RICE UNIVERSITY

Developing A Scalable, Extensible Parallel Performance Analysis Toolkit

by

Adam Bordelon

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree Master of Science

APPROVED, THESIS COMMITTEE:

Dr. John Mellor-Crummey, Chair, Associate Professor, Computer Science, Rice University

DECEASED

Dr. Ken Kennedy, Thesis Director, John and Ann Doerr University Professor, Computer Science, Rice University

Dr. Keith Cooper, Professor, Computer Science, Rice University

Dr. Matteo Pasquali, Associate Professor, Chemical and Biomolecular Engineering, Rice University

HOUSTON, TEXAS

APRIL, 2007
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.
Abstract

Developing A Scalable, Extensible Parallel Performance Analysis Toolkit

by

Adam Bordelon

Modern parallel systems and applications are constantly increasing in scale and complexity, and consequently good parallel performance is impossible to achieve without the help of performance tools. However, monitoring application performance on these large-scale systems generates massive amounts of performance data. Current performance tools are insufficient for practical analysis of such large-scale data, typically either showing only basic summary information, or bombarding the user with all of the performance details with little help for pinpointing useful patterns. This thesis presents HPCVision, an extensible tool framework with a novel approach for scalable parallel performance analysis and visualization. This framework provides two performance toolkits for examining similarities and differences in parallel performance among an ensemble of processes, identifying equivalence classes of behavior, and pinpointing performance anomalies. HPCVision presents the performance data and analysis results in an intuitive, scalable manner to provide insight into application performance, automating the tuning cycle and increasing the productivity of the human analyst.
Acknowledgments

First I would like to express my deepest thanks to the late Dr. Ken Kennedy, my former advisor and patriarch, for giving me the needed vision and calm confidence for this work. I have learned much of what it takes to be a researcher and a scientist from him. Second, I would like to thank my thesis advisor Dr. John Mellor-Crummey for stepping in when I needed him, and for his constant flow of new ideas. I am especially grateful for all the last-minute work he has done to help me turn my ideas into reality. A special thanks goes to my former co-advisor Dr. Rob Fowler, now at the Renaissance Computing Institute, for pushing me hard and getting me on a great start. Much of the credit for this work should go to him for suggesting so many ideas and guiding me along the way. I am also grateful to Dr. Matteo Pasquali and the Pasquali Complex Flow of Complex Fluids group for providing their testcases and supporting me in this work. I would like to express my appreciation to Dr. Bradley Broom for his advice about statistical clustering analysis techniques, and to Dr. Jan Hewitt for her thesis-writing advice and instructive feedback along the way. I also thank the Rice Parallel Compilers Group and all the graduate students who have assisted me and my research. Finally, I thank my family and friends for their continuing love and support.

This work used the Rice Terascale Cluster funded by NSF under Grant EIA-0216467, and a partnership between Rice University, Intel, and HP. This work was supported by the Department of Energy’s Office of Science through its SciDAC program and by the Department of Energy under Contract Nos. 03891-001-99-4G, 74837-001-03 49, 86192-001-04 49, and/or 12783-001-05 49 from the Los Alamos National Laboratory.
Contents

List of Illustrations vi
List of Tables viii

1 Introduction ........................................ 1
   1.1 Problem Statement ............................ 2
   1.2 Thesis Statement ............................. 3
   1.3 Contribution ................................ 3
   1.4 Thesis Organization ......................... 4

2 Background and Related Work .................. 5
   2.1 Performance Tools ........................... 5
      2.1.1 Parallel Systems ....................... 5
      2.1.2 Performance Monitoring ............... 6
      2.1.3 Performance Analysis .................. 8
      2.1.4 Performance Visualization ............. 13
      2.1.5 Performance Tool Design ............... 16
   2.2 Eclipse Platform ........................... 17
      2.2.1 Rich Client Platform ................... 17
      2.2.2 Extension Points ....................... 17
      2.2.3 Graphical User Interface ............... 19
      2.2.4 Related Projects ....................... 19
   2.3 Summary .................................... 21

3 Tool Design .................. 22
5 Conclusions and Future Work

5.1 Future Work

Bibliography
Illustrations

2.1 Eclipse and the Rich Client Platform [61] ................. 18
2.2 Workbench Component Diagram [24] ..................... 20

3.1 HPCViewer Perspective, showing Performance Map View ........ 26
3.2 HPCViewer Perspective, showing Source View ............ 27
3.3 Cluster Perspective ......................................... 28
3.4 HPCVision Component Architecture ....................... 31

4.1 Outline of Sweep3D Algorithm Flow ...................... 38
4.2 Sweep3D Performance Overview using the HPCViewer Perspective .... 40
4.3 Sweep3D: Scope Tree Performance Breakdown for Procedure sweep .. 40
4.4 Sweep3D K-Means Cluster List (k=15) .................... 42
4.5 Sweep3D K-Means Cluster (a) ............................ 44
4.6 Sweep3D K-Means Cluster (b) ............................ 44
4.7 Sweep3D K-Means Cluster (c) ............................ 45
4.8 Sweep3D K-Means Cluster (d) ............................ 45
4.9 Sweep3D K-Means Cluster (e)............................ 47
4.10 Sweep3D K-Means Cluster (g) ............................ 47
4.11 Sweep3D K-Means Cluster (h) ............................ 48
4.12 Sweep3D K-Means Cluster (i) ............................ 48
4.13 Sweep3D K-Means Cluster (j) ............................ 49
4.14 Outline of LU SSQR Algorithm Flow ...................... 51
Tables

4.1 Sweep3D Performance Phases (from sample profile on node n26). . . 41
4.2 LU Performance Phases (from sample profile on node n26). . . . . . 55
Chapter 1

Introduction

Modern parallel systems include a wide range of architectures. Over the years, parallel systems have continued to increase in scale and complexity. The architectural complexity and sheer level of parallelism in today's large-scale systems makes it difficult to develop high performance applications that can fully exploit the capabilities of these systems. Today, compilers provide little assistance for anything other than optimizing individual node performance; as a result, users must become intimately involved in the analysis and tuning of applications for scalable parallel systems.

The application tuning cycle begins with the original source code, which is then compiled into an executable. The application is run and profiled, logging performance data during execution. This profile data is typically analyzed and displayed to the user by any number of available tools, which typically require the user to sort through the displayed data, track down the inefficiencies, and modify the code by hand to produce freshly tuned code to start the cycle over again.

Providing a scalable tool for parallel performance analysis and visualization can guide users in the identification and correction of application performance problems, therefore automating the application tuning cycle and increasing the productivity of the human analyst. To effectively and productively utilize large-scale parallel performance data, modern performance tools must supply sophisticated analysis techniques to delve into the massive performance data and automatically discover key performance issues. Intuitive methods for scalable data visualization are needed to guide the user from a global performance overview to a display of analysis results and down to the source code or systemic circumstances contributing to the poor performance. De-
signing a tool for portability, modularity, and extensibility enables component reuse and increases the likelihood of widespread adoption and further development.

1.1 Problem Statement

As modern systems scale up to thousands or hundreds of thousands of processors, and applications grow to millions of lines of code, the performance data generated while monitoring parallel programs becomes overwhelming. Analyzing these performance profiles manually would not only be time-consuming and error-prone, but also impossible for a human to glean any useful information from the mountains of data. Large-scale parallel performance data makes performance analysis more difficult and presents new challenges for scalable performance tools.

Current performance tools suffer from numerous shortcomings that make them insufficient for practical scalable performance analysis. Typically, they emphasize descriptive information, merely displaying the collected performance data without attempting to guide the user to significant issues or suggest methods for improvement. Many tools show summary information and ignore useful details hiding in the complex data, or they bombard the user with all of the performance details with little help for pinpointing useful patterns. Other tools may provide creative analysis methods, but they are complicated and unusable in practice, preventing widespread acceptance by the community. Modern tools have yet to successfully automate parallel performance analysis, and must still rely on the user for intelligent analysis and complex code-tuning.

This state of affairs presents a series of challenges for performance tool design. How can a parallel performance tool be designed to increase the user's productivity by automating performance analysis and tuning without being overwhelmed by complex, large-scale performance data? How can a tool use parallel performance data to automatically pinpoint performance anomalies, locate bottlenecks, and identify load imbalances? How can this information be presented to the user in a scalable, struc-
tured manner to provide insight into application performance and assist the user in correcting the performance issues found?

1.2 Thesis Statement

Understanding the execution behavior of parallel applications requires identifying patterns and anomalies in the performance across processes. Clustering techniques can organize performance data to make patterns and anomalies more accessible and help users comprehend the behavior of Single-Program, Multiple-Data (SPMD) parallel applications.

1.3 Contribution

This thesis describes my research and development work on HPCVision, an extensible framework for scalable analysis and visualization of parallel performance profiles. It presents an improved clustering analysis toolkit that implements methods for analyzing performance data using advanced statistical clustering. These methods can be used to examine similarities and differences in parallel performance among an ensemble of processes, identify equivalence classes of behaviors for groups of source code regions across nodes, or pinpoint anomalous node behavior that degrades overall application performance. HPCVision presents performance data and analysis results in an intuitive, scalable, structured manner to provide insight into application performance, automating the application tuning cycle and increasing the productivity of the human analyst. HPCVision is a modular, extensible framework built on the Eclipse Platform [23]. It is designed to enable future developers to easily contribute additional functionality or integrate it with other tools, like the Eclipse IDE and the Parallel Tools Platform [59]. We test the design and effectiveness of the HPCVision framework by producing and evaluating a prototype system.
1.4 Thesis Organization

The following chapters describe the design, implementation, and evaluation of our approach to developing HPCVision as a scalable, extensible parallel performance analysis toolkit. Chapter 2 begins with a brief background of performance tool concepts and descriptions of the approaches used by existing tools. Chapter 3 follows with details on the HPCVision design and prototype implementation. In Chapter 4, the HPCVision prototype is applied to two application case studies to demonstrate the efficacy of the implemented techniques. Finally, Chapter 5 concludes with a summary of our work so far and suggestions for future work.
Chapter 2

Background and Related Work

Before delving into tool design details and performance evaluation, we must first explain the key concepts underlying current performance tools and evaluate their utility for our applications. This chapter also introduces the Eclipse Platform as an extensible development foundation on which to build our tool.

2.1 Performance Tools

The process for performance analysis and tuning on parallel systems is similar to that of serial applications, but introduces additional complexity due to data volume, distribution, and scale. However, the application tuning cycle still follows the same steps: compile the code, launch and monitor the execution, analyze the performance data, visualize the results to the user, tune the code and repeat.

Current tools for performance analysis and visualization are limited in their ability to do useful scalable parallel performance analysis and report the results back to the user in an easy-to-understand interface. This section first describes the kinds of parallel systems being analyzed, and then discusses the features and challenges presented by current performance tools, motivating the design for our own parallel performance tool.

2.1.1 Parallel Systems

We focus here on parallel applications that run multiple processes divided among many nodes in a high-end parallel system. Most common parallel applications follow
the Single-Program, Multiple-Data (SPMD) paradigm, where each processor executes
the same program on its own piece of the data, communicating with other processors
as additional information is needed. Application performance is affected by many dif-
fferent factors, including operating system activity, load imbalance, serialization, data
copying, data access locality, communication inefficiencies, and underutilization of
processor functional units [44]. To find and correct these performance issues, various
tools have been developed for performance monitoring, analysis, and visualization.

2.1.2 Performance Monitoring

Application performance is typically measured through either simulation, tracing, or
profiling. Each method has trade-offs for data accuracy and performance overhead.

Simulation can be used to model performance in real or theoretical scenari-
os [72, 19], but accurate simulation measurements typically require extremely com-
plex models and lots of processing power. Though simulation is essential for modeling
performance for future systems or theoretical algorithms, when studying applications
on existing systems, measurement based performance analysis can often offer most of
the insight at a fraction of the cost.

Trace instrumentation involves inserting trace collection points around certain
functions or code regions to log program activity. While some tools rely on direct
source access and provide interactive instrumentation GUIs [18], others use wrapper
libraries [29] or direct binary instrumentation [38] to avoid recompiling or modifying
the executable. Some tools even provide dynamic instrumentation [21, 12] that can be
attached on the fly to a running process. Many tracing tools specifically monitor MPI
communication calls [69, 65] to explore parallel structure, whereas our work targets
on-node performance of parallel systems. Since tracing interferes with performance
in the instrumented parts of the application, instrumenting frequently-called code
can skew performance and drastically increase monitoring overhead. For large-scale
systems with lots of instrumentation on long-running applications, the generated trace files can be enormous, since they provide precise temporal performance information for each thread in the application's execution.

**Statistical profiling** operates by sampling low-level counters periodically at runtime to estimate hardware performance statistics with little overhead. Modern systems include hardware performance counters that consist of a small set of registers that monitor and count system events [11]. There are many tools available that interface with these counters and interpret the statistical profiles they produce [11, 43, 20, 41, 47, 7]. These monitoring tools can vary the sampling frequency to collect enough data for the desired level of accuracy without sampling so often that the performance is notably disturbed. The resulting profile data is more compact than trace files, but contains no temporal performance data. Associating the profile data with call-stack samples can reveal more information about a program's control flow without resorting to full temporal traces. For the most flexibility, some tools provide interfaces for both tracing and profiling [52].

**Data format.** The data produced by all of these tracing and profiling tools comes in a variety of formats, depending on whether a particular tool prioritizes the small size of binary logs, the readability of XML data [47], the flexibility of a self-defining data format (SDDF) [18], or the capabilities of a relational database [54, 52]. Regardless of the format, performance monitoring of large-scale applications on large-scale systems generates a large volume of data, and tools must decide how to present the data in a manner that is both usable and meaningful for further analysis.

Though these are all important distinctions when monitoring application performance, this work is not explicitly concerned with performance monitoring and data collection. HPCVision takes performance data generated by other monitoring tools, performs scalable post-mortem analysis, and visualizes the resulting data in a scal-
able, easy-to-use interface.

HPCVision currently uses a data model for XML data generated by HPC-Toolkit [47]. HPCToolkit profiles program execution by sampling hardware performance counter events [11, 43] and produces profile output for each process upon program completion. It interprets and aggregates these profiles, extracts the program structure from the binary, and correlates the profile data with the source code. The final output of a run is an XML performance database representing a hierarchical data structure that is intended for consumption by the provided hpcviewer graphical performance exploration tool. The whole process requires no source instrumentation or binary modification as long as the compiler recorded minimal debugging information.

A more recently developed HPCToolkit profiling method gathers these hardware counter values along with call-stack information, so that performance metrics can be attributed to calling contexts in a similar hierarchical callpath tree [28]. Both profile formats are viewable in the most recent hpcviewer tool [35], but our work thus far has focused on flat profiles. We are still investigating the use of statistical analysis with callgraph profiles.

2.1.3 Performance Analysis

There are many data analysis techniques that can be applied to performance data to simplify or summarize the data, discover performance issues, or compare data from multiple executions. Though some tools perform online dynamic analysis of running programs using custom runtime monitoring systems [54, 12], we choose to focus on post-mortem analysis of previously collected performance data to minimize runtime overhead and get a better picture of actual program behavior. Below we outline the principal analysis strategies that were the considered for this work.
Dimensionality Reduction. For statistical analysis, trace and profile data can be organized into multi-dimensional vectors, where each vector represents a thread profile and vector dimensions correspond to profile values for monitored source regions. With hundreds, thousands, or more profile events being measured, any analysis of this very high-dimensional data suffers from the curse of dimensionality. To improve the performance of later analysis, many tools support dimensionality reduction through feature selection and/or feature extraction.

Some feature selection methods use covariance analysis to find correlated metrics [2] and then analyze representatives from each group, while others use a value threshold to filter out less significant dimensions [36]. Our tool provides the latter, filtering out source regions with values below a user-defined threshold percent.

Feature extraction techniques transform the feature space by applying a mapping of the high-dimensional space onto lower-dimensional space. Principal Component Analysis (PCA) [2, 36] and Singular Value Decomposition are common factor analysis techniques that combine related variables into a few factors that explain most of the variance. The resulting lower-dimensional data can be analyzed more efficiently, and the results can be mapped back to the original feature space. Though our tool performs no feature extraction, modules implementing such methods could be integrated easily.

Supervised Classification. Many tools seek to automate the discovery of performance issues through supervised learning techniques like bottleneck pattern recognition [53, 49, 3] or decision tree classification [64], relying on training data to develop issue classes. These methods can be extremely valuable in that they can identify known classes of bottlenecks and failures, and are more able to guide the user in correcting such issues. However, any issues not covered by known patterns or classes remain undiscovered. To analyze performance data without relying on a priori information, we turn to unsupervised learning techniques, like statistical clustering
analysis.

**Statistical Clustering.** Clustering is the process of grouping data into classes of similar objects, so that objects within a cluster are very similar to each other, but are dissimilar from other objects outside of the cluster. Similarity can be measured by any number of methods, depending on the type of data and goal of clustering. Euclidean distances are the most common metric, but using other distance metrics such as the Manhattan or Chebychev metrics may produce more informative results. Normalizing the data or heavily weighting more important dimensions can also improve the quality of clusters found [30].

Clustering analysis can be used to identify equivalence classes and detect anomalies in parallel performance data, reducing massive amounts of data to representative groups and interesting subsets. Simple distance-based outlier detection can be used to identify anomalous processes [50], but this analysis ignores the larger equivalence classes discovered by conventional clustering. Clustering can be applied to parallel performance data to find groups of parallel threads, metrics, source events, or some combination thereof.

Basic clustering algorithms can be either hierarchical or partitional. Hierarchical clustering forms cluster hierarchies from previously determined clusters, either bottom-up, starting with each object in its own cluster, or top-down with all objects starting in the same cluster. Tools that use this method typically visualize the resulting hierarchy using a dendrogram [36]. On the other hand, partitional algorithms divide the feature space to find all clusters at the same time.

The K-Means algorithm partitions data objects into k distance-based clusters. Beginning with k initial centers, each object is assigned to the cluster whose center is nearest according to the distance metric. Once all objects are assigned, the cluster centers are recalculated as the mean of their cluster members. The data objects are reassigned using the new cluster centers, and this process repeats until convergence.
The clusters found by this algorithm vary depending on the number of clusters (k) to be found, the initial selection of the centers, and the distance metric used. We, like others [36], have implemented this algorithm in our tool due to its simplicity and speed.

Gene expression data from DNA microarrays follows a similar format to our parallel performance data, comprising numerous experimental samples with expression levels measured for a large number of genes. Analyzing and visualizing such data presents a similar challenge to the bioinformatics community. Extensive research has been done on the use of clustering and biclustering analysis on gene expression data [45]. We borrow from the knowledge gained in these experiments to select promising biclustering algorithms for parallel performance data.

In earlier research [9], we used the GeneClust [22] software package developed at M.D. Anderson Cancer Center to apply the 'gene shaving' [31] statistical analysis method to line and loop level histograms of performance metrics for processes. Gene shaving can find overlapping clusters of similar genes (or source line events in our case) with high variance across samples (or profiles). The program was able to differentiate interior processes from edge processes in data from parallel Sweep3D [1] runs. These promising results motivated an investigation of other algorithms being applied to gene expression data. After further investigation, other algorithms appeared more promising and easier to implement or integrate into our prototype tool, so our work with gene shaving analysis has been suspended for now.

Cheng and Church introduce the concept of biclustering for high-dimensional gene expression data [15], though it was Mirkin [51] who first coined the term to describe simultaneous clustering of both row and column sets in a data matrix. Cheng and Church use mean squared residue as a similarity measure of the coherence of the genes and conditions in the bicluster. Their biclustering algorithm finds overlapping clusters of coherent values one at a time, starting with the entire matrix and greedily removing (or adding) the row or column that will achieve the largest decrease of the
mean squared residue score. This continues until the score falls below a threshold \( \delta \), or cannot decrease anymore. The final set of rows and columns define the first \( \delta \)-bicluster. Once a bicluster is discovered, the elements in the original matrix corresponding to it are masked by random numbers in the same range as the whole matrix, and the algorithm is reapplied to find the next bicluster. The expectation was that these random values would not form recognizable patterns and thus would be leading candidates to be removed in the node deletion phase. However, when we implemented this algorithm and applied it to our parallel performance data, the first bicluster found represented all of the closely-related low-valued items. Once all of these values were masked with random values, each consecutive cluster contained increasing amounts of noise from the random replacements. Instead of masking with random values, better biclusters may be achievable by applying an orthogonalization technique similar to that in the gene shaving algorithm. We did not pursue any improvements for this algorithm or include it in the prototype.

After surveying the literature for biclustering gene expression data, we found the pCluster algorithm [66], notable for its ability to simultaneously find overlapping, nonexclusive clusters based on pattern similarity [45]. When the algorithm begins, it uses pairwise clustering to find the Maximum Dimension Sets (MDS) for each pair of rows and for each pair of columns. A MDS for a pair of rows is a set of columns that defines a maximum width bicluster on the pair of rows. A similar definition holds for column-pair MDSs. The candidate MDSs are then pruned to eliminate invalid pairwise clusters based on the user-defined parameters for the minimum number of cluster rows and columns and the intersections of the row-pair and column-pair clusters. Then, the remaining row-pair clusters are inserted into a prefix tree, where each tree node contains the list of cluster rows indexed in the tree by an ordering of the cluster columns (of course, the roles of the rows and columns can switched for a different approach). In a post-order tree traversal, the algorithm prunes the objects in each node, and detects (and outputs) the pClusters contained within. The
node objects are then propagated up the tree to nodes which represent a subset of the columns in the current node. This repeats until all valid pClusters in the tree are found. We are currently implementing this algorithm in our tool, but have not achieved satisfactory results yet.

Once the data has been reduced to a set of clusters, these clusters must be evaluated to determine how the clustered objects are similar and which clusters are most important and why. Clusters can be summarized and ranked by traditional statistics such as sum, size, mean, variance, minimum, and maximum. The similarity score used by the clustering algorithm may also prove informative. The quality of a cluster is often ranked by variance analysis, such as the F-ratio defined as $\frac{\text{Between-Cluster Variance}}{\text{Within-Cluster Variance}}$ [2]. HPCVision presents basic statistics for each cluster found to aid the user in cluster ranking and evaluation.

**Comparative Analysis.** In addition to analyzing data from a single execution run, many tools also provide comparative analysis between two or more executions. This could be a straightforward side-by-side visual performance comparison [55], or more complex charts showing parallel efficiency [18], speedup, or parallel runtime breakdown by event [36]. This kind of analysis is useful for regression testing and scalability analysis. The prototype tool does not perform cross-experiment comparisons, but such functionality could be added in a future module.

### 2.1.4 Performance Visualization

For performance data and analysis results to be useful, they must be presented to the user in a straightforward, logical manner. It is common for tools to begin by displaying a global overview of system or application performance, allowing interactive navigation to guide the user to more detailed information about specific performance characteristics or different aspects of the data. It is important that the user interface (UI) be easy to use, and automate the discovery of important performance issues.
Many tools fall victim to the performance visualization pitfall of bombarding the user with too much data or too many visualizations with no intuitive method for isolating interesting phenomena and guiding performance tuning. Below we discuss the different types of views that tools use to present performance information. It should be noted that an ideal performance analysis tool would use many of these visualizations, with an intuitive interface for interactive navigation among and between the different views.

**Basic Views** Most tools use data tables or simple bar charts to display the performance data. Some may allow more complex queries to a performance database, displaying the resulting data or statistics in a table [54]. Other tools use histograms to express the frequency distribution of the data among a number of categories [5, 49, 55]. These basic views provide a familiar perspective on the data, but may be too crowded when displaying large-scale data, making useful insights difficult to discover.

**System Views** When visualizing parallel performance data, the user often wants to be able to view data organized according to the system topology. This could be displayed as a system tree with correlated metric values, organized by racks, machines, processors, and processes, or as a colored topology map, arranged to match the data distribution pattern. Some tools just display all of the processes in a grid, with summary statistics or charts for each process, allowing easy visual cross-process comparisons [55]. For large numbers of processes, showing too much per-process information can result in visual overload. In these cases, a simple colored topology map would be the most scalable visualization.

**Application Source Correlation** Since performance tuning usually requires modifying the application source code, a good performance tool directs the user to the root of a problem, usually by mapping performance information to the source code [40, 48, 55, 3]. Some tools directly display the program source code, annotating
important lines with corresponding metric values or colored icons [18, 3, 48]. Others display source structure abstractions such as a program scope tree [48], a call-graph tree [55, 10, 48, 3], or just a list of important functions, loops, or branches [10]. Once the user has identified a performance issue, it is extremely important to be able to relate it to specific source regions, so that the code can be edited and tuned. Integrating a performance analysis tool into a common IDE would augment it with an easy and familiar editor interface and help automate the application tuning cycle by reducing the number of steps between performance problem discovery and the compilation and running of tuned code.

**Trace Timelines** The temporal data in trace files is frequently visualized as a timeline or Gantt chart [71, 60, 5, 55], showing one line or bar per process over time, with communication events shown as lines connecting the sending and receiving process of a message. For trace data over many processes, the clutter can be reduced by grouping processes into equivalence classes or just showing overall average/\(\min/\max\) trace statistics over time. One alternative tracefile visualization technique uses VCR-like playback controls to step through the temporal data, showing message events between processes organized topologically or in a circle [69]. HPCVision does not currently handle trace data, but future releases that do could take advantage of these visualization techniques.

**Cluster Views** Tools that perform clustering analysis use a variety of views to summarize and display the clusters found. While dendrograms are popular for displaying sets of one-way hierarchical clusters [2], other tools display one-way clusters using cluster size histograms, PCA scatterplots, topological membership maps, and typical statistical barcharts [36]. For clusters beyond one or two dimensions, however, these visualizations can only show part of the picture. Even colored three-dimensional charts can only express four dimensional data. To combat this, the HPCVision Cluster Perspective separates cluster visualization into a view of cluster row membership
and a view for cluster columns. Since a well-ordered ranking of clusters is equally important for cluster analysis, HPCVision displays the cluster list as a sortable table with important statistical attributes reported for each cluster found.

**Advanced Visualization** Besides these typical classes of performance data visualization, performance tools that offer comparative analysis, supervised classification, or other special techniques provide custom visualizations to display their analysis results. Performance problem classification tools may show a problem tree [53] or list of problem classes, colored by issue severity [20, 53, 10]. The Virtue tool provides immersive performance visualization using advanced hardware such as 3D mice, CyberGloves, video cameras, virtual reality displays, and immersive CAVEs [4]. We do not attempt any of these visualization models due to our choice of analysis techniques and hardware restrictions.

### 2.1.5 Performance Tool Design

Many of the performance tools in the field suffer from design and distribution limitations. Some hardware vendors provide platform-specific tools for their systems [37, 38, 40], which may be able to perform more detailed system-dependent resource monitoring, but are not globally useful. Many tools require paid commercial licenses, and do not distribute the program source with the tool. These closed-source tools are by design difficult or impossible to modify or extend, leading researchers to turn elsewhere for custom data, analysis, or visualization requirements. Some tools restrict themselves to specific data models, excluding potential users who are not ready to switch from an existing performance data format. More flexible tools are able to import and export between multiple data formats [53, 52]. For these reasons, we seek to develop a portable, extensible, open-source performance analysis and visualization tool that is self-contained, but able to integrate easily with other tools.
2.2 Eclipse Platform

We selected the Eclipse IDE Platform as a framework for HPCVision because of its mature, portable, near-native user interface and extensible platform for tool development and IDE integration. Eclipse is an open-source community project comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle [23]. Eclipse is built on an extensible framework composed of modules known as plug-ins, which encapsulate source and configuration files providing program functionality and user interface contributions.

2.2.1 Rich Client Platform

The Rich Client Platform is defined as the minimal set of Eclipse plug-ins needed to build a standalone platform application with a UI [46]. Building an application on top of the RCP provides a full-featured runtime as well as the Standard Widget Toolkit (SWT) and JFace UI framework, with a minimal required footprint. Figure 2.1 illustrates how Eclipse itself is built on the RCP.

An application built as a standalone RCP application can be easily integrated into the Eclipse IDE with a few minor changes. We implement HPCVision first as a standalone application to target parallel developers not yet ready to buy into the Eclipse IDE. Future IDE integration could help lure developers to a consistent interface for parallel performance analysis and application development.

2.2.2 Extension Points

The loose coupling of plug-in modules in Eclipse is achieved partially through the mechanism of extensions and extension points. When a host plug-in wants to allow other plug-ins to enhance parts of its functionality, it declares an extension-point describing a contract for extending its functionality. An extending plug-in simply needs to implement the interface defined by the contract to add the desired functionality. The key feature of extension-points is that the host plug-in needs to know nothing
Figure 2.1: Eclipse and the Rich Client Platform [61].
about the plug-ins connecting to it beyond the scope of the extension-point contract. Thus, plug-ins built by different organizations can integrate seamlessly without knowing much about each other. Also, since each plug-in uses its own classloader, a host plug-in can query metadata about existing extensions to its extension-point, select and load the appropriate extension for a certain situation, and never load unnecessary plug-ins into memory.

2.2.3 Graphical User Interface

The Standard Widget Toolkit (SWT) is Eclipse’s fast, thin alternative to the Swing or AWT UI toolkits. To provide a mostly native UI, SWT uses the Java Native Interface (JNI) to access widgets natively available on a host platform and uses emulation to provide widgets that are not natively available. JFace is a Java application framework based on SWT providing a set of reusable components to simplify GUI application development. JFace provides common GUI components such as wizards, actions, and dialogs as well as an adapter component called a viewer, which sits between an SWT widget and a user-provided data model.

The entire Eclipse UI is called the workbench (see Figure 2.2). It is displayed in one or more workbench windows, each containing a title bar, menu bar, tool bar, and status line. The main body of a workbench window is represented by a workbench page, containing views and editors which are layed out in the page according to its perspective. There is a service called the SelectionService that tracks all selection changes within the views and editors of a workbench page. Any listeners added to this service are automatically notified of selection changes.

2.2.4 Related Projects

The BIRT Chart Engine is a complete charting library that fully integrates with Eclipse and SWT applications to produce interactive charts, including bar, line, area, pie, scatter, stock, and meter charts, even in 3D [26]. We utilize the BIRT charting
Figure 2.2: Workbench Component Diagram [24].
capabilities to display the results of our clustering analysis.

The Eclipse Parallel Tools Platform [67] aims to be a highly integrated environment specifically designed for parallel application development, providing a standard parallel language IDE for various architectures and runtime systems, a scalable parallel debugger, integration support for parallel tools, and a simple environment for user-interaction with parallel systems. We can take advantage of this environment to couple performance data collection and management with the runtime framework, and then link the output to HPCVision for custom analysis and visualization. Integrating performance analysis and tuning tools with a parallel IDE and job launcher reduces all of the stages of the parallel application tuning cycle to a single graphical user interface.

2.3 Summary

Drawing from the experiences of other performance analysis tools, we set out to design a scalable performance analysis and visualization tool that supports automated analysis of parallel performance profiles using statistical clustering algorithms. We leverage the Eclipse Rich Client Platform to provide a portable, extensible framework and an effective user interface for our tool. The rest of this thesis demonstrates how these techniques can be implemented to provide a user with insight into the performance characteristics of large-scale parallel applications and then guide the user to the discovery and correction of significant performance issues, thereby automating the application tuning cycle and increasing the productivity of the human analyst.
Chapter 3

Tool Design

HPCVision is an extensible tool framework for scalable parallel performance problem identification. It analyzes parallel performance data with four goals in mind: (1) to assess general performance characteristics of the system and application; (2) to recognize equivalence classes of behavior; (3) to identify performance anomalies in the parallelism across the nodes; and (4) to discover other performance issues that could indicate bottlenecks, load imbalances, or other opportunities for optimization. Once the data is loaded, HPCVision analyzes it behind the scenes and presents the performance information in an intuitive scalable interface that helps the user understand the results and points back to the related source code regions. HPCVision leverages the Eclipse Rich Client Platform (RCP) [46] to emerge as a portable, extensible tool with a familiar, easy-to-use interface.

3.1 Design Considerations

Before designing a new software tool, a developer must first address some design considerations: what assumptions need to be made about the data, system, and user, and what key features need to be implemented to achieve the desired functionality? Once a plan is set, it is then time to construct the actual design of the program, accounting for specific platform implementation issues, program structure details, and user interface layout.
3.1.1 Assumptions

For the purpose of this work, certain assumptions were made about the format of the data being handled, the performance information it represents, and the user's familiarity with the parallel system.

Though HPCVision is designed for data format independence, the current prototype focuses on loading, analyzing, and displaying HPCToolkit data because of its availability in our research group. The HPCToolkit project [34] has been lead by John Mellor-Crummey and Rob Fowler, and it is under continuing development at Rice University. A flat HPCToolkit profile represented in XML is structured as a hierarchical program scope tree with scopes for load modules, files, procedures, loops, and statements. Associated with each scope node are metric count values for each performance metric profiled, with leaf values aggregated up the tree to the root. HPCToolkit collects this information by monitoring on-node performance for each program thread throughout the execution of a parallel application.

Assuming the parallel application follows the typical Single-Program, Multiple-Data (SPMD) paradigm, a well-behaved program should exhibit little variation across processor profiles for most scopes. Therefore, there are three important scenarios to diagnose in parallel performance data: (1) common characteristics of system or application performance; (2) partitions in the processor space, representing equivalence classes of behavior; (3) performance anomalies where one or a few processors behave differently from all the rest. Common characteristics can aid in the discovery of general application performance issues by locating the parts of the program where all of the processors spend large amounts of time and resources. Clean partitions between groups of profiles can identify load imbalances due to system or data differences, as well as equivalence classes created by the program structure. Performance anomalies involving only a small fraction of the profiles can help a user pinpoint problems caused by systemic anomalies, hardware failures, or application bugs. Identification of these scenarios is vital in helping the user understand and improve the performance of a
target application.

Finally, we assume that the user of this tool is generally familiar with the software being analyzed, as well as with the parallel hardware used. Therefore, the user should be able to recognize the program's structure and decipher the meaning of the source code regions the tool points them to. The user should understand what data is being passed to the program, and what the program is supposed to be doing with that data. It is assumed that the user knows how many of what kind of processors the program was executed on and is familiar with the basics of the system's memory hierarchy. Making these assumptions enables the development of a tool that can guide a somewhat-technical user to discover and correct parallel performance issues. Designing such a performance tool requires a multi-faceted approach.

3.1.2 Feature Requirements

Application performance analysis can be divided into five data phases: data generation, data loading, data analysis, data visualization, and data management. This section describes the purpose of each of these phases and how they are used or represented by HPCVision.

Data Generation. During program execution, profile or trace information is collected and stored in a filesystem or database. For parallel jobs, performance data should be collected for each node or processor on which the application is run. We leave this stage up to the user, and assume that the data is stored as a flat profile in an HPCToolkit XML file [33]. Future implementations will handle other data sources, and the analysis tool may be integrated with HPCToolkit through the Eclipse PTP parallel job launcher [58].

Data Loading. Once the data has been generated, it must be loaded in the analysis tool and parsed into a format that the tool can understand. When HPCVision first launches, the user is directed to select the datafile location and the toolkits to apply to
the data. Then the datafile is loaded by the appropriate data plug-in and interpreted into an internal data model. This data model is passed to the user-selected toolkits for analysis and visualization.

Data Analysis. For the prototype implementation, we have created two separate toolkits to handle performance data analysis and visualization. Each toolkit can have multiple analysis modules to handle different datatypes or analysis needs. The HPCViewer Toolkit is modeled after HPCToolkit's hpcviewer GUI tool [48]. It performs no complex analysis, but simply loads the data and displays it in a three-paneled interface. The Cluster Toolkit transforms the performance data to a two-dimensional matrix, applies a user-specified statistical clustering algorithm, and returns a list of clusters found. Future developers should be able to easily integrate new analysis modules to these toolkits, or even add a whole new toolkit with its own analysis module(s).

Data Visualization. Each toolkit must convey the data and/or analysis results in an easily-navigable interface that enables the user to explore the performance of their application execution. Each toolkit has its own visualization perspective, composed of one or more views displaying different aspects of the performance information. Clicking on different items in the views should trigger events in other views and, if relevant, other toolkit perspectives. The following describes the features of the various views in each toolkit perspective.

HPCViewer Perspective: Figures 3.1 and 3.2 show how the HPCViewer Toolkit displays different aspects of the performance data using the following views. System View: Shows a list of the the processor/profile nodes for the parallel job. Selecting a profile node in the list updates the Scope and Performance Map views to show values specific to the selected profile.
Performance Map View: Shows a graphical representation of the program scope tree
Figure 3.1: HPCViewer Perspective, showing Performance Map View.

for a process with a column for each depth level in the tree and blocks in each column sized according to their percent values. Clicking on a block in the performance map navigates to the correlated scope in the Scope View.

Scope View: Shows the source scope tree and the associated metric values for a selected profile. This view allows Flatten and Zoom actions like those in the original hpcviewer tool [48]. Double-clicking a file or lower scope in the tree launches the Source View scrolled to show the beginning of the selected source region.

Source View: Shows a read-only copy of the source file with corresponding line numbers. Selecting a line in the source with associated performance information triggers an event in the Scope View to navigate to the correlated scope and its values.
Figure 3.2: HPCViewer Perspective, showing Source View.

Cluster Perspective: Once the Cluster Toolkit has performed its analysis on the performance data matrix, it receives a list of Cluster items as the result. These clusters are presented to the user in the following views as shown in Figure 3.3.

Cluster List View: Shows the list of the clusters found, displaying sortable columns for various cluster attributes, depending on the algorithm or cluster type. These attributes could include cluster name/id, size, cluster-mean, variance, similarity score, etc. Selecting a cluster in the cluster list opens up the Cluster Chart Views for that cluster.

Cluster Chart Views: Since the performance matrix can be clustered on both rows (source regions) and columns (process profile), the Cluster Toolkit displays two charts
Figure 3.3 : Cluster Perspective.

showing cluster membership and statistics across both rows and columns. Clicking on a data point or row/column title could lead the user back to the HPCViewer Perspective, focusing on the appropriate profile, scope, and metric.

**Data Management.** Besides the analysis and visualization of the data, it is important to be able to save the results of the analysis for future reference. In this case, it would be useful to be able to save Cluster groups or an application performance model based on an average profile. Building a performance model based on this data would enable a future toolkit to compare one execution’s performance against the model representing a previous execution. Though not a priority for the prototype, a future implementation should support the exporting of a data model or analysis result set to a file or database. A Model Comparison Toolkit would be valuable for regression testing, scalability analysis, or cross-platform performance comparisons.
3.1.3 Design Considerations Summary

These assumptions and feature requirements lay the foundation for the development of a scalable, extensible parallel performance analysis tool based on two functional toolkits. The HPCViewer Toolkit is designed to direct the user from an overview of the application’s parallel performance to more detailed information about specific performance issues, even pointing to the misbehaving source code regions. The Cluster Toolkit provides a means for analyzing performance profiles to locate patterns in groups of profiles and groups of scopes, which the toolkit then visualizes in a way that enables the user to discover parallel performance anomalies and equivalence classes. This tool provides a scalable user interface for automatic parallel performance problem identification that guides performance tuning efforts and increases the productivity of the human analyst.

3.2 Implementation

This prototype implementation of the HPCVision framework addresses three fundamental objectives: (1) applying statistical clustering analysis to parallel performance profiles to easily and scalably discover patterns; (2) visualizing the performance data and analysis results in an intuitive, interactive display that assists the human analyst in locating performance bottlenecks; and (3) building a portable performance tool framework based on the Eclipse Rich Client Platform to extend with custom data, analysis, and visualization modules. The prototype is limited in the features that have been implemented, providing sufficient support for the HPCToolkit data model and two analysis toolkits, one for interactive performance data exploration and one for statistical clustering analysis. Design choices were made early on to abstract specific functionality into plug-in modules that can be added, removed, or replaced at will by the user. Actual implementation has unveiled some issues in certain algorithms, design interfaces, and data visualizations. Lessons learned from this implementation have been included in a refactoring and redesign strategy for the tool, some of
which have already been integrated, while others are in the works for a future release. This section describes the application’s component architecture and discusses implementation details. Installation instructions, user guide, demos, and developer documentation will be distributed with the tool and its source, and hence are not fully covered in this thesis text.

### 3.2.1 Plug-in Framework

HPCVision is implemented as a series of interconnected Eclipse Plug-ins. Building on Java and Eclipse creates a portable, extensible tool with a familiar, easy-to-use interface. The prototype is implemented as a standalone Rich Client Platform application [25], so that a user unwilling to buy into Eclipse as an IDE is not be forced to download the entire Eclipse IDE just to use this tool. However, the RCP functionality is separated into the Base plug-in, so that a future implementation could use a different base plug-in to integrate the analysis tool with Eclipse [23], the PTP [59], or other related frameworks. Close integration between the IDE, parallel launcher, and analysis tool helps automate the application development and tuning cycle by reducing the number of applications the user must learn, execute, and monitor.

Figure 3.4 shows a component architecture diagram demonstrating the interactions between the different HPCVision plug-ins. The following sections describe each component in detail.

### 3.2.2 Base Plug-in

The Base plug-in acts as a driver for the whole tool and defines the interfaces for the data and toolkit plug-ins to implement. In addition to configuring basic RCP application settings like window size and title, the Base Plug-in also sets the default window perspective and initializes menus and toolbars. By default, the prototype launches with an empty HPCViewer Perspective, though this could be a blank perspective or some sort of Data Manager Perspective. Besides basic actions like Quit and Help, we
have added an Open action, present in the File Menu and the main toolbar, which launches the Open Datafile Wizard.

In the Open Datafile Wizard, the user chooses the datafile and selects which of the available toolkits to apply to the data. Once the toolkits are selected and the user clicks Next, each chosen toolkit is configured in its own page in the wizard. In this way, the user sets all necessary input parameters in a consistent wizard interface. When the wizard finishes, the datafile is loaded by the appropriate data model plug-in (only HPCToolkit data so far) and then the data model is passed to each of the user-selected toolkits.

3.2.3 HPCViewer Data Plug-in

The HPCVision prototype includes the HPCViewer Data Plug-in, which takes in a path to an HPCToolkit datafile and loads the data into an Experiment object. Much of the code for this data model was ripped directly from the hpcviewer Java
source [35] and minimally adapted for our purposes. The contents of the input datafile are read in and passed to the Apache Xerces XML parser library [42], which uses the XML data to populate the Experiment data model. The Experiment model includes data structures for the Scope Tree and its associated MetricValues, the Metric list, Profile Node list, and SourceFile references. Once all of the data is parsed into the model, the Experiment object is returned. A future implementation will update the Experiment model to handle Callgraph experiments like the latest Hpcviewer. Other data model plug-ins could load data from database sources, performance matrix flat-files, trace files, or other models.

### 3.2.4 HPCViewer Toolkit Plug-in

The first toolkit implemented was the HPCViewer Toolkit, which can either take in a path to an HPCToolkit datafile and load it with the data plug-in, or just take in the Experiment data model directly. We have migrated much of the functionality from the original Hpcviewer tool and modified our version to handle larger numbers of parallel profiles. This toolkit has no analysis phase and simply displays the performance to the user in a three-paneled Perspective using the following views.

The **System View** uses a Table Viewer to show a list of profile nodes (processor, process, or thread). Selecting a profile in the list notifies the SelectionService of a change in profile selection. A future implementation might mimic or extend the Machines View from the PTP Runtime Perspective [57], showing this list instead as a resizable grid of profile node icons with additional information displayed below the grid. The PTP Machines View has a number of other user interface features that could improve the System View.

The **Scope View** uses a Tree Viewer to display the source scope tree and the associated metric values for a selected profile. The view registers listeners for changes in scope or profile selection, to change scope focus or update the display with new profile metric values. Selecting or double-clicking a scope node in this view notifies
the Selection Service.

The Source View is launched once per sourcefile on a double-click selection of a File / Procedure / Loop / Statement Scope in the Scope View. It displays the appropriate sourcefile as a TableViewer where each row shows a numbered source code line. This view listens for changes in scope selection, scrolling to the newly-selected source region, if it is located in this view’s sourcefile.

The Performance Map View uses a custom Viewer subclassing the Composite widget to show a graphical representation of the Scope Tree. The custom viewer walks the scope tree and draws a proportionately sized block for each scope if the scope’s metric value is a non-trivial fraction of the whole, placing scopes at the same tree-depth in the same column. The blocks are currently colored by a four-color palette, though we realize that a future implementation could use different colors to convey different scope-types, variability levels, or other additional information. Clicking on a block in the performance map performs a lookup of the associated scope and notifies the Selection Service of a change in scope. This view itself listens for profile or metric selection changes, and updates the performance map to show the relevant data.

An improved HPCViewer Toolkit could perform basic statistical analysis across all profiles to generate an average profile and associated standard deviation values. We would also like to migrate the Callgraph tree views from the latest hpcviewer release to visualize callgraph data.

3.2.5 Cluster Toolkit Plug-in

The Cluster Toolkit converts the HPCViewer Experiment model to a two-dimensional matrix, which it then passes through one of multiple possible clustering analysis algorithms. The clustering results are represented by a list of Cluster objects which are then displayed to the user as a pair of interactive two-dimensional BIRT charts.
Data. HPCViewer Experiment data comes in a hierarchical scope-tree data structure, but cluster analysis performs better on a simpler, lower-dimensional data structure. For this reason, the Cluster Toolkit gathers all of the Experiment objects and transforms a set of profile scope trees (representing a collection of processes) into a two-dimensional PMatrix structure. The PMatrix data model walks down the entire scope tree adding a row to the matrix for each leaf node, storing one column for each profile value. It also records the scope names as row titles and lists of metrics and profile nodes as column titles. This data conversion should be abstracted into a separate PMatrix data model plug-in that would also enable the loading of flat matrix data files. After conversion, the completed performance matrix structure is returned to the Cluster Toolkit for analysis.

Analysis. Once the performance matrix is ready, it is passed to the user-specified clustering algorithm for analysis. This prototype includes a basic implementation of K-Means clustering and a partial implementation of the pCluster algorithm, but future plug-ins can contribute additional algorithms by implementing the provided interface. A clustering algorithm takes a PMatrix as input, searches for clusters, and returns the results in a ClusterData object, which hosts the list of Clusters and an interface to global cluster attribute data.

The included simple implementation of the K-Means algorithm [17] begins by setting k (user-defined) initial centroid values. Then, it clusters each data row around the closest centroid, and recalculates the centroid based on the cluster values. This repeats until the centroids become stable. The resulting KMeansClusterData provides access to each of the k KMeansClusters, as well as each cluster's name, size, sum, mean, and average row variance.

The second clustering module attempts to implement a more complex biclustering algorithm based on pCluster [66]. This algorithm seeks to cluster objects that exhibit similar patterns on a subset of dimensions. It uses a deterministic depth-first approach
for simultaneous subspace clustering that detects overlapping clusters and is resilient to outliers. First it finds row-pair and column-pair maximum dimension sets, then prunes these sets based on user-defined parameters for minimum number of rows and columns in a cluster, as well as a delta similarity threshold. Once pruning is complete, the maximum dimension sets paired on one dimension are inserted into a prefix tree structured around the other dimension. As the prefix tree is traversed in post-order, each node represents a candidate cluster of rows and columns. Each candidate cluster is pruned again and is added to the final cluster list if it remains within the minimum size bounds. The cluster members are then propagated up the tree to all valid subsets. When complete, the resulting PClusterData contains all valid delta-pClusters.

**Visualization** Once the Cluster Toolkit has finished its clustering analysis, it displays the ClusterData results in its own Perspective, starting with the Cluster List View, and then showing a pair of Cluster Chart Views for a user-selected Cluster.

The *Cluster List View* lists the discovered clusters in a TableViewer, displaying sortable columns for various cluster attributes, whose type and content depend on the algorithm or cluster type. These attributes could include the cluster name/id, size, cluster-mean, similarity score, percent variation explained, etc. Selecting a cluster in the cluster list notifies the Selection Service of a change in profile node selection, opening the Cluster Chart Views for that cluster.

The *Cluster Chart Views* use the BIRT Chart Engine [26] to display a boxplot-style or line-series chart for each dimension of the selected cluster. The datasets needed for the chart data are loaded directly from the ClusterData and Cluster objects. Each Chart View also adds a selection listener so that the chart data can be updated when the Cluster selection is changed.
3.2.6 Implementation Summary

In implementing the HPCVision prototype, we have sought to build a scalable, portable, and easy-to-use graphical performance analysis tool structured around a modular, extensible framework. Through the use of the Eclipse plug-in architecture, the HPCVision toolkit framework is designed to allow for easy extension of functionality through additional data, analysis, and visualization modules. Our prototype implementation provides a data model plug-in for HPCToolkit XML data and two toolkit plug-ins, one that allows interactive exploration of the HPCViewer performance data and one that performs statistical clustering analysis and displays the results using BIRT charts. The HPCViewer Toolkit provides a simple interface for examining the overall application performance on a system and navigating to important performance details. The Cluster Toolkit analyzes the parallel performance data to automatically discover patterns representing performance anomalies and equivalence classes of behavior. These tools help automate the application tuning cycle by assisting the human analyst in locating bottlenecks and directing the user to misbehaving source code regions.

3.3 Tool Summary

HPCVision is a tool for scalable performance problem identification developed to help automate the application tuning cycle and increase the productivity of the human analyst. This tool loads, analyzes, and visualizes parallel performance data to help the user locate performance bottlenecks, load imbalances, or other optimization opportunities in the application source code. The tool design and prototype implementation have been guided by design goals of portability, modularity, extensibility, scalability, and ease of use. To show that HPCVision can serve its performance analysis purpose while upholding these goals, it was field-tested on two case studies exemplifying different performance scenarios.
Chapter 4

Evaluation

In this chapter we apply HPCVision to performance data case studies for two well-known parallel application benchmarks and explore the performance results in search of interesting performance patterns. HPCVision’s success in these two performance analysis scenarios validates the tool design and demonstrates the effectiveness of the techniques used. Detailed demonstrations of HPCVision’s operation in practice illustrate its automated analysis procedure and simple, easy to use graphical user interface. This chapter concludes with a discussion of the strengths and weaknesses of the approach and imparts lessons learned through our implementation and analysis experiences.

4.1 ASCI Benchmark: Sweep3D

I first describe the ASCI Sweep3D benchmark and the parallel system and problem set on which it was executed. Then, I delve into an analysis of Sweep3D’s performance with HPCVision and discuss the insights into application performance obtained using HPCVision.

4.1.1 Sweep3D Program Description

The ASCI Sweep3D Benchmark [1] solves a one-group time-independent discrete ordinates (Sn) 3D Cartesian (XYZ) geometry neutron transport equation on an orthogonal mesh. The XYZ geometry is represented by an IJK logically rectangular grid of cells. The angular dependence is handled by discrete angles with a spherical
harmonics treatment for the scattering source. The solution involves two steps: the
streaming operator is solved by sweeps for each angle and the scattering operator is
solved iteratively.

The benchmark code is written almost entirely in Fortran77 and uses MPI for
inter-process communication. Figure 4.1 lists pseudocode that presents the basic flow
of the sweep method.

```
DO iq=1,8     ! octants
  DO mo=1,mmo
    DO kk=1,kb
      RECV E/W
      RECV N/S
      DO idia=1,jt+nk-1+mni-1 ! JK-diagonals with MMI pipelining
        DO jkm=1,ndia
          DO i=1,iti;ENDDO ! I-lines on this diagonal
          IF .NOT.do_fixups
            DO i=i0,i1,i2;ENDDO ! Sn equation
          ELSE
            DO i=i0,i1,i2;ENDDO ! Sn equation w/ fixups
          ENDIF
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
SEND E/W       ! send block I-outflows
SEND N/S       ! send block J-outflows
ENDDO
ENDDO

Figure 4.1 : Outline of Sweep3D Algorithm Flow.
```
Sweep3D exploits parallelism via a multidimensional wavefront process using a 2D spatial domain decomposition onto a 2D processor array in the I- and J-directions, so that each processor is assigned one columnar domain. To improve parallel efficiency, blocks of work are pipelined through this 2D processor array. Sweep3D uses message passing to communicate between processors as wavefronts propagate diagonally across this 3D space in eight directions. Thus, the wavefront exploits parallelism in both I- and J-directions simultaneously.

4.1.2 System and Problem Description

Sweep3D was executed on Rice University’s Terascale Cluster [62], a large Linux cluster of dual processor 900MHz Intel Itanium 2 nodes with 4GB of memory per node. This job was launched on 64 processors (one per node), over a Myrinet-2000 [8] network. It solved a 300x300x300 grid, decomposed onto an 8x8 processor array. This 64-processor execution was profiled using HPCToolkit to collect the PAPI_TOT_CYC metric, measuring total cycle counts for each source code line.

Sweep3D has been analyzed by others [13, 63, 14, 16] and its execution and performance behavior are well understood. This case study assesses HPCVision’s ability to easily and scalably uncover these well understood performance characteristics while also looking for any undiscovered performance issues. Following are the analysis results and discussion.

4.1.3 HPCVision Performance Analysis

After loading the Sweep3D data into HPCVision and selecting standard parameters for HPCViewer and K-Means analysis (k=15), the user is first presented with the HPCViewer Perspective. After analyzing the performance data in the HPCViewer Perspective, the user switches to the Cluster Perspective to explore the K-Means clustering analysis results.
Figure 4.2: Sweep3D Performance Overview using the HPCViewer Perspective.

Figure 4.3: Sweep3D: Scope Tree Performance Breakdown for Procedure sweep.
HPCViewer Analysis. Figure 4.2 shows the Application Performance Map View and corresponding Scope View for an example interior node (n26.rtc) in the Sweep3D data. Looking at the Performance Map, it is very clear that a large majority (83.4%) of the compute cycles were spent inside the sweep method, almost exclusively in the loop on lines 355-498, iterating through the I-lines for a diagonal sweep. Of the 16.6% spend outside of sweep, most was devoted to MPI communication calls and the rest fell to minor miscellany. In Figure 4.3, the Scope View is zoomed to show the breakdown of the cycle counts in the diagonal’s I-lines loop at lines 355-498. By double-clicking on any scope, the user is navigated to the associated source code. Relating the line numbers to the corresponding phases in the code yields the phase performance distribution shown in Table 4.1.

<table>
<thead>
<tr>
<th>%Cycles</th>
<th>Lines</th>
<th>Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.5%</td>
<td>385-389</td>
<td>Compute source from Pn moments</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(4.8% first loop, 13.7% second loop)</td>
</tr>
<tr>
<td>14.1%</td>
<td>398-409</td>
<td>I-line recursion without flux fixup</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.5% init, 11.9% balance, 1.9% diamond aux)</td>
</tr>
<tr>
<td>20.3%</td>
<td>417-468</td>
<td>I-line recursion with flux fixup</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1.2% init, 14.0% balance, 2.0% diamond aux, 2.4% ijk-fixup, 0.7% cleanup)</td>
</tr>
<tr>
<td>19.1%</td>
<td>475-479</td>
<td>Compute flux Pn moments</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(4.7% first loop, 14.4% second loop)</td>
</tr>
<tr>
<td>9.4%</td>
<td>487-491</td>
<td>Compute DSA face currents</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3.1% i, 0.0% j, 6.2% k)</td>
</tr>
</tbody>
</table>

Table 4.1 : Sweep3D Performance Phases (from sample profile on node n26).

After examining the performance distribution across the different phases of the sweep algorithm, certain regions stand out as prime candidates for exploring opti-
mization opportunities. A performance analyst would want to focus tuning efforts on areas likely to achieve the highest runtime improvement, such as the balance equation calculation in the I-line recursion phases, the computation of source from incoming Pn moments, and the computation of the outgoing flux Pn moments. It is important to note that since this data represents loop and line-level statistics for phases within the sweep method, typical procedure-level performance analysis would have been insufficient to provide these insights.

**K-Means Analysis.** In the Cluster Perspective, the user is presented with a list of (k=15) clusters found, along with basic cluster statistics. Sorting the list by cluster sum or mean values encourages the user to begin looking at clusters that account for the largest percentage of total execution time. In this example, we sort by cluster mean and examine some of the more significant clusters, mapping line numbers back to the corresponding algorithm phases discovered earlier. The list of clusters and corresponding statistics is shown in Figure 4.4.

![Cluster List View Table](image)

Figure 4.4: Sweep3D K-Means Cluster List (k=15).

The first cluster, Cluster (a) in Figure 4.5 with a mean value of 46.9 billion cycles, reveals that lines 389 (second loop of the source from Pn moments calculation) and 479
(second loop from the flux Pn moments calculation) in sweep.f have the highest mean line-level cycle counts. The performance distribution for these two lines across all processors is mostly regular, though there appear to be several higher-valued outliers for line 479 on seemingly random processors and a few lower-valued outliers for line 389 on other processors. Line 389 has a value range of 2.5 billion cycles (5% of this line’s mean, 0.8% of total) and line 479 has a range of 4.1 billion cycles (9% of this line, 1.2% of total), so this load imbalance may not require immediate investigation.

Figure 4.6 shows that the second cluster, with a mean value of 16.5 billion cycles, exhibits an extremely even distribution across all processors, implying that the source lines in this cluster are executed evenly across the system. These well-balanced source lines correspond to sweep phases for: the first loop for computing the source from Pn moments (385); balance equation computations during I-line recursion with (420,421) and without (401,402) flux fixup; the first loop of the flux Pn moments calculation (475); and the computation of the DSA face currents along the k dimension. These source regions perform consistently across all processors and indicate no opportunities for parallel optimization, though improving their serial performance could greatly improve the overall application performance due to the high cycle count values.

The next two clusters have mean values of 7.9 billion (Figure 4.7) and 3.4 billion cycles (Figure 4.8), and though the overall value distributions across processors displays significant imbalances, an examination of the variance per source region reveals that the majority of the imbalance can be attributed to MPI and Myrinet library calls. The apparent noisiness of the variance is likely due to contention on the Myrinet layer or other complex communication patterns created during the pipelined multi-dimensional wavefront passes. While the mean values of the sweep.f source lines in these clusters exist in the same range as the calls to the communication libraries, their variances are much lower. A performance analyst examining such clusters for potential tuning should focus on the communication library calls, since the sweep lines are well-balanced.
Figure 4.5: Sweep3D K-Means Cluster (a).

Figure 4.6: Sweep3D K-Means Cluster (b).
Figure 4.7: Sweep3D K-Means Cluster (c).

Figure 4.8: Sweep3D K-Means Cluster (d).
Here some of the limitations of the K-Means algorithm begin to surface. The communication calls and sweep code lines were clustered together based on the algorithm's similarity metric, calculated using mean values in this K-Means implementation. The problem of measuring distances between or comparing points in high-dimensional space [32] is a well-known issue in large-scale data analysis. As the number of dimensions grows, Euclidean distance becomes less and less relevant. The shape of these clusters suggests the use of a similarity measure that relies not just on source mean values but also on cross-processor variance.

The next set of clusters (Figures 4.9-4.13) exhibit an exciting common performance pattern. Since the data is decomposed onto an 8x8 processor array, the 6x6 grid of processors in the middle of the array have neighbors on all four sides to communicate with, while nodes on the edges only have three neighbors and corner nodes just have two. In these clusters, processors corresponding to corners and edges spend more cycles computing flux fixup or computing other boundary cases, while interior nodes skip over these sections and do more communication instead.

The variance in clusters (e) and (g) is split across i, j, and k fixup code, and the cross-processor distribution shows the lowest values for interior nodes and higher values for edge and corner nodes. Curiously, for line 443 in (e) and , the north-south edges have a W-shaped distribution, with the two innermost nodes (columns 4 and 5) of each row exhibiting higher values than the next most inner nodes (columns 3 and 6) in those rows.

Cluster (i) exhibits a similar W-shaped pattern for north-south edges, but yields a crescent-shaped curve overall, such that the four innermost rows have the lowest values and the north-south edges have the highest, while the second and seventh rows lie in the middle, along with all inner-row east-west edge nodes. Cluster (h) displays the same crescent curve, but the corners are not the highest-valued nodes in the north-south edges. Instead it is the two innermost nodes of the top and bottom rows that yield the maximum values in the cluster. The most variable source lines in both
Figure 4.9: Sweep3D K-Means Cluster (e).

Figure 4.10: Sweep3D K-Means Cluster (g).
Figure 4.11: Sweep3D K-Means Cluster (h).

Figure 4.12: Sweep3D K-Means Cluster (i).
Cluster (h) and (i) correspond to j fix-up calculations, so it seems likely that some aspect of the pipelined wavefront sweeps contributes to the rounded distribution in these clusters.

The final interesting cluster structured around the grid-based data decomposition is cluster (j), which exhibits clear processor partitions between north-south edges, east-west edges, and interior nodes. Examining the source lines with the most variance points to code that zero-initializes phiib and phijb if a block receives no east-west or north-south inflows, respectively, as well as code that calculates the i/j outflow leakage if there are no sends to perform in that direction. This makes perfect sense, because edge nodes need to perform these kinds of operations while interior nodes execute communication code instead.

The remaining clusters show a lot of noise due to the grouping of unrelated low-valued source regions. The last cluster in the list, Cluster (o), is home to all of the nearly-zero valued source lines. Although there may be some interesting anomalies
where a source line is zero-valued on all but one or a few processors, any source lines represented in these lower clusters account for such a small fraction of the overall performance that they are not likely to be worthy of optimization efforts.

**Sweep3D Analysis Summary.** We were able to use HPCVision to gain insight into the performance characteristics of the Sweep3D benchmark. The HPCViewer Toolkit correlates program-scope structure with performance metric values to enable the user to drill down the scope-tree to important source regions. In this case, the HPCViewer Perspective helped analyze loop and line-level statistics to provide the analyst with a fine-grained performance breakdown of the source regions where the program spent the most time. The Cluster Toolkit, on the other hand, groups similar valued regions together to assist the user in the discovery of significant performance patterns. For this application, the cluster analysis was able to distinguish sweep code that was well-balanced across all processors from communication calls that exhibited noisy variance across processors. The Cluster Perspective also helped the user identify equivalence classes for edge, corner, and interior nodes, not only finding the expected partitions for inflow and outflow boundary calculations, but also discovering that edge nodes spent more time in i/j/k fixup code. Exposing the user to these performance patterns can guide application tuning and help improve overall performance.

### 4.2 NAS Parallel Benchmark: LU

I first describe the NAS LU parallel benchmark and the parallel system and problem set on which it was executed. Then, I delve into an analysis of LU’s performance with HPCVision and discuss the insights into application performance obtained using HPCVision.
4.2.1 LU Program Description

The NAS Parallel Benchmarks are a small set of programs widely used in industry and academia to evaluate the performance of parallel supercomputers. Of the eight MPI-based programs distributed with NPB 2.3 [56], we have selected LU, a simulated computational fluid dynamics (CFD) application which uses symmetric successive over-relaxation (SSOR) to solve a block lower triangular - block upper triangular system of equations resulting from an unfactored implicit finite-difference discretization of the Navier-Stokes equations in three dimensions [6].

The LU benchmark code from NPB 2.3 is implemented in Fortran77 with a few common extensions that are also a part of Fortran 90. It uses MPI for inter-process communication. Figure 4.14 lists pseudocode that describes the algorithm flow for the main SSOR procedure.

```fortran
CALL RHS
DO ISTEP=1,ITMAX
  rsd()=dt*rsd()
  DO k=2,nz-1
    CALL JACLD
    CALL BLTS
  END DO
  DO k=nz-q,2,-1
    CALL JACU
    CALL BUTS
  END DO
  Update variables (u)
  CALL RHS
END DO
```

Figure 4.14: Outline of LU SSOR Algorithm Flow.

LU requires a power-of-two number of processors. A 2-D partitioning of the three-dimensional Cartesian grid onto processors occurs by halving the grid repeatedly in
the first two dimensions, alternately x and then y, until all power-of-two processors are assigned, resulting in vertical pencil-like grid partitions on the individual processors [6]. Each processor works on one tile at a time, using column-based relaxation for the SSOR procedure. Relaxation starts in an active corner point of the grid on the bottommost active grid plane, and progresses along each column in this processor’s bottom-plane tile in turn.

The first communication occurs only after all points on this processor’s bottom-plane tile have been relaxed. The row of values just computed for which i assumes its maximum on this tile are sent to the “eastern” neighbor. Likewise, the column of values for which j assumes its maximum on this tile are sent to the “northern” neighbor. These neighboring processors now begin to relax their points for the bottommost grid plane, and, simultaneously, the first processor proceeds to relax points on its tile on the next plane. Note that the ensuing pipeline is fully balanced, which means that once the pipeline is filled (i.e. the pencil on the north-eastern corner of the grid is reached), the load on all processors is completely balanced. [70]

4.2.2 System and Problem Description

LU was executed on the Rice Terascale Cluster, using 32 processors (one per node) on the Myrinet-2000 network. The test case was run on a class C problem with a grid of size 162x162x162. This grid was split across the 32 processors in 8 rows and 4 columns. The MPI ranks were assigned in reverse numerical order, so that node n87 is the NW corner, and n13 is the SE corner. The execution was profiled using HPCToolkit to collect the PAPI_TOT_CYC metric, measuring total cycle counts for each source code line.

LU is a common parallel benchmark application that has been used in numerous performance studies on a variety of platforms [27, 70, 68, 39]. The NPB 2.0 implementation was based on a diagonal wavefront scheme discovered to have significant load balance issues on non-vector machines due to excessive MPI communication of many
small messages. These issues were supposedly resolved in the recent column-based relaxation implementations. This case study uses HPCVision to assess communication overhead and load balancing issues in the NPB 2.3 reimplementation of LU. Following are the analysis results and discussion.

4.2.3 HPCVision Performance Analysis

After loading the LU performance data into HPCVision and selecting parameters for HPCViewer and K-Means analysis (k=25), the user is first presented with the HPCViewer Perspective. After analyzing the performance data in the HPCViewer Perspective, the user switches to the Cluster Perspective to explore the clustering analysis results.

Figure 4.15: LU Performance Overview using the HPCViewer Perspective.
HPCViewer Analysis. In Figure 4.15, the Performance Map View shows that a large majority of the time – between 81.2% on n13 (SE corner: row 8, col 4) and 92.4% on n49 (row 2, col 2) – is spent in the lu module, mostly split among five source files and their corresponding procedures: blts, buts, jacld, jacu, and rhs. The remainder of the time is split among MPI and Myrinet communication library calls, particularly gmpi_net_lookup, MPID_RecvComplete, MPID_CH_Check_incoming, and MPID_DeviceCheck. A user correlating program source information with data from the Scope View for a typical interior node like n26 (row 4, col 3) could generate the phase performance breakdown shown in Table 4.2.

Since the source regions using up the most compute cycles correspond to the phases of expected heavy computation, the program appears to be well-designed, though optimization of any of these heavy computation regions would certainly lead to noticable on-node performance improvements across the entire parallel execution. It is important to note, however, that 10-20% of the execution time is still spent on communication calls, polling different layers of the communication substrate for incoming messages. Communication overhead could be decreased if it is possible to overlap some of this communication wait time with unrelated computation, although the base algorithm may imply a data dependency chain that makes this impossible.

K-Means Analysis. In the Cluster Perspective, the user is presented with a list of (k=25) clusters found, along with basic cluster statistics. Sorting the list by cluster variance (here measured as the per-row cross-processor variance averaged across all cluster rows) encourages the user to begin looking at clusters that convey the largest cross-processor variances, implying load imbalances or performance partitions. In this analysis example, we examine some of the more significant clusters, mapping source line numbers back to the corresponding algorithm phases discovered earlier. The list of clusters and corresponding statistics is shown in Figure 4.16.
<table>
<thead>
<tr>
<th>%Cycles</th>
<th>Procedure</th>
<th>Lines</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.6%</td>
<td>libmpich</td>
<td></td>
<td>MPI and Myrinet (gm) library calls</td>
</tr>
<tr>
<td>2.7%</td>
<td>ssor</td>
<td></td>
<td>Symmetric successive over-relaxation solver</td>
</tr>
<tr>
<td>(1.1%)</td>
<td></td>
<td>110</td>
<td>SSOR iteration at beginning of timestep</td>
</tr>
<tr>
<td>(1.5%)</td>
<td></td>
<td>160</td>
<td>Update variables at end of timestep</td>
</tr>
<tr>
<td>31.5%</td>
<td>rhs</td>
<td></td>
<td>Compute the right hand sides between time steps</td>
</tr>
<tr>
<td>(1.4%)</td>
<td></td>
<td>42</td>
<td>Initialize rsd</td>
</tr>
<tr>
<td>(3.0%)</td>
<td></td>
<td>68-86</td>
<td>Xi-direction flux initialization, after communication</td>
</tr>
<tr>
<td>(4.2%)</td>
<td></td>
<td>88-202</td>
<td>Xi-direction flux differences, fourth-order dissipation</td>
</tr>
<tr>
<td>(2.0%)</td>
<td></td>
<td>223-240</td>
<td>Eta-direction flux initialization, after communication</td>
</tr>
<tr>
<td>(4.6%)</td>
<td></td>
<td>242-358</td>
<td>Eta-direction flux differences, fourth-order dissipation</td>
</tr>
<tr>
<td>(16.2%)</td>
<td></td>
<td>363-478</td>
<td>Zeta-direction flux differences, fourth-order dissipation</td>
</tr>
<tr>
<td>11.0%</td>
<td>jacld</td>
<td></td>
<td>Form the lower triangular part of the jacobian matrix</td>
</tr>
<tr>
<td>13.7%</td>
<td>blts</td>
<td></td>
<td>Solve the block lower triangular system</td>
</tr>
<tr>
<td>(1.8%)</td>
<td></td>
<td>62</td>
<td>Receive data from north and west, compute v</td>
</tr>
<tr>
<td>(11.8%)</td>
<td></td>
<td>74-250</td>
<td>Diagonal block inversion and back substitution</td>
</tr>
<tr>
<td>10.5%</td>
<td>jacu</td>
<td></td>
<td>Form the upper triangular part of the jacobian matrix</td>
</tr>
<tr>
<td>13.7%</td>
<td>buts</td>
<td></td>
<td>Solve the block upper triangular system</td>
</tr>
<tr>
<td>(1.5%)</td>
<td></td>
<td>61</td>
<td>Receive data from south and east, compute tv</td>
</tr>
<tr>
<td>(12.1%)</td>
<td></td>
<td>72-250</td>
<td>Diagonal block inversion and back substitution</td>
</tr>
</tbody>
</table>

Table 4.2: LU Performance Phases (from sample profile on node n26).

After sorting by variance, the first cluster (Figure 4.17) in the Cluster List is also the cluster with the highest mean value. The execution times for the solver code lines in this cluster are well-balanced, but execution time for `gmpi_net_lookup`, a call to poll the Myrinet interface for new messages, is spread in the range around 1 billion to 10 billion cycles, with the lowest value of 1.2 billion cycles coming from node n49.
An alert analyst will recall from the HPCViewer analysis that compared to the other processors, n49 spent the largest percentage (92.4%) of execution time in the lu module, and hence the least fraction of its time in communication modules. This seems somewhat odd for an interior processor in row 2 of 8, column 2 of 4. Examining other clusters, it becomes clear that n49 also holds the low value for MPID_RecvComplete (b), MPID_CH_CheckIncoming (c), and line 963 from gm ntoh u8 in gm.h (c). The first three functions poll communication channels for incoming messages, and gm ntoh u8 performs the endian conversion from the network to the host. Why doesn’t n49 need to wait as long for messages, and what is it spending its time on instead?

Besides holding the lowest values for these communication calls, node n49 also holds the highest values for a number of other significant functions involving socket buffer allocation (g,j), memory management (n,i,f), pthreads (i,p,r), the MPICH Device Interface (j,k), and MPI Sends (s,p). So, though node n49 is not spending as
Figure 4.17: LU K-Means Cluster (a).

Figure 4.18: LU K-Means Cluster (b).
much time waiting on communication messages, it is still actively sending and receiv-
ing messages, and for some reason spends more time than other processors performing
low-level memory and resource management.

This anomalous behavior does not appear to be bound to the data decomposi-
tion across the processor domain, and suggests a potential hardware-based anomaly.
These kinds of anomalies create inherent load imbalances, since this processor and its
neighbors are out of sync and they must wait on each other to synchronize at certain
stages before any of them can continue execution. Gathering additional information
from system logs, call-stack profiles, or debug output may provide additional insight
into the conditions surrounding the anomalous performance. The anomaly may dis-
appear in a repetition of the execution on different nodes, under different conditions,
or at a different time.

Moving beyond the n49 anomaly, when the human analyst examines cluster (d)
in Figure 4.30, a domain-based performance pattern becomes evident. Though its
performance is irregular across most processors, MPID_DeviceCheck holds markedly
lower values for all nodes on the north and south edges, except the northeast corner.
These north-south edges apparently spend less time checking the communication de-
vice for incoming messages, though the northeast corner (n20) is exempt from this
behavior. These imbalances may be effects of the uneven domain decomposition of a
square grid onto 32 nodes. Since there are twice as many rows as columns, each tile
is shorter in the xi-direction than in the eta-direction. The communication pipeline
starts from the southwest corner and reaches all of the columns in the south edge
before it reaches all of the rows in the west edge. Therefore, interior nodes will spend
more time waiting for north-south communication and the northeast corner would be
the last to see the wave from the southwest corner.

Also apparent in cluster (d) are the mirrored behavior patterns of ssor.f lines 108
(rsd adjustment i-loop) and 158 (u update i-loop). Line 108 has noticeably higher
values for nodes on the south edge, and lower values for the processor row just below
Figure 4.19: LU K-Means Cluster (c).

Figure 4.20: LU K-Means Cluster (f).
Figure 4.21: LU K-Means Cluster (g).

Figure 4.22: LU K-Means Cluster (h).
Figure 4.23: LU K-Means Cluster (i).

Figure 4.24: LU K-Means Cluster (j).
Figure 4.25: LU K-Means Cluster (k).

Figure 4.26: LU K-Means Cluster (n).
Figure 4.27: LU K-Means Cluster (p).

Figure 4.28: LU K-Means Cluster (r).
Figure 4.29: LU K-Means Cluster (s).

Figure 4.30: LU K-Means Cluster (d).
the north edge. Line 158 is the opposite, with low values on the south edge and high values on the row below the north edge. So, the more time is spent adjusting residuals before solving the triangular systems, the less time is spent updating the variables after the solution.

Continuing to other clusters, another pattern surfaces around the the source lines in procedure \texttt{rhs}. After exchanging data with neighbors and computing the xi-direction flux differences, each node must perform fourth order dissipation to smooth the solution. As shown in the Cluster Processor Views for related clusters (e,i,j,n), the north (e,i) and south (j,n) dissipation boundaries are computed almost exclusively on the north and south edges respectively. Since these boundary computations are surrounded by conditionals checking for the existence of north or south neighbors, this behavior is certainly expected. What is surprising is that the values for non-boundary nodes are not necessarily zero. Similar behavior is uncovered for the eta-direction fourth order dissipation boundaries for the western edge on line 316 (g) and the eastern edge on line 345 (e). Cluster (f) also shows somewhat higher values on the east-west edges for the eta-direction flux difference calculations, which seems counter-intuitive since these edges actually execute one fewer j-loop iteration than the nodes on the interior.

Examining the performance characteristics of the \texttt{exchange.1} and \texttt{exchange.3} procedures for neighbor data communication, expected edge patterns arise once again for each direction (h,k,l,t), e.g. code for communication with southern neighbors is zero-valued on the south edge. A few clusters (f,l,m) also show patterns where the south edge in particular exhibits higher cycle counts for forming the triangular parts of the jacobian matrix (\texttt{jac1d}, \texttt{jacu}) and computing the solution, even though it performs one less i-loop iteration than the interior rows. All of these cluster insights were easily discovered simply by scrolling through the cluster list and visually examining the charts for noticeable patterns.
Figure 4.31: LU K-Means Cluster (e).

Figure 4.32: LU K-Means Cluster (l).
Figure 4.33: LU K-Means Cluster (t).

Figure 4.34: LU K-Means Cluster (m).
LU Analysis Summary. In this section, we used HPCVision to gain insight into the performance characteristics of the NAS LU benchmark. The HPCViewer Toolkit exposes general application-wide performance characteristics and identifies which code regions dominate the majority of the execution time, here revealing a communication overhead of 10-20% per processor and itemizing the most influential communication procedures. The Cluster Toolkit, on the other hand, groups similar valued regions together to assist the user in the discovery of significant performance patterns. After examining the cluster analysis results for this LU test case, we discovered an anomalous node (n49) that spent less time than others waiting for incoming data messages and more time in various other communication and resource management procedures. The cluster charts also identified expected equivalence classes for edge and interior nodes and exposed unexpected load imbalance patterns for SSOR code surrounding the triangular system solutions as well as eta-direction code for flux difference calculation. Further investigation into the causes of the communication overhead, anomalous node behavior, and load imbalances could yield significant optimization opportunities and improve application performance.

4.3 Discussion

We have applied the HPCVision performance analysis approach to two application case studies to verify the efficacy of the tool's design, implementation, and approach. HPCVision seeks to scalably analyze parallel performance data to provide the user with insight into application structure and performance. It examines similarities and differences in on-node performance among an ensemble of processors to assist the user in identifying general performance characteristics, load imbalances and performance partitions, and performance anomalies. After helping the user detect such performance issues, the tool directs the user to the source regions at the root of problem, thereby automating the application tuning cycle and increasing the productivity of the human analyst.
In the above application case studies, HPCVision was successfully able to identify general application-wide performance characteristics and uncover various kinds of load imbalances. The HPCViewer Toolkit provided a fine-grained performance breakdown that emphasized the dominating code regions and separated the performance due to main algorithm computations from the overhead of communication library calls. The Cluster Toolkit grouped source regions by their performance and examined the cross-processor variance for significant source regions in an attempt to uncover important load imbalances. When applied to the case studies, this analysis was able to uncover performance patterns that demonstrated uneven communication distribution, verified expected equivalence classes for edge, corner, and interior processors, revealed unexpected performance partitions for major computation regions, and identified the anomalous behavior of a single processor.

HPCVision automates performance analysis and provides an intuitive user interface to guide the human analyst from performance data to performance tuning. HPCVision’s automated analysis techniques give the user insight into system and application behavior by revealing notable performance patterns. The HPCViewer Perspective provides a simple overview of the application performance across the entire system, but still make it easy to navigate to fine-grained source-level statistics for specific profiles. The Cluster Perspective logically directs the user to the most significant clusters found and supports easy graphical exploration of cross-processor performance distribution in source region clusters. Once the user has applied the tool to identify performance patterns or issues, it points back to relevant source regions for further investigation and performance tuning. Thus, HPCVision automates the application tuning cycle and increases the productivity of the human analyst.

The rest of this section discusses the open issues in our approach and shares the lessons learned through our experiences. Discussion is divided into topics on data types and data management, clustering analysis algorithms, user interface issues, scalability, and extensibility.
4.3.1 Data Discussion

For these experiments we measured the metric for total cycles and stored the sample counts in HPCToolkit flat profiles. We made minor modifications to the file format to add system node names for each profile node so that the user interface would be able to identify each profile by an informative name. By collecting other performance metrics (cache misses, floating point instructions, etc.) and examining the correlation between multiple metrics across processors, an analyst may be able to gain additional insight into application performance. Though HPCVision can accept profiles containing different performance metrics, cluster analysis behavior for examining multiple metrics is as of yet undefined, and the graphical user interface currently assumes a single performance metric per processor. Adding support for call-graph profiles or tracefiles would add another dimension to the data, providing the analyst with information about where, when, and why certain functions were called, thereby simplifying the process of tracking performance issues down to the root cause. However, adding more information to the performance data inflates its size and increases the complexity of data analysis.

Feature selection and feature extraction are common dimensionality reduction techniques that can reduce the size and complexity of performance data before analysis. HPCVision takes a very basic feature selection approach, simply filtering out source regions with the lowest average values below some threshold. Though this threshold is currently static or user-defined, an improved approach could automatically select a filter threshold that would remove a certain number or percent of all values, or set the threshold to a percentage of the total application value. Another method for reducing the number of source lines passed into the clustering analysis toolkit would involve aggregating some of the lower-valued line-level scopes up to loop, procedure, or file scopes. This frontier selection could be performed by the user in an extended scope-tree interface, or simply automated by adding a scope-tree node itself to the analysis matrix instead of its children if its average or maximum
value falls below a certain threshold percentage of the total. In the case of a profile with multiple metrics, since some metrics are highly correlated and any one may be representative of an entire group, the user should be able to select if all or only some of the metrics are to be involved in the performance analysis. More advanced feature extraction techniques like Principal Component Analysis (PCA), Singular Value Decomposition (SVD), and Factor Analysis can drastically reduce the data dimensionality while preserving much of the variance, but these approaches are more complicated to implement and are left as possible future work.

4.3.2 Clustering Analysis

The current prototype tool uses a basic K-Means clustering algorithm to group performance data by source regions with similar mean values across processors. Though this analysis finds groups of similar performance data, the clusters found are not necessarily optimal. Items found in one cluster may more rightly belong in another cluster, and just because items group together in a cluster, this does not necessarily mean that they or related or that they exhibit good performance.

The K-Means algorithm is not guaranteed to return a global optimum, but varying the selection of initial set of clusters or altering the distance metric used can yield different or improved cluster solutions. The k initial sets can be selected by setting the initial centroid values to random data points or by randomly partitioning the data into k even sets. Some tools run multiple random trials and present the user with a choice of the best (according to some similarity criteria) solutions found. The HPCVision prototype merely initializes the centroids to random data points in the data range and reinitializes centroids for empty sets by splitting the biggest cluster. A more flexible approach that runs multiple trials for multiple values of k could enable the user to identify the most appropriate clustering solution and improve the analysis results.

Selection of an appropriate distance or similarity metric is of utmost importance
to any clustering algorithm. Although our K-Means prototype groups source regions by their mean values across all processors, other algorithm implementations use Manhattan, Euclidean, or other distance metrics. However, extending any of these metrics to high-dimensional space invokes the curse of dimensionality and brings into question the quality of the distance measure in such a large, sparse space [32]. Fortunately, if each of the dimensions represents a unique performance profile, then each dimension should hold equal weight and common statistical measures may prevail. Ideally, the clustering analysis would be able to group source regions not just by mean value or Euclidean distance, but by coherent cross-processor distribution patterns, accounting for scaling and inversion. Thus, the analysis could group source regions together that exhibit common patterns relating to edge communication and boundary computation or other correlated performance partitions and anomalous behavior.

HPCVision is designed to support any number of one-way or two-way clustering algorithms, but so far only the K-Means algorithm implementation provides meaningful results. Relying only on pseudocode and explanations from the paper by Wang et al. [66], I attempted an implementation of the pCluster algorithm to simultaneously find overlapping, nonexclusive clusters based on pattern similarity. However, a number of implementation issues have arisen, and the current version produces very uninteresting results, showing numerous overlapping clusters on pairs of processors that often only vary by a single source line. Current work involves upgrading the postorder prefix tree traversal code to dynamically handle the data changes from the final phase of algorithm that propagates candidate clusters up the tree. Once fully functional and configured with intelligent values for the $\delta$-threshold and minimum cluster size parameters, the pCluster algorithm should be able to analyze parallel performance data to discover coherent performance patterns for groups of source regions across groups of processors. It would also be worthwhile to further investigate other clustering algorithms and the approaches used by other tools.
4.3.3 User Interface

HPCVision provides two data visualization perspectives in a familiar Eclipse-style user interface, enabling intuitive interactive navigation between data in different views and perspectives. Though these perspectives provide a straightforward, logical method for the exploration of performance data and the discovery of performance issues, this program, like all GUI applications, could benefit from user interface improvements that add functionality or automate user tasks.

The HPCViewer Perspective provides a three-paneled interface for exploring performance data from HPCToolkit profiles. The Scope View shows a program scope tree with associated metric values for each metric in a particular profile. It supports scope zooming and double-click navigation to source regions in the Source View. Though viewing individual profile values is informative, adding another column for average profile values or side-by-side profile comparisons would really help fit individual profiles into the global perspective and expose anomalous source regions. Once call-graph information is supported, the Scope View area could be expanded to show a calling context view and callers view in addition to the current flat profile view.

The Performance Map View provides a general application overview and facilitates navigation to more detailed performance information. This view could also benefit from many of the same UI features as the Scope View, such as zooming, flattening, and scrolling, as well as improved color information representing scope significance, cross-profile variance, or performance metric values.

The System View is a simple list of node names, but could be improved by also listing additional node information such as hardware and OS platforms, process and thread IDs, or total node-wide statistical data. This information could be isolated to a separate System Node Info View, while the System View itself is migrated to a more informative format such as a colored topology map based on the data distribution pattern or a system structure tree hierarchically organized by racks, nodes, processors,
and processes.

Currently, navigation between perspectives is restricted to selecting the appropriate tab from the Perspective Bar, but the Cluster Toolkit could benefit from being launched directly from the HPCViewer Toolkit, where selected profile nodes and scope regions could focus the clustering analysis on specific data.

**The Cluster Perspective** lists the clusters found and then displays charts showing cluster membership and value distribution for each dimension of a selected cluster. The Cluster List shows for each K-Means cluster the cluster size, the sum of row means, the overall mean value, and the average cross-processor variance of the cluster rows. Other cluster statistics may provide additional insight or better ranking criteria, but these have proven useful in practice.

The Cluster Processor Chart shows bars representing the range of cluster values for each processor and a line series of the mean values for each processor. In the prototype, this is overlayed with a line series for each of the three cluster source regions with the largest value ranges. Though very informative for the case studies, these source line overlays can clutter up the chart and would be most useful if the user could select which source lines to show or hide dynamically while exploring the analysis results. It would also be instructive if hovering over or clicking on individual data points revealed their corresponding values in a tooltip box nearby.

The Cluster Source Chart simply shows data points for the minimum, mean, and maximum values for each source line in the cluster. Line series do not make sense for this chart, since the source lines are essentially categorical values with no guaranteed linear relationship. A more reasonable visualization upgrade would be to express the cross-processor value distribution for each source line as a boxplot-style diagram, using the spacing between the different parts of the box to indicate variance, skew, and outliers. As the size of these clusters increases, this chart becomes cluttered with too many source lines and too many data points. One proposed method for reducing
this clutter would involve hiding source lines whose value-range or variance falls below some threshold, leaving only the high-variance lines for the user to investigate. Once a user has found an interesting source region, the Cluster Perspective should provide a way to direct the user back to the actual source code, either in HPCViewer’s Source View or an editor in an Eclipse IDE perspective.

The application case studies showed that a user can gain performance insight from the Cluster Perspective and discover interesting performance patterns. However, many of the patterns discovered were split across multiple clusters, and the human analyst was required to correlate these disconnected patterns by hand. In fact, the K-Means analysis simply separates source regions into groups with similar mean values, paying no attention to cross-processor variance patterns. Hopefully, improved clustering algorithms can find better clusters grouped around coherent performance patterns. The analysis results also suggest that scope structure information could be exploited to logically group related source regions in the user display to assist understanding and ease tuning efforts.

Though the current cluster charts are easily able to display biclusters that only exist on a subset of processors, to help the user understand bicluster patterns the display should somehow convey both intrACLuster and intercluster variance for individual source lines. When visualizing data involving larger numbers of processors, these charts may not scale well beyond a hundred processors or a few dozen source lines. While techniques like feature selection and filtering out low-variance source lines have already been suggested for reducing source line clutter, and similar techniques may be useful for handling large numbers of processors, it may be more logical to switch the Cluster Processor Chart to a colored topological map that concisely displays the performance distribution across the system.

Overall the HPCVision toolkits are rather straightforward and easy-to-use, and they can help automate the analysis procedure that sits between performance data and performance tuning. The current user interface is noticeably bare and could
benefit from a number of UI improvements. Though it was successful in handling performance data for up to 64 processors, the tool visualization will need significant modifications before it can handle hundreds or thousands of processors.

4.3.4 Scalability

One-way clustering techniques such as K-Means can only reduce performance data on one dimension: source regions or processors. Visualizing analysis results from such techniques becomes impossible when the dimension not being clustered grows to a massive scale. Using two-way clustering analysis, or biclustering, enables a tool to discover patterns within groups of source regions and groups of processors. By reducing each dimension down to an interesting subset displaying similar values or similar patterns within the cluster, scalable interpretation and visualization of performance data becomes much easier. A useful scalable overview display would take each cluster discovered and display one line or bar on each 2D chart (or on each independent axis on a 3D chart) for the average values of each cluster, and then enable the user to drill down into individual clusters of interest. Grouping values within a cluster into to equivalence classes of behavior could also reduce the complexity of cluster charts and aid the user’s comprehension.

4.3.5 Extensibility

Implementation of this tool as an extensible Eclipse Rich Client Application has required an initially steep learning curve, but once the basics of the platform were understood many more technical aspects fell into place. One of the most promising features of the Eclipse Platform, extension-points, is also one of the most poorly documented, but once it is integrated into HPCVision, it promises to provide an extremely high degree of modularity and extensibility to the tool design.

Updating HPCVision’s modular design to take advantage of extension-points would facilitate the integration of new data models or analysis algorithms already
implemented in other languages or external libraries. The new design would also be easier to integrate with the Eclipse IDE and Parallel Tools Platform. Imagine the new development cycle: write and compile code in the Eclipse IDE, launch and monitor a parallel job through the PTP Runtime Perspective, load and analyze the performance profiles in the HPCVision Perspectives, and find a performance issue that automatically opens up the relevant source code in the IDE. Eclipse IDE integration would provide an easy and familiar editor interface, and help automate the application tuning cycle by reducing the number of steps between performance problem discovery and the compilation and running of tuned code.

4.4 Evaluation Summary

We have applied the HPCVision performance analysis approach to two application case studies to verify the tool design and demonstrate the effectiveness of the techniques used, specifically illustrating the automated analysis procedure and simple, easy-to-use graphical user interface.

Though we have uncovered several open issues in our approach during implementation and evaluation, the evaluation of the HPCVision process on the above case studies has proven that the tool we designed can scalably analyze large-scale parallel performance data to automatically discover interesting load balance patterns that identify general application performance characteristics, performance partitions and equivalence classes, and anomalous node behavior.

In each case study scenario, the HPCViewer Toolkit's straightforward interface for interactive performance exploration instinctively led the user from a broad performance overview to a fine-grained performance breakdown emphasizing the dominating code regions, easily distinguishing main algorithm performance from communication overhead.

The Cluster Toolkit successfully utilized K-Means clustering analysis to automatically discover performance patterns representing significant cross-processor load
balance issues. The clustering analysis examines similarities and differences in performance among an ensemble of processors and reduces massive amounts of data to representative groups and interesting subsets. The K-Means algorithm groups source regions by their measured performance so the user can examine the cross-processor variance per cluster in the chart views. After sorting the resulting cluster list by relevant statistics, the user is logically directed to examine the most significant clusters first. Through attentive exploration of the cluster charts, the human analyst was able to uncover performance patterns in cross-processor variance that demonstrated uneven communication distribution; verified expected equivalence classes for edge, corner, and interior processors; revealed unexpected performance partitions for major computation regions; and identified the anomalous behavior of a single processor.

Through evaluation by example, we have shown how easily HPCVision can scalably analyze parallel performance data to provide insight into application structure and performance. HPCVision provides an intuitive user interface to guide the human analyst from a general performance overview to the discovery of specific performance issues, and once a performance issue is discovered, the tool directs the user to relevant source regions for further investigation and performance tuning. The synthesis of these techniques constitutes a performance analysis approach that automates the application tuning cycle and increases the productivity of the human analyst.
Chapter 5

Conclusions and Future Work

HPCVision is a scalable, extensible tool framework for automatic parallel performance problem identification. Since current performance tools are insufficient for useful analysis and visualization of large-scale performance data, we have designed HPCVision using scalable analysis and visualization techniques to help automate the application tuning cycle and increase the productivity of the human analyst.

For this thesis, the HPCVision prototype was implemented to assess three key aspects of our performance tool development approach: (1) applying statistical clustering analysis to parallel performance profiles to easily and scalably discover patterns; (2) visualizing the performance data and analysis results in an intuitive, interactive display that assists the human analyst in locating performance bottlenecks; and (3) building a portable performance tool framework based on the Eclipse Rich Client Platform and evaluating this strategy with custom data, analysis, and visualization modules.

The HPCVision framework is a collection of data, analysis, and visualization plug-in modules. The prototype plug-ins support one performance data model and two performance toolkits. The data model plug-in populates internal data structures from an HPCToolkit XML profile containing performance measures for each processor thread from a parallel execution of a SPMD application. After some basic dimensionality reduction, these data structures are passed to the performance toolkits for post-mortem analysis and visualization.

The HPCViewer Toolkit provides a straightforward interface for interactive performance exploration, beginning by examining overall application performance on a
parallel system and then navigating through the program scope to important performance details.

The Cluster Toolkit performs statistical clustering analysis on the performance data and displays each resulting cluster in a pair of illustrative charts. The clustering analysis examines similarities and differences in performance among an ensemble of processors and reduces massive amounts of data to representative groups and interesting subsets. The K-Means clustering implementation groups source regions by their measured performance so the user can examine the cross-processor variance in the chart views. Sorting the list of results by cluster statistics logically directs the user to the most significant clusters first.

We applied the HPCVision performance analysis approach to two application case studies to verify the tool design and demonstrate the effectiveness of the techniques used, specifically illustrating the automated analysis procedure and simple, easy-to-use graphical user interface. During evaluation testing, the HPCViewer analysis instinctively led the user from a broad performance overview to a fine-grained performance breakdown emphasizing the dominating code regions, even helping the user distinguish main algorithm performance from communication overhead. The case study evaluation also showed that the clustering analysis approach was very successful in automatically discovering performance patterns representing significant load balance issues. The Cluster Toolkit was able to uncover cross-processor variance patterns demonstrating uneven communication distribution, expected equivalence classes for edge, corner, and interior processors, unexpected performance partitions for major computation regions, and anomalous behavior of a single processor.

HPCVision scalably analyzes and visualizes parallel performance data to help the human user gain insight into application structure and performance. To deliver such insights, the program outlines general application performance characteristics, automatically identifies performance issues, and guides the user to correct these performance problems. Understanding common characteristics in system or application
performance aids in the discovery of general application performance issues by locating the parts of the program where most processors spend large amounts of time and resources. Since a well-behaved SPMD program should exhibit little variation across processor profiles for most scopes, many common performance issues can be identified by locating source regions that display significant cross-processor performance variance in the form of performance partitions or anomalies. Clean partitions in the processor space (between groups of performance profiles) can identify load imbalances due to system or data irregularity, as well as equivalence classes created by the program structure. The discovery of performance anomalies where one or a few processors behave differently from all the rest can help a user pinpoint application bugs, failures, or systemic anomalies. Once performance issues have been identified, they should be corrected or improved, so the tool directs the user to relevant source regions for further investigation and performance tuning. The entire process naturally guides the human analyst from massive performance data to detailed performance tuning.

5.1 Future Work

While implementing and evaluating HPCVision, we uncovered several issues with our approach to large-scale performance data, statistical clustering analysis, user interface design, and software extensibility. These issues suggest many areas for improvement and opportunities for future work.

Continuing development will involve some basic enhancements to the current prototype, including the implementation of extension points and improvements to the clustering analysis and user interface. By abstracting much of the plug-in module functionality into extension-point definitions and extensions, HPCVision will become much more flexible, modular, and transparent, and it will be able to take advantage of dynamic classloading to minimize the memory load caused by a large assortment of potential plug-ins. Basic clustering analysis improvements include tweaks to parameters and methods of the K-Means algorithm, investigation into alternative distance
metrics for high-dimensional data, debugging and optimization of the pCluster algorithm, and experiments in parameter auto-discovery. The HPCViewer user interface could benefit from the following minor changes: an improved, scalable System View; better use of color in the Performance Map (perhaps indicating high, low, or normal performance per block); new Scope View features like columns for average profile values and other global statistics for comparison, a GUI input model for derived metrics, and better support for other hpcviewer features. The Cluster Toolkit desperately needs chart interactivity, so that users can select specific source lines or data points to dynamically alter the appearance of the charts. There is also a memory leak in the Cluster View code for creating BIRT charts that needs to be eradicated.

Beyond these basic improvements, we are also pursuing several directions of related research in data, analysis, visualization, and design and distribution. Extended data management research suggests a kind of Data Source Explorer interface for discovering and loading a wide variety of different data sources including remote files, databases, and dynamic data streams. Before delving into the realm of very different datatypes, we will first look at upgrading HPCVision to support call-stack profiles and profiles with multiple metrics. To improve the performance of our more sophisticated analysis techniques, we will investigate methods for dimensionality reduction, including data normalization, weighting, and filtering. We have already begun to explore a feature reduction technique based on "source frontiers" that aggregates metric data below a certain value threshold from lower-level program scopes up to a higher level in the program scope tree, so that insignificant values can be amassed into a single, larger value for a file, procedure, or loop instead of swamping the analysis module with trivial line-level data. Preliminary experiments have demonstrated a tunable decrease in data size and a possible improvement in analysis results as aggregated data appears in nontrivial clusters exhibiting significant variation, therefore suggesting potential load imbalances at the procedure-level. Future analysis research would involve integrating additional clustering and/or bichustering algorithms, per-
haps most easily accomplished by supporting the assimilation of external third-party analysis tools and libraries. For additional data visualizations, we propose the following: a 3D cluster map overview of the entire sorted performance matrix with the discovered clusters highlighted and clickable; a system topology map colored by cluster membership or similarity score; silhouette plots demonstrating how well-separated each cluster is from the rest; and a filtered HPCViewer perspective that focuses on the profiles and scope regions from a particular cluster. When refactoring and re-designing HPCVision, we will consider implementing the cluster analysis and other complex calculations as parallel or multi-threaded algorithms. Though HPCVision is currently distributed as a standalone RCP application, its modular plug-in design prepares it for integration with the Eclipse IDE and Parallel Tools projects, an improvement that would narrow the application tuning cycle and simplify the work of the human analyst.

Longer range projects for future work include data models for communication modeling with tracefile data; analysis modules for Principal Component Analysis, supervised classification, or cross-execution scalability analysis; and advanced toolkit plug-ins for Grid and distributed systems, performance modeling and cross-execution comparison, and automated tuning. Any of these ideas would be interesting research topics and worthy projects for investigation.
Bibliography


