In order to begin to address the needs of extensibility and process control, new software components are necessary to meet the desired goals. Figure 15 shows the primary PerfExplorer interface layers, and indicates which layers are new in the redesign. A new component interface layer is designed to allow the GUI and Scripting interfaces to access the data and analysis components through common methods. The performance data and accompanying metadata are stored in the PerfDMF database. Performance data is used as input for statistical analysis and data mining operations, as was the case in the original version of PerfExplorer. The new design adds the ability to make all intermediate analysis data and final results persistent. Expert knowledge

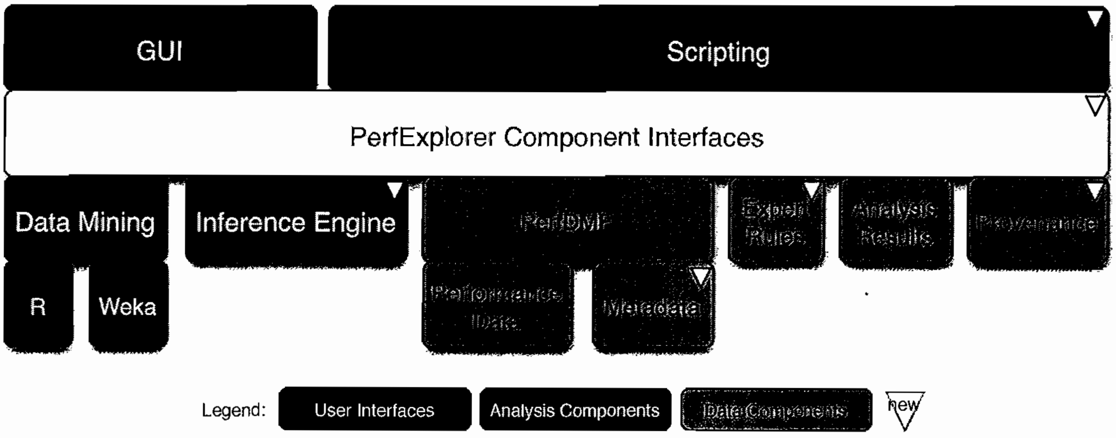


FIGURE 15: The redesigned PerfExplorer component integration, with the new components indicated.

is incorporated into the analysis, and these new inputs allow for higher-level analysis. An inference engine is added to combine the performance data, analysis results, expert knowledge and execution metadata into a performance characterization (to be described in Chapter VII). The provenance of the analysis result is stored with the result, along with all intermediary data, using object persistence. The whole process is contained within a control framework, which provides user control over the performance characterization.

Figure 16 shows the interaction between components in the new PerfExplorer design. Data components which represent outputs from analysis components could potentially be used as inputs for further analysis. All intermediate operations and outputs are stored as provenance objects in the performance database, and are available for future analysis. With this design, any analysis result can be used as an input to another analysis operation.

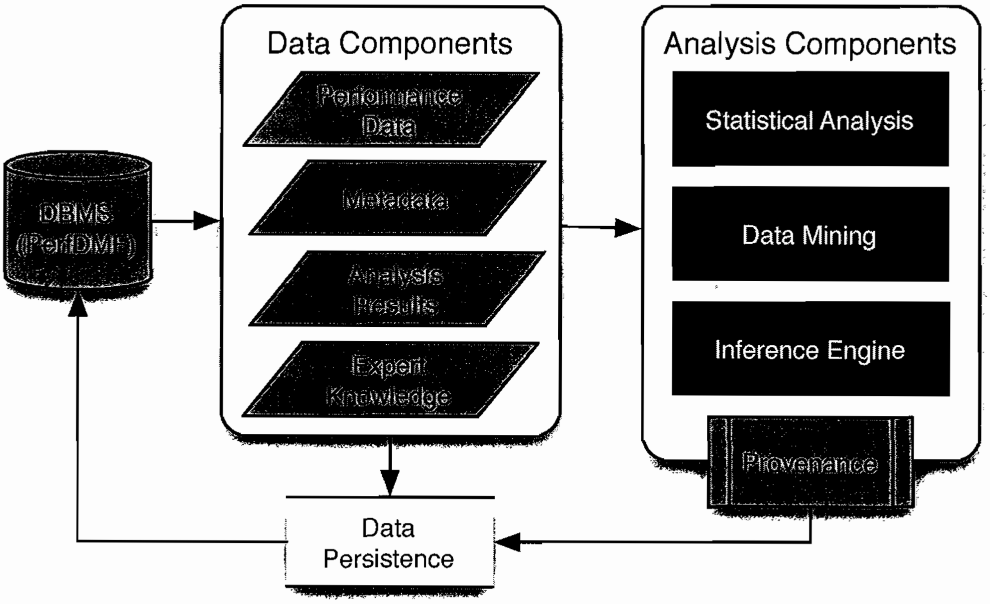


FIGURE 16: PerfExplorer components and their interactions.

All of the analysis operations described in Chapters IV and V, as well as several new ones, have been wrapped with a programming interface in its own Java package. The `glue` package (shown in Figure 8) provides a user-accessible programming interface, with limited exposure to a number analysis data objects and process objects, as shown in Figure 17. The top level interface for the processing classes is the `ProcessAnalysisOperation`, which defines the interface for all process objects. The interface consists of methods to define input data, process the inputs, get output data objects, and reset the process. The `AbstractPerformanceOperation` is an abstract implementation of the `ProcessAnalysisOperation` interface, and provides basic internal member variables, such as the input data and output data. Finally, the `DeriveMetricOperation` class is an example of a concrete extension of the

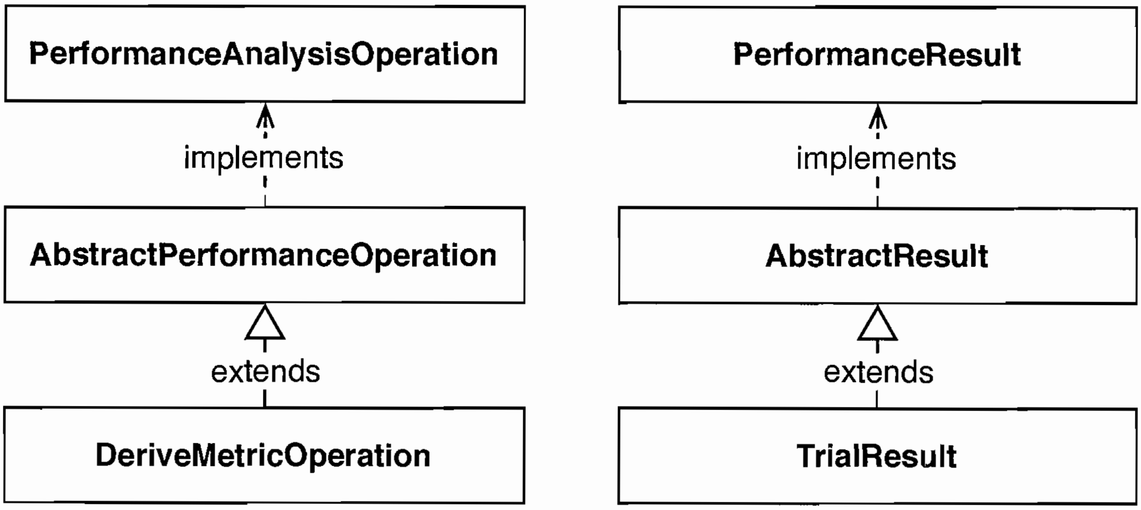


FIGURE 17: PerfExplorer script object hierarchy.

AbstractPerformanceOperation class, and will take one more more input data sets with two or more metrics each, and generate a derived metric representing either the addition, subtraction, multiplication, or division of one metric with the other. Corresponding with the operation hierarchy is the data hierarchy. At the top of the hierarchy is the PerformanceResult interface, which defines basic methods for accessing the profile data within. The abstract implementation of the interface is AbstractResult class, which defines many internal data structures and static constants. The TrialResult class is an example of a class which is a concrete implementation of the abstract class, and provides an object which holds the performance profile data for a given trial, when loaded from PerfDMF.

Table 7 shows a list of the available operations. It is important to note that the output from each of the operations is a new object, and the input data objects are effectively immutable, in that they are not changed by the operation. While

TABLE 7: PerfExplorer operations available through the script interface.

AbstractPerformanceOperation
BasicStatisticsOperation
CQoSClassifierOperation
CopyOperation
CorrelationOperation
DefaultOperation
DeriveAllMetricsOperation
DeriveMetricOperation
DifferenceMetadataOperation
DifferenceOperation
ExtractCallpathEventOperation
ExtractContextEventOperation
ExtractEventOperation
ExtractMetricOperation
ExtractNonCallpathEventOperation
ExtractPhasesOperation
ExtractRankOperation
ExtractUserEventOperation
KMeansOperation
LinearRegressionOperation
LogarithmicOperation
MergeTrialsOperation
MetadataClusterOperation
NaiveBayesOperation
PCAOperation
PerformanceAnalysisOperation
RatioOperation
SaveResultOperation
ScalabilityOperation
ScaleMetricOperation
SplitTrialPhasesOperation
SupportVectorOperation

this results in number of new objects being created, we rely on the built-in garbage collection of Java to ensure that we do not consume too much memory. If it is necessary to reduce the memory footprint of an analysis process, input data references should be set to null once they are no longer needed. The provenance mechanism requires that the intermediate data from any operation be available after it has been used as input to another operation.

Process Control

One of the key aspects of the new PerfExplorer design is the requirement for process control. While graphical user interfaces and data visualization are useful for interactive data exploration, performance investigations invariably involve a pipeline of operations with the results of one step determining the next. Manual control of such a multi-step process is error-prone and limits automation. To increase the analysis and discovery power of PerfExplorer, it is necessary to develop programmable process control and analysis abstraction.

In order to synthesize analysis results, expert knowledge, and metadata into higher-level meta-analysis process, PerfExplorer needed an extension mechanism for creating higher-order analysis procedures. One way of doing this is through a scripting interface. Adding a scripting interface to Java applications is relatively straightforward. As long as the script interpreter is written completely in Java, it can be called from the application, and has access to all of the Java classes. We decided to use Jython[109], a Python interpreter. With such an interface, the analysis process is under the control of the script author, who is able to produce the characterization at runtime by modifying the logic of a given analysis process. If we decide to use a different scripting language (or to support a range of scripting languages), it is easy to replace the Jython interface with any other script interpreter written in Java that can be integrated into a Java program. There are many such examples[69], such as Jacl (Tcl), Rhino (JavaScript), JRuby (Ruby), BeanShell (Java), Groovy (Python/Ruby-like), JudoScript (JavaScript-like), and Pnuts (Java-like).

All of the application objects are theoretically available to the script interface, but we restrict the access to the `glue` package. With the aforementioned `glue` interface, it is straightforward to derive new metrics, perform analysis, and automate the processing of performance data. An example script is shown in Figure 18. This simple

```
# import the PerfExplorer packages
import glue.*
import edu.uoregon.tau.perfdmf.*
# create a rulebase for processing
ruleHarness = RuleHarness.useGlobalRules("openuh/OpenUHRules.drl")
# load a trial
trial = Utilities.getTrial("Fluid Dynamic", "rib 45", "1_8")
result = TrialMeanResult(trial)
# calculate the derived metric
stalls = "BACK_END_BUBBLE_ALL"
cycles = "CPU_CYCLES"
operation = DeriveMetricOperation.DIVIDE
operator = DeriveMetricOperation(result, stalls, cycles, operation)
derived = operator.processData().get(0)
# compare values to average for application
for event in derived.getEvents():
  MeanEventFact.compareEventToMain(derived, mainEvent, derived, event)
# process the rules
ruleHarness.processRules()
```

FIGURE 18: Sample script for PerfExplorer.

example loads some inference rules, loads a trial from PerfDMF, derives an inefficiency metric, and then compares each event's exclusive value with the inclusive value of main before processing the rules, where an event is defined as any instrumented code region. The inference rule processing will be explained in depth in Chapter VII.

Provenance and Data Persistence

Any scientific endeavor is considered to be of "good provenance" when it is adequately documented in order to allow reproducibility. For parallel performance analysis, data and analysis provenance includes all raw data, analysis methods and parameters, intermediate results, and inferred conclusions. In addition to reproducibility, performance provenance makes it possible to rationalize analysis decisions. Explanations of performance results are contextualized by the chain of evidence that produced them. Furthermore, the ability to make the outcome of all analysis operations persistent, not just the final summarization, is important for enabling dynamic, inference-driven analysis that depends on evaluation of intermediate results. Some related analysis workflows may begin by performing the same initial operations, and then branch in a number of possible directions. Rather than recompute intermediate results, those results can and should be stored for later use.

One of the operations listed in Table 7 is the `SaveResultOperation`. This operation is used to save intermediate analysis results in the database. For example, when deriving metrics for analysis, once the metric is derived it makes sense to save the result in the database to be used for later analysis, or with other analysis tools which interface with the data management system. Reusable scripts which derive metrics can check for the existence of the derived metric first, before doing the operation.

However, it is not enough just to store the derived metric in the database. In order to have good provenance, the database needs to also store how the results were generated. PerfExplorer automatically creates a `Provenance` object whenever analysis is performed. In the base constructor of the `AbstractPerformanceOperation`, the operation is automatically added to the `Provenance` object, which maintains a list of operations which are performed on the data. When the `SaveResultOperation` object is called, a metadata field is generated to record the sequence of operations which were used to generate the derived data, and the options used for each operation. The metadata is intended to be both human and machine readable, so that the intermediate or final results can be regenerated and understood.

Application

Power Modeling with GenIDLEST

With increasing energy costs and green supercomputing gaining in popularity, there is an increased call for more integrated tool infrastructures that support both performance and power analysis capabilities. In addition to the performance-centric analysis capabilities of PerfExplorer discussed so far, we will demonstrate how using a similar process, PerfExplorer may be applied for power analysis. In a study to model processor power consumption and energy efficiency, we used PerfExplorer to compute a power metric based on the work of Bui et al. [11]. The power metric is based on hardware performance counters and on the on-die components of the processor. We wrote PerfExplorer scripts to obtain power dissipation and energy consumption estimates for the GenIDLEST application running on Itanium hardware. The GenIDLEST application is described in detail on page OpenUH - GenIDLEST. We used GenIDLEST running the 90rib dataset as a power/energy case study.

Different levels of standard optimizations for the OpenUH compiler were applied ranging from O0 (all optimizations are disabled) to O3 (applies the most aggressive optimizations including loop nest optimizations). The application was run in parallel with MPI on 16 processors on the Altix 300.

The PerfExplorer scripts load the data sets, derive the power estimation metrics, save the derived power metrics, and output the results. Substituting the access rates and scaling factors of our hardware to the power modeling equations, our final power estimation formula is shown in Equations VI.1 and VI.2 (units are in Watts).

$$\begin{aligned} CPU\ Power &= (Total\ Instructions\ /\ Total\ Cycles) * (0.0459 * 122) \\ L1\ Power &= (L1\ References\ /\ Total\ Cycles) * (0.0017 * 122) \\ L2\ Power &= (L2\ References\ /\ Total\ Cycles) * (0.0171 * 122) \\ L3\ Power &= (L3\ References\ /\ Total\ Cycles) * (0.935 * 122) \end{aligned} \tag{VI.1}$$

$$TotalPower = CPU\ Power + L1\ Power + L2\ Power + L3\ Power \tag{VI.2}$$

In order to compute the energy consumption for each event, we multiply the power consumption for the event with the time spent in the event, in seconds. In order to compute the FLOPS/Joule, we divide the number of floating point operations in each event by the energy consumed in the event. The results reported in Table 8 are for the entire application, and are normalized to the unoptimized case.

TABLE 8: GenIDLEST relative differences for different optimization settings, using 16 MPI processes on a 90riblet problem. Optimization level O0 is the baseline.

Metric	**-O0**	**-O1**	**-O2**	**-O3**
Time	1.0	0.338	0.071	0.049
Instructions Completed	1.0	0.471	0.059	0.056
Instructions Issued	1.0	0.472	0.063	0.061
Instructions Completed Per Cycle	1.0	1.397	0.857	1.209
Instructions Issued Per Cycle	1.0	1.400	0.909	1.316
Power Consumed (Watts)	1.0	1.025	1.001	1.029
Energy Consumed (Joules)	1.0	0.346	0.071	0.050
FLOP/Joule	1.0	2.867	13.684	19.305

The results from the case study show that power dissipation generally increases with higher optimization levels while energy decreases as more aggressive compiler optimizations are applied. These results are consistent with previous studies that examine the effects of compiler optimizations on power and energy efficiency [144, 115]. Also consistent with a previous research study [144], we find that the instruction count is directly proportional to energy consumption and a similar relationship exists between instructions per cycle (IPC) and power dissipation. A higher instruction count translates to more work for the CPU and so energy increases.

Our long-term goal with this work is to provide feedback from PerfExplorer analysis to the OpenUH compiler, to direct the compiler to target optimizations based on metrics other than performance. A thorough discussion of our work with the OpenUH compiler is on pages 171 and 179. PerfExplorer might be able to direct either the compiler or programmer to optimize for low power, low energy, or both using inference rules (see Chapter VII). The results from Table 8 suggest that O0 should be enabled for low power, O3 enabled for low energy, and O2 for both power and energy efficiency for the OpenUH compiler. Compiling for low energy can be important for embedded and scientific applications, whereas compiling for low power has more significant long-term effects in terms of system reliability and reduced cooling and operational costs for large-scale servers.

Summary

In this chapter, we discussed the need for process automation, extensibility, provenance, and persistence. For easy extensibility and control, a script interpreter was added to the analysis framework. Our solution consists of wrapping analysis operations with component interfaces, integrating a script interpreter, adding object persistence and archival of intermediate and final results. With these extensions to our framework, we have shown how analysis processes can be captured and extended, and demonstrated the new capabilities through the automated generation of derived metrics in a power estimation study. In the next chapter, we will discuss our final contribution, the integration of context metadata and expert knowledge into the meta-analysis process.

CHAPTER VII

KNOWLEDGE ENGINEERING

Overview

PerfExplorer has strategies for managing large data sets both for performance studies and for analysis of profiles with thousands of processors. The initial research prototype has provided insight into several performance analysis projects. However, as Chapters IV and V describe, there is still a need to reason about the possible causes of performance phenomena. Our experiences in those projects and others led us to two key conclusions about how to make PerfExplorer a better analysis tool. First, in order to understand the performance analysis result, we have to capture source code descriptions, the build environment, and the run environment as metadata. That metadata can then be used to explain performance analysis outcomes with the context in which they are derived. Second, in order to truly diagnose performance problems, we have to encode knowledge about known performance problems, application properties, and machine properties, and apply them to the results.

We discussed our integration of metadata in Chapter III, and will revisit it in this context later in this chapter. However, the exploration of expert knowledge is a relatively new topic to our discussion. Many common performance problems in parallel applications can often be identified by expert analysts who know what they are looking for. These experts draw upon their past experiences in examining application performance, and can therefore apply lessons learned from one evaluation or tuning study to another study. However, the use of individual experts is not a scalable solution, and they are not available to assist with every analysis investigation. By including expert knowledge into the analysis process, PerfExplorer can begin to interpret performance phenomena. Once the metadata is in the performance database, we can include it in the analysis process, and apply rule-based deduction systems to the performance analysis results. By extending the analysis into the realm of *meta analysis*, we begin to find possible diagnoses for our performance symptoms.

In addition to using metadata in the performance characterization and diagnosis process, metadata can also be used in runtime recommendation systems. By examining the context in which the application is running, and comparing the context to results from previous executions of the application, we can recommend algorithmic or parametric changes to the application to improve performance. We will describe our technique for using our data analysis framework to construct a classifier, which is used as the basis for a runtime recommendation system.

Metadata Integration

There are several types of performance study commonly seen in the parallel performance literature (see page 118). In one such example, a scalability study, the number of processors used and/or the input problem size is varied, and empirical performance results are compared with expected results, based on baseline comparisons. For more general performance studies, we have identified eight common categories of metadata, listed in Table 9, along with example values for each category and an example of a known assumption, or *expert knowledge*, in that category that could be helpful in analyzing the performance of an experiment.

As an example, the first category, "machines", includes differences between architectures, such as when porting an application, or performing an application benchmarking study on more than one architecture. Parameters such as CPU type and speed, the amount of cores per CPU, the number of CPUs per node, and other hardware characteristics all represent key information when comparing two or more architectures. In order to use this information, performance assumptions can be made in the analysis process which will help guide the analysis. For example, consider an application executed with the same configuration on two different machines. If the metadata shows that the only difference between the two machines is the speed of

TABLE 9: Parametric categories and corresponding example assumptions (expert knowledge) in those categories.

Category	**Parameter Examples**	**Possible Assumptions**
Machines	processor speed/type, memory size, number of cores, cache hierarchies	CPU A faster than CPU B
Components	MPI implementation, linear algebra library, runtime component	component A faster than B
Input	problem size, input data, problem decomposition	smaller problem means faster execution, vice-versa
Algorithms	FFT vs. DFT	algorithm A faster than B for problem $>$ X
Configurations	number of processors, runtime parameters, number of iterations, environment variables	more processors means faster execution, vice-versa
Compiler	compiler choice, compiler options, pre-compiler usage, code transformations	execution time: -O0 $\geq$ -O1 $\geq$ -O2 $\geq$ -O3 $\geq$ -fast
Code Relationships	call order, send-receive partners, concurrency, functionality	code region has expected concurrency of X
Code Changes	code change between revisions	newer code expected to be faster

the CPU, then the analysis should relate the performance differences between the two executions to the differences in speed. As another example, suppose that we can identify a region of code as inherently sequential. Any scalability analysis of this region could then assume that regardless of the number of processors used, this region of code will take the same amount of time, and not be flagged as an unexpected bottleneck, but rather a known scaling issue. While these are simple examples, they illustrate the potential utility that expert knowledge about an execution can provide to the performance analysis. Some expert knowledge would be specific to the analysis task at hand, while other examples would be reusable across many performance studies.

Rule-Based Systems

One way to include expert analysis processing into a performance data mining and diagnosis system is to integrate a *rule-based system* [153]. A rule-based system is a software application or library which attempts to replicate the diagnostic capabilities of an experienced human expert within a specific domain. Rule-based systems logically infer conclusions through a method of deductive reasoning. Rules are defined as if-then statements, where the *if clause*, or *antecedent*, establishes the required preconditions for the rule to execute, and the *then clause*, or *consequent* contains new facts, commands, and conclusions. A simple example of rules in a rule base is the following pair of rules, R1 and R2:

R1 **if** x has gills
then x is a fish

R2 **if** x is a fish
then x swims

With this rule base, we can infer whether or not an object x swims. Rule-based systems have a *working memory*, in which facts are asserted. If we assert the fact "Nemo has gills", then the object "Nemo" is bound to the variable x in the working memory. The fact "Nemo has gills" matches a pattern in the rule base, namely the "x has gills" antecedent for rule R1, and so rule R1 asserts true. As a result, the consequent "x is a fish" is populated with the object "Nemo". We now have a new fact asserted in the working memory, "Nemo is a fish", which then causes the antecedent of R2 to assert true, and the consequent of R2 is executed. We have now made two deductions from our initial premise, and in addition to our initial knowledge that Nemo has gills, we have concluded that Nemo is a fish and that Nemo swims.

This type of rule processing, where facts are asserted and rules with true if clauses are processed, is called *forward chaining.* Forward chaining is used to answer questions such as "Does Nemo swim?". Another type of rule processing is called *backward chaining.* With backward chaining, we can answer questions such as "Does anything swim?". In that case, we start by processing any rules in which "x swims" is a conclusion, and work backwards to see if we have enough evidence to conclude that something does, in fact, swim. We match to the consequent of rule R2, which leads us to check if there are any other rules in which the antecedent of R2 is in a consequent of any rules, or is an asserted fact in our working memory. The consequent of rule R1 matches, which leads us to check if there are any asserted facts in our working memory which match the pattern "x has gills". If we have asserted that Nemo has gills, then we now enough evidence to prove our original hypothesis.

Rule-based systems are processed by inference engines[153]. A rudimentary implementation of an inference engine would be to exhaustively check every rule in a rule base to check for rules which assert true, at every step of the deductive process. Instead, an efficient and commonly used algorithm is the *rete* algorithm.

The rete algorithm constrains its search behavior to a sequence of connected nodes of a logical net, reducing the search space considerably.

With regard to performance analysis, a rule-based system could be used to perform diagnosis of known performance problems. Intermediate and final analysis results could be asserted as facts, and expert knowledge to interpret those results can be defined as rules in a rule base. When a diagnostic conclusion is reached, the analysis system would report a diagnosis to the analyst, and include suggestions for improving the performance of the application.

Machine Learning

Parametric measurement studies are typically undertaken to obtain a representative sample of the performance space of a parallel code. From this performance data, a statistical characterization or model of the space can be obtained. One use could be for predicting the performance at another location in the space. Another common application is to build recommender systems based on classifiers. Classification systems are a type of machine learning in which training data is input into a decision tree or space partitioning algorithm to construct a classifier. Classifiers belong a particular class of machine learning known as *supervised learning*, in which vetted training data with pre-selected attributes and known class types are used to train the classifier, as opposed to exploratory, unsupervised learning methods such as clustering, in which the class identifications are not known ahead of time. With a trained classifier, new test data can be identified as belonging to one of the identified classes. Classifiers can also be used to perform numerical prediction.

There are a number of possible classification methods [154], such as decision trees, statistical methods, and neural networks. The first type, decision trees, are constructed with a "divide-and-conquer" approach. The individuals in the training set are separated into two or more groups, as defined by the value of a significant attribute. Each group is then subdivided by another attribute, and the process continues until there are no more attributes to process, the individuals in the group all belong to the same class, or the subgroup contains only one individual. The trees are then pruned, to avoid the problem of overfitting to the less significant attributes of the training data.

Another group of classifiers are those which use statistical methods, such as the probabilistic Naïve Bayes method or Support Vector Machines. When a test individual is evaluated with Naïve Bayes, the probability that the individual would belong to a class is computed based on the probability that a member of that class would have the same attribute values as the test individual. Support Vector Machines evaluate the complete space of individuals, project their attribute space into an orthogonal set of dimensions, and compute the hyperplanes which divide the individuals into their respective classes.

Neural networks are a third type of classifier. Neural networks have the useful property of performing accurate classification in data sets where there are non-linear relationships between the input variables and the identifying classes. One type of neural network is a multilayer perceptron. Like operators in propositional logic, many layers of perceptrons can be linked together to create arbitrarily complex classification methods. The first perceptron layer accepts values from each the input parameters, as well as a bias value. Each successive perceptron layer takes the outputs from the previous layer, as well as a bias value. The final layer aggregates the output from the previous layer, and provides the classification. Because of their ability to approximate non-linear functional relationships in data, neural networks can be very accurate classifiers.

There are a few pitfalls when working with classifiers. The first couple of problems are related to each other, false positives and false negatives. A *false positive* is when a classifier incorrectly identifies a test instance as belonging to a class when it does not actually belong to that class. A *false negative* occurs when a classifier incorrectly fails to identify a test instance that does actually belong to that class. Measurements of these failures are used to compute the accuracy of a classifier. One way to quantitatively evaluate the accuracy is with a *kappa* statistic. A kappa statistic represents the number of instances correctly predicted by a classifier as compared to a perfect predictor. The maximum value for the kappa statistic is 1.0, indicating complete agreement. A random agreement would be represented by 0.0, and a negative value indicates negative agreement.

Another common problem with classifiers is *overfitting.* Overfitting occurs when a classifier does very well in validation tests, but does not accurately classify instances that are not similar to the training data. This often happens when too many variables are included in the training criteria. One method for preventing overfitting with decision trees is pruning, where lower levels of decision trees are eliminated. Ensuring that the training data is a representative statistical sample of the probable (not just possible) test instances will also protect against overfitting.

We are interested in one particular application of classification with regard to performance analysis. Classification can be implemented as part of a recommender system, which to be used for parameter selection. Once performance data for a given application has been collected, it can be used as training data for one or more classifiers. The data can be pre-selected to help train the classifier to optimize for speed, accuracy, efficiency, or some other metric. Then, given the parameters of a test instance at runtime, the recommender system can provide a classification of the problem, and suggest an optimal parameter combination.

Approach

Inference Rules

In order to provide the type of higher-level reasoning and meta-analysis we require in our design, we have integrated an inference rule engine, JBoss Rules[110]. The

JBoss Rules engine implements the efficient Rete algorithm, and also provides the possibility of developing a domain specific language to define the processing rules. JBoss Rules is a pure Java implementation, which makes it ideal for integrating into PerfExplorer. It is also JSR-94 [127] compliant, scalable, and includes a business rules management system for managing rules within applications. The inference engine will be integrated as part of a forward-chaining expert system, in which the analysis results and performance metadata are asserted, and relevant rules are constructed to explain correlations, ratios, and other relationships in the data. JBoss Rules does not support backward-chaining, in which performance diagnoses would be asserted, and then the rules would process to see if there is enough data to confirm any of the diagnoses.

An example rule is shown in Figure 19. This example rule will fire for any and all events which have a higher than average stall per cycle rate (as measured with Itanium hardware counters), and also account for at least 10% of the total run time of the application. The rule can be summarized as:

High Stalls per Cycle:	**if**	metric = STALLS/CYCLES
	and	direction = HIGHER
	and	fact type = “Compared to Main”
	and	severity $<$ 10%
	then	*higher than average stall per cycle* is true

The rule itself has two parts, a `when` clause and a `then` clause. After the declaration of the name of the rule, the `when` clause defines the conditions under which the rule will be fired. In JBoss Rules rules, Java objects can be bound to local variables, and member variables of those objects can also be bound to local variables. In this

```
rule "Stalls per Cycle"
  when f : MeanEventFact (
    m : metric == "(BACK_END_BUBBLE_ALL / CPU_CYCLES)",
    h : higherLower == MeanEventFact.HIGHER,
s : severity > 0.10,
    e : eventName, a : mainValue, v : eventValue,
    factType == "Compared to Main" )
  then
    System.out.println("Event " + e +
      " has a higher than average stall / cycle rate");
    System.out.println("\tAverage stall / cycle: " + a);
    System.out.println("\tEvent   stall / cycle: " + v);
    System.out.println("\tPercentage of total runtime: " + s);
end
```

FIGURE 19: Sample JBoss Rules rule.

example, a `MeanEventFact` object is bound to the local variable `f`, and if the rule asserts true, then the variable `f` is used to refer to the object in the `then` clause, and the Java code in the `then` clause is executed. In this example, an output is printed to the console, but the `then` clause can also assert new facts, launch new analysis scripts, or execute any arbitrary Java code.

Machine Learning

Because Weka is already integrated into PerfExplorer for clustering and correlation purposes, it makes sense to take advantage of the many classifier implementations and supporting infrastructure available in Weka. In order to construct a classifier to function as a parameter recommender system, we first need to collect training data. As shown in Figure 20, a *Training Driver* is used to generate training data for the classifier construction. The training driver can be one or more scripts to execute an application, or for component applications, a master component which iteratively makes calls to computation components. The training data consists of the properties of the problem being solved, the method used to solve it, and the accuracy (or other measurements of "quality" for which we are interested), and possibly detailed performance information. That data is stored to the PerfDMF database as performance profiles and metadata. PerfExplorer uses this training data to build a classifier (see Figure 21.). The classifier is output to disk, using Java serialization. In order to use the classifier at runtime, a Java runtime component reads the classifier from disk, and uses the properties of the new problem as test input data for the classifier, which recommends the most appropriate solver. This general purpose framework can be used for different types of recommender systems, as we will see in later examples.

In Figure 21, we see the detailed process for building the classifier. The metadata and measurements of "quality" are retrieved from the database. Of all the training data generated, unique combinations, or tuples, of relevant properties are found from the metadata. For each unique tuple, the optimal method for solving problems with those properties is selected, depending on the requirements for recommendation. Of those selections, PCA is optionally performed to determine which input parameters contribute to the variance in the data set. Dimension reduction may not be necessary for some classifier methods which are less sensitive to high degree of dimensions, such

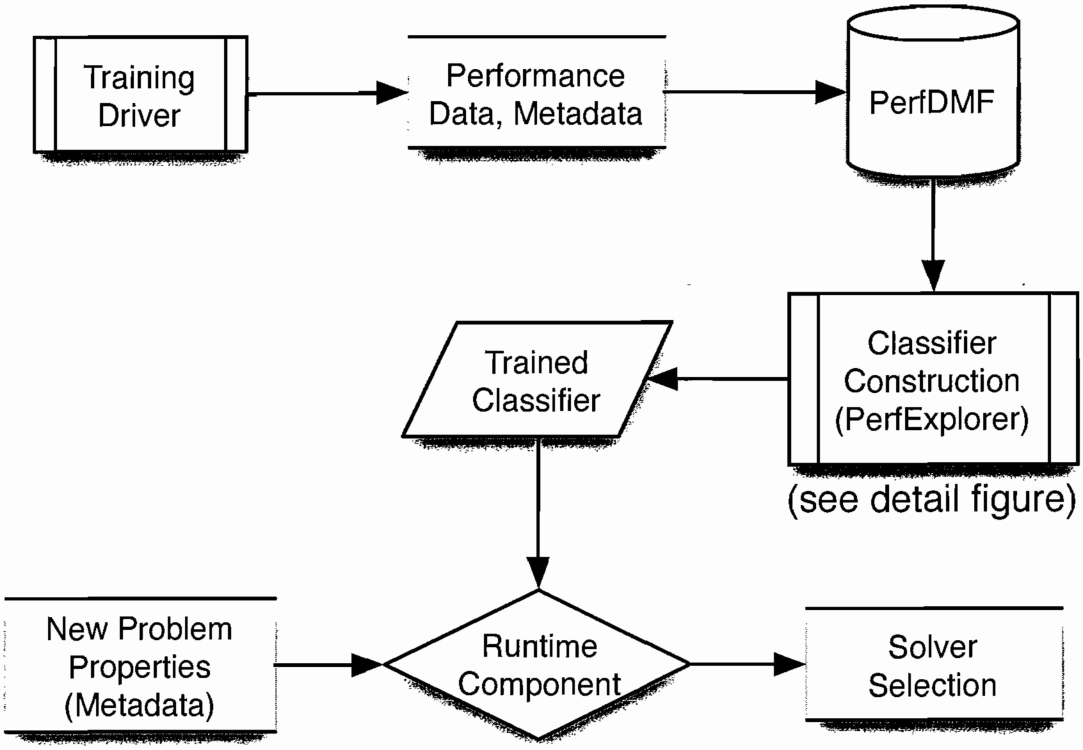

Training Driver
Performance Data, Metadata
PerfDMF
Classifier Construction (PerfExplorer)
(see detail figure)
Trained Classifier
New Problem Properties (Metadata)
Runtime Component
Solver Selection

FIGURE 20: PerfExplorer classifier construction. The details of the classifier construction are shown in Figure 21.

as support vector machines. The reduced data is used to train the classifier, and the trained classifier is output to disk so that the runtime component can use it. Several classifier types have been tested in PerfExplorer, due to the wide variety of classifiers available in Weka. Some classifiers in Weka are disqualified due to their in ability to handle non-numerical data. The classifiers tested thus far include three decision tree methods (Alternating Decision Tree, Random Tree, and J48), two statistical methods (Naïve Bayes and Support Vector Machine), and one neural network method (Multilayer Perceptron). Weka includes code to perform N-fold cross validation of

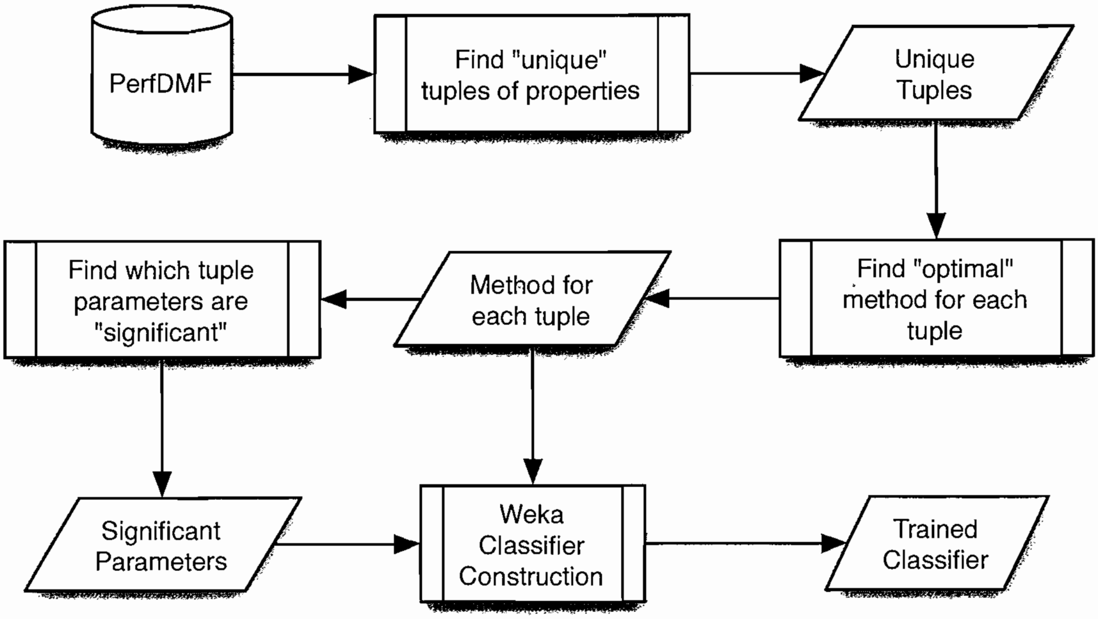

PerfDMF
Find "unique" tuples of properties
Unique Tuples
Find "optimal" method for each tuple
Method for each tuple
Find which tuple parameters are "significant"
Significant Parameters
Weka Classifier Construction
Trained Classifier

FIGURE 21: PerfExplorer classifier construction detail.

classifiers, and PerfExplorer exposes an interface to evaluate any classifier constructed from training data.

After the classifier is constructed and written to disk, a runtime Java component will load the classifier. Given the properties of a new problem, the runtime component will return a recommendation which will ideally outperform the default settings of the application or library.

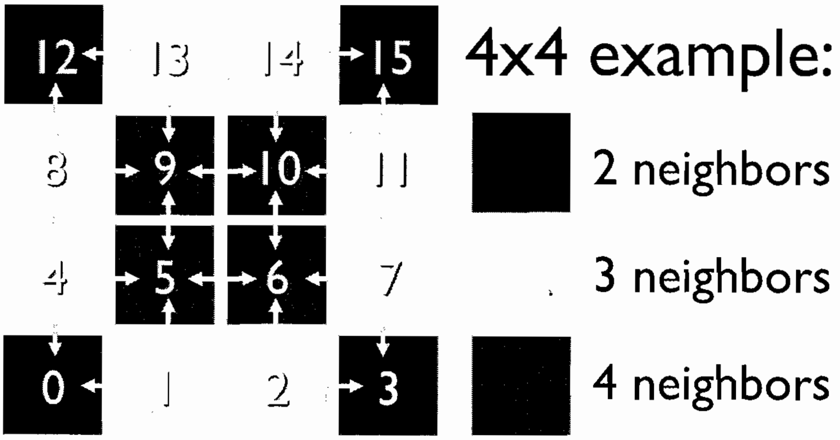
Application

Sweep3D

Sweep3D [84] is an ASCI benchmark code which solves a 1-group, time-independent, discrete ordinates, 3D Cartesian geometry neutron transport problem. The geometry is represented as an IxJxK logically rectangular grid of cells. The main algorithm is a wavefront process across the I and J dimensions, and is pipelined along the K dimension. The algorithm gets its parallelism from the I, J domain decomposition. Sweep3D is written in Fortran 77, and uses MPI.

A sweep in the algorithm proceeds as follows. For a given angle, each grid cell has four equations with seven unknowns (six grid cell faces plus one central). Boundary conditions for each cell complete the system of equations. The solution is by a direct ordered solve known as a *sweep*. Three known inflows allow the cell center and three outflows to be solved. Each cell's solution then provides inflows to three adjoining cells (one each in the I, J, and K directions). This represents a wavefront evaluation with a recursion dependence in all three grid directions.

The algorithm begins with the topmost cell (of the K axis) of the southwestern-most subdivision of cells (cell 0 of Figure 22(a)). After that cell solves for its three outflows, those solutions are communicated with the northern and eastern neighbors (cells 4 and 1, respectively), using MPI_Send() and MPI_Recv(). There is no communication in the K dimension. The next time step, the processor handling cell 0



(a) Problem decomposition for 16 processor problem.

Cell Type	Neighbors	MPI_Send calls	MPI_Recv calls
Corner	2	1440	1440
Edge	3	2160	2160
Interior	4	2880	2880

(b) MPI behavior in Sweep3D.

FIGURE 22: Sweep3D MPI behavior.

advances to compute the next value in the K dimension, and the processors handling cells 4 and 1 begin computing a solution to the first value in the K dimension. The solution then continues, until the sweep completes at the diagonally opposite corner of the collection of cells. There are two more sweeps across the I, J dimensions, one starting from the north east corner of the I, J grid (cell 15), and another starting from the south east corner of the grid (cell 3). What is important to note about the communication is that the corner cells only have two neighbors, the edge cells only have three neighbors, and interior cells have four neighbors, as shown in Figure 22. With respect to MPI communication, there are three classes of cells in the problem solution, and we should expect to see a correlation in the communication behavior of the application and a given cell's number of neighbors. The table in Figure 22(b) shows that there is a direct correlation for a 16 processor example.

For our experiment, we executed Sweep3D on Jaguar [97], the Cray XT3/XT4 system at Oak Ridge National Laboratory. At the time, Jaguar was in the process of transitioning from an XT3 system to an XT4 system, and there were two primary differences between the two node types. The XT3 nodes have slower memory (DDR-400MHz, 5.986GB/second) and a slower network interconnect (Seastar SS1, 1.109GB/second), whereas the XT4 nodes have faster memory (DDR2-667MHz, 7.116GB/second) and a faster network interconnect (Seastar SS2, 1.873GB/second). If an application is allocated a hybrid mix of nodes, performance can be affected when either memory intensive code or communication intensive code is executed. When collecting performance data, we used the PAPI interface to collect hardware counter data for total instructions, L1, and L2 cache behavior, floating point operations, and wall clock time.

We executed Sweep3D on 256 processors, with a 800x800x1000 problem setup. The I, J domains would be partitioned 16 ways, with 50x50 grid cells per processor. As described earlier, communication performance should be affected by the number of neighbors that a cell has. Our goal with this experiment is two-fold. We wish to detect the per-node communication performance difference, and correlate it with the neighbor count. Secondly, if we are given a hybrid allocation, will we see performance differences, and if so, can we correlate those differences with hardware properties?

```
# load the rules used to process our analysis results
ruleHarness = RuleHarness.useGlobalRules("rules/GeneralRules.drl")
ruleHarness.addRules("rules/ApplicationRules.drl")
ruleHarness.addRules("rules/MachineRules.drl")

# set the PerfDMF configuration, then select the trial
Utilities.setSession("apart")
trial = Utilities.getTrial("sweep3d", "jaguar", "256")
trialResult = TrialResult(trial)
trialMetadata = TrialThreadMetadata(trial)

# get the top 5 events from the trial
metric = input.getTimeMetric()
getTop5 = TopXEvents(input, time, AbstractResult.EXCLUSIVE, 5)
top5 = getTop5.processData().get(0)

# correlate performance results with metadata per thread
correlator = CorrelateEventsWithMetadata(input, meta)
outputs = correlator.processData()
RuleHarness.getInstance().assertObject(outputs.get(0));

RuleHarness.getInstance().processRules()
```

FIGURE 23: Jython script to analyze Sweep3D data.

The script we used to analyze the performance is shown in Figure 23. The script has four phases, and executes them in order in the Jython default main execution. First, the rules are loaded into our inference engine. The rules are loaded first, so that if performance assertions are made during intermediate analysis, the rules are already loaded. Secondly, the database configuration is selected, and the performance data and metadata are loaded. The top five events, as a percentage of the total application runtime, are extracted from the full data set. Finally, for each of those five events, each of the hardware counter measurements for each of the processes was correlated

```
rule "High Correlation with Metadata"
  when
    c : CorrelationResult ()
    d : FactData ( e : event, m : metric, t : type,
      e2 : event2, m2 : metric2 )
    eval ( d.getType() == AbstractResult.EXCLUSIVE ||
      d.getType() == AbstractResult.CALLS )
    eval ( d.getType2() == CorrelationResult.CORRELATION )
    eval ( Math.abs(d.getValue()) >= 0.9 &&
      Math.abs(d.getValue()) < 0.95 )
    then
      StringBuffer buf = new StringBuffer();

      buf.append(e + ": ’" + m + ":" +
        AbstractResult.typeToString(t.intValue()));
      buf.append("’ metric is ");
      if (d.getValue() < 0.0) {
        buf.append("inversely ");
      }
      buf.append("correlated with the metadata field ’");
      buf.append(e2 + "’.");
      System.out.println(buf.toString());
      NumberFormat f = new DecimalFormat ("0.000");
      System.out.println("\tThe correlation is " +
        f.format(d.getValue()) + " (high).");
end
```

FIGURE 24: Example rule to interpret Sweep3D results.

with the metadata for the node on which the process ran. All of the correlation data was asserted as facts. At the end of the script, the rules are processed, and the output is generated.

Figure 24 shows one of several rules used to evaluate the correlation between performance results and process metadata. In this example, we are looking for a *high* correlation, which we have defined as a correlation coefficient between 0.9 and

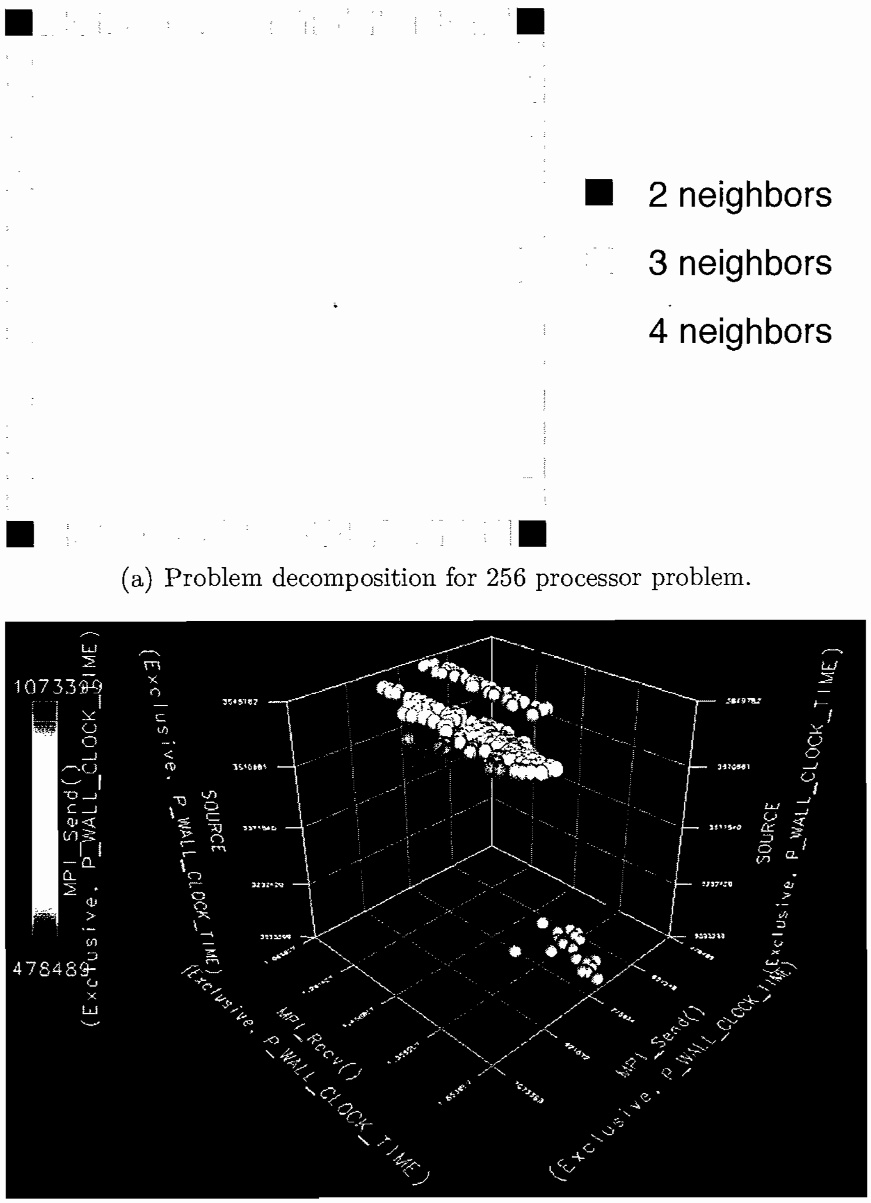
0.95. Other rules are looking for a *direct* correlation (greater than 0.99), a *very high* correlation (between 0.95 and 0.99), or a *moderate* correlation (between 0.85 and 0.9). Of course, all of these "strength of correlation" definitions are relative and arbitrary, but they are the ones we chose to use for our analysis. In this example, we are looking for a correlation between any metadata field and the exclusive value of an event or the number of times that an event was called. When this rule asserts true, then a message is printed to the user's console. Note that the correlation can be inversely correlated, and is handled by the same code.

Output from the analysis script is shown in Figure 25. The framework detected that the performance of the main computation routine, `SOURCE`, was inversely affected by the memory speed and network interconnect speed. That is, the XT3 nodes executed slower than the XT4 nodes for this event (the higher speeds resulted in shorter execution times). The framework also detected a direct correlation between the number of neighbors that a process has and the number of calls to MPI communication routines, and the amount of time spent in those routines. The performance data is shown in a four dimensions in Figure 26(b). The vertical separation between clusters is due to the differences between the XT3 and XT4 nodes - the faster processes located on the bottom of the view. The horizontal and color separation between clusters is due to the difference between corner, edge and interior cells, with respect to the time spent in `MPI_Send()`. A correspondingly color-coded layout of the 256 processes is shown in Figure 26(a).

```
Firing rules...

SOURCE [{source.f} {2,18}]: 'P_WALL_CLOCK_TIME:EXCLUSIVE' metric is
inversely correlated with the metadata field 'Seastar Speed (MB/s)'.
    The correlation is -0.996 (direct).
SOURCE [{source.f} {2,18}]: 'P_WALL_CLOCK_TIME:EXCLUSIVE' metric is
inversely correlated with the metadata field 'Memory Speed (MB/s)'.
    The correlation is -0.998 (direct).

        (redundant output removed)

MPI_Send(): 'P_WALL_CLOCK_TIME:CALLS' metric is correlated with the
metadata field 'total Neighbors'.
    The correlation is 1.000 (direct).
MPI_Send(): 'PAPI_FP_INS:EXCLUSIVE' metric is correlated with the
metadata field 'total Neighbors'.
    The correlation is 0.860 (moderate).

        (redundant output removed)

MPI_Recv(): 'P_WALL_CLOCK_TIME:CALLS' metric is correlated with
the metadata field 'total Neighbors'.
    The correlation is 1.000 (direct).
MPI_Recv(): 'PAPI_FP_INS:EXCLUSIVE' metric is correlated with
the metadata field 'total Neighbors'.
    The correlation is 1.000 (direct).
```

FIGURE 25: Selected PerfExplorer output from Sweep3D analysis. The rules identified the correlation between the number of neighbors for each processor and the performance of MPI_Send() and MPI_Recv(). The rules also identified a negative correlation between the performance of the main computation routine and the memory bus speed, which identifies whether the process ran on an XT3 or XT4 node.

These results demonstrate only a small fraction of what is possible with the expert system integration into our analysis framework. The power of the expert system analysis is limited only by the time and effort to capture knowledge as inference rules. The amount of effort required to capture that knowledge should not be



(a) Problem decomposition for 256 processor problem.

(b) 3D View of MPI performance behavior.

FIGURE 26: Sweep3D problem problem decomposition for 256 processors, and how many neighbors each processor will have.

underestimated, however. Constructing and maintaining an analysis rule base will be a significant undertaking, even with a well defined domain such as this. That said, we consider our framework to be extensible and flexible enough to capture analysis knowledge as we encounter new examples, and will eventually come to represent a powerful diagnostic system.

CQoS - GAMESS

Some scientific applications or libraries have a vast number of parameters and parameter options. These libraries are difficult to use efficiently, because while the default parameters might provide the right solution, they may not provide the most efficient solution, and they may even fail to provide a correct solution. It is often likely that the optimal parameters for one data set are not the optimal parameters for a different data set. The Computational Quality of Service (CQoS) project[101, 86] seeks to build component-based recommender systems for scientific applications to aid in parameter selection in order to optimize performance with respect to time, accuracy or some other metric. As part of this effort, PerfExporer is being used to analyze performance data and construct the recommender system.

One such scientific application is the General Atomic and Molecular Electronic Structure System (GAMESS), which is used for first principles quantum chemistry experiments. GAMESS computes the ground-state wavefunctions and energy of an atom or a molecule using various Hartree-Fock methods such as closed-shell (RHF), high- or low-spin coupled restricted open-shell (ROHF), spin-unrestricted (UHF), and generalized valence bond (GVB). GAMESS also supports the multi-configurational self-consistent field (MCSCF) method. There are three basic run types in GAMESS:

the computation of the ground-state energy, the gradient (first derivative of the energy) or Hessian (second derivative of the energy).

GAMESS is a complex application, and has many options for solving the properties of the wavefunctions. The solution is in the form of a linear combination of (usually) orthogonal basis functions. There are various basis sets available, which provide varying levels of accuracy in the solution. In addition to the methods mentioned in the previous paragraph, there are two implementations for each method, *conventional* and *direct*. The conventional implementation was first, but was too resource intensive in terms of disk space and file I/O requirements on some systems. The direct version was developed to avoid storing intermediate integrals on disk, thus requiring some redundant computations, and is typically two to three times slower than the conventional. However, in parallel environments at higher processor counts, the direct method outperforms the conventional method which has more parallel overhead. The actual point where it makes sense to change methods depends on the wavefunction solution method, the input molecule or atom, as well as the basis set. In addition, at one stage of the algorithm, a second-order Møller-Plesset program (MP2) can be used as an alternate approach to electron correlation. The MP2 method consumes more memory and disk space, and is only advisable under certain conditions. One goal of the recommender system, with regard to the GAMESS application, is to suggest whether to use the direct or conventional method, given the other parameter selections.

TABLE 10: Classifier accuracy for GAMESS using different classifier methods.

Classifier	**Conventional right/wrong**	**Direct right/wrong**	**% correct**	**Kappa**
ADTree	142/1	8/3	97.4%	0.7863
Multi. Perceptron	143/0	7/4	97.4%	0.7647
Random Tree	142/1	3/8	94.2%	0.3762
Naïve Bayes	143/0	4/7	95.5%	0.5149
Support Vector	143/0	3/8	94.8%	0.4105
J48	143/0	0/11	92.9%	0

In order to build a classifier for the recommender system, the GAMESS application was run in a parametric study. The performance data was in the form of some TAU profiles, and GAMESS log files. A PerfDMF parser was written for the GAMESS log files in order to store the performance data and the associated metadata defining the parameter selection. GAMESS was run on Bassi, an IBM POWER5 system at the National Energy Research Scientific Computing Center (NERSC). Bassi has 111 compute nodes, with 8 processors and 32 GB of memory per node. The parameters for the study were modeled after an earlier performance study by Feller [36]. Seven molecules were used: benzene, benzene-dimer (a benzene pair), C60 (a carbon molecule also known as fullerene), GC (a DNA base pair of guanine and cytosine), AT (another DNA base pair adenine and thymine), naphthalene, and napthalene-dimer (a naphthalene pair). GAMESS was executed with MP2 enabled and disabled (MP0). Only the RHF method was used, only one basis set was used, and only the energy computation was performed. The numbers of processing nodes and cores was varied, with the number of cores per node ranging from 1-8, and the number of nodes ranging from 1-32, for a total number of processes between 1 and 256.

In order to build the classifier, a PerfExplorer script was written to perform the work, using the technique described on page VII. The script loaded the trials of interest, and passed them into the `CQoSClassifierOperation` object. The script also identified the metadata fields which were to be used as input to the classification, and which field was to be identified as the class. The input fields identified were the molecule name, the node count, the core count, and whether MP2 was used. The class variable identified was the method (direct or conventional). After processing to find the unique tuples, 154 training instances were selected. The script then constructed the recommendation classifier using six different classification techniques.

For each classification method, the script requested a 10-fold cross validation test, example results of which are shown in Table 10. The best-performing classifier, ADTree, was serialized to disk, in order to be used later as a runtime recommender system. When considering the data in Table 10, it is important to remember that the default method is the conventional method. For 143 of the 154 instances, conventional is the optimal method. However, for larger numbers of processes (extrapolations from the training data, as only the benzine and C60 molecules were executed on more than 64 processors), the direct method has been found to be faster.

To measure the effectiveness of the classifier, another parametric study was performed, using both the default settings for GAMESS and the recommended settings. The recommendations of the ADTree classifier is shown in Figure 11. The table shows which processor counts the classifiers recommend the direct method is

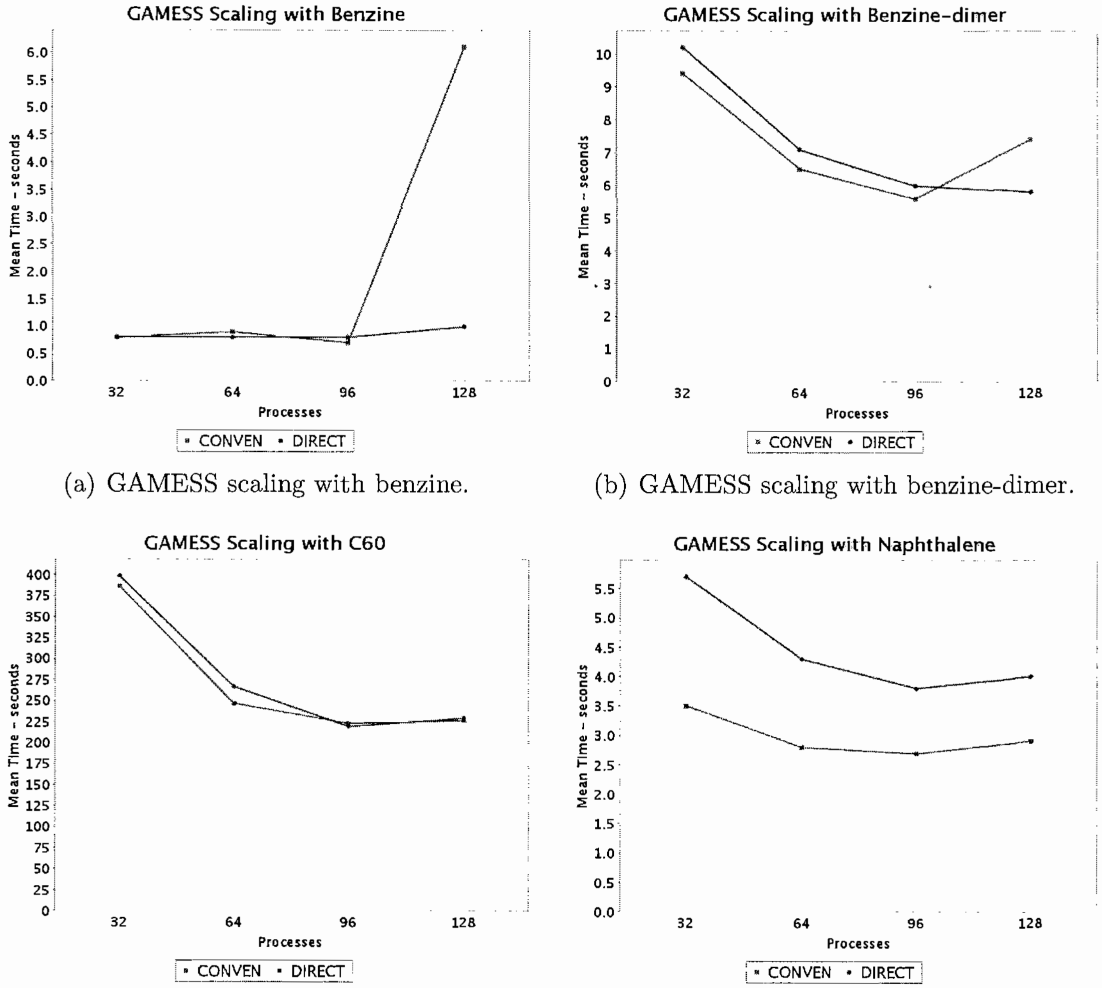
TABLE 11: Direct method classifications for the GAMESS application.

Molecule	MP0/MP2	Recommendation	Actual
AT	MP0	none	none
benzine	MP0	64,96,128	64,128
benzine-dimer	MP0	none	128
C60	MP0	64,96 128	96
GC	MP0	none	none
naphthalene	MP0	none	none
naphthalene-dimer	MP0	none	none

better than the conventional. For 64 processors, only the benzine and C60 molecules have the the direct method recommended. MP2 results were equivalent.

To test the recommendations, GAMESS was executed on Mist, a 128 processor cluster at the University of Oregon. Mist has 16 compute nodes, each with two 2.33GHz Intel Xeon quad-core processors. Each of the molecules was run with MP2 disabled, with 32, 64, 96 and 128 processors. On first glance, it would appear that the classifier did a poor job of recommending the direct method for one of the benzine cases and two of the C60 cases. However, closely comparing the performance of the two methods as well as the input data offers some insight. The scaling behavior of the benzine, benzine-dimer, C60, and naphthalene molecules are shown in Figure 27.

Benzine and benzine-dimer are small molecules, and the parallel and I/O overhead in the 128 processor example dominates runtime for the conventional case. This was correctly predicted for the benzine molecule, but not for the benzine-dimer molecule. It should be noted that no training data was available for benzine-dimer over 64 processors, which was likely the cause for misclassification. As for the C60 case, the misprediction for the 128 processor case has small cost, as the performance of



(a) GAMESS scaling with benzine.

(b) GAMESS scaling with benzine-dimer.

(c) GAMESS scaling with C60.

(d) GAMESS scaling with naphthalene.

FIGURE 27: GAMESS scaling behavior for four molecules: benzine, benzine-dimer, C60 and napthalene. Scaling behavior for AT, GC, and naphthalene-dimer is similar to naphthalene.

the two methods was nearly equal (229 seconds for direct, as compared to 227 for conventional). The C60 case is interesting, in that it is the longest-running molecule in the training data.

Other factors may contribute to the subtle differences between the training data results and the test data results, and is likely due to the differences between the hardware used to collect the training data and the machine used to test the predictions of the classifier. The obvious differences between the two systems are the processor types and speed. There are other differences between the systems which would have a significant impact on scaling performance. Bassi has a dedicated parallel file system, and the conventional method in GAMESS relies heavily on I/O performance for caching intermediate results. The disk space for mist is an NFS link to a common file server which serves many other machines, and local disk for scratch space. To a lesser extent, Bassi is networked using an IBM "Federation" switch, which has dual network cards per node. Mist also has dual network cards, but GAMESS was not configured to use both network connections on the test machine. Adding hardware characteristics to the classifier variables would likely improve the classification. The training data would have to be collected from more than one machine to be useful for classification. Another way to improve the accuracy of the classifier, and make it more useful to other test cases, would to collect metadata about the molecules, other than just their names. This work is part of a multi-year collaborative effort, and is ongoing. Improvements to the classifier are expected as our experience with the application increases.

Summary

In this chapter, we discussed the need to integrate metadata into the analysis process, both between threads or processes of one execution, and also between different two or more executions. We also gave an overview of rule-based systems and discussed using rule-based systems to capture expert knowledge and apply it to the analysis process. Rules can be constructed which contain performance assumptions and known problems which manifest themselves as particular performance behaviors, under certain conditions represented by the captured metadata. Finally, we gave an overview of classifiers, and discussed the use of metadata when generating recommender systems based on classification techniques. In the next chapter, we will present application examples which demonstrate the utility of our combined analysis framework, consisting of the contributions discussed over the last five chapters.

CHAPTER VIII

EXPERIMENTAL STUDIES

Our research contributions in performance databases, data mining, analysis automation, and knowledge representation and inference, discussed in the preceding five chapters, have been captured in the PerfDMF and PerfExplorer tools. At the end of each of the chapters, we discussed small examples of performance analysis studies which demonstrated and in some cases motivated the design and implementation of those contributions. In this chapter, we will describe six analysis examples in depth. Each of the examples benefit from the PerfExplorer application support, as well as the PerfDMF integration. Some of the examples focus on different aspects the other three contributions: data mining, automation, and knowledge engineering. Each of the examples begins with a brief introduction of the application and the platform on which it is executed, followed by a description of the application of our contributions and the results derived, and concludes with a discussion of the results.

Each of the examples in this chapter exercise most if not all of the contributions described in the previous chapters. All of the examples utilize both the PerfDMF and PerfExplorer frameworks. The Miranda study demonstrates the data mining capabilities. The GTC study demonstrates the automation support and phase analysis support in PerfExplorer. The S3D study demonstrates the data mining, automation and knowledge engineering support. The OpenUH studies demonstrate the automation and knowledge engineering support in the framework. Finally, the CQoS/PETSc study demonstrates the automation and machine learning functionality in the framework.

Miranda

Miranda [12] is a research hydrodynamics code ideal for simulating Rayleigh-Taylor and Richtmyer-Meshkov instability growth. It is written in Fortran 95 and uses MPI for communication between processes. The code uses 10th-order compact (spectral-like) or spectral schemes in all directions to compute global derivative and filtering operations. The data is transposed to perform sparse linear (pentadiagonal) solutions and Fast Fourier Transforms (FFTs), which requires mostly synchronous communication in the form of `MPI_Alltoall()` on $\sqrt{N_p \, x, y}$ communicators for global operations, where N_p is processors, and x and y define the problem size. There are some MPI reductions and broadcasts for statistics. The Miranda application has been ported to the BlueGene/L (BG/L) machine [82] at LLNL, and has been executed in configurations up to 32K processors. Prior to the experiments on BG/L, Miranda showed good scalability when tested up to 1728

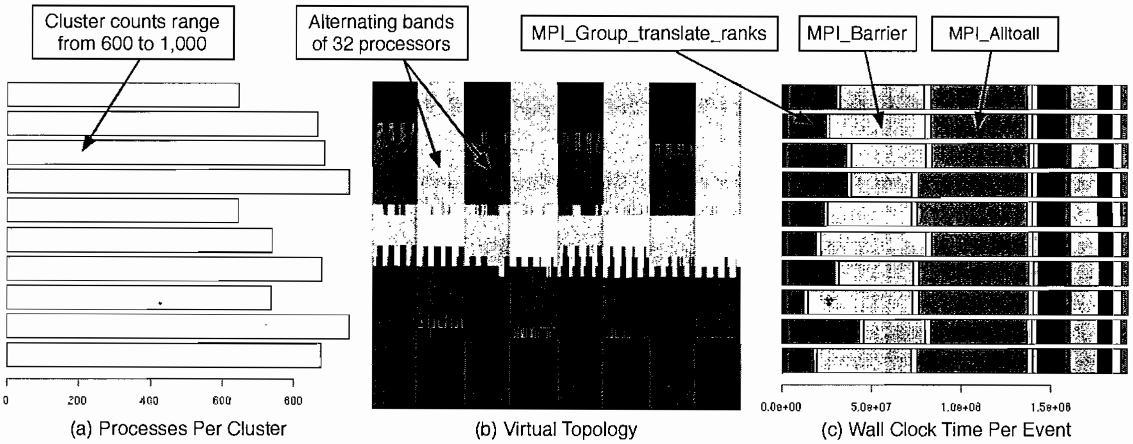


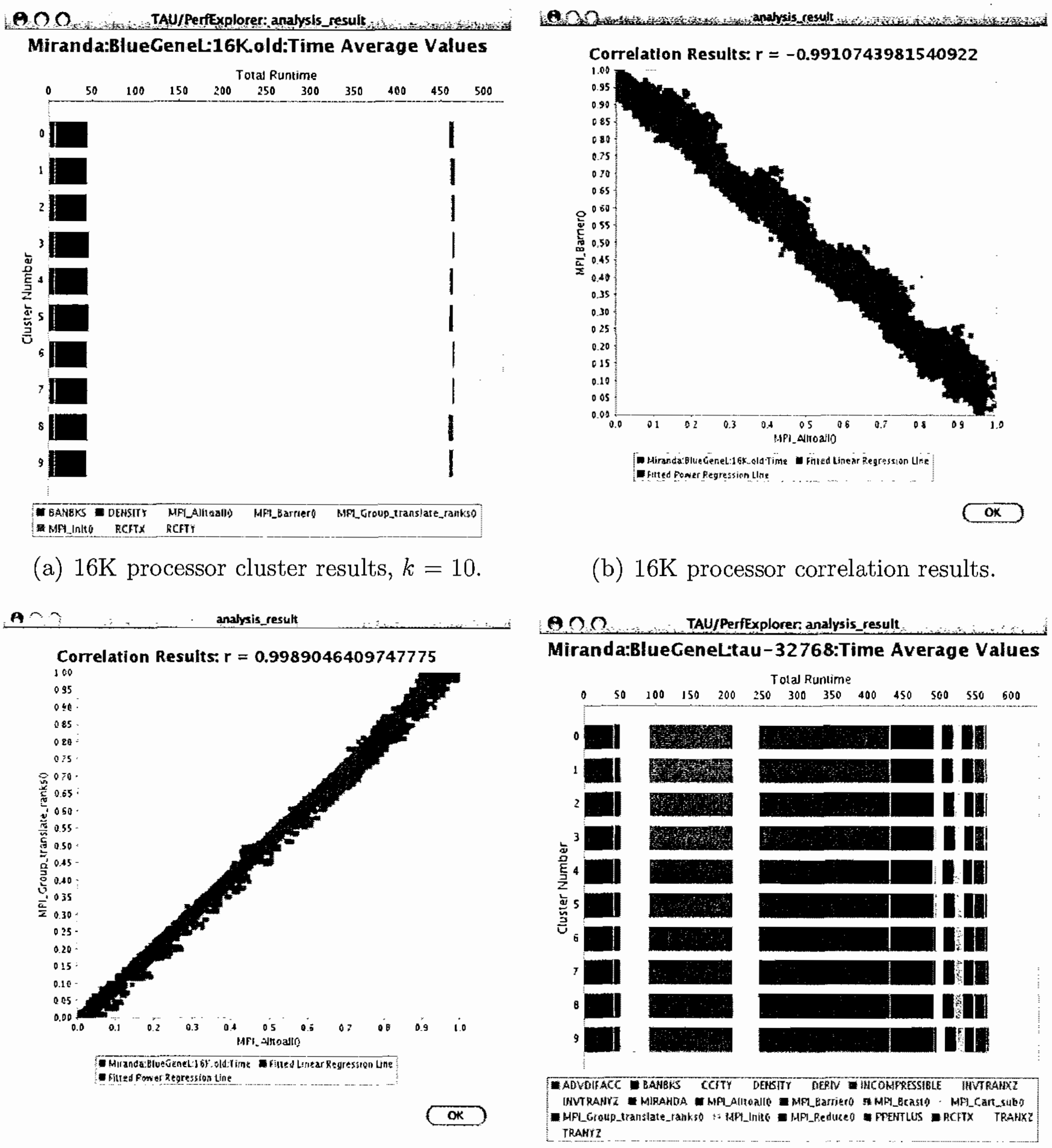
FIGURE 28: Miranda clustering results in PerfExplorer. These figures show the relationship between threads of execution when executing Miranda on BG/L.

CPUs, with communication scaling for fixed workload per CPU. TAU was used to instrument and measure the Miranda application on BG/L. Experiments were done with 4K, 8K, 16K and 32K processors and the performance profiles were loaded into the PerfDMF database.

Figure 28 shows some of the clustering results for the 8K processor data. What is immediately apparent is the pattern of cluster assignment to virtual topology. Unfortunately, this data was collected before we were collecting metadata with TAU performance data, so we cannot be sure what causes this behavior. This performance pattern is likely related to the physical design of the BG/L system – there are 32 processor chips per node board, and 32 node boards per cabinet. The events that primarily caused the clusters were the `MPI_Barrier()` calls and the `MPI_Alltoall()` calls. For clustering on execution time with $k = 10$, the histogram in Figure 28(a) shows a relatively small variation for cluster membership counts, and the graphic in

Figure 28(b) shows the virtual topology of the 8192 processes organized arbitrarily in 32 rows of 256 processes. The alternating bands of behavior are clearly visible as the viewer progresses from left to right, bottom to top, from processor 1 to processor 8192. Figure 28(c) shows the average behavior for each cluster. Although the clusters are not ordered, with careful viewing it should be obvious that there are two groups of behavior for `MPI_Alltoall()`, and a negative linear relationship between `MPI_Group_translate_ranks()` and `MPI_Barrier()`.

Interestingly, there were no such patterns of 32 in the 16K data, shown in Figure 29(a), due to the fact that the 16K run occurred later than the 8K run, and reflected improvements to both the communication infrastructure and to the code itself. However, both datasets showed the same gradual change from first to last processor, showing a relationship between `MPI_Barrier()`, `MPI_Alltoall()` and `MPI_Group_translate_ranks()` (although the variance in the `MPI_Alltoall()` data is very subtle). Specifically, the Miranda application on BG/L has an inverse linear relationship between the functions `MPI_Barrier()` and `MPI_Group_translate_ranks()`, and a linear relationship between the functions `MPI_Alltoall()` and `MPI_Group_translate_ranks()`. As the MPI rank of the process gets larger, the `MPI_Barrier()` call takes more time, and the `MPI_Group_Translate_Ranks()` and `MPI_Alltoall` calls take less time. Figures 29(b) and 29(c) show the results of PerfExplorer correlation analysis of Miranda for these events. Scripted correlation and rule processing of the limited metadata in the


TAU/PerfExplorer: analysis_result
Miranda:BlueGeneL:16K.old:Time Average Values
Total Runtime
0 50 100 150 200 250 300 350 400 450 500
Cluster Number
0 1 2 3 4 5 6 7 8 9
BANBKS DENSITY MPI_Alltoall() MPI_Barrier() MPI_Group_translate_ranks()
MPI_Init() RCFTX RCFTY
analysis_result
Correlation Results: r = -0.9910743981540922
MPI_Barrier()
MPI_Alltoall()
1.00 0.95 0.90 0.85 0.80 0.75 0.70 0.65 0.60 0.55 0.50 0.45 0.40 0.35 0.30 0.25 0.20 0.15 0.10 0.05 0.00
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
Miranda:BlueGeneL:16K.old:Time Fitted Linear Regression Line
Fitted Power Regression Line
OK
analysis_result
Correlation Results: r = 0.9989046409747775
MPI_Group_translate_ranks()
MPI_Alltoall()
1.00 0.95 0.90 0.85 0.80 0.75 0.70 0.65 0.60 0.55 0.50 0.45 0.40 0.35 0.30 0.25 0.20 0.15 0.10 0.05 0.00
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
Miranda:BlueGeneL:16K.old:Time Fitted Linear Regression Line
Fitted Power Regression Line
OK
TAU/PerfExplorer: analysis_result
Miranda:BlueGeneL:tau-32768:Time Average Values
Total Runtime
0 50 100 150 200 250 300 350 400 450 500 550 600
Cluster Number
0 1 2 3 4 5 6 7 8 9
ADVDIFACC BANBKS CCFTY DENSITY DERIV INCOMPRESSIBLE INVTRANXZ
INVTRANYZ MIRANDA MPI_Alltoall() MPI_Barrier() MPI_Bcast() MPI_Cart_sub()
MPI_Group_translate_ranks() MPI_Init() MPI_Reduce() PPENTLUS RCFTX TRANXZ
TRANYZ

(a) 16K processor cluster results, $k = 10$.

(b) 16K processor correlation results.

(c) 16K processor correlation results.

(d) 32K processor cluster results.

FIGURE 29: Miranda 16K processor cluster and correlation analysis.

performance data detected a correlation between the MPI rank of a process, and the time spent in two of the events, as shown in Figure 30.

```
--------------- JPython test script start ------------
doing single trial correlation analysis for Miranda on BGL
Loading Rules...
loading the data...
Firing rules...

MPI_Group_translate_ranks(): 'Time:EXCLUSIVE' metric is correlated
with the metadata field 'MPI Rank'.
        The correlation is 1.000 (direct).
MPI_Barrier(): 'Time:EXCLUSIVE' metric is inversely correlated with
the metadata field 'MPI Rank'.
        The correlation is -0.910 (high).

...done with rules.
---------------- JPython test script end -------------
```

FIGURE 30: Miranda inference rule processing output. Using processing scripts and rules, correlations between performance data and MPI rank were detected in the 16k performance data.

This behavior in the application is due to the MPI communication implementation on the BG/L architecture, and unofficial reports from LLNL are that IBM continued to improve the performance of the communication architecture. In the later sets of data, the negative linear relationship between `MPI_Barrier()` and `MPI_Group_Translate_Ranks()` has also disappeared, again probably due to improvements in the performance of the machine, as shown in Figure 29(d).

However, in more recent 8K data, subtle relationships still exist in the nearly uniform data, in the form of alternating bands of 32 processes. Figure 31 shows some performance data from the later 8K processor execution of the Miranda application, after additional modifications have been made. What the two isometric views show is the relationships between the four MPI events. This view is constructed by

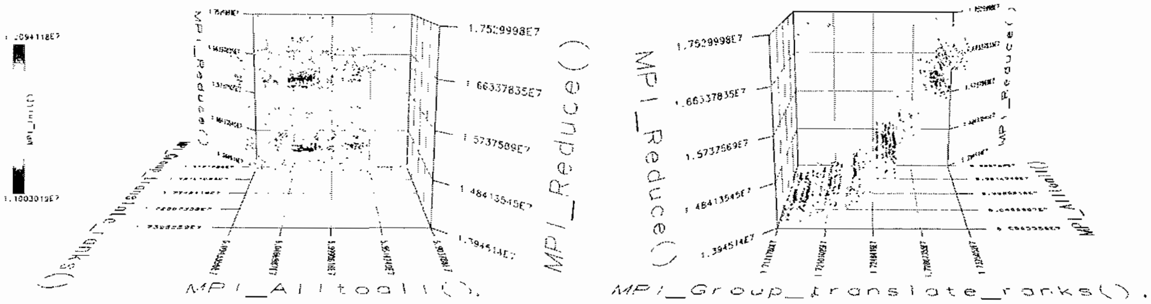


FIGURE 31: Miranda clustering results in PerfExplorer after modifications. These figures show a 4 Dimensional scatterplot of Miranda events after modifications were made to the source code. The events shown were selected because these events have the most variance within their range, weighted by percentage of total execution, across the 8K processors.

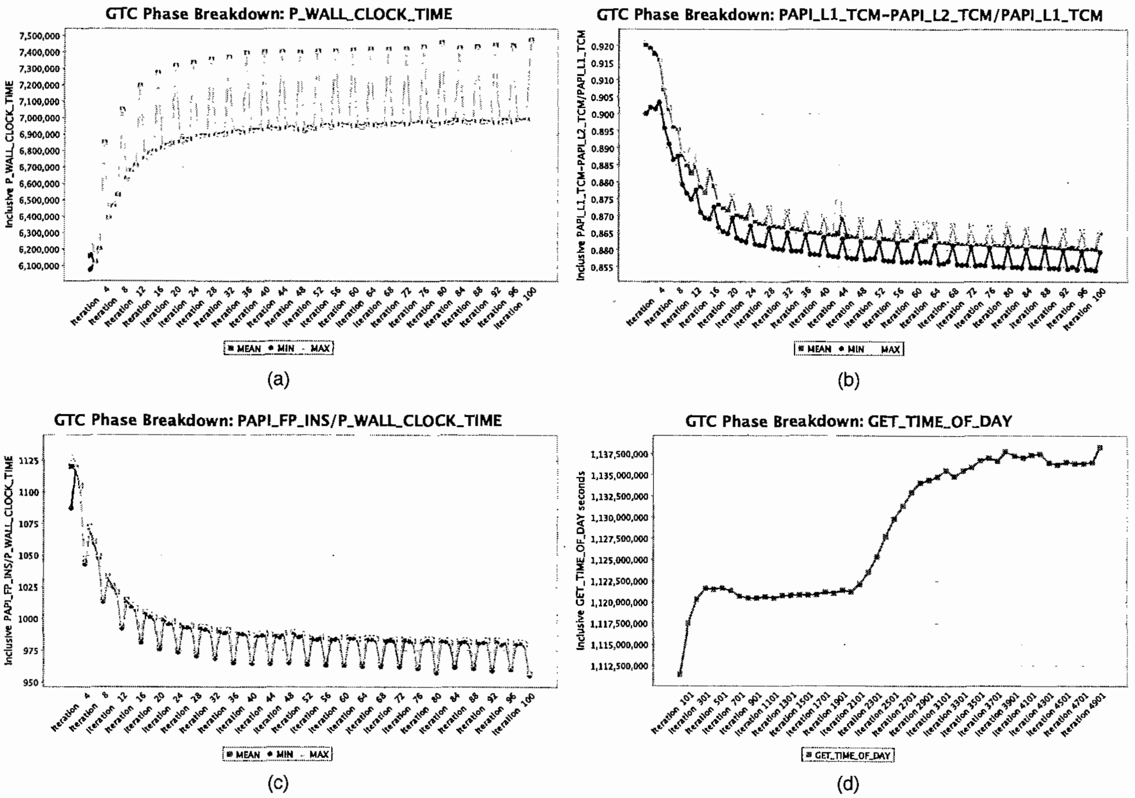
selecting the four events in the data which experience the most variance in their respective ranges, weighted by percentage of total execution time, across processes. Because the Miranda application is very regular and balanced, the four events selected happen to be MPI events, and variation in their values is mostly determined by their physical and logical distance from the root MPI process. However, there are other patterns, such as the staircase of eight clusters when examining `MPI_Reduce` to `MPI_Group_translate_ranks` reflecting the hardware configuration of eight racks of 1K processors.

GTC

The Gyrokinetic Toroidal Code (GTC)[30] is a particle-in-cell physics simulation which models the turbulence between particles in the high energy plasma of a fusion reactor. Scalability studies of the original large-scale parallel implementation of GTC (there are now a small number of parallel implementations, as the development has fragmented) show that the application scales very well - in fact, it scales at a better than linear rate. However, discussions with the application developers revealed that it had been observed that the application gradually slows down as it executes[151] - each successive iteration of the simulation takes more time than the previous iteration.

In order to measure this behavior, the application was auto-instrumented with TAU, and manual instrumentation was added to the main iteration loop to capture dynamic phase information. The application was executed on 64 processors of the Cray XT3/XT4 system at ORNL for 100 iterations, and the performance data was loaded into PerfDMF. A simple analysis script was constructed in order to examine the dynamic phases in the execution. The script was used to load the performance data, extract the dynamic phases from the profile, calculate derived metrics (L1 and L2 cache hit ratios, FLOPs), calculate basic statistics for each phase, and graph the resulting data as a time series showing average, minimum and maximum values for each iteration.

As shown in Figure 32, during a 100 iteration simulation, each successive iteration takes slightly more time than the previous iteration. Over the course of the test simulation, the last iteration takes nearly one second longer than the first iteration. As a minor observation, every fourth iteration results in a significant increase in execution time. Hardware counters revealed that the L2 cache hit-to-access ratio decreases from 0.92 to 0.86 (L1 cache hit-to-access ratios also decrease, but to a



(a)

(b)

(c)

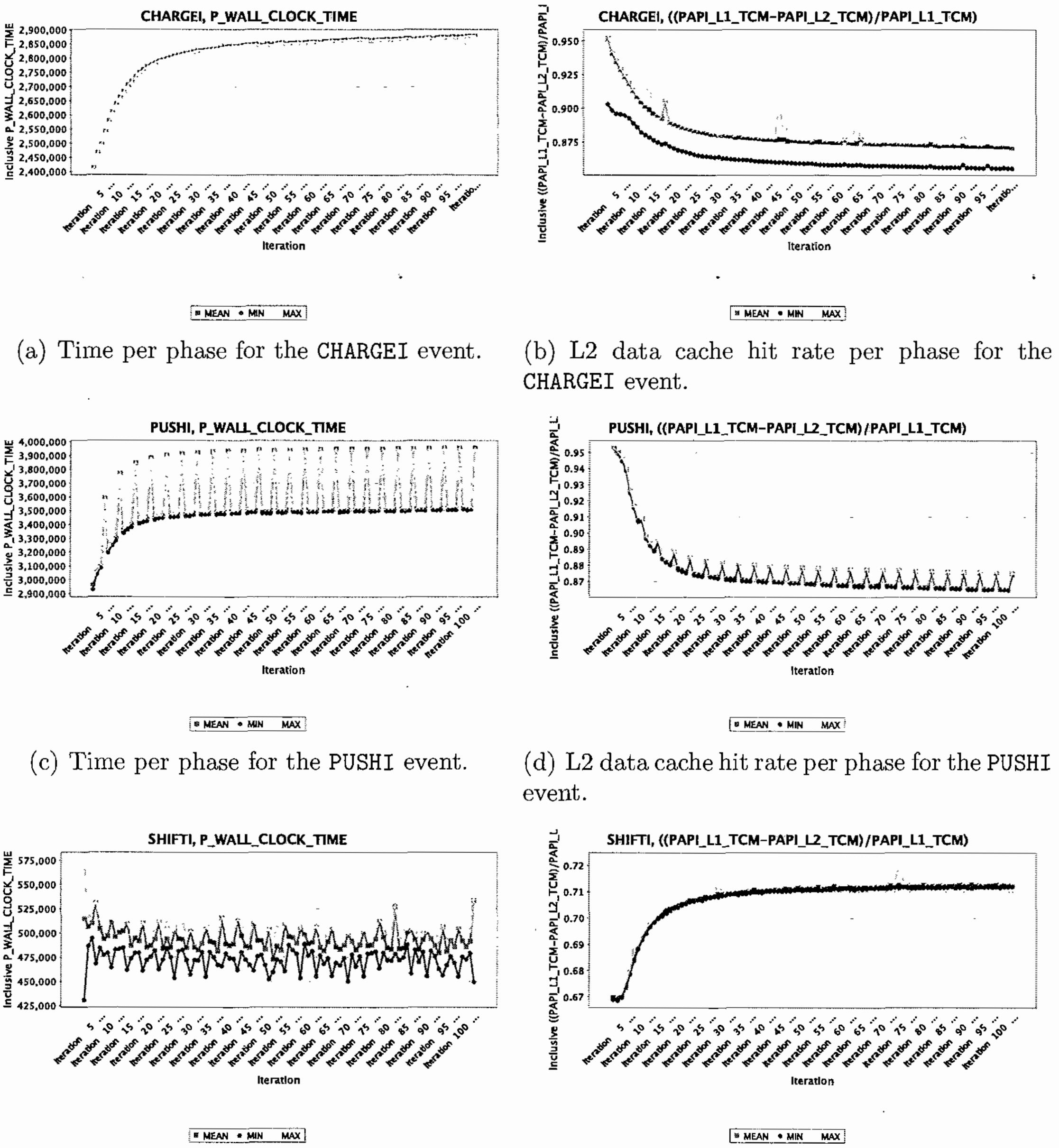
(d)

FIGURE 32: GTC phase analysis. (a) shows the increase in runtime for each successive iteration, over 100 iterations. (b) shows the decrease in L2 hit ratio, from 0.92 to 0.86, and (c) shows the decrease in GFLOPs from 1.120 to 0.979. (d) shows the larger trend when GTC is run for 5000 iterations, each data point representing an aggregation of 100 iterations.

lesser extent). Subsequently, the GFLOPs per processor rate decreases from 1.120 to 0.979. Further analysis of the routines called from the main loop show that the decrease in execution is limited to two routines, CHARGEI and PUSHI. In the CHARGEI routine, each particle in a region of the physical subdomain applies a charge to up to four cells, and in the PUSHI routines, the particle locations are updated by the respective cells after the forces are calculated. The increase in time every fourth iteration is due to a diagnostic call, which happens every ndiag iterations, an input parameter captured as metadata.

A second script for this problem was also developed, which loaded the performance data, extracted the top ten time consuming code regions, derived the L1, L2 and FLOPs metrics, and then compared each code region to the overall performance of the application. An inference rule was constructed which identified code regions which had lower than average cache hit ratios. The combination of this script and rule identified the same code functions, CHARGEI and PUSHI, as having poor cache behavior. The phase behavior for those routines is shown in Figure 33. The time per phase for both of the events, as shown in Figures 33(a) and 33(c). We also see that the L2 data cache hit rate in Figures 33(b) and 33(d) is dropping for both events, indicating a memory usage problem. The sawtooth pattern in the PUSHI figures are the result of a auto-restart process which writes the application process to disk every four time steps. This operation only happens during the PUSHI event. It is also interesting to note the cache behavior and performance of another event, SHIFTI, which is another time consuming event in the application, but it actually shows good cache reuse, which increases as the application runs, as shown in Figures 33(e) and 33(f). Its timing is relatively stable over the course of the application run.

Discussions with other performance analysis experts on the project revealed that the CHARGEI and PUSHI routines have good spatial locality when referencing the particles, however over time, they have poor temporal locality when referencing the



(a) Time per phase for the CHARGEI event.

(b) L2 data cache hit rate per phase for the CHARGEI event.

(c) Time per phase for the PUSHI event.

(d) L2 data cache hit rate per phase for the PUSHI event.

(e) Time per phase for the SHIFTI event.

(f) L2 data cache hit rate per phase for the SHIFTI event.

FIGURE 33: GTC phase analysis, broken down by event.

grid cells. As a result, access to the grid cells becomes random over time. Further analysis is necessary to determine whether the expense of re-ordering the particles at the beginning of an iteration could be amortized over a number of iterations, and whether this added cost would yield a benefit in the execution time. While it appears that the performance degradation levels out after roughly 30 iterations, it should be pointed out that a full run of this simulation is at least 10,000 iterations. GTC was executed on Ocracoke, a BlueGene/L machine at RENCI, for 5,000 iterations. As the results show in Figure 32 (d), the performance continues to degrade. Assuming a 10,000 iteration execution would take an estimated 20 hours to complete on the Cray XT3/XT4, potentially 2.5 hours of computation time *per processor* could be saved by improving the cache hit ratios. This represents a significant savings in time and computational expense.

S3D

S3D[14] is a multi-institution collaborative effort to develop a terascale parallel implementation of a turbulent reacting flow solver. S3D uses direct numerical simulation (DNS) to model combustion science which produces high-fidelity observations of the micro-physics found in turbulent reacting flows as well as the reduced model descriptions needed in macro-scale simulations of engineering-level systems. The examples described here were run on Jaguar[97], the hybrid Cray XT3/XT4 system at Oak Ridge National Laboratory (ORNL).

During scalability tests (from 1 to 12,000 processors) of S3D instrumented with TAU, it was observed that as the number of processors exceeded 1728, the amount of time spent in communication began to grow significantly, and `MPI_Wait()` in

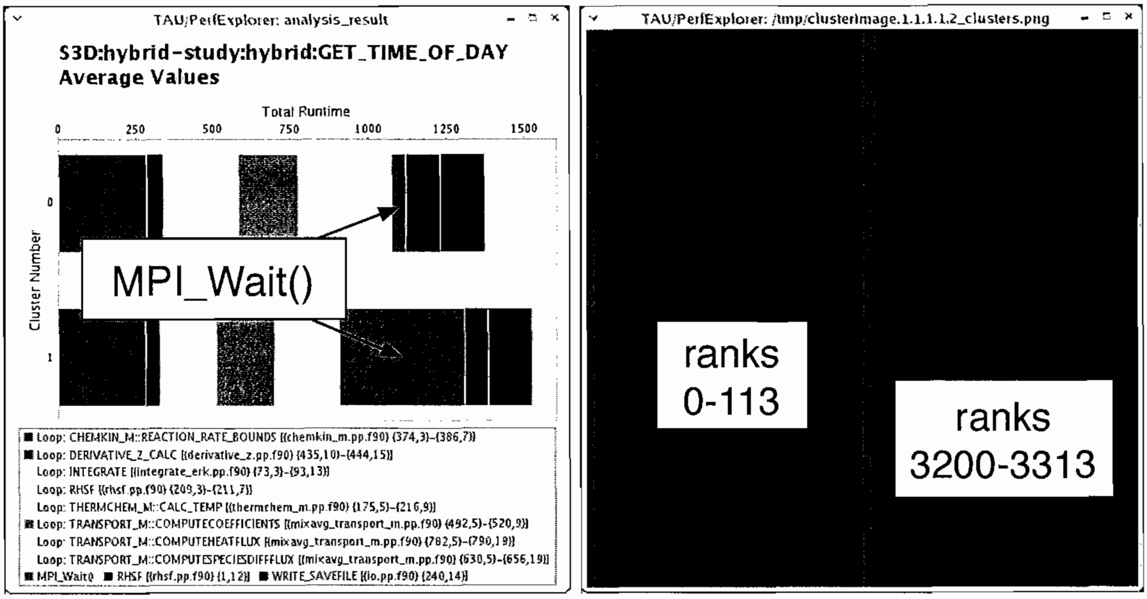

TAU/PerfExplorer: analysis_result
S3D:hybrid-study:hybrid:GET_TIME_OF_DAY
Average Values
Total Runtime
0
250
500
750
1000
1250
1500
0
1
Cluster Number
MPI_Wait()
■ Loop: CHEMKIN_M::REACTION_RATE_BOUNDS [{chemkin_m.pp.f90} {374,3}-{386,7}]
■ Loop: DERIVATIVE_Z_CALC [{derivative_z.pp.f90} {435,10}-{444,15}]
Loop: INTEGRATE [{integrate_erk.pp.f90} {73,3}-{93,13}]
Loop: RHSF [{rhsf.pp.f90} {209,3}-{211,7}]
Loop: THERMCHEM_M::CALC_TEMP [{thermchem_m.pp.f90} {175,5}-{216,9}]
■ Loop: TRANSPORT_M::COMPUTECOEFFICIENTS [{mixavg_transport_m.pp.f90} {492,5}-{520,9}]
Loop: TRANSPORT_M::COMPUTEHEATFLUX [{mixavg_transport_m.pp.f90} {782,5}-{790,19}]
Loop: TRANSPORT_M::COMPUTESPECIESDIFFFLUX [{mixavg_transport_m.pp.f90} {630,5}-{656,19}]
■ MPI_Wait() ■ RHSF [{rhsf.pp.f90} {1,12}] ■ WRITE_SAVEFILE [{io.pp.f90} {240,14}]
TAU/PerfExplorer: /tmp/clusterImage.1.1.1.1.2_clusters.png
ranks 0-113
ranks 3200-3313

FIGURE 34: S3D cluster analysis. The figure on the left shows the difference in (averaged mean) execution behavior between the two clusters of processes. The figure on the right shows a virtual topology of the MPI processes, showing the locations of the clustered processes. the red processes ran on XT3 nodes, and the blue processes ran on XT4 nodes.

particular composed a significant portion of the overall run time (approximately 20%). By clustering the performance data in PerfExplorer, it was observed that there were two natural clusters in the data. The first cluster consisted of a majority of the processes, and these nodes spent less time in main computation loops, but a long time in `MPI_Wait()`. The other cluster of processes spent slightly more time in main computation loops, and far less time in `MPI_Wait()`.

By automatically collecting the MPI host names with the TAU metadata collection, we were able to determine, at runtime, the names of the nodes on which the processes ran. The node IDs were stored in the metadata with the performance data. In the case of a 6400 process run, as shown in Figure 34, there were again two clusters, with 228 processes in one cluster having very low `MPI_Wait()` times (about 40 seconds), and the remainder of the processes in one cluster having very high `MPI_Wait()` times (over 400 seconds). The metadata was then manually correlated with information about the hardware characteristics of each node, identified the slower nodes as XT3 nodes, and the faster nodes as XT4 nodes. As described on page 143, there are two primary differences between the XT3 and XT4 partitions. The XT3 nodes have slower DDR-400 memory (5986 MB/s) than the XT4 nodes' DDR2-667 memory (7147 MB/s), and the XT3 partition has a slower interconnection network (1109 MB/s v. 2022 MB/s). Because the application is memory intensive, the slower memory modules have a greater effect on the overall runtime, causing the XT3 nodes to take longer to process, and subsequently causing the XT4 nodes to spend more time in `MPI_Wait()`.

In order to remove this last manual step to correlating application performance with hardware characteristics, we needed more information about the nodes than was available from the metadata. By using the `nodeinfo` utility available from the batch system, we were able to collect information about each node in the allocation, including the memory speed and interconnect speed, which directly identify the XT3 and XT4 nodes in the full machine. Using a python script, the `nodeinfo` data was formatted as XML, and loaded with the performance data using the third method outlined on page 60. A PerfExplorer script was written which loaded the trial data, extracted the five most time consuming code regions and correlated the event performance with the metadata fields for each thread of execution. An inference rule was used to identify the code regions which had an effectively inverse correlation between run time and both memory speed and interconnect speed, similar to the analysis of Sweep3D on page 141.

Running S3D on an XT4-only configuration yielded roughly a 12% time to solution reduction over the hybrid configuration, primarily by reducing the time spent in computation on slower XT3 nodes, which correspondingly reduced `MPI_Wait()` times from an average of 390 seconds down to 104 seconds. If this application is to be run on a heterogeneous configuration of this machine or on another heterogeneous machine, load balancing should be integrated which takes into consideration the computational and memory capacity of each respective node. The use of metadata would also be important for this optimization.

OpenUH - MSAP

Molecular biologists frequently compute multiple sequence alignment (MSA) to compare protein sequences with unknown functionality to a set of known sequences to detect functional similarities[158]. The steady growth in the size of sequence databases means that the comparisons require increasingly significant scan times. Because the time and space complexities for MSA are in the order of the product of the lengths of the sequences, many heuristic alignment methods have been developed. Among them, progressive alignment[37] is a widely used heuristic. The popular MSA program ClustalW[135] is an application which uses this heuristic. It consists of three stages. First, all pairs of sequences are pairwise aligned to each other in order to calculate a distance matrix measuring the difference between each pair of sequences. Then, a guide tree is calculated from the distance matrix, using a Neighbor-Joining method. Finally, the sequences are progressively aligned according to the branching order in the guide tree. The main purpose of MSA is to infer homology (shared ancestry) between sequences.

Profiling of the ClustalW program on a single processor showed that almost 90% of the run time is spent in the first stage, computing the distance matrix [26]. This first stage is based on the Smith-Waterman algorithm [121], a dynamic programming approach that computes the optimal local alignment of two sequences. Full details of the MSA stages and the SW algorithm can be obtained from [135] and elsewhere. A team from the University of Houston and Nanyang Technological University parallelized the SW algorithm using OpenMP for the main computational loops but did not get a solution that scaled for large numbers of threads. For 16 threads, the speedup was only 11.078, which is a relative scaling efficiency of only 69.2%. The OpenMP compiler used for this effort was OpenUH.

The OpenUH compiler[79] is an open source, portable OpenMP research compiler for C/C++ and Fortran, developed primarily at the University of Houston. OpenMP is a specification for a number of compiler directives which define a high level multithreaded programming interface for parallel computing on Symmetric

Multi-Processor and Multi-core computing architectures. Compilers which support OpenMP interpret the compiler directives, and generate object code for parallel execution. The OpenUH compiler combines an optimizing infrastructure with both object code generation for x86, Opteron, and Itanium platforms and source-to-source output for portability to other platforms.

To improve OpenMP performance, we used `schedule` clauses to specify how the iterations of the main loop should be allocated to the threads. Among the different scheduling mechanisms, we applied static and dynamic scheduling on different protein sequences and varied dynamic chunk sizes to drill down to a suitable scheduling strategy that scaled. Chunk sizes define the size of the task assigned to each thread per iteration. In the process, we found that *static even assignment* (the default), and *dynamic even assignment* scheduling experienced load imbalances. Although even tasks were assigned to each thread, the work within each task varied (due to early termination from some tasks), and an uneven work load was distributed to the processing units, as shown in Figure 35. Varying the chunk sizes assigned to each thread, we found that the imbalance was due to uneven distribution of work. Because the parallel loops in the algorithm were at a very coarse grained level containing four nested loops, small chunk sizes gave the best speedup. Larger dynamic chunk sizes tend to change the scheduling behavior to be more like the static even behavior. When applying the right dynamic scheduling, the load imbalances were reduced and it produced a speedup of 14.877, which is a scaling efficiency of 93% with 16 threads on a

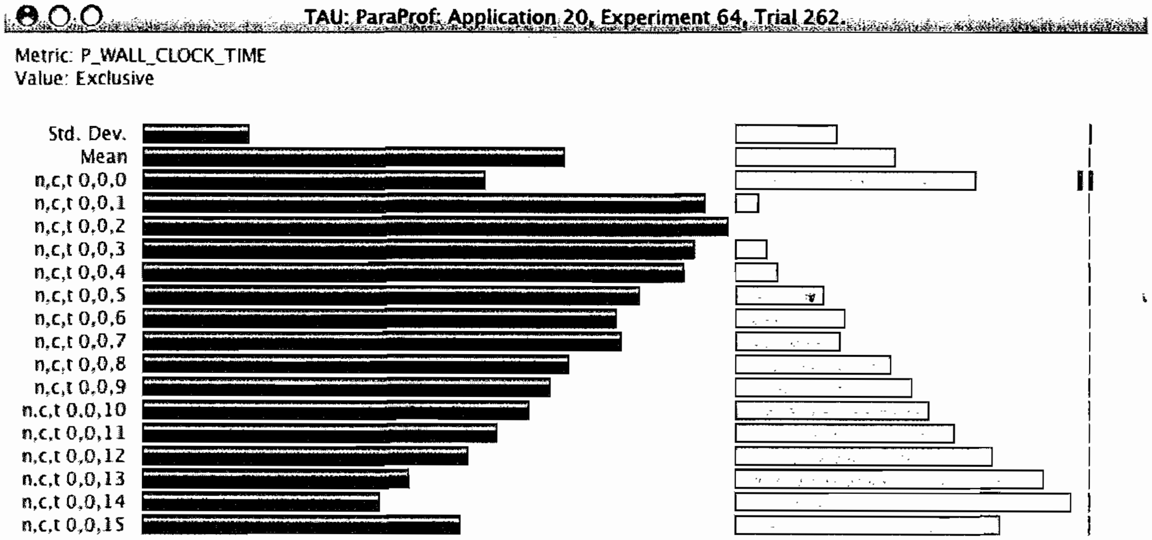


FIGURE 35: MSAP scaling behavior for 400 sequence problem set, showing load imbalance in inner loop (left bars) and outer loop (right bars) when using 16 threads.

400 sequence set when using a chunk size of one. Figure 36 shows the scaling efficiency of different schedules and chunk sizes. A larger test showed a scaling efficiency of up to 80% with 128 threads on a 1000 sequence set.

In order to capture this analysis with PerfExplorer, we developed a script which performed a load balancing test of the code. The pseudocode for the script and the rules for this example are shown in Figure 37. For each instrumented region, or event, we computed the mean and standard deviation of time across all threads, and then computed the ratio of the standard deviation to the mean. Because the outer loop was waiting for the inner loop to complete, we also wanted to detect that for a parent-child relationship in the callgraph, an increase in the time spent in the inner loop meant a shorter time spent in the outer loop. That was done by correlating the

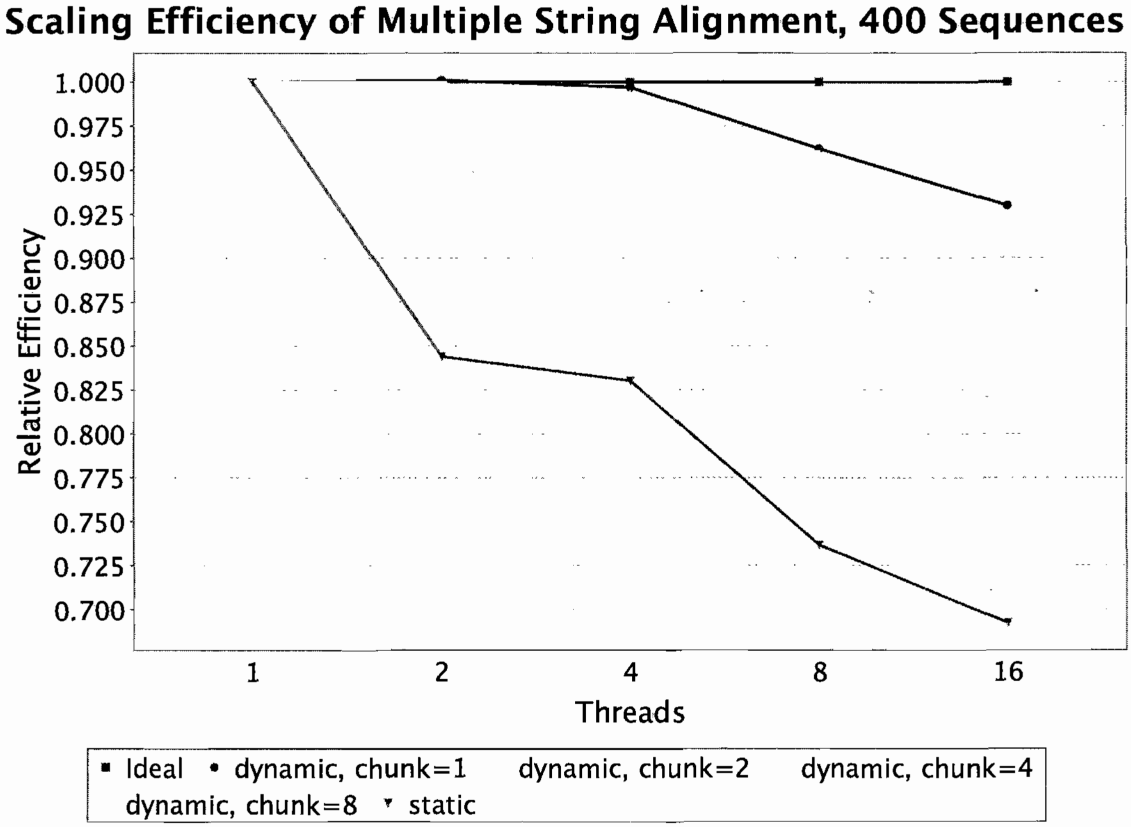


FIGURE 36: MSAP scaling efficiency for 400 sequence problem set, showing the relative efficiency for various schedules and chunk sizes. Scaling efficiency of 1.0 is ideal.

times spent in the loops and getting a high negative correlation. We also wanted to compute the amount of useful work done in the outer loop, which would indicate if threads were working or were waiting at the barrier for other threads to finish.

The load imbalance detection rule, shown in Figure 38, is activated when the following facts are true. First, two loops have a high standard deviation to mean ratio (> 0.25), which indicates that they are unbalanced across the threads. Second, the loops occupy more than 5% of the total runtime, which indicates the severity that

```
for each instrumented region:
  1. compute mean, stddev across all threads
  2. compute, assert stddev/mean ratio (ratio)
  3. correlate region against all other regions
  4. assert correlation (correlation)
  5. assert callpath information (calls)
  6. assert exclusive time of event (severity)

Rule1: IF severity(r) > 0.05 AND ratio(r) > 0.25
  THEN alert(load imbalance: r1) AND assert imbalanced(r)
Rule2: IF imbalanced(r1) AND imbalanced(r2) AND calls (r1,r2)
  AND correlation(r1,r2) < -0.85
  THEN alert(new schedule suggested: r1, r2)
```

FIGURE 37: Script pseudocode and associated rule pseudocode for load imbalance detection.

```
rule "Load Imbalance"
    when
        f : MeanEventFact (m : metric,
            b : betterWorse == MeanEventFact.HIGHER,
            s : severity > 0.05, e : eventName,
            a : mainValue > 0.05, v : eventValue > 0.25,
            factType == "Load Imbalance" )
    then
        System.out.println("The event " + e +
            " has a high load imbalance for metric " + m);
        System.out.println("\tMean/Stddev ratio: " + a +
            ", Stddev actual: " + v);
        System.out.println("\tPercentage of total runtime: " +
            f.getPercentage() + "\n");
        assert(new FactWrapper("Imbalanced Event", e, null));
end
```

FIGURE 38: First rule for OpenMP load imbalance detection.

this load imbalance has on the runtime. When those facts are asserted true, the rule fires, and asserts a new fact, that an unbalanced event exists.

```
rule "New Schedule Suggested"
    when
        f1 : FactWrapper (
            factName == "Imbalanced Event", e1 : factType )
        f2 : FactWrapper (
            factName == "Imbalanced Event", e2 : factType != e1 )
        f3 : FactWrapper (
            factName == "Callpath name/value", e3 : factType )
        f4 : FactData ( t : type == CorrelationResult.CORRELATION,
            e4 : event == e1, e5 : event2 == e2, v : value <= -0.85 )
        eval ( e3.equals( e1 + " => " + e2) )
    then
        System.out.println(e1 + " calls " + e2 +
          ", and they are both showing signs of load imbalance.");
        System.out.println("If these events are in an OpenMP " +
          "parallel region, consider methods to balance the workload," +
          " including dynamic instead of static work assignment.\n");
end
```

FIGURE 39: Second rule for OpenMP load imbalance detection.

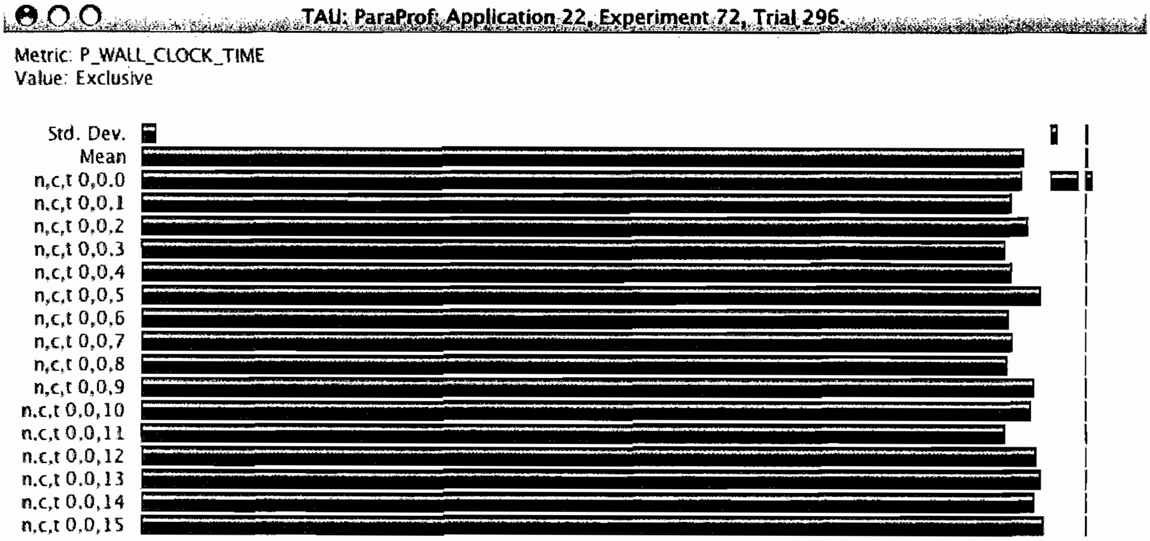
The second rule, shown in Figure 39, will fire when two negatively correlated events are both experiencing a load imbalance. The rule first checks for two events that have been identified as load imbalanced. Secondly, the rule checks if the events are nested - that is, one of the events calls the other in the call graph. Finally, the rule checks if the times in the events are highly negatively correlated - that is, a thread that finishes the inner loop early will spend more time in the outer loop waiting at the barrier, whereas a thread which spends more time in the inner loop will spend less time in the barrier. When all these facts are asserted true, the rule will fire and the user will be indicated of the problem, and a suggested scheduling change.

```
The event LOOP #3 [file:/mnt/netapp/home1/khuck/openuh/src/fpga/
msap.c <63, 163>] has a high load imbalance for metric
P_WALL_CLOCK_TIME
    Mean/Stddev ratio: 0.667, Stddev actual: 6636425.1875
    Percentage of total runtime: 27.15%

The event LOOP #2 [file:/mnt/netapp/home1/khuck/openuh/src/fpga/
msap.c <65, 158>] has a high load imbalance for metric
P_WALL_CLOCK_TIME
    Mean/Stddev ratio: 0.260, Stddev actual: 1.74530281875E7
    Percentage of total runtime: 71.40%

LOOP #3 [file:/mnt/netapp/home1/khuck/openuh/src/fpga/msap.c <63,
163>] calls LOOP #2 [file:/mnt/netapp/home1/khuck/openuh/src/fpga/
msap.c <65, 158>], and they are both showing signs of load
imbalance.
    If these events are in an OpenMP parallel region, consider
    methods to balance the workload, such as dynamic instead of
    static work assignment.
```

FIGURE 40: Rule base output from the MSAP example.

Sample output from running the script and the rules on the MSAP example is shown in Figure 40.

This multiple sequence alignment example shows the ability of well constructed rules to help recognize performance behavior patterns, and suggest a possible diagnosis. Running the analysis on the optimized application does not result in any detection of a load imbalance, as shown in Figure 41. An inexperienced OpenMP programmer would easily benefit from this analysis, as the default parallelization pragma would result in less than optimal scaling. By changing the parallelization from "divide and conquer" (the default static scheduling) to "bag of tasks" (dynamic scheduling with small chunk size), the application will achieve higher parallelization,



Metric: P_WALL_CLOCK_TIME
Value: Exclusive

FIGURE 41: MSAP scaling behavior for 400 sequence problem set, showing balanced workload in inner loop when using 16 threads (the outer loop no longer consumes significant time).

and shorter time to solution. This is the type of expert analysis and code modification we hope to continue to develop in the inference rules.

OpenUH - GenIDLEST

Generalized Incompressible Direct and Large-Eddy Simulations of Turbulence (GenIDLEST)[128] solves the incompressible Navier-Stokes and energy equations and is a comprehensive and powerful simulation code with two-phase dispersed flow modeling capability, turbulence modeling capabilities, and boundary conditions to make it applicable to a wide range of real world problems. It uses an overlapping multi-block body-fitted structured mesh topology in each block combining it with an unstructured inter-block topology. The multi-block approach provides a basic framework for parallelization, which can be exploited by Single Program, Multiple Data (SPMD) parallelism using MPI, OpenMP or a hybrid of the two. Computing a representative problem with n computational blocks, it can use up to n MPI processors or equivalently n OpenMP threads or various combinations of MPI-OpenMP without loss of generality. Further, within each block, "virtual cache blocks" are used. The "virtual" blocks are not explicitly reflected in the data structure but are used in two-level Additive or Multiplicative Schwarz preconditioners for solving linear systems. In addition to the favorable preconditioning properties, the small "cache" blocks also allow efficient use of cache on hierarchical memory systems in modern chip architectures[150]. The virtual cache blocks also provide an additional level of parallelism.

Two test cases which investigate the internal cooling of turbine blades are presented here: a fully-developed flow in a 45-degree ribbed internal cooling duct using Detached Eddy-Simulations (45rib); and another case with the same geometry but with a 90 degree rib and using the method of Large-Eddy Simulations (90rib). The former has a grid consisting of 128x80x64 decomposed into 8 blocks of 128x80x8 and the latter has a grid of 128x128x128 decomposed into 32 blocks of 128x128x4. The two cases are executed using both MPI and OpenMP on up to 8 and up to 32 processors of the SGI Altix, for the 45 and 90 degree problems, respectively.

In the code framework each computational block has ghost cells at inter-block boundaries and also at periodic boundaries which are used in the flow direction. Ghost cell updates on each processor employ asynchronous MPI communications and involve two additional temporary buffers that enable some overlapping of the `MPI_Isend()` and `MPI_Ireceive()` operations for greater efficiency. In the 45rib and 90rib examples on 8 and 16 MPI processors respectively, two `MPI_Isend()` / `MPI_Ireceives()` are invoked on each processor with 2 on-processor copies, noting that these are done in parallel across MPI processes. However, when using standalone OpenMP (with 8 threads for 45rib and 16 or 32 threads for 90rib), all boundary updates are copies in shared memory initiated by the master thread. Hence there are 30 on-processor copies for 45rib and 126 on-processor copies for 90rib, all initiated by the master thread.

Our methodology is part of an application tuning cycle that consists of iterative application runs which enable scalable instrumentation and feedback optimizations, as follows:

Profiling with Selective Instrumentation

Our selective instrumentation method [46] is designed to create a scoring mechanism for regions of interest based on their importance in the code and call graph. We want to avoid instrumenting regions of code that have small weights (e.g. few basic blocks, statements) and are invoked many times. In this run we focus on procedure level instrumentation. The goal of the initial run is to determine where the processor bottlenecks are located. Depending on whether the application is integer or floating-point based, we select: Wall Clock Time, Total Cycles (equivalent), Total Stall Cycles and either number of floating point or integer instructions. The formula to calculate the inefficiency for this purpose is:

$$Inefficiency = Floating\_Point\_Operations * (Total\_Stall\_Cycles/Total\_Cycles)$$

This formula is calculated using PerfExplorer scripts for each region being measured. The regions with the highest inefficiency are the regions that the programmer and compiler should focus on optimizing.

Collection of in-depth performance information for the inefficient regions in profiling mode

In this run we do not turn on all the instrumentation, but only instrument specific code regions or procedures of interest, collecting more fine-grain information. This includes instrumenting loops, branches, calls, and possibly individual statements. During this run we collect hardware counters to perform the processor bottleneck analysis. The general formula we have adopted for this purpose is the following based on Jarp [61]:

$$Total\_Stall\_Cycles = L1D\_Cache\_Misses + Branch\_Misprediction + \\ Instruction\_Misses + Stack\_Engine\_stalls + Floating\_Point\_Stalls + \\ Pipeline\_Inter\_Register\_Dependencies + Processor\_Frontend\_Flushes$$

We primarily collect performance data for stall cycles from the Level 1 Data Cache Misses and Floating Point Stalls (on the Itanium, the floating-point registers are fed directly from level 2 cache). If 90% of the stalls are due to these two causes, we ignore other sources of stalls in the formula. If that is not the case, we will have to perform additional runs to calculate the other components of the formula. The choice of 90% is a general guideline based on behavior seen in different applications.

Memory Analysis Metrics

In the same way as the second run, we use hardware counters to perform the memory bottleneck analysis based on the following formula:

$$\begin{gathered} Memory\_Stalls = (L2\_data\_references\_L2\_all - L2\_misses) * \\ L2\_Memory\_Latency + (L2\_misses - L3\_missed) * L3\_Memory\_Latency + \\ (L3\_misses - Number\_of\_remote\_memory\_accesses) * Local\_Memory\_Latency + \\ (Number\_of\_remote\_memory\_accesses) * Remote\_memory\_access\_latency + \\ TLB\_misses * TLB\_miss\_penalty \end{gathered}$$

$$\begin{gathered} Remote\_Memory\_Accesses\_Ratio = \\ Number\_of\_remote\_memory\_accesses / L3\_misses \end{gathered}$$

The coefficients in this formula are the different latencies (in cycles) for the different levels of memory for the Itanium 2 processor (Madison), and the interconnection latencies of the SGI NumaLINK 4 for local and remote memory

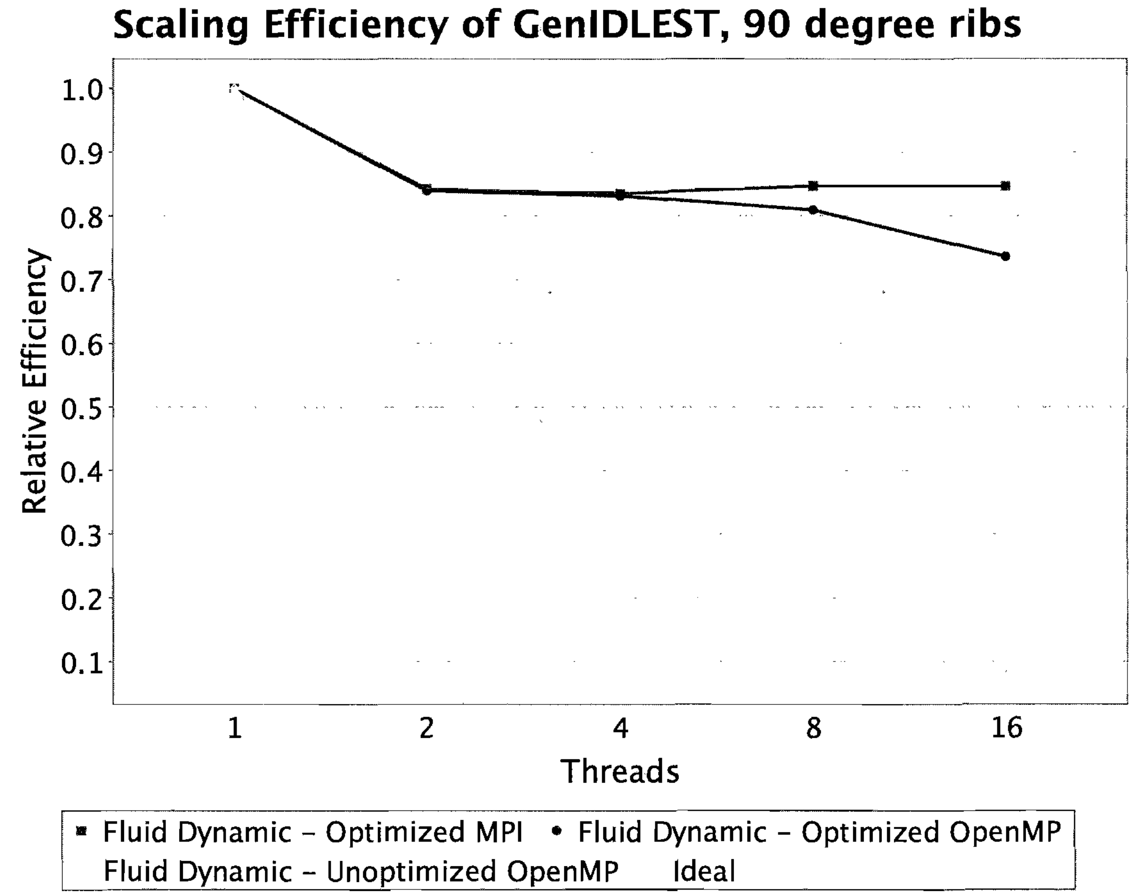


FIGURE 42: Speedup of optimized and unoptimized OpenMP, and optimized MPI.

accesses. The value for remote memory latency accesses is an estimation of the worst-case scenario for a pair of nodes with the maximum number of hops and is system dependent.

In this case study we wanted to understand why the OpenMP implementation of this application does not scale as well when compared to the MPI implementation in the SGI Altix. The OpenMP version lagged by a factor of 11.16 behind its MPI counterpart for the case of 90rib and 3.48 for the 45rib case. The unoptimized OpenMP version of the application does not scale at all as seen in Figure 42.

We constructed PerfExplorer scripts to derive the metrics, and created rules to examine the results. Pseudocode for the scripts and rules described for this example are shown in Figure 43, and sample output from each of the rules is shown in Figure 44. For the first metric, we constructed a script which loaded the data, derived the inefficiency metric, and then a rule searched for events with high inefficiency. We used the script and accompanying rules to examine a 16-thread run of the OpenMP implementation on the 90rib problem. Six procedures with poor scaling were identified with a higher than average stall per cycle rate. We constructed a second script which derived the total stall metric. The rule for the second metric was to look for events which had 90% or more of their stalls caused by floating point stalls or memory stalls. The same six events, plus two more, were identified as having a high percentage of stalls from those two sources. We constructed a third script to examine the causes of the memory stalls. The script was primarily concerned with the numbers of L3 cache misses and the ratio of local memory references to remote memory references.

The performance slowdown is mostly caused by a data locality difference between the MPI and OpenMP versions of the code. This was indicated by higher number of L3 cache misses and latencies in the OpenMP version. Figure 45 shows that the main computation procedures `bicgstab`, `diff_coeff`, `matxvec`, `pc`, `pc_jac_glb` (among others) do not scale. Data locality is important for achieving good performance in the SGI Altix. SGI Altix provides the default *first-touch* policy for placing data, in which a page of memory is allocated/moved to the local memory of the first process

```
for each instrumented region, exclusive:
  1. derive, assert inefficiency metric (inefficiency)
  2. derive, assert memory/total stall cycles metric (tsm)
  3. derive, assert memory cycles metric (memory)
  4. derive, assert local memory accesses ratio metric (local)
  5. assert exclusive time of region (severity)
also compute values for main, inclusive

Rule1: IF severity(r) > 0.02 AND inefficiency(r) > inefficiency(main)
  THEN alert (inefficient, r) AND assert(inefficient(r))
Rule2: IF inefficient(r) AND tsm(r) > 0.9
  THEN alert (memory stalls, r) AND assert (memstall(r))
Rule3: IF memstall(r) AND memory(r) > memory(main)
  THEN alert (memory cycles, r)
Rule4: IF memstall(r) AND local(r) < local(main)
  THEN alert (remote references, r)
```

FIGURE 43: Script pseudocode and associated rule pseudocode for memory inefficiency detection.

```
The event exchange_var__ has a higher than average stall / cycle rate
Average stalls per cycle: 0.79877, Event stalls per cycle: 0.95439
    Percentage of total runtime: 31.16%

The event exchange_var__ has a high percentage of stalls due to L1 data
cache misses and FP Stalls.
    Percent of Stalls due to these two reasons: 99.88%

The event exchange_var__ has a higher than average number of cycles
handling memory references. Consider reviewing this section of code to
reduce memory operations.
    Average memory cycles: 73.72%, Event memory cycles: 100.09%

The event bicgstab_ has a lower than average local memory reference
percentage. If this is an OpenMP parallel region, consider methods for
parallelizing data initialization.
    Average percentage: 93.77%, Event ratio: 90.44%
```

FIGURE 44: Rule base output from the GenIDLEST example.

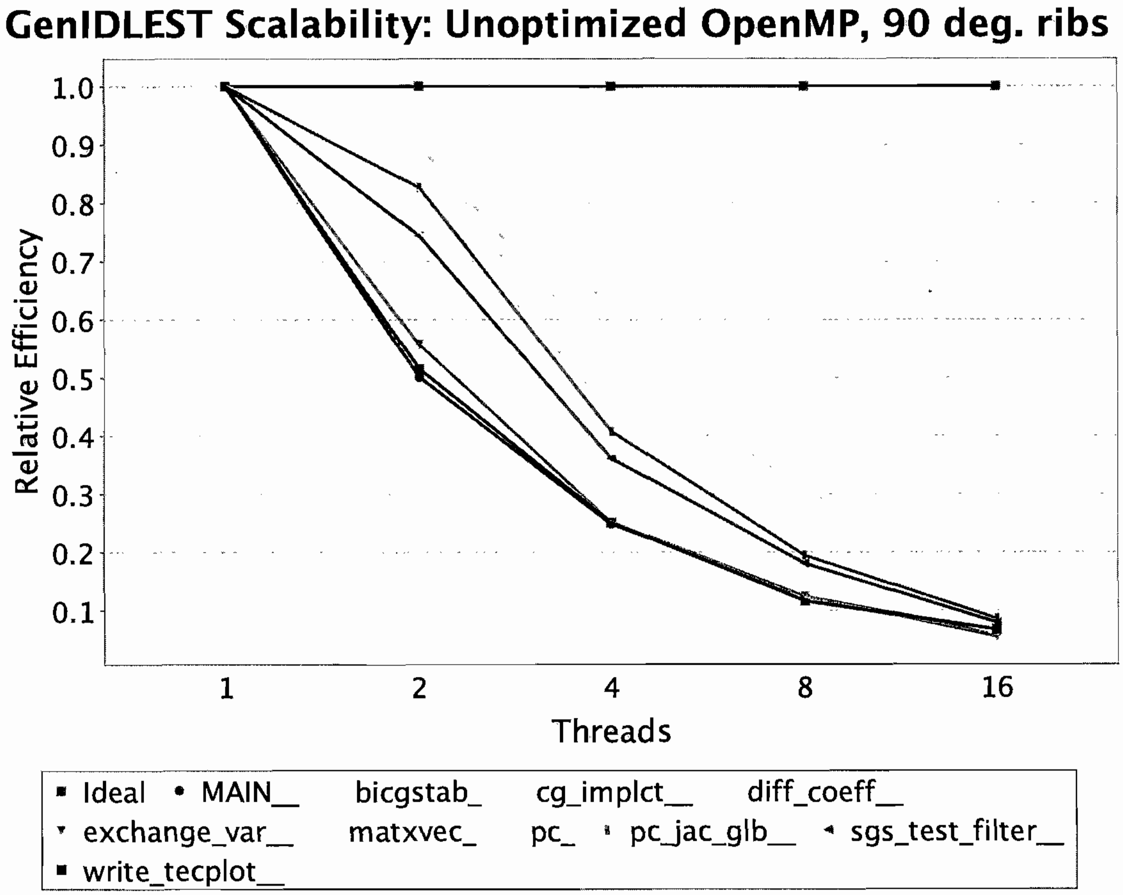


FIGURE 45: GenIDLEST Speedup per event, unoptimized OpenMP. GenIDLEST scaling behavior for 90rib problem, speedup per event for the unoptimized OpenMP implementation.

to access the page. The use of a default first-touch policy has worked very well on a single threaded or MPI processes code on many NUMA platforms, but may lead to poor performance with OpenMP. In MPI all the memory accesses are to local memory by default. OpenMP has the flexibility to use the first-touch policy to place data in the different nodes since the data are not explicitly mapped to processors as with MPI. In addition, OpenMP has a privatization feature where data can be defined as local to each thread.

The final major source of performance degradation is caused by the procedure exchange_var as seen in Figure 45. This procedure is responsible for driving the exchange of data in the ghost cells. Because one of its subroutines (mpi_send_recv_ko) is sequential, it limits the scalability of the application. In the old implementation of the boundary update procedure, which was primarily written for the MPI paradigm, the on-processor copies were done sequentially since most of the work was distributed over MPI processes. However, this became a major bottleneck in the OpenMP paradigm. Four of the events from the previous script were identified as having a lower ratio of local to remote memory references than the application on average. One of these events, exchange_var__, represented 31% of the runtime, and was scaling very poorly, which confirms its sequential nature and its local data.

Since PerfExplorer was able to determine that the main problems in the computational procedures were L3 misses and remote memory accesses (when compared to MPI) we discovered that the application was initializing most of its data sequentially, resulting in data being placed on one node. We fixed all the initializations by parallelizing the initialization loops to make sure we place the data correctly across processors.

To remedy the exchange_var__ problem the on-processor copies were parallelized by eliminating two intermediate steps in the update procedure: that of filling the intermediate send buffer with data to be copied and copying this buffer into an intermediate receive buffer, which are inherently serial operations. In the the

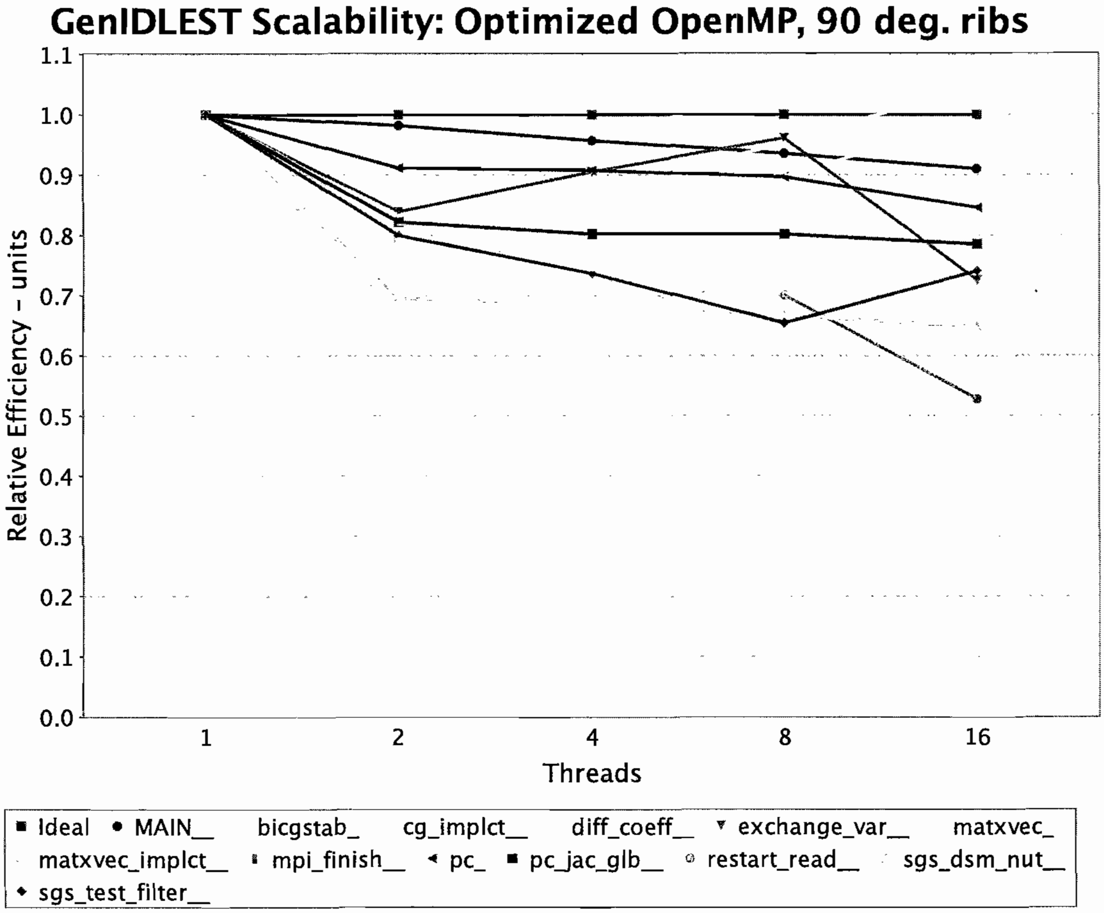


FIGURE 46: Speedup per event, optimized OpenMP. GenIDLEST scaling behavior for 90rib problem, speedup per event for the optimized OpenMP implementation.

optimized version, an OpenMP do parallel loop is applied to the blocks residing on the processor (8 for 45rib and 32 for 90rib) and direct copies are initiated from the send buffer to the destination array.

After optimization, both the MPI and OpenMP baseline performance improved, and the OpenMP implementation scaled nearly as well as MPI, as seen in 42. The performance difference between the MPI and OpenMP implementations become minimal, in the range of 15% for 90rib and 16.8% for 45rib which is a big improvement from the unoptimized version. Figure 46 shows the scaling behavior of the significant events after optimization, as compared to the unoptimized events in Figure 45. The lesson learned here is that we need to provide feedback to the compiler to tell it that it should focus on improving the L3 optimizations by targeting reduction of the cycles predicted in the cache model. We must also feed back information to the inter-procedural array region analyzer to make sure that all the data are initialized and accessed consistently across procedures to improve data locality via the first-touch policy. The feedback presented to the user includes suggestions for the `exchange_var__` procedure.

CQoS - PETSc

The Common Component Architecture Forum (CCA)[101] is a consortium of researchers working to define a standard component architecture for high performance computing. One of the working groups within CCA is the Computational Quality of Service (CQoS) group. The CQoS group is focused on automating runtime component selection to optimize performance and accuracy of components and/or an entire simulation. There are two key aspects to this work. First, components need to be evaluated in parametric study to examine their performance in various contexts, and build predictive performance models. Secondly, at runtime, the runtime environment needs to be captured and used to select the "best" component for the current context. The runtime behavior of the component will also be compared to parametric predictions for component behavior, both in performance and accuracy. If the performance and/or accuracy is not as expected, then the runtime can then swap the component out for a better performing alternative.

There are a number of sub-projects within the CQoS group which are seeking to construct performance databases with CQoS capabilities, and could benefit from the application of PerfExplorer analysis. The individual sub-projects include adaptive linear solver components, components for chemistry simulations such as GAMESS[5], MPQC[20] and NWChem[105], and grid partitioning components. All of the projects have common goals as stated above, to select the appropriate component for a given context, and to ensure that the component performance meets the needs of the particular application. The chemistry work was already described on page 149. In this Section, we will discuss the work relating to adaptive linear solver components.

The application of interest in this discussion is example application which comes with PETSc, the Portable, Extensible Toolkit for Scientific Computation[100]. PETSc is a collective library of data structures and routines for constructing parallel scientific applications. In particular, PETSc specializes in solving partial differential equations. The example application is a modified driven cavity flow simulation. The problem is modeled by a differential equation system in the unit square, which is uniformly discretized in the x and y directions. The application solves the differential system using a non-linear equation solver. The non-linear solver is constructed by selecting a preconditioner and a linear solver.

The linear solver selected is iteratively called by the non-linear solver framework, until convergence or the maximum number of iterations has been reached. There are over fifteen Krylov linear solvers, over eighteen direct linear solvers, and over sixteen preconditioners, and each of them have a number of parameters. The obvious conclusion is that there is considerable confusion about linear solver to use, and which preconditioner. The choices made will determine whether a solution is found or not, and how efficiently. The choice of linear solver and preconditioner are influenced by the properties of the problem, such as the resolution of the solution, the error tolerance, and the properties of the input matrix. A recommender system is clearly required, and attempts have been made at constructing a special purpose non-linear solver recommender system [9, 27]. However, the CQoS project is looking to create a general purpose recommender system for the PETSc library, to be used with all CCA component based applications.

Bhowmick et al. [9] and Eijkhout and Fuentes [27], as described on page 41, have previously constructed task-specific recommender systems for non-linear solvers in PETSc. In this exercise, we were attempting to recreate their results with our general-purpose recommender system, and do it in a slightly different way. First, those authors built a solution which is restricted to non-linear matrix solvers. Our goal is to build a recommender from our general-purpose classifier framework. Second, their method was to construct binary classifiers, which classified the solver / preconditioner combination as "good" or "bad". Our classifier is not restricted

TABLE 12: Parameters for PETSc ex27 application, and values used in data collection to generate data to train the classifier. * indicates an application parameter, all others are PETSc parameters. The `ksptype` parameter is the linear solver type, and the `pc` parameter is the preconditioner type.

Parameter	**Default**	**Values**
lidvelocity*	1/(gridsize)	10, 50, 100
grashof*	1.0	100, 1e.3, 1.e5
gridsize	4x4	16x16, 32x32, 64x64
cflini	50	1.0e-1, 10, 20
srtol	1.0e-8	1.0e-8
krtol	1.03-5	1.0e-5, 1.0e-4
snestype	ls	ls, tr
kspmaxit	10000	200, 400, 600
pcfactorlevel	0	0
ksptype	gmres	fgmres, gmres, cg, bcgs, tfqmr
pctype	ilu	ilu, jacobi, bjacobi, icc, cholesky, sor, asm, none

to binary classification, and returns the recommended solver / preconditioner in a two-step process.

In order to train the classifiers, batch scripts were used to iteratively execute the test program, which was instrumented with TAU to collect performance data and the metadata for each iteration. In order to do this, phase-based profiling from TAU was used, to collect performance data for each iteration of the non-linear solver. Table 12 shows the parameters used for the program, and the various values. Each of these values was saved as a metadata value, as were some properties of the sparse matrix data, including column variability, diagonal average, diagonal sign, diagonal variance, and row variability. Metadata values also include whether the non-linear solver converged or not, and whether each iterative call of the linear solver converged.

The analysis script to process this data started by loading all trials generated by the parametric study, which was 3684 trials. Of these trials, any which failed to converge were discarded, resulting in 1847 valid trials. From those valid iterations, 521 unique tuples of the input metadata parameters were found. The input parameters to the classifier were the grashof, cflini, gridsize, pc, matrixsize, ksprtol, and kspmaxit. The class to be determined by the classifier was the actual linear solver, which had four possible values in the unique tuples: bcgs, tfqmr, gmres and fgmres. The others failed to converge, and had therefore been discarded.

For each unique tuple of the input parameters, the best performing iteration was found, and added to the training data. Once the 521 training individuals were selected, they were passed in to build the classifiers. Four different classifier methods were used, including J48, Naïve Bayes, Support Vector Machine, and Multilayer Perceptron. The Alternating Decision Tree and Random Tree classifiers could not be used because they require a two-class problem. For each classification method, the script requested a 10-fold cross validation test, example results of which are shown in Table 13. The accuracy of the classifier is high, but the kappa statistic for each classifier is not promising - suggesting at best a 39% chance of chosing the right solver method. In the training data, 494 of the 521 training instances are `bcgs`, the remaining are split among the other three types, and those are not consistently classified correctly in the cross-fold tests.

TABLE 13: Solver classifier results, 10-fold cross validation.

Classifier	Accuracy	Kappa
Multi. Perceptron	95.01%	0.3920
J48	95.59%	0.3137
Naïve Bayes	90.02%	0.2354
Support Vector	95.39%	0.2186

TABLE 14: Preconditioner classifier results, 10-fold cross validation.

Classifier	Accuracy	Kappa
Multi. Perceptron	47.77%	0.1127
J48	48.09%	0.1108
Naïve Bayes	49.36%	0.1177
Support Vector	51.27%	0.0861

A second group of classifiers was constructed using the same input data. These classifiers were to recommend the preconditioner to use, given the recommended solver. The same 3684 trials were loaded, and unique tuples were selected from the following list of parameters, grashof, cflini, gridsize, pc, matrixsize, ksprtol, kspmaxit and the recommended linear solver. 314 unique tuples were found, which were used to train four classifiers, using the same four methods. These four classifiers were then validated using 10-fold cross validation, the results of which are shown in Table 14. Unlike the previous group of classifiers, the accuracy is not high, and the kappa statistic for each classifier is also not promising. of choosing the right solver method. In the training data, the preconditioner choice is roughly split between `ilu` and `bjacobi`, although not always correctly.

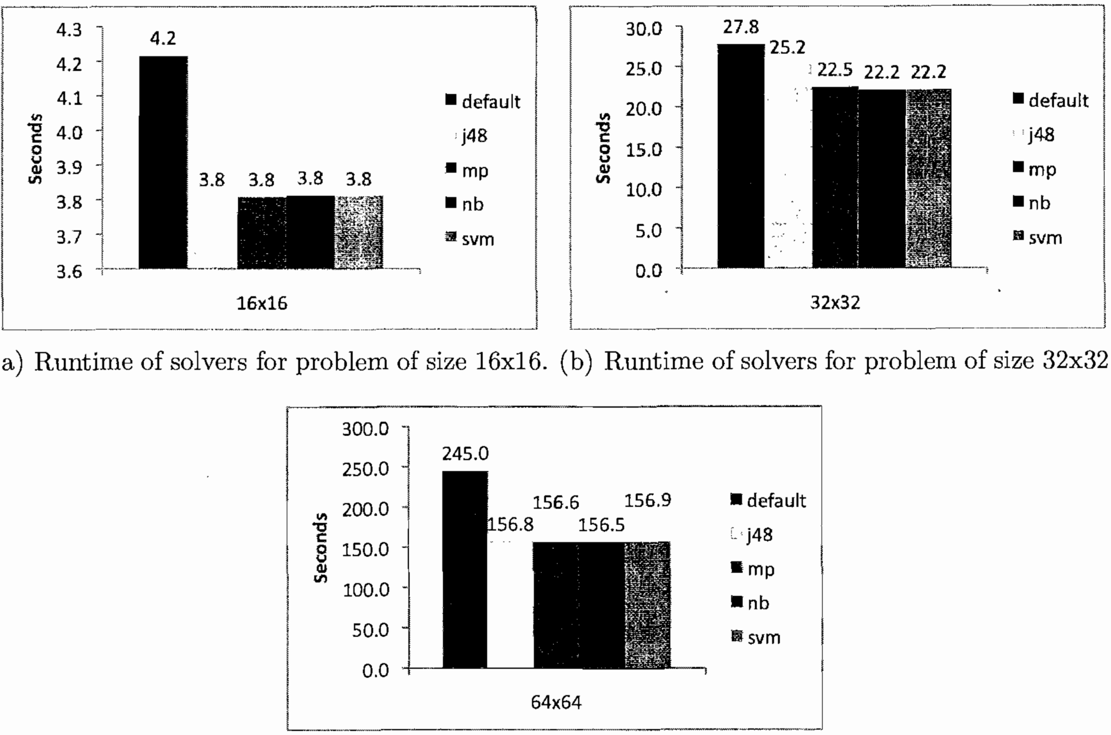
In order to evaluate the runtime usefulness of the classifiers, we constructed an experiment to iterate over 18 test examples, using `lidvelocity` values of 10 and 20, `grashof` values of 100, 500 and 100, and three grid sizes. The default solver

TABLE 15: Classifications for PETSc non-linear solver parameters.

grid size	lidv	gras	J48	Multilayer Perceptron	Naïve Bayes	Support Vector Machine
16x16	10	100	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu
		500	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu
		1000	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu
	20	100	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu
		500	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu
		1000	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu
32x32	10	100	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu
		500	fgmres,bjacobi	bcgs,bjacobi	bcgs,ilu	bcgs,ilu
		1000	fgmres,bjacobi	bcgs,bjacobi	bcgs,ilu	bcgs,ilu
	20	100	bcgs,ilu	bcgs,bjacobi	bcgs,ilu	bcgs,ilu
		500	fgmres,bjacobi	bcgs,bjacobi	bcgs,ilu	bcgs,ilu
		1000	fgmres,bjacobi	bcgs,bjacobi	bcgs,ilu	bcgs,ilu
64x64	10	100	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu
		500	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu
		1000	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu
	20	100	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu
		500	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu
		1000	bcgs,ilu	bcgs,ilu	bcgs,ilu	bcgs,ilu

is `gmres`, and the default preconditioner is `ilu`. For each combination, a solver recommendation was requested from each of the four solver classifiers, and then a preconditioner recommendation was requested from each of the four peconditioner classifiers. The recommendations given are listed in Table 15.

The run times of the default and recommended solvers are shown in Figure 47. Clearly, the recommended solver and preconditioner for each problem size and parameter combination was significantly faster than the default solver, as shown in Tables 16, 17, and 18. This matches the result from the previous work, with one difference. The previous authors consistently chose the `bcgs` and `ilu` combination, whereas our Multilayer Perceptron based recommender found the `bcgs` and `bjacobi`



(a) Runtime of solvers for problem of size 16x16. (b) Runtime of solvers for problem of size 32x32.

(c) Runtime of solvers for problem of size 64x64.

FIGURE 47: Speed improvement of recommended solvers. Most classifiers recommended the `bcgs` solver with the `ilu` preconditioner. The J48 classifier sometimes recommended `fgmres` with `bjacobi` for the 32x32 problem, which was a less optimal selection, but still faster than the default.

combination equally suited for the 32x32 problems. Also, the J48 classifier was the least accurate of the classifiers tested, as it selected a poorer performing solver, `fgmres`, for the 32x32 problem.

Like our work with the GAMESS project, described in section VII, this effort is ongoing. We will continue to improve our classifier with matrix information, and explore how parallelism affects the solver and preconditioner selection. In particular, some solver / preconditioner combinations are not available in parallel mode, such as the `ilu` and `icc` preconditioners. A more complex problem we hope to tackle in the

TABLE 16: Improvement, in seconds, between default solver and recommended solver on the 16x16 problem. Each of the classifiers used recommended the `bcgs` solver with the `ilu` or `bjacobi` preconditioner, so the results are nearly identical.

		Multilayer	**Naïve**	**Support Vector**
	J48	**Perceptron**	**Bayes**	**Machine**
Mean	0.41	0.41	0.40	0.40
95% C.I.	(0.34, 0.48)	(0.34, 0.47)	(0.34, 0.47)	(0.34, 0.47)
St.Dev.	0.09	0.08	0.08	0.08
two-tail p	0.0001	0.0001	0.0001	0.0001

TABLE 17: Improvement, in seconds, between default solver and recommended solver on the 32x32 problem. Three of the classifiers used recommended the `bcgs` solver with the `ilu` preconditioner, but the J48 classifier sometimes chose the less optimal `fgmres` solver with the `bjacobi` preconditioner.

		Multilayer	**Naïve**	**Support Vector**
	J48	**Perceptron**	**Bayes**	**Machine**
Mean	2.56	5.26	5.64	5.63
95% C.I.	(0.84, 4.28)	(4.83, 5.70)	(5.11, 6.16)	(5.18, 6.08)
St.Dev.	2.15	0.54	0.65	0.56
two-tail p	0.0334	< 0.0001	< 0.0001	< 0.0001

TABLE 18: Improvement, in seconds, between default solver and recommended solver on the 64x64 problem. Each of the classifiers used recommended the `bcgs` solver with the `ilu` preconditioner, so the results are nearly identical.

		Multilayer	**Naïve**	**Support Vector**
	J48	**Perceptron**	**Bayes**	**Machine**
Mean	88.24	88.38	88.49	88.13
95% C.I.	(84.32, 92.16)	(84.28, 92.49)	(84.51, 92.46)	(83.55, 92.70)
St.Dev.	4.90	5.13	4.96	5.72
two-tail p	< 0.0001	< 0.0001	< 0.0001	< 0.0001

future is the ability to replace non-converging or poorly performing linear solvers at runtime. We will discuss this more in depth in Chapter X.

Summary

The applications case studies discussed in this chapter demonstrate the benefits of the research contributions and their realization in the PerfDMF and PerfExplorer frameworks. The frameworks continue to be extended, and new capabilities added as necessary. In the next and final chapter, we will summarize the presentation of this work, and describe our ongoing research with these tools.

CHAPTER IX

DISCUSSION

This dissertation represents an approach to several aspects of parallel performance profile management and analysis. We designed a data management framework for the archival and management of performance data. We designed a framework for large scale profile analysis, both with regard to the number of profiles being analyzed and their respective size. We applied data mining methods to explore large scale profiles and reduce their complexity for analysis. We addressed the need for repeatable analysis which can be used to reuse successful process workflows. Finally, we integrated rule-based systems and classification methods to provide analysis interpretation and to construct parameter recommendation systems, respectively. In this chapter we will discuss the respective contributions with regard to our successes and to the lessons learned from the design decisions we made.

Performance Data Management

Our goal with respect to performance data management was to create a portable, scalable, and reusable management framework. The database schema which we used was designed to be used on a wide variety of database management systems (DBMS), and with a few exceptions, was successful. We have tested the database schema with two commercial databases, two open source databases, and with one embedded open source database. Some code changes were necessary to support non-standard SQL support. In all cases, we did not encounter obstacles which precluded us from supporting a new DBMS. However, this did prevent us from using a vast array of sophisticated database features, such as database views, stored procedures, and complex query support which is not supported on all DBMS. Focusing on one DBMS could allow us to improve the performance and reliability of the framework, but at the expense of our portability goal.

Our use of Java certainly addresses our portability concerns, as Java programs are executed in a virtual machine environment, and can be used anywhere a Java virtual machine (JVM) is available. And therein lies one problem - after collecting data on a leadership (large scale) parallel supercomputer, the data often has to be transferred to another machine before it can be loaded into the data management framework. That is because the leadership machines often have specialized hardware, and a JVM is not available. While this causes problems with regard to automated data collection and archival, it is not insurmountable. The data collection utilities chould be re-written in in Python (which is more commonly supported on leadership class machines) to provide a data archiving approach that would maintain our portability goals. Because the data collection utilties are a small component of the framework, we could re-write them in a compiled language, but there is the concern that a given DBMS might not have a client library ported to unusual hardware.

The decision to support only parallel profiles was a result of our scalability goal. Trace files would have been far too large to keep around for an extended period of time, and with support for phase-based and snapshot profiles (see page 213), we can maintain some resolution along the time axis, albeit at a large granularity. It is important to remember that there is no one tool for all situations, and that traces are valuable in detailed performance studies involving diagnosis of communication inefficiencies. However, profiles work very well in our context of summarized performance results for the purpose of comparison to other performance data. But we do recognize that there is a loss of temporal resolution when we limit ourselves to only profile data. While there are opportunities to improve the performance of our tools, we have also demonstrated in Chapter III that the database performs very well with large amounts of performance data, and that the summary tables in the database are key to scalably comparing the performance from tens or hundreds of experiments at a time.

The inclusion of metadata has also demonstrated the flexibility and extensibility of the database schema. The use of XML allowed us to add metadata to the database schema without explicitly specifying columns for each possible data value, and without providing an unstructured "hash map"-like solution, using name-value pairs.

One interesting possible redesign would be explicit support for callpath data within the database. Currently, callpath data is supported via encoded strings in the database. One possible extension of the schema would be to add foreign key references and hierarchical relationships in the database to represent callpaths. As a concrete example, suppose we have an application where the `MAIN` function calls the functions `A` and `B`, which each in turn call the function `C`. In a flat profile, we would have measurements for the amount of time spent in `MAIN`, `A`, `B`, and `C`. A callpath profile with a depth of 1 would add values for `MAIN => A`, `MAIN => B`, `A => C`, and `B => C`. A callpath profile with more depth would also add values for `MAIN => A => C` and `MAIN => B => C`. Currently, the database supports this data by having 10 event timers, one for each callpath depth. It would appear that this kind of hierarchical data is a natural fit for foreign key relationships in the database, where each event would have an additional column called PARENT, which would hold a reference to the calling function (in the case of `MAIN` the PARENT column would be null. This would create explicit relationships in the database, but at the expense of query performance - data loading and selection queries would take longer, with questionable added value. A prototype implementation would be necessary to determine the usefulness and the subsequent effect on performance.

PerfExplorer Analysis Framework

Supporting more than one data analysis engine in the analysis framework has resulted in additional programming overhead with regard to adding new functionality to the framework. For example, if a new analysis technique is added to the framework, interfaces to both the Weka and the R analysis engines would also be required. This does make the extensibility claim for the framework rather suspect, due to the additional work required. As a result, one engine could get more robust support for one engine at the expense of the other. That said, Octave [63] is third potential analysis engine for which we are considering support. For the long-term viability of the project, it would make sense to chose one analysis engine and support it robustly.

Parallel Performance Data Mining

The clustering and correlation operations have proven valuable in performance analysis, identifying similarities between threads of execution and helping to identify rouge processes or groups of processes. However, there is a problem with the analysis results - in many cases, interpretation of the results is left up to the analyst. This was a motivation for integrating the rule-based system. In those instances where there is sufficient information for automated diagnosis, the rule-based system can do so. However, there are gray areas of interpretation which are not easily delineated into clear diagnosis classes.

In the evaluation of k-means cluster results, there is the obvious problem of choosing a value for k. That is an open research question in statistics. There are useful heuristics, an example of which is the Gap Statistic [136]. However, even that method has its limitations. It assumes that the data is a mixture of Gaussian distributions, whereas our experience is that parallel performance data is usually a mixture of long-tailed distributions. Regardless, it does provide some guidance for the user and for the rule-based system. Our efforts to automate the interpretation of analysis results will be hindered without better interpretive techniques.

Automation and Componentization

Using scripts for automation in the analysis framework has been successful at capturing process workflow and reusing them with additional data sources. If we were to provide something in the way of a visual programming interface in order to link the analysis components together, it would likely generate scripts which would then be saved and processed like manually written scripts. This could be a logical progression from the script automation to visual component automation, and building a data-flow interface into the analysis framework.

Automation can also be integrated into an Integrated Development Environment (IDE), like Eclipse, for parallel development. In fact, we have created a prototype implementation of what that integration [57]. With support for C/C++, Fortran, MPI, OpenMP, and performance tools, the Eclipse IDE is a serious contender as a programming environment for parallel applications. There is interest in adding capabilities in Eclipse for conducting workflows where an application is executed under different scenarios and its outputs are processed. For instance, parametric studies are a requirement in many benchmarking and performance tuning efforts, yet there was no experiment management support available for the Eclipse IDE. We designed an extension of the Parallel Tools Platform (PTP) plugin for the Eclipse IDE. The extension provides a graphical user interface for selecting experiment parameters, launches build and run jobs, manages the performance data, and launches an analysis application to process the data. As the Eclipse IDE is integrated into more large scale parallel environments, and subsequently becomes more popular as a parallel development tool, our parametric study support would be extended in order to request targeted performance analysis. For example, if a given analysis rule base requires hardware counter data, then hardware counter data would be requested and collected through the IDE, and the analysis workflow would be processed.

Knowledge Engineering

There are number of topics for discussion with regard to knowledge engineering. Needless to say, correlation between metadata fields and performance measurements, clustering on metadata fields, and classification of performance data is not possible unless the metadata is collected. Unfortunately, choosing the right metadata fields to collect is not obvious. Relevant metadata fields need to be collected, not just everything, but in the long run, it is better to err on the side of collecting too many fields, rather than not enough. As we have demonstrated, If hundreds of fields are collected, analysis can determine whether any of them are correlated with performance. And while the reasonable list of potentially useful metadata fields is large, it is not unbearably so. In Chapter VII we listed the fields which we believe will be useful in the analysis process. This is not intended to be an exhaustive list, but rather a solid first start. As our research in this area continues, we plan to determine which fields are essential in performing our goal analysis scenarios.

In addition to solving the problem of which fields to select, the collection of metadata fields needs to be easier. While there are easy to implement methods in TAU for instrumenting source code to capture runtime parameters, that requires effort on the part of the analyst or the code developer. In the case of the analyst, they might not be familiar with the code, and may not realize what various input fields do, in order to relate them parallel performance. In the case of the code developer, while they may understand the details of their algorithmic decisions, they may not realize that some fields may affect the runtime performance of the code. Certainly, any parameter which is runtime configurable is a candidate for inclusion, but for some codes that could be a significant list. It would be useful to provide an auto-instrumentation technique which would use pragmas to identify which local variables in a given source file should be captured as metadata fields. The technique would be similar to OpenMP pragmas, in that a compiler which does not support OpenMP would just ignore the pragmas, and treat them as comments. The application could still safely be compiled without a performance measurement tool (such as TAU) without requiring the temporary or permanent removal of the metadata statements. The collection and formatting of build-time metadata fields is also cumbersome, but there are projects such as PerfTrack [65] which are trying to automate that process.

The amount of effort required to create a useful rule base cannot be underestimated. Creating the rules, applying them to the appropriate scenarios, and maintaining them requires a significant amount of work. We are currently exploring methods used by other bottleneck detection and performance diagnosis tools, investigating how their methods might be integrated as inference rules. Once a solid rule base is developed, it will require ongoing maintenance, as new HPC platforms are designed, new HPC languages are adopted, and new infrastructure libraries are released. Regardless, maintaining the rule base will require a commitment beyond what is usually required to support analysis source code.

As described in Chapter VII, the inference engine used by our research prototype does not support backward chaining. Backward chaining allows the inference system to examine the potential diagnoses, or conclusions, and search for evidence which might support or reject them. Backward chaining would be a more natural fit for exploratory performance analysis, in that inconclusive analyses could request that additional data be collected by the user, targeting the types and amount of performance data required for the analysis. It should be noted that the forward chaining inference engine is equally capable of requesting missing performance data from the user, but the backward chaining model is a more natural fit for targeted diagnosis. When a discrete set of known problems form the basis of the diagnoses, it is more intuitive to work backwards to search for corresponding symptoms.

There have been many lessons learned from our work in using classifiers to build parameter recommendation systems. First, using Java will likely be an obstacle on leadership class HPC systems, for the same reason it is difficult to load performance data into a performance data repository - Java is often not available on leadership class systems. That is merely a implementation issue, however - the data which is produced by the respective classifiers can be loaded by a runtime component written in C or C++, provided that the code which does the classification of test instances is easily translated to a new programming language.

Like the selection of parameters for clustering and correlation, the selection of parameters for classification can be difficult. However, in many domains, the developers who would be building a recommendation system often have heuristics for guiding parameter selection, and those can be used as a starting point for selecting appropriate classification parameters. Regardless, there is some trial-and-error involved in building and pruning the parameter list. Again, there are analysis methods which are useful in determining which parameters contribute to the variance in the data, such as Principle Components Analysis (PCA) and Analysis of Variance (ANOVA).

Additional Observations

In the current high performance parallel analysis climate, the research community has been developing tools based on the measurements available. The community should spend more effort focusing on the opposite dynamic, that of requesting measurements based on the analysis requirements of the tools. This is a larger extension of the backward chaining model. We suggest that guiding the performance data collection process by the requirements of the analysis would lead to new insights into the behavior characteristics of parallel codes. However, automation of this effort would require integration of the performance analysis tool into the development environment, which we discussed earlier in this chapter. From an IDE, the analysis tool could request parametric studies of the application, and configure the performance measurement to capture the required data to perform that analysis, whether or not includes hardware counters, callpath profiling, phase based profiling, or extensive metadata capture.

There are automatic performance analysis and optimization projects (APART, PERI for example) which have the goal of automated performance analysis. An analysis and data mining framework such as ours has been collaborating in those projects, and we feel that there is common performance analysis knowledge available from those efforts which should be integrated into the inference rule base of our framework. For example, there are symptoms which can identify specific performance diagnoses, and we should integrate those symptoms into our performance analysis framework.

Finally, our performance analysis framework could be modified to interact with other performance analysis tools. The combined analyses of these tools could lead to new insights into performance characterization. For example, the Paraver tool [107] has added the ability to perform cluster analysis of a parallel trace along the time axis, to reduce a long-running parallel trace into a representative sample of the behavior. In future work, the cluster analysis in our framework will be used to then cluster along the thread-of-execution axis, reducing the data further, and identifying classes of behavior. We could also integrate the rule-based analysis of the reduced traces, to identify bottlenecks and diagnose inefficiencies.

CHAPTER X

CONCLUSION

Contribution Summary

This dissertation has presented a performance data analysis framework which supports both advanced analysis techniques and extensible meta analysis of performance results. With each of our application examples we have shown how parallel performance tools can be more effective with the integration of context specific aspects and properties of applications, parallel software, and high performance computing systems. Two key ingredients which we have integrated into the our analysis process are context metadata and expert knowledge. The inclusion of these additional data have created exciting new possibilities for intelligent performance data analysis.

With this research, we have explored a number of areas with respect to scalable knowledge assisted performance analysis. First, in order to effectively manage performance profiles, we explored the use of a general purpose performance data management framework consisting of a database schema, programming API and related tools. These data management components form the foundation of the performance data repository. Second, we explored differences between performance profiles using the data management framework. This process proved to be far easier than processing performance data on disk, extracting key features of the data, and then manually constructing visual comparisons of profiles using spreadsheet or plotting software. Third, inspired by the use of data mining techniques on performance data, we reproduced and further explored data reduction, clustering and correlation of large performance profiles. Fourth, we automated and extended the application of performance analysis processes in order to help eliminate costly errors and benefit from previous work. More importantly, we provided an extensible interface to our performance analysis process workflows. Finally, we integrated metadata and expert knowledge into our analysis, eliminating the need to speculate on possible factors affecting computational efficiency. We extended that analysis to provide recommendations based on empirical data to optimize performance for runtime decision making.

Future Work

Our collaboration with the PERI project provides avenues for us to explore the exchange of performance data with other performance tools, and benefit from collaborative analysis. Using our analysis methods, we will continue to collaborate with the performance analysis work in the PERI project, and contribute to the data exchange effort. This effort is related to our long term goals with the OpenUH collaboration as well. The exchange of performance data and analysis results could enable us to interconnect analysis tools with compilers or source code transformation tools, providing run time analysis information to aid in the compile time code optimization and transformation.

The TAU project has recently developed snapshot profiles[91]. Snapshot profiles are a special type of profile which are written to disk at regular intervals. Each *snapshot* represents the performance behavior of the application since the previous snapshot, or the beginning of the execution. Snapshot profiles are similar to phase-based profiles which were discussed in this dissertation. However, phase based profiles represent a sub-tree of the call graph in the profile, whereas a snapshot profile represents a subset of the runtime of the application. As of yet, we have not designed a database schema solution for snapshot profiles in our data management framework. This is something we would like to explore in the near future.

With regard to knowledge-based meta-analysis, we will continue to expand the rule set for our rule based system, and incorporate new types of performance metadata as we discover their relevance. We will continue to look for opportunities to include expert knowledge into our rule based system. One possible way of doing that is incorporating the inference knowledge from the Hercule work discussed on page 38. The rules encoded for that work are specific to analyzing trace files within the context of parallel performance models, but there is analysis expertise there that could be re-encoded for use in our framework. Secondly, the performance analysis capabilities in Java PSL discussed on page 36 could be integrated into our analysis rule base. While there is already some overlap, there is additional analysis expertise available in that specification language.

When incorporating the classification capabilities into our framework, we realized that with some adjustments, the numeric classifiers available in Weka would make it possible to perform numeric prediction. Having constructed a detailed model of an application's performance behavior, predictions could be performed with respect to adjusting hardware values. This type of analysis could be useful in porting applications to new hardware, and in the procurement and evaluation of new hardware.

Another area of high current interest in the high performance computing community is in heterogeneous computing with specialized accelerator hardware. The IBM Cell processor, General Purpose Graphical Programming Units (GPGPU), general purpose hardware accelerators, and to a lesser extent Field Programmable Gate Arrays (FPGA) are no longer exotic solutions. Heterogeneous computing is very challenging to do well, as would be programming with any specialized hardware. Regardless, two very high profile supercomputers currently have or will have heterogeneous platforms. The first, Tsubabme (Tokyo Institute of Technology), is a Grid Cluster of Sun Fire x4600/x6250 Cluster compute nodes, each of which has two 2.4/2.6 GHz quad-core Opteron CPUs and ClearSpeed Accelerators with 2.833 GHz Xeon E5440 processors. Tsubame is currently being upgraded with multiple GPGPUs per compute node, providing a third processor architecture for each node. Before the upgrade, Tsubame is the 24th fastest supercomputer in the world.

The second machine, Roadrunner (Los Alamos National Lab), is cluster of IBM BladeCenter nodes, each of which will have four dual-core 1.8GHz Opteron processors, and eight 3.2GHz PowerXCell 8i processors. To date, Roadrunner is the fastest supercomputer in the world. Programming either Tsubame or Roadrunner is very challenging. Decisions have to be made about how to split up work among the computational resources. Two types of analysis in our framework would be useful in analyzing how to split up the workload. Our rule-based system could be extended to analyze hardware counter data for a given application, and identify regions of code which would most benefit from being migrated to the accelerator hardware. Secondly, our classifier-based recommender system could potentially be used to make suggestions about how to proportionally distribute the workload among the various hardware resources, given their capabilities. Regardless, the need for good analysis tools to aid in software development on heterogeneous supercomputers will only continue to grow.

Finally, we will continue our work with the Computational Quality of Work (CQoS) project. We discussed our work with regard to GAMESS on page 149, and with regard to PETSc applications on page 190. Our recommender frameworks will continue to evolve, and the accuracy and usefulness of our predictions will improve. The PETSc project has interesting aspirations which extend beyond previous work in non-linear solver recommender systems. Current work has focused on providing a recommendation for the linear solver with respect to the initial properties of the problem. While that does provide some performance benefit, the interesting extension is to periodically recompute the intermediate properties of the problem, and ask for a recommendation on whether to change linear solvers. This is an interesting problem because while some slower solvers may consistently converge to a solution on a difficult data set, once the problem has been reduced, a faster solver may be able to complete the job in less time. This would require the ability to examine the performance of the application with regard to iteration phases, which we are currently doing. We will continue to extend this work, with the eventual goal of efficiently replacing the linear solver at runtime with a faster solver.

BIBLIOGRAPHY

[1] Advanced Computing Technology Center. https://domino.research.ibm.com/comm/research_projects.nsf/pages/actc.index.html, 2008.

[2] K. R. Abbott and S. K. Sarin. Experiences with workflow management: Issues for the next generation. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 113–120, New York, NY, USA, 1994. ACM Press.

[3] M. Addis, J. Ferris, M. Greenwood, P. Li, D. Marvin, T. Oinn, and A. Wipat. Experiences with e-science workflow specification and enactment in bioinformatics. In *UK e-Science All Hands Meeting*, 2003.

[4] D. Ahn and J. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of Supercomputing*, Baltimore, Maryland, 2002.

[5] Ames Laboratory/Iowa State University. The General Atomic and Molecular Electronic Structure System (GAMESS). http://www.msg.chem.iastate.edu/gamess/, 2007.

[6] R. Becker and J. Chambers. *S: An Interactive Environment for Data Analysis and Graphics.* Wadsworth & Brooks/Cole, Pacific Grove, CA, USA, 1984.

[7] R. Bell, A. Malony, and S. Shende. A portable, extensible, and scalable tool for parallel performance profile analysis. In *Proc. EUROPAR 2003 Conference (EUROPAR03)*, 2003.

[8] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Lecture Notes in Computer Science*, volume 1540, pages 217–235, 1999.

[9] S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes. Application of machine learning to selecting solvers for sparse linear systems. In *SIAM Conference on Parallel Processing*, 2006.

[10] S. S. Bhowmick, V. Vedagiri, and A. Laud. Hyperthesis: The grna spell on the curse of bioinformatics applications integration. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 402–409, New York, NY, USA, 2003. ACM Press.

[11] V. Bui, B. Norris, K. Huck, L. C. McInnes, L. Li, O. Hernandez, and B. Chapman. A component infrastructure for performance and power modeling of parallel scientific applications. In *CBHPC '08: Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*, pages 1–11, New York, NY, USA, 2008. ACM.

[12] W. Cabot, A. Cook, and C. Crabb. Large-scale simulations with miranda on bluegene/l. Presentation from BlueGene/L Workshop, Reno, 14–15 October 2003.

[13] CASC. Mpip: Lightweight, scalable mpi profiling. `http://www.llnl.gov/CASC/mpip`.

[14] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummy, N. Podhorski, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Institute of Physics (IOP) Journal*, 2008. (*in publication*).

[15] M. J. Clement and M. J. Quinn. Multivariate statistical techniques for parallel performance prediction. In *HICSS '95*, pages 446–455. IEEE, 1995.

[16] M. J. Clement and M. J. Quinn. Automated performance prediction for scalable parallel computing. *Parallel Comput.*, 23(10):1405–1420, 1997.

[17] M. Colgrove. Querying geographically dispersed, heterogeneous data stores: The pperfxchange approach. Masters thesis, Portland State University Computer Science Department, 2002.

[18] M. Colgrove, C. Hansen, and K. Karavanic. Managing parallel performance experiments with pperfdb. In *IASTED International Conference on Parallel and Distributed Computing and Networking (PDCN 2005)*, Innsbruck, Austria, 2005.

[19] D. Cunningham, E. Subrahmanian, and A. Westerberg. User-centered evolutionary software development using Python and Java. In *Proceedings of the 6th International Python Conference*, pages 1–9, San Jose, Ca., Oct. 1997.

[20] Curtis Janssen. Massively Parallel Quantum Chemistry Program (MPQC). `http://www.mpqc.org/`, 2007.

[21] D. Gilbert and T. Morgner. Jfreechart. `http://www.jfree.org/jfreechart/`.

[22] S. Dasgupta. Experiments with random projection. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 143–151. Morgan Kaufmann Publishers Inc., 2000.

[23] L. A. de Rose and D. A. Reed. Svpablo: A multi-language architecture-independent performance analysis system. In *ICPP '99: Proceedings of the 1999 International Conference on Parallel Processing*, page 311, Washington, DC, USA, 1999. IEEE Computer Society.

[24] L. A. DeRose. The hardware performance monitor toolkit. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, *Euro-Par 2001: Parallel Processing: 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings*, volume 2150/2001, pages 122–132. Springer-Verlag Heidelberg, 2001.

[25] J. Dongarra. Basic linear algebra subprograms technical forum standard. *International Journal of High Performance Applications and Supercomputing*, 1(16):1–111, 2002.

[26] J. Ebedes and A. Datta. Multiple sequence alignment in parallel on a workstation cluster. *Bioinformatics*, 20(7):1193–1195, 2004.

[27] V. Eijkhout and E. Fuentes. Software architecture of an intelligent reommender system. Technical Report TACC Technical Report TR-08-01, Texass Advanced Computing Center (TACC), Austin, TX, January 2008.

[28] A. Espinosa, T. Margalef, and E. Luque. Automatic performance evaluation of parallel programs. In *IEEE Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, Madrid, Spain, January 1998.

[29] A. Espinosa, T. Margalef, and E. Luque. Automatic performance analysis of PVM applications. In *EuroPVM/MPI*, volume LNCS 1908, pages 47–55, 2000.

[30] S. Ethier, W. Tang, and Z. Lin. Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms. *Journal of Physics: Conference Series*, 16:1–15, 2005.

[31] European Center for Parallelism of Barcelona. Paraver. `http://www.cepba.upc.es/paraver/`, 2008.

[32] M. Fahey and J. Candy. Gyro: A 5-d gyrokinetic-maxwell solver. In *SC '04: Proceedings of the Proceedings of the ACM/IEEE SC2004 Conference (SC'04)*, page 26, Washington, DC, USA, 2004. IEEE Computer Society.

[33] T. Fahringer. *Automatic performance prediction of parallel programs.* Kluwer Academic Publishers, Boston, 1996.

[34] T. Fahringer and C. S. Júnior. Modeling and detecting performance problems for distributed and parallel programs with JavaPSL. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 35–35, New York, NY, USA, 2001. ACM Press.

[35] T. Fahringer and C. Seragiotto. Experience with aksum: A semi-automatic multi-experiment performance analysis tool for parallel and distributed applications. In *Workshop on Performance Analysis and Distributed Computing*, 2002.

[36] D. Feller. The emsl ab initio methods benchmark report: A measure of hardware and software performance in the area of electronic structure methods. Technical report, Pacific Northwest National Laboratory, June 1997. `http://www.emsl.pnl.gov/docs/tms/abinitio/cover.html`.

[37] D. F. Feng and R. Doolittle. Progressive sequence alignment as a prerequisite to a correct phylogenetic trees. *Journal of Molecular Evolution*, 25:351–360, 1987.

[38] A. L. Fisher and T. Gross. Teaching empirical performance analysis of parallel programs. In *SIGCSE '92: Proceedings of the twenty-third SIGCSE technical symposium on Computer science education*, pages 309–313, New York, NY, USA, 1992. ACM Press.

[39] J. A. for Marine-Earth Science and Technology. The earth simulator center. `http://www.jamstec.go.jp/esc/index.en.html`.

[40] K. Fürlinger and M. Gerndt. ompP: A profiling tool for openmp. In *Proceedings of the First International Workshop on OpenMP (IWOMP 2005)*, Eugene, Oregon, USA, May 2005. Accepted for publication.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[42] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable parallel trace-based performance analysis. In *Proc. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *LNCS*, pages 303–312, Bonn, Germany, September 2006. Springer.

[43] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. In *Distributed and Parallel Databases*, volume 3, pages 119–152, April 1995.

[44] S. Graham, P. Kessler, and M. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126. ACM Press, 1982.

[45] D. Gunter, K. Huck, K. Karavanic, J. May, A. Malony, K. Mohror, S. Moore, A. Morris, S. Shende, V. Taylor, X. Wu, and Y. Zhang. Performance database technology for SciDAC applications. *Journal of Physics: Conference Series*, 78, June 2007.

[46] O. Hernandez, H. Jin, and B. Chapman. Compiler support for efficient instrumentation. In *ParCo '07: Proceedings of the International Conference ParCo 2007*, pages 661–668, Julich, Germany, 2007. NIC-Directors.

[47] A. Hinneburg, C. Aggarwal, and D. Keim. What is the nearest neighbor in high dimensional spaces? In *The VLDB Journal*, pages 506–515, 2000.

[48] J. Hoffman. Pperfgrid: A grid services-based tool for the exchange of heterogeneous parallel performance data. Master's thesis, Portland State University Computer Science Department, 2004.

[49] D. Hollingsworth. The workflow reference model. Technical Report Issue 1.1, Workflow Management Coalition, `http://www.wfmc.org`, January 1995.

[50] J. K. Hollingsworth, A. Snavely, S. Sbaraglia, and K. Ekanadham. Emps: An environment for memory performance studies. In *NSF Next Generation Software Program Workship (held in conjunction with IPDPS)*, April 2005.

[51] K. Huck, A. Malony, R. Bell, and A. Morris. Design and Implementation of a Parallel Performance Data Management Framework. In *Proceedings of the International Conference on Parallel Computing (ICPP2005)*, pages 473–482, Oslo, Norway, 2005. *(Chuan-lin Wu Best Paper Award)*.

[52] K. A. Huck, O. Hernandez, V. Bui, S. Chandrasekaran, B. Chapman, A. D. Malony, L. C. McInnes, and B. Norris. Capturing performance knowledge for automated analysis. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–10, Piscataway, NJ, USA, 2008. IEEE Press.

[53] K. A. Huck and A. D. Malony. PerfExplorer: A Performance Data Mining Framework for Large-Scale Parallel Computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'05)*, Washington, DC, USA, 2005. IEEE Computer Society.

[54] K. A. Huck, A. D. Malony, S. Shende, and A. Morris. TAUg: Runtime Global Performance Data Access Using MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI)*, volume 4192/2006 of *Lecture Notes in Computer Science*, pages 313–321, Bonn, Germany, 2006. Springer Berlin / Heidelberg.

[55] K. A. Huck, A. D. Malony, S. Shende, and A. Morris. Scalable, Automated Performance Analysis with TAU and PerfExplorer. In *Parallel Computing (ParCo2007)*, Aachen, Germany, 2007.

[56] K. A. Huck, A. D. Malony, S. Shende, and A. Morris. Knowledge Support and Automation for Performance Analysis with PerfExplorer 2.0. *Scientific Programming, special issue on Large-Scale Programming Tools and Environments*, 16(2-3):123–134, 2008.

[57] K. A. Huck, W. Spear, A. D. Malony, S. Shende, and A. Morris. Parametric Studies in Eclipse with TAU and PerfExplorer. In *Proceedings of Workshop on Productivity and Performance (PROPER 2008) at EuroPar 2008*, Las Palmas de Gran Canaria, Spain, 2008.

[58] J. Hugunin. Python and java: The best of both worlds. In *Proceedings of the 6th International Python Conference*, pages 11–20, San Jose, Ca., October 1997.

[59] IBM. Ibm software - db2 universal database for linux, unix and windows. `http://www-306.ibm.com/software/data/db2/udb/`.

[60] R. K. Jain. *The Art of Computer Systems Performance Analysis.* Wiley, April 1991.

[61] S. Jarp. A methodology for using the itanium-2 performance counters for bottleneck analysis. Technical report, HP Labs, August 2002.

[62] Jim Rosinski. Gptl timing library home page. `http://www.burningserver.net/rosinski/gptl/index.html`, 2008.

[63] John W. Eaton. Octave home page. `http://www.octave.org/`.

[64] J. Jorba, T. Margalef, and E. Luque. Performance analysis of parallel applications with kappapi2. *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, 33:155–162, 2006.

[65] K. Karavanic, J. May, K. Mohror, B. Miller, K. Huck, R. Knapp, and B. Pugh. Integrating Database Technology with Comparison-Based Parallel Performance Diagnosis: The Perftrack Performance Experiment Management Tool. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'05)*, Washington, DC, USA, 2005. IEEE Computer Society.

[66] K. Karavanic and B. Miller. *On-Line Monitoring Systems and Computer Tool Interoperability*, chapter A Framework for Multi-Execution Performance Timing. Nova Science Publishers, New York, USA, 2003.

[67] R. E. Kass and L. Wasserman. A reference bayesian test for nexted hypotheses and its relationship to the schwartz criterion. *Journal of the American Statistical Association*, 90(431):928–934, 1995.

[68] D. Kearns. Java scripting languages: Which is right for you. *JavaWorld*, April 2002.

[69] D. Kearns. Choosing a java scripting language: Round two. *JavaWorld*, March 2005.

[70] KOJAK. Kojak. `http://www.fz-jeulick.de/zam/kojak/`, 2006.

[71] S. Krishnan and D. Gannon. Xcat3: A framework for cca components as ogsa services. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pages 90–97, April 2004.

[72] A. N. Laboratory. Performance visualization for parallel programs. `http://www-unix.mcs.anl.gov/perfvis/software/viewers/index.htm`, 2008.

[73] R. Laddad. Scripting power saves the day for your java apps. *JavaWorld*, October 1999.

[74] B. LaRose. The development and implementation of a performance database server. M.s. thesis, University of Tennessee, August 1993.

[75] L. Li and A. D. Malony. Model-based performance diagnosis of master-worker parallel computations. In *Europar 2006*, Dresden, Germany, 2006.

[76] L. Li and A. D. Malony. Automatic performance diagnosis of parallel computations with compositional models. pages 1–8, 2007.

[77] L. Li and A. D. Malony. Knowledge engineering for automatic parallel performance diagnosis. *Concurrency and Computation: Practice and Experience*, 19(11):1497–1515, 2007.

[78] L. Li, A. D. Malony, and K. Huck. Model-Based Relative Performance Diagnosis of Wavefront Parallel Computations. In *International Conference on High Performance Computing and Communications (HPCC2006)*, Munich, Germany, 2006.

[79] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An optimizing, portable OpenMP compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, April 2007.

[80] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Proceedings of SC2000: High Performance Networking and Computing Conference*, Dallas, November 2000.

[81] D. S. Linthicum. Process automation and eai. *eAI Journal*, pages 12–18, March 2000.

[82] LLNL. Bluegene/l. `https://asc.llnl.gov/computing_resources/bluegenel/`.

[83] LLNL. The asci sppm benchmark code. `http://www.llnl.gov/asci/purple/benchmarks/limited/sppm/`, 2006.

[84] LLNL. The asci sweep3d benchmark. `http://www.llnl.gov/asci/purple/benchmarks/limited/sweep3d/`, 2006.

[85] A. D. Malony. Event-based performance perturbation: a case study. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 201–212, New York, NY, USA, 1991. ACM Press.

[86] L. C. McInnes, J. Ray, R. Armstrong, T. L. Dahlgren, A. M. lony, B. Norris, S. Shende, J. P. Kenny, and J. Steensland. Computational quality of service for scientific CCA applications: Composition, substitution, and reconfiguration. Technical Report ANL/MCS-P1326-0206, Argonne National Laboratory, Feb 2006.

[87] J. Mellor-Crummey, R. Fowler, and G. Marin. Hpcview: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002.

[88] B. Milenova and M. Campos. O-cluster: Scalable clustering of large high dimensional data sets. Oracle Corporation, 2002.

[89] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[90] J. A. Miller, D. Palaniswami, A. P. Sheth, K. J. Kochut, and H. Singh. Webwork: Meteor2's web-based workflow management system. *Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems*, 10(2):185–215, March/April 1998.

[91] A. Morris, W. Spear, A. D. Malony, and S. Shende. Observing performance dynamics using parallel profile snapshots. In *Euro-Par*, LNCS, pages 162–171, 2008.

[92] P. Mucci. Dynaprof. `http://www.cs.utk.edu/~mucci/dynaprof`, 2008.

[93] P. Mucci, J. Dongarra, S. Moore, F. Song, and F. Wolf. Automating the large-scale collection and analysis of performance data on linux clusters. In *Proceedings of the 5th LCI International Conference on Linux Clusters: The HPC Revolution*, Austin, TX, May 2004.

[94] MySQL AB. Mysql: The world's most poplular open source database. `http://www.mysql.com`.

[95] NASA. Clips: A tool for building expert systems. `http://www.ghg.net/clips/CLIPS.html`.

[96] NASA. Nas project: Columbia. `http://www.nas.nasa.gov/About/Projects/Columbia/columbia.html`.

[97] National Center for Computational Sciences. Resources - National Center for Computational Sciences (NCCS). `http://info.nccs.gov/resources/jaguar`, September 2007.

[98] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. Perfsuite. `http://perfsuite.ncsa.uiuc.edu/`.

[99] National Energy Research Scientific Computing Center (NERSC). Nersc integrated performance monitoring (ipm). `http://www.nersc.gov/nusers/resources/software/tools/ipm.php`, 2008.

[100] NERSC. Petsc: Portable, extensible toolkit for scientific computation. `http://acts.nersc.gov/petsc`.

[101] B. Norris, J. Ray, R. Armstrong, L. C. McInnes, D. E. Bernholdt, W. R. Elwasif, A. D. Malony, and S. Shende. Computational quality of service for scientific components. In *Proceedings of the International Symposium on Component-Based Software Engineering (CBSE7)*, May 24-25 2004.

[102] U. of California San Diego. PMaC: Performance Modeling and Characterization. `http://www.sdsc.edu/pmac/`, 2008.

[103] U. of Maryland. DYNINST API. `http://www.dyninst.org/`, 2008.

[104] Oracle Corporation. Oracle database. `http://www.oracle.com`.

[105] Pacific Northwest Laboratory. NWChem. `http://www.emsl.pnl.gov/docs/nwchem/nwchem.html`, 2007.

[106] PERC. Performance evaluation research center. `http://perc.nersc.gov/`.

[107] V. Pillet, V. Pillet, J. Labarta, T. Cortes, T. Cortes, S. Girona, S. Girona, and D. D. D. Computadors. Paraver: A tool to visualize and analyze parallel code. In *In WoTUG-18*, pages 17–31, 1995.

[108] R. Prodan and T. Fahringer. A web service-based experiment management system for the grid. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 2003.

[109] Python Software Foundation. The Jython Project. `http://www.jython.org/`.

[110] Red Hat Middleware, LLC. Jboss.com - jboss rules. `http://www.jboss.com/products/rules`.

[111] D. Reed, L. DeRose, and Y. Zhang. Svpablo: A multi-language performance analysis system. In *10th International Conference on Performance Tools*, pages 352–355, September 1998.

[112] Rice University. HPCToolkit Home. `http://www.hipersoft.rice.edu/hpctoolkit/`, 2008.

[113] P. Roth. Towards automatic performance diagnosis on thousands of nodes. 5th International APART Workshop, SC2003 Conference, November 2003.

[114] S. Sarukkai and D. Gannon. Sieve: A performance debugging environment for parallel programs. *J. Parallel Distrib. Comput.*, 18(2):147–168, 1993.

[115] J. S. Seng and D. M. Tullsen. The effect of compiler optimizations on pentium 4 power consumption. In *INTERACT '03: Proceedings of the Seventh Workshop on Interaction between Compilers and Computer Architectures*, page 51, Washington, DC, USA, 2003. IEEE Computer Society.

[116] S. Shah, D. He, J. Sawkins, J. Druce, G. Quon, D. Lett, G. Zheng, T. Xu, and B. Ouellette. Pegasys: Software for executing and integrating analyses of biological sequences. *BMC Bioinformatics*, 5(1), 2004.

[117] S. Shende and A. D. Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–331, Summer 2006.

[118] S. Shende, A. D. Malony, and R. Ansell-Bell. Instrumentation and measurement strategies for flexible and portable empirical performance evaluation. 3:1150–1156, 2001.

[119] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57. ACM Press, October 2002.

[120] A. Sheth. Workflow automation: Applications, technology and research. In *SIGMOD '95*, pages 469–469, 1995.

[121] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Molec. Biol.*, 147:195–197, 1981.

[122] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An algebra for cross-experiment performance analysis. In *Proceedings of 2004 International Conference on Parallel Processing (ICPP'04)*, pages 63–72, Montreal, Quebec, Canada, 2004.

[123] R. D. Stevens, A. J. Robinson, and C. A. Goble. Mygrid: Personalised bioinformatics on the information grid. *Bioinformatics*, 19:302–304, January 2003.

[124] E. A. Stohr and J. L. Zhao. Workflow automation: Overview and research issues. *Information Systems Frontiers*, 3(3):281–296, 2001.

[125] Sun Microsystems. Java. `http://java.sun.com`.

[126] Sun Microsystems. Jdk 5.0 swing. `http://java.sun.com/j2se/1.5.0/docs/guide/swing/`.

[127] Sun Microsystems. The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR#94. `http://jcp.org/en/jsr/detail?id=94`, 2008.

[128] D. K. Tafti. Genidlest - a scalable parallel computational tool for simulating complex turbulent flows. In *Proceedings of the ASME Fluids Engineering Division*, November 2001.

[129] V. Taylor, X. Wu, and R. Stevens. Prophesy: An infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev.*, 30(4):13–18, 2003.

[130] The Apache Software Foundation. Apache derby. `http://db.apache.org/derby/`.

[131] The Omega Project for Statistical Computing. The r & s interface to omegahat and java. tt http://www.omegahat.org/RSJava.

[132] The Open—SpeedShop Team. Open—speedshop for linux. `http://www.openspeedshop.org/`, 2008.

[133] The PostgreSQL Global Development Group. Postgresql. `http://www.postgresql.org`.

[134] The R Foundation for Statistical Computing. R project for statistical computing. `http://www.r-project.org`, 2007.

[135] J. Thompson, D. Higgins, and T. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucl. Acids Res.*, 22:4673–4680, 1994.

[136] W. G. Tibshirani R. and H. T. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(2):411–424, 2001.

[137] Top500.org. Top 500 supercomputing sites. `http://top500.org/`, 2008.

[138] H.-L. Truong and T. Fahringer. Scalea: A performance analysis tool for parallel programs. *Concurrency and Computation: Practice and Experience*, 15(11-12):1001–1025, 2003.

[139] O. o. A. S. C. R. United States Department of Energy, Office of Science. Science applications and science application partnerships. `http://www.scidac.gov/app_areas.html`, 2006.

[140] O. o. S. United States Department of Energy. Federal plan for high-end computing, report of the high-end computing revitalization task force. 2004.

[141] University of Southampton. Graphical benchmark information service. http://www.netlib.org/parkbench/gbis/html/gbis.html.

[142] K. University of Tennessee. Lapack – linear algebra package. http://www.netlib.org/lapack/.

[143] US Dept. of Energy. Performance Engineering Research. http://www.scidac.gov/compsci/PERI.html, 2007.

[144] M. Valluri and L. John. Is compiling for performance == compiling for power, 2001.

[145] Vampir. Vampir. http://www.pallas.com/e/products/vampir/index.htm.

[146] J. S. Vetter and D. A. Reed. Managing performance analysis with dynamic statistical projection pursuit. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 44, New York, NY, USA, 1999. ACM Press.

[147] J. S. Vetter and P. H. Worley. Asserting performance expectations. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–13, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[148] R. Vuduc, J. Demmel, and J. Bilmes. Statistical models for automatic performance tuning. In *Proceedings of the International Conference on Computational Sciences-Part I*, pages 117–126. Springer-Verlag, 2001.

[149] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl.* O'Reily, third edition, 2000.

[150] G. Wang and D. K. Tafti. Uniprocessor performance enhancement with additive schwarz preconditioners on origin 2000. *Adv. Eng. Softw.*, 29(3-6):425–431, 1998.

[151] N. Wichmann, M. Adams, and S. Ethier. New advances in the gyrokinetic toroidal code and their impact on performance on the Cray XT series. In *Cray Users Group*, Seattle, Washington, 2007.

[152] T. Williams and C. Kelley. gnuplot homepage. http://www.gnuplot.info/.

[153] P. H. Winston. *Artificial Intelligence.* Addison-Wesley, third edition, 1992.

[154] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005. `http://www.cs.waikato.ac.nz/~ml/weka/`.

[155] F. Wolf and B. Mohr. Automatic performance analysis of SMP cluster applications. Technical Report 05, Research Centre Julich, 2001.

[156] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. 23(1):20–24, 1995.

[157] Y. Zhang, R. Fowler, K. Huck, A. Malony, A. Porterfield, D. Reed, S. Shende, V. Taylor, and X. Wu. US QCD Computational Performance Studies with PERI. *Journal of Physics: Conference Series*, 78:24–28, June 2007.

[158] A. Zomaya, editor. *Parallel Computing for Bioinformatics and Computational Biology: Models, Enabling Technologies, and Case Studies.* Wiley Series on Parallel and Distributed Computing. Wiley Interscience, 1 edition, 2006.