

THESIS

3

2004

5662 1209



This is to certify that the
dissertation entitled

EVALUATING PERFORMANCE INFORMATION FOR
MAPPING ALGORITHMS TO ADVANCED ARCHITECTURES

presented by

Nayda G. Santiago Santiago

has been accepted towards fulfillment
of the requirements for the

Ph.D. degree in Electrical Engineering

A handwritten signature in dark ink, appearing to read "Dennis Horn", is written over a horizontal line.

Major Professor's Signature

7/17/03

Date

PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.
MAY BE RECALLED with earlier due date if requested.

DATE DUE	DATE DUE	DATE DUE

**EVALUATING PERFORMANCE INFORMATION FOR
MAPPING ALGORITHMS TO ADVANCED
ARCHITECTURES**

By

Nayda G. Santiago Santiago

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering

2003

ABSTRACT

**EVALUATING PERFORMANCE INFORMATION FOR
MAPPING ALGORITHMS TO ADVANCED
ARCHITECTURES**

By

Nayda G. Santiago Santiago

The development of efficient code for scientific and engineering applications on advanced computing systems is not a trivial task. To accomplish this task, a code developer has to be concerned not only about algorithmic correctness and robustness, but also about performance and implementation details. These additional factors impose a burden on the typical scientific computing expert, preventing the user from effectively leveraging the computational resources available to the application. Two major factors can be identified among those making this task particularly difficult. First, the complex interactions between the target platform and the application software tend to hide information about the existing relations between different entities in the system. Second, the high dimensionality of the performance data conceals interesting patterns in the observations which could lead to insights into the system behavior. While a multiplicity of tools have been developed to solve these problems, many obstacles still exist when characterizing the relations among high-level factors and low-level performance information. These problems not only make difficult the task of efficient coding, but also prevent the development of automated performance analysis tools to assist application programmers to tune their code.

This dissertation proposes a new methodology for obtaining information about the relations emerging when compute-intensive applications are mapped onto advanced architectures. The proposed methodology incorporates knowledge and techniques from multiple areas that include statistics, operational research, pattern recognition, data mining, and performance evaluation to enable the extraction of performance information during the mapping process. The methodology is composed of four steps: *problem analysis*, *design of experiments*, *data collection*, and *data analysis*. In the first two steps, analyses of the application itself are completed to determine the appropriate design of experiments for establishing relations between changes in high-level abstractions and performance outcomes. Feature subset selection is proposed for identifying important system metrics. An evaluation of different statistical analysis alternatives was carried out to characterize the types of data obtained in performance studies.

Several interesting results emerged from the application of this methodology on a computational electromagnetic case study. First, a correlation analysis embedded in the proposed methodology revealed that software instrumentation metrics exhibit collinearity. This implies a redundant information content in the data, limiting the set of statistical methods applicable for its analysis. Intrinsic dimensionality estimation and unsupervised feature subset selection identified the metrics containing the most performance information. On average, only 18% of the metrics were found to be important. Other results include identification of equivalency between multiple compiler options, reducing the actual set of options necessary at compile time. Also, a categorization of these options was obtained according to their effect in the application execution time. In summary, the application of the proposed methodology reveals that a detailed problem study preceding a systematic design of experiments, yields useful data on which appropriate statistical tools can provide unbiased information about the application-system interactions. Moreover the information obtained from this methodology can be converted into appropriate suggestions, observations, and guidelines for the scientific computing expert to tune applications to a particular computing system.

Copyright © by

Nayda G. Santiago Santiago

2003

To my family: Diana Alexandra, Víctor Manuel, and Manuel.
To my parents Héctor and Icsida, and to my sisters Yaira, Damaris, and Betzaida.

ACKNOWLEDGMENTS

I have been at Michigan State University for many years. Enough to get adjusted and learn to appreciate and enjoy living in Michigan. There have been many people that have made this transition process much more enjoyable.

First of all, I would like to express my gratitude to Diane T. Rover. I am completing this degree because of her and her constant encouragement and support. She has been my stronger supporter for all these years. She was always finding ways to motivate me, alternatives to solve the problems along the way, and has been advisor and friend. I have learned so much from her ... I still cannot figure out how she has so much energy and how can she always manage to have time for everything.

I would like to thank my committee members John R. Deller, Jr., Michael Frazier, Richard Enbody, and Domingo Rodríguez for their time and effort to review this document and their insights in the development of this research work. I specially would like to thank Michael Frazier. I wish I had his ability to convey information to students. I would be well off even if I were half as good as him as a professor. Also, former committee member, Robert Nowak, provided a lot of guidance while he was professor at Michigan State University. Shawn Hunt represented Domingo Rodríguez in the dissertation defense and provided useful comments on the dissertation.

Domingo Rodríguez deserves special thanks. He has been a friend and mentor for many years and an advisor for the last part of my dissertation. He has taken me as his graduate student and provided resources, energy and motivation for my research. Plus he has been my support and friend when things were not going right. Thanks Domingo, from the bottom of my heart.

Leo Kempel's assistance has been very important in the completion of my dissertation. He has provided all the resources and code for my experiments. He was always willing to help whenever we needed something or when we just needed an explanation.

The people who worked at the Scalable Computing Systems Lab were always my friends and partners and they deserve my appreciation and gratitude. These are Ken Wright, Sandeep Rao, Srinivas Kanamata, Timo Vogt, Sharad Kumar, and Vijay Kesavan. Vijay has been more than a partner, he has been my sounding board and my soul mate. I am profoundly grateful to you for being always there for me.

I thank Jeff Meese for providing all the time and effort to keep the system working and installing the software for my experiments. Kennie J. Cruz, Pablo J. Rebollo, Iomar Vargas, and Iván David have provided the technical assistance to keep the system working at the University of Puerto Rico. They have worked extra hours to assist me in anything they could do for me.

I would like to thank the ECE Department Staff for all their dedication. In particular I thank Marylin Shriver, the former graduate secretary, who has always been friendly and helpful to me and Vanessa Mitchner for assisting me many times with paperwork. I would like to thank Barbara O'Kelly and Percy Pierre, from the Sloan Engineering Program at MSU, for all their assistance all these years.

This work was supported by the following grants: NSF EIA-9700732, NSF ACI-9624149, and NSF EIA-9977071. I would like to thank Susan Kingston and Don Gunning from INTEL for their assistance with KAP/Pro and also Dr. William Kent of Mission Research Corporation for his support.

My friends have been my moral supporters all along. I thank Ziad Youssfi, María de los Angeles Torres, Andrés Díaz, Brenda Ortiz, Oscar Hernández, Ron Wright, Freddy Pérez, Amarilis Cuaresma, Hilaura Nava, Daniel Burbano, Judy Rosado, and Gihan Mandour. Ziad Youssfi is an unconditional friend and a wonderful human being. Hilaura has given me the strength I needed when I was in trouble. Gihan deserves special thanks. She has laughed and cried with me all along and is my soul mate. She is one of the special friends

that has been there for me, always..., no matter what. Thanks!

I would like to thank the Congregation of Sisters of Charity of the Sancha Cardinal (Hermanas de la Caridad del Cardenal Sancha - HCCS) at Santo Domingo, Dominican Republic for their constant prayers. Their prayers kept my faith strong along the way.

I want to thank family. My sisters, Bechi, Damaris, and Yari (Sor Yaira), have always prayed for me and given me encouragement and assisted me with all they could do. My mom, Icsida Santiago, is my role model and she is one of the strongest woman I have ever known, not only in character, but also in faith to God. My dad, Héctor Santiago, is one of the nicest people on this world. My husband, Manuel A. Jiménez, has been there with me all along and supported me 100%. Finally, my children Víctor Manuel and Diana Alexandra who are my motivation and joy. This dissertation is dedicated to you all since you are my strength and motivation in life.

I want to thank *God*. Without him, nothing is possible.

TABLE OF CONTENTS

LIST OF TABLES	xiv
LIST OF FIGURES	xvi
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	3
1.3 A Methodology for Evaluating Performance Information	4
1.4 Contributions	7
1.5 Dissertation Overview	8
2 Related Work	9
2.1 Introduction	9
2.2 Relating Performance Information and High-Level Abstractions	10
2.3 Using Statistical Analysis on Performance Data	12
2.3.1 Statistical Analysis of Algorithms and Heuristics	12
2.3.2 Scalability Analysis using Factorial Designs.	13
2.3.3 Statistical Analysis of Memory Hierarchy	14
2.4 Multivariate Methods for Performance Data Analysis	15
2.5 Automatic Performance Evaluation	18
2.6 Summary	20
3 Proposed Methodology	21
3.1 Introduction	21
3.2 Preliminary Problem Analysis	22
3.3 Experiment Specification	22
3.4 Data Collection	24
3.5 Data Analysis	24
3.6 Summary	26
4 Preliminary Problem Analysis	28
4.1 Introduction	28
4.2 Problem and System Definition	28

8.1.3	Dimensionality	92
8.1.4	Metric Selection	95
8.1.5	ANOVA	95
8.1.6	Another method for subset selection	96
8.2	Experiment 2: Serial Implementation of Prism	98
8.2.1	Correlation Analysis.	98
8.2.2	ANOVA	99
8.2.3	Dimensionality	99
8.2.4	Metric Selection	100
8.3	Experiment 3: Inefficient Memory Access Pattern Algorithm	102
8.3.1	Correlation Analysis	102
8.3.2	ANOVA	103
8.3.3	Dimensionality	104
8.3.4	Metric Selection	104
8.4	Experiment 4: Matrix-Vector Multiplication Tests	106
8.4.1	Correlation Analysis	106
8.4.2	ANOVA	107
8.4.3	Dimensionality	107
8.4.4	Metric Selection	108
8.5	Analysis of Results	110
8.6	Scientific Programmer Actions	112
8.7	Summary	113
9	Conclusion	114
9.1	Research Summary	114
9.2	Contributions	115
9.2.1	A Methodology for Obtaining Relevant Performance Information	116
9.2.2	The Use of Design of Experiments for Performance Analysis Experimentation	117
9.2.3	The Usage of Data Reduction and Statistical Analysis	118
9.3	Validation	121
9.4	Conclusions	122
9.5	Future Work	122
A	Foundations of Computational Science and Engineering	125
A.1	Mathematical Preliminaries	125
A.1.1	Other Terms	128
A.2	Application	129
A.2.1	Finite Elements Analysis	129
A.2.2	Iterative Solvers	129

A.2.3	Matrix-Vector Multiplication	130
A.3	Advanced Architectures	132
A.4	Languages and Environments	134
A.4.1	Shared Memory	134
A.4.2	Message Passing	135
A.4.3	Problem Solving Environments	135
A.5	Performance Measurement	135
A.5.1	Tools	136
A.5.2	Statistical Terms	137
A.6	Summary	146
B	Glossary	148
C	Matrix-Vector Multiplication Algorithms	151
C.1	Algorithm A	151
C.2	Algorithm B	152
C.3	Algorithm C	153
C.4	Algorithm D	153
C.5	Algorithm E	154
C.6	Algorithm F	155
C.7	Algorithm G	155
D	Experiment 1	157
D.1	Order of Execution of Experimental Runs for Experiment 1	157
D.2	Anova on the metrics obtained in Experiment 1	167
E	Experiment 2	170
E.1	Order of Execution of Experimental Runs for Experiment 2	170
E.2	Anova on the metrics obtained in Experiment 2	180
F	Experiment 3	183
F.1	Order of Execution of Experimental Runs for Experiment 3	183
F.2	Anova on the metrics obtained in Experiment 3	198
G	Experiment 4	201
G.1	Order of Execution of Experimental Runs for Experiment 4	201
G.2	Anova on the metrics obtained in Experiment 4	205
H	Additional Fortran files	211
H.1	Program to test new routines	211

I	Matlab Files	214
I.1	Program to compute order of experimental runs	214
I.2	Routine to compute entropy cost function	216
I.3	Routine to show scree test and the Kaiser-Guttman criteria	217
I.4	Program to validate intrinsic dimensionality estimators	218
J	Perl Script files	221
J.1	Script A: Generating Summary of Metrics	221
J.2	Script B: Create Crontab file	230
J.3	Script C: Convert data to minitab 13 format	233
J.4	Script D: Convert data to SAS format	236
	BIBLIOGRAPHY	240

LIST OF TABLES

2.1	OpenMP Metrics	19
5.1	Compiler Options in Experiment 1.	47
5.2	Compiler Options in Experiment 2.	48
5.3	Compiler Options in Experiment 3.	50
6.1	Metrics obtained from the SAR command	58
6.2	Metrics obtained from the IOSTAT command	59
6.3	Metrics obtained from the VMSTAT command	60
8.1	Metrics with largest correlation with execution time in experiment 1.	91
8.2	Effect of factors and interactions on the most correlated metrics with execution time for experiment 1.	92
8.3	Number of metrics to keep variability of the current data according to three different criteria for experiment 1.	94
8.4	Metrics with highest information content in experiment 1.	95
8.5	ANOVA on the metrics shown in table 8.4. Main effects.	96
8.6	Metrics with highest information content selected by SVD for experiment 1.	97
8.7	ANOVA on the metrics shown in table 8.6. Main effects.	97
8.8	Metrics with largest correlation with execution time for experiment 2.	98
8.9	Effect of factors and interactions on the most correlated metrics with execution time in experiment 2.	99
8.10	Number of metrics to keep variability of data according to three different criteria in experiment 2.	100

8.11	Metrics with highest information content in experiment 2.	100
8.12	ANOVA on the metrics shown in table 8.11. Main effects.	101
8.13	Most important metrics for experiment 2 according to SVD.	101
8.14	ANOVA on the metrics shown in table 8.13. Main effects.	102
8.15	Metrics with largest correlation with execution time for experiment 3. . . .	103
8.16	Effect of factors and interactions on the most correlated metrics with execu- tion time for experiment 3.	103
8.17	Estimate of the intrinsic dimension of this data set.	104
8.18	Metrics with highest information content in experiment 3.	104
8.19	ANOVA on the metrics shown in table 8.18. Main effects.	105
8.20	Most important metrics for experiment 3 according to SVD.	105
8.21	ANOVA on the metrics shown in table 8.20. Main effects.	106
8.22	Metrics with largest correlation with execution time.	107
8.23	Effect of factors and interactions on the most correlated metrics with execu- tion time for experiment 3.	107
8.24	Estimate of the intrinsic dimension in experiment 4.	107
8.25	Metrics with highest information content for experiment 4	108
8.26	ANOVA on the metrics shown in table 8.18. Main effects.	109
8.27	Most important metrics for experiment 4 according to SVD.	109
8.28	ANOVA on the metrics shown in table 8.27. Main effects.	109
8.29	Percentage of metrics kept for the analysis.	110
A.1	Order of experiments for a fully randomized experiment.	144
D.1	Order of execution of experiments	157
D.2	ANOVA	167
E.1	Order of execution of experiments	170
E.2	ANOVA	180

F.1	Order of execution of experiments	183
F.2	ANOVA	198
G.1	Order of execution of experiments	201
G.2	ANOVA - main factors effect in experiment 4	205
G.3	ANOVA - two term interaction effect in experiment 4	207
G.4	ANOVA - three and four term interaction effect in experiment 4	209

LIST OF FIGURES

1.1	Typical analysis flow for tuning an application.	4
1.2	Integrative performance analysis.	5
1.3	Proposed approach for application tuning.	6
1.4	Proposed methodology.	6
3.1	Proposed methodology to extract information in an <i>OCS</i>	22
3.2	Model of an experiment.	23
3.3	Feature Subset Selection Scheme.	26
3.4	The combination of feature selection and feature extraction for performance data analysis.	27
4.1	Preliminary Problem Analysis is the first step in the proposed methodology.	29
4.2	Some representative finite elements.	31
5.1	Design of Experiment step in the methodology.	40
5.2	Compiler operating on selected software codes.	41
5.3	Linker and loader operating on compiled codes.	42
5.4	Example of one block in our split-split plot design.	44
6.1	Data Collection step integrated with the methodology.	53
6.2	Stages in the program mapping process [1].	54
6.3	Collinearity problem: Those metrics obtained by the operating system may come from the same groups of variables.	57
7.1	Data Analysis is the last step in the proposed methodology.	63

7.2	Performance Data Analysis Architecture.	65
7.3	Graphical View of a Discrete-Time Continuous Value Stochastic Process . .	66
7.4	Example of a matrix format used for the performance data.	68
7.5	Two principal components of the validation data - no normalization	71
7.6	Two principal components of the validation data - Min-Max normalization .	72
7.7	Two principal components of the validation data - Euclidean normalization	73
7.8	Visual display of the correlation matrix of the data obtained from the validation experiment matrix-vector multiplication	75
7.9	Feature subset selection	77
7.10	Classification scheme of feature selection measures [2]	79
8.1	Eigenvalues of correlation matrix in experiment 1.	93
8.2	Eigenvalues of correlation matrix for synthetic data.	94
9.1	Typical analysis flow for tuning an application.	115
9.2	Proposed approach for application tuning. The dashed line shows the part of this tuning methodology addressed by this research.	116
9.3	Proposed methodology to extract information in an <i>observable computing system (OCS)</i>	117
9.4	Summary of statistical analysis techniques used for extracting information about performance outcomes.	119
A.1	Venn diagram of mathematical signals.	127
A.2	Experiment illustrating execution time of two simple comparative studies.	142
A.3	Execution time when Machine B is used in the study	142

CHAPTER 1

Introduction

1.1 Motivation

Finding a suitable, high-performance, computer-based solution of a real-world problem is a complex process. The number of different possibilities for programming style, algorithms, parameters, operating system environment variables, compiler and flags, and architecture, among others, create a set of entangled interactions. A clear understanding of these interactions and how they relate will assist the programmer in the decision-making process.

The process of solving a real-world problem is composed of two major steps: conceptualization and instantiation. Conceptualization is the process of developing a new idea to solve a problem. Instantiation is the action of describing the idea as a series of steps to solve the problem. There are different levels of instantiation [3] from the highest level of abstraction to the most detailed solution of the problem where all parameters have been defined. An algorithm is a well-defined procedure to solve a problem in a finite number of steps. In this context, an instantiation can be expressed as a collection of algorithms seeking the solution to a problem. An implementation is defined in this work as an instantiation where all parameters and algorithms have been determined.

For a person solving a real-world problem, many different criteria might be considered to measure success in a given implementation. Some might consider robustness, usability, or speed as criteria for measuring how well suited is the implementation. Robustness refers here to the capability of software to properly react to unusual requirements [4]. Usability

is related to the characteristics of software that makes it easy to learn, efficient to use, easy to remember, error tolerant, and pleasant to use [5]. The most common measure used is speed: the faster the algorithm, the better the performance.

Different factors affect computer performance of an implementation. For instance, speed is determined by a series of factors such as programming style, language, compiler options, and architecture and these are selected by the application programmer as part of the implementation process. Application programmers are usually experts in one area. For example, application developers in signal processing are proficient in applying mathematical concepts to solve their problems. Their level of expertise is usually concentrated in one of the levels of instantiation, typically in the highest level of abstraction. This leads to the selection of alternatives without having a complete understanding of the relationship among each of the factors and the obtained performance.

Mapping refers in this work to the relation between a language of a high-level abstraction and a language of a concrete architecture [6]. It is still unknown, even for experts in the area of performance, what are the relationships among the different parts of the mapping process. This is due to the vast number of platforms, compilers, compiler options, algorithms, and programming styles associated with a particular implementation. With the advent of advanced computer architectures with parallel units or parallel organization, we add to this list different programming paradigms for parallel processing.

This dissertation is addressing the problem of obtaining information about relationships between various factors and the computer performance of an implementation. It introduces a statistics-based methodology to bridge the gap between the high-level abstraction and the low level implementation information. A case study in the area of computational electromagnetics illustrates the formulation of real-world problems of large scale modelling of physical systems. This methodology uses an empirical analysis and a statistical approach to understand how different computer performance metrics at different levels are affected by the selection of parameters in the implementation process.

1.2 Problem Statement

The performance obtained when intensive applications are mapped to a computing platform is highly dependent on how well adapted is the application to the platform. However, the tuning process used in most of today's applications still leaves room for improvement. The information required to establish existing relations among high-level factors and low-level performance data is not easily obtained due to the complexity of the system. This contributes to the difficulty experienced by scientific programmers to obtain acceptable performance on advanced systems. This is the main problem addressed by this dissertation. A number of issues contribute to this problem.

First, the mapping process is not one to one. Source code lines get optimized by the compiler and linked to advanced libraries in a way in which executable code do not correspond directly to source code. Also, the order of execution on the actual system is rearranged during run time, making it difficult to associate performance costs to specific code or segments of instructions. Moreover, communication patterns among processes might be affected by unpredicted asynchronous situations in the system.

Second, performance analysis incorporates the application programmer's insight into the tuning process, which prevents automatic performance evaluation. This is illustrated in Figure 1.1.

The application programmer needs to understand instrumentation, learn the appropriate tools, and interpret data and its relation to the code, in order to optimize the code for a particular system. This method is complex and prone to wrong interpretations [7]. Also, important performance information might be overlooked, hidden by the large amounts of performance data collected by instrumentation systems. Moreover, as architectures increase in complexity and larger problems are solved, the performance analyst will require a greater experience or expertise, which only a select group of people might have.

Finally, current performance analysis tools are not necessarily portable and scientific application programmers do not find them intuitive or appealing [8].

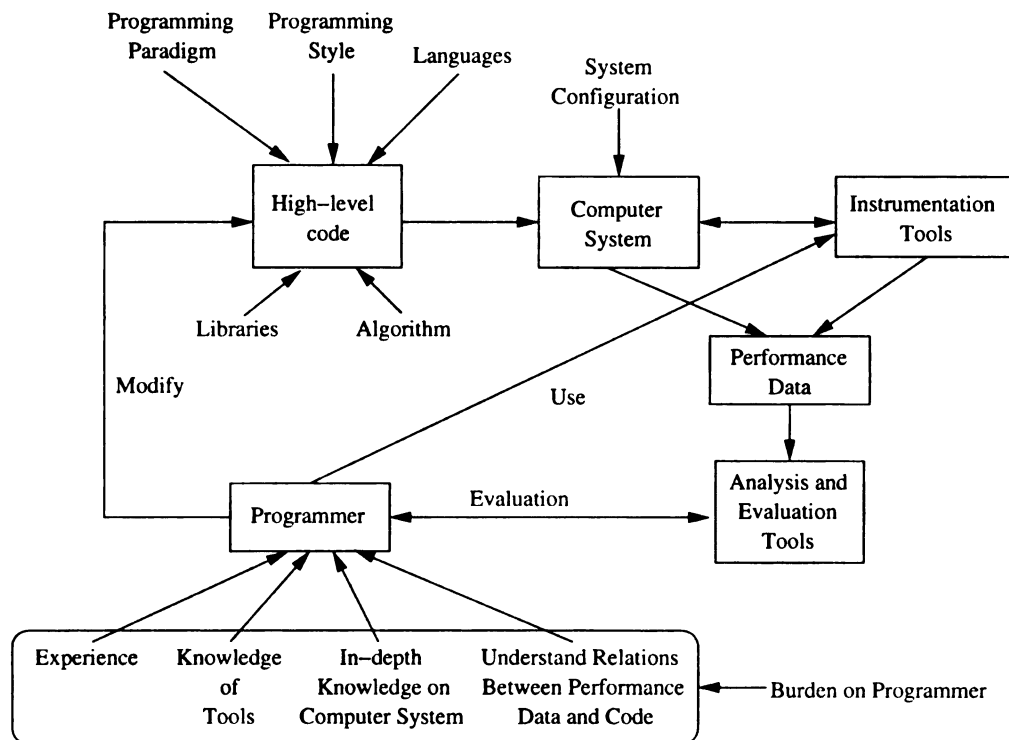


Figure 1.1. Typical analysis flow for tuning an application.

1.3 A Methodology for Evaluating Performance Information

The proposed solution is based on the integration of theories and methodologies that are related to different aspects of the performance analysis problem posed in Section 1.2. We have borrowed ideas from other disciplines to find a solution to the problem.

The use of an integrative approach for performance analysis is proposed to combine information at different levels and present it to the scientific programmer in a meaningful form. Figure 1.2 shows a description of this environment. Performance data analysis is integral to this approach. The traditional formulation for performance tuning, shown in Figure 1.1, should be modified to satisfy the scientific programmer's needs. We suggest the tuning methodology presented in Figure 1.3.

This dissertation proposes a methodology which integrates four main components. These are systematically applied to an observable computing system to extract relevant informa-

Integrative Performance Analysis

Integrative: directed toward coordination with the user's environment

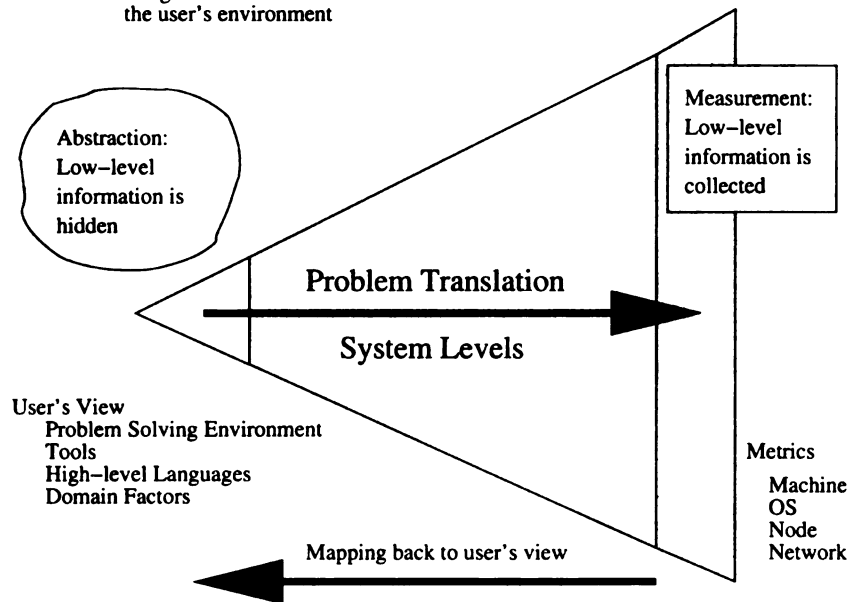


Figure 1.2. Integrative performance analysis.

tion to assist scientific programmers to tune applications to advanced architectures. The four steps are: *problem analysis*, *design of experiments*, *data collection*, and *data analysis* as illustrated in Figure 1.4.

A preliminary problem analysis is used to visualize what is affecting performance and gather preliminary information. Screening experiments are used to establish which factors are mostly affecting performance and select a subset of factors for experimentation.

Design of experiments is used to collect appropriate information from the smallest number of experimental runs. There are a large number of experimentation strategies from which we can select the most appropriate one based on the characteristics of the system and software.

The third step is data collection. This is determined by the particular system, language, and instrumentation tools used. Data are collected at runtime and analyzed *post mortem*.

Data analysis begins by extracting the data to an appropriate matrix format. The per-

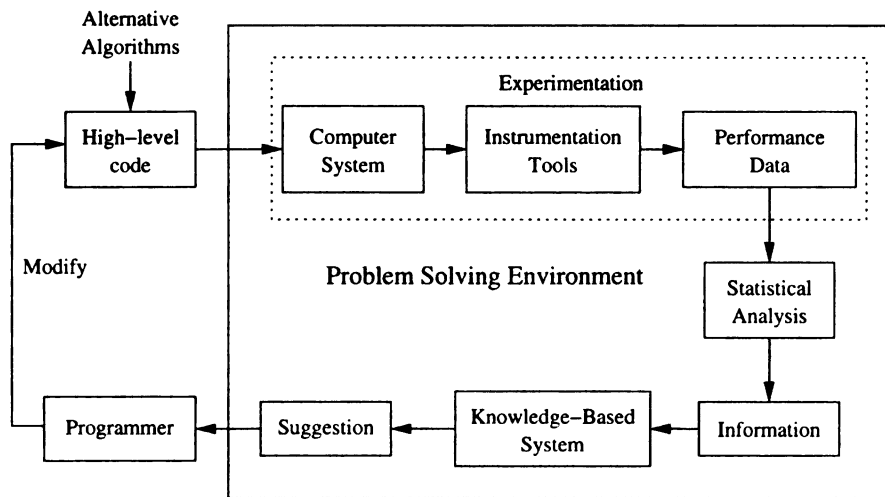


Figure 1.3. Proposed approach for application tuning.

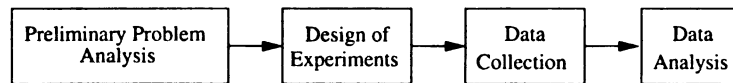


Figure 1.4. Proposed methodology.

formance data matrix columns represent metrics and each row represent one experimental run. Dimension normalization is applied to this matrix. The correlation matrix is computed to determine which metrics are linearly related to execution time. Then we proceed by extracting relevant metrics to analyze.

Multivariate statistical methods are used to extract this information. Intrinsic dimensionality estimators are used to estimate how many metrics explain the variability of the data. Feature subset selection methods are used to extract the most important metrics for the analysis.

ANOVA is used to test the hypothesis that no factors are affecting performance. When this hypothesis is rejected, *post hoc* comparisons and analysis of means are used to determine which factors are affecting relevant metrics.

Thesis Statement: Design of experiments, instrumentation, dimensionality estimation,

feature subset selection, and ANOVA can be systematically combined to obtain information relevant to performance analysis when mapping algorithms to advanced architectures. The use of these techniques will assist in locating, in an unbiased manner, sources of performance improvement.

1.4 Contributions

The contributions of this work are as follows.

Our first contribution is a systematic methodology to obtain information on the existing relations when mapping compute-intensive applications to advanced architectures. This methodology is composed of four steps: *problem analysis*, *design of experiments*, *data collection*, and *data analysis*.

Second, we have identified the need of screening experiments to limit the number of factors when experimentation is used. The use of large number of factors in the experimentation phase can be unfeasible in terms of time and resources, for real applications and advanced computing systems.

Third, we have defined a performance characterization experiment (PCE) as the procedure of selecting a software code in a given computer language, applying a parallelizing compiler with an ordered set of directives, running the code on a target machine, and retrieving a well defined set of performance parameters.

We also establish design of experiments (DOE) as necessary to establish causal relations between high-level factors and low level performance information [9]. If we do not use DOE, only correlational relations can be established. When design of experiments is used, the performance analyst do not require extensive knowledge about the code in order to obtain information on the relations. With the traditional tuning methodology, the performance analyst must incorporate previous knowledge and experience into the process of tuning an application.

The measurements obtained from performance instrumentation vary largely in scale. We have identified the need preprocessing before some of the statistical methods can be applied.

We examined three different types of preprocessing schemes: log normalization, min-max normalization, and dimension normalization. Dimension normalization resulted the most appropriate one for our type of data. In addition, correlation analysis identified those metrics most linearly related with execution time and revealed collinearity in measurements obtained through software instrumentation.

Multidimensional data analysis methods were identified as appropriate tools for extracting relevant information content in performance data. Sequential forward search was used to identify those metrics most important for the performance evaluation process. Entropy was used as a measure of information content in the response data set. Evaluation of three intrinsic dimensionality estimators - scree test, Kaiser-Guttman, and cumulative percentage of total variance - revealed that even though all produce similar results, scree test is not appropriate for automated performance evaluation. Scree test requires the visual evaluation of a graph.

Finally, analysis of variance and *post hoc* comparisons were used describe which factors, if any, are affecting individual performance metrics, and which ones are statistically similar or different. From these results we learned that compiler directives might be grouped into similar categories, where their effect is statistically not distinguishable.

1.5 Dissertation Overview

This dissertation is organized into ten chapters.

Chapter 1 called *Introduction* presents the motivation and objective of this research work. Chapter 2 gives a review of the current status of research. Chapter 3 presents an overview of the proposed methodology used to relate high-level abstractions to low-level performance information. Chapters 4 to 7 will expand on the methodology, giving details on the purpose of the steps included in the methodology. Finally, chapters 8 and 9 present results and conclusions.

CHAPTER 2

Related Work

2.1 Introduction

A diversity of methods have been proposed to reach the goal of automated performance analysis. Some work has been done in relating performance analysis to high-level abstractions [8, 10]. Also in the use of statistical methods for performance data analysis [3, 11, 12] and on the use of multidimensional data analysis for studying performance data [13, 14, 15]. Moreover, the APART group is working towards the advancement of automated performance tools [16]. However, we know of no other research working on the integration of all these aspects into a coherent and general methodology for extracting information on the existing relations of performance information obtained at the lowest level and the highest level of abstraction.

In the following sections we present different approaches which are related to the topics of this dissertation. Section 2.2 presents different approaches for relating low level performance information to high level abstractions. Section 2.3 discusses how statistical methods have played an important role in obtaining unbiased information about the performance of a system. In section 2.4 we present multivariate analysis methods used for performance data analysis. Finally, in section 2.5, we present the collective work of a group of researchers working in the area of automatic performance evaluation tools: APART.

2.2 Relating Performance Information and High-Level Abstractions

Early work on performance analysis tools proposed the types of information to be collected at different instrumentation levels. Irvin and Miller [10, 17, 18] proposed a framework called the NV model (noun-verb model) for the identification of fundamental information to be collected by performance tools in order to correlate high-level abstractions to low level performance information. In their work, a *noun* is an element from which a measurement is taken and a *verb* is an action taken by or on the noun. A *level of abstraction* is then defined in terms of the collection of nouns and verbs associated to a specific point in a mapping process. Irvin and Miller have defined four different levels of abstraction: source code, runtime library, operating system, and hardware.

The relationship between nouns and verbs at one level of abstraction and those at another are known as a *mapping* in the NV model. Those mappings are classified as static or dynamic. Static mappings occur previous to runtime while dynamic mappings occur during runtime. The NV model was implemented in Paradyn [10] with CM-Fortran.

This work is similar to our research in the goal of correlating performance information across levels of abstraction. Moreover, we have adopted their definition of levels of abstractions in our work. However, the NV framework described by Irvin and Miller should be used by tool developers to relate information across levels while our methodology aims to aid scientific programmers to find relations across levels of abstractions with existing tools, regardless of their use of the NV model.

Another approach to relate performance information to high-level constructs was used by Mellor-Crummey, *et al.* in [8]. They have correlated low level information to source code by creating a tool called HPCView. In their work, the authors have identified the main reasons for lack of user support for performance evaluation. In general they claim that performance tools do not improve the productivity of codes for three main reasons: usability, scope of metrics, and appropriate assignment of data and source. First, the lack of usability of existing tools comes from the absence of both language and architecture portability, and

the need of user intervention for instrumentation. Second, the scope of performance metrics need to be expanded by presenting collective information and this information should be presented with respect to relevant parts of the code. Finally, assigning performance data to source code implies the correct assignment, after compiler optimizations, of performance costs to source information.

HPCView is a toolkit designed to correlate performance information to source code. It has been implemented for the following platforms: Alphas running Tru64, IA-32 machines running Linux, IA-64 machines running Linux, SGI systems running IRIX64, and Sun SPARC machines running SunOS. HPCView takes profiling data collected by platform dependent profilers and combines them with an estimate of the program structure obtained from a tool called *bloop*, included in this toolkit. This information is then used to produce a hyperlinked database viewable from any web browser. Basically, HPCVIEW requires a configuration file containing the paths to source code files, a set of performance metrics obtained from the system, and a set of parameters to configure the display. It produces html and javascript files which can be read by any web browser to produce an interactive display that can be used by the programmer to identify metric - source code correlations. Moreover, derived metrics can be computed by HPCView by means of MathML expressions suggested by the programmer/analyst. There are two main disadvantages of HPCView. First, currently the tool relies on system dependent profiling tools, which may not provide accurate performance information. Second, the accuracy of *bloop* depends on the compiler used, since mapping information is collected from the associations of the symbol table generated by the compiler.

As we have stated previously, our goal is to correlate performance information to high level construct, which was achieved by the HPCView toolkit. Therefore we can state that our research work are complementary. Our methodology is system and tool independent while theirs is available for certain platforms only. It would be interesting to use HPCView to verify results obtained by our methodology, something not done previously since the tool was not ported to SUN machines at the time of experimentation of this study. Another basic

difference is the use of user’s intuition in selecting which metrics are going to be displayed in HPCView. Our methodology points out to some metrics of interest for the user to pay attention to them.

2.3 Using Statistical Analysis on Performance Data

Statistical analysis has been used in the past for analyzing certain aspects of performance analysis such as execution time, memory performance, and scalability. We will describe these in the following sections.

2.3.1 Statistical Analysis of Algorithms and Heuristics

The most common approach to compare algorithms in literature is to compare times published in literature with the best time obtained from a new algorithm. However, the actual running time of a coded algorithm is affected by the machine, compiler, language, programming style, and workload, among different factors. To fairly compare two algorithms, Coffin and Saltzman [3] suggest statistical analysis of algorithms. This will show the relationships between the problem and the algorithm.

According to Coffin and Saltzman, there are basically two different approaches to study algorithms: theoretical analysis and empirical analysis. In theoretical analysis, an analysis previous to the implementation is performed based on the parameters of the problem. In empirical analysis, the actual time is evaluated by implementing it in computer code. In our case, we will use empirical analysis.

There are very important results from Coffin and Saltzman’s studies. One of their most relevant conclusions is that statistical evaluations can provide surprising results or conclusions different from superficial evaluation of results. A general procedure is suggested for comparing algorithms and making recommendations of which one to use. There are basically three steps in the general procedure. First, the *data collection* is done. Here a careful experiment design must be done. Some possible design approaches are: completely randomized, randomized block, factorial, and fractional factorial. The second step is *exploratory*

data analysis. This analysis is done graphically to identify possible patterns or trends in the data. Finally, the last step is *formal statistical analysis*. Here some basic methods such as hypothesis testing, parameter estimation, and confidence interval calculations are performed.

There are some statistical considerations for adequately comparing algorithms. The experimental design is important for the analysis and reproducibility of results. The model and analysis done on the data will depend on whether exploratory analysis or confirmatory analysis is done. In exploratory data analysis, data is observed graphically to visualize trends. In confirmatory data analysis, a model is preconceived and a qualitative analysis is done to confirm or reject the model. Another important consideration is to identify the experimental unit in order to analyze the data in terms of the unit. An experimental unit is the unit to which a treatment is applied. Another consideration is the sample size. Not enough data will have low power and a confidence interval too wide. Too many observations will reject any hypothesis and will lead to every factor being important.

In their analysis Coffin and Saltzman concluded, after analyzing several examples, that running times are often nonnormal and they exhibit heteroskedasticity (nonconstant variance). Therefore the analysis performed should be robust to nonnormality or nonconstant variance. Analysis of variance methods are robust in this sense.

2.3.2 Scalability Analysis using Factorial Designs.

The work of Alabdulkareem *et al.* is close to our research [11]. In this work, scalability of large codes is studied from the perspective of experimental design. Similar to our work, they use experimental design and ANOVA to study parallel codes from what they call a “black-box” perspective. Main differences between their study and ours are that they concentrate only on scalability issues while we want an overview of the state of the system in general. In their study, fractional factorial designs were used to control the number of experiments, with large numbers of factors, each with two levels. In contrast, we have limited the number of experiments by designing screening experiments and having the insight of the programmer to select a few important factors for experimentation. Therefore, in our case, we have used

full factorial designs with fewer factors and more than two levels per factor. In their work, measurements of execution time are used to estimate scalability of the system. Some of their conclusions are applicable to our work. First, knowledge of the code is necessary for the selection of factors since the results obtained in the study depend on the selection of factors; however, not extensive knowledge is required. Also, a method for limiting the number of experiments is required due to the time consuming task of experimenting with intrinsically long running codes. Their results demonstrate that the CRAYJ90 exhibits better scalability than the IBMSP2 for the application code they are using. This application is the weather prediction code ARPS (Advanced Regional Prediction System) from the Center for the Analysis of Prediction of Storm (CAPS) of the University of Oklahoma. Two main routines were identified by their method as having a large effect on the scalability for the ARPS code.

2.3.3 Statistical Analysis of Memory Hierarchy

Sun, *et al.* have proposed the use of multivariate regression, factorial design, contrast and *post hoc* comparisons, and ANOVA for the analysis of hierarchical memories [12, 19, 20]. In this research, a four level methodology was developed for the evaluation of memory hierarchies on single processor performance. The dependent variables used to assess memory performance are *cpi* (cycles per instruction) and cache hits, therefore this methodology assumes that both can be measured on the system under study is conducted has a method of providing cycles, instructions, and cache hit ratios.

The methodology is composed of four steps: main effect study, code/machine classification, scalability comparison, and memory hierarchy study [20]. The *main effect study* examines the effect of code and machine on the variable *cpi*. ANOVA is used on a two-level full factorial design to determine if there is a significant effect of machine or code on the cycles per instructions on the system. The second step is *code/machine classification*. If there is significant effect on the code or machine, a *post hoc* comparison can be made, where significant differences among means is studied to classify code or machine into similar statistical groups. The least significant difference (LSD) *post hoc* method was used in this

study. Third, a *scalability comparison* is done using regression analysis. Here problem size and machine are studied versus *cpi*. Finally, the last level of analysis uses cache hits to locate which memory components are causing the variations found in the previous three levels of analysis. This last step is dependent on the kind of measurements available on a particular system while the other three levels are independent of the system.

This study was done on two SGI systems: Origin 2000 and a Power Challenge, both with the same processor but with different memory hierarchies. Results obtained by Sun *et al.* show that the Origin 2000 has better scalability for the types of codes used in the study.

Like this study, our research is using ANOVA and design of experiments to study performance. However, we do not concentrate our work on only one metric and we do not propose a multilevel analysis. While Sun uses a full factorial fully randomized design, we are using a full factorial split-split plot design. Our work also concentrates on overall metrics on a multiprocessor system in contrast to their work on single processor analysis.

2.4 Multivariate Methods for Performance Data Analysis

There are several works in the area of reduction of multidimensional performance evaluation data.

Early work by Nickolayev *et al.* [13] demonstrates that the use of statistical data clustering techniques on performance trace data is useful for the reduction of large volumes of data while keeping important system behavior. They use dynamic clustering to select a subset of traces for representing trace behavior on the system. Both clustering and entropy-based feature subset selection are classified as unsupervised feature classification methods. They also use normalization as one data preprocessing technique on the data. However, two important differences exist: the dimension along which the reduction is done and its associated cost function. In Nickolayev's work, a subset of traces is selected as representative, reducing the trace space while leaving the dimension of performance metrics intact. In our work, on the other hand, we reduce along the metric space, reducing the number of metrics showing important information. Another difference is the cost function used. While we use

entropy as cost function, their work bases selection on Euclidean distance which is more appropriate for the goal of trace reduction.

In their work [14], Vetter and Reed used statistical projection pursuit, a multidimensional projection technique to identify “interesting” performance metrics from a monitoring system. This work aims to reduce the number of metrics and the dimensionality of the data. They do this reduction dynamically by periodically using projection pursuit for identifying which metrics are important. Projection pursuit is a dimension reduction technique where multivariate data sets of high dimension are projected to two or three dimensions according to the “best” projection. The “best” projection angle is selected according to a projection index, which determines the outcome of the method. This index is a cost function determined by the objective of projection pursuit and is usually based on the amount of structure found on the projection. Vetter and Reed dynamically projected tracing data reducing the number of metrics to three interesting metrics at a given sampling time. This was used on data extracted by the Pablo performance tool.

There are some similarities between this work and ours. First both studies concentrate on the automatic selection of performance metrics by selecting interesting metrics based on a cost function. Moreover, the projection index used by Vetter and Reed is based on an entropy estimate, as well as ours. In both studies we perform data preprocessing: they use data smoothing, centering, normalization to a range, and sphering on the data while we use Euclidean normalization.

On the other hand, they use a linear combination of measurements for the projection and selection of metrics, in contrast to our use of subset selection techniques. Moreover, we use dimensionality estimation to determine the number of metrics to select before starting the selection process, different to three metrics obtained by projection pursuit. The selection of metrics in our work is *post mortem* while in their case is dynamic.

One surprising similarity that we found is that even though they are using MPI and a distributed memory system for their case study, and we are using OpenMP and a shared memory system, both methods selected the metric *bwrite* as an important metric. In all

our examples this metric was selected by our subset selection method. In most examples presented in their work [14], it was selected by projection pursuit. The reason might be that both cost functions are based on an entropy estimate.

A more recent piece of work is presented in [15]. In this study, Ahn and Vetter used several multivariate statistical techniques on hardware performance metrics to characterize high-performance computing systems. They specifically evaluated the use of principal component analysis (PCA), clustering, and factor analysis to extract performance information. Factor analysis and clustering are combined to gain insight into the behavior of metrics, selecting important metrics to observe, and classifying or categorizing metrics together. They apply these methods on three different applications on two different IBM SP systems and the parallel code was developed both with MPI and OpenMP. Their results show that, for homogeneous systems, metrics coming from processors with similar tasks are categorized together. Master and worker threads show different behavior and were classified into different clusters by their method. Also different memory behavior caused the classification of processors into different groups.

In the same way, we have identified multidimensional statistical analysis techniques as fundamental for the automated detection of patterns in the information provided by the large volumes of data obtained by performance tools. Like Ahn and Vetter, we have used a correlation matrix to establish linear relation among metrics. We have also suggested the use of a knowledge based system for recommending optimizations to the programmer. Our work is also applicable to homogeneous systems. Likewise Ahn and Vetter keep metrics which account for most of the variations of the data. However, there some differences in our work we should mention. First, our multidimensional analysis combines feature subset selection, dimensionality estimation with entropy cost function, and ANOVA for extracting information from the dataset whereas Ahn and Vetter have combined clustering, factor analysis, and principal component analysis in their study. Principal component analysis was used for visualization purposes. Second, we have studied software performance metrics while they examine hardware associated metrics. Third, they do not estimate the dimensionality

of the data set, nor use any cost function associated to entropy. On the other hand, we have not used derived metrics as in their work, which can complement our study.

2.5 Automatic Performance Evaluation

APART stands for Automatic Performance Analysis: Resources and Tools. It started in 1999 as a group of researchers, institutions, and companies investigating the area of automatic performance analysis. They have been working towards the formalization of the language and methods to present performance information and also have identified the requirements for automatic performance analysis tools based on their vast experience in the area.

In order to automate the analysis of performance data, the APART group worked on three different aspects of the problem. APART first identified the requirements for automatic performance analysis [7]. Some important conclusions were reached by this first study. Analysis should take into consideration the application, programmer, and performance monitoring support. Also programming style and architecture should be taken into account in the analysis. They also determined that there are two styles for performance evaluation: hardware utilization and the identification of the relation between performance data and source code. Finally, the complex existing interactions on these types of systems may cause that slight variations produce large performance differences.

The second study completed by the group identified the need for an infrastructure for measurement, modeling, and analysis of performance [21]. They developed the APART Specification Language (ASL). ASL describes performance properties using an object oriented model of performance properties [16, 22, 23] and a corresponding syntax. In this work performance properties were identified according to the programming model used.

Table 2.1 contains some of the Apart performance properties defined for OpenMP. The first group of metrics are related to memory utilization, from which we can identify cache usage as of importance. Synchronization and parallel organization metrics are also listed. Since we are using KAP/Pro, we can measure only number of threads and synchronization

time.

Table 2.1. OpenMP Metrics

Name	Level	Tools for measurement	Category
Instruction Cache misses	Low Level	x	Memory
Data Cache misses	Low Level	x	Memory
Instruction cache hits	Low Level	x	Memory
Data Cache hits	Low Level	x	Memory
Cache hit rate	Low Level	x	Memory
Disk Access	Low Level	x	Memory
Buffer size	Low Level	x	Memory
Number of loads and stores	Low Level	x	Memory
Time of context switches	Low Level	x	Memory
Remote reference (page) count	Low Level	x	Memory
Number of threads	High Level	xosview, top	Memory
Synchronization time	Low Level	KAP/Pro	Synchronization
Synchronization counts	High Level	x	Synchronization
Number of Iterations per Thread	High Level	x	Parallel Organization
Execution time of parallel loops	Low Level	x	Parallel Organization
Loop organization overhead	Low Level	x	Parallel Organization
Loop execution time	Low Level	x	Parallel Organization
Loop overhead	Low Level	x	Parallel Organization

We are using some of these defined metrics as our response variable in our study.

The third study concentrated of implementation related issues. A survey of existing tools was conducted and a categorization scheme was proposed. Integration of tools and experimentation were identified as key issues in automated performance.

2.6 Summary

Researchers have worked in different aspects pertaining our work. Early work in performance analysis established a model to be used by tool developers to relate high-level abstractions to low-level performance information. This is called the NV model [17, 18, 10]. Some tool developers have addressed this problem with a different approach. HPCView relates source code to performance data by estimating the program structure and combining compiler information with profiling information [8]. Even though these two works address the same problem as ours, we complement their work by providing a methodology applicable to platforms where their tools are not available.

ANOVA and design of experiments have been used in for the analysis of execution time of algorithms for operational research [3]. They have also been used for the study of scalability of large codes [11] and the analysis of memory hierarchies [12, 19, 20].

A third aspect used in our methodology is the application of multivariate methods for performance data analysis. Nickolayev *et al.* used data clustering for the reduction of trace data along the trace space dimension [13]. Vetter and Reed used statistical projection pursuit for identifying important metrics on a system [14]. Finally, Ahn and Vetter evaluated Principal Component Analysis, clustering, and factor analysis to extract performance information on collected with hardware counters on distributed system [15].

Finally, we present the work of the APART group whose goal is to investigate different aspects of automated performance analysis and to move forward the research in this area in a coordinated effort among diverse research groups, institutions, and companies [7, 16, 22, 23]. Automatic performance analysis is a quite active research area, kept alive by the APART group.

CHAPTER 3

Proposed Methodology

3.1 Introduction

This work addresses the problem of loss of information when mapping scientific applications to *observable computing systems*. Information mapping is crucial for automated performance evaluation. Our research proposes a well-defined methodology to extract relevant information from a set of observable measures which try to describe the performance of an *observable computing system (OCS)*. This methodology uses a combination of carefully designed experimentation, multidimensional data analysis, exploratory data analysis (EDA), and confirmatory data analysis (CDA). EDA is characterized by no preliminary knowledge about the possible relations of the variables under study and the use of statistics and graphical summaries to understand the information data is conveying. In CDA, formal statistical methods are used to confirm or reject a hypothesis about the population under study. Experimentation is used to collect unbiased data to confirm or reject the hypotheses and establish causal relationships among controlled factors and resulting observations. Multivariate analysis is used to extract meaningful relations among large sets of data.

The proposed methodology applies to multiinput-multioutput (MIMO) systems with a set of observable outputs [24] arranged into four basic steps as illustrated in Figure 3.1. First, a *preliminary problem analysis* is performed. Here we can visualize in general what is affecting performance and gather preliminary information. The second step is to *specify the experiment design* to collect enough unbiased information to be analyzed for establishing

relationships. The third step is to *collect the data*. Finally, in the last step is *data analysis*. A description of each step follows.

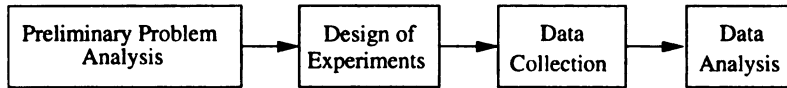


Figure 3.1. Proposed methodology to extract information in an *OCS*.

3.2 Preliminary Problem Analysis

A performance problem-solving process starts with the analysis of the problem specification. Here, the components of the observable computing system are identified. These include hardware and software components and measurement tools. In addition, information about the programmer's goal, the performance problem, and the application itself are collected. This delimits the problem scope.

Once the system, application, and performance goal are clear, the next step is to profile the code to identify potential functions to optimize. Analysis continues with the identification of possible factors affecting performance. These include environment factors, algorithms to solve those functions to optimize, and hardware specific factors. Next, a subset of factors is selected for the experiment, considering controllability, feasibility, practicability, and constraints.

3.3 Experiment Specification

The second step in the methodology is the *experiment specification*. The theory of *design of experiments* allow us to take an objective approach in the experimentation process [25]. Experimental relationships allow for the identification of causality among variables [9]. A well known model of the experimentation process is shown in Figure 3.2 [25].

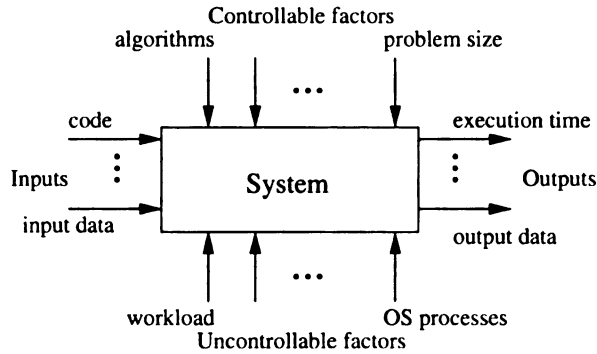


Figure 3.2. Model of an experiment.

Studying all possible factors and levels of these factors is an intractable problem. A level refers here to the different possible values of one factor considered in an experiment. In order to obtain the total number of experimental runs, it is necessary to calculate all possible assignment of factors when varying all at the same time. The next step is to select the random order in which the experimental runs will be executed. Randomization is required to avoid the influence of uncontrollable factors in the outcome. We must also have at least two replicates of the experiment [25].

The effect of each factor is obtained through experimentation by the use of a factorial design. In this type of design, all combinations of all levels of all factors are tested, usually in complete random order [26, 27]. For practical considerations, in certain cases a completely random set of runs might not be easily implemented. A completely randomized run would imply that from run to run any factor may change. For most computer applications, this is impractical. For example, in our study, changing the problem size from experimental run to experimental run results in excessive time and limits our ability to automatically control experimentation. So a *split-split-plot* design was used. Split-split-plot is a special case of a split-plot design. A *split-plot* design is a general case of a factorial design in which randomization is restricted. In this design, one factor is selected for a treatment. A treatment is a set of levels of controllable factors administered to an experimental run. The order in which the treatments will be applied to this factor is selected at random. Once this is fixed, a second factor is selected and, given the order for experimental runs selected for the

first factor, randomization is done on the second factor. This could be repeated successively. When a third factor follows the same restrictions, this is called a *split-split plot* design [25]. A partial randomization of experiments causes a higher experimentation error so split-split plot is suggested only when a completely randomized design is not possible for practical reasons.

3.4 Data Collection

The data collection step is the only one determined particularly by the computer system, language, and the tools used. This is due to the large variation of metrics available for different computer systems and at different levels. One group working towards standardization of performance metrics is the APART (Automatic Performance Analysis: Resources and Tools) group [7, 21]. Their work moves towards the formalization of the language and methods to present performance information and to identify the requirements for automatic performance analysis tools. APART workpackage 2 presents a set of metrics defined using ASL for determining some performance properties for shared memory, message passing, and high performance Fortran [21].

During this step we identify which metrics are measurable for the paradigms and systems being used. Specifically, we identify the instrumentation tools that are available and the metrics that are measurable at the operating system, application, and hardware levels. Then from these, for a given paradigm, we select the APART-recommended set of metrics. Important metrics suggested by the application programmer should also be selected. Once a set of performance metrics is selected, instrumentation is activated to collect the data. Code is compiled and linked as needed, and performance data are collected during execution.

3.5 Data Analysis

After data collection, analysis begins. Those metrics obtained from the OCS are a sample drawn from a stochastic process [28]. We assume this process is mean-square ergodic in the mean, that is, the corresponding time average converges to the ensemble average in the

mean-square sense. These assumptions are necessary to make use of the statistical methods explained below.

Performance metric data is first formatted to support the statistical techniques to be used. For one experiment, a matrix format is used. Each element of the matrix is either an average or absolute metric value. An average value is computed as the sum of all metric sample values divided by the number of samples, where the samples of the metric values are taken during execution time. Each column of the *performance data matrix* contains the measurement of one performance metric over a set of experimental runs and each row contains information about one experimental run. Several statistical techniques may be applied to this matrix, as described below.

The data should be preprocessed. There are several normalization methods which can be applied to high-dimensional performance metrics: absolute, log normalization, min-max, and vector normalization. Absolute refers to no normalization. Log normalization is the use of a logarithmic transform. Min-max is to transform the data to the $(0, 1)$ range. Finally, vector normalization normalizes each column by dividing by the Euclidean norm.

We have used the correlation coefficient to find linear relation among variables. The correlation coefficient is a measure of the linear association between two variables. The correlation matrix is a two-dimensional array of correlations where all correlation coefficients are organized systematically. The value of each element (i, j) in the correlation matrix contains the correlation coefficient between metric i and metric j .

In performance data analysis, large volumes of data with complex relationships contain the information on the behavior of the system. We use unsupervised feature subset selection methods for the automatic selection of important features describing the system. Figure 3.3 illustrates this process.

Two important issues of unsupervised automatic feature selection are the order identification or *dimensionality* of the data set [29] and the subset generation method [30]. Intrinsic dimensionality estimation methods have been used in the past to estimate the number of components to retain and the number of features to keep [31, 32]. This is illustrated in

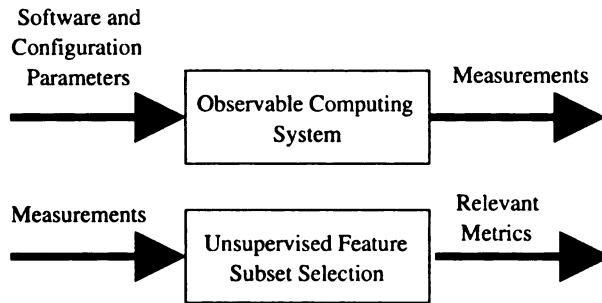


Figure 3.3. Feature Subset Selection Scheme.

figure 3.4.

We have tested several methods for intrinsic dimensionality estimation. Once the dimension of the data set is estimated, a subset search method should be selected. We have used sequential forward search with entropy cost function to select the most important metrics in the data set.

Analysis of variance (ANOVA) is a statistical procedure for the analysis of the response of an experiment. It is used to estimate the contribution of each factor to the variations in the outcome. We are using ANOVA to determine whether there is influence of any of the factors on the result obtained for each performance metric. Post hoc methods will then identify which differences are significant in the data.

3.6 Summary

This chapter presented an overview of the methodology developed to understand the relationship between high-level abstractions and low-level performance information in an observable computing system. This methodology is composed basically of four steps: preliminary problem analysis, specification of the experiment design, data collection, and data analysis. An overview of each step was given. Specific details of the methodology are given in chapters 4, 5, 6, and 7.

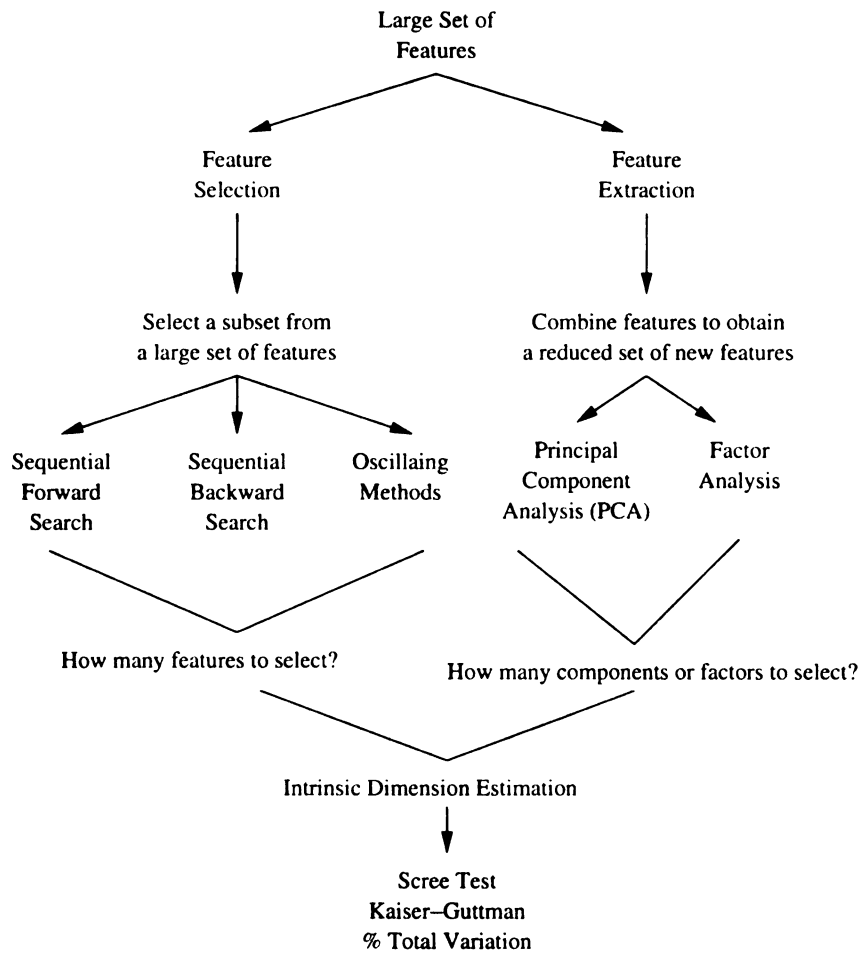


Figure 3.4. The combination of feature selection and feature extraction for performance data analysis.

CHAPTER 4

Preliminary Problem Analysis

4.1 Introduction

Most problem solving techniques applicable to the area of computer performance evaluation start with a problem analysis. Problem solving can be broadly viewed from two different perspectives: a *behaviorist* approach and an *information-processing*-based approach [33, 34]. The behaviorist view is based on stimulus and response without considering the process to solve the problem. The information processing view is concerned with the process leading to problem solution [33]. This second perspective is more appropriate for the performance evaluation problem. In the information processing approach a pattern or form of solution is suggested to get the desired goal.

General problem solving patterns in literature establish different steps to reach a solution [34]. Despite their differences, all of them concur on three basic steps: *problem and system definition*, *current situation assessment*, and *evaluation of alternatives*. These steps conform the preliminary problem analysis, illustrated in Figure 4.1, and will be described in the following sections.

4.2 Problem and System Definition

The target problem selected for this research consisted of porting a computational electromagnetics application to a symmetric multiprocessor system with four processors. Specifi-

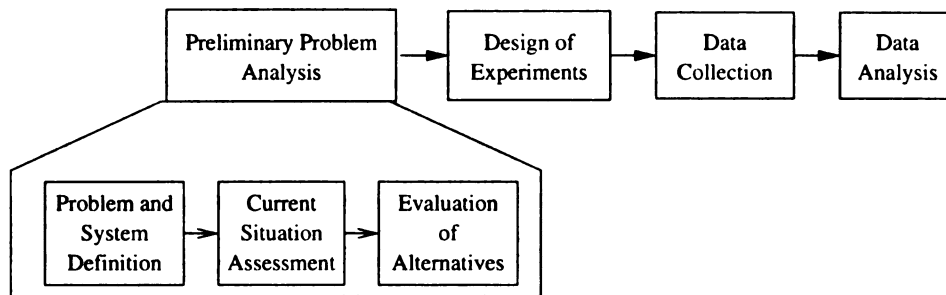


Figure 4.1. Preliminary Problem Analysis is the first step in the proposed methodology.

cally it implemented finite elements method for conformal antenna analysis. The goal was to parallelize the serial code, taking advantage of the system, and reducing the execution time.

The code in this application could be considered legacy code since it was developed in Fortran77 over a period of time [35]. The programmer's expertise is in electromagnetics and numerical methods. However, to tune the application to the target system required detailed knowledge of the computer system and tools.

4.2.1 Finite Element Method in Electromagnetics

The analysis and design of antennas require the characterization of associated electromagnetic fields. Maxwell's equations form the basis of electromagnetic theory and apply to general fields. In the general form, these equations are not easily solved by direct analytical methods. However, they are simplified and tailored to specific conditions by making appropriate assumptions. To completely determine the set of equations, boundary conditions are imposed in addition to complementary condition equations. Numerical methods are then applied to find a feasible solution. Those methods impose a heavy computational load on the target system. Two numerical methods used in computational electromagnetics (CEM) are integral-equation methods, also known as *methods of moments*, and finite element-frequency domain (FE-FD) methods [36].

The code in the target application implements a finite-element boundary-integral (FE-

BI) method for conformal antenna analysis. A conformal antenna is an antenna which adapts or "conforms" to the surface to which it is mounted on [37]. These antennas are attractive to use on vehicles due to their low weight and flexibility [38]. Even though detailed information about the application itself is not required, a general understanding of the FE-BI method is needed for optimizing and comprehending the code.

Integral-equation methods, also known as the method of moments (MoM), start with an integral equation, generally involving Green's function, in the time domain. They assume that an integrated function can be approximated by a linear combination of a set of basis or expansion functions [36, 39, 40]. This method converts the integral equation to the matrix form

$$\mathbf{Z}\{I\} = \{F\} \quad (4.1)$$

where Z is the impedance matrix, I is the currents data vector, and F is the excitation data vector [36, 40]. Important characteristics of the method of moments are that the generated matrices are dense, the method is computationally intensive, and has large memory requirements [36, 41].

When considering the solution of Maxwell's equations in differential form, the finite elements method (FEM) applies. This method is based on the decomposition of the equation domain into nonoverlapping subregions called finite elements [39]. This decomposition is called meshing [42]. In each subregion, a simple function approximates the solution of the equation which might be complex over the larger region [39, 43]. If the elements are small enough, this approximation is close to the solution. In this case, *small enough* implies smaller than $\frac{1}{10}$ of the wavelength per side [42]. For two-dimensions, the elements are polygons. Simple geometric shapes are used as elements for three dimensions, such as those shown in Figure 4.2 [44].

Instead of solving the original formulation, which may include higher order derivatives in the differential equation, the finite element method uses a *weak formulation*, which reduces the differentiability requirements [44]. This allows the use of piecewise functions as

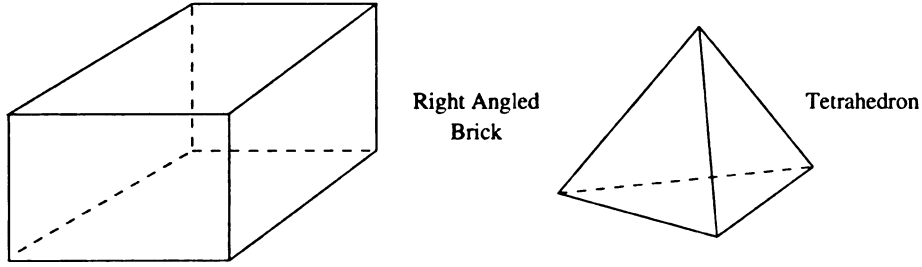


Figure 4.2. Some representative finite elements.

approximation functions [43, 45].

When Maxwell's equations are integrated over the finite elements and boundary conditions are imposed, a system of linear equations is constructed that can be expressed in the form

$$A \{H_z\} = \{I\}, \quad (4.2)$$

where A represents a square, sparse matrix, H_z denotes the magnetic field vector, and I is the excitation column vector. The advantage of the FEM is that A results in a sparse matrix so it has lower memory requirements. A disadvantage is that it is difficult to evaluate the boundary conditions when the domain is infinite. Appropriate boundary conditions for terminating the mesh are required. Volakis *et al.* summarize the steps to generate and solve a FEM system as follows [36]:

- The domain of the problem is determined.
- The method for terminating the mesh is selected.
- The finite elements are chosen.
- The mesh is generated.
- The matrix is generated by using the wave function.
- Boundary conditions are applied to construct the linear system.

- The solver is selected and the system is solved.
- Those parameters of interest are computed. These might include capacitances, impedances, scattering matrices, etc.

The finite-element boundary-integral (FE-BI) method combines a finite elements method with integral equations to represent the fields outside the surface and to terminate the mesh. Exact boundary conditions are used to terminate the mesh [44]. The resulting system is partly sparse and partly dense. The advantage of this method is that for a certain class of problems, it can be solved efficiently.

The code used in this particular application implements FE-BI for the analysis of conformal antennas. It was developed by Leo C. Kempel, from the Electromagnetics Laboratory at Michigan State University [46]. The FE-BI equations solved by the method are obtained from the weak form of the wave equation explained by Kempel in [46] and shown below

$$\begin{aligned} \int_V \left[\nabla \times \mathbf{W}_i \cdot \bar{\bar{\mu}}_r^{-1} \cdot \nabla \times \mathbf{W}_j \right] dV - k_0^2 \int_V \left[\mathbf{W}_i \cdot \bar{\bar{\epsilon}}_r \cdot \mathbf{W}_j \right] dV \\ + jk_0 \int_{S_R} \left[\frac{(\hat{n}_R \times \mathbf{W}_i)(\hat{n}_R \times \mathbf{W}_j)}{R_e} \right] dS \\ - k_0^2 \int_{S_a} \int_{S_a} \left[\mathbf{W}_i \cdot \hat{z} \times \bar{\bar{G}}_{e2} \times \hat{z} \cdot \mathbf{W}_j \right] dS' dS = f_i^{int} + f_i^{ext} \quad (4.3) \end{aligned}$$

The first term on the left side of the equation is related to the magnetic field, the second with the electric field, the third with the resistive transition conditions, and the last one is the boundary integral term. The terms f_i^{int} and f_i^{ext} are functions of the internal and external excitations. The code computes the input impedance of the antenna.

As explained in section A.2.2, the solution of a large system of linear equations can be found either using direct or iterative solvers. Direct solvers determine the solution in a finite number of steps, while iterative solvers begin with an initial guess of the solution and iteratively improve it until a *good enough* solution is obtained. As we previously explained there are stationary and nonstationary methods. Stationary methods include the Jacobi and the Gauss-Seidel. Nonstationary methods include the conjugate gradient (CG), Generalized

Minimal Residual (GMRES), BiConjugate Gradient (BiCG), Conjugate Gradient Squared (CGS), and Biconjugate Gradient Stabilized (Bi-CGSTAB). Iterative methods implemented in the application code are BiCG, CGS, and Bi-CGSTAB. The convergence rate of the method can be improved through *preconditioning* as previously explained. A diagonal preconditioner, also known as Jacobi preconditioner, was used in the code [42].

In general, the application consists of a biconjugate iterative solver for a system of linear equations of the form

$$[A] \{E\} + \begin{bmatrix} G & 0 \\ 0 & 0 \end{bmatrix} \{E\} = \{F\}, \quad (4.4)$$

where A is a sparse matrix and G is a symmetric dense matrix. Only the lower triangle part of G is saved in memory using the compressed sparse row format [42]. Dense matrices are very large so memory becomes a variable of concern when solving the problem. One of the input parameters is the error threshold. It determines the number of iterations. In the implemented code we worked with relatively small problems of variable sizes, where the smallest number of unknowns was 6033.

A preliminary study using a profiler pointed to a dense matrix-vector multiplication subroutine as the code bottleneck, taking most of the execution time and being by far the most time consuming task. Changing the problem size gave a similar profile, pointing to the same routine as bottleneck. We ran the application on an SMP-4 processor Sparc Enterprise machine. OpenMP directives were used to take advantage of the SMP architecture of the machine.

4.2.2 Observable Computing System

An observable computing system (OCS) is any given computing system with a set of observable measures. In our context, these observables represent a physical quantity measurable from the particular system and are traditionally called *performance metrics*.

The OCS used in this case study was composed of an SMP computing system, a set of software tools, and a set of performance measurement tools with their corresponding metrics.

Architecture

We ran our experiments on a quad-processor Sun Enterprise 450 Server. This machine is a shared-memory, symmetric multiprocessor system (SMP). Each one of the processors is an UltraSparcTM II running at 400MHz with 2MB of local high-speed external cache memory. The UltraSparc II is a superscalar/superpipeline 64 bit RISC microprocessor [47] with a nine-stage pipeline and nine concurrent execution units. The execution units are four integer execution units, three floating-point execution units, and two graphic execution units [48]. The UltraSparc II has a specialized instruction set called VIS (Visual Instruction Set)TM designed to accelerate multimedia, image processing, and networking applications. The processor contains a 16K non-blocking data cache and a 16K instruction cache with 2-bit branch prediction. The processors connect to main memory and I/O via the Ultra Port Architecture (UPA) data bus [49].

This particular machine has 640MB of main memory, with 4-way memory interleaving. The connection from each processor to main memory is through a crossbar switch configured to obtain uniform access to memory. There are two levels of cache. The first level, contains both an instruction cache (I-cache) and a data cache (D-cache). The I-cache is associative with 32-byte cache lines, while the D-cache is directly mapped with two 16-byte sub-blocks per line [50]. The second level is the external cache.

Software and Analysis Tools

In addition to the operating system, three main components were used for software development and measurement: the KAP/Pro toolset, a profiler, and operating system measurement calls.

The operating environment was Solaris 7 (SunOS 5.7 operating system) which supports the UltraSparc architecture and multithreading [51]. The KAP/Pro toolset was used for software development, measurement, and performance analysis. KAP/Pro has three components: Guide, Assure, and GuideView. Guide is the compiler and linker component of the toolset. It is actually a precompiler on top of a Fortran compiler. In our case, we

used the Forte Fortran HPC 6 Fortran compiler. Guide supports OpenMP directives for Fortran 77 and includes a statistical library which allows instrumentation of multithreaded code. Assure is the debugger/thread analyzer and GuideView is the visualization tool for performance analysis. Performance of code instrumented with Guide is visualized through GuideView.

The profiler used was gprof. A profiler is used to determine which portion of the time is spent on each routine. This gives us a rough idea of where should we start optimizing the code. The operating system measurement calls used were sar, iostat, and vmstat.

4.3 Current Situation Assessment

A general assessment of the performance of the code was obtained through the gprof profiler. It was important to determine which routines were the most time consuming and how large was the difference between them. This information is useful because sometimes improving only one routine will cause a large difference in performance.

Profiling the EMAGs code pointed to the routine BiMATVECCav taking up most of the execution time. BiMATVECCav was performing a matrix-vector multiplication operation on a dense matrix. The routine required double precision complex number operations. It was noticed that the dense matrix was originally saved in a vector in column-major order. Changing the data structure to save this matrix in row-major order improved the execution time by 40%. After the data structure was changed, the same routine was still taking a significant amount of time. Other functions and subroutines executed by the program were consuming only a small percentage of the time. This included algorithms related to sparse matrix-vector multiplication.

It was important to determine a list of possible factors affecting the performance of the application. From these, a subset was selected for experimentation. Some of the factors considered were included:

- **Compiler options:** The compiler options selected by the programmer affect performance. In some compilers, the order in which the compiler options are given affects

the outcome. Since in our case, the order affected the size of the executable, we assumed that the order of compiler flags was also a factor to consider. There were combinations of flags not allowed by the compiler so these were discarded as possible options.

- **User workload:** When the system is being heavily used, the performance is completely different than when the system is dedicated to only one task.
- **Sampling time:** How often the system was sampled for measurement affects performance. If the system is sampled too often, there is a heavy load imposed on the system. On the other hand, if the system is sampled at large intervals, important transient information might be missed.
- **Number of processors:** The number of processors working on a problem might be changed to determine system speedup.
- **Problem size:** This number indicated how large was the linear system of equations to be solved by the code. It was controlled by the antenna specifications.
- **Algorithms:** Different algorithms for solving dense matrix-vector multiplication might change performance. Also algorithms for sparse matrix-vector multiplication or any other important kernel of the code.
- **Iterative solvers:** Different types of iterative solvers might be programmed to solve the particular application under study.
- **Hardware:** The system might be changed itself. For instance, memory might be increased, operating system changed, etc.

From these possible factors, a subset was selected for experimentation. Those whose effect was not as noticeable as others were left out from the subset.

4.4 Evaluation of Alternatives

The parsimony or Pareto principle establishes that few factors will have the most effect on the outcome while others will contribute very little. The process of pre-selecting those factors which will be considered for experimentation is called *screening* [52, 53].

In this phase of experimentation we have used the *one-factor-at-a-time* approach described by Montgomery [25] as the screening method. We selected a baseline or set of levels for each factor, and then we varied one factor at a time and obtained a general assessment of which factors were having a larger effect on the response.

Some criteria in selecting the factors to use in experimentation were controllability, practicability, and constraints. Controllability refers to whether or not we can control the factors themselves. Practicability indicates the usefulness of the variation of the factor in the experimentation process. Constraints, such as time, were also imposed on the experiment design.

As explained in Chapter A, factors are classified as design, held-constant, and allowed-to-vary factors. From the list of factors described above, the type of iterative solver, the number of processors, and the hardware were held constant. This was due to practical reasons. Workload was allowed to vary but under certain constraints: no user was allowed on the machine when experiments were conducted. This limited the workload to processes ran by the operating system. Finally, compiler options, sampling time, algorithms, and problem size were selected for screening.

Once the screening was complete, sampling time was discarded since the three levels selected were basically not affecting the outcome of the experiment. The only measurement taken for the screening process was execution time.

In summary, the factors will be considered in the following way:

- Held-constant factors
 - Iterative solvers
 - Number of processors

- Hardware
- Allowed-to-vary factors
 - User workload (with limited use of resources)
 - Sampling time
- Design factors
 - Compiler options
 - Problem size
 - Algorithms

The code was parallelized using OpenMP calls on the BiMATVECCav routine. This routine was by large taking most of the execution time. Two main reasons were behind selecting OpenMP over MPI for parallelization. First, the system was a shared memory machine, therefore, OpenMP would map directly to the system. Second, the application programmers were not familiar to parallel processing and changes to the code in OpenMP would be easier to understand than MPI changes.

4.5 Summary

This chapter has presented an overview of the *Preliminary Problem Analysis* step of the proposed methodology. Problem solving techniques suggest three basic phases of preliminary problem analysis: problem and system definition, current situation assessment, and evaluation of alternatives. These three phases were explained in the context of our application. The application code used in this work deals with the problem of the use of finite element (FE) methods for the analysis of conformal antennas. It specifically implements a finite-element boundary-integral method which combines FE with integral equations (IE) to represent the fields outside the surface and to terminate the mesh. This leads to the solution of large systems of linear equations using iterative methods. The iterative solvers

implemented in this code are BICG, CGS, and Bi-CGSTAB. A profile of the code pointed to a dense matrix-vector multiplication as the code bottleneck. We ran the application on a Sun Enterprise 450 SMP system with four processors, parallelizing it with OpenMP. The KAP/Pro toolset was used for software development, measurement, and analysis. A list of possible factors affecting performance was compiled and evaluated. Screening experiments were held to study the factors and limit this list to three design factors: problem size, dense matrix-vector multiplication algorithm, and compiler options. Next chapter presents a description of the *Experiment Specification* step of the methodology.

CHAPTER 5

Specifications for the Experiment

5.1 Introduction

Empirical studies were identified as key step for automatic performance evaluation by the APART group in workpackage 3 of the phase one of the study [54]. The goal of experimentation is to understand the interactions in the system. Causal associations can be made in properly designed experimental studies [9]. Design of experiments is an effective tool for understanding relations in processes and is typically used in industrial statistics. DOE provides a minimum set of experiments necessary to obtain the information we need in the most effective order. Experiment specification is the second step in the integrated methodology, and it is illustrated in Figure 5.1. We will explain how DOE is used to obtain relevant information on the system-software interactions in our work.

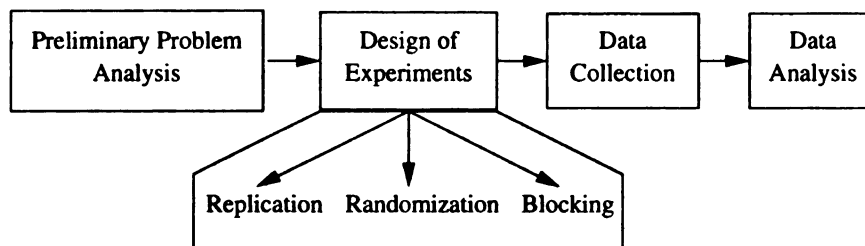


Figure 5.1. Design of Experiment step in the methodology.

5.2 Performance Characterization Experiments

Before proceeding further, we need to define some terminology.

Definition 5.1 *Compiling*

Let C_S be the set of all selected software codes to be mapped to a target machine and their variants obtained from a given canonical formulation. Let C_C denote the set of compiled codes. Let P_C^{IDC} denote the parallel compiler that operates on the selected software codes and d_k a compiler option for the selected compiler. Then for a software code $C_i \in C_S$ and its compiled version $\widehat{C}_i \in C_C$ there is a relation

$$P_C^{IDC}(d_k, \dots, d_m) \{C_i\} = \widehat{C}_i \quad (5.1)$$

This is called *compiling*.

Figure 5.2 show the process of converting the selected software codes to a set of compiled codes.

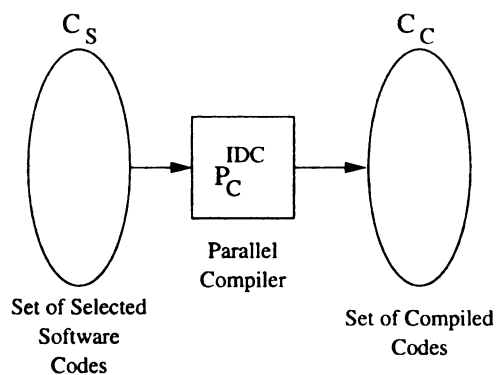


Figure 5.2. Compiler operating on selected software codes.

Label IDC will identify a specific compiler. For example, P_C^{KAP} will identify the KAP/Pro Compiler for Fortran called guidef77.

Definition 5.2 *Linking*

Let L_L denote the linker and loader used on the code and let C_E be the set of executable codes. The relation on C_C and C_E is $L_L(l_s, l_m, \dots, l_r)\{a_k\} = \{b_k\}$ where L_L denotes the linker and loader used; $l_s, l_m, \dots, l_r$ are the different libraries used in the *linking* process; and $a_k \in C_C, b_k \in C_E$.

Figure 5.3 shows the process of linking and loading the compiled code. It produces an executable file.

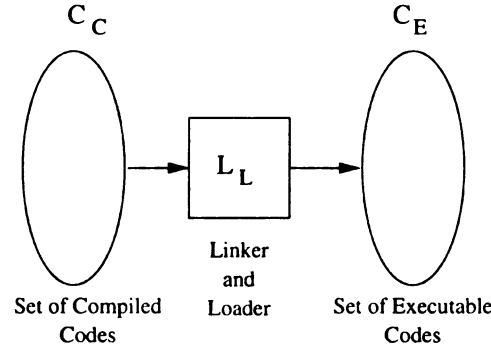


Figure 5.3. Linker and loader operating on compiled codes.

Definition 5.3 *Metrics*

Let Γ denote the *set of metrics* used to measure performance and $\rho^{os_{ver}}$ the operating system under which measurements are taken. Then for $a_k \in C_E$ and $\gamma_k \in \Gamma$, $\rho^{os_{ver}}(p_m, p_n, \dots, p_r, p_s)\{a_k\} = \gamma_k$ where $p_m, p_n, \dots, p_r, p_s$ denote parameters used to obtain the metrics from the system.

Definition 5.4 *Performance Characterization Experiment*

A *performance characterization experiment* is a composition of the form $\rho^{os_{ver}} \circ L_L \circ P_C^{IDC}$ acting on a selected code. In other words, a *performance characterization experiment* or (PCE) is defined as the procedure of selecting a software code in a given computer language, applying a parallelizing compiler with an ordered set of directives, running the code on a target machine, and retrieving a well defined set of performance parameters.

5.3 Design of Experiment

High performance computing systems were designed to be used at maximum potential, however, most of the time, performance is not even close to full potential. Even though performance analysis tools were designed to aid researchers to achieve maximum performance, they are not widely used. Reasons include the lack of guidance on possible problems in the code and the fact that users are expected to understand the data and views presented by the tools and associate them to their codes [8]. A solution to this problem could be obtained through the use of automatic performance tools that guide the user in the analysis and the solution of the performance problem.

HPCview addresses this problem by correlating data to source code [8]. However, these utilities are not necessarily available for all platforms. Given the large variety of systems available, nowadays we need a general methodology to correlate performance data from advanced platforms to variations in source code or changes in any other important factor considered relevant.

This can be done through the use of design of experiments. A carefully designed experiment can establish relations among factors and outcomes on an observable computing system. In the screening process explained in Section 4.4 we have done preliminary studies to detect the most influential factors on the outcome. A simple design was used for screening purposes. Once the set of factors is selected, an experiment should be designed.

As explained in Section A.5.2, there are three basic criteria to consider in an experiment: replication, randomization, and blocking. We require to have at least two replicates of an experiment [25]. A full-factorial design was used. A simple experiment can be used for screening purposes but for complex interactions as those obtained in high performance computing systems, it is not efficient.

The randomization scheme is of importance in deciding a specific design of experiment. In a completely randomized design, the order in which experimental runs are arranged is randomly allocated. When in a factorial experiment we are unable to completely randomize the order of the runs, a split-plot, or as in our case a split-split plot design might be used.

A partial randomization of experiments causes a higher experimentation error so split-split plot is suggested only when a completely randomized design is not feasible.

Figure 5.4 shows a graphical description of a block of our split-split-plot design. A block refers to a replicate or repetition of the basic experiment. In this figure, a block in the design is divided into whole plots where the the problem size (1, 2, and 3) was selected at random. The subplot factor is the matrix multiplication algorithm (A and B). Then sub-subplots will contain the compiler options ($a - m$) that were tested randomly.

2	1	3
A b,l,a,...,h	B d,c,g,...,b	B a,i,d,...,c
B j,h,e,...,c	A k,b,g,...,a	A m,a,e,...,h

Figure 5.4. Example of one block in our split-split plot design.

5.4 Detailed Description of the Experiment

Four different experiments were performed: two characterization experiments using the application code, and two validation experiments, one with the application code and another one with a matrix-vector multiplication algorithm.

The goal of all experiments was to identify existing relations among different compiler options, problem size, and code to the performance metrics obtained when running the experiment. The effect of uncontrollable factors in the experiments caused by external workload has been minimized by selecting a no workload system. We have selected a subset of compiler options, algorithms, and metrics to test our methodology.

The name of our application code is Prism. It implements a finite-element boundary-integral (FE-BI) method for conformal antenna analysis as explained in section 4.2.1. The

code was developed by the electromagnetics research group (EMRG) at Michigan State University.

5.4.1 Experiment 1: Parallel implementation of Prism

As explained earlier, we parallelized Prism using OpenMP directives. The objective of the parallelization was to improve a matrix-vector multiplication routine taking up most of the execution time. Three different factors were selected for the experiment: compiler options, algorithm, and problem size. We performed three repetitions of the experiment. A full factorial experiment design was used, but as explained earlier, a fully randomized problem size is impractical. So we worked with a split-split plot design.

When establishing the experiment design, we devised the following number of levels for each of the three factors: two algorithms, three problem sizes, and sixteen compiler options. Compiler options are evaluated from left to right in one pass in our target compiler. Therefore, permutations in the order of compiler options are considered a different set of compiler options. The sixteen compiler options we are using were obtained by permutations and combinations of three different compiler flags. We selected flags `-fast`, `-unroll=2`, and `-xcrossfile` after performing a preliminary study of the effect of different flags on the code. These were the three flags with most effect on the results. Compiler option `-Wgstat` will always show in the compiler options since it is required for measuring performance with KAP/Pro's statistical library. However, when studying compiler option `-xcrossfile` in more in detail, we found that it requires optimization level four or more, so it is ignored in the following cases:

- Compiler option `-xcrossfile -Wgstats`, is equivalent to `-Wgstats`
- Compiler option `-unroll=2 -xcrossfile -Wgstats` is equivalent to `-unroll=2 -Wgstats`
- Compiler option `-xcrossfile -unroll=2 -Wgstats` is equivalent to `-unroll=2 -Wgstats`

Therefore the experiment consist of two algorithms, three problem sizes, and sixteen compiler options. This experiment then consists of 234 experimental runs. This comes from formula A.10.

The sequence of experiments were labelled E1 to E234.

Therefore, we ended having the following factor levels in the experiment:

1. Algorithm

- Algorithm A is the original matrix multiplication algorithm shown in Appendix C.
- Algorithm B is the modified matrix multiplication algorithm shown in Appendix C.

2. Problem Size

- The original problem size was $N = 6033$.
- The problem was modified by changing the antenna specifications which affected most the dense matrix and we obtained a size of $N = 6337$.
- The problem was modified by changing the antenna specifications which affected most the sparse and we obtained a size of $N = 13857$.

3. Compiler Options

We had three replications of the experiment. Experiments 1 to 78 were replication one, experiments 79 to 156 were replication two, and experiments 157 to 234 were replication three. The order in which the algorithms, compiler options, and size were selected was randomized in the following way. First, we selected a random problem size. For that size we randomly selected the order of the algorithms. Third, for each option, at random we pick the 13 compiler options to perform the experiments. This is called a split-split-plot design. The metrics shown in Tables 6.1 to 6.3 were selected for assessing the experimental results. One problem we had to confront when designing the experiment was that statistical

Table 5.1. Compiler Options in Experiment 1.

1	No flags -Wgstats
2	-fast -Wgstats
3	-unroll=2 -Wgstats
4	-fast -unroll=2 -Wgstats
5	-fast -xcrossfile -Wgstats
6	-unroll=2 -fast -Wgstats
7	-xcrossfile -fast -Wgstats
8	-fast -unroll=2 -xcrossfile -Wgstats
9	-unroll=2 -xcrossfile -fast -Wgstats
10	-xcrossfile -fast -unroll=2 -Wgstats
11	-fast -xcrossfile -unroll=2 -Wgstats
12	-unroll=2 -fast -xcrossfile -Wgstats
13	-xcrossfile -unroll=2 -fast -Wgstats

tools such as Minitab do not allow 13 levels of one factor. These types of statistical tools are designed for industrial experiments, so that number of factors in one variable is very uncommon. Therefore, we programmed our own design of experiment random generator to obtain the order in which the experimental runs were allocated. The code is shown in Appendix I. The final order in which the experiments were executed is shown in Appendix I.1. We would also like to mention that since we used a split-split plot design, the ANOVA calculations were rather different than the fully-randomized calculations, typically found in statistical softwares. We used the software SAS to specify the split-split plot model and the appropriate error term calculations.

5.4.2 Experiment 2: Serial implementation of Prism

A serial version of Prism was studied. We had three repetitions of the experiment for which a split-split plot design was used.

The experiment consisted of two algorithms, three problem sizes, and thirteen compiler options. This experiment had 234 experimental runs.

Therefore we used the following factor levels in the experiment:

1. Algorithm

- Algorithm D is the matrix multiplication algorithm shown in appendix C.
- Algorithm E is the matrix multiplication algorithm shown in appendix C.

2. Problem Size

- The original problem size was $N = 6033$.
- The problem was modified by changing the antenna specifications which affected most the dense matrix and we obtained a size of $N = 6337$.
- The problem was modified by changing the antenna specifications which affected most the sparse and we obtained a size of $N = 13857$.

3. Compiler Options

Table 5.2. Compiler Options in Experiment 2.

1	No flags -Wgstats
2	-fast -Wgstats
3	-unroll=2 -Wgstats
4	-fast -unroll=2 -Wgstats
5	-fast -xcrossfile -Wgstats
6	-unroll=2 -fast -Wgstats
7	-xcrossfile -fast -Wgstats
8	-fast -unroll=2 -xcrossfile -Wgstats
9	-unroll=2 -xcrossfile -fast -Wgstats
10	-xcrossfile -fast -unroll=2 -Wgstats
11	-fast -xcrossfile -unroll=2 -Wgstats
12	-unroll=2 -fast -xcrossfile -Wgstats
13	-xcrossfile -unroll=2 -fast -Wgstats

The order in which the algorithms, compiler options, and sizes were selected was random in the same way they were selected for Experiment 1. The same metrics were selected for

assessing experimental results. The final order in which the experiment were executed is shown in Appendix E.

5.4.3 Experiment 3: Inefficient memory access pattern in Prism, validation experiment

A parallel version of Prism was studied, this time adding a third algorithm with an inefficient memory access pattern. We had three repetitions of the experiment and similarly a split-split plot design was used.

The experiment consisted of three algorithms, three problem sizes, and thirteen compiler options. This experiment had 351 experimental runs.

Therefore we used the following factor levels in the experiment:

1. Algorithm

- Algorithm A is the matrix multiplication algorithm shown in appendix C.
- Algorithm B is the matrix multiplication algorithm shown in appendix C.
- Algorithm C is the matrix multiplication algorithm shown in appendix C.

2. Problem Size

- The original problem size was $N = 6033$.
- The problem was modified by changing the antenna specifications which affected most the dense matrix and we obtained a size of $N = 6337$.
- The problem was modified by changing the antenna specifications which affected most the sparse and we obtained a size of $N = 13857$.

3. Compiler Options

The order in which the algorithms, compiler options, and sizes were selected at random in the same way they were selected for Experiment 1. The same metrics were selected for assessing experimental results. The final order in which the experiment were executed is shown in Appendix F.

Table 5.3. Compiler Options in Experiment 3.

1	No flags -Wgstats
2	-fast -Wgstats
3	-unroll=2 -Wgstats
4	-fast -unroll=2 -Wgstats
5	-fast -xcrossfile -Wgstats
6	-unroll=2 -fast -Wgstats
7	-xcrossfile -fast -Wgstats
8	-fast -unroll=2 -xcrossfile -Wgstats
9	-unroll=2 -xcrossfile -fast -Wgstats
10	-xcrossfile -fast -unroll=2 -Wgstats
11	-fast -xcrossfile -unroll=2 -Wgstats
12	-unroll=2 -fast -xcrossfile -Wgstats
13	-xcrossfile -unroll=2 -fast -Wgstats

5.4.4 Experiment 4: Matrix-vector multiplication, validation experiment

The kernel routine in Prism is a dense matrix-vector multiplication algorithm. We decided to study only this algorithm for validation purposes. The complexity of this problem allowed for a fully randomized full factorial experiment design.

The experiment consisted of three dense matrix-vector multiplication algorithms, two problem sizes, four compiler options, and two data structures with two repetitions. This experiment had 96 experimental runs.

Therefore we used the following factor levels in the experiment:

1. Problem Size:

- Size 1 refers to a 100 multiplications of a matrix of 500×500 elements.
- Size 2 refers to a 100 multiplications of a matrix of 1000×1000 elements.

2. Dense Matrix-Vector Multiplication Algorithm:

- Algorithm A described in Section 5.4.1
- Golub's algorithm described in [55] and shown as Algorithm F in appendix C.

- Algorithm G described in Appendix C. Inverse reading.

3. Compiler Options: Four levels

- Compiler Option 1: No flags -Wgstats
- Compiler Option 2: -fast -Wgstats
- Compiler Option 3: -O5 -Wgstats
- Compiler Option 4: -fast -O5 -Wgstats

4. Data Structure: Two levels

- The matrix is accessed row by row
- The matrix is accessed column by column

The order in which the algorithms, compiler options, and sizes were selected was random in the same way they were selected for Experiment 1. The same metrics were selected for assessing the experimental results.

5.5 Summary

A performance characterization experiment is a procedure for selecting a software code in a given computer language, applying a parallelization compiler with an ordered set of directives, running the code on a target machine, and retrieving a well defined set of performance parameters.

Design of experiments is key for determining whether factors are significantly affecting the outcome on a particular system. A screening experiment can be performed with a simple experiment to determine the most important factors affecting performance. Then a full factorial experiment can be used for studying the effect of these factors. A split-split plot design was used in our case for feasibility constraints.

Four different experiments were described in detail for our case study. The first three experiments were done using the full application code while the last one was done with a

matrix-vector multiplication algorithm. Experiment one studies the parallel implementation of Prism. Experiment two considers its serial implementation. In experiment three, an algorithm with a bad memory access pattern was introduced to the parallel implementation for studying the behavior of the system. Finally, the last experiment is a validation experiment in which algorithms for matrix-vector multiplication were studied for different problem sizes, data structures, and compiler options. This last experiment is fully randomized while the first three use a split-split plot design.

CHAPTER 6

Data Collection

6.1 Introduction

Data collection is the process of acquiring information about the actual behavior of a computer system and its associated software through measurements. Instrumentation is the group of modules to collect and manage from a program while it runs on a parallel or distributed system [56]. Data collection is the only step in the proposed methodology tied to architecture specific details. As previously mentioned, the proposed methodology consists of preliminary problem analysis, experiment design, data collection, and data analysis as depicted on Figure 6.1.

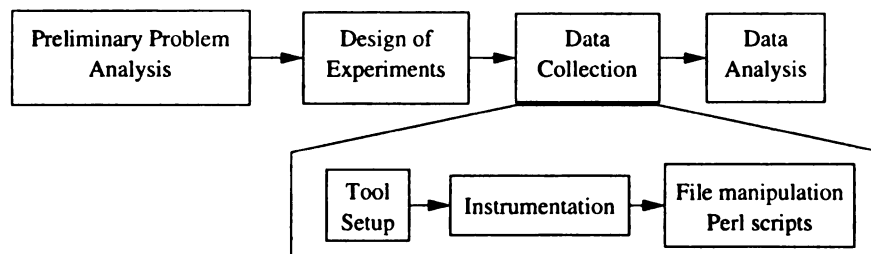


Figure 6.1. Data Collection step integrated with the methodology.

Instrumentation techniques are applied to different components in the system: programs, operating system, and processor [1]. *Program instrumentation* collects information of the

application code and its interactions with the system. When measurements are collected from the operating system, it is called *operating system instrumentation*. The operating system keeps track of the behavior of the memory, file system, cache, and processor status. Additional information about the processor may be directly obtained from hardware counters. In any observation, there is perturbation of the observed system. This is called intrusion. When hardware counters are used, the intrusion on the behavior of the system is smaller than software instrumentation. Hardware counters can provide information on memory behavior, floating point executions, instructions executed, and branching among others. For grid computing, there is also network instrumentation techniques. Here the goal is to monitor network traffic and find possible problems with the communication. In our work, we concentrated our efforts in program and operating system instrumentation techniques.

Software instrumentation can be inserted at different stages in the software mapping process [1]. Figure 6.2 illustrates this mapping process.

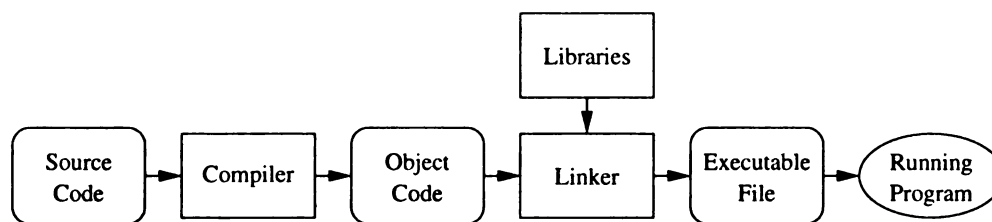


Figure 6.2. Stages in the program mapping process [1].

Some of the stages at which instrumentation can be inserted are: source code, compiler, object code, library, executable, and running code [57]. When instrumentation occurs at the source code level, either the programmer inserts instrumentation calls manually to the source code or a preprocessor automatically inserts them. These calls will collect event information from the system. Instrumentation can also be introduced through the use of libraries at compile time. Wrappers are used to insert instrumentation calls to the code.

Wrapper here refers to an interface to convert information from a software source to an application. MPE (Multi-Processing Environment) is an example of one of such libraries. It is distributed with the MPI programming language [58]. Another technique used is binary rewriting which edits the executable file and rewrites it with added instrumentation code. Atom is a example of these tools [59]. The last technique is called dynamic instrumentation. In dynamic instrumentation the running program is modified to generate performance data. This technique has been successfully used by Dyninst [60].

Computer performance metrics can be classified into two different types: traces and profiles [57]. Traces are a collection of events with associated information about the state of the system when the event occurred. Profiles are counts of events or summaries of events occurred in a specific period of time during the execution of a program.

There are different types of trace formats for the data: MPICL, SDDF, VTE, Vampir trace format, ALOG, SLOG, Epilog, and Paraver trace format are some of the typical formats found used in current performance tools. Each one contains a set of records to generate information about the event. Whichever format we select for the tracing data, it has to be manipulated to extract information suitable for the statistical analysis, as it is described in section 7.3.1. We are using perl scripts to manipulate the trace files and extract the data.

6.2 Tools

We have used software instrumentation at the library level and operating system instrumentation to collect information about the behavior of our application

6.2.1 Software Instrumentation

Software instrumentation at the library level was done through the use of the KAP/Pro toolset. The KAP/Pro is composed of three independent tools: the Guide Compiler, Assure, and GuideView. Guide is a precompiler for OpenMP. It accepts Fortran 77 code with OpenMP directives and produces Fortran code with thread programming [61]. Assure is

a thread debugger/analyzer. Guideview is a tool for performance analysis which shows a visual description of the data collected by the Guide instrumentation library.

Guide for Fortran, version 3.9, was used as preprocessor for Forte Fortran/HPC 6 to implement OpenMp. Guide contains the `-Wgstats` library to collect profiling information about the execution of a program. The compiler directive `-guide_stats` is used during the compiling/linking phase to collect performance information at run time.

Some statistics collected by the `guide_stats` library are: number of CPUs, start time and stop time, number of serial regions, number of parallel regions, number of barrier regions, CPU time, CPU utilization, elapsed time, imbalance time per thread, parallel time per thread, and total serial time.

6.2.2 Operating System Metrics

The concept of metrics is central to this work. By a metric we mean the variations of the observable quantities of a target computing system stored in the form of variables. The variables used for the metrics presented in this work are either of the interval or ratio types, as defined in Section A.5.2. For large scale complex systems, the dimension of the set of variables under consideration tend to be very high and the volumes of data tend to be extremely large [15]. The dimension of variables, and the size of the data associated with this type of observations of computing systems, necessitate the use of new methodologies and techniques of analysis in order to extract information relevant to an application programmer. As previously discussed in Chapter 3, our methodology centers on the integration of instrumentation-based data collection, systematic experimental design, and the selection of appropriate statistical analysis techniques. The instrumentation-based data collection on the target computing system is effected by the operating system of the machine (in our case Solaris). It is important to point out the *collinearity problem* that arises when tools such as an operating system is used for data collection in a computing system. Collinearity is studied in the context of regression and refers to the independent variables or *regressor variables*. Two variables are *exactly collinear* if there is a linear equation describing their relationship [62]. *Approximate collinearity* occurs if the linear equation approximately

gives the relationship among variables. Some metrics collected by the operating system log information from the same groups of variables causing large correlations between different metrics [50]. This is illustrated in Figure 6.3.

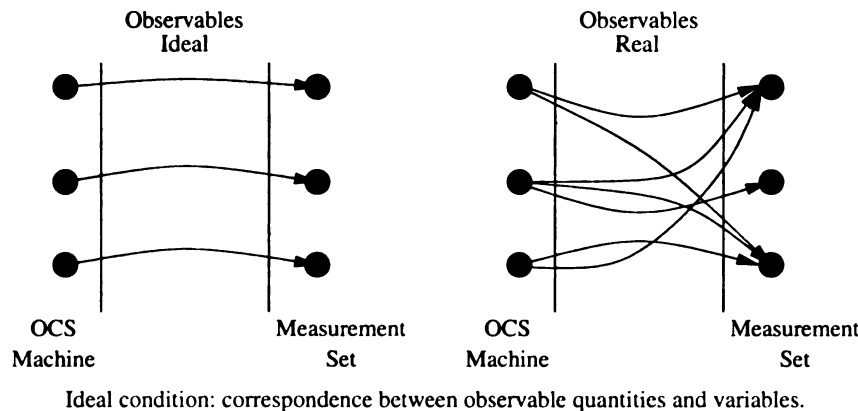


Figure 6.3. Collinearity problem: Those metrics obtained by the operating system may come from the same groups of variables.

We are studying collinearity on the dependent variables or the response variables. According to Sundberg in [63], the degree of multicollinearity can be estimated by principal component analysis of the sample correlation matrix of the data. If the smallest eigenvalue is less than 0.05 then there is collinearity.

From linear algebra it is known that highly correlated variables cause a large condition number of the observation matrix, making the use of a large number of statistical methods unfeasible. This problem can be alleviated by the use of subset selection methods [2].

At the operating system level performance information was collected using the *sar*, *iostat*, and *vmstat* commands. We used a sampling period of 20 seconds for each one of the commands. According to [50], as long as the sampling period is greater than 5 seconds, the system is not affected by the collection of data.

The *sar* command (system activity reporter) is used by system administrators to collect baseline system activity information when the system is having normal workload and then used to determine the reasons why a system is having a different performance. Among

the data sar will provide is buffer and paging activity, CPU usage, and system swapping activity. Some of the metrics of interest from this command are shown in table 6.1.

Table 6.1. Metrics obtained from the SAR command

Label	Name	Description	Category
m1	bread/s	Reads per second of data to system buffers from disk	Buffer Activity
m2	lread/s	Accesses of systems buffers to read	Buffer Activity
m3	%rcache	Cache hit ratios for read as percentage	Buffer Activity
m4	bwrit/s	Writes per second of data from system buffers to disk	Buffer Activity
m5	lwrit/s	Accesses of system buffers to write	Buffer Activity
m6	%wcache	Cache hit ratios for write as percentage	Buffer Activity
m7	pgout/s	Page-out requests per second	Paging Activity
m8	ppgout/s	Pages paged-out per second	Paging Activity
m9	pgfree/s	Pages per second placed on the free list by the page stealing daemon	Paging Activity
m10	pgscan/s	Pages per second scanned by the page stealing daemon.	Paging Activity
m11	atch/s	Page faults per second that are satisfied by reclaiming a page currently in memory (attaches per second)	Paging Activity (2)
<i>continued on next page</i>			

Table 6.1 (cont'd).

Label	Name	Description	Category
m12	pgin/s	Page-in requests per second	Paging Activity (2)
m13	ppgin/s	Pages paged-in per second	Paging Activity (2)
m14	pflt/s	Page faults from protection errors per second (illegal access to page)	Paging Activity (2)
m15	vflt/s	Address translation page faults per second (valid page not in memory)	Paging Activity (2)
m16	%usr	Portion of time running in user mode	CPU utilization
m17	%sys	Portion of time running in system mode	CPU utilization
m18	%wio	Portion of time running idle with some process waiting for block I/O	CPU utilization
m19	%idle	Portion of time running idle	CPU utilization
m20	pswch/s	Process switches	System swapping activity

Iostat reports input/output statistics from the system. Those metrics we observed using the *iostat* command are shown in table 6.2. *Iostat* provides statistics about CPU utilization and disk utilization (per physical disk).

Table 6.2. Metrics obtained from the IOSTAT command

Label	Name	Description	Category
m21	disk1/rps	Read per second per disk	I/O
m22	disk1/wps	Write per second per disk	I/O
<i>continued on next page</i>			

Table 6.2 (cont'd).

Label	Name	Description	Category
m23	disk1/util	Percentage of disk utilization per disk	I/O
m24	disk2/rps	Read per second per disk	I/O
m25	disk2/wps	Write per second per disk	I/O
m26	disk2/util	Percentage of disk utilization per disk	I/O
m27	cpu/us	Report the percentage of time the system has spent in user mode.	CPU utilization
m28	cpu/sy	Report the percentage of time the system has spent in system mode	CPU utilization
m29	cpu/wt	Report the percentage of time the system has spent waiting for I/O	CPU utilization
m30	cpu/id	Report the percentage of time the system has spent idling.	CPU utilization

Vmstat stands for virtual memory statistics. This command reports aggregate information about virtual memory statistics in the system. Those metrics we observed from the *vmstat* command are shown in table 6.3

Table 6.3. Metrics obtained from the VMSTAT command

Label	Name	Description	Category
m31	memory/swap	Usage of virtual and real memory. Amount of swap space currently available (Kbytes)	Virtual Memory Statistic
<i>continued on next page</i>			

Table 6.3 (cont'd).

Label	Name	Description	Category
m32	memory/free	Usage of virtual and real memory. Free size of the free list (Kbytes)	Virtual Memory Statistic
m33	page/re	Page reclaims per second.	Paging activity
m34	page/mf	Minor faults per second.	Paging activity
m35	page/pi	Kilobytes paged in per second.	Paging activity
m36	page/po	Kilobytes paged out per second.	Paging activity
m37	page/fr	Kilobytes freed per second.	Paging activity
m38	page/sr	Pages scanned by clock algorithm per second.	Paging activity
m39	disk/s0	Disk operations per second.	Disk
m40	disk/s1	Disk operations per second.	Disk
m41	faults/in	Trap/Interrupt rates per second. Non-clock device interrupts.	Memory faults
m42	faults/sy	Trap/Interrupt rates per second. System calls.	Memory faults.
m43	faults/cs	Trap/Interrupt rates per second. CPU context switches.	Memory faults.
m44	cpu/us	Percentage usage of CPU time. Av- erage across all processors. User time.	CPU utilization.
m45	cpu/sy	Percentage usage of CPU time. Av- erage across all processors. System time.	CPU utilization.
m46	cpu/id	Percentage usage of CPU time. Av- erage across all processors. Idle time.	CPU utilization.

6.2.3 Output Format

When we collect profiling data from Guide, we get is a scalar containing aggregate information about the runs executed. We have programmed some perl scripts to convert the data from the format established by Guide to a format appropriate for the statistical analysis tools.

Also, every 20 seconds we collected information from the operating system in form of a vector containing all measurements. At the end of one experimental run, a matrix containing all measurements from the time the application started running to the end of the run was generated. A vector containing the average of all metrics was generated per experimental run. A matrix containing all experimental runs from one experiment will be created. All these manipulations were done using perl scripts. Some of these scripts are shown in appendix J.

6.3 Summary

The data collection step of the methodology is the only one tied to the specific architecture and software available on the particular platform where the software runs. Data can be collected in the form of a profile as a scalar, or in the form of a trace as a vector. Instrumentation can be inserted at programs, operating system, and processor level. For software instrumentation data can be collected inserting instrumentation at the source code, compiler, object code, library, executable, and running code. We have inserted instrumentation at the library level. We have also used operating system instrumentation. A description of the metrics used for data collection was given in this chapter.

CHAPTER 7

Data Analysis

7.1 Introduction

The last step in the integrated methodology for obtaining information across levels is data analysis. This is depicted in Figure 7.1.

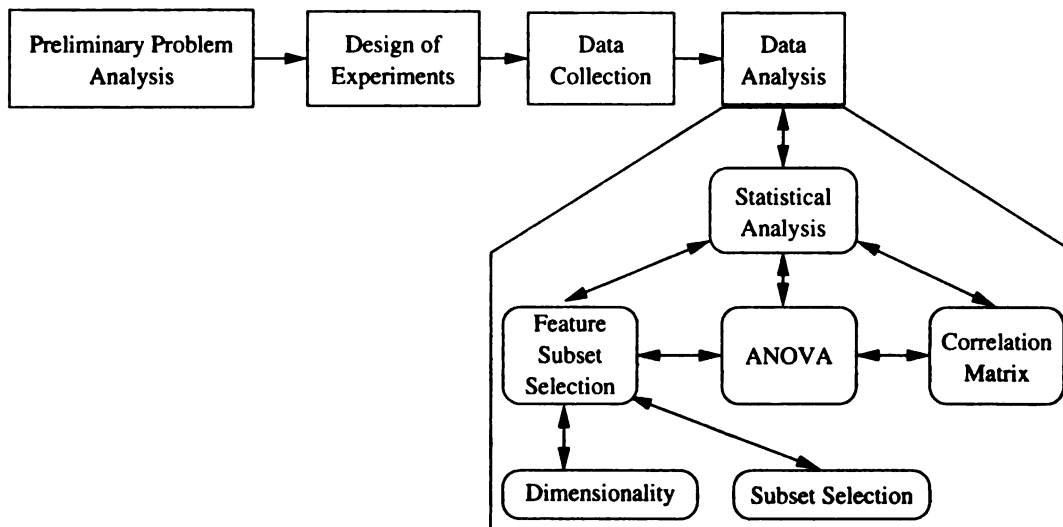


Figure 7.1. Data Analysis is the last step in the proposed methodology.

Performance data collected during experimentation will not yield useful information until it is carefully analyzed. Statistical methods are the basis for data analysis. Specifically,

three statistical techniques were used in our case study: correlation analysis, multidimensional data subset selection, and analysis of variance. It should be mentioned, however, that additional methods may be used for data analysis to extract information. For example, multiple regression analysis might be used to model the outcome from the system.

This process can be seen as parallel to the steps for a knowledge discovery process. The goal of knowledge discovery is to effectively transform raw data into information [2]. Knowledge discovery is composed of several steps [64], which include: data preprocessing, data reduction, modeling and hypothesis selection, and data mining.

In this chapter we describe the statistical data analysis used for our study.

7.2 Statistical Models for Performance Analysis

Statistical analysis provide a powerful tool for evaluating and interpreting data objectively. Statistical methods have been used for performance data analysis and interpretation for a long time [26, 27]. However, the proposed integrated use of design of experiments, correlation analysis, and feature selection has not been used for automatic performance evaluation.

Statistical methods are the basis for analysis of multivariate data. Three distinct statistical techniques are combined in our study to assess the status of the system performance: correlation, multivariate analysis, and analysis of variance (ANOVA).

The use of design of experiments techniques in our experimentation process prompts the use of ANOVA or any general linear model on the data. We have used correlation analysis and ANOVA to establish relations in the data. Design of experiments techniques have been used in the past in the area of computer performance for analyzing the behavior of algorithms and heuristics in terms of execution time [3], scalability [11], and for performance evaluation of memory hierarchies [12, 19]. We introduce its use for the empirical study of performance instrumentation data.

We were particularly interested in correlating high-level programming decisions to low-level metric information.

7.3 Measuring Relationships in Multidimensional Data

A block diagram of the components of a performance data analysis system is shown in Figure 7.2. It follows the structure of a general pattern recognition system [65].

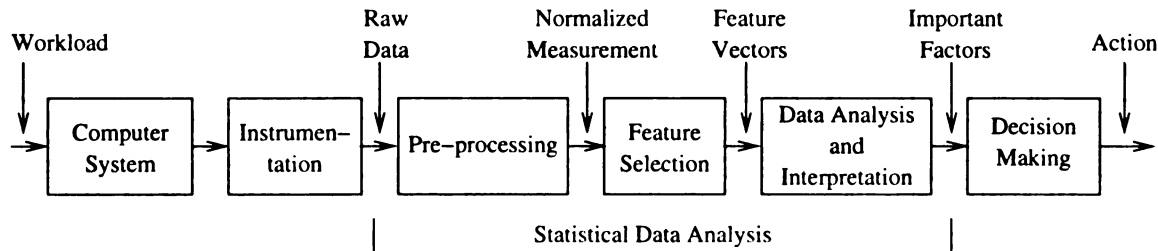


Figure 7.2. Performance Data Analysis Architecture.

Statistical methods used for data analysis begin with the raw data and conclude with the information obtained about the factors affecting the system performance.

After data collection, we have a set of measurements drawn from an observable computing system (OCS) which are a sample from a stochastic process. It is important to emphasize the stochastic nature of the performance data since variations in the outcome from the measurements might be significant (due to real factors affecting the system), or insignificant (due to the random nature of the process itself, and therefore, not important). The time variable might be discrete or continuous. Similarly, the value might be countable or uncountable, giving rise to four types of stochastic processes: discrete time-discrete value, discrete time-continuous value, continuous time-discrete value, and continuous time-continuous value processes. Figure 7.3 shows a graphical depiction of a discrete-time continuous-value random process. Tracing data measurements taken in our system come from a discrete-time continuous-value random process.

We assume that this process is mean-square ergodic in the mean. The ergodicity assumption is needed to compute the averages of the metrics.

We have selected a very small problem size for our experiments due to time constraints.

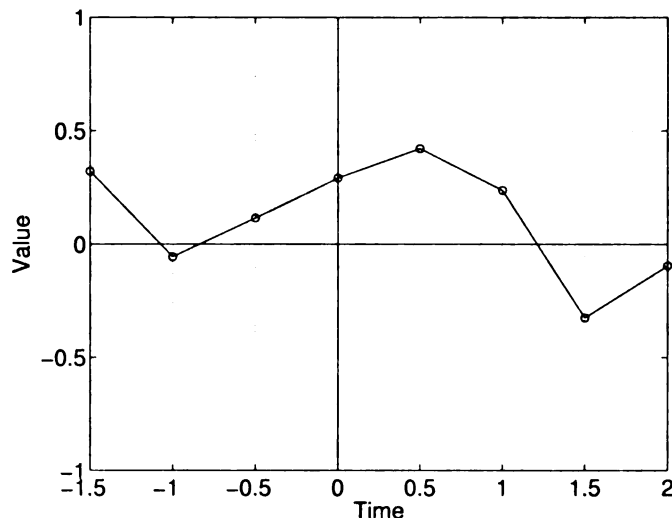


Figure 7.3. Graphical View of a Discrete-Time Continuous Value Stochastic Process

However, typical running times for this code is in the order of days, and depends on the physical characteristics of the antenna to be analyzed.

7.3.1 Formatting Data for Statistical Methods

Data obtained from an instrumentation system come in different formats. If data are received in the form of summaries or aggregate information, like those obtained from KAP/Pro, each experimental run will provide a set of average measurements of the variable of interest or an absolute value. Some examples of these types of measurements, taken by KAP/Pro, are percentage of imbalance time per thread, percentage of barrier time per thread, CPU time, and idle time. Another type of measurement is obtained as a trace file. Here, measurements are either taken at specific time intervals or a times triggered by an event of interest in the system. Operating system metrics on Unix or Linux systems taken with the `sar`, `iostat`, `vmstat`, or `mpstat` commands are examples of measurements taken at specific time intervals. In these commands, one of the arguments is the sampling time at which all values will be reported, allowing for specifying the recording times. Other tools, such as Vampirtrace or MPE, will trace MPI activities at the instant of time when they actually occur, and not at regular time intervals.

This leads to three different types of variables to manipulate: absolute, average, and regularly sampled measurements. Absolute measurements are variables which record a total count of an event or a real number representing total time. Average measurements are values computed by the instrumentation system where the system itself records a total number and performs an average calculation on the overall measurement and presents this number. Finally, regularly sampled measurements are recorded at regularly spaced sampling times.

In order to support data analysis using statistical methods, the data obtained from experiments should be formatted as a matrix. Each experiment consists of a series of experimental runs, each supplying a set of measurements. These measurements may come in the form of a random number in the case of an average or absolute value, or as a sample of a discrete time - continuous value random process in the case of trace data. This type of trace data can be regarded as time series and can be modelled as piecewise independent stochastic process [66]. When data is presented in the form of a time series, the temporal average of the realization is computed to estimate the statistical average, assuming the process is mean-ergodic. The temporal average is computed by

$$\mu_x = \left(\frac{1}{N} \right) \sum_{i=0}^{N-1} x[i] \quad (7.1)$$

where N is the number of points in the time series [67].

For one experiment, one matrix is formed by all measurements. Each element is either a temporal average or an absolute metric value. Page faults per second is an example of a temporal average metric. We will denote an average metric as m^{avg} . An absolute value, denoted m^{abs} is a metric whose value is obtained as a total value at the end of the execution time. It could be either a discrete value or a continuous value. Total CPU time is an example of an absolute metric.

One experiment consists of R experimental runs executed in a pre-specified order. This order is randomized during the experiment specification phase of the methodology to minimize the number of experiments required to obtain useful information. The randomization scheme determines the precision of the results obtained. Let P denote the number of per-

formance metrics measured during an experimental run, k denotes the experimental run where $0 \leq k \leq K - 1$, and p the metric identification number, where $0 \leq p \leq P - 1$. Let M denote following performance data matrix:

$$M = \begin{bmatrix} m^a[0,0] & m^a[0,1] & \cdots & m^a[0,P-1] \\ m^a[1,0] & m^a[1,1] & \cdots & m^a[1,P-1] \\ \vdots & \vdots & \ddots & \vdots \\ m^a[K-1,0] & m^a[K-1,1] & \cdots & m^a[K-1,P-1] \end{bmatrix}, \quad (7.2)$$

where $M \in l^2(\mathbb{Z}_K \times \mathbb{Z}_P)$. In a more compact form, $M = \{m^a[k,p] : k \in \mathbb{Z}_K; p \in \mathbb{Z}_P; a = \text{abs or } a = \text{avg}\}$. We can also use the notation $m^a[k,p] = m_k^a[p] = m_p^a[k]$, where $m^a[k,p]$ denotes the average or absolute metric value for experimental run k and metric p , and a is either *avg* or *abs*. Note that each column of the performance data matrix M consists of measurements of one performance metric over a set of experimental runs and each row is a P dimensional vector containing all different measurements for one given experimental run. The notation $m_k^a[p]$ refers to one row of the matrix and $m_p^a[k]$ to one column of the matrix.

For example, if we perform one experiment consisting of four experimental runs, measuring *ExTime*, *pgfaults/s*, *cachehits/sec*, *cachemisses/sec*, and *idleTime*, then matrix M will be a 4×5 matrix with the format shown in Figure 7.4:

$$M = \begin{array}{ccccc} \begin{array}{c} \text{metric 0} \\ \downarrow \\ \text{ExTime(0)} \\ \text{ExTime(1)} \\ \text{ExTime(2)} \\ \text{ExTime(3)} \end{array} & \begin{array}{c} \text{metric 1} \\ \downarrow \\ \text{pgfaults/sec(0)} \\ \text{pgfaults/sec(1)} \\ \text{pgfaults/sec(2)} \\ \text{pgfaults/sec(3)} \end{array} & \begin{array}{c} \text{metric 2} \\ \downarrow \\ \text{cachehits/sec(0)} \\ \text{cachehits/sec(1)} \\ \text{cachehits/sec(2)} \\ \text{cachehits/sec(3)} \end{array} & \begin{array}{c} \text{metric 3} \\ \downarrow \\ \text{cachemisses/sec(0)} \\ \text{cachemisses/sec(1)} \\ \text{cachemisses/sec(2)} \\ \text{cachemisses/sec(3)} \end{array} & \begin{array}{c} \text{metric 4} \\ \downarrow \\ \text{idleTime(0)} \\ \text{idleTime(1)} \\ \text{idleTime(2)} \\ \text{idleTime(3)} \end{array} \end{array} \begin{array}{l} \leftarrow \text{run 0} \\ \leftarrow \text{run 1} \\ \leftarrow \text{run 2} \\ \leftarrow \text{run 3} \end{array}$$

Figure 7.4. Example of a matrix format used for the performance data.

When the matrix is formatted in this fashion, it is ready to apply statistical techniques, such as: correlation, ANOVA, and feature subset selection. Before applying any of these methods, preprocessing is required on this type of data.

7.3.2 Preprocessing

Most statistical techniques are biased by the magnitude or order of the data to be processed. For example, principal component analysis is not scale invariant. When the values are measured in largely different scales, principal component analysis is biased towards the largest values [68]. In pattern recognition, since these techniques are applied mostly to images, and pixels in images have values in the range of 0 to 255, problems with differences in magnitude are not typically found. However, in areas such as data mining [69, 70] and sensor data [71], the magnitude of a measurement may differ by a large amount from another measurement. This is similar to our case. For example, variable `pswch/s` (process switches per second) is in the order of thousands, while `disk/util` (disk utilization) is in the (0,1) range. This problem is mitigated by scaling the inputs to the statistical methods.

Normalization

Data normalization makes all measurements comparable to all subsequent methods, therefore making sure that statistical methods are not sensitive to large changes in the scales of the data. Global normalization methods are applicable to performance data. Different methods are available for normalization, among which we considered:

- **Absolute:** The data is not preprocessed and it is left as it is.
- **Log Normalization:** The log of each element of the performance data matrix M is computed. We will denote the normalized matrix by N . This technique was used by Nickolayev, Roth, and Reed in [13] to extend the dynamic range of the data.

$$n^a[k, p] = \log(m^a[k, p]) \quad (7.3)$$

- **Min-max normalization:** This is a linear transformation transforming each metric measurement to the $(0, 1)$ range [70, 72].

$$n^a[k, p] = \frac{m^a[k, p] - \min_k[m_p^a[k]]}{\max_k[m_p^a[k]] - \min_k[m_p^a[k]]} \quad (7.4)$$

- **Dimension normalization:** Each metric vector is divided by its Euclidean norm so that it is forced to lie on a hypersphere of unit radius [65].

$$n^a[k, p] = \frac{m_p^a[k]}{\text{Euclid_Norm}(m_p^a[k])} = \frac{m^a[k, p]}{\sqrt{\sum_{k=0}^{K-1} (m^a[k, p])^2}} \quad (7.5)$$

To determine how good a normalization scheme is, we can project the multivariate data using principal component analysis along the two principal components. We have used the validation experiment using matrix-vector multiplication for this purpose. In this experiment we can identify two classes based on execution time: acceptable and poor performance. Good execution time is on the order of 10 seconds while bad performance execution time is on the order of a 100 seconds. We need to remember that a good feature should separate the classes as far as possible. When we project the data along the two main components, a good normalization scheme should separate the classes. All three different normalization schemes were used on the validation data to test which one gives the best separation.

Results show that when using no normalization, as expected, those metrics with the highest range of values were biasing the results. Figure 7.5 shows the projection along the two principal components of the data. It can be observed that the two classes are mixed in the plot, therefore the classes cannot be easily distinguished from the figure.

Log normalization cannot be applied to our data set. This data set contains several values of zero. This type of normalization is applicable to the case where measurements are always greater than one. In the case of [13], it is applied on performance data filtered using a low pass filter and averaged out over a sliding window.

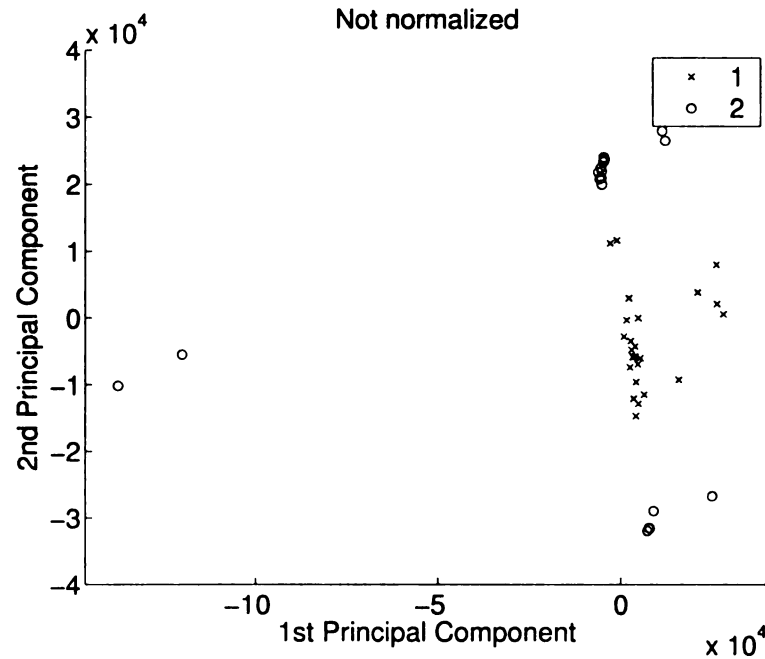


Figure 7.5. Two principal components of the validation data - no normalization

Third, min-max normalization was applied to the data. Figure 7.6 shows the two main components of this data set for the validation experiment. The discriminatory effect of this normalization is seen from the figure, where all values of class 1 are located at the left in the graph and all values of class 2 are located at the right side of the graph. However, there are two small clusters of class 1, one at the top of the graph and one at the bottom. A desirable discriminatory separation should gather all elements of one class together.

Finally, Euclidean normalization was applied to the data. Figure 7.7 shows the projection along the two main components of the data. It distinguishes the two classes and shows all components of class 1 together (except for one outlier), and all components of class 2 together. This is a better normalization for the validation data.

From all different normalization schemes, the one which presented a better discrimination power was Euclidean normalization and it is appropriate for the types of data we have in performance analysis. Therefore we have used Euclidean normalization in all our data sets. This normalized matrix N is then analyzed using other statistical methods.

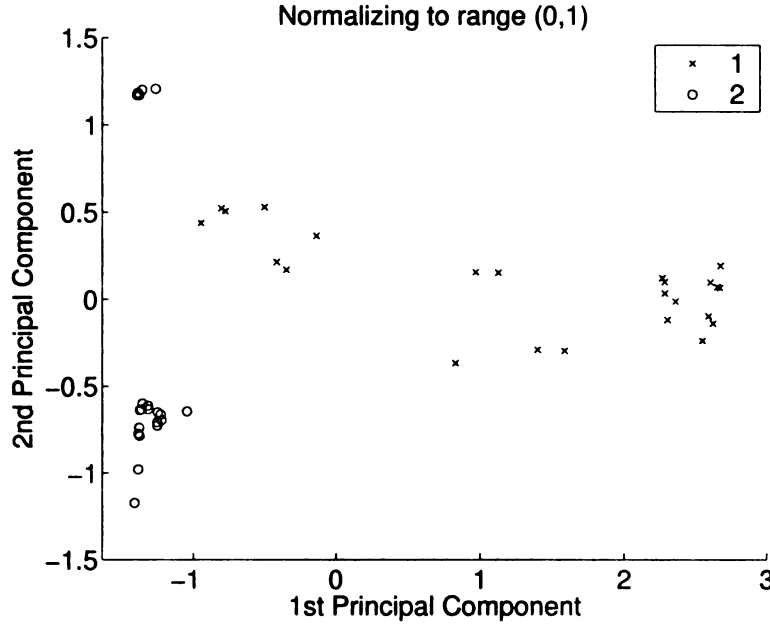


Figure 7.6. Two principal components of the validation data - Min-Max normalization

7.3.3 Correlation Analysis.

The degree of association between two variables, if any exists, is obtained through correlation, which measures the linear association among variables. No causal relationship can be made about correlated variables. This computation is not appropriate for nominal or ordinal variables. Some assumptions are required to hold for correlation analysis: the variables are random, the relationship is linear in nature, and the variables follow a normal distribution [9]. We define a random variable as a function whose domain is a sample space of an experiment and whose range is a subset of the real line [9, 67].

The *product moment correlation coefficient* or *correlation coefficient* measures the linear correlation among pairs of variables and it is denoted by ρ (rho).

The estimate of the correlation coefficient between two variables $x = m^a[k, p_i]$ and $y = m^a[k, p_j]$ is denoted by r and is computed as

$$r = \frac{\sum_{i=1}^K (x_i - \bar{x})(y_i - \bar{y})}{(K - 1) S_x S_y}, \quad (7.6)$$

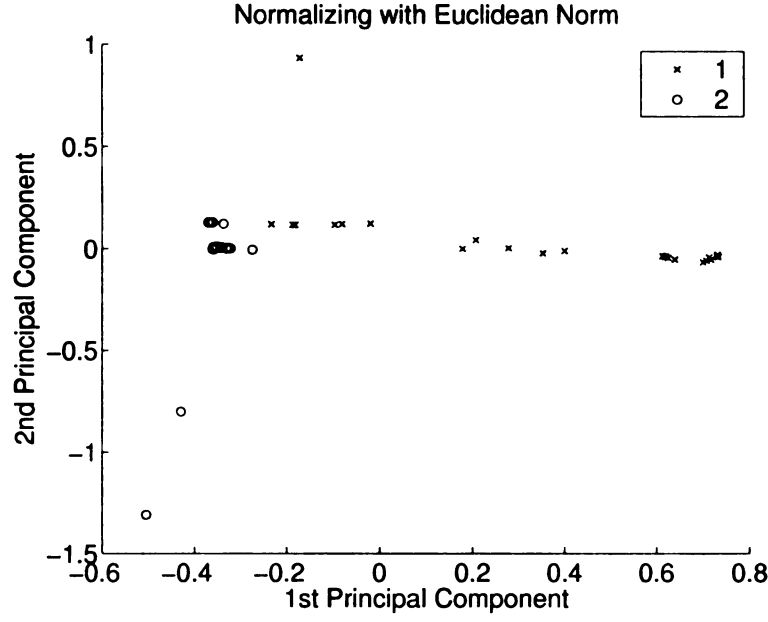


Figure 7.7. Two principal components of the validation data - Euclidean normalization

where $\bar{x}$ is the mean of x , that is, $\bar{x} = \frac{\sum_{i=1}^K x_i}{K}$, $\bar{y}$ is the mean of y , $\bar{y} = \frac{\sum_{i=1}^K y_i}{K}$, S_x and S_y are the sample estimate of the standard deviation of x and y , respectively, that is

$$S_x = \sqrt{\frac{\sum_{i=1}^K (x_i - \bar{x})^2}{K - 1}}, S_y = \sqrt{\frac{\sum_{i=1}^K (y_i - \bar{y})^2}{K - 1}}, \quad (7.7)$$

and K is the number of observations, or in this case, the number of experimental runs [9]. The value of r ranges from -1 to 1 where -1 indicates perfect negative correlation, $+1$ indicates perfect positive correlation, and 0 indicates no correlation. Correlation will indicate to what extent two variables vary simultaneously. When more than two variables are involved, a correlation matrix is used.

Let $\mathbf{N}$ denote the matrix containing the normalized observation data. Rows of matrix $\mathbf{N}$ contain observations and columns of $\mathbf{N}$ contain variables. Let $\mathbf{S} = \mathbf{x}'\mathbf{x} - \frac{1}{n}(\mathbf{x}'\mathbf{1})(\mathbf{1}'\mathbf{x})$, where $\mathbf{1}'$ denotes a unit row vector, $\mathbf{1}$ denotes a unit column vector and $'$ denotes the transpose operation. Then $\mathbf{R} = \frac{1}{n-1}D^{-\frac{1}{2}}SD^{-\frac{1}{2}}$, where $D^{-\frac{1}{2}}$ is a diagonal matrix whose entries along

the diagonal are the reciprocals of the standard deviations of the variables in $\mathbf{x}$. That is, $D^{-\frac{1}{2}} = \text{diag} \left(\frac{1}{s_{x_1}} \quad \frac{1}{s_{x_2}} \quad \dots \quad \frac{1}{s_{x_n}} \right)$ [68]. $\mathbf{R}$ is called the correlation matrix. Elements (i, j) in $\mathbf{R}$ are the correlation coefficients between row i and column j , therefore matrix $\mathbf{R}$ is symmetric and its diagonal elements are equal to 1.

If we use dimension normalization, the correlation matrix can be obtained either from matrix M or from matrix N and the result is the same matrix R . This is not true for log or min-max normalization. If $x_1 = W_1x$ and $y_1 = W_2y$ then $\overline{x_1} = W_1\overline{x}$ and $\overline{y_1} = W_2\overline{y}$. Also $S_{x_1} = W_1S_x$ and $S_{y_1} = W_2S_y$. Therefore, the computation of each r is the same for every pair of variables in the matrix.

Why is it interesting to obtain a correlation matrix from our data set? High positive or negative correlation will indicate clearly that two variables are linearly related. A variable that is highly correlated with execution time is clearly a target variable to observe since there might be a causal relation among this variable and execution time itself. Also, if there are any coefficients of value +1 or -1 off the diagonal of R , this means that the two variables involved are exactly the same variable with different names, or a multiple of another variable and one of them should be eliminated from the analysis. Please note that when the correlation coefficient of two variables is one, either the two columns are the same measurement or a linear combination of measurements, therefore the data matrix is singular, not invertible, and the solution of any linear system of equations relating these variables is not possible to compute. Many statistical methods do not apply in this case.

We have computed matrix R from one of our data sets. A visual display of a correlation matrix obtained in one of our experiments is shown in Figure 7.8. This is from the validation experiment using matrix-vector multiplication.

In this case, the metrics most correlated with execution time were those related to **paging activity/faults** and **paging activity/page reclaims** and **percentage of the time the CPU was idle**. In these cases, the correlation was negative, indicating that as the metric value increased, the execution time decreased.

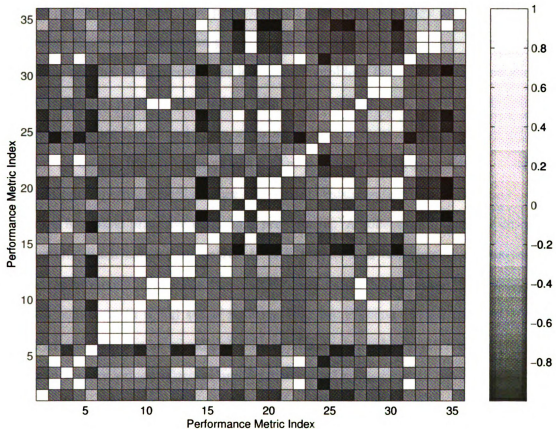


Figure 7.8. Visual display of the correlation matrix of the data obtained from the validation experiment `matrix-vector multiplication`.

7.3.4 Multidimensional Metric Subset Selection.

If we recall the definition of $\mathbf{N}$ from the previous section, columns of $\mathbf{N}$ are normalized vectors containing measurements obtained for each specific metric, and each element of this vector corresponds to an observation. That is

$$\mathbf{N} = \begin{bmatrix} n_p^a[0] & n_p^a[1] & \cdots & n_p^a[K-1] \end{bmatrix}, \quad (7.8)$$

where $n_p^a[k]$ is a column vector containing a set of normalized measurements for performance metric k .

For parallel code running on high performance systems, the number of metrics, K , is

extremely large [15]. This has prompted the use of visualization tools to aid programmers understand performance data. The programmer's interpretation plays an integral role in understanding the relations between performance data and code.

The large number of measurements has been a problem faced in other areas such as artificial intelligence, pattern recognition, and data mining [73, 30, 31, 74]. The solution to this problem is not trivial and many approaches have been proposed. We have selected the method we believe is most appropriate for the problem of high-performance computer metrics data and automatic performance evaluation: feature subset selection.

A *feature* is a basic primitive defining a problem [2]. A collection of features describes an application. Each column of matrix M is a feature and contains measurements for one performance metric. Therefore, we identify each metric as a feature. The amount of information contained in a full set of features may be occluded by the large amount of metrics. This problem can be solved either by two methods used in pattern recognition and data mining: feature subset selection and feature subset extraction. In the first method, a subset of the original set of features is selected. In the second, a combination of features is used to generate new features, which in turn contains a smaller number of features than the original one.

Let F denote a set of features containing K features. The problem of feature selection would be defined as finding the optimal set of p features such that $p < K$ (see Figure 7.9). What is considered to be optimal? It means to optimize a cost function $J(\cdot)$ selected by the objective of the selection. In feature extraction, the goal is to use a transformation W such that the new set WF of transformed features contains q features such that $q < K$ and it is the best transformation in terms of optimizing a cost function $J(\cdot)$. In our case, feature selection is more suitable than feature extraction since feature selection is done in the measurement space, therefore, the physical meaning of a specific feature is not lost in the selection process. In feature extraction, a transformation is applied to the metrics involved in the calculation and therefore new metrics are created, not necessarily having a physical meaning.

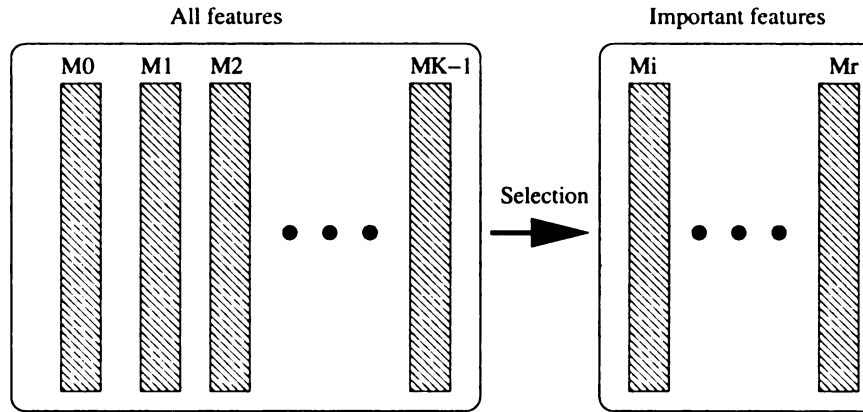


Figure 7.9. Feature subset selection

Feature selection has also been defined as the process of selecting relevant features for a particular task. A feature is relevant if when it is removed from the set, the set will deteriorate. This is a function of the measure selected as objective function or cost function [2]. In Artificial Intelligence, improving the learning process is the goal so relevant features are the ones that are required for learning [75]. From [76] the relevance of a feature is defined in terms of the classification problem.

The feature selection process has several benefits. First, it reduces data redundancy. As explained before, when there is a problem of collinearity of two metrics, the matrix of data is not invertible, and some statistical methods do not apply. Second, it can aid in finding natural groups in the data [77]. Third, it allows a better understanding of the data. When feature selection is used for classification, it minimizes the *curse of dimensionality*. The curse of dimensionality refers to the fact that as the number of features increases, the number of observations required for classification grows exponentially, creating the need of enormous amounts of observations for proper classification. Feature selection can alleviate this problem.

There are three basic questions we need to answer in order to apply feature selection to any set of data. First, we need to identify the search process for finding the optimal features. Second, what is the criteria for determining the best set, that is, what is the cost function to use for evaluation. Finally, what strategy is going to be used to add or delete

features to the current subset. These questions have been addressed in this study for the particular case of performance metrics. Notice that our goal is not to improve classification since we do not have classes in our data. Our main goal is to simplify our results so that it improves the comprehensibility of the data.

Three basic search methods for feature selection [2, 78] are exhaustive search, heuristic search, and nondeterministic search. In exhaustive search, all possible solutions are examined and compared. This method is time consuming and sometimes unfeasible, given the large amounts of data to be processed. Heuristic search or *weak methods* refer to a guided search where not all possibilities are exhausted. It may lose some optimal solutions but in general it finds good solutions. These searches are faster than exhaustive search. Finally nondeterministic search refers to finding possible solutions at random and evaluating them. Given the amount of data in typical performance evaluation problems, exhaustive search becomes unfeasible due to time constraints. Both heuristic search and random search are applicable to our problem. Heuristic search was used in our case study since it is the most common method used in pattern recognition. Future analysis may include evaluating random search methods.

The search direction in which a subset of features is generated depends on the final number of features desired in the subset (p) compared to the total number of features (K). If we do not have knowledge of p , any search direction is possible. There are three different possibilities for search direction: sequential forward search, sequential backward search, and bidirectional search. In sequential forward search, features are added, one at a time, to the final subset based in the desired cost function. This is an appropriate method if $p \ll K$ making the search time smaller. In sequential backward search, features are discarded one at a time based on the cost function. Irrelevant features are discarded. When $p \not\ll K$, the search time is smaller with this method. Finally, bidirectional search is appropriate when p is unknown. In our case, we need to identify p before starting the feature selection process. However, results show that the obtained values of p are much smaller than K , therefore sequential forward search is preferred over backward search methods.

The last question that remains unanswered is what cost function is appropriate for our problem? In order to answer this question, we have explored the possibilities from [2]. In this work, they have classified existing measures for feature selection. Figure 7.10 shows the taxonomy for measures used for feature selection methods.

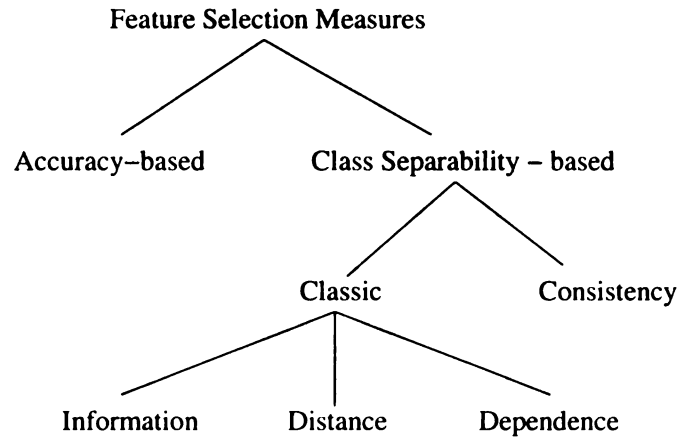


Figure 7.10. Classification scheme of feature selection measures [2]

Accuracy measures are those based on the accuracy of the classifier used for data classification. This means that they are based on a specific classifier. This does not apply for our case since we are not having a classification problem. For our case, class separability methods are more appropriate. These are based on how to separate the data into its natural groups. These are subdivided into classic measures and consistency measures. From those, consistency measures are also discarded. It refers to maintaining a minimum consistent set where all instances are classified as one class without any inconsistencies. It has to do with classification itself. We are left with three classic measures: information, distance, and dependence. All these three are appropriate measures for our data. Information measures select the subset based on those features which minimize uncertainty. Distance measures are those which try to separate classes as far as possible using a distance function. Dependence measures select features on association with interesting variables. We have selected information measure for our study.

The feature selection problem can be further subdivided into supervised or unsupervised selection. There is vast literature in supervised feature selection methods [75, 76, 74, 2, 73, 30, 79]. Supervised feature selection refers to methods devised when we have data to train the system, that is, we know instances where there is a class associated to each instance, therefore we know what the *correct* classification of an instance is. Supervised learning is subdivided into two different models according to the measures used to obtain the metrics: wrapper and filter model. The wrapper model [76] uses the classifier accuracy as cost function. The filter model [80] is independent of the classification method. Measures based on distance or information are used and the results are based on the data itself. Both filter and wrapper methods are composed of four parts: feature generation, feature evaluation, stopping criteria, and testing. Feature generation and evaluation have been previously discussed. Stopping criteria is deciding when will the search process will stop. Will it be based on a threshold, a criterion, or a specific number of features to select? Testing refers to how to evaluate the accuracy of the results. The wrapper method can be viewed as a machine learning approach and the filter method can be seen as a data mining approach [2].

Even if there is vast literature in supervised selection, our problem is classified into unsupervised selection. We do not have a specific class associated to performance data metrics. A possible class associated to the data would be: acceptable performance and not acceptable performance. But this is very subjective. According to Liu and Motoda in [2], there are two different methods for unsupervised learning: clustering and entropy based methods. In clustering, features are grouped together according to some measure. Ahn and Vetter in [15] have used clustering to identify which metrics are relevant. When clustering is used for feature selection, an ordered list of features based on relevance is not obtained. Methods based on entropy can rank features and are used for unsupervised feature selection. We have used the entropy based method described in [81] since it is based on the concept of information about the system.

Entropy based methods

In his classic paper [82], Shannon described a communication system as composed of source of information, a transmission medium, and a receiver. According to Shannon, a signal is an entity which carries information. Information is anything that can be sent from one point to another in the physical world. He also introduced a measure of the amount of information contained in a message: *entropy*. Given its random nature described above, an observable computing system is a source of information. We would like to reconstruct the message that the system is giving us to improve the system's performance.

Let X denote a discrete random variable. All possible values of X are $x_0, x_1, \dots, x_{n-1}$ with probability mass function $P_{\mathbf{x}}(x_i)$, where $P_{\mathbf{x}}(x_i) = P[X = x_i] = p_i$. Entropy will measure the average amount of uncertainty of the random variable and it is computed by

$$H(X) = - \sum_i p_i \log p_i, \quad (7.9)$$

where n is the number of possible values of the random variable [83]. When the log function is base two, information is measured in bits. Notice that the more probable a message is, it contains the less information. An unusual message contains more information than a regularly received message. Relative entropy measures the distance between two probability distribution functions. The joint entropy of random variables X and Y with the distribution function $p(x, y)$ is defined as $H(X, Y) = - \sum_x \sum_y p(x, y) \log p(x, y)$. The conditional entropy of random variable X and Y following the distribution function $p(x, y)$ is defined as $H(Y|X) = \sum_x \sum_y p(x, y) \log p(y|x)$. Mutual information is a measure of the amount of information a variable has about another variable. It is defined as $I(X; Y) = \sum_{x,y} p(x, y) \log \frac{p(x,y)}{p(x)p(y)} = H(X) - H(X|Y)$. This means that it is the amount of information contained in X minus the information of X given that Y is known. Also $I(X; Y) = H(X) + H(Y) - H(X, Y)$.

The measure of entropy used in this work is:

$$E = - \sum_{i=1}^{N-1} \sum_{j=i+1}^N (S_{ij} \times \log S_{ij} + (1 - S_{ij}) \times \log (1 - S_{ij})) \quad (7.10)$$

where S_{ij} is called the similarity value of two instances. It is defined as $S_{ij} = e^{-\alpha \times D_{ij}}$ where D_{ij} is the Euclidean distance between instances x_i and x_j . Alpha is an empirical value computed by the data and it is defined as $\alpha = \frac{-\ln 0.5}{\bar{D}}$, and $\bar{D}$ is the average distance of all the metrics.

This was the cost function used for our search. We selected sequential forward search as the search strategy and entropy as the cost function. Results are shown in chapter 8. Now a question arises: how many metrics are required to explain the behavior of the system? Feature selection methods assume that the number of features to select, or its equivalent, the number of clusters to identify, k has been previously defined. To answer this question we have addressed the problem of data dimensionality estimation techniques.

Dimensionality

Dimensionality has been studied for a long time [31, 32, 84, 85]. The dimension of the data is important to determine the number of features required to represent the data. There are two main definitions of dimension used in pattern recognition: spanning dimension, and intrinsic dimension [86, 31]. Intrinsic dimension comes from linear algebra, and it is defined as the smallest set of vectors required to span the data set. It is also known as embedding dimension. The spanning dimension corresponds to the smallest number of parameters required to model the data without degrading the data set [31]. Intrinsic dimension is appropriate for determining the number of features representing the data [31]. The difficulty resides in estimating the intrinsic dimension from a data set.

A series of classical methods have been used to estimate intrinsic dimension. These are based on principal component analysis of the data.

- **Cumulative Percentage of Total Variance:** The most popular method for determining intrinsic dimension is called the cumulative percentage of total variation method or also known as K-L algorithm [87, 68]. It is based on the computation of

the eigenvalues of the correlation matrix of the data. Each eigenvalue contributes to a percentage of the total variance. Those eigenvalues whose eigenvectors explain most of the variance are selected. The number of eigenvalues required to reach certain threshold in terms of percentage are selected as k . A typical threshold is 95% of the total variance.

- **Kaiser-Guttman:** The eigenvalues of the correlation matrix of the data are computed. Those eigenvalues greater than one are selected and the number of eigenvalues greater than one is k [87, 88].
- **Scree test:** The eigenvalues of the correlation matrix of the data set are sorted in descendent order and plotted. The point where the curve flattens is selected as the cutoff point, and this is the number of principal components to select. This estimates the intrinsic dimensionality of the data and it is the value k [87].

The first two methods are appropriate for automatic performance analysis since the computation of k occurs without the programmer's intervention. In scree test, a visual inspection of the graph showing the eigenvalues of the correlation matrix is required, making the method not automatic. All three methods were used in our data analysis for reference purposes.

There are additional methods for dimensionality estimation to explore for future work. In recent literature, Dy and Brodley studied the order identification problem from the wrapper model point of view. They wrap the search of k around the clustering algorithm for unsupervised learning [29, 77, 89], computing k as well as the feature subset at the same time. We have not used these methods since they are based on a classifier and we are working with unsupervised data.

Once the important metrics describing the system are identified, we use ANOVA to analyze the results.

7.3.5 ANOVA

When an experiment is designed using DOE techniques, causal relations can be established among independent factors and results [9]. Analysis of Variance (ANOVA) is a technique used in this cases to determine if the differences in the results obtained are due to chance or to significant effects of the controlled factors (see Section A.5.2). We proceed to explain ANOVA for the 4-way factorial design and the split-plot designs used in our case-study.

We will explain ANOVA using one of our experimental results. In experiment four, we conducted a validation experiments using matrix-vector multiplication algorithms. We designed a full factorial experiment with four factors: problem size, algorithm, data structure, and compiler options. There were two problem sizes, three algorithms, two data structures, and four compiler options. We wanted to study the effects of these factors on the set of computer performance metrics obtained from our system. Since there are four factors, the four-way ANOVA model for this experiment is

$$Y_i = \mu + A_i + B_j + C_k + D_l + A_i B_j + A_i C_k + A_i D_l + B_j C_k + B_j D_l + \quad (7.11)$$

$$+ C_k D_l + A_i B_j C_k + A_i B_j D_l + A_i C_k D_l + B_j C_k D_l + A_i B_j C_k D_l + \epsilon \quad (7.12)$$

where A_i represents the effect of problem size, B_j effect of algorithm for matrix-vector multiplication, C_k effect of data structure, and D_l the effect of compiler options. Additional terms account for the interactions among factors.

One of the important performance metrics selected by the feature selection method previously defined in section 7.3.4 for this experiment was **memory/free**. We used the software SAS for data analysis and we are showing its output for illustrative purpose. We compared **memory/free** based on the factors previously described and we obtained the following results:

Dependent Variable: memory_free

Sum of					
Source	DF	Squares	Mean Square	F Value	Pr > F

Model	47	45030792685	958101972	1.14	0.3308
Error	48	40488348470	843507260		
Corrected Total	95	85519141155			

R-Square	Coeff Var	Root MSE	memory_free Mean
0.526558	2.253901	29043.20	1288575

Source	DF	Type I SS	Mean Square	F Value	Pr > F
Size	1	2833428803	2833428803	3.36	0.0730
CompOpt	3	227278711	75759570	0.09	0.9653
Size*CompOpt	3	4874495705	1624831902	1.93	0.1379
Alg	2	10491710221	5245855110	6.22	0.0040
Size*Alg	2	918326201	459163100	0.54	0.5838
CompOpt*Alg	6	770134103	128355684	0.15	0.9877
Size*CompOpt*Alg	6	5079982917	846663820	1.00	0.4340
DataStr	1	871925920	871925920	1.03	0.3144
Size*DataStr	1	167768175	167768175	0.20	0.6576
CompOpt*DataStr	3	1191135735	397045245	0.47	0.7041
Size*CompOpt*DataStr	3	1136874217	378958072	0.45	0.7190
Alg*DataStr	2	2635759532	1317879766	1.56	0.2201
Size*Alg*DataStr	2	1809587849	904793924	1.07	0.3502
CompOpt*Alg*DataStr	6	5622169555	937028259	1.11	0.3701
Size*CompOpt*Alg*DataS	6	6400215040	1066702507	1.26	0.2914

We perform our analysis at $\alpha - level = 0.05$. Here, results show that the algorithm is the only factor affecting the variable `memory/free` which accounts for the pages of RAM in KBytes that are accessible when a process needs memory. Using Duncan's test on the algorithm factor as *post hoc* test we get the following results:

t Tests (LSD) for `memory_free`

Alpha	0.05
Error Degrees of Freedom	48
Error Mean Square	8.4351E8
Critical Value of t	2.01063
Least Significant Difference	14599

Means with the same letter are not significantly different.

t Grouping	Mean	N	Alg
A	1298664	32	1
A			
A	1292889	32	2
B	1274171	32	3

Using contrasts to analyze the algorithm factor we get:

Dependent Variable: memory_free

Contrast	DF	Contrast SS	Mean Square	F Value	Pr > F
Comp. Alg 1 & 2	1	533689623	533689623	0.63	0.4303
Comp. Alg 2 & 3	1	5605473681	5605473681	6.65	0.0131
Comp. Alg 1 & 2 with 3	1	9958020598	9958020598	11.81	0.0012

This states that algorithm one and two are not significantly different from each other at the 0.05 level for variable memory/free. Algorithm three is significantly worse than algorithm 1 or 2.

The previously mentioned procedure is used when a factorial design is analyzed. However, sometimes a full factorial design cannot be done for feasibility constraints. This was our case when doing the experiments with the application code. Randomizing problem size from experimental run to experimental run would have caused excessive amount of time. We used a split-split plot design of experiment [25]. The linear model for this experiment differs from the previous one in that an additional factor, *block*, needs to be added to the model and considered into the error terms in the model. The following SAS code shows the error terms added to the model where the variable *block* was incorporated to the model.

```
proc anova;
class block Size Alg CompOpt;
model memory_free = block | Size | Alg | CompOpt ;
test h=Size e=block*Size;
test h=Alg Size*Alg e=block*Size*Alg;
test h=CompOpt Alg*CompOpt Size*CompOpt Size*Alg*CompOpt
e=block*Size*Alg*CompOpt;
run;
quit;
```

In the first experiment we compared the effects of thirteen compiler options, three problem sizes, and two different multiplication algorithms on the result which was assessed by a set of metrics we selected. We have decided to run three replicates of the experiment. This yields 234 experimental runs for one experiment. The number of iterations for obtaining the solution of the iterative solver has been fixed to remove the impact of reduced matrix conditioning. In each one of our three blocks we select at random the problem size, then for each of these, at random we select the matrix multiplication algorithm used to solve the problem. Then in each one of these subplots we randomly select the compiler options used to produce the executable code. This means we have more precision in looking at the effect

of compiler options and least precision for problem size effect.

7.4 Summary

We have presented statistical methods as a powerful tool in the analysis of performance data. Depending on whether the data comes from traces or from summaries, we can classify them as an output from a random process or a random variable. A performance data matrix format was specified for applying statistical methods to the data. Preprocessing techniques typically used in pattern recognition were tested on our data set to verify which one was more appropriate. Dimension normalization turned out to be the most effective preprocessing technique.

Some of the techniques used were: correlation analysis, feature subset selection, and ANOVA. Correlation analysis establishes the linear relationship among variables. Feature subset selection was used to determine how many and which important performance metrics should be looked at using an entropy cost function. ANOVA established which controlled factors were causing variations on the performance metrics selected by the feature selection method. *Post hoc* analysis and analysis of means can provide additional information on the results after the null hypothesis is rejected.

CHAPTER 8

Results

In this chapter we present the results obtained in a case study to test the proposed methodology. We show results from four different experiments. Experiment one and two are used to characterize the observable computing system (OCS). Experiments three and four are used to validate the methodology.

Reviewing the proposed methodology and its details, we now present a summary of the steps:

1. Preliminary problem analysis

The case study code used in this research is called Prism and it implements a finite element boundary-integral (FE-BI) numerical method for the analysis of conformal antennas. According to the input parameters, the iterative solver method is selected by the code and preconditioning is either enabled or disabled. This application runs on a Sun Enterprise 450 and profiling pointed to a dense matrix-vector multiplication subroutine as the most time consuming routine. Prism was parallelized using OpenMP directives. The design factors selected for experimentation were: compiler options, problem size, and algorithm.

2. Design of experiment

Two different experiment designs were used. The first type was a split-split plot design and the second type was a fully-randomized full-factorial design. The experiments were:

- Experiment 1: Parallel implementation of Prism
- Experiment 2: Serial implementation of Prism
- Experiment 3: Inefficient memory access pattern in Prism, validation experiment.
- Experiment 4: Matrix-vector multiplication kernel, validation experiment.

3. Data Collection

We used both software and operating system instrumentation. Software instrumentation was done using the KAP/Pro statistical library. Operating system data collection was done using the unix commands `sar`, `iostat`, and `vmstat`.

4. Data Analysis

Perl scripts extracted the data and converted it to a format used by two widely used statistical packages: Minitab 13 and SAS v8. The extracted data was normalized using dimension normalization with Euclidean norm. Correlation analysis was used to determine the most correlated metrics with execution time. We estimated the intrinsic dimension of the data using three commonly used estimators: scree test, KG, and cumulative percentage of variance. Sequential forward search with entropy cost function was used to determine the most important metrics. Once the important metrics were identified, ANOVA and *post hoc* comparisons were used to establish which factors affected important metrics and reach conclusions.

Results of these experiments are presented in the following sections.

8.1 Experiment 1: Parallel Implementation of Prism

The first experiment was used to characterize the interactions between our application and the system. Prism was parallelized using OpenMP constructs. As described in section 5.4.1, two different algorithms for matrix-vector multiplication were used, thirteen compiler options and three different problem sizes were tested. The experiment design was done

using a split-split plot design and the actual order of execution of each experimental run is shown in appendix D. We ran 234 experimental runs and 47 different values were measured.

8.1.1 Correlation Analysis.

Those metrics most correlated with execution time, with correlation higher than 0.9, are shown in table 8.1, where the correlation was negative in all cases. Negative correlation is interpreted as follows: execution time increases when the metric value decreases.

Table 8.1. Metrics with largest correlation with execution time in experiment 1.

Order	Label	Description	Category	Correlation
1	lwrit/s	Accesses of system buffer cache to write	Buffer Activity	-0.965
2	lread/s	Accesses of system buffer cache to read	Buffer Activity	-0.965
3	c0t0d0/wps	Write per second per disk	I/O	-0.960
4	c0t0d0/util	Percentage of disk utilization per disk	I/O	-0.958
5	disk/s0	Disk operations per second	Disk	-0.948
6	page/mf	Minor faults in units per second	Paging activity	-0.910
7	vflt/s	Address translation page faults per second	Paging activity	-0.908

All correlations shown in table 8.1 are high and significant. The two most correlated metrics are access to buffer cache to read and to write. These report logical I/O requests and occur if a program opens device for I/O. Then the next two metrics are also related to I/O for a specific disk. These two measurements are writes to disk per second and percentage of disk utilization. Similarly, metric 5 is disk operations per second. As we can see from this pattern, it shows that this application's bottleneck is I/O or disk access. The last two metrics are related to paging activity. From inspecting the code, we understand that this is a typical behavior since a dense matrix-vector multiplication algorithm dominates the computation and the matrices involved in this application are extremely large.

8.1.2 ANOVA

Three-way ANOVA at significance level $\alpha = 0.05$ was performed. ANOVA analyzes the effect of qualitative factors on one dependent variable. Table 8.2 shows ANOVA results for those metrics obtained in table 8.1. This table shows whether the factors had an effect or not on the variable.

Table 8.2. Effect of factors and interactions on the most correlated metrics with execution time for experiment 1.

Item	Name	Problem Size (S)	Algorithm (A)	Compiler Option (C)	Interactions			
					$S * A$	$S * C$	$A * C$	$S * A * C$
0	Execution time	Yes	Yes	Yes	No	No	Yes	No
1	lwrit/s	No	Yes	Yes	No	No	No	No
2	lread/s	No	Yes	Yes	No	No	No	No
3	c0t0d0/wps	No	Yes	Yes	No	Yes	No	No
4	c0t0d0/util	No	Yes	Yes	No	No	No	No
5	disk/s0	Yes	Yes	Yes	No	Yes	No	No
6	page/mf	Yes	Yes	Yes	No	Yes	No	Yes
7	vfft/s	Yes	Yes	Yes	No	Yes	No	Yes

Recall that all the shown metrics are correlated with execution time, therefore we also include an ANOVA analysis of execution time. It is interesting to notice from the analysis that the choice of algorithm and compiler options significantly affect all metrics correlated with execution time.

Following this analysis we proceed to obtain the number of metrics required to describe the behavior of the system. We normalized the data using the Euclidean norm and then estimated the intrinsic dimension of the data.

8.1.3 Dimensionality

In section 7.3.4, we described three different methods to estimate the intrinsic dimension of the data set: cumulative percentage, Kaiser-Guttman (KG), and scree test. Figure

8.1 shows an example of a plot of the eigenvalues of the resulting correlation matrix for this experiment. This graph illustrates the *scree test* and the *Kaiser-Guttman criteria* (eigenvalues greater than one).

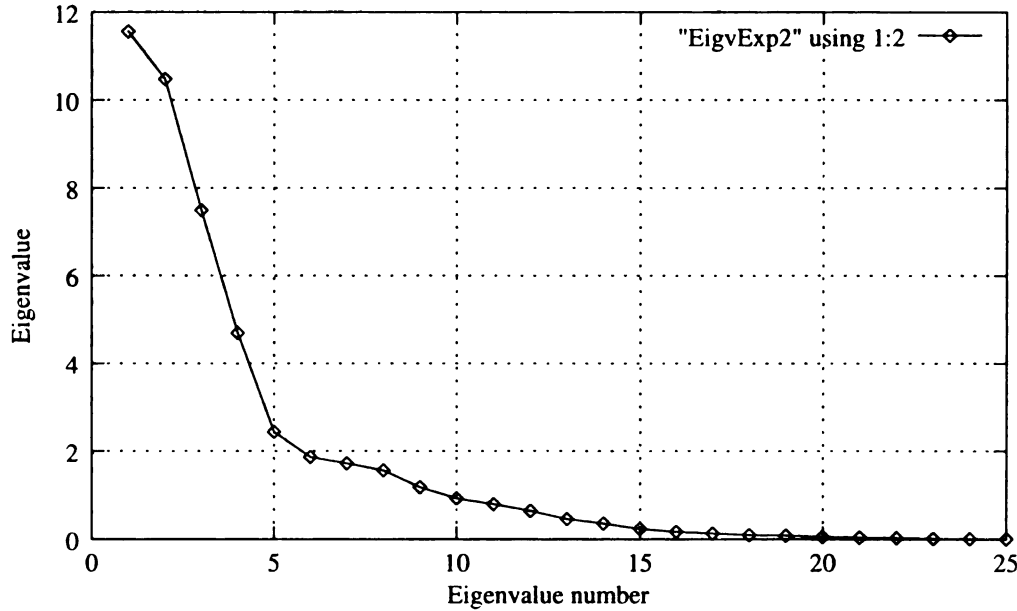


Figure 8.1. Eigenvalues of correlation matrix in experiment 1.

Notice the change in the slope of the curve at five eigenvalues and at eight eigenvalues. Scree test might have two or three inflection points in the curve and this is one of the cases. Notice that nine eigenvalues are greater than one. This is the KG criteria. Table 8.3 show how many metrics should be kept to preserve the variability of the data, according to the three estimation methods. For this data set, nine metrics can explain the variability of the data.

To validate these tests for intrinsic dimensionality estimation we have created a synthetic data set by the use of a random number generator. Nine columns were generated at random. Then subsequent columns are multiples of the first nine columns plus noise added to them. The columns of this matrix are then manipulated to have the same mean and variance as the matrix from the data set obtained in this experiment. We use this matrix as input to

Table 8.3. Number of metrics to keep variability of the current data according to three different criteria for experiment 1.

Test	Estimated intrinsic dimension
Scree test	8
Cumulative percentage (95%)	9
K-G	9
Maximum of the three methods	9

the three estimators of dimensionality and we get that all three methods found that nine is the dimension of the data set. Appendix I shows the code used for this test.

Figure 8.2 shows the scree test and the KG criteria for the synthetic data set.

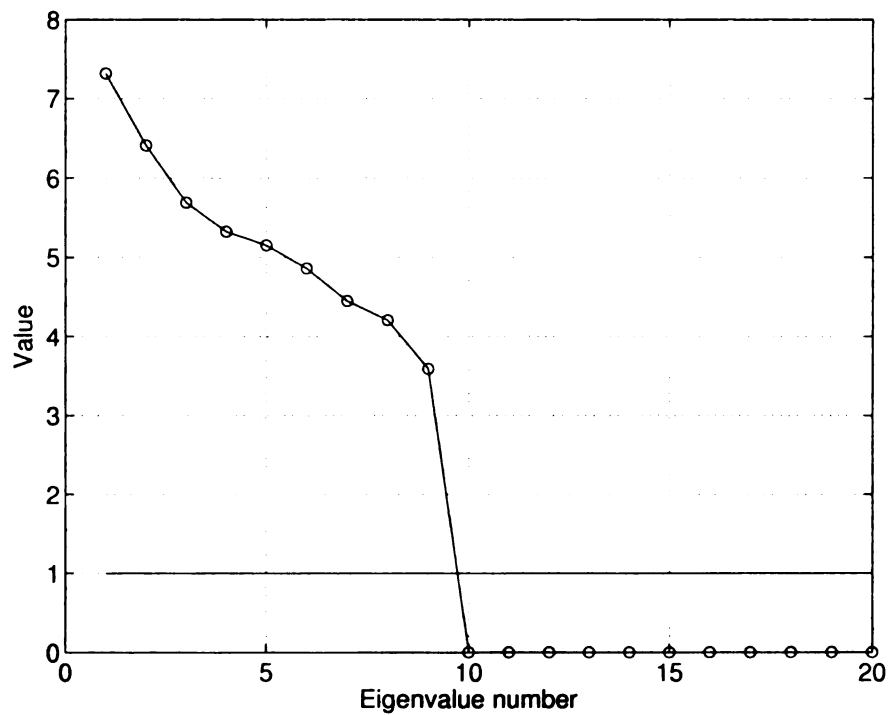


Figure 8.2. Eigenvalues of correlation matrix for synthetic data.

8.1.4 Metric Selection

Sequential forward search (SFS) was applied to determine which subset of metrics will preserve most of the data variability. Table 8.4 show those metrics with the highest information content for this experiment, as selected by the SFS algorithm. The cost function used for performing the search was entropy as described in [81].

Table 8.4. Metrics with highest information content in experiment 1.

Item	Name	Description	Category
1	memory/free	Usage of virtual and real memory. Free size of the free list (Kbytes).	Virtual Memory Statistics
2	cpu/sy	Percentage of time in system mode.	CPU utilization
3	bwrit/s	Writes per second of data from system buffers to disk.	Buffer activity
4	%wcache	Cache hits ratios for write as percentage.	Buffer activity
5	cpu/us	Percentage of time in user mode.	CPU utilization.
6	page/fr	Paging activity in units per second. Kilobytes freed.	Paging
7	vflt/s	Address translation page faults per second.	Paging activity
8	atch/s	Page faults per second that are satisfied by reclaiming a page currently in memory.	Paging activity
9	%wio	Portion of time running idle with some process waiting for block I/O.	CPU utilization

We should notice that this method has selected a variety of metrics rather than only I/O or memory related ones. This time we have measurements from virtual memory statistics, CPU utilization, buffer activity, and paging.

8.1.5 ANOVA

Once the metrics have been selected, we proceed to analyze which of the factors studied significantly affect these metrics. The three factors studied in this case were problem size,

algorithm, and compiler options. Table 8.5 show analysis of variance results for those metrics.

Table 8.5. ANOVA on the metrics shown in table 8.4. Main effects.

Factor	Metrics affected by the factors
Size (S)	cpu/sy, vflt/s
Algorithm (A)	cpu/sy, bwrit/s, vflt/s
Compiler Option (C)	memory/free, cpu/sy, bwrit/s, %wcache, cpu/us, vflt/s

We notice here that even though that kilobytes freed in paging activity, attaches per second, and portion of time idle waiting for I/O were selected as important metrics, none of the factors studied affect them. There might be other factors affecting these metrics. On the other hand percentage of the time in system mode and address translation per faults were affected by all three factors.

8.1.6 Another method for subset selection

The independence of metrics can be used as the cost function to explain the variability of the data since it is related to the amount of information contained in the performance data matrix. In the subset selection method suggested by Vélez and Jiménez [90], the criterion of independence between columns is used as a measure for subset selection. Those features that are most independent and explain the highest correlation are selected based on principal component analysis (PCA) and singular value decomposition (SVD). Principal component analysis is a method used to project actual variables into new uncorrelated variables. The SVD of matrix A is $A = U \sum V^T$ where U and V are orthogonal and $\sum = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_r)$ with $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r \geq 0$. The σ_i s are called the singular values. The algorithm proposed in this work has been used in the past for unsupervised feature subset selection applied to hyperspectral imagery.

Table 8.6 show those metrics with the highest variability for this experiment, as selected

by the SVD algorithm.

Table 8.6. Metrics with highest information content selected by SVD for experiment 1.

Item	Name	Description	Category
1	memory/free	Usage of virtual and real memory. Free size of the free list (Kbytes).	Virtual Memory Statistics
2	pflt/s	Page faults from protection errors per second (illegal access to page).	Paging Activity
3	page/re	Paging activity in units per second. Page reclaims.	Paging
4	c0t1d0/wps	Writes per second per disk.	I/O
5	%wio	Portion of time running idle with some process waiting for block I/O.	CPU Utilization
6	page/sr	Paging activity in units per second. Pages scanned by clock algorithm.	Paging
7	page/pi	Paging activity in units per second. Kilobytes paged in.	Paging
8	page/po	Paging activity in units per second. Kilobytes paged out.	Paging
9	faults/cs	Trap/Interrupt rates per second. CPU context switches.	Memory Faults

The selected metrics describe activity which experts usually look for when tuning a program: paging activity, cpu utilization, memory faults, and virtual memory statistics. Table 8.7 show analysis of variance results for those metrics. Only two of the metrics selected by SVD were selected by SFS.

Table 8.7. ANOVA on the metrics shown in table 8.6. Main effects.

Factor	Metrics affected by the factors
Size (S)	faults/cs
Algorithm (A)	faults/cs
Compiler Option (C)	memory/free, page/po, faults/cs

It stands out that CPU context switches is affected by all three studied factors. Algorithm and problem size do not affect any of the other metrics.

8.2 Experiment 2: Serial Implementation of Prism

Here the application was using similar algorithms as in the parallel experiment but without OpenMP calls. All other factors remain the same. The two algorithms used in this experiment are shown in Appendix C. These are identified as Algorithms D and E. The actual order of execution of each experimental run is shown in appendix E.

8.2.1 Correlation Analysis.

Once again, those metrics most correlated with execution time were computed. Table 8.8 shows those metrics with correlation higher than 0.9 with execution time.

Table 8.8. Metrics with largest correlation with execution time for experiment 2.

Rank	Label	Description	Category	Correlation
1	c0t0d0/wps	Writes per second per disk	I/O	-0.985
2	disk/s0	Disk operations per second	Disk	-0.985
3	lwrit/s	Accesses of system buffer cache to write	Buffer Activity	-0.981
4	c0t0d0/util	Percentage of disk utilization per disk	I/O	-0.981
5	lread/s	Accesses of system buffer cache to read	Buffer Activity	-0.980

Notice that all these were also highly correlated with execution time in Experiment 1. In this case, regardless of whether the application is running serially or in parallel, the metrics which have a linear relation with execution time are the same.

8.2.2 ANOVA

For those metrics most correlated with execution time, ANOVA was used to determine which ones of the factors affect the obtained metrics. Table 8.9 show ANOVA for these metrics.

Table 8.9. Effect of factors and interactions on the most correlated metrics with execution time in experiment 2.

Item	Name	Problem Size (S)	Algorithm (A)	Compiler Option (C)	Interactions			
					$S * A$	$S * C$	$A * C$	$S * A * C$
0	execution time	Yes	Yes	Yes	No	Yes	Yes	No
1	c0t0d0/wps	No	No	No	No	No	No	No
2	disk/s0	No	No	No	No	No	No	No
3	lwrit/s	No	Yes	Yes	No	No	Yes	No
4	c0t0d0/util	No	Yes	Yes	No	No	Yes	No
5	lread/s	No	Yes	Yes	No	No	Yes	No

We can notice that the problem size does not affect any of the metrics correlated with execution time. Also disk writes per second (c0t0d0/wps) and disk operations per seconds (disk/s0) are not affected by any of the studied factors. As in experiment 1, execution time is affected by all three studied factors but in this case there interaction between problem size and compiler options. This means that compilers options do not cause the same behavior as problem size varies.

8.2.3 Dimensionality

The three methods explained previously were used to estimate the intrinsic dimensionality of the data. Table 8.10 shows the number of metrics to keep to preserve the variability of the data. The maximum values of the three methods was used as the dimension of the data.

Table 8.10. Number of metrics to keep variability of data according to three different criteria in experiment 2.

Test	Experiment with serial implementation
Scree test	7
Cumulative percentage (95%)	6
K-G	8
Maximum of the three methods	8

8.2.4 Metric Selection

Using SFS and the results from Table 8.10, those metrics shown in Table 8.11 were obtained as the most relevant ones. The cost function used in this search is entropy.

Table 8.11. Metrics with highest information content in experiment 2.

Item	Name	Description	Category
1	memory/free	Usage of virtual and real memory. Free size of the free list (Kbytes).	Virtual Memory Statistics
2	pswch/s	Process switches	System swapping activity
3	%wcache	Cache hit ratios for write as percentage	Buffer activity
4	cpu/sy	% of time the system spent in system mode	CPU utilization
5	bwrit/s	Writes per second of data from system buffers to disk	Buffer activity
6	faults/cs	CPU context switches. Interrupts per second.	Memory faults
7	faults/in	Non-clock device interrupts. Interrupts per second.	Memory faults
8	pgout/s	Page-out requests per second.	Paging activity

We can compare this table with table 8.4 and identify that half of the metrics were also selected for the parallel code but process switches, memory faults, and page-out requests are additional metrics in this case.

Table 8.12 shows ANOVA results for these metrics. From these results we observe that

process switches, percentage of time in system mode, and CPU context switches are not affected by any of the studied factors.

Table 8.12. ANOVA on the metrics shown in table 8.11. Main effects.

Factor	Metrics affected by the factors
Size (S)	%wcache, bwrit/s
Algorithm (A)	faults/in, pgout/s
Compiler Option (C)	memory/free, bwrit/s, faults/in

Using the method presented in [90], those metrics shown in Table 8.13 were obtained as the most relevant ones. These metrics describe buffer and paging activity, virtual memory statistics, and CPU utilization. This time, execution time was selected as a relevant metric.

Table 8.13. Most important metrics for experiment 2 according to SVD.

Item	Name	Description	Category
1	atch/s	Page faults per second that are satisfied by reclaiming a page currently in memory (attaches per second).	Paging Activity
2	pfIt/s	Page faults from protection errors per second (illegal access to page).	Paging Activity
3	bread/s	Reads per second of data to system buffers from disk.	Buffer Activity
4	memory/free	Usage of virtual and real memory. Free size of the free list (Kbytes).	Virtual Memory Statistics
5	page/po	Paging activity. Kilobytes paged out per second.	Paging
6	pgin/s	Page-in requests per second.	Paging Activity
7	cpu/wt	Report the percentage of time the system has spent waiting for I/O.	CPU Utilization
8	execution time	Total execution time.	Overall

Table 8.14 shows ANOVA results for these metrics. Notice that only execution time is affected by all three factors.

Table 8.14. ANOVA on the metrics shown in table 8.13. Main effects.

Factor	Metrics affected by the factors
Size (S)	execution time
Algorithm (A)	cpu/wt, execution time
Compiler Option (C)	atch/s, memory/free, cpu/wt, execution time.

8.3 Experiment 3: Inefficient Memory Access Pattern Algorithm

In this experiment we are testing algorithms A, B, and C as described in appendix C. Algorithm C is purposely having an inefficient matrix-vector multiplication algorithm. It accesses rows and columns in reverse order, resulting in a reduction in data locality. This algorithm is used to validate results by exposing metrics related to memory access. All other factors have the same levels as in the previous experiments. The actual order in which experimental runs were executed for this experiment are shown in appendix F.

8.3.1 Correlation Analysis

Those metrics most correlated with execution time are shown in table . Table 8.15 shows those metrics with correlation higher than 0.9 with execution time.

These are the same metrics found in experiment one. We are perceiving a pattern in those metrics most correlated with execution time since they are approximately the same metrics. However, we cannot generalize since all three examples come from the same code and application. We should study a different application to make a fair comparison.

Table 8.15. Metrics with largest correlation with execution time for experiment 3.

Rank	Label	Description	Category	Correlation
1	lwrit/s	Accesses to system buffers to write.	Buffer activity	-0.9656
2	lread/s	Accesses of system buffers to read.	Buffer activity	-0.9612
3	page/mf	Minor faults per second.	Paging	-0.9235
4	vflt/s	Address translation page faults per second.	Paging activity	-0.9214

8.3.2 ANOVA

ANOVA at α level equal to 0.05 was computed. Table 8.16 shows ANOVA results for those metrics obtained in Table 8.15.

Table 8.16. Effect of factors and interactions on the most correlated metrics with execution time for experiment 3.

Item	Name	Problem Size (S)	Algorithm (A)	Compiler Option (C)	Interactions			
					$S * A$	$S * C$	$A * C$	$S * A * C$
0	execution time	Yes	Yes	Yes	No	No	Yes	No
1	lwrit/s	No	Yes	Yes	No	No	Yes	No
2	lread/s	No	Yes	Yes	No	No	Yes	No
3	page/mf	Yes	Yes	Yes	No	Yes	Yes	Yes
4	vflt/s	Yes	Yes	Yes	No	Yes	Yes	Yes

Like in experiment one, the selection of algorithm and compiler options affect those metrics most correlated with execution time. All three factors affect execution time and paging activity. Notice that buffer activity is not affected by problem size.

8.3.3 Dimensionality

Table 8.17 shows the dimension estimated by all three methods. Nine metrics were estimated as necessary for this data set.

Table 8.17. Estimate of the intrinsic dimension of this data set.

Test	Experiment with serial implementation
Scree test	7
Cumulative percentage (95%)	9
K-G	9
Maximum of the three methods	9

8.3.4 Metric Selection

The metrics selected by SFS with entropy cost function are presented in Table 8.18.

Table 8.18. Metrics with highest information content in experiment 3.

Item	Name	Description	Category
1	memory/free	Usage of virtual and real memory. Free size of the free list (Kbytes).	Virtual Memory Statistics
2	bwrit/s	Writes per second of data from system buffers to disk	Buffer activity
3	%wcache	Cache hit ratios for write as percentage	Buffer activity
4	lwrit/s	Accesses of system buffer cache to write	Buffer activity
5	cpu/sy	Percentage of time in system mode	CPU utilization
6	cpu/id	Percentage of time the system has spent idling	CPU utilization
7	page/po	Kilobytes paged out per second	Paging
8	pflt/s	Page faults from protection errors per second (illegal access to page).	Paging activity
9	fd0/wps	Writes per second per disk	I/O

There are a variety of metrics including virtual memory statistics, buffer activity, cpu utilization, paging, and I/O related.

Table 8.19 shows ANOVA results for these metrics. This time, the algorithm selected has an effect on almost all metrics. This is as expected since we designed this experiment for contrasting three matrix-vector multiplication with different behavior, one of which has a bad memory access pattern.

Table 8.19. ANOVA on the metrics shown in table 8.18. Main effects.

Factor	Metrics affected by the factors
Size (S)	cpu/sy
Algorithm (A)	memory/free, bwrit/s, %wcache, lwrit/s, cpu/sy, cpu/id, pflt/s, fd0/wps
Compiler Option (C)	memory/free, bwrit/s, %wcache, lwrit/s, cpu/sy, cpu/id, page/po

Using the method presented in [90], those metrics shown in Table 8.20 were obtained as the most relevant ones. Likewise, the algorithm factor affects a large number of metrics as expected from the validation experiments. We compare this with Table 8.7, where only one metric is affected by the algorithm.

Table 8.20. Most important metrics for experiment 3 according to SVD.

Item	Name	Description	Category
1	pflt/s	Page faults from protection errors per second	Paging activity
2	disk/s2	Disk operations per second	Disk
3	cpu/wt	Percentage of the time the system has spent idling	CPU utilization
4	page/sr	Pages scanned by clock algorithm	Paging
5	c0t0d0/rps	Reads per second per disk	I/O
6	bread/s	Reads per second of data to system buffers from disk	Buffer activity
7	page/po	Kilobytes paged out per second	Paging
8	faults/cs	CPU context switches.	Memory faults
9	fd0/wps	Write per second per disk	I/O

Table 8.21 shows ANOVA results for these metrics.

Table 8.21. ANOVA on the metrics shown in table 8.20. Main effects.

Factor	Metrics affected by the factors
Size (S)	faults/cs
Algorithm (A)	pflt/s, disk/s2, c0t0d0/rps, faults/cs, fd0/wps
Compiler Option (C)	c0t0d0/rps, page/po, faults/cs

We also realize that, for this experiment, compiler options have a effect on many different metrics, in contrast to the previous experiment. This is indicative of interaction between algorithm and compiler option. Examining Appendix F, we find three metrics having interaction with algorithm: lwrit/s, cpu/sy, and cpu/id.

8.4 Experiment 4: Matrix-Vector Multiplication Tests

In this experiment we are testing only matrix-vector algorithms. This time the design of experiment is fully-randomized full-factorial. Four factors are used in the experiment: problem size, algorithm, data structure, and compiler options. The actual order in which experimental runs were executed for this experiment are shown in appendix G.

8.4.1 Correlation Analysis

Table 8.22 shows those metrics with correlation with execution time higher than 0.6. Notice that correlations are much lower than in previous experiments. Moreover, the number of metrics correlated with execution time is drastically reduced. One possibility of this behavior is that the algorithm is only exercising memory usage of the system while the complete application is using different aspects of the system.

Table 8.22. Metrics with largest correlation with execution time.

Rank	Label	Description	Category	Correlation
1	page/mf	Minor faults per second	Paging	-0.6746
2	page/re	Page reclaims per second	Paging	-0.6343

8.4.2 ANOVA

ANOVA at α level equal to 0.05 was obtained. Table 8.23 shows ANOVA results for those metrics obtained in Table 8.15. We also include execution time in the analysis.

Table 8.23. Effect of factors and interactions on the most correlated metrics with execution time for experiment 3.

Item	Name	Problem Size (S)	Algorithm (A)	Compiler Option (C)	Data
					Structure (D)
0	execution time	Yes	Yes	Yes	Yes
1	page/mf	Yes	Yes	Yes	Yes
2	page/re	No	Yes	Yes	Yes

8.4.3 Dimensionality

Table 8.24 shows the dimension estimated by all three methods. Seven metrics were estimated as necessary for this data set.

Table 8.24. Estimate of the intrinsic dimension in experiment 4.

Test	Experiment with serial implementation
Scree test	6
Cumulative percentage (95%)	7
K-G	6
Maximum of the three methods	7

8.4.4 Metric Selection

The metrics selected by SFS with entropy cost function are presented in Table 8.25. As in experiments one through three, we still observe a variation in the types of metrics selected. We obtain CPU utilization, virtual memory statistics, buffer activity, and paging activity related metrics. In contrast to previous experiments, here one type of metric does not dominate the results. In experiment 1, paging activity related metrics would be represented more than others. In experiment 2, metrics associated with memory faults were more visible. In experiment 3, buffer activity related metrics were relevant.

Table 8.25. Metrics with highest information content for experiment 4 .

Item	Name	Description	Category
1	memory/free	Usage of virtual and real memory. Free size of the free list (Kbytes).	Virtual Memory Statistics
2	%sys	Portion of time running in system mode	CPU utilization
3	memory/swap	Amount of swap space currently available	Virtual memory statistics
4	bwrit/s	Writes per second of data from system buffers to disk	Buffer activity
5	page/re	Page reclaims per second	Paging
6	cpu/sy	Percentage of the time the system has spent in system mode.	CPU utilization
7	pgout/s	Page-out requests per second	Paging activity

Table 8.26 shows ANOVA results for these metrics. ANOVA this time show that the selected factors affect the metrics, as expected. We have to remember that we selected a variety of levels causing significant effects in the results on purpose to validate the methodology.

Using the method presented in [90], those metrics shown in Table 8.27 were obtained as the most relevant ones. Here execution time was selected as important. Similar to the previous method, there is a large variety of selected metrics.

Table 8.26. ANOVA on the metrics shown in table 8.18. Main effects.

Factor	Metrics affected by the factors
Size (S)	%sys, bwrit/s, cpu/sy, pgout/s
Algorithm (A)	memory/free, %sys, page/re,cpu/sy, pgout/s
Compiler Option (C)	bwrit/s, page/re, pgout/s
Data Structure (D)	%sys, bwrit/s, page/re, cpu/sy, pgout/s

Table 8.27. Most important metrics for experiment 4 according to SVD.

Item	Name	Description	Category
1	ExecTime	Execution Time	Overall
2	pgin/s	Page-in requests per second	Paging activity
3	c1t1d0/util	Percentage of disk utilization	I/O
4	bwrit/s	Writes per second of data from system buffers to disk	Buffer activity
5	ppgout/s	Pages paged out per second	Paging activity
6	faults/cs	CPU context switches	Memory faults
7	page/po	Kilobytes paged out per second	Paging

Table 8.28 shows ANOVA results for these metrics and it shows that the studied factors have an effect on the resulting metrics.

Table 8.28. ANOVA on the metrics shown in table 8.27. Main effects.

Factor	Metrics affected by the factors
Size (S)	Execution time, bwrit/s, ppgout/s, faults/cs, page/po
Algorithm (A)	Execution time, ppgout/s, faults/cs, page/po
Compiler Option (C)	Execution time, bwrit/s, ppgout/s, faults/cs, page/po
Data Structure (D)	Execution time, bwrit/s, ppgout/s, faults/cs, page/po

8.5 Analysis of Results

Results show some interesting findings. First, three metrics are selected as important in the different experiments: memory/free, bwrit/s and cpu/sy. These indicate usage of virtual memory, writes to disk, and percentage of the time the system is in system mode. Their relevance in different variations of the experiment indicate that they should be observed when performing experiments. Moreover, bwrit/s also show up in the work by Ahn and Vetter in [15]. This is a surprising results since their application is using MPI on a distributed memory system while we use OpenMP on a shared memory system.

Second, the scree test does not provide a reliable estimate of the dimension of the data set. The graph might have two or three inflection points, making it hard to determine the estimated dimensionality of the data set.

Another analysis we did is the percentage of metrics kept by the dimensionality estimator.

Table 8.29. Percentage of metrics kept for the analysis.

Experiment	Orig. No. Metrics	Estimated Dimension	% of Metrics Retained
Exp 1	47	9	19.15%
Exp 2	43	8	18.60%
Exp 3	52	9	17.31%
Exp 4	36	7	19.44%

Table 8.29 shows that approximately 18% of the metrics are retained as important.

When comparing SVD and SFS with entropy cost function, SFS provided metrics more in accordance to what our experience would tell us were more important than the SVD method. This might be due to the use of an entropy cost function which represents the amount of information contained in the data.

Once that we know that a factor is affecting a metric, we can apply *post hoc* comparisons and analysis of means to study the causes. In experiment one, we noticed that execution

time is affected by the compiler options. We used analysis of means along with the least significant difference (LSD) *post hoc* comparison to study the classification of compiler options according to the test. We obtained the following result in SAS:

t Tests (LSD) for ExecTime

Means with the same letter are not significantly different.

t Grouping	Mean	N	Comp
			Opt
A	1162.00	2	1
A			
A	1160.00	2	3
B	663.00	2	13
B			
B	662.50	2	2
B			
B	623.50	2	4
B			
B	616.50	2	11
B			
B	615.50	2	6
B			
B	602.00	2	5
B			
B	601.50	2	8

B			
B	601.50	2	7
B			
B	600.00	2	10
B			
B	599.50	2	12
B			
B	598.00	2	9

When we carefully analyze these results we can observe that compiler options one and three correspond to combinations of flags not containing the -fast flag while the remaining options contain the -fast flag. Therefore, in this particular case, the -fast flag is the only one significant. Compiler option 9 will give the best execution time. We can conclude that some flags are more important than others and that the execution time depends on the compiler used.

8.6 Scientific Programmer Actions

Once the interpretation or evaluation of the observable values is available, this knowledge can be converted into suggestions on how to improve the system-software interactions. The methodology we have described is very general and applies to any observable computing system. An additional step is required for automatic performance tuning, however, this step is system dependent. We suggest one possible implementation for this action. It involves signal classification and knowledge from the analysis incorporated into a knowledge based system.

As explained earlier in section 7.3.4, feature subset selection is used for obtaining the relevant metrics describing the observable computing system. From this set of metrics application signatures can be extracted describing the trajectory of the metrics. An application signature is a piecewise linear fitting of the curve obtained from the trajectory of any given

metric in the value-time plane. In the work by Lu and Reed [91], the authors suggest using a performance contract in which the observed signature is compared to a model signal to adaptively control a grid application. The comparison is made through the use of a degree of similarity measure.

Since the degree of similarity of application signatures can point to signatures which indicate possible sources of problems, we suggest the use of the information obtained from the degree of similarity algorithm and the information from the statistical analysis of the proposed methodology to classify the type of problem in the system and prescribe a solution to the diagnosis. This information can be given to a knowledge based system with a set of rules to prescribe a solution at the high level. Performance evaluation systems which contain a knowledge-based system for prescribing a solution include Kappa-Pi [92], and KOJAK [93] so their model can be followed.

8.7 Summary

This chapter has presented results obtained for the four experiments performed. Correlations, dimension estimation, feature subset selection using sequential forward search with entropy cost function, subset selection using SVD, ANOVA, and *post hoc* comparisons are presented. Validation experiments showed that the method indeed point out the causes of variations introduced in the code.

CHAPTER 9

Conclusion

9.1 Research Summary

The efficient implementation of an application on an advanced architecture requires a tight integration between software and hardware. This task is particularly difficult to achieve due to the growing complexity of nowadays systems and the large number of factors that might affect performance. For instance, execution time is determined by factors such as programming style, programming paradigm, language, compiler, libraries, architecture, and algorithms [94]. These factors are typically selected by the application programmer without knowing the actual effect of these decisions until the implementation is complete. After this initial step, a tuning process is initiated where the implementation is improved until an acceptable level of performance is achieved.

A widely accepted tuning methodology is shown in Figure 9.1. It incorporates the application programmer knowledge and expertise into the loop, causing several problems which withhold the widespread use of performance evaluation tools.

The first problem is that scientific programmers are required to interact with instrumentation and analysis and evaluation tools. Most of the time, they are experts in their respective fields but not on performance evaluation. If the converse is true, then the performance analyst might not have enough insight into the application to understand the relations between performance data and code. Moreover, scientific programmers also need experience and in-depth knowledge on the particular computer system to tune their appli-

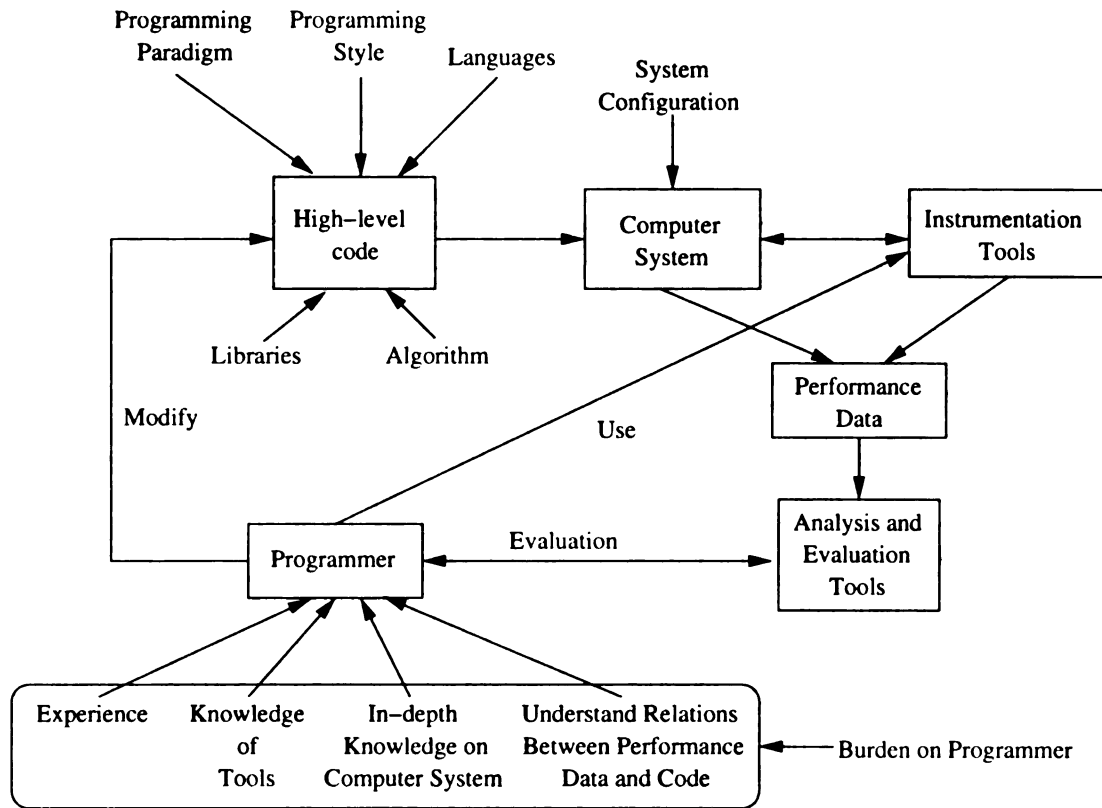


Figure 9.1. Typical analysis flow for tuning an application.

cation. The expertise level required for tuning an application to a particular platform limits the acceptance of performance tools among the scientific community.

An alternative tuning methodology is proposed to overcome some of these problems. The key point in this methodology is to obtain the appropriate information. A diagram with the approach of the proposed alternative is shown in Figure 9.2.

The main goal of this research was to obtain relevant information to improve the process of tuning applications on advanced architectures.

9.2 Contributions

The contributions of this work can be summarized as follows.

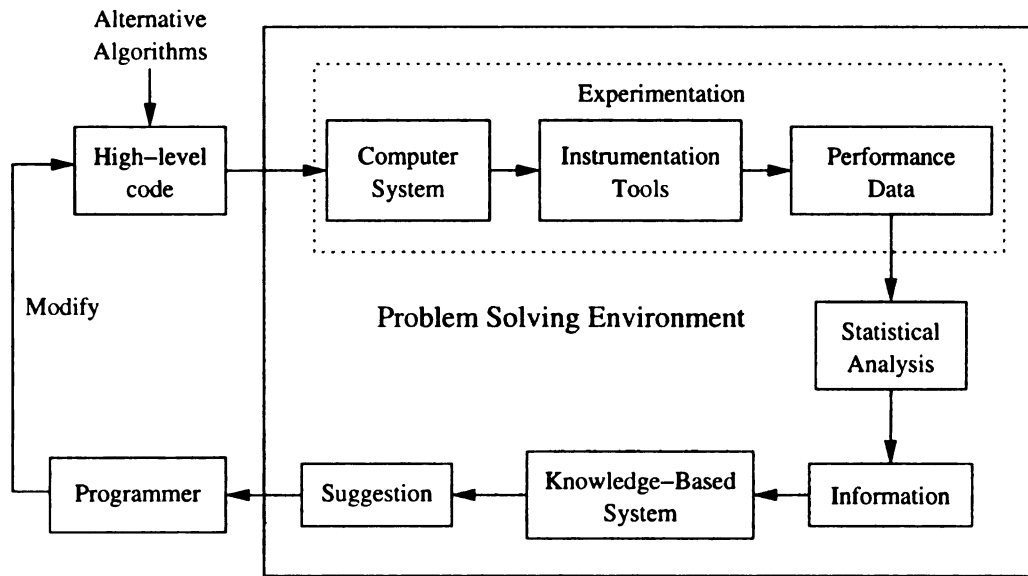


Figure 9.2. Proposed approach for application tuning. The dashed line shows the part of this tuning methodology addressed by this research.

9.2.1 A Methodology for Obtaining Relevant Performance Information

A methodology for determining the relation between high-level factors and performance data was developed [95, 96]. This methodology is novel in two main aspects:

- The integration of statistical methods is used to establish relations in the mapping process
- It removes the burden from the scientific programmer to interpret performance data.

First, this methodology is the only one which combines several statistical procedures to relate factors to response variables on the performance data analysis problem. No other method has approached the problem from this perspective in which associations are obtained through statistics.

In previous literature, approaches to relate high-level factors to performance information either suggest tool developers the type of information to be collected from the system at each level in the mapping process to establish relations [10], or it tries to estimate the information lost in the mapping process by incorporating knowledge on the behavior of

software/hardware, [8]. These methods are not appealing since they still have a portability problems and depend on the user expertise.

Second, the scientific programmer is not required to interpret performance information from tools. The methodology obtains unbiased information and relates it to high-level factors. This methodology is composed of four steps: *problem analysis*, *design of experiments*, *data collection*, and *data analysis* as illustrated in Figure 9.3.

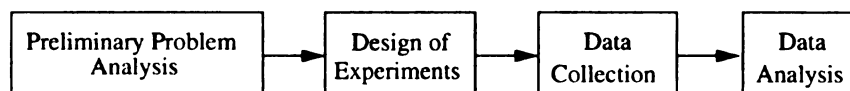


Figure 9.3. Proposed methodology to extract information in an *observable computing system (OCS)*.

Integration is the key to obtain the information in an unbiased manner. A computational electromagnetics case study was used to illustrate the usefulness of this methodology. As an example, one of the experiments demonstrated that for this particular application, memory usage related metrics were important and automatically selected by the method. Moreover, when execution time was examined, the analysis of compiler options showed that only the `-fast` flag was having a significant effect on execution time. The proposed methodology may be incorporated to future automatic performance evaluation tools.

9.2.2 The Use of Design of Experiments for Performance Analysis Experimentation

We have identified a systematic way of performing the design of experiments (DOE) for performance analysis. Design of experiments refers to the planning of experiments to extract the most amount of information with the minimum effort. It concerns with the way in which the treatments will be administered to the subjects in a study. A correct design should minimize the effects of uncontrollable factors and determines whether variations in the response are significant or due to the random nature of the process.

Several designs are available for the experimenter, from which we have selected two appropriate for the performance evaluation problem:

- full-factorial design
- split-split plot design

When DOE is used in the experimentation step of the methodology, then Analysis of Variance (ANOVA) can be used for data analysis. The combination of ANOVA with DOE allows reaching conclusions about the effect of factors at the high-level on the performance metrics obtained by instrumentation tools. These conclusions are unbiased, based on probability, and removed from subjective judgements. The use of DOE also minimizes the effects of factors not considered during experimentation.

Previous work in applying DOE and ANOVA to the performance analysis problem were limited in the type of performance data they were applied to: either execution time or CPI (cycles per instruction) [3, 11, 12, 19, 20], and from those, only the work by Alabdulkareem *et al.* analyzed large parallel codes.

This research has shown that the use of screening experiments along with the proper design of experiment limits the number of factors in the experiment, and making more feasible the entire experimentation.

9.2.3 The Usage of Data Reduction and Statistical Analysis

The last step in the proposed methodology is *data analysis*. Performance data collected during experimentation will not yield useful information until it is carefully analyzed. Our contribution to data analysis is the combination of dimensionality estimation, feature subset selection, and ANOVA to obtain information relevant to performance analysis when mapping algorithms to advanced architectures. The use of these techniques assists in locating, in an unbiased manner, sources of performance improvement.

In data analysis each metric is considered a feature. Statistical methods are the basis for data analysis. Specifically, four statistical techniques were used in the selected case study:

- correlation analysis
- normalization
- dimension estimation
- feature subset selection

This is illustrated in Figure 9.4.

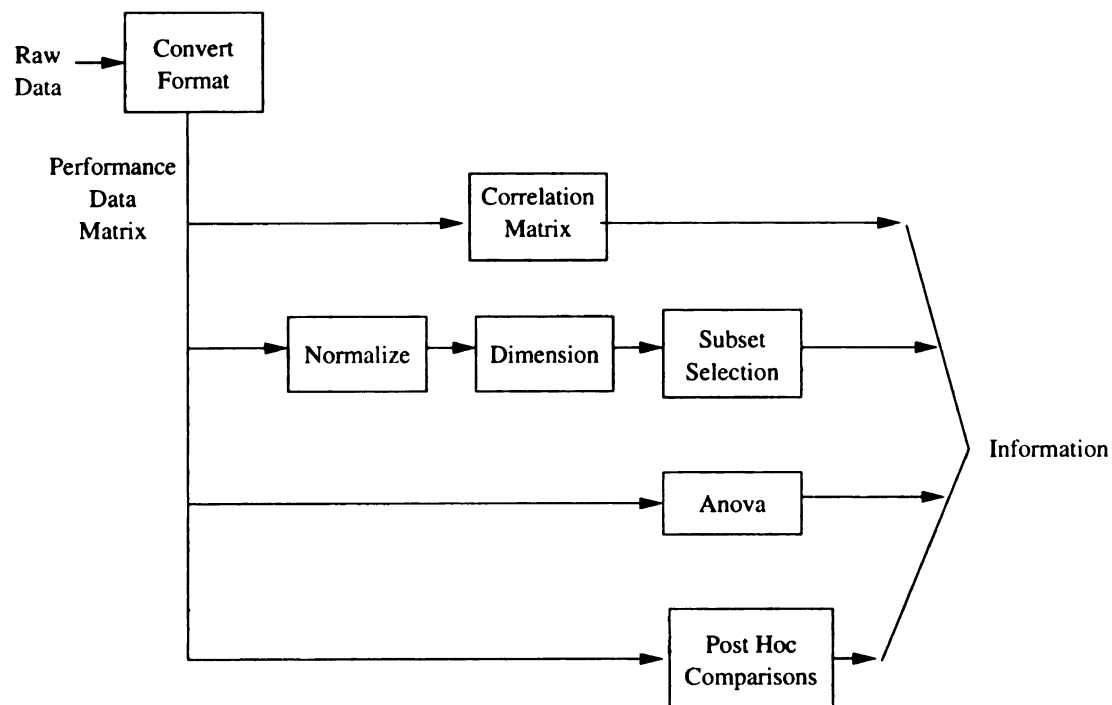


Figure 9.4. Summary of statistical analysis techniques used for extracting information about performance outcomes.

Correlation

The degree of association between two variables, if any exists, is obtained through correlation, which measures the linear association among variables. This measure was used

to extract which metrics were most associated to execution time and also to remove the collinearity problem present in software instrumentation data.

Correlation analysis revealed that software instrumentation metrics exhibit collinearity. This implies a redundant information content in the data, limiting the set of statistical methods applicable for its analysis.

Normalization

Most statistical techniques are biased by the magnitude or order of the data to be processed. We *have identified the need of data normalization* before the utilization of statistical methodologies to minimize the bias [97]. Three different normalization schemes were tested on performance data:

- log normalization
- min-max normalization
- dimension normalization

Dimension normalization, in which each metric vector is divided by its Euclidean norm, was selected for performance data analysis based on a separability criteria.

Dimensionality Estimation

Dimensionality estimation along with sequential forward feature subset selection were used to identify which metrics are the most important from a large set of performance data [97]. Three methods for estimating intrinsic dimensionality were tested: Cumulative Percentage of Total Variance, Kaiser-Guttman, and Scree test.

Intrinsic dimensionality estimation and unsupervised feature subset selection identified the metrics containing the most performance information. On average, only 18% of the metrics were found to be important.

Feature Subset Selection

Sequential forward search was used and an entropy based cost function was selected as the most appropriate for the type of data we are working with [96]. Entropy measures the amount of information content in the data.

Multidimensional analysis methods have been used in the past to reduce the dimension of performance data. In their work [14], Vetter and Reed used statistical projection pursuit, a multidimensional projection technique to identify “interesting” performance metrics from a monitoring system. In [15], Ahn and Vetter used several multivariate statistical techniques on hardware performance metrics to characterize high-performance computing systems. They specifically evaluated the use of principal component analysis (PCA), clustering, and factor analysis to extract performance information. None of these works have used sequential forward search for the selection of important metrics or evaluated metrics based on the amount of information present in the data. Moreover, our work is the only one which studied the effect of different normalization schemes for performance data analysis and which estimated the dimensionality of the data previous to discarding relevant data.

9.3 Validation

In order to validate our results, two experiments were designed. In the first, an algorithm with an inefficient memory access pattern was used. This was purposely introduced to visualize the effect of algorithms on the metric values. The results of this experiment showed that metrics associated to disk writes and memory access were selected as the most important ones. In contrast to previous experiments, ANOVA results showed that most of the metrics were affected by the selected algorithm. This demonstrated that the chosen algorithm affects the most important metrics of the system.

The second validation experiment was designed as a full factorial fully randomized test of matrix-vector multiplication algorithms. Four factors were studied: problem size, compiler options, algorithm, and data structure. Using screening experiments we studied compiler options to select those having a dissimilar effect on the execution time. A large variety

of metrics were selected as important by the proposed methodology, in contrast to other experiments, for which metrics associated to memory access were the most important ones. This outcome assured that the subset selection mechanism was performing appropriately.

To validate intrinsic dimensionality estimation tests, a synthetic data set was created by a random number generator. Nine columns were generated at random. Subsequent columns were set as multiples of the first nine columns plus noise added to them. The columns of this matrix were then manipulated to have the same mean and variance as the matrix from the data set obtained in Experiment one. When this synthetic performance data matrix was used as input to the three dimensionality estimators we found that all three methods concurred indicating that the dimension of the data set was nine. Figure 8.2 shows the scree test and the K-G criteria for this synthetic data set. This outcome shows that the evaluated dimensionality estimators were consistent.

9.4 Conclusions

In summary, the application of the proposed methodology reveals that a detailed problem study preceding a systematic design of experiments, yields useful data on which appropriate statistical tools can provide unbiased information about the application-system interactions. Moreover the information obtained from this methodology can be converted into appropriate suggestions, observations, and guidelines for the scientific computing expert to tune applications to a particular computing system.

9.5 Future Work

The next step in the development of an automated performance evaluation system would be the design of a knowledge-based system with a set of rules to provide suggestions to scientific programmers.

To obtain additional information about this data set, we might consider the assignment of classes to performance outcomes. Examples of classes that might be appropriate for this purpose include good and bad memory accesses, excessive idle time, large communication

overhead, etc. Once classes are assigned to particular experimental runs, a classifier might be designed for classifying incoming sets of performance metrics. Here, metric space reduction would be particularly useful to improve the accuracy of the classifier. Moreover, subset selection may be wrapped around the classification criteria.

Additional research ideas that have emerged from this work include:

- Establishing a comparison between different entropy estimators for performance data evaluation. This could improve the accuracy of the metric subset selection method.
- Evaluating the use of sequential backward search and oscillating methods for metric subset selection. This evaluation could also lead to accuracy improvement in selecting important metrics.
- Establishing a comparison between hardware and software metrics for performance evaluation. Although this research has been based on software metrics, hardware metrics could provide alternate information content about the observable computing system.
- Comparing results between different architectures and programming paradigms. This could highlight differences and similarities between them, providing additional insight into the automated performance evaluation problem.

APPENDICES

APPENDIX A

Foundations of Computational Science and Engineering

A.1 Mathematical Preliminaries

In this section we will present the basic mathematical concepts which together with other fundamental concepts in computer science will serve as the basis for our theoretical framework formulated throughout the work.

We will start describing concepts such as information, signal, function, and vector space. We will then continue with mathematical concepts associated with the scientific and engineering applications treated in this thesis.

In the most general sense, in this work, the concept of *information* can be defined as anything which can be sent from one point to another in the physical world. A *signal* can be defined as the entity which carries information. There cannot be a transferring of information from a given point to another without an associated signal.

It is important to point out that we define a *high performance computing machine* in the most general sense as a computational structure with a well defined number of computing processors or nodes and an associated network topology.

Definition A.1 *Cartesian Product of Two Sets.*

Let A and B be any two arbitrary sets. The *Cartesian or direct product* of the set A

times B is a new set, denoted by $A \times B$ defined as follows:

$$A \times B = \{(a_k, b_l) : a_k \in A, b_l \in B\}. \quad (\text{A.1})$$

The above expression is read as follows: $A \times B$ is the set formed with all ordered pairs (a_k, b_l) such that a_k belongs to the set A and b_l belongs to the set B .

In general, the Cartesian product of N sets, for instance, $A_0, A_1, A_2, \dots, A_{N-1}$, is a new set defined as follows:

$$\begin{aligned} A_0 \times A_1 \times \dots \times A_{N-1} &= \{(a_{k_0}, a_{k_1}, \dots, a_{k_{N-1}}) : \\ &a_{k_0} \in A_0, a_{k_1} \in A_1, \dots, a_{k_{N-1}} \in A_{N-1}\}. \end{aligned} \quad (\text{A.2})$$

Definition A.2 *Relation*.

Let $A \times B$ be the Cartesian product of the sets A and B . The *relation* ρ defined on this set is a proper subset of $A \times B$. That is $\rho \subset A \times B$. We call a set G a proper subset of a set H if G is not the null set or the set H itself. If $\rho \subset A \times B$ and $(a_k, b_l) \in \rho$, we then say that a_k is related to b_l .

Definition A.3 *Function*.

Let $A \times B$ be the Cartesian product of the sets A and B . A *function* f defined from A to B is a relation such that the first entry of every pair in the relation is unique; that is, it only appears once and only once through the pair of elements of the relation. If the relation $f \subset A \times B$ is a function, we then call the set A the domain of the function and the set B the co-domain of the function. We will use the following notation to describe a given function $f \subset A \times B$:

$$\begin{aligned} f : A &\longrightarrow B \\ a_k &\mapsto b_l = f(a_k). \end{aligned} \quad (\text{A.3})$$

Definition A.4 *Natural Indexing Set*.

We define the set $Z_N = \{0, 1, 2, \dots, N-1\}$ as the *natural indexing set* of N objects.

Definition A.5 *Mathematical Signal.*

A *mathematical signal* is defined as any mathematical function used to represent a physical signal. Not all physical signals admit mathematical representation and not all mathematical functions can be associated with the physical world (see Figure A.1).

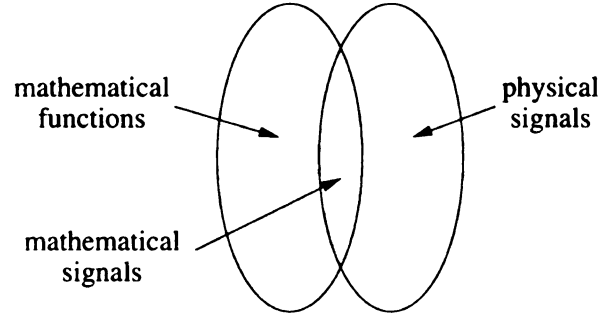


Figure A.1. Venn diagram of mathematical signals.

In this work we are interested in physical signals that admit mathematical representations. Most of the signals used in this work will be of the random or statistical nature.

A signal is called *real* or *complex* if its co-domain is the set of real numbers or the set of complex numbers, respectively. A signal is said to be *continuous* if its domain is a continuous subset of the set of real numbers. We call a signal a *discrete signal* if its domain is a subset in one to one correspondence with a subset of the set of integers. Finally, a signal is said to be a *digital signal* if its co-domain is a finite set.

Definition A.6 *Metric Space.*

A *metric space* (X, d) is a set X with a map $d : X \times X \rightarrow \mathbb{R}^+ \cup 0$, such that

1. $d(x, y) = 0 \Leftrightarrow x = y$
2. $d(x, y) = d(y, x)$
3. $d(x, z) \leq d(x, y) + d(y, z) \forall x, y, z.$

The function d is called a *metric* on X [98].

Let A be the set of possible states in the computer system. The function f that relates $f : A \rightarrow \mathbb{R}$ is called a *measurement*. A measurement describes a physical characteristic of the system under study.

A.1.1 Other Terms

The following terminology will allow us to describe unequivocally the context of our work and its scope.

A *model* is a mathematical expression describing the behavior of a system which can predict the observation based on a error measure. A model is good depending on the criteria selected to determine the modeling error.

A *System* is a set of objects and their interrelationships according to a prescribed set of rules. An *Observable Computing System or OCS* is any given computing system with a defined set of observable measures. *Observable* is the physical manifestation of a given quantity or variable. An *observable* is capable of exchanging information between an observer and a system.

A *stochastic process* is an indexed family of random variables over the same sample space. This index is typically time. A stochastic process is mean-square ergodic in the mean if the corresponding time average converges to the ensemble average in the mean-square sense. A random sequence $X[n]$ converges in the *mean-square sense* to the random variable X if $E\{|X[n] - X|^2\} \rightarrow 0$ as $n \rightarrow \infty$, where $E\{\}$ denotes expected value. The expected value of a discrete random variable is defined as

$$E[\mathbf{X}] = \sum_i x_i P_{\mathbf{x}}(x_i) \quad (\text{A.4})$$

where $P_{\mathbf{x}}(x_i)$ denotes the probability mass function of the discrete random variable $\mathbf{X}$ and it is defined as $P_{\mathbf{x}}(x_i) = P[X = x_i]$ [99].

A.2 Application

Our case study uses finite elements analysis for a computational electromagnetic application. It uses an iterative solver with a diagonal preconditioner to find the solution.

A.2.1 Finite Elements Analysis

In engineering sometimes problems are not easily solved using analytical methods. Finite Element Analysis (FEM) is a numerical method used to solve problems in areas such as aerospace, automotive, civil, mechanical and electrical engineering that involve differential equations. These equations are transformed into a finite dimensional space for solution purposes. The general procedure of this method is to take a large system under study and divide it into smaller elements of finite dimensions. These elements are called *finite elements*. These elements are joined together to form the larger system through “nodes”. Equations for individual elements are formulated and solved taking into consideration boundary conditions. The use of finite elements convert the problem to a solution fo a system of linear equations.

A.2.2 Iterative Solvers

The solution of a large system of linear equations can be found either using direct or iterative solvers. Direct solvers determine the solution in a finite number of steps, while iterative solvers begin with an initial guess of the solution and iteratively improve it until a *good enough* solution is obtained. Iterative methods can be either stationary or nonstationary. In stationary methods, computation of the next step is based on a matrix-vector multiplication operation plus a vector addition. These operators do not vary from iteration to iteration. In non stationary methods, the information required for the next approximation vary with each iteration [100].

Stationary methods include the Jacobi and the Gauss-Seidel. Nonstationary methods include the conjugate gradient (CG), Generalized Minimal Residual (GMRES), BiConjugate Gradient (BiCG), Conjugate Gradient Squared (CGS), and Biconjugate Gradient Stabilized

(Bi-CGSTAB). Iterative methods implemented in the target code are BiCG, CGS, and Bi-CGSTAB.

The rate at which iterative methods converge to the solution depends on the eigenvalues of the coefficient matrix. The convergence rate of the method can be improved through *preconditioning*. This step transforms the system into an equivalent one with the same solution but with different eigenvalues [100, 101]. A diagonal preconditioner is commonly used for this purpose. If A denotes the coefficient matrix in the linear system, the matrix for a diagonal preconditioner is formed by

$$C_{ij} = \begin{cases} a_{i,i} & \text{if } i = j, \\ 0 & \text{otherwise} \end{cases} \quad (\text{A.5})$$

A matrix-vector multiplication algorithm is the basis of the iterative method used in our case study. We now proceed to explain matrix-vector multiplication schemes.

A.2.3 Matrix-Vector Multiplication

Operations on vectors and matrices are the basis of our work.

Let $\mathbb{N}$ be the set of natural numbers. For $m, n \in \mathbb{N}$, a rectangular array A of $m \times n$ elements belonging to a field $\mathbb{F}$ is called a *matrix* and is represented as

$$A = (a_{ij}) = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \quad (\text{A.6})$$

where $a_{ij} \in \mathbb{F}$ for all $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$. The parameter m represents the number of rows and n the number of columns [102].

Special matrices

Let $\mathbb{R}^n$ be the vector space of real n -vectors such that $x \in \mathbb{R}^n \Leftrightarrow x = \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix}^T$ where $x_i \in \mathbb{R}$. A *column vector* x is a $n \times 1$ matrix with n components [55]. Let $\mathbb{C}^n$ be the vector space of complex n -vectors. Then $x \in \mathbb{C}^n$ is a *complex vector*.

A matrix is called *square* if its number of rows is equal to its number of columns. Let A be a $n \times n$ real matrix. A is *symmetric* if $(a_{ij}) = (a_{ji})$ [103]. Let B be $m \times m$ complex matrix. B is called *complex symmetric* if $(b_{ij}) = (b_{ji})$.

Definition A.7 Matrix-Vector Multiplication.

Let $A \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^n$, that is, A is a $m \times n$ real matrix and x is a n -elements vector. The *matrix-vector product* is the m -elements vector $y = [y_i]$ with $y_i = \sum_{j=1}^n a_{ij}x_j$.

The matrix-vector multiplication operation can be viewed as an inner product or as a linear combination of vectors [104]. Consider matrix A a combination of row vectors, that is

$$A = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \quad (\text{A.7})$$

Then the matrix-vector multiplication can be viewed as an inner product of vectors a_i with vector x . The following algorithm will perform the multiplication as an inner product:

```

y = 0 for i = 1 to m
  for j = 1 to n
    y(i) = y(i) + a(i,j)x(j)
  end
end
end
```

If matrix A is stored by rows, as the language C does, it will favor this type of algorithm. On the other hand, the operation can be viewed as a linear combination of the columns of A . Consider A to be a combination of column vectors, that is

$$A = \begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix} \quad (\text{A.8})$$

Then the matrix-vector multiplication can be performed by multiplying each vector a_i with element x_i . The following algorithm will perform the multiplication in this fashion:

```
y = 0
for j = 1 to n
  for i = 1 to m
    y(i) = y(i) + a(i,j)x(j)
  end
end
```

This scheme will favor matrices stored by columns, like Fortran's convention.

From the computational point of view, the organization of the matrix in memory and the scheme used to access memory locations affect the solution time. The effect of the size of cache memory on performance is of great importance. An example on a Sun platform performance in [50] show that 96% of cache hit may cause a program to be running at half of its potential speed. When programming algorithms to perform matrix-vector multiplication the effect of the architecture is an important consideration since different memory access patterns will favor or degrade on performance.

A.3 Advanced Architectures

The computing power requirements for scientific applications have led to the development of different approaches for meeting the demands of processing speed, memory size and latency, and data input/output rates. Increases in performance have come from several advances [105]. Some of these advances relate to:

- Microprocessors have become faster through the use of instruction-level parallelism, multilevel caches, and faster clock speeds.
- Different schemes have been developed for effective interconnection between processors and memory.
- Users and compiler developers have learned how to use multiple processors and deep memory hierarchies.

- Software tools have been improved.

Technology changes fast and processors architectures change in months [106].

In high-performance systems, efficient memory access schemes are needed. Multiple level caches are used to speed up data and instruction accesses. Instruction reordering and data prefetching are used to avoid latency caused by slow memories. This causes compilers to be left with the task of generating efficient codes to take advantage of the hardware. Memory access in shared memory systems has to incorporate cache coherence mechanisms to avoid the access of non-valid data from cache.

Processor architectures for high performance systems derive speed mainly from high clock rates, deep cache memory hierarchies, superscalar-superpipeline designs, out of order execution, and branch prediction schemes [107]. One example of a microprocessor for high performance is the Intel Itanium 2 processor. The Itanium 2 is a 64-bit processor running at 1.3, 1.4, or 1.5 GHz, with three levels of cache: L1 is 32KB, L2 is 256 KB, and L3 is 3, 4, or 6 MB of cache. It is based on the EPIC (Explicitly Parallel Instruction Computer) architecture. This architecture allows programmers or compilers to explicitly indicate parallelism to the processor [108]. The Itanium 2 has six arithmetic logic units and four memory ports allowing two integer loads and two integer stores per cycle [109].

Interconnection networks and system architectures are two important issues for achieving high performance in computing systems. Among the currently used architectures used for advanced systems we can find shared memory, distributed memory, distributed-shared memory, and grid computing. In shared memory systems, parallelism is implemented when processors write and read from a global shared address space. Cache coherence mechanisms are required for preventing problems with data consistency. Distributed memory, distributed-shared memory, and grid computing, on the other hand, are implemented through the use of interconnection networks. In distributed memory, processors collaborate to the solution of a problem communicating via a local area network. One example of this type of architecture is a cluster. Distributed-shared memory is a hybrid, combining shared memory with a distributed memory system. Each node is composed of a shared memory

collection of processors and they are then interconnected as a cluster.

Grid computing is another approach to obtain high-performance systems. The idea behind grid computing is to obtain a highly powerful distributed system composed of physically distributed computers to obtain the best resources available to jointly solve a problem. The system should be transparent to the users who will concentrate on the solution of their respective problems and not on the computational requirements of the problem [110].

There are several issues in the development of such a global computing platform, such as software compatibility, high-performance networks, security, and user-friendly interfaces. Grid computing involves the interconnection of high-performance networks, implementing a distributed file system, coordinating user access to different computational structures, and making the environment easy-to-use and transparent to the user.

A.4 Languages and Environments

Two main programming styles are used for programming in parallel. One follows the shared memory model and the other, the distributed memory model.

A.4.1 Shared Memory

OpenMP is an application program interface designed for supporting shared-memory multi-threaded parallel programming. It is based on the fork-join model of parallelism where multiple threads will share the workload.

OpenMP is a standard developed by a group of hardware, software, and application vendors and it consists of a group of compiler directives and library for shared memory parallelism. These libraries can be used with C, C++, and Fortran. The advantage of OpenMP over other parallel libraries is the simplicity of its use. Parallelism can be incorporated incrementally into a program which was originally designed to be run in serial mode. In other models, such as message passing, the introduction of parallel constructs is more complex and time consuming.

A.4.2 Message Passing

Traditionally, message passing libraries have been used for parallel programming. These consist of a set of routines and libraries for the execution of point to point communication among processors. The standard for message passing libraries is called MPI.

MPI implements a distributed memory paradigm. The two most widely used implementations of MPI are MPICH and LAM. MPI is used with C, C++, or Fortran. It is based on point-to-point communication between processors. MPI implements both blocking and non-blocking instructions.

A.4.3 Problem Solving Environments

A problem solving environment (PSE) is an integrated computing system for developing and running applications in a particular domain with the goal of improving the productivity of research scientists by providing an natural interface to construct applications [111]. PSEs have been cited as one of the key technologies required for enabling petascale computing on real applications by 2010 in [112].

A.5 Performance Measurement

Collecting information about tasks performed while the execution of a program is done through instrumentation and performance metrics. Instrumentation is the process of generating a trace of the execution of the program either through software or hardware [113]. Performance metrics is the data collected by the instrumentation system [114] which provides information about the status of the code at different times while the code is executing. Some of the performance metrics are summaries of statistics while others contain detailed information of the status of the system at different times. A trace is a series of measurements that provide information on the status of a system over a period of time.

Performance instrumentation can be inserted at different levels in the system, that is at hardware, system software, run time software, and application code [115]. Instrumentation calls for the application code can be inserted at different points in the software life cycle,

that is, at the source code, at compile time, at object code while linking the libraries, and at the running executable file.

The conventional method used for performance optimization is to find the most time consuming kernel of a program and optimizing it [116].

A.5.1 Tools

A great effort has been placed on the development of tools for high-performance computing [117, 118]. Given the high complexity of interactions among different components in a high performance systems, tools are needed to aid the application programmers to develop their applications. Tools for the development of serial code are very mature: there are a variety of debuggers and profiling tools to aid programmers in code development and performance monitoring. Tools for parallel code development do not follow a standard. These tools take different approaches, different data formats, and different display techniques [119].

There are three steps in performance analysis: data collection, data transformation, and data visualization and rendering. One data collection technique is the use of a profilers, which record the amount of time spent in different parts of the program. Gprof is an example of such a tool for the Unix operating system. Data transformation and visualization modifies and presents performance data to be comprehensible and useful to the user.

It is important that tools provide the appropriate visualization techniques to display information. Two basic principles for visualization of performance data are given in [120]. The first one states that the displays should be linked directly to the performance model. The second one suggests that visualization techniques should be designed and applied in an integrated environment. Therefore the selection of the appropriate models for showing performance information to the user is extremely important.

Paradyn, ParaGraph, AIMS, VAMPIR, Pablo, Scalea, and KAP/PRO are examples of performance tools for the evaluation of parallel programs [121, 122, 123]. ParaGraph, AIMS, and VAMPIR are tools for the visualization of message-passing parallel programs. Pablo and Scalea support both message-passing and data-parallel programming models. KAP/PRO supports shared memory parallel programming.

Paradyn is one of the most successful tools for performance diagnosis and it was the first tool to follow an automated search approach [124]. When it was first introduced, it combined several novel technologies, two of which were unique in its class: dynamic instrumentation and automated bottleneck search. Dynamic instrumentation is the process of instrumenting executable code while it is executing. It executes its instrumentation through the use of “trampolines”, points in the code where the instrumentation is inserted, and the code is deviated from its normal flow to an alternate path to collect information. Since Paradyn does not require either compilation or linking with source code, it is one of the few tools available which can perform instrumentation on proprietary code. The second technology Paradyn introduced is automated bottleneck search. Paradyn uses of a set of general hypothesis about why, where, and when there is a performance bottleneck and it will use instrumentation to collect information on whether the hypothesis is true or false. A predefined set of thresholds defined by the user will be used for hypothesis testing. Paradyn is used for long running codes of hours or days.

Scalea is another performance analysis tool for parallel programs [125]. It is composed of an instrumentation system, runtime system, a performance repository, and a performance analysis and visualization system. It supports OpenMP, MPI, HPF, and OpenMP/MPI programs. Two novel features about Scalea is the classification of performance overheads and the support for multiple experiments.

Once the application is targeted to a specific platform statistics may be used to study the behavior of the system.

A.5.2 Statistical Terms

In this section we present some statistical terminology useful in our work.

Accuracy refers to how close a measurement is to the real or actual value of a physical quantity. *Precision* determines how close are measurements within one another, independent of whether or not a measurement is accurate. Precision indicates the reproducibility of repeated measurements under the same conditions.

Datum is an observation obtained from our system. More than one observation is

collected as data. There are two different types of association among data observations: descriptive and experimental. *Descriptive* relations are those involved data not controlled by any means. We observe the system without controlling any factors and we establish relationships just from the observations. *Experimental* relationships, on the other hand are those in which an experiment is conducted and data collected. In this case, causal relations can be established [9]. We call an *object* to the entity producing data. A *variable* is a feature of the system with two or more values. The values of a variable are represented on one of four different scales: *nominal*, *ordinal*, *interval*, or *ratio*. A *nominal scale* refers to variables whose values cannot be ranked in any order. Variables differ by kind or category only. *Ordinal scale* is that in which the values can be ranked in order and they are represented by numbers but they represent a hierarchy. Both nominal and ordinal scales are considered non-metric scales. Third, in the *interval scale*, equal differences between scales have meaning but ratios have no meaning. Finally, the *ratio scale* carries the most information. Here ratios have meaning and there is a zero point in the scale. Interval and ratio scales are considered metric scales.

A *statistical hypothesis* is a statement about the characteristics of a population. The claim initially believed to be true is called the *null hypothesis* or *prior belief* and is denoted by H_0 . The *alternative hypothesis* is a statement contradictory to the null hypothesis and is denoted H_1 . For example, one experiment may test the effect of problem size on execution time for two different problem sizes. If it is believed that problem size does not affect execution time then

$$\begin{aligned} H_0 : \mu_1 &= \mu_2 \\ H_1 : \mu_1 &\neq \mu_2 \end{aligned} \tag{A.9}$$

where μ_1 is the mean execution time for problem size one and μ_2 is the mean execution time for problem size two.

Hypothesis testing is a method to sample data to decide if the null hypothesis should be rejected or not. A method based on sample data to decide whether or not to reject H_0 is called a *test procedure*. It is composed of a *test statistic* and a *rejection region*. A test

statistic is a function computed with sample data used to reach a decision about H_0 . A rejection region is a set of values of the test statistic for which H_0 will be rejected. There are two different types of errors in hypothesis testing: type I and type II [126].

- **Type I error:** The probability of error of selecting the alternate hypothesis when the null hypothesis is true.
- **Type II error:** The probability of error of selecting the null hypothesis when it is false.

A type I error is denoted by α (also called *alpha value*). Typical values of α to make a decision are 0.10, 0.05, and 0.01. The most commonly used value of α is 0.05. The *F-ratio* is the ratio of two sampling variances. The *p-value or significance level* is the probability of obtaining a statistic value as contradictory to the null hypothesis as the resulting one when assuming that the null hypothesis is true [126]. It is determined by the *F* ratio and the degrees of freedom associated to each sampling variance. The degrees of freedom is the number of samples in the data that are allowed to vary when the statistic is computed. The p-value will tell us the probability of obtaining the value of the resulting *F* ratio or larger by chance [127]. The smallest the p-value, the more contradictory is the data to the null hypothesis. If $p - value \leq \alpha$, the null hypothesis is rejected at level α . If $p - value > \alpha$, then the null hypothesis is not rejected at level α .

The methodological design of an experiment is done to obtain the most information possible with the smallest number of tests. A *factor* is an independent variable influencing the results and might have two or more *levels*. Factors are classified as design, held-constant, and allowed-to-vary factors [25]. *Design factors* are controlled in the experiment, *held-constant factors* will be kept at a specific level during experimentation, and *allowed-to-vary factors* are ignored and not controlled. We may also have *nuisance factors*, which are factors not considered in the experiment. An *experiment* is a study in which changes are made to controlled inputs to the system to observe and understand effects on the output. A *replication* is a repetition of the same experiment. *Randomization* refers to the arrangement

of individual runs of the experiment in random order. A set of levels of controllable factors administered to an experimental unit is called a *treatment*.

Design of experiments (DOE) refers to the way in which the experiment is arranged, specifically, the way in which the treatments will be administered to the subjects in the study. A correct design will minimize the effect of uncontrollable factors and will determine whether variations in the output are random or significant effects. An *experimental unit* or *experimental run* is the basic unit to which a treatment is applied. *Experimental error* refers to a measure of the variations in the observations with the same treatment.

The process of designing an experiment involves seven steps [25]:

1. **Problem Statement:** Establishment of the goals of the experiment.
2. **Selection of factors and levels:** Classification of factors into design, held-constant, or allowed-to-vary factors. Selection of levels to be tested in each factor.
3. **Decision on the output variable:** Identification of the response variable for the experiment. In our case, a set of multiple responses are used as output variables.
4. **Selection of an experimental design:** Selection of the number of replicates, number of samples to take, the order of runs for experimental units, and the randomization scheme to be used.
5. **Perform the experiment:** Actual experimental run.
6. **Analysis of data:** Selection of a statistical model for the response variable. Test of the model.
7. **Conclusion:** Formulation of conclusions drawn from the experiment.

There are three basic criteria to consider in an experiment: replication, randomization, and blocking. *Replication* refers to when a treatment is applied more than once in an experiment [128]. It improves precision and allows the calculation of an estimate of the experimental error. We require to have at least two replicates of an experiment [25].

Randomization refers to the order in which experimental runs will be executed. Statistical methods require that the measurements must be independent and identically distributed (iid) random variables. This is ensured by randomization and it will average out the effects of nuisance factors. *Blocking* is the allocation of experimental runs into homogeneous conditions to improve comparisons. Blocking restricts complete randomization since factors are only randomized within a block.

Some types of experiment designs are: simple, factorial, and full-factorial. A *simple design* is the most basic experiment. It refers to an experiment with only one factor [26]. If the factor only has two levels, that is, we are comparing two treatments, it is called a *simple comparative study*. We can use this type of experiment for screening purposes, but for complex interactions as those obtained in high performance computing systems, they are limited in the information they can provide.

In a *factorial design*, two or more factors are tested simultaneously. This allows to detect interactions among factors. When the response of a factor depends on the level of another factor we say there is *interaction* between them. For illustrative purposes we present an example of interaction.

Assume we are studying the effect of problem size and the machine type on the execution time for a particular application. We can plan two different experiments: one to test problem size and another to test the type of machine. In our first experiment we select machine A and vary the problem size from size 1 to size 2. In the second experiment we select problem size 1 and vary the type of machine: A and B. Assume we get the results shown in Figure A.2.

From these graphs we might conclude that problem size does not affect execution time while machine type does. However, it might have been possible that if the test of problem size had been done on machine B, we could have a graph like in Figure A.3.

In this second graph we can definitely see that problem size affects execution time. When we examine Machine A, execution time stays almost constant when varying problem size, but when Machine B is used, execution time varies as problem size changes. There

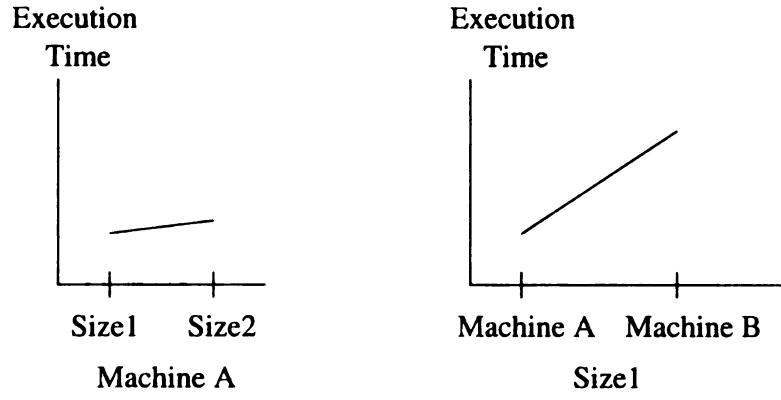


Figure A.2. Experiment illustrating execution time of two simple comparative studies.

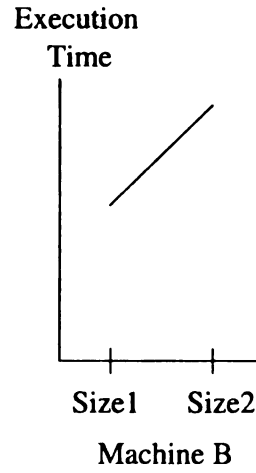


Figure A.3. Execution time when Machine B is used in the study .

is *interaction* between problem size and machine because variations in problem size have a different effect under different levels of the factor machine type.

A *full-factorial design* involves studying every level of all combinations of factors [129] at the same time. If we let F denote the number of factors and l_k denote the number of levels for factor k , the total number of experimental runs for one repetition of the experiment is represented as:

$$TotExp = \prod_{k=0}^{F-1} l_k. \quad (A.10)$$

A full-factorial design involving only two levels for each factor is called a 2^k *factorial*

design.

The randomization scheme is of importance in deciding a specific design of experiment. In a *completely randomized design*, the order in which experimental runs are arranged is randomly allocated. When in a factorial experiment we are unable to completely randomize the order of the runs, a *split-plot* design might be used. In this design, one factor is selected for a treatment, and the order in which the treatments are applied to this factor is chosen either at random, or on blocks. Next, a second factor is selected and, keeping the order for experimental runs selected for the first factor, a randomization scheme is selected for the second factor. This could be repeated successively. When a third factor follows the same restrictions, this is called a *split-split plot* design [25]. A partial randomization of experiments causes a higher experimentation error and a more complex computer analysis so split-split plot is suggested only when a completely randomized design is not feasible.

We will illustrate the concept with an example. Imagine we have two different servers and we want to measure system response time under three different types of workloads. Suppose that there is a restriction that we can only experiment on the servers at certain times of the day, and not simultaneously. We have two factors: server type and workload, with two and three levels each, respectively.

- Factor A: Server 1 (S1) and Server 2 (S2)
- Factor B: Workload 1 (W1), Workload 2 (W2), and Workload 3 (W3).

The *Yates* or *standard order* to list experiments is the following: first, factors are listed in alphabetical order and then the levels are listed from lowest to highest level. This does not correspond to the *run order*. The order of running experiments should be randomized to minimize the influence of nuisance factors.

In this example, the standard order is as shown in Table A.1.

For a fully randomized experiment, the last column shows a typical order for experimental runs. That is S2W2, S1W3, S2W1, S2W3, S1W2, and S1W1. As can be noticed from this order, we start running on server one, then change to server two, and then switch back

Table A.1. Order of experiments for a fully randomized experiment.

Standard Order	Factor A (Server)	Factor B (Workload)	Fully Randomized
1	S1	W1	6
2	S1	W2	5
3	S1	W3	2
4	S2	W1	3
5	S2	W2	1
6	S2	W3	4

to server one. This is impractical given the existing restrictions on the use of the servers. A more appropriate design would be to select randomly the server to run the experiment on, and then, select the workload. For example:

- Server 2, Workload 2
- Server 2, Workload 1
- Server 2, Workload 3
- Server 1, Workload 1
- Server 1, Workload 3
- Server 1, Workload 2

This is called a split-plot design. The *whole-plot* is the server type and the *subplot* is the workload.

Data obtained from a design of experiment can readily be analyzed with ANOVA procedures.

ANOVA

Analysis of variance (ANOVA) is a statistical procedure for the analysis of the response of an experiment to identify what is the cause of variations in the obtained data. In ANOVA,

the goal is to determine if there is an effect of different treatments on a population. The null hypothesis tested by ANOVA is that no factor will influence the solution and that there is no interaction between any factors. Once the alpha level for the ANOVA test is selected, a set of test statistics are computed and a conclusion on whether the null hypothesis is probable or not is reached. In our case, ANOVA at α level 0.05 will be used to establish relationships among factors and performance metrics. The ANOVA test is used when there are more than two treatments to compare.

There are three assumptions for the ANOVA test. First, the treatments are independent of each other. This is assured by the use of randomization in the experiment. Second, the distribution of the sample means should be normal. This is ensured by having a large enough group of samples. Finally, the variances of the groups should be approximately equal. This is known as the homogeneity of variance assumption. ANOVA is robust to deviations from these assumptions if the design is *balanced*, that is, if the number of samples from each population are the same.

A typical ANOVA procedure is summarized in the following steps:

- Assume the null hypothesis H_0 is that all means are equal.
- Assume that the alternative hypothesis H_1 is that at least one mean is different.
- Assume
 - Treatments are independent
 - Treatments follow a normal distribution
 - Homogeneity of variance
- Set the α level, that is the allowed type I error on the results.
- Determine the F ratio.
- Determine the p -value and conclude based on whether p -value $> \alpha$. If p -value $> \alpha$ then the null hypothesis is assumed true. Otherwise, we conclude that there are significant differences in the means of the population.

If the null hypothesis is true, this means that factors do not affect results. For example, if we obtain a $p - value > 0.05$ in our previous example, this means that for this significance level, neither the server type, nor the workload significantly affect the execution time. Variations in execution time are due to the random nature of the measurement.

On contrast, if the null hypothesis is false, at least one of the means is statistically different than the others. This does not determine which ones are different or how significantly different they are. Multiple comparisons are used to determine this. Two types of comparisons are used: *a priori* comparisons and *a posteriori* comparisons [130]. *A priori comparisons* are planned before the experiment and are based on a previous theory we might have about the data itself. *A posteriori* or *post hoc comparisons* are not planned and are done after the data is collected to propose a hypothesis [130]. A typical *a priori* comparison used in statistical analysis is orthogonal contrasts, in which specific comparisons are studied [25]. Some methods used for *post hoc* comparisons are Least Significant-difference (LSD), Student-Neuman-Keul's, Tukey's, and Scheffe's test. In most research situations, the outcome is the same regardless of the test used.

A.6 Summary

This chapter has presented the foundations for this dissertation. First a presentation of the mathematical foundations is found. A mathematical signal was defined as a mathematical function used to represent a physical signal. The response of our system can be regarded as a physical signal. Finite elements and iterative solvers are the basis of our case study. These are presented along with matrix-vector multiplication algorithms, which is the kernel routine in our case study. Advanced architectures are described. Important characteristics of advanced architectures are high clock rates, deep memory hierarchies, superscalar-superpipeline designs, out of order execution, and branch prediction algorithms. We continue by describing languages for parallel programming and tools for improving performance. The two most widely used languages for parallel programming are OpenMP and MPI. Performance tools collect information about the system and present it in a useful way.

Two performance tools presented as examples are Paradyne and Scalea. Some important statistical terms were defined. We are using hypothesis testing to determine if the results were due to chance or to a significant effect of a factor. Design of experiments (DOE) refers to the careful arrangement of experimental treatments in a systematic study. ANOVA is used for the analysis of the response of an experiment. *Post hoc* comparisons are used for classifying differences in the data.

APPENDIX B

Glossary

Abstraction: The act of leaving out of consideration one or more properties of a complex object so as to attend to others to analyze or classify it. The process of formulating general concepts by abstracting common properties of instances. A general concept formed by extracting common features from specific examples. Generalization ignoring or hiding details to capture some kind of commonality between different instances. Each abstraction has a number of primitive elements and composition rules [131].

Accuracy: How close a measurement is to the real or actual value of a physical quantity.

Algorithm: An algorithm is a well defined procedure to solve a problem in a finite number of steps.

Algorithm Performance: A measurement of the computer performance of the implementation of an algorithm.

Conceptualization: The process of developing a new idea to solve a problem.

Correlation: A statistic representing how closely two variables co-vary; it can vary from -1 (perfect negative correlation) through 0 (no correlation) to +1 (perfect positive correlation). A measure of how strongly related two variables x and y are in a sample [126]. Coefficient indicating the linear relationship between a variable and another. A correlation coefficient close to one indicates high correlation. It is computed as the covariance between two variables divided by the product of the standard deviation of each variable.

Computational Electromagnetic: The application of numerical methods for the

solution of partial differential equations and of integral equations in the application of electromagnetic to areas such as guided waves, antennas, and scattering [39].

Computer performance characterization: A detailed description of the operation of a computer executing a set of instructions. It includes information on hardware and software execution.

Implementation: An implementation is the task of turning an algorithm into a computer program.

Instantiation: To describe the idea as a series of steps to solve the problem. An instantiation is a collection of algorithms to solve a problem.

Instrumentation: Instrumentation is the group of modules to collect and manage from a program while it runs on a parallel or distributed system [56].

Mapping Abstraction: A rule of correspondence established between sets that associates each element of a vocabulary describing a high-level abstraction with an element in the set of concrete architectures [6].

Metric: The valuations of observable quantities of a target computing system stored in the form of variables.

Observable: The physical manifestation of a given quantity or variable.

Observable Computing System or *OCS*: Any given computing system with a defined set of observable measures.

Operation: A process or an action, such as addition, substitution, transposition, or differentiation, performed in a specified sequence and in accordance with specific rules.

Parallel Programming: The decomposition of a program for executing in multiple processing units at the same time.

Precision: How close are measurements within one another, independent of whether or not a measurement is accurate.

Random Variable: Function whose domain is the sample space and of an experiment and whose range is a subset of the real line.

Relation: A subset of the product of two sets, $R: A \times B$. If (a, b) is an element of R

then we write $a R b$, meaning a is related to b by R . A relation may be: reflexive ($a R a$), symmetric ($a R b \Rightarrow b R a$), or transitive ($a R b$ and $b R c \Rightarrow a R c$).

System: A set of objects and their interrelationships according to a prescribed set of rules.

APPENDIX C

Matrix-Vector Multiplication Algorithms

C.1 Algorithm A

This is the original matrix-vector multiplication algorithm used in the application but modified for OpenMP. This algorithm is more inefficient when running in serial mode than the original algorithm implemented in serial mode but when in parallel, avoids thread interaction and allows implementation using OpenMP. It causes converges in the code both when running serially and in parallel.

```
Subroutine BiMATVECCav(vector,product)
Implicit NONE
Include 'prism.inc'
Complex*16 vector(*),product(*)
c c Local variables
Integer row,col,index
Complex matEntry
c c Do the MATVEC c
!$OMP PARALLEL PRIVATE(index, col, matEntry)
!$OMP DO
Do row = 1,apUnk
Do col = 1,apUnk
If(row .LT. col) Then
index = BIrowEndPoint(row)+col
Else
index = BIrowEndPoint(col)+row
EndIf
matEntry = Ybi(index)
```

```

        product(row) = product(row) + matEntry*vector(col)
    EndDo
EndDo
!$OMP END DO
!$OMP END PARALLEL
Return
End

```

C.2 Algorithm B

This algorithm is similar to algorithm A but removes the if condition by doing two different loops which splits the matrix by the opposite diagonal elements. There is no thread interaction so it causes convergence both when running serially and in parallel.

```

Subroutine BiMATVECCav(vector,product)
Implicit NONE
Include 'prism.inc'
Complex*16 vector(*),product(*)
Integer row,col,index
Complex matEntry
c c Do the MATVEC c
!$OMP PARALLEL PRIVATE(index, col, matEntry)
!$OMP DO
    Do row = 1,apUnk
        Do col = 1,row
            index = BIrowEndPoint(col)+row
            matEntry = Ybi(index)
            product(row) = product(row) + matEntry*vector(col)
        EndDo
    EndDo
!$OMP END DO
!$OMP END PARALLEL

!$OMP PARALLEL PRIVATE(index, col, matEntry)
!$OMP DO
    Do row = 1,apUnk
        Do col = row+1,apUnk
            index = BIrowEndPoint(row)+col
            matEntry = Ybi(index)
            product(row) = product(row) + matEntry*vector(col)
        EndDo
    EndDo
!$OMP END DO
!$OMP END PARALLEL

Return

```

End

C.3 Algorithm C

Algorithm C is the most inefficient matrix-vector multiplication algorithm we have implemented. It was used to verify the observability of metrics indicating a bad memory access pattern. It prevents convergence of the code when running in parallel.

```
Subroutine BiMATVECCav(vector,product)
Implicit NONE
Include 'prism.inc'
Complex*16 vector(*),product(*)
Integer row,col,index,k
Complex matEntry
c c Do the MATVEC c
!$OMP PARALLEL PRIVATE (index, k, row,matEntry)
!$OMP DO
  Do col = apUnk,1,-1
    k = apUnk -1
    index = col
    Do row = 1,col-1
      matEntry = Ybi(index)
      product(row) = product(row) + matEntry*vector(col)
      product(col) = product(col) + matEntry*vector(row)
      index = index + k
      k = k-1
    EndDo
    product(col) = product(col) + Ybi(index)*vector(col)
  EndDo
!$OMP END DO
!$OMP END PARALLEL

Return
End
```

C.4 Algorithm D

This is the original matrix-vector multiplication algorithm used in the application, similar to Algorithm A. It is serial and causes convergence in the application code.

```
Subroutine BiMATVECCav(vector,product)
```

```

Implicit NONE
Include 'prism.inc'
Complex*16 vector(*),product(*)
Integer row,col,index
Complex matEntry
c c Do the MATVEC c

Do row = 1,apUnk
  Do col = 1,apUnk
    If(row .LT. col) Then
      index = BIrowEndPoint(row)+col
    Else
      index = BIrowEndPoint(col)+row
    EndIf
    matEntry = Ybi(index)
    product(row) = product(row) + matEntry*vector(col)
  EndDo
EndDo
Return
End

```

C.5 Algorithm E

This algorithm is similar to algorithm B but runs serially. It causes convergence in the application code.

```

Subroutine BiMATVECCav(vector,product)
Implicit NONE
Include 'prism.inc'
Complex*16 vector(*),product(*)
Integer row,col,index
Complex matEntry
Do row = 1,apUnk
  Do col = 1,row
    index = BIrowEndPoint(col)+row
    matEntry = Ybi(index)
    product(row) = product(row) + matEntry*vector(col)
  EndDo
EndDo

Do row = 1,apUnk
  Do col = row+1,apUnk
    index = BIrowEndPoint(row)+col
    matEntry = Ybi(index)
    product(row) = product(row) + matEntry*vector(col)
  EndDo

```

EndDo

Return

End

C.6 Algorithm F

Matrix-vector multiplication algorithm described in page 22 of [55] to be used for a validation experiment. It was implemented in parallel using OpenMP directives. To avoid thread overwriting the information an ATOMIC directive is included.

```
Subroutine MatVectMult2(ProblSize,DenseMatrix,VectorIn,product)
c Golub & Van Loan algorithm
Integer ProblSize
Complex*16 DenseMatrix(*),VectorIn(*),product(*)
Integer row,col,index
c c Do the MATVEC multiplication c
!$OMP PARALLEL PRIVATE(index, row)
!$OMP DO
  Do col = 1, ProblSize
    Do row = 1, col-1
      index = (row-1)*ProblSize - row*(row-1)/2 + col
!$OMP ATOMIC
      product(row) = product(row)+DenseMatrix(index)*VectorIn(col)
    EndDo
    Do row = col, ProblSize
      index = (col-1)*ProblSize - col*(col-1)/2 + row
!$OMP ATOMIC
      product(row) = product(row)+DenseMatrix(index)*VectorIn(col)
    EndDo
  EndDo
!$OMP END DO
!$OMP END PARALLEL c
Return
End
```

C.7 Algorithm G

Matrix-vector multiplication algorithm modified to read the data in reverse. It was implemented in parallel using OpenMP directives. We expect it to have a poor performance and it is used for validation purposes.

```
Subroutine MatVectMult5(ProblSize,DenseMatrix,VectorIn,product)
```

```

Integer ProblSize
Complex*16 DenseMatrix(*),VectorIn(*),product(*)
Integer row,col,index,k
Complex*16 matEntry
c Do the MATVEC multiplication
!$OMP PARALLEL PRIVATE (index, k, row, matEntry)
!$OMP DO
    Do col = ProblSize, 1, -1
        k = ProblSize - 1
        index = col
        Do row = 1, col-1
            matEntry = DenseMatrix(index)
!$OMP ATOMIC
            product(row) = product(row) + matEntry*VectorIn(col)
            product(col) = product(col) + matEntry*VectorIn(row)
            index = index + k
            k = k-1
        EndDo
!$OMP ATOMIC
        product(col) = product(col) + DenseMatrix(index)*VectorIn(col)
    EndDo
!$OMP END DO
!$OMP END PARALLEL
Return
End

```

APPENDIX D

Experiment 1

D.1 Order of Execution of Experimental Runs for Experiment 1

The design of the experiment was randomized as a split-split plot design. The main plot was the repetition where three repetitions were done. The subplots were selected at random where problem size and matrix vector multiplication algorithm was selected. In each of these subplots, compiler options for generating the executable files were selected at random using an uniform distribution random number generator. The following table contains the actual order in which the experimental runs were performed given this randomization scheme.

Table D.1. Order of execution of experiments

Experimental Run	Size (N)	Algorithm	Compiler Options
1	6033	B	-fast -WGstats
2	6033	B	-unroll=2 -fast -xcrossfile -WGstats
3	6033	B	No flags -WGstats
4	6033	B	-xcrossfile -fast -WGstats
5	6033	B	-fast -xcrossfile -WGstats
6	6033	B	-fast -xcrossfile -unroll=2 -WGstats
7	6033	B	-xcrossfile -unroll=2 -fast -WGstats
8	6033	B	-unroll=2 -fast -WGstats
<i>continued on next page</i>			

Table D.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
9	6033	B	-unroll=2 -WGstats
10	6033	B	-unroll=2 -xcrossfile -fast -WGstats
11	6033	B	-fast -unroll=2 -WGstats
12	6033	B	-xcrossfile -fast -unroll=2 -WGstats
13	6033	B	-fast -unroll=2 -xcrossfile -WGstats
14	6033	A	-fast -unroll=2 -WGstats
15	6033	A	-unroll=2 -fast -xcrossfile -WGstats
16	6033	A	-unroll=2 -xcrossfile -fast -WGstats
17	6033	A	-xcrossfile -fast -WGstats
18	6033	A	-xcrossfile -fast -unroll=2 -WGstats
19	6033	A	-fast -xcrossfile -WGstats
20	6033	A	-fast -xcrossfile -unroll=2 -WGstats
21	6033	A	-unroll=2 -WGstats
22	6033	A	No flags -WGstats
23	6033	A	-fast -unroll=2 -xcrossfile -WGstats
24	6033	A	-unroll=2 -fast -WGstats
25	6033	A	-xcrossfile -unroll=2 -fast -WGstats
26	6033	A	-fast -WGstats
27	13857	B	No flags -WGstats
28	13857	B	-unroll=2 -xcrossfile -fast -WGstats
29	13857	B	-fast -unroll=2 -WGstats
30	13857	B	-fast -WGstats
31	13857	B	-fast -xcrossfile -unroll=2 -WGstats
32	13857	B	-xcrossfile -fast -WGstats
33	13857	B	-xcrossfile -unroll=2 -fast -WGstats
<i>continued on next page</i>			

Table D.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
34	13857	B	-fast -xcrossfile -WGstats
35	13857	B	-unroll=2 -fast -WGstats
36	13857	B	-xcrossfile -fast -unroll=2 -WGstats
37	13857	B	-fast -unroll=2 -xcrossfile -WGstats
38	13857	B	-unroll=2 -fast -xcrossfile -WGstats
39	13857	B	-unroll=2 -WGstats
40	13857	A	-unroll=2 -fast -xcrossfile -WGstats
41	13857	A	-unroll=2 -WGstats
42	13857	A	-xcrossfile -unroll=2 -fast -WGstats
43	13857	A	-fast -unroll=2 -xcrossfile -WGstats
44	13857	A	-fast -xcrossfile -unroll=2 -WGstats
45	13857	A	-fast -xcrossfile -WGstats
46	13857	A	-xcrossfile -fast -unroll=2 -WGstats
47	13857	A	-xcrossfile -fast -WGstats
48	13857	A	-fast -WGstats
49	13857	A	-unroll=2 -fast -WGstats
50	13857	A	No flags -WGstats
51	13857	A	-fast -unroll=2 -WGstats
52	13857	A	-unroll=2 -xcrossfile -fast -WGstats
53	6337	B	-unroll=2 -WGstats
54	6337	B	-fast -unroll=2 -WGstats
55	6337	B	-unroll=2 -xcrossfile -fast -WGstats
56	6337	B	-fast -xcrossfile -unroll=2 -WGstats
57	6337	B	No flags -WGstats
58	6337	B	-fast -xcrossfile -WGstats
<i>continued on next page</i>			

Table D.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
59	6337	B	-xcrossfile -unroll=2 -fast -WGstats
60	6337	B	-unroll=2 -fast -xcrossfile -WGstats
61	6337	B	-unroll=2 -fast -WGstats
62	6337	B	-fast -WGstats
63	6337	B	-xcrossfile -fast -WGstats
64	6337	B	-xcrossfile -fast -unroll=2 -WGstats
65	6337	B	-fast -unroll=2 -xcrossfile -WGstats
66	6337	A	-fast -unroll=2 -xcrossfile -WGstats
67	6337	A	-xcrossfile -unroll=2 -fast -WGstats
68	6337	A	-fast -WGstats
69	6337	A	-unroll=2 -xcrossfile -fast -WGstats
70	6337	A	-xcrossfile -fast -WGstats
71	6337	A	-fast -unroll=2 -WGstats
72	6337	A	-fast -xcrossfile -unroll=2 -WGstats
73	6337	A	-unroll=2 -fast -WGstats
74	6337	A	-unroll=2 -WGstats
75	6337	A	-unroll=2 -fast -xcrossfile -WGstats
76	6337	A	-fast -xcrossfile -WGstats
77	6337	A	No flags -WGstats
78	6337	A	-xcrossfile -fast -unroll=2 -WGstats
79	13857	A	No flags -WGstats
80	13857	A	-fast -unroll=2 -WGstats
81	13857	A	-xcrossfile -unroll=2 -fast -WGstats
82	13857	A	-fast -WGstats
83	13857	A	-xcrossfile -fast -unroll=2 -WGstats
<i>continued on next page</i>			

Table D.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
84	13857	A	-unroll=2 -fast -WGstats
85	13857	A	-unroll=2 -WGstats
86	13857	A	-unroll=2 -xcrossfile -fast -WGstats
87	13857	A	-fast -unroll=2 -xcrossfile -WGstats
88	13857	A	-xcrossfile -fast -WGstats
89	13857	A	-fast -xcrossfile -WGstats
90	13857	A	-unroll=2 -fast -xcrossfile -WGstats
91	13857	A	-fast -xcrossfile -unroll=2 -WGstats
92	13857	B	-xcrossfile -fast -unroll=2 -WGstats
93	13857	B	-xcrossfile -fast -WGstats
94	13857	B	-unroll=2 -WGstats
95	13857	B	-fast -unroll=2 -xcrossfile -WGstats
96	13857	B	-fast -xcrossfile -WGstats
97	13857	B	-fast -unroll=2 -WGstats
98	13857	B	-unroll=2 -fast -WGstats
99	13857	B	-unroll=2 -fast -xcrossfile -WGstats
100	13857	B	-fast -WGstats
101	13857	B	-fast -xcrossfile -unroll=2 -WGstats
102	13857	B	-xcrossfile -unroll=2 -fast -WGstats
103	13857	B	-unroll=2 -xcrossfile -fast -WGstats
104	13857	B	No flags -WGstats
105	6033	A	-unroll=2 -WGstats
106	6033	A	-fast -xcrossfile -WGstats
107	6033	A	-unroll=2 -fast -WGstats
108	6033	A	-fast -unroll=2 -WGstats
<i>continued on next page</i>			

Table D.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
109	6033	A	-xcrossfile -unroll=2 -fast -WGstats
110	6033	A	-xcrossfile -fast -unroll=2 -WGstats
111	6033	A	-unroll=2 -xcrossfile -fast -WGstats
112	6033	A	-fast -xcrossfile -unroll=2 -WGstats
113	6033	A	-fast -WGstats
114	6033	A	-fast -unroll=2 -xcrossfile -WGstats
115	6033	A	-unroll=2 -fast -xcrossfile -WGstats
116	6033	A	No flags -WGstats
117	6033	A	-xcrossfile -fast -WGstats
118	6033	B	-xcrossfile -fast -WGstats
119	6033	B	-fast -xcrossfile -WGstats
120	6033	B	-xcrossfile -unroll=2 -fast -WGstats
121	6033	B	-fast -xcrossfile -unroll=2 -WGstats
122	6033	B	-fast -unroll=2 -xcrossfile -WGstats
123	6033	B	-unroll=2 -fast -xcrossfile -WGstats
124	6033	B	-unroll=2 -WGstats
125	6033	B	-fast -unroll=2 -WGstats
126	6033	B	-unroll=2 -fast -WGstats
127	6033	B	-xcrossfile -fast -unroll=2 -WGstats
128	6033	B	-unroll=2 -xcrossfile -fast -WGstats
129	6033	B	-fast -WGstats
130	6033	B	No flags -WGstats
131	6337	A	-fast -unroll=2 -WGstats
132	6337	A	-unroll=2 -fast -xcrossfile -WGstats
133	6337	A	-xcrossfile -unroll=2 -fast -WGstats
<i>continued on next page</i>			

Table D.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
134	6337	A	-fast -unroll=2 -xcrossfile -WGstats
135	6337	A	No flags -WGstats
136	6337	A	-unroll=2 -WGstats
137	6337	A	-fast -WGstats
138	6337	A	-unroll=2 -xcrossfile -fast -WGstats
139	6337	A	-fast -xcrossfile -WGstats
140	6337	A	-xcrossfile -fast -unroll=2 -WGstats
141	6337	A	-xcrossfile -fast -WGstats
142	6337	A	-unroll=2 -fast -WGstats
143	6337	A	-fast -xcrossfile -unroll=2 -WGstats
144	6337	B	-fast -WGstats
145	6337	B	-fast -xcrossfile -unroll=2 -WGstats
146	6337	B	-unroll=2 -WGstats
147	6337	B	-xcrossfile -unroll=2 -fast -WGstats
148	6337	B	-xcrossfile -fast -WGstats
149	6337	B	-xcrossfile -fast -unroll=2 -WGstats
150	6337	B	-fast -unroll=2 -xcrossfile -WGstats
151	6337	B	-unroll=2 -fast -WGstats
152	6337	B	-fast -xcrossfile -WGstats
153	6337	B	-unroll=2 -fast -xcrossfile -WGstats
154	6337	B	No flags -WGstats
155	6337	B	-fast -unroll=2 -WGstats
156	6337	B	-unroll=2 -xcrossfile -fast -WGstats
157	6337	B	-fast -xcrossfile -unroll=2 -WGstats
158	6337	B	-xcrossfile -unroll=2 -fast -WGstats
<i>continued on next page</i>			

Table D.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
159	6337	B	-fast -xcrossfile -WGstats
160	6337	B	-xcrossfile -fast -WGstats
161	6337	B	-unroll=2 -xcrossfile -fast -WGstats
162	6337	B	-fast -unroll=2 -WGstats
163	6337	B	-unroll=2 -fast -WGstats
164	6337	B	No flags -WGstats
165	6337	B	-unroll=2 -WGstats
166	6337	B	-unroll=2 -fast -xcrossfile -WGstats
167	6337	B	-fast -unroll=2 -xcrossfile -WGstats
168	6337	B	-fast -WGstats
169	6337	B	-xcrossfile -fast -unroll=2 -WGstats
170	6337	A	-fast -unroll=2 -xcrossfile -WGstats
171	6337	A	No flags -WGstats
172	6337	A	-unroll=2 -WGstats
173	6337	A	-xcrossfile -fast -unroll=2 -WGstats
174	6337	A	-fast -xcrossfile -unroll=2 -WGstats
175	6337	A	-fast -unroll=2 -WGstats
176	6337	A	-xcrossfile -fast -WGstats
177	6337	A	-fast -WGstats
178	6337	A	-unroll=2 -fast -xcrossfile -WGstats
179	6337	A	-fast -xcrossfile -WGstats
180	6337	A	-unroll=2 -xcrossfile -fast -WGstats
181	6337	A	-unroll=2 -fast -WGstats
182	6337	A	-xcrossfile -unroll=2 -fast -WGstats
183	6033	B	-xcrossfile -fast -unroll=2 -WGstats
<i>continued on next page</i>			

Table D.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
184	6033	B	-fast -xcrossfile -unroll=2 -WGstats
185	6033	B	-fast -WGstats
186	6033	B	-unroll=2 -xcrossfile -fast -WGstats
187	6033	B	-unroll=2 -fast -WGstats
188	6033	B	-unroll=2 -WGstats
189	6033	B	-fast -xcrossfile -WGstats
190	6033	B	No flags -WGstats
191	6033	B	-xcrossfile -fast -WGstats
192	6033	B	-unroll=2 -fast -xcrossfile -WGstats
193	6033	B	-xcrossfile -unroll=2 -fast -WGstats
194	6033	B	-fast -unroll=2 -WGstats
195	6033	B	-fast -unroll=2 -xcrossfile -WGstats
196	6033	A	-xcrossfile -fast -unroll=2 -WGstats
197	6033	A	-fast -WGstats
198	6033	A	-xcrossfile -fast -WGstats
199	6033	A	-fast -xcrossfile -unroll=2 -WGstats
200	6033	A	-fast -unroll=2 -xcrossfile -WGstats
201	6033	A	-unroll=2 -fast -xcrossfile -WGstats
202	6033	A	-xcrossfile -unroll=2 -fast -WGstats
203	6033	A	-unroll=2 -fast -WGstats
204	6033	A	No flags -WGstats
205	6033	A	-fast -xcrossfile -WGstats
206	6033	A	-unroll=2 -WGstats
207	6033	A	-unroll=2 -xcrossfile -fast -WGstats
208	6033	A	-fast -unroll=2 -WGstats
<i>continued on next page</i>			

Table D.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
209	13857	B	-xcrossfile -unroll=2 -fast -WGstats
210	13857	B	-fast -xcrossfile -unroll=2 -WGstats
211	13857	B	-fast -unroll=2 -xcrossfile -WGstats
212	13857	B	-unroll=2 -fast -WGstats
213	13857	B	-fast -WGstats
214	13857	B	-fast -xcrossfile -WGstats
215	13857	B	-xcrossfile -fast -unroll=2 -WGstats
216	13857	B	-xcrossfile -fast -WGstats
217	13857	B	-unroll=2 -xcrossfile -fast -WGstats
218	13857	B	-unroll=2 -fast -xcrossfile -WGstats
219	13857	B	No flags -WGstats
220	13857	B	-fast -unroll=2 -WGstats
221	13857	B	-unroll=2 -WGstats
222	13857	A	-unroll=2 -fast -xcrossfile -WGstats
223	13857	A	-fast -unroll=2 -xcrossfile -WGstats
224	13857	A	-fast -WGstats
225	13857	A	-xcrossfile -unroll=2 -fast -WGstats
226	13857	A	-xcrossfile -fast -unroll=2 -WGstats
227	13857	A	-fast -xcrossfile -unroll=2 -WGstats
228	13857	A	-fast -unroll=2 -WGstats
229	13857	A	-unroll=2 -WGstats
230	13857	A	-unroll=2 -fast -WGstats
231	13857	A	-fast -xcrossfile -WGstats
232	13857	A	No flags -WGstats
233	13857	A	-xcrossfile -fast -WGstats
<i>continued on next page</i>			

Table D.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
234	13857	A	-unroll=2 -xcrossfile -fast -WGstats

D.2 Anova on the metrics obtained in Experiment 1

Table D.2: ANOVA

Label	Name	Problem	Algorithm	Compiler	Interactions			
		Size (S)	(A)	Option (C)	$S * A$	$S * C$	$A * C$	$S * A * C$
m0	Exec. Time	Yes	Yes	Yes	No	No	Yes	No
m1	bread/s	No	No	No	No	No	No	No
m2	lread/s	No	Yes	Yes	No	No	No	No
m3	%rcache	No	No	No	No	No	No	No
m4	bwrit/s	No	Yes	Yes	No	Yes	No	No
m5	lwrit/s	No	Yes	Yes	No	No	No	No
m6	%wcache	No	No	Yes	No	No	No	No
m7	pgout/s	No	No	Yes	No	Yes	No	No
m8	ppgout/s	No	No	Yes	No	Yes	No	No
m9	pgfree/s	No	No	No	No	No	No	No
m10	pgscan/s	No	No	No	No	No	No	No
m11	atch/s	No	No	No	No	No	No	No
m12	pgin/s	No	No	No	No	No	No	No
m13	ppgin/s	No	No	No	No	No	No	No
m14	pflt/s	No	No	No	No	Yes	No	No
m15	vflt/s	Yes	Yes	Yes	No	Yes	No	Yes
m16	%usr	No	No	Yes	Yes	No	Yes	No
<i>continued on next page</i>								

Table D.2: (cont'd).

Label	Name	Problem Size (S)	Algorithm (A)	Compiler Option (C)	Interactions			
					$S * A$	$S * C$	$A * C$	$S * A * C$
m17	%sys	Yes	Yes	Yes	No	No	Yes	No
m18	%wio	No	No	No	No	No	No	No
m19	%idle	No	Yes	Yes	Yes	No	Yes	No
m20	pswch/s	Yes	Yes	Yes	No	No	Yes	No
m21	c0t0d0/rps	No	No	No	No	No	No	No
m22	c0t0d0/wps	No	Yes	Yes	No	Yes	No	No
m23	c0t0d0/util	No	Yes	Yes	No	No	No	No
m24	c0t1d0/rps	No	No	No	No	No	No	No
m25	c0t1d0/wps	No	No	No	No	No	No	No
m26	c0t1d0/util	No	No	No	No	No	No	No
m27	cpu/us	No	No	Yes	Yes	No	Yes	No
m28	cpu/sy	Yes	Yes	Yes	No	No	Yes	No
m29	cpu/wt	No	No	No	No	No	No	No
m30	cpu/id	No	Yes	Yes	Yes	No	Yes	No
m31	memory/swap	No	No	Yes	No	No	No	No
m32	memory/free	No	No	Yes	No	No	No	No
m33	page/re	No	No	No	No	No	No	No
m34	page/mf	Yes	Yes	Yes	No	Yes	No	Yes
m35	page/pi	No	No	No	No	No	No	No
m36	page/po	No	No	Yes	No	Yes	No	No
m37	page/fr	No	No	No	No	No	No	No
m38	page/sr	No	No	No	No	No	No	No
m39	disk/s0	Yes	Yes	Yes	No	Yes	No	No
<i>continued on next page</i>								

Table D.2: (cont'd).

Label	Name	Problem	Algorithm	Compiler	Interactions			
		Size (S)	(A)	Option (C)	$S * A$	$S * C$	$A * C$	$S * A * C$
m40	disk/s1	No	No	No	No	No	No	No
m41	faults/in	No	No	Yes	No	No	No	No
m42	faults/sy	Yes	Yes	Yes	No	No	Yes	No
m43	faults/cs	Yes	Yes	Yes	No	No	Yes	No
m44	cpu/us_1	No	No	Yes	Yes	No	Yes	No
m45	cpu/sy_1	Yes	Yes	Yes	No	No	Yes	No
m46	cpu/id-1	No	Yes	Yes	Yes	No	Yes	No

Yes implies the hypothesis is rejected at alpha level 0.05.

APPENDIX E

Experiment 2

E.1 Order of Execution of Experimental Runs for Experiment 2

This is a split-split plot design. The main plot was the repetition, where three repetitions were done. The subplots were selected at random where problem size and matrix-vector multiplication algorithm was selected. In each of these subplots, compiler options for generating the executable files were selected at random using an uniform distribution random number generator. The following table contains the actual order in which the experimental runs were performed following the randomization scheme described above. There were a total of 234 experimental runs in this experiment.

Table E.1. Order of execution of experiments

Experimental Run	Size (N)	Algorithm	Compiler Options
1	6337	D	-fast -unroll=2 -WGstats
2	6337	D	-unroll=2 -fast -xcrossfile -WGstats
3	6337	D	-xcrossfile -fast -unroll=2 -WGstats
4	6337	D	-xcrossfile -fast -WGstats
5	6337	D	-unroll=2 -WGstats
6	6337	D	-xcrossfile -unroll=2 -fast -WGstats
7	6337	D	-fast -unroll=2 -xcrossfile -WGstats
<i>continued on next page</i>			

Table E.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
8	6337	D	-fast -xcrossfile -unroll=2 -WGstats
9	6337	D	-unroll=2 -xcrossfile -fast -WGstats
10	6337	D	No flags -WGstats
11	6337	D	-fast -xcrossfile -WGstats
12	6337	D	-unroll=2 -fast -WGstats
13	6337	D	-fast -WGstats
14	6337	E	-xcrossfile -fast -unroll=2 -WGstats
15	6337	E	No flags -WGstats
16	6337	E	-xcrossfile -fast -WGstats
17	6337	E	-unroll=2 -WGstats
18	6337	E	-fast -xcrossfile -unroll=2 -WGstats
19	6337	E	-unroll=2 -fast -xcrossfile -WGstats
20	6337	E	-xcrossfile -unroll=2 -fast -WGstats
21	6337	E	-unroll=2 -fast -WGstats
22	6337	E	-fast -WGstats
23	6337	E	-fast -unroll=2 -WGstats
24	6337	E	-fast -xcrossfile -WGstats
25	6337	E	-fast -unroll=2 -xcrossfile -WGstats
26	6337	E	-unroll=2 -xcrossfile -fast -WGstats
27	6033	D	-xcrossfile -fast -unroll=2 -WGstats
28	6033	D	-unroll=2 -fast -WGstats
29	6033	D	-unroll=2 -xcrossfile -fast -WGstats
30	6033	D	-fast -WGstats
31	6033	D	-xcrossfile -unroll=2 -fast -WGstats
32	6033	D	No flags -WGstats
<i>continued on next page</i>			

Table E.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
33	6033	D	-xcrossfile -fast -WGstats
34	6033	D	-fast -unroll=2 -WGstats
35	6033	D	-unroll=2 -WGstats
36	6033	D	-fast -xcrossfile -unroll=2 -WGstats
37	6033	D	-fast -xcrossfile -WGstats
38	6033	D	-unroll=2 -fast -xcrossfile -WGstats
39	6033	D	-fast -unroll=2 -xcrossfile -WGstats
40	6033	E	-unroll=2 -xcrossfile -fast -WGstats
41	6033	E	-unroll=2 -fast -xcrossfile -WGstats
42	6033	E	-unroll=2 -fast -WGstats
43	6033	E	-xcrossfile -fast -WGstats
44	6033	E	-fast -WGstats
45	6033	E	-xcrossfile -unroll=2 -fast -WGstats
46	6033	E	-xcrossfile -fast -unroll=2 -WGstats
47	6033	E	No flags -WGstats
48	6033	E	-fast -xcrossfile -unroll=2 -WGstats
49	6033	E	-unroll=2 -WGstats
50	6033	E	-fast -unroll=2 -xcrossfile -WGstats
51	6033	E	-fast -xcrossfile -WGstats
52	6033	E	-fast -unroll=2 -WGstats
53	13857	D	-unroll=2 -fast -xcrossfile -WGstats
54	13857	D	-fast -xcrossfile -unroll=2 -WGstats
55	13857	D	No flags -WGstats
56	13857	D	-xcrossfile -fast -WGstats
57	13857	D	-xcrossfile -fast -unroll=2 -WGstats
<i>continued on next page</i>			

Table E.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
58	13857	D	-fast -unroll=2 -xcrossfile -WGstats
59	13857	D	-xcrossfile -unroll=2 -fast -WGstats
60	13857	D	-fast -WGstats
61	13857	D	-unroll=2 -xcrossfile -fast -WGstats
62	13857	D	-fast -xcrossfile -WGstats
63	13857	D	-fast -unroll=2 -WGstats
64	13857	D	-unroll=2 -WGstats
65	13857	D	-unroll=2 -fast -WGstats
66	13857	E	-unroll=2 -fast -WGstats
67	13857	E	-fast -xcrossfile -WGstats
68	13857	E	-fast -unroll=2 -WGstats
69	13857	E	-xcrossfile -fast -WGstats
70	13857	E	-unroll=2 -WGstats
71	13857	E	-fast -WGstats
72	13857	E	-fast -unroll=2 -xcrossfile -WGstats
73	13857	E	No flags -WGstats
74	13857	E	-xcrossfile -unroll=2 -fast -WGstats
75	13857	E	-xcrossfile -fast -unroll=2 -WGstats
76	13857	E	-unroll=2 -xcrossfile -fast -WGstats
77	13857	E	-fast -xcrossfile -unroll=2 -WGstats
78	13857	E	-unroll=2 -fast -xcrossfile -WGstats
79	6337	D	-unroll=2 -xcrossfile -fast -WGstats
80	6337	D	-fast -unroll=2 -xcrossfile -WGstats
81	6337	D	-xcrossfile -unroll=2 -fast -WGstats
82	6337	D	-fast -xcrossfile -unroll=2 -WGstats
<i>continued on next page</i>			

Table E.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
83	6337	D	-fast -xcrossfile -WGstats
84	6337	D	-unroll=2 -WGstats
85	6337	D	-unroll=2 -fast -xcrossfile -WGstats
86	6337	D	No flags -WGstats
87	6337	D	-fast -WGstats
88	6337	D	-xcrossfile -fast -unroll=2 -WGstats
89	6337	D	-xcrossfile -fast -WGstats
90	6337	D	-unroll=2 -fast -WGstats
91	6337	D	-fast -unroll=2 -WGstats
92	6337	E	-fast -unroll=2 -WGstats
93	6337	E	No flags -WGstats
94	6337	E	-unroll=2 -fast -xcrossfile -WGstats
95	6337	E	-fast -unroll=2 -xcrossfile -WGstats
96	6337	E	-xcrossfile -fast -unroll=2 -WGstats
97	6337	E	-xcrossfile -fast -WGstats
98	6337	E	-unroll=2 -WGstats
99	6337	E	-fast -xcrossfile -WGstats
100	6337	E	-fast -xcrossfile -unroll=2 -WGstats
101	6337	E	-unroll=2 -fast -WGstats
102	6337	E	-fast -WGstats
103	6337	E	-unroll=2 -xcrossfile -fast -WGstats
104	6337	E	-xcrossfile -unroll=2 -fast -WGstats
105	13857	D	-fast -unroll=2 -xcrossfile -WGstats
106	13857	D	-unroll=2 -WGstats
107	13857	D	-fast -xcrossfile -unroll=2 -WGstats
<i>continued on next page</i>			

Table E.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
108	13857	D	-fast -xcrossfile -WGstats
109	13857	D	-unroll=2 -fast -WGstats
110	13857	D	-fast -unroll=2 -WGstats
111	13857	D	-unroll=2 -xcrossfile -fast -WGstats
112	13857	D	-xcrossfile -fast -unroll=2 -WGstats
113	13857	D	No flags -WGstats
114	13857	D	-xcrossfile -unroll=2 -fast -WGstats
115	13857	D	-fast -WGstats
116	13857	D	-xcrossfile -fast -WGstats
117	13857	D	-unroll=2 -fast -xcrossfile -WGstats
118	13857	E	-fast -unroll=2 -xcrossfile -WGstats
119	13857	E	-unroll=2 -WGstats
120	13857	E	-fast -unroll=2 -WGstats
121	13857	E	No flags -WGstats
122	13857	E	-fast -xcrossfile -WGstats
123	13857	E	-xcrossfile -fast -WGstats
124	13857	E	-unroll=2 -fast -xcrossfile -WGstats
125	13857	E	-unroll=2 -xcrossfile -fast -WGstats
126	13857	E	-xcrossfile -fast -unroll=2 -WGstats
127	13857	E	-fast -WGstats
128	13857	E	-fast -xcrossfile -unroll=2 -WGstats
129	13857	E	-unroll=2 -fast -WGstats
130	13857	E	-xcrossfile -unroll=2 -fast -WGstats
131	6033	E	-xcrossfile -fast -WGstats
132	6033	E	-fast -unroll=2 -WGstats
<i>continued on next page</i>			

Table E.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
133	6033	E	-unroll=2 -fast -xcrossfile -WGstats
134	6033	E	-xcrossfile -unroll=2 -fast -WGstats
135	6033	E	No flags -WGstats
136	6033	E	-fast -WGstats
137	6033	E	-unroll=2 -fast -WGstats
138	6033	E	-unroll=2 -WGstats
139	6033	E	-unroll=2 -xcrossfile -fast -WGstats
140	6033	E	-fast -xcrossfile -unroll=2 -WGstats
141	6033	E	-fast -unroll=2 -xcrossfile -WGstats
142	6033	E	-fast -xcrossfile -WGstats
143	6033	E	-xcrossfile -fast -unroll=2 -WGstats
144	6033	D	-fast -unroll=2 -xcrossfile -WGstats
145	6033	D	-unroll=2 -xcrossfile -fast -WGstats
146	6033	D	-fast -xcrossfile -unroll=2 -WGstats
147	6033	D	No flags -WGstats
148	6033	D	-xcrossfile -fast -WGstats
149	6033	D	-unroll=2 -WGstats
150	6033	D	-xcrossfile -fast -unroll=2 -WGstats
151	6033	D	-xcrossfile -unroll=2 -fast -WGstats
152	6033	D	-fast -xcrossfile -WGstats
153	6033	D	-fast -unroll=2 -WGstats
154	6033	D	-unroll=2 -fast -xcrossfile -WGstats
155	6033	D	-fast -WGstats
156	6033	D	-unroll=2 -fast -WGstats
157	13857	D	-unroll=2 -WGstats
<i>continued on next page</i>			

Table E.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
158	13857	D	-unroll=2 -fast -WGstats
159	13857	D	-fast -xcrossfile -unroll=2 -WGstats
160	13857	D	-fast -unroll=2 -xcrossfile -WGstats
161	13857	D	-unroll=2 -xcrossfile -fast -WGstats
162	13857	D	-xcrossfile -unroll=2 -fast -WGstats
163	13857	D	-xcrossfile -fast -unroll=2 -WGstats
164	13857	D	-xcrossfile -fast -WGstats
165	13857	D	-fast -unroll=2 -WGstats
166	13857	D	-fast -WGstats
167	13857	D	No flags -WGstats
168	13857	D	-unroll=2 -fast -xcrossfile -WGstats
169	13857	D	-fast -xcrossfile -WGstats
170	13857	E	-fast -unroll=2 -WGstats
171	13857	E	-xcrossfile -fast -WGstats
172	13857	E	No flags -WGstats
173	13857	E	-xcrossfile -fast -unroll=2 -WGstats
174	13857	E	-unroll=2 -WGstats
175	13857	E	-fast -xcrossfile -WGstats
176	13857	E	-unroll=2 -fast -xcrossfile -WGstats
177	13857	E	-fast -WGstats
178	13857	E	-unroll=2 -fast -WGstats
179	13857	E	-unroll=2 -xcrossfile -fast -WGstats
180	13857	E	-fast -xcrossfile -unroll=2 -WGstats
181	13857	E	-fast -unroll=2 -xcrossfile -WGstats
182	13857	E	-xcrossfile -unroll=2 -fast -WGstats
<i>continued on next page</i>			

Table E.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
183	6033	E	-fast -unroll=2 -WGstats
184	6033	E	-unroll=2 -fast -xcrossfile -WGstats
185	6033	E	-unroll=2 -xcrossfile -fast -WGstats
186	6033	E	No flags -WGstats
187	6033	E	-xcrossfile -fast -WGstats
188	6033	E	-fast -xcrossfile -WGstats
189	6033	E	-unroll=2 -fast -WGstats
190	6033	E	-unroll=2 -WGstats
191	6033	E	-xcrossfile -unroll=2 -fast -WGstats
192	6033	E	-fast -unroll=2 -xcrossfile -WGstats
193	6033	E	-fast -xcrossfile -unroll=2 -WGstats
194	6033	E	-fast -WGstats
195	6033	E	-xcrossfile -fast -unroll=2 -WGstats
196	6033	D	-unroll=2 -WGstats
197	6033	D	-fast -xcrossfile -unroll=2 -WGstats
198	6033	D	-xcrossfile -fast -WGstats
199	6033	D	-fast -WGstats
200	6033	D	-fast -unroll=2 -WGstats
201	6033	D	-unroll=2 -fast -WGstats
202	6033	D	-xcrossfile -fast -unroll=2 -WGstats
203	6033	D	-fast -unroll=2 -xcrossfile -WGstats
204	6033	D	-xcrossfile -unroll=2 -fast -WGstats
205	6033	D	-unroll=2 -xcrossfile -fast -WGstats
206	6033	D	No flags -WGstats
207	6033	D	-unroll=2 -fast -xcrossfile -WGstats
<i>continued on next page</i>			

Table E.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
208	6033	D	-fast -xcrossfile -WGstats
209	6337	D	-fast -xcrossfile -WGstats
210	6337	D	-unroll=2 -WGstats
211	6337	D	-xcrossfile -fast -unroll=2 -WGstats
212	6337	D	No flags -WGstats
213	6337	D	-unroll=2 -fast -xcrossfile -WGstats
214	6337	D	-fast -unroll=2 -xcrossfile -WGstats
215	6337	D	-unroll=2 -fast -WGstats
216	6337	D	-xcrossfile -fast -WGstats
217	6337	D	-unroll=2 -xcrossfile -fast -WGstats
218	6337	D	-fast -WGstats
219	6337	D	-fast -xcrossfile -unroll=2 -WGstats
220	6337	D	-fast -unroll=2 -WGstats
221	6337	D	-xcrossfile -unroll=2 -fast -WGstats
222	6337	E	-unroll=2 -fast -WGstats
223	6337	E	-fast -xcrossfile -unroll=2 -WGstats
224	6337	E	-unroll=2 -fast -xcrossfile -WGstats
225	6337	E	-unroll=2 -WGstats
226	6337	E	-fast -xcrossfile -WGstats
227	6337	E	-xcrossfile -fast -unroll=2 -WGstats
228	6337	E	-xcrossfile -unroll=2 -fast -WGstats
229	6337	E	-fast -unroll=2 -WGstats
230	6337	E	-xcrossfile -fast -WGstats
231	6337	E	No flags -WGstats
232	6337	E	-fast -unroll=2 -xcrossfile -WGstats
<i>continued on next page</i>			

Table E.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
233	6337	E	-unroll=2 -xcrossfile -fast -WGstats
234	6337	E	-fast -WGstats

E.2 Anova on the metrics obtained in Experiment 2

Table E.2: ANOVA

Label	Name	Problem	Algorithm	Compiler	Interactions			
		Size (S)	(A)	Option (C)	$S * A$	$S * C$	$A * C$	$S * A * C$
p0	execution time	Yes	Yes	Yes	No	Yes	Yes	No
p1	bread/s	No	No	No	No	No	No	No
p2	lread/s	No	Yes	Yes	No	No	Yes	No
p3	%rcache	No	Yes	No	Yes	No	No	No
p4	bwrit/s	Yes	No	Yes	No	No	No	No
p5	lwrit/s	No	Yes	Yes	No	No	Yes	No
p6	%wcache	Yes	No	No	No	No	No	No
p7	pgout/s	No	Yes	No	No	No	No	No
p8	ppgout/s	No	No	No	No	No	No	No
p9	pgfree/s	No	No	No	No	No	No	No
p10	pgscan/s	No	No	No	No	No	No	No
p11	atch/s	No	No	Yes	No	No	No	No
p12	pgin/s	No	No	No	No	No	No	No
p13	ppgin/s	No	No	No	No	No	No	No
p14	pft/s	No	No	No	No	No	No	No
p15	vft/s	Yes	Yes	Yes	No	No	No	No
<i>continued on next page</i>								

Table E.2: (cont'd).

Label	Name	Problem	Algorithm	Compiler	Interactions			
		Size (S)	(A)	Option (C)	$S * A$	$S * C$	$A * C$	$S * A * C$
p16	%usr	No	No	Yes	No	No	No	No
p17	%sys	No	Yes	Yes	No	No	No	No
p18	%wio	No	No	No	No	No	No	No
p19	%idle	No	No	No	No	No	No	No
p20	pswch/s	No	No	No	No	No	No	Yes
p21	c0t0d0/rps	No	No	No	No	No	No	No
p22	c0t0d0/wps	No	No	No	No	No	No	No
p23	c0t0d0/util	No	Yes	Yes	No	No	Yes	No
p24	cpu/us	No	No	No	No	No	No	No
p25	cpu/sy	No	No	No	No	No	No	No
p26	cpu/wt	No	Yes	Yes	No	No	No	No
p27	cpu/id	No	Yes	Yes	No	No	No	No
p28	memory/swap	No	No	No	No	No	No	No
p29	memory/free	No	No	Yes	No	No	No	No
p30	page/re	No	No	Yes	No	No	No	No
p31	page/mf	No	No	No	No	No	No	No
p32	page/pi	Yes	Yes	Yes	No	No	No	No
p33	page/po	No	No	No	No	No	No	No
p34	page/fr	No	No	No	No	No	No	No
p35	page/sr	No	No	No	No	No	No	No
p36	disk/s0	No	No	No	No	No	No	No
p37	faults/in	No	Yes	Yes	No	No	Yes	No
p38	faults/sy	No	Yes	Yes	No	No	No	No
<i>continued on next page</i>								

Table E.2: (cont'd).

Label	Name	Problem Size (S)	Algorithm (A)	Compiler Option (C)	Interactions			
					$S * A$	$S * C$	$A * C$	$S * A * C$
p39	faults/cs	No	No	No	No	No	No	No
p40	cpu/us_1	No	No	No	No	No	No	No
p41	cpu/sy_1	No	No	No	No	No	No	No
p42	cpu/id-1	No	Yes	Yes	No	No	No	No

Yes implies the hypothesis is rejected at alpha level 0.05.

APPENDIX F

Experiment 3

F.1 Order of Execution of Experimental Runs for Experiment 3

This is a split-split plot design. The main plot was the repetition, where three repetitions were done. The subplots were selected at random where problem size and matrix-vector multiplication algorithm was selected. In each of these subplots, compiler options for generating the executable files were selected at random using an uniform distribution random number generator. The following table contains the actual order in which the experimental runs were performed following the randomization scheme described above. There were a total of 351 experimental runs in this experiment.

Table F.1. Order of execution of experiments

Experimental Run	Size (N)	Algorithm	Compiler Options
1	6033	B	-fast -WGstats
2	6033	B	-unroll=2 -fast -xcrossfile -WGstats
3	6033	B	No flags -WGstats
4	6033	B	-xcrossfile -fast -WGstats
5	6033	B	-fast -xcrossfile -WGstats
6	6033	B	-fast -xcrossfile -unroll=2 -WGstats
7	6033	B	-xcrossfile -unroll=2 -fast -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
8	6033	B	-unroll=2 -fast -WGstats
9	6033	B	-unroll=2 -WGstats
10	6033	B	-unroll=2 -xcrossfile -fast -WGstats
11	6033	B	-fast -unroll=2 -WGstats
12	6033	B	-xcrossfile -fast -unroll=2 -WGstats
13	6033	B	-fast -unroll=2 -xcrossfile -WGstats
14	6033	A	-fast -unroll=2 -WGstats
15	6033	A	-unroll=2 -fast -xcrossfile -WGstats
16	6033	A	-unroll=2 -xcrossfile -fast -WGstats
17	6033	A	-xcrossfile -fast -WGstats
18	6033	A	-xcrossfile -fast -unroll=2 -WGstats
19	6033	A	-fast -xcrossfile -WGstats
20	6033	A	-fast -xcrossfile -unroll=2 -WGstats
21	6033	A	-unroll=2 -WGstats
22	6033	A	No flags -WGstats
23	6033	A	-fast -unroll=2 -xcrossfile -WGstats
24	6033	A	-unroll=2 -fast -WGstats
25	6033	A	-xcrossfile -unroll=2 -fast -WGstats
26	6033	A	-fast -WGstats
27	13857	B	No flags -WGstats
28	13857	B	-unroll=2 -xcrossfile -fast -WGstats
29	13857	B	-fast -unroll=2 -WGstats
30	13857	B	-fast -WGstats
31	13857	B	-fast -xcrossfile -unroll=2 -WGstats
32	13857	B	-xcrossfile -fast -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
33	13857	B	-xcrossfile -unroll=2 -fast -WGstats
34	13857	B	-fast -xcrossfile -WGstats
35	13857	B	-unroll=2 -fast -WGstats
36	13857	B	-xcrossfile -fast -unroll=2 -WGstats
37	13857	B	-fast -unroll=2 -xcrossfile -WGstats
38	13857	B	-unroll=2 -fast -xcrossfile -WGstats
39	13857	B	-unroll=2 -WGstats
40	13857	A	-unroll=2 -fast -xcrossfile -WGstats
41	13857	A	-unroll=2 -WGstats
42	13857	A	-xcrossfile -unroll=2 -fast -WGstats
43	13857	A	-fast -unroll=2 -xcrossfile -WGstats
44	13857	A	-fast -xcrossfile -unroll=2 -WGstats
45	13857	A	-fast -xcrossfile -WGstats
46	13857	A	-xcrossfile -fast -unroll=2 -WGstats
47	13857	A	-xcrossfile -fast -WGstats
48	13857	A	-fast -WGstats
49	13857	A	-unroll=2 -fast -WGstats
50	13857	A	No flags -WGstats
51	13857	A	-fast -unroll=2 -WGstats
52	13857	A	-unroll=2 -xcrossfile -fast -WGstats
53	6337	B	-unroll=2 -WGstats
54	6337	B	-fast -unroll=2 -WGstats
55	6337	B	-unroll=2 -xcrossfile -fast -WGstats
56	6337	B	-fast -xcrossfile -unroll=2 -WGstats
57	6337	B	No flags -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
58	6337	B	-fast -xcrossfile -WGstats
59	6337	B	-xcrossfile -unroll=2 -fast -WGstats
60	6337	B	-unroll=2 -fast -xcrossfile -WGstats
61	6337	B	-unroll=2 -fast -WGstats
62	6337	B	-fast -WGstats
63	6337	B	-xcrossfile -fast -WGstats
64	6337	B	-xcrossfile -fast -unroll=2 -WGstats
65	6337	B	-fast -unroll=2 -xcrossfile -WGstats
66	6337	A	-fast -unroll=2 -xcrossfile -WGstats
67	6337	A	-xcrossfile -unroll=2 -fast -WGstats
68	6337	A	-fast -WGstats
69	6337	A	-unroll=2 -xcrossfile -fast -WGstats
70	6337	A	-xcrossfile -fast -WGstats
71	6337	A	-fast -unroll=2 -WGstats
72	6337	A	-fast -xcrossfile -unroll=2 -WGstats
73	6337	A	-unroll=2 -fast -WGstats
74	6337	A	-unroll=2 -WGstats
75	6337	A	-unroll=2 -fast -xcrossfile -WGstats
76	6337	A	-fast -xcrossfile -WGstats
77	6337	A	No flags -WGstats
78	6337	A	-xcrossfile -fast -unroll=2 -WGstats
79	13857	A	No flags -WGstats
80	13857	A	-fast -unroll=2 -WGstats
81	13857	A	-xcrossfile -unroll=2 -fast -WGstats
82	13857	A	-fast -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
83	13857	A	-xcrossfile -fast -unroll=2 -WGstats
84	13857	A	-unroll=2 -fast -WGstats
85	13857	A	-unroll=2 -WGstats
86	13857	A	-unroll=2 -xcrossfile -fast -WGstats
87	13857	A	-fast -unroll=2 -xcrossfile -WGstats
88	13857	A	-xcrossfile -fast -WGstats
89	13857	A	-fast -xcrossfile -WGstats
90	13857	A	-unroll=2 -fast -xcrossfile -WGstats
91	13857	A	-fast -xcrossfile -unroll=2 -WGstats
92	13857	B	-xcrossfile -fast -unroll=2 -WGstats
93	13857	B	-xcrossfile -fast -WGstats
94	13857	B	-unroll=2 -WGstats
95	13857	B	-fast -unroll=2 -xcrossfile -WGstats
96	13857	B	-fast -xcrossfile -WGstats
97	13857	B	-fast -unroll=2 -WGstats
98	13857	B	-unroll=2 -fast -WGstats
99	13857	B	-unroll=2 -fast -xcrossfile -WGstats
100	13857	B	-fast -WGstats
101	13857	B	-fast -xcrossfile -unroll=2 -WGstats
102	13857	B	-xcrossfile -unroll=2 -fast -WGstats
103	13857	B	-unroll=2 -xcrossfile -fast -WGstats
104	13857	B	No flags -WGstats
105	6033	A	-unroll=2 -WGstats
106	6033	A	-fast -xcrossfile -WGstats
107	6033	A	-unroll=2 -fast -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
108	6033	A	-fast -unroll=2 -WGstats
109	6033	A	-xcrossfile -unroll=2 -fast -WGstats
110	6033	A	-xcrossfile -fast -unroll=2 -WGstats
111	6033	A	-unroll=2 -xcrossfile -fast -WGstats
112	6033	A	-fast -xcrossfile -unroll=2 -WGstats
113	6033	A	-fast -WGstats
114	6033	A	-fast -unroll=2 -xcrossfile -WGstats
115	6033	A	-unroll=2 -fast -xcrossfile -WGstats
116	6033	A	No flags -WGstats
117	6033	A	-xcrossfile -fast -WGstats
118	6033	B	-xcrossfile -fast -WGstats
119	6033	B	-fast -xcrossfile -WGstats
120	6033	B	-xcrossfile -unroll=2 -fast -WGstats
121	6033	B	-fast -xcrossfile -unroll=2 -WGstats
122	6033	B	-fast -unroll=2 -xcrossfile -WGstats
123	6033	B	-unroll=2 -fast -xcrossfile -WGstats
124	6033	B	-unroll=2 -WGstats
125	6033	B	-fast -unroll=2 -WGstats
126	6033	B	-unroll=2 -fast -WGstats
127	6033	B	-xcrossfile -fast -unroll=2 -WGstats
128	6033	B	-unroll=2 -xcrossfile -fast -WGstats
129	6033	B	-fast -WGstats
130	6033	B	No flags -WGstats
131	6337	A	-fast -unroll=2 -WGstats
132	6337	A	-unroll=2 -fast -xcrossfile -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
133	6337	A	-xcrossfile -unroll=2 -fast -WGstats
134	6337	A	-fast -unroll=2 -xcrossfile -WGstats
135	6337	A	No flags -WGstats
136	6337	A	-unroll=2 -WGstats
137	6337	A	-fast -WGstats
138	6337	A	-unroll=2 -xcrossfile -fast -WGstats
139	6337	A	-fast -xcrossfile -WGstats
140	6337	A	-xcrossfile -fast -unroll=2 -WGstats
141	6337	A	-xcrossfile -fast -WGstats
142	6337	A	-unroll=2 -fast -WGstats
143	6337	A	-fast -xcrossfile -unroll=2 -WGstats
144	6337	B	-fast -WGstats
145	6337	B	-fast -xcrossfile -unroll=2 -WGstats
146	6337	B	-unroll=2 -WGstats
147	6337	B	-xcrossfile -unroll=2 -fast -WGstats
148	6337	B	-xcrossfile -fast -WGstats
149	6337	B	-xcrossfile -fast -unroll=2 -WGstats
150	6337	B	-fast -unroll=2 -xcrossfile -WGstats
151	6337	B	-unroll=2 -fast -WGstats
152	6337	B	-fast -xcrossfile -WGstats
153	6337	B	-unroll=2 -fast -xcrossfile -WGstats
154	6337	B	No flags -WGstats
155	6337	B	-fast -unroll=2 -WGstats
156	6337	B	-unroll=2 -xcrossfile -fast -WGstats
157	6337	B	-fast -xcrossfile -unroll=2 -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
158	6337	B	-xcrossfile -unroll=2 -fast -WGstats
159	6337	B	-fast -xcrossfile -WGstats
160	6337	B	-xcrossfile -fast -WGstats
161	6337	B	-unroll=2 -xcrossfile -fast -WGstats
162	6337	B	-fast -unroll=2 -WGstats
163	6337	B	-unroll=2 -fast -WGstats
164	6337	B	No flags -WGstats
165	6337	B	-unroll=2 -WGstats
166	6337	B	-unroll=2 -fast -xcrossfile -WGstats
167	6337	B	-fast -unroll=2 -xcrossfile -WGstats
168	6337	B	-fast -WGstats
169	6337	B	-xcrossfile -fast -unroll=2 -WGstats
170	6337	A	-fast -unroll=2 -xcrossfile -WGstats
171	6337	A	No flags -WGstats
172	6337	A	-unroll=2 -WGstats
173	6337	A	-xcrossfile -fast -unroll=2 -WGstats
174	6337	A	-fast -xcrossfile -unroll=2 -WGstats
175	6337	A	-fast -unroll=2 -WGstats
176	6337	A	-xcrossfile -fast -WGstats
177	6337	A	-fast -WGstats
178	6337	A	-unroll=2 -fast -xcrossfile -WGstats
179	6337	A	-fast -xcrossfile -WGstats
180	6337	A	-unroll=2 -xcrossfile -fast -WGstats
181	6337	A	-unroll=2 -fast -WGstats
182	6337	A	-xcrossfile -unroll=2 -fast -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
183	6033	B	-xcrossfile -fast -unroll=2 -WGstats
184	6033	B	-fast -xcrossfile -unroll=2 -WGstats
185	6033	B	-fast -WGstats
186	6033	B	-unroll=2 -xcrossfile -fast -WGstats
187	6033	B	-unroll=2 -fast -WGstats
188	6033	B	-unroll=2 -WGstats
189	6033	B	-fast -xcrossfile -WGstats
190	6033	B	No flags -WGstats
191	6033	B	-xcrossfile -fast -WGstats
192	6033	B	-unroll=2 -fast -xcrossfile -WGstats
193	6033	B	-xcrossfile -unroll=2 -fast -WGstats
194	6033	B	-fast -unroll=2 -WGstats
195	6033	B	-fast -unroll=2 -xcrossfile -WGstats
196	6033	A	-xcrossfile -fast -unroll=2 -WGstats
197	6033	A	-fast -WGstats
198	6033	A	-xcrossfile -fast -WGstats
199	6033	A	-fast -xcrossfile -unroll=2 -WGstats
200	6033	A	-fast -unroll=2 -xcrossfile -WGstats
201	6033	A	-unroll=2 -fast -xcrossfile -WGstats
202	6033	A	-xcrossfile -unroll=2 -fast -WGstats
203	6033	A	-unroll=2 -fast -WGstats
204	6033	A	No flags -WGstats
205	6033	A	-fast -xcrossfile -WGstats
206	6033	A	-unroll=2 -WGstats
207	6033	A	-unroll=2 -xcrossfile -fast -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
208	6033	A	-fast -unroll=2 -WGstats
209	13857	B	-xcrossfile -unroll=2 -fast -WGstats
210	13857	B	-fast -xcrossfile -unroll=2 -WGstats
211	13857	B	-fast -unroll=2 -xcrossfile -WGstats
212	13857	B	-unroll=2 -fast -WGstats
213	13857	B	-fast -WGstats
214	13857	B	-fast -xcrossfile -WGstats
215	13857	B	-xcrossfile -fast -unroll=2 -WGstats
216	13857	B	-xcrossfile -fast -WGstats
217	13857	B	-unroll=2 -xcrossfile -fast -WGstats
218	13857	B	-unroll=2 -fast -xcrossfile -WGstats
219	13857	B	No flags -WGstats
220	13857	B	-fast -unroll=2 -WGstats
221	13857	B	-unroll=2 -WGstats
222	13857	A	-unroll=2 -fast -xcrossfile -WGstats
223	13857	A	-fast -unroll=2 -xcrossfile -WGstats
224	13857	A	-fast -WGstats
225	13857	A	-xcrossfile -unroll=2 -fast -WGstats
226	13857	A	-xcrossfile -fast -unroll=2 -WGstats
227	13857	A	-fast -xcrossfile -unroll=2 -WGstats
228	13857	A	-fast -unroll=2 -WGstats
229	13857	A	-unroll=2 -WGstats
230	13857	A	-unroll=2 -fast -WGstats
231	13857	A	-fast -xcrossfile -WGstats
232	13857	A	No flags -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
233	13857	A	-xcrossfile -fast -WGstats
234	13857	A	-unroll=2 -xcrossfile -fast -WGstats
235	6337	C	-unroll=2 -xcrossfile -fast -WGstats
236	6337	C	-xcrossfile -fast -unroll=2 -WGstats
237	6337	C	-fast -xcrossfile -unroll=2 -WGstats
238	6337	C	-unroll=2 -fast -xcrossfile -WGstats
239	6337	C	-xcrossfile -unroll=2 -fast -WGstats
240	6337	C	-unroll=2 -fast -WGstats
241	6337	C	-fast -WGstats
242	6337	C	-fast -unroll=2 -xcrossfile -WGstats
243	6337	C	-fast -unroll=2 -WGstats
244	6337	C	-fast -xcrossfile -WGstats
245	6337	C	No flags -WGstats
246	6337	C	-xcrossfile -fast -WGstats
247	6337	C	-unroll=2 -WGstats
248	6033	C	No flags -WGstats
249	6033	C	-fast -WGstats
250	6033	C	-unroll=2 -fast -xcrossfile -WGstats
251	6033	C	-xcrossfile -fast -unroll=2 -WGstats
252	6033	C	-fast -unroll=2 -xcrossfile -WGstats
253	6033	C	-unroll=2 -fast -WGstats
254	6033	C	-xcrossfile -unroll=2 -fast -WGstats
255	6033	C	-unroll=2 -WGstats
256	6033	C	-xcrossfile -fast -WGstats
257	6033	C	-unroll=2 -xcrossfile -fast -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
258	6033	C	-fast -xcrossfile -WGstats
259	6033	C	-fast -unroll=2 -WGstats
260	6033	C	-fast -xcrossfile -unroll=2 -WGstats
261	13857	C	-unroll=2 -WGstats
262	13857	C	-fast -xcrossfile -WGstats
263	13857	C	No flags -WGstats
264	13857	C	-unroll=2 -xcrossfile -fast -WGstats
265	13857	C	-xcrossfile -fast -unroll=2 -WGstats
266	13857	C	-fast -unroll=2 -WGstats
267	13857	C	-xcrossfile -fast -WGstats
268	13857	C	-fast -xcrossfile -unroll=2 -WGstats
269	13857	C	-xcrossfile -unroll=2 -fast -WGstats
270	13857	C	-unroll=2 -fast -WGstats
271	13857	C	-fast -unroll=2 -xcrossfile -WGstats
272	13857	C	-fast -WGstats
273	13857	C	-unroll=2 -fast -xcrossfile -WGstats
274	6337	C	-unroll=2 -fast -WGstats
275	6337	C	-unroll=2 -xcrossfile -fast -WGstats
276	6337	C	-fast -unroll=2 -xcrossfile -WGstats
277	6337	C	-fast -xcrossfile -unroll=2 -WGstats
278	6337	C	-xcrossfile -fast -unroll=2 -WGstats
279	6337	C	-fast -WGstats
280	6337	C	-fast -unroll=2 -WGstats
281	6337	C	-xcrossfile -unroll=2 -fast -WGstats
282	6337	C	-xcrossfile -fast -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
283	6337	C	-unroll=2 -WGstats
284	6337	C	-unroll=2 -fast -xcrossfile -WGstats
285	6337	C	No flags -WGstats
286	6337	C	-fast -xcrossfile -WGstats
287	13857	C	-unroll=2 -fast -WGstats
288	13857	C	-fast -WGstats
289	13857	C	-fast -xcrossfile -WGstats
290	13857	C	-fast -xcrossfile -unroll=2 -WGstats
291	13857	C	-xcrossfile -fast -WGstats
292	13857	C	-unroll=2 -WGstats
293	13857	C	-xcrossfile -fast -unroll=2 -WGstats
294	13857	C	-fast -unroll=2 -WGstats
295	13857	C	-unroll=2 -fast -xcrossfile -WGstats
296	13857	C	-xcrossfile -unroll=2 -fast -WGstats
297	13857	C	-unroll=2 -xcrossfile -fast -WGstats
298	13857	C	No flags -WGstats
299	13857	C	-fast -unroll=2 -xcrossfile -WGstats
300	6033	C	-xcrossfile -fast -unroll=2 -WGstats
301	6033	C	No flags -WGstats
302	6033	C	-fast -xcrossfile -WGstats
303	6033	C	-fast -WGstats
304	6033	C	-xcrossfile -fast -WGstats
305	6033	C	-xcrossfile -unroll=2 -fast -WGstats
306	6033	C	-fast -unroll=2 -xcrossfile -WGstats
307	6033	C	-unroll=2 -fast -xcrossfile -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
308	6033	C	-fast -unroll=2 -WGstats
309	6033	C	-unroll=2 -WGstats
310	6033	C	-fast -xcrossfile -unroll=2 -WGstats
311	6033	C	-unroll=2 -fast -WGstats
312	6033	C	-unroll=2 -xcrossfile -fast -WGstats
313	6033	C	-fast -unroll=2 -xcrossfile -WGstats
314	6033	C	-xcrossfile -fast -WGstats
315	6033	C	-fast -WGstats
316	6033	C	-fast -unroll=2 -WGstats
317	6033	C	-unroll=2 -xcrossfile -fast -WGstats
318	6033	C	No flags -WGstats
319	6033	C	-unroll=2 -fast -WGstats
320	6033	C	-fast -xcrossfile -unroll=2 -WGstats
321	6033	C	-unroll=2 -fast -xcrossfile -WGstats
322	6033	C	-fast -xcrossfile -WGstats
323	6033	C	-xcrossfile -unroll=2 -fast -WGstats
324	6033	C	-xcrossfile -fast -unroll=2 -WGstats
325	6033	C	-unroll=2 -WGstats
326	6337	C	-fast -unroll=2 -WGstats
327	6337	C	-xcrossfile -fast -unroll=2 -WGstats
328	6337	C	-fast -xcrossfile -WGstats
329	6337	C	-xcrossfile -unroll=2 -fast -WGstats
330	6337	C	-unroll=2 -fast -WGstats
331	6337	C	No flags -WGstats
332	6337	C	-unroll=2 -WGstats
<i>continued on next page</i>			

Table F.1 (cont'd).

Experimental Run	Size (N)	Algorithm	Compiler Options
333	6337	C	-xcrossfile -fast -WGstats
334	6337	C	-fast -WGstats
335	6337	C	-unroll=2 -fast -xcrossfile -WGstats
336	6337	C	-unroll=2 -xcrossfile -fast -WGstats
337	6337	C	-fast -unroll=2 -xcrossfile -WGstats
338	6337	C	-fast -xcrossfile -unroll=2 -WGstats
339	13857	C	-unroll=2 -xcrossfile -fast -WGstats
340	13857	C	No flags -WGstats
341	13857	C	-fast -xcrossfile -unroll=2 -WGstats
342	13857	C	-unroll=2 -fast -xcrossfile -WGstats
343	13857	C	-unroll=2 -WGstats
344	13857	C	-xcrossfile -fast -unroll=2 -WGstats
345	13857	C	-unroll=2 -fast -WGstats
346	13857	C	-fast -xcrossfile -WGstats
347	13857	C	-fast -unroll=2 -xcrossfile -WGstats
348	13857	C	-xcrossfile -unroll=2 -fast -WGstats
349	13857	C	-fast -unroll=2 -WGstats
350	13857	C	-xcrossfile -fast -WGstats
351	13857	C	-fast -WGstats

F.2 Anova on the metrics obtained in Experiment 3

Table F.2: ANOVA

Label	Name	Problem	Algorithm	Compiler	Interactions			
		Size (S)	(A)	Option (C)	$S * A$	$S * C$	$A * C$	$S * A * C$
q0	execution time	Yes	Yes	Yes	No	No	Yes	No
q1	bread/s	No	No	No	No	No	No	No
q2	lread/s	No	Yes	Yes	No	No	Yes	No
q3	%rcache	No	No	No	No	No	No	No
q4	bwrit/s	No	Yes	Yes	No	Yes	No	No
q5	lwrit/s	No	Yes	Yes	No	No	Yes	No
q6	%wcache	No	Yes	Yes	No	No	No	No
q7	pgout/s	No	No	Yes	No	Yes	No	No
q8	ppgout/s	No	No	Yes	No	Yes	No	No
q9	pgfree/s	No	No	No	No	No	No	No
q10	pgscan/s	No	No	No	No	No	No	No
q11	atch/s	No	Yes	Yes	No	No	Yes	No
q12	pgin/s	No	No	No	No	No	No	No
q13	ppgin/s	No	No	No	No	No	No	No
q14	pf1t/s	No	Yes	No	No	No	No	No
q15	vf1t/s	Yes	Yes	Yes	No	Yes	Yes	Yes
q16	%usr	No	Yes	Yes	No	No	Yes	No
q17	%sys	Yes	Yes	Yes	No	No	Yes	No
q18	%wio	No	No	No	No	No	No	No
q19	%idle	No	Yes	Yes	No	No	Yes	No
q20	pswch/s	Yes	Yes	Yes	No	No	Yes	No
<i>continued on next page</i>								

Table F.2: (cont'd).

Label	Name	Problem Size (S)	Algorithm (A)	Compiler Option (C)	Interactions			
					$S * A$	$S * C$	$A * C$	$S * A * C$
q21	fd0/wps	No	Yes	No	No	No	No	No
q22	fd0/util	No	Yes	No	No	No	No	No
q23	c0t0d0/rps	No	No	No	No	No	No	No
q24	c0t0d0/wps	No	Yes	Yes	No	Yes	Yes	Yes
q25	c0t0d0/util	No	Yes	Yes	No	No	Yes	No
q26	c0t1d0/rps	No	No	No	No	No	No	No
q27	c0t1d0/wps	No	No	No	No	No	No	No
q28	c0t1d0/util	No	No	No	No	No	No	No
q29	c1t6d0/wps	Yes	Yes	Yes	Yes	No	Yes	Yes
q30	c1t6d0/util	Yes	Yes	Yes	Yes	No	Yes	Yes
q31	cpu/us	No	Yes	Yes	No	No	Yes	No
q32	cpu/sy	Yes	Yes	Yes	No	No	Yes	No
q33	cpu/wt	No	No	No	No	No	No	No
q34	cpu/id	No	Yes	Yes	No	No	Yes	No
q35	memory/swap	No	Yes	Yes	No	No	Yes	No
q36	memory/free	No	Yes	Yes	No	No	No	No
q37	page/re	No	Yes	Yes	No	No	Yes	No
q38	page/mf	Yes	Yes	Yes	No	Yes	Yes	Yes
q39	page/pi	No	No	No	No	No	No	No
q40	page/po	No	No	Yes	No	Yes	No	No
q41	page/fr	No	No	No	No	No	No	No
q42	page/sr	No	No	No	No	No	No	No
q43	disk/s0	Yes	Yes	Yes	No	Yes	Yes	No
<i>continued on next page</i>								

Table F.2: (cont'd).

Label	Name	Problem Size (S)	Algorithm (A)	Compiler Option (C)	Interactions			
					$S * A$	$S * C$	$A * C$	$S * A * C$
q44	disk/s1	No	No	No	No	No	No	No
q45	disk/s2	No	Yes	No	No	No	No	No
q46	faults/in	No	Yes	Yes	Yes	No	Yes	No
q47	faults/sy	Yes	Yes	Yes	No	No	Yes	No
q48	faults/cs	Yes	Yes	Yes	No	No	Yes	No
q49	cpu/us1	No	Yes	Yes	No	No	Yes	No
q50	cpu/sy1	Yes	Yes	Yes	No	No	Yes	No
q51	cpu/id1	No	Yes	Yes	No	No	Yes	No

Yes implies the hypothesis is rejected at alpha level 0.05.

APPENDIX G

Experiment 4

G.1 Order of Execution of Experimental Runs for Experiment 4

This is a fully randomized full-factorial design. The following table contains the actual order in which the experimental runs were performed following a fully randomized scheme. There were a total of 96 experimental runs in this experiment.

Table G.1. Order of execution of experiments

Experimental Run	Size	Compiler Option	Algorithm	Data Structure
1	2	-fast -O5	G	Col-by-col
2	1	No flags	G	Col-by-col
3	1	No flags	F	Row-by-row
4	1	-fast	G	Col-by-col
5	2	-fast	G	Row-by-row
6	2	-O5	A	Row-by-row
7	2	No flags	A	Row-by-row
8	1	-O5	G	Col-by-col
9	2	-O5	G	Row-by-row
10	1	-fast -O5	F	Row-by-row
11	2	-fast	A	Row-by-row
<i>continued on next page</i>				

Table G.1 (cont'd).

Experimental Run	Size	Compiler Option	Algorithm	Data Structure
12	1	-O5	A	Col-by-col
13	2	No flags	G	Row-by-row
14	1	-fast -O5	F	Col-by-col
15	2	No flags	F	Row-by-row
16	1	No flags	A	Col-by-col
17	2	-O5	F	Col-by-col
18	2	No flags	F	Col-by-col
19	2	No flags	F	Col-by-col
20	2	-O5	G	Row-by-row
21	2	-fast	G	Row-by-row
22	2	No flags	A	Col-by-col
23	2	-fast	G	Col-by-col
24	1	No flags	F	Col-by-col
25	1	No flags	A	Col-by-col
26	2	-O5	A	Row-by-row
27	2	-fast -O5	A	Col-by-col
28	1	-fast	F	Row-by-row
29	2	-fast -O5	F	Col-by-col
30	1	-O5	F	Row-by-row
31	1	-fast	F	Row-by-row
32	1	-fast -O5	G	Row-by-row
33	1	-O5	F	Col-by-col
34	2	-fast -O5	G	Row-by-row
35	1	No flags	G	Row-by-row
36	1	-fast -O5	A	Col-by-col
<i>continued on next page</i>				

Table G.1 (cont'd).

Experimental Run	Size	Compiler Option	Algorithm	Data Structure
37	1	-fast -O5	F	Row-by-row
38	1	-fast	F	Col-by-col
39	1	-fast -O5	A	Col-by-col
40	2	-O5	G	Col-by-col
41	1	-fast	A	Row-by-row
42	1	-fast	G	Col-by-col
43	2	-fast -O5	F	Col-by-col
44	2	-O5	F	Row-by-row
45	2	No flags	G	Col-by-col
46	1	-O5	A	Row-by-row
47	1	-O5	A	Row-by-row
48	2	No flags	A	Col-by-col
49	1	-fast -O5	A	Row-by-row
50	2	-O5	F	Col-by-col
51	2	No flags	A	Row-by-row
52	1	-fast	A	Col-by-col
53	1	-fast	A	Col-by-col
54	2	-fast	F	Col-by-col
55	1	-O5	G	Col-by-col
56	2	-fast	G	Col-by-col
57	1	-fast -O5	G	Col-by-col
58	1	No flags	A	Row-by-row
59	2	-fast -O5	A	Col-by-col
60	2	-fast	A	Col-by-col
61	2	-fast -O5	A	Row-by-row
<i>continued on next page</i>				

Table G.1 (cont'd).

Experimental Run	Size	Compiler Option	Algorithm	Data Structure
62	2	-fast	F	Row-by-row
63	2	-fast -O5	A	Row-by-row
64	1	-fast -O5	F	Col-by-col
65	2	No flags	G	Row-by-row
66	1	-fast -O5	G	Row-by-row
67	1	No flags	F	Row-by-row
68	1	No flags	G	Row-by-row
69	1	-fast	F	Col-by-col
70	1	-fast	A	Row-by-row
71	2	-fast	A	Row-by-row
72	1	-O5	G	Row-by-row
73	2	-O5	A	Col-by-col
74	1	-O5	F	Row-by-row
75	2	-fast -O5	G	Row-by-row
76	2	No flags	G	Col-by-col
77	2	-O5	A	Col-by-col
78	1	No flags	A	Row-by-row
79	1	-fast	G	Row-by-row
80	2	-fast	A	Col-by-col
81	1	-fast -O5	A	Row-by-row
82	2	-fast	F	Row-by-row
83	1	-fast	G	Row-by-row
84	1	-O5	A	Col-by-col
85	1	-fast -O5	G	Col-by-col
86	2	-fast -O5	F	Row-by-row
<i>continued on next page</i>				

Table G.1 (cont'd).

Experimental Run	Size	Compiler Option	Algorithm	Data Structure
87	1	-O5	G	Row-by-row
88	1	No flags	F	Col-by-col
89	2	-fast	F	Col-by-col
90	1	-O5	F	Col-by-col
91	1	No flags	G	Col-by-col
92	2	-fast -O5	F	Row-by-row
93	2	-O5	F	Row-by-row
94	2	-fast -O5	G	Col-by-col
95	2	-O5	G	Col-by-col
96	2	No flags	F	Row-by-row

G.2 Anova on the metrics obtained in Experiment 4

Table G.2: ANOVA - main factors effect in experiment 4

Label	Name	Problem Size (<i>S</i>)	Compiler Option (<i>C</i>)	Algorithm (<i>A</i>)	Data Structure (<i>D</i>)
n0	Execution time	Yes	Yes	Yes	Yes
n1	lread/s	No	No	No	No
n2	bwrit/s	Yes	Yes	No	Yes
n3	lwrit/s	No	No	No	No
n4	%wcache	No	No	Yes	Yes
n5	pgout/s	Yes	Yes	Yes	Yes
n6	ppgout/s	Yes	Yes	Yes	Yes
n7	pgfree/s	Yes	Yes	Yes	Yes
<i>continued on next page</i>					

Table G.2: (cont'd).

Label	Name	Problem Size (<i>S</i>)	Compiler Option (<i>C</i>)	Algorithm (<i>A</i>)	Data Structure (<i>D</i>)
n8	atch/s	Yes	No	Yes	Yes
n9	pgin/s	No	No	No	No
n10	ppgin/s	No	No	No	No
n11	pflt/s	Yes	No	Yes	Yes
n12	vflt/s	Yes	No	Yes	Yes
n13	%usr	Yes	Yes	Yes	Yes
n14	%sys	Yes	No	Yes	Yes
n15	%wio	Yes	No	Yes	Yes
n16	%idle	Yes	Yes	Yes	Yes
n17	pswch/s	Yes	No	Yes	Yes
n18	c0t0d0/wps	Yes	Yes	Yes	Yes
n19	c0t0d0/util	Yes	Yes	Yes	Yes
n20	clt1d0/wps	No	No	Yes	No
n21	clt1d0/util	No	No	No	No
n22	memory/swap	No	No	No	No
n23	memory/free	No	No	Yes	No
n24	page/re	No	Yes	Yes	Yes
n25	page/mf	Yes	Yes	Yes	Yes
n26	page/pi	No	No	No	No
n27	page/po	Yes	Yes	Yes	Yes
n28	page/fr	Yes	Yes	Yes	Yes
n29	disk/s0	Yes	Yes	Yes	Yes
n30	faults/in	No	No	Yes	No
<i>continued on next page</i>					

Table G.2: (cont'd).

Label	Name	Problem Size (<i>S</i>)	Compiler Option (<i>C</i>)	Algorithm (<i>A</i>)	Data Structure (<i>D</i>)
n31	faults/sy	Yes	Yes	Yes	Yes
n32	faults/cs	Yes	Yes	Yes	Yes
n33	cpu/us	Yes	Yes	Yes	Yes
n34	cpu/sy	Yes	No	Yes	Yes
n35	cpu/id	Yes	Yes	Yes	Yes

Yes implies the hypothesis is rejected at alpha level 0.05.

Table G.3. ANOVA - two term interaction effect in experiment 4

Label	Name	S*C	S*A	S*D	C*A	C*D	A*D
n0	Execution time	Yes	Yes	Yes	Yes	Yes	Yes
n1	lread/s	No	No	No	No	No	No
n2	bwrit/s	Yes	Yes	Yes	Yes	No	Yes
n3	lwrit/s	No	No	No	No	No	No
n4	%wcache	Yes	Yes	No	No	No	Yes
n5	pgout/s	No	Yes	Yes	Yes	No	Yes
n6	ppgout/s	No	Yes	Yes	No	Yes	Yes
n7	pgfree/s	No	Yes	Yes	No	Yes	Yes
n8	atch/s	Yes	Yes	Yes	No	No	Yes
n9	pgin/s	No	No	No	No	No	No
n10	ppgin/s	No	No	No	No	No	No
n11	pflt/s	Yes	Yes	Yes	No	No	Yes
n12	vflt/s	Yes	Yes	Yes	No	No	Yes
<i>continued on next page</i>							

Table G.3 (cont'd).

Label	Name	S*C	S*A	S*D	C*A	C*D	A*D
n13	%usr	Yes	Yes	Yes	Yes	Yes	Yes
n14	%sys	No	Yes	No	No	No	No
n15	%wio	Yes	Yes	Yes	No	No	Yes
n16	%idle	Yes	Yes	Yes	Yes	Yes	Yes
n17	pswch/s	No	Yes	Yes	No	No	Yes
n18	c0t0d0/wps	Yes	Yes	Yes	Yes	Yes	Yes
n19	c0t0d0/util	Yes	Yes	Yes	Yes	Yes	Yes
n20	c1t1d0/wps	No	Yes	No	No	No	Yes
n21	c1t1d0/util	No	No	No	No	No	No
n22	memory/swap	No	Yes	No	No	No	No
n23	memory/free	No	No	No	No	No	No
n24	page/re	Yes	No	No	Yes	Yes	Yes
n25	page/mf	Yes	Yes	No	Yes	Yes	Yes
n26	page/pi	No	No	No	No	No	No
n27	page/po	Yes	Yes	No	No	No	Yes
n28	page/fr	Yes	Yes	No	No	No	Yes
n29	disk/s0	Yes	Yes	Yes	Yes	Yes	Yes
n30	faults/in	No	No	No	No	No	No
n31	faults/sy	Yes	Yes	Yes	Yes	Yes	Yes
n32	faults/cs	Yes	Yes	Yes	Yes	Yes	Yes
n33	cpu/us	Yes	Yes	Yes	Yes	Yes	Yes
n34	cpu/sy	No	No	No	No	No	Yes
n35	cpu/id	Yes	Yes	Yes	Yes	Yes	Yes

Yes implies the hypothesis is rejected at alpha level 0.05.

Table G.4. ANOVA - three and four term interaction effect in experiment 4

Label	Name	S*C*A	S*C*D	C*A*D	S*C*A*D
n0	Execution time	Yes	Yes	Yes	Yes
n1	lread/s	No	No	No	No
n2	bwrit/s	Yes	Yes	Yes	Yes
n3	lwrit/s	No	No	No	No
n4	%wcache	Yes	No	No	No
n5	pgout/s	Yes	No	No	Yes
n6	ppgout/s	Yes	Yes	Yes	Yes
n7	pgfree/s	Yes	Yes	Yes	Yes
n8	atch/s	Yes	Yes	No	Yes
n9	pgin/s	No	No	No	No
n10	ppgin/s	No	No	No	No
n11	pft/s	Yes	Yes	No	Yes
n12	vft/s	Yes	Yes	No	Yes
n13	%usr	Yes	Yes	Yes	Yes
n14	%sys	No	No	No	Yes
n15	%wio	Yes	Yes	Yes	Yes
n16	%idle	Yes	Yes	Yes	Yes
n17	pswch/s	No	No	No	Yes
n18	c0t0d0/wps	Yes	Yes	Yes	Yes
n19	c0t0d0/util	Yes	Yes	Yes	Yes
n20	c1t1d0/wps	No	No	No	No
n21	c1t1d0/util	No	No	No	No
n22	memory/swap	No	No	No	No
<i>continued on next page</i>					

Table G.4 (cont'd).

Label	Name	S*C*A	S*C*D	C*A*D	S*C*A*D
n23	memory/free	No	No	No	No
n24	page/re	Yes	Yes	Yes	Yes
n25	page/mf	Yes	Yes	No	Yes
n26	page/pi	No	No	No	No
n27	page/po	Yes	Yes	No	Yes
n28	page/fr	Yes	Yes	No	Yes
n29	disk/s0	Yes	Yes	Yes	Yes
n30	faults/in	No	No	No	No
n31	faults/sy	Yes	Yes	Yes	Yes
n32	faults/cs	Yes	Yes	Yes	Yes
n33	cpu/us	Yes	Yes	Yes	Yes
n34	cpu/sy	No	Yes	No	No
n35	cpu/id	Yes	Yes	Yes	Yes

Yes implies the hypothesis is rejected at alpha level 0.05.

APPENDIX H

Additional Fortran files

H.1 Program to test new routines

This is a Fortran 77 program used to test the correctness of new routines used to perform matrix-vector multiplication. In this particular program, a routine presented in by Golub and Van Loan in their book Matrix Computations was tested.

```

      Program Testing
c c Testing BiMATVECCav with a matrix read from a file
c Doing a matrix-vector multiplication where the
c matrix is symmetric and the upper diagonal is
c saved row by row
      Integer apUnk,i
      Integer MaxapUnk,tot
      Complex*16 temp1(200),temp2(200),bestGuess(200)
      Complex*16 error(7), Ybi(20100)
c c Ybi is the lower triangle of the symmetric matrix
      apUnk = 200
      MaxapUnk=30000

      Open(unit=27,file='matrixIn',status='old')
      tot=0
      Do i = 1 , MaxapUnk
        Read(27,fmt=*,end=99)Ybi(i)
        tot = tot + 1
c      Print*,tot,i,Ybi(i)
      EndDo
99  close(unit=27) c c      Print the matrix
      Do i = 1, tot
        Print*,i,Ybi(i)
      EndDo

      Open(unit=28,file='vectorIn',status='old')
      tot=0
      Do i = 1 , MaxapUnk
```

```

        Read(28,fmt=*,end=79)bestGuess(i)
        tot = tot + 1
    EndDo
79  close(unit=28)

c    Print the vector
    Do i = 1, tot
        Print*,i,bestGuess(i)
    EndDo

c Calling the original version of BiMATVECCav
    Call BiMATVECCav1(apUnk,Ybi,bestGuess,temp1)
    Print*,'Temp1 Original'
    Do i = 1,apUnk
        Print*,temp1(i)
    EndDo

c Calling the modified version of BiMATVECCav
    Call BiMATVECCav2(apUnk,Ybi,bestGuess,temp2)
    Print*,'Temp2 Modified'
    Do i = 1, apUnk
        Print*,temp2(i)
    EndDo

    Print*,'Error'
    Do i = 1, apUnk
        error(i)=temp1(i)-temp2(i)
    EndDo
    Print*,error(i)
    EndDo

End
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
Subroutine BiMATVECCav1(apUnk,Ybi,vector,product)

c First algorithm to solve in parallel with OpenMP
    Integer apUnk
    Integer BIrowEndPoint(200)
    Complex*16 Ybi(*),vector(*),product(*)
c c Local variables c
    Integer row,col,index
    Complex*16 matEntry
c c Load the diag BIrowEndPoint vectors c
    BIrowEndPoint(1) = 0
    Do row = 2,apUnk
        BIrowEndPoint(row) = BIrowEndPoint(row-1)+
&                                (apUnk - row+1)
    EndDo
c c Do the MATVEC c

```

```

    Do row = 1,apUnk
      Do col = 1,apUnk
        if (row .LT. col) then
          index = BIrowEndPoint(row)+col
        else
          index = BIrowEndPoint(col)+row
        endif
        matEntry = Ybi(index)
        product(row) = product(row) + matEntry*vector(col)
      EndDo
    EndDo

  Return
End

cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      Subroutine BiMATVECCav2(apUnk,Ybi,vector,product)

c Golub & Van Loan algorithm to solve in parallel with OpenMP
      Integer apUnk
      Complex*16 Ybi(*),vector(*),product(*)
c c Local variables c
      Integer row,col,index
c c Do the MATVEC c
      Do col = 1, apUnk
        Do row = 1, col-1
          index = (row-1)*apUnk - row*(row-1)/2 + col
c          Write(6,*)'index', index
          product(row) = product(row)+Ybi(index)*vector(col)
        EndDo
        Do row = col, apUnk
          index = (col-1)*apUnk - col*(col-1)/2 + row
c          Write(6,*)'index', index
          product(row) = product(row) + Ybi(index)*vector(col)
        EndDo
      EndDo

  Return
End

```

APPENDIX I

Matlab Files

I.1 Program to compute order of experimental runs

This example Matlab code computes the order in which experimental runs will be executed. Since it contains a random number generator, each run will result in a different result or order in which the experiment will run.

```
% Program to generate order of experiments
% For Split-Plot Design where 1st the size is
% selected, then the algorithm, and last
% the compiler options.
% Cannot use Minitab 13 since max no. of levels
% allowed is 9
%
% The output is a matrix with 3 columns where
% column 1 is size, column 2 is algorithm, and
% column 3 is compiler option
%
% Nayda G. Santiago July 20, 2001
%
clear
r=input('What is the name of the output file? ','s');
diary (r);
rand('state',sum(100*clock));
experiments=[];
number_alg=1; % Number of Levels in Algorithms
number_co=13; % Number of Levels in Compiler Options
number_si=3; % Number of Levels in Size
number_rep=3; % Number of Repetitions of the experiment
% Number of experiments
number_exp=number_alg*number_si*number_rep;
for p=1:number_exp,
    a=200*rand(1,200);
    b=mod(a,number_co);
    c=ceil(b);
```

```

    exper(1)=c(1);
    k=2;
    for i=2:number_co,
        exper(i)=c(k);
        k=k+1;
        j=1;
        while j <= (i-1),
            if exper(i) == exper(j),
                exper(i)=c(k);
                k=k+1;
                j=1;
            else
                j=j+1;
            end
        end
    end
    experiments=[experiments;exper];
end
sizes=[];
for p=1:number_rep
    a=300*rand(1,200);
    b=mod(a,3);
    c=ceil(b);
    tamano(1)=c(1);
    k=2;
    for i=2:number_si,
        tamano(i)=c(k);
        k=k+1;
        j=1;
        while j <= (i-1),
            if tamano(i) == tamano(j),
                tamano(i)=c(k);
                k=k+1;
                j=1;
            else
                j=j+1;
            end
        end
    end
    sizes=[sizes;tamano];
end
%
algorithm=[3]; % Algorithm C is 3

% Computing vector containing experiments for perl file
% Each row will be [Size, Algorithm, Compiler Option]
s=reshape(sizes',number_rep*number_si,1); %convert size
                                         % into a column

```

```

ord_exp=[];
for i=1:number_si*number_rep
    for k=1:number_co
        % We only have one algorithm
        elem=[s(i) algorith experiments(i,k)];
        ord_exp=[ord_exp;elem];
    end
end
sizes
algorith
experiments
ord_exp
diary off

```

I.2 Routine to compute entropy cost function

```

function entropy=entropydash(A)
% Function to compute entropy as defined by Dash, et. al. in
% M. Dash, H. Liu, and J. Yao, Dimensionality Reduction
% of Unsupervised Data, Proc. of the 9th IEEE Intl. Conference
% on Tools with Artificial Intelligence, p. 532-539, Nov 1997
%
[N,M]=size(A); normalization=max(A)-min(A); for i=1:N
    normalizationMatrix(i,:)=normalization;
end
normalizedA=A./normalizationMatrix; %Normalization
D=dist(normalizedA');

k=1; for i=1:N
    for j=i+1:N
        v(k)=D(i,j);
        k=k+1;
    end
end Daverage= mean(v);
alpha=-log(0.5)/Daverage; % Alpha used in equation (2)

for i=1:N
    for j=1:N
        S(i,j)= exp(-(alpha*D(i,j)));
    end
end

for i=1:N
    for j=1:N
        if S(i,j) == 1
            E(i,j)=0;

```

```

        else
            % Equation 1 from paper
            E(i,j)=S(i,j)*log2(S(i,j))+(1-S(i,j))*log2(1-S(i,j));
        end
    end
end
entropy = -(sum(sum(E)));

```

I.3 Routine to show scree test and the Kaiser-Guttman criteria

```

function [total] = KG_scree(data,names);
% data - matrix unnormalized data
%
%*****
[rows,cols] = size(data);

% Normalize data using norm 2
for i=1:cols
    normalization(i)=norm(data(:,i));
end
normalizeddata =data./(ones(rows,1)*normalization);
A=normalizeddata;
% End Normalization

% Plot the eigenvalues of the correlation matrix
% Kaiser-Guttman method eig > 1
correlacion= corrcoef(A);
EigValues=eig(correlacion);
EigValues=flipud(EigValues);

kg=1; for i=1:cols
    if (EigValues(i) > 1)
        kg=i;
    end
end
disp(sprintf('There are %d eigenvalues larger
than one.',kg));

w=20;          % How many eigenvalues to plot
if (w < cols)
    plot(EigValues(1:w),'b-');
    hold on
    line=ones(1,20);
    plot(EigValues(1:w),'ro');
    plot(line)
    ylabel('Value');

```

```

        title('Correlation Matrix Eigenvalues');
        grid
        hold off
    else
        disp('You have requested too many eigenvalues')
    end
end

```

I.4 Program to validate intrinsic dimensionality estimators

This Matlab code generates a synthetic data set with the same statistics as the data set from experiment 2. However, the first nine columns of the matrix are independent. All the additional columns are multiples of the first nine plus small noise. This forces this matrix to have an intrinsic dimension of nine. Then the code proceeds to apply all three dimensionality estimator methods to estimate the dimension of the data.

```

% This file will create a synthetic data set with known
% dimension of 9. We will use the three different estimators
% used in our data set to estimate the intrinsic dimensionality
% of the data set.
%
load Exp2
[rows,cols] = size(data);
% Synthetic data
A=250*rand(rows,9);
noise=0.0001*randn(size(A));
B=[A 2*A+noise 3*A-noise 4*A+2*noise 5*A-2*noise 6*A(:,1:2)];
% Estimate mean and variance of validation data.
mean1=mean(data);
sigma1=std(data);
%Estimate mean and variance of synthetic data.
mean2=mean(B);
sigma2=std(B);
%
for i=1:rows
    for j=1:cols
        data2(i,j)=(((B(i,j)-mean2(j))/sigma2(j))*sigma1(j))+mean1(j);
    end
end
% Normalize the data with norm 2 for covariance matrix
for i=1:cols
    normalization(i)=norm(data2(:,i));
end normalizeddata =data2./(ones(rows,1)*normalization);

```

```

% End Normalization
%
% Principal Component Analysis using the covariance matrix
% explainedcov contains the variance explained by each eigenvalue
covdata=cov(normalizeddata);
% Compute the covariance matrix of the data
[pccov,latentcov,explainedcov]=pcacov(covdata);
%
%
percentage=95;
i=1;
sum1=0;
while sum1 < percentage;
    sum1=sum(explainedcov(1:i));
    i=i+1;
end
numComponentsCov=i-1;
disp(sprintf('Retain %d components from covariance matrix.',
numComponentsCov));
disp('(95% variance retained)');
%*****
% Plot the eigenvalues of the correlation matrix
% Kaiser-Guttman method eig > 1
C=normalizeddata;
correlacion= corrcoef(C);
Autovalores=eig(correlacion);
Autovalores=flipud(Autovalores);

kg=1; for i=1:cols
    if (Autovalores(i) > 1)
        kg=i;
    end
end
disp(sprintf('There are %d eigenvalues larger than one.',kg));

w=20;                % How many eigenvalues to plot
if (w < cols)
    plot(Autovalores(1:w),'b-');
    hold on
    linea=ones(1,20);
    plot(Autovalores(1:w),'ro');
    plot(linea)
    ylabel('Value');
    title('Correlation Matrix Eigenvalues');
    grid
    hold off
%
else

```

```
    disp('You have requested too many eigenvalues')  
end
```

APPENDIX J

Perl Script files

J.1 Script A: Generating Summary of Metrics

This script generates a summary file with all metrics. It determines how many lines of metrics to read according to the time when the application finished running.

```
#!/usr/bin/perl -w

$file_timetrack="timetrack";
$file_time="time_out";
$file_sar="sar_out";
$file_iostat="iostat_out";
$file_vmstat="vmstat_out";
$file_mpstat="mpstat_out";
$file_summary="summary_output";

for ($k=1; $k<=123; $k++){
    $directory = "E".$k;
    #$directory = "Etest";
    print 'Directory ' . $directory . "\n";
    chdir($directory);

    # Get the name of makefile since the names are all
    # different (makefile1 to makefile13)
    $file_make='ls | grep makef';
    chop $file_make;

    # Get the total number of unknowns to solve from the
    # file descr wich was created by prism
    $Number_unknowns='grep Total descr | grep unk';
    chop $Number_unknowns;

    # OPENING OUTPUT FILE
    open(FILE_OUT,">$file_summary") or die "Cannot write to file \n";
    print FILE_OUT 'Directory ' . $directory . "\n";
    print FILE_OUT "Sampling every 20 seconds, total of 474 iterations\n";
```

```

print FILE_OUT $Number_unknowns."\n";

# READING MAKEFILE
open(FILE_2,"<$file_make") or die "Cannot read file \n";
# Find line with compiler options
while (<FILE_2>){
    if($_ =~ /\s*$/) {next;}; # Remove blank lines
    chomp; # Remove newline character
    @space_sep = split; # Split line at blank space
    if ($space_sep[0] eq "FOPT=") {
        shift(@space_sep); # Eliminate 1st element of space_sep array
        print FILE_OUT "Compiler Options: ";
        foreach $option (@space_sep) { #Print all compiler options
            print FILE_OUT $option." ";
        }
        print FILE_OUT "\n";
    }
}

}
close(FILE_2);

# READING TIMETRACK
open(FILE_3,"<$file_timetrack") or die "Cannot read file \n";
# Find line with 'real' string and get the elapsed time
while (<FILE_3>){
    if($_ =~ /\s*$/) {next;}; # Remove blank lines
    chomp; # Remove newline character
    @space_sep = split; # Split line at blank space
    if ($space_sep[0] eq "Start"){
        $a=<FILE_3>;
        chomp ($a);
        @time_line = split(/\s+/, $a); # Split line at blank space
        $start_day = $time_line[0]; # First element is day
        $start_time = $time_line[3]; # Third element is time
        print FILE_OUT "Start: ".$start_day." ".$start_time."\n";
    }
    if ($space_sep[0] eq "End"){
        $a=<FILE_3>;
        chomp ($a);
        @time_line = split(/\s+/, $a); # Split line at blank space
        $end_day = $time_line[0]; # First element is day
        $end_time = $time_line[3]; # Third element is end time
        @time_fields = split(/:/, $end_time); # Split at min
        print FILE_OUT "End: ".$end_day." ".$end_time."\n";
        if ($start_day ne $end_day){
            print FILE_OUT "Not same day\n";
        }
    }
}

```

```

}
close(FILE_3);

# READING TIME
open(FILE_4,"<$file_time") or die "Cannot read file \n";
# Find line with 'real' string and get the elapsed time
while (<FILE_4>){
    if($_ =~ /\s*$/) {next;}; # Remove blank lines
    chomp; # Remove newline character
    @space_sep = split; # Split line at blank space
    if ($space_sep[0] eq "real") {
        $elapsed_time = $space_sep[1]; # 2nd element is elapsed time
        @min_fields = split(/:/,$elapsed_time); # Split at min
        $minutes = $min_fields[0]; # Get minutes
        $seconds_dec = $min_fields[1];
        @sec_fields = split(/\./,$seconds_dec); # Split at secs
        $seconds = $sec_fields[0];
        $total_time_prism = $minutes*60 + $seconds;
        print FILE_OUT "Prism time in sec: ".$total_time_prism."\n";
    }
}
close(FILE_4);

# READING SAR
$no_metrics = "yes"; # Flag: when we can start reading the metrics
$count_metrics = 0;

# Only 28 of the metrics are relevant. We do not need those measured
# by the -v flag. Initialize the cumulative sum to zero.
for ($l=0; $l<=28; $l++){
    $cum_metrics1[$l]=0;
}

open(FILE_5,"<$file_sar") or die "Cannot read file \n";
while(<FILE_5>){
    if($_ =~ /\s*$/) {next;}; # Remove blank lines
    chomp; # Remove newline character
    @space_sep = split; # Split line at blank space

    # Read the lines with metrics
    if ($no_metrics eq "no" ){ # We can start reading metrics
        @metrics = @space_sep;
        # Get the number of elements in the array @metrics
        $num_of_elements = scalar(@metrics);
        if ($metrics[0] =~ /:/) { # Match time with ':' character
            $line_index = $num_of_elements;
            # End loop when the time matches the end sampling time
            # Remove comment for E123 since there is a change in date

```

```

        # and ends before it can finish.
        last if ($metrics[0] gt $end_sampling_time);# End while loop
        for ($m = 1; $m<= $line_index-1; $m++) {
            $cum_metrics1[$m]=$cum_metrics1[$m] + $metrics[$m];
        }
        $count_metrics = $count_metrics + 1;
    } elsif ($metrics[0] !~ /\//) {    # Do not use line with
                                        # slash (/) character

        $old_line_index = $line_index;
        $line_index = $line_index + $num_of_elements;
        for ($m = $old_line_index; $m<= $line_index-1; $m++) {
            $cum_metrics1[$m]=$cum_metrics1[$m] +
                $metrics[$m-$old_line_index];
        }
    }
}

if ($space_sep[0] eq "/bin/sar") {
    $period = $space_sep[2];          # 2nd element is elapsed time
    $repetitions = $space_sep[3];     # 2nd element is elapsed time
    print FILE_OUT "Period: ".$period."\n";
    print FILE_OUT "Repetitions: ".$repetitions."\n";
    $total_time_sar=$period*$repetitions;
    print FILE_OUT "Sar time: ".$total_time_sar."\n";
    if ($total_time_prism > $total_time_sar){
        print FILE_OUT "ERROR taking sar metrics. Sar was short
            in time.\n";
    }
}

if ($space_sep[0] eq "SunOS") {
    $a=<FILE_5>;
    $a=<FILE_5>;
    chomp ($a);
    @first_line = split(/\s+/, $a);    # Split line at blank space
    $start_time = $first_line[0];      # 2nd element is elapsed time
    print FILE_OUT "Initial time: ".$start_time."\n";
    @time_fields = split(/: +/, $start_time);# Split line at : symbol
    $start_hour = $time_fields[0];
    $start_min = $time_fields[1];
    $start_sec = $time_fields[2];
    @name_metrics = @first_line;
    $a=<FILE_5>;
    chomp ($a);
    @name_metrics = (@name_metrics, split(/\s+/, $a));    # Split
                                                            # line at blank space

    $a=<FILE_5>;
    chomp ($a);
    @name_metrics = (@name_metrics, split(/\s+/, $a));    # Split
                                                            # line at blank space

```

```

$a=<FILE_5>;
chomp ($a);
@name_metrics = (@name_metrics, split(/\s+/, $a)); # Split
# line at blank space

$a=<FILE_5>;
# Remove the next two lines since I am not including
# the metrics by -v flag into consideration
#chomp ($a);
#@name_metrics = (@name_metrics, split(/\s+/, $a)); # Split
# line at blank space

$a=<FILE_5>;
chomp ($a);
@name_metrics = (@name_metrics, split(/\s+/, $a)); # Split
# line at blank space

#
# COMPUTING THE END SAMPLING TIME
#
# Sampling time is the time to take all samples while
# prism is running
$sampling_time = $total_time_prism -
                ($total_time_prism%$period);
print FILE_OUT "Sampling total time: ".$sampling_time."\n";
$mins_plus = 0;
$hours_plus = 0;

# SECONDS
$secs = $sampling_time%60;
$end_secs = $start_sec + $secs;
if ($end_secs >= 60){
    $end_secs = $end_secs - 60;
    $mins_plus = 1;
}
if ($end_secs <= 9){
    $end_secs = "0".$end_secs;
}

# MINUTES
$mins = ($sampling_time - $secs) / 60;
$end_mins = $start_min + $mins + $mins_plus;
if ($end_mins >= 60){
    $end_mins = $end_mins - 60;
    $hours_plus = 1;
}
if ($end_mins <= 9){
    $end_mins = "0".$end_mins;
}

```

```

# HOURS
$hrs = ($sampling_time - $secs - $mins*60) / 60;
$end_hour = $start_hour + $hrs + $hours_plus;
if ($end_hour >= 24){
    $end_hour = $end_hour - 24;
}
if ($end_hour <= 9){
    $end_hour = "0".$end_hour;
}

# END SAMPLING TIME
$end_sampling_time = $end_hour.":".$end_mins.":".$end_secs;
print FILE_OUT "End sampling time: ".$end_sampling_time."\n";
$no_metrics = "no";
}
}

print FILE_OUT "ANALYSIS OF SAR\n";
print FILE_OUT "Count of metrics: ".$count_metrics."\n";
for ($l=0; $l<=28; $l++){
    $avg_metrics[$l]=$cum_metrics1[$l]/$count_metrics;
}
# Remove the first element of the array
shift(@name_metrics);
shift(@avg_metrics);
foreach $name_metric (@name_metrics){print FILE_OUT $name_metric," ";}
print FILE_OUT "\n";
foreach $cum_metric (@avg_metrics) { print FILE_OUT $cum_metric, " ";}
print FILE_OUT "\n";
close(FILE_5);

# READING IOSTAT
print FILE_OUT "ANALYSIS OF IOSTAT\n";
$read_metrics = "no"; # Flag, when we can start reading the metrics
$lineNumber = 0;
$tot_lines = $count_metrics-1;

# We have 18 metrics in iostat.
for ($l=0; $l<=17; $l++){
    $cum_metrics2[$l]=0;
}
open(FILE_6,"<$file_iostat") or die "Cannot read file \n";
while(<FILE_6>){
    if($_ =~ /\s*$/) {next;}; # Remove blank lines
    chomp; # Remove newline character
    @space_sep = split; # Split line at blank space

# READ METRICS DESCRIPTION
if ($space_sep[0] eq "tty" && $read_metrics eq "no"){ # Next

```

```

                                # few lines contain metrics
print FILE_OUT $_."\n"; # Print first line of description
$a = <FILE_6>;
chomp($a);
print FILE_OUT $a."\n"; # Print second line of description
$a = <FILE_6>;          # Discard 1st line of measurements
                        # See description of IOSTAT command
$read_metrics = "yes"; # May read metrics
}
# Here we can read the metrics
if (($read_metrics eq "yes") && ($space_sep[0] ne "tty") &&
($space_sep[0] ne "tin") ) {
    @metrics = @space_sep;
    # Get the number of elements in the array @metrics
    $num_of_elements = scalar(@metrics);
    last if ($lineNumber >= $tot_lines); # End main while loop
    for ($m = 0; $m<= $num_of_elements-1; $m++) {
        $cum_metrics2[$m]=$cum_metrics2[$m] + $metrics[$m];
    }
    $lineNumber = $lineNumber + 1;
}
}
for ($l=0; $l<=$num_of_elements-1; $l++){
    $avg_iostat[$l]=($cum_metrics2[$l]/$tot_lines);
}
foreach $cm (@avg_iostat) { print FILE_OUT $cm, " ";}
print FILE_OUT "\n";
close(FILE_6);

# READING VMSTAT
print FILE_OUT "ANALYSIS OF VMSTAT\n";
$read_metrics = "no"; # Flag to know when we can start
                    # reading the metrics

$lineNumber = 0;

# We have 22 metrics in iostat.
for ($l=0; $l<=21; $l++){
    $cum_metrics3[$l]=0;
}
open(FILE_7,"<$file_vmstat") or die "Cannot read file \n";
while(<FILE_7>){
    if($_ =~ /\s*$/) {next;}; # Remove blank lines
    chomp;                  # Remove newline character
    @space_sep = split;     # Split line at blank space

    # READ METRICS DESCRIPTION
    if ($space_sep[0] eq "procs" && $read_metrics eq "no"){
        # Next few lines contain metrics

```

```

    print FILE_OUT $_."\n"; # Print first line of description
    $a = <FILE_7>;
    chomp($a);
    print FILE_OUT $a."\n"; # Print second line of description
    $a = <FILE_7>;      # Discard 1st line of measurements
                        # See description of VMSTAT command
    $read_metrics = "yes"; # Now we may read metrics
}
# Here we can read the metrics
if (($read_metrics eq "yes") && ($space_sep[0] ne "procs") &&
($space_sep[0] ne "r") ) {
    @metrics = @space_sep;
    # Get the number of elements in the array @metrics
    $num_of_elements = scalar(@metrics);
    last if ($lineNumber >= $tot_lines); # End main while loop
    for ($m = 0; $m<= $num_of_elements-1; $m++) {
        $cum_metrics3[$m]=$cum_metrics3[$m] + $metrics[$m];
    }
    $lineNumber = $lineNumber + 1;
}
}
for ($l=0; $l<=$num_of_elements-1; $l++){
    $avg_vmstat[$l]=($cum_metrics3[$l]/$tot_lines);
}
foreach $cm (@avg_vmstat) { print FILE_OUT $cm, " ";}
print FILE_OUT "\n";
close(FILE_7);

# READING  MPSTAT
print FILE_OUT "ANALYSIS OF MPSTAT\n";
$read_metrics = "no"; # Flag to know when we can start reading
                    # the metrics
$lineNumber = 0;

# We have 16 metrics in mpstat.
for ($l=0; $l<=15; $l++){
    $cum_metrics4_0[$l]=0;
    $cum_metrics4_1[$l]=0;
    $cum_metrics4_2[$l]=0;
    $cum_metrics4_3[$l]=0;
}
open(FILE_8,"<$file_mpstat") or die "Cannot read file \n";
while(<FILE_8){
    if($_ =~ /\s*$/) {next;}; # Remove blank lines
    chomp;                  # Remove newline character
    @space_sep = split;     # Split line at blank space

# READ METRICS DESCRIPTION

```

```

if ($space_sep[0] eq "CPU" && $read_metrics eq "no"){
# Next few lines contain metrics
    print FILE_OUT $_."\n"; # Print first line of description
    $a = <FILE_8>;          # Discard 1st four lines of measurements
    $a = <FILE_8>;          # See description of VMSTAT command
    $a = <FILE_8>;
    $a = <FILE_8>;
    $read_metrics = "yes"; # Now we may read metrics
}
# Here we can read the metrics. There are four lines, one
# per processor
if (($read_metrics eq "yes") && ($space_sep[0] ne "CPU")) {
    last if ($lineNumber >= $tot_lines); # End main while loop

    @metrics = @space_sep;
    # Get the number of elements in the array @metrics
    $num_of_elements = scalar(@metrics);
    $cpu_id = $metrics[0];
    if ($cpu_id == 0) {
        for ($m = 0; $m<= $num_of_elements-1; $m++) {
            $cum_metrics4_0[$m]=$cum_metrics4_0[$m] + $metrics[$m];
        }
    } elsif ($cpu_id == 1) {
        for ($m = 0; $m<= $num_of_elements-1; $m++) {
            $cum_metrics4_1[$m]=$cum_metrics4_1[$m] + $metrics[$m];
        }
    } elsif ($cpu_id == 2) {
        for ($m = 0; $m<= $num_of_elements-1; $m++) {
            $cum_metrics4_2[$m]=$cum_metrics4_2[$m] + $metrics[$m];
        }
    } elsif ($cpu_id == 3) {
        for ($m = 0; $m<= $num_of_elements-1; $m++) {
            $cum_metrics4_3[$m]=$cum_metrics4_3[$m] + $metrics[$m];
        }
        $lineNumber = $lineNumber + 1;
    } else {
        die "Cannot identify cpu id \n";
    }
}
}
for ($l=0; $l<=$num_of_elements-1; $l++){
    $avg_mpstat0[$l]=($cum_metrics4_0[$l]/$tot_lines);
    $avg_mpstat1[$l]=($cum_metrics4_1[$l]/$tot_lines);
    $avg_mpstat2[$l]=($cum_metrics4_2[$l]/$tot_lines);
    $avg_mpstat3[$l]=($cum_metrics4_3[$l]/$tot_lines);
}
foreach $cm (@avg_mpstat0) { print FILE_OUT $cm, " "; }
print FILE_OUT "\n";

```

```

        foreach $cm (@avg_mpstat1) { print FILE_OUT $cm, " "; }
        print FILE_OUT "\n";
        foreach $cm (@avg_mpstat2) { print FILE_OUT $cm, " "; }
        print FILE_OUT "\n";
        foreach $cm (@avg_mpstat3) { print FILE_OUT $cm, " "; }
        print FILE_OUT "\n";
        close(FILE_8);

        chdir("..");
    }
}

```

J.2 Script B: Create Crontab file

Script to generate the crontab file that will automatically run the OS calls to measure performance.

```

#!/usr/local/bin/perl -w
#
# Syntax:
#       create_crontab.pl <nfex> <tint> <inda> <inti> [> outputfile]
# where:
#       nfex = Number of First EXperiment
#       tint = Time INTerval in minutes
#       inda = INitial DATE in mm/dd
#       inti = INitial TIme in military format HH:MM
#
# Command scratchpad
$command1 = '/home/nayda/private/Fresh/Testing/torunprism';
$command2 = '/bin/iostat -cDnt 20';
$command3 = '/bin/vmstat 20';
$command4 = '/bin/mpstat 20';
$command5 = '/bin/sar -bgpuvw 20';

# Input filename
$infile = 'ord_exp';
$noexp = 13;          # Number of experiments to process

# Splits the initial time and date information
@inti = split(":",$ARGV[3]);
@inda = split("/", $ARGV[2]);
@curtida = ($inti[1], $inti[0], $inda[1], $inda[0]);
# First othertime is for compiler options 1 and 3
$othertime1 = ($ARGV[1] - 8 + 2) * 60 / 20;

```

```

# $othertime1 = ($ARGV[1] - 10 + 2) * 60 / 20;
# Cambiamos el 10 por 8 para agilizar los experimentos.
# Second othertime is for all other compiler options
$othertime2 = ($ARGV[1] - 22 + 2) * 60 / 20;

# Searches for the number of the first experiment in input file
$lncnt = 1;
open(ORDEXP, "$infile") || die "Could not find $infile";
while (<ORDEXP>) {
    if ($lncnt < $ARGV[0]) {
        # Advanced in input file until line given by nfex
        $lncnt += 1;
    }
    elsif ($noexp) {
        # Generates the command lines for the next 13 lines
        @expdata = split;
        if ($expdata[2] == 1) {
            $othertime = $othertime1;
        }
        elsif ($expdata[2] == 3) {
            $othertime = $othertime1;
        }
        else {
            $othertime = $othertime2;
        }

        # The first command
        print "$curtida[0] $curtida[1] $curtida[2] $curtida[3] *
$command1$expdata[2]\n";

        # Advances time 1 second and generate the other 4 commands
        @curtida = &get_new_time($curtida[0], $curtida[1], $curtida[2],
$curtida[3], 1);
        print "$curtida[0] $curtida[1] $curtida[2] $curtida[3] *
$command2 $othertime\n";
        print "$curtida[0] $curtida[1] $curtida[2] $curtida[3] *
$command3 $othertime\n";
        print "$curtida[0] $curtida[1] $curtida[2] $curtida[3] *
$command4 $othertime\n";
        print "$curtida[0] $curtida[1] $curtida[2] $curtida[3] *
$command5 $othertime\n";

        # Update the time counters for next experiment
        @curtida = &get_new_time($curtida[0], $curtida[1], $curtida[2],
$curtida[3], $ARGV[1]);
        $noexp -= 1;
    }
    else {

```

```

        # Terminates execution
        last;
    }
}

# A subroutine to compute the time and date
sub get_new_time{
    $minutes = $_[0];
    $hours = $_[1];
    $days = $_[2];
    $months = $_[3];

    $minutes += $_[4];
    if ($minutes > 59) {
        $minutes -= 60;
        $hours += 1;

        if ($hours > 23) {
            $hours -= 24;
            $days += 1;

            if (($days > 31) && (($months == 1) || ($months == 3)
                || ($months == 5) || ($months == 7) || ($months == 8)
                || ($months == 10) || ($months == 12))) {
                $days -= 31;
                $months += 1;
            }
            elsif (($days > 30) && (($months == 4) || ($months == 6)
                || ($months == 9) || ($months == 11))) {
                $days -= 30;
                $months += 1;
            }
            elsif (($days > 28) && ($months == 2)) {
                $days -= 28;
                $months += 1;
            }
        }
    }

    if ($months > 12) {
        $months -= 12;
    }

    return ($minutes, $hours, $days, $months);
}

```

J.3 Script C: Convert data to minitab 13 format

Script to generate the input file compatible with minitab 13 to analyze the data from each summary file in each directory containing the data from one experimental run.

```
#!/usr/bin/perl -w
#
# Creates file with results for Minitab
# Syntax:
#       create_minitab.pl
#

# Input/Output filenames
$infile1 = 'ord_exp';
$infile2 = 'metrics_names';
$infile3 = 'summary_output';
$outfile = 'experiment_outcome.txt';

# LABELS for METRICS
$labelmetric1 = 'SAR';
$labelmetric2 = 'IOSTAT';
$labelmetric3 = 'VMSTAT';
$labelmetric4 = 'MPSTAT';

#$noexp = 234;          # Number of experiments to process

# Open files to process initial data
open(ORDEXP, "<$infile1") || die "Could not find $infile1\n";
open(NAMES, "<$infile2") || die "Could not find $infile2\n";
open(OUTFILE, ">$outfile") || die "Could not find $outfile\n";

# Get names of metrics from input file
while(<NAMES>){
    # Do if eof not reached
    if($_ =~ /\s*$/) {next;}; # Remove blank lines
    chomp;                  # Remove newline character
    @space_sep = split;     # Split line at blank space
                           # Care with more than one trailing blank spaces!!
    if ($space_sep[0] eq $labelmetric1) {
        $names1 = <NAMES>;
        chomp($names1);
        @name1 = split(/\s+/, $names1);
    } elsif ($space_sep[0] eq $labelmetric2) {
        $names2 = <NAMES>;
        chomp($names2);
        @name2 = split(/\s+/, $names2);
    } elsif ($space_sep[0] eq $labelmetric3) {
        $names3 = <NAMES>;
```

```

        chomp($names3);
        @name3 = split(/\s+/, $names3);
    } elsif ($space_sep[0] eq $labelmetric4) {
        $names4 = <NAMES>;
        chomp($names4);
        @name4 = split(/\s+/, $names4);
        $names5 = <NAMES>;
        chomp($names5);
        @name5 = split(/\s+/, $names5);
        $names6 = <NAMES>;
        chomp($names6);
        @name6 = split(/\s+/, $names6);
        $names7 = <NAMES>;
        chomp($names7);
        @name7 = split(/\s+/, $names7);
    } else {
        die "Should not read this line\n";
    }
}
close(NAMES);

# Create a long line with all the metrics
@Metric_Names=(@name1, @name2, @name3, @name4, @name5, @name6, @name7);

@Order_of_exp = <ORDEXP>;
close(ORDEXP);

print OUTFILE "Size\tAlgorithm\tCompilerOption\tUnknowns\tPrismTime
\tCountOfMetrics";
foreach (@Metric_Names) {
    print OUTFILE "\t";
    print OUTFILE;
}
print OUTFILE "\n";

# Searches for the number of the first experiment in input file
for ($k=1; $k<=234; $k++){
    $directory = "E".$k;
#    $directory = "Etest";
    print 'Directory ' . $directory . "\n";
    chdir($directory);

    open(SUMMARY, "<$infile3") || die "Could not find $infile3\n";

    while (<SUMMARY>) {
        if($_ =~ /\s*$/) {next;};      # Remove blank lines
        chomp;                        # Remove newline character
        @space_sep = split;           # Split line at blank space
    }
}

```

```

        if ($space_sep[0] eq "Total") {
            $Unk = $space_sep[3]; # Get number of unknowns
        } elsif ($space_sep[0] eq "Prism") {
            $prism_time = $space_sep[4];
        } elsif ($space_sep[0] eq "Count") {
            $count_metrics = $space_sep[3];
        } elsif ($space_sep[0] eq "bread/s") {
            $m_sar = <SUMMARY>;
            chomp($m_sar);
            @metrics_sar = split(/\s+/, $m_sar);
        } elsif ($space_sep[0] eq "tin") {
            $m_iostat = <SUMMARY>;
            chomp($m_iostat);
            @metrics_iostat = split(/\s+/, $m_iostat);
        } elsif ($space_sep[0] eq "procs") {
            $m_vmstat = <SUMMARY>;
            $m_vmstat = <SUMMARY>;
            chomp($m_vmstat);
            @metrics_vmstat = split(/\s+/, $m_vmstat);
        } elsif ($space_sep[0] eq "CPU") {
            $m_mpstat = <SUMMARY>;
            chomp($m_mpstat);
            @metrics_mpstat0 = split(/\s+/, $m_mpstat);
            shift(@metrics_mpstat0);
            $m_mpstat = <SUMMARY>;
            chomp($m_mpstat);
            @metrics_mpstat1 = split(/\s+/, $m_mpstat);
            shift(@metrics_mpstat1);
            $m_mpstat = <SUMMARY>;
            chomp($m_mpstat);
            @metrics_mpstat2 = split(/\s+/, $m_mpstat);
            shift(@metrics_mpstat2);
            $m_mpstat = <SUMMARY>;
            chomp($m_mpstat);
            @metrics_mpstat3 = split(/\s+/, $m_mpstat);
            shift(@metrics_mpstat3);
            @metrics_mpstat = (@metrics_mpstat0, @metrics_mpstat1,
                             @metrics_mpstat2, @metrics_mpstat3);
        }
    }

    chdir("../");

# Get the description of experiment from file ord_exp line k
$line_ord=$Order_of_exp[$k-1];
@ord_elem = split(/\s+/, $line_ord);
shift(@ord_elem);
@tot_metrics = (@ord_elem, $Unk, $prism_time, $count_metrics,
                @metrics_sar, @metrics_iostat, @metrics_vmstat, @metrics_mpstat);

```

```

    $first = "yes";
    foreach (@tot_metrics) {
        if ($first eq "yes") {
            print OUTFILE;
            $first = "no";
        } else {
            print OUTFILE "\t";
            print OUTFILE;
        }
    }
    print OUTFILE "\n";
}

```

J.4 Script D: Convert data to SAS format

Script to generate the input file compatible with SAS from the file to be used by Minitab

13.

```

#!/usr/bin/perl -w
#
# Creates file with results for Minitab
# Syntax:
#     create_2filesSAS.pl
# where:
#
# By Nayda G. Santiago
# Created: 07/26/2001
# Modified: 03/25/2002

# Input/Output filenames
$infile1 = 'exp5MinitabNoZeros.txt';
$outfile1 = 'outcome1SASNoZeros.txt';
$outfile2 = 'outcome2SASNoZeros.txt';

# Open files to process initial data
open(INFILE, "<$infile1") || die "Could not find $infile1\n";
open(OUTFILE1, ">$outfile1") || die "Could not find $outfile1\n";
open(OUTFILE2, ">$outfile2") || die "Could not find $outfile2\n";

# Get first line with the names of metrics from input file
$metric_names = <INFILE>;
chomp($metric_names);
@Indnames = split(/\s+/, $metric_names);

```

```

# Get the number of columns in the array, i. e. number of metrics
$numberOfMetrics = $#Indnames+1;
if (($numberOfMetrics%2) == 0){      # Even number of metrics
    $Limit = ($numberOfMetrics/2)+2; # Add 3 since the first 6 columns
    # are common. Otherwise the 2nd file will be longer by 6
}
else {                                # Odd number of metrics
    $Limit = (($numberOfMetrics-1)/2)+2;
}
@metricsNames1=@Indnames[0...$Limit];
@metricsNames2=(@Indnames[0...4], @Indnames[$Limit+1...$#Indnames]);

$first = "yes"; # This variable is used to prevent the first element
                # from being a blank space.
foreach (@metricsNames1) {
    if ($first eq "yes") {
        print OUTFILE1;
        $first = "no";
    } else {
        print OUTFILE1 " "; # Uses blank space as separator
        print OUTFILE1; # Prints each element of $Metric_Name1
    }
}
print OUTFILE1 "\n";

$first = "yes"; # This variable is used to prevent the first element
                # from being a blank space.
foreach (@metricsNames2) {
    if ($first eq "yes") {
        print OUTFILE2;
        $first = "no";
    } else {
        print OUTFILE2 " "; # Uses blank space as separator
        print OUTFILE2; # Prints each element of $Metric_Name2
    }
}
print OUTFILE2 "\n";

# Get data and change tabs to blank spaces
while(<INFILE>){
    # Do if eof not reached
    if($_ =~ /\s*$/) {next;}; # Remove blank lines
    chomp;                    # Remove newline character
    @data_line = split;       # Split line at blank space
    # Care with more than one trailing blank spaces!!
    @data1 = @data_line[0...$Limit];
    @data2 = (@data_line[0...5],@data_line[$Limit+1...$#data_line]);

    $first = "yes"; # This variable is used to prevent the first

```

```

        # element from being a blank space.
foreach (@data1) {
    if ($first eq "yes") {
        print OUTFILE1;
        $first = "no";
    } else {
        print OUTFILE1 " "; # Uses blank space as separator
        print OUTFILE1;     # Prints each element of $data1
    }
}
print OUTFILE1 "\n";

$first = "yes"; # This variable is used to prevent the first
                # element from being a blank space.
foreach (@data2) {
    if ($first eq "yes") {
        print OUTFILE2;
        $first = "no";
    } else {
        print OUTFILE2 " "; # Uses blank space as separator
        print OUTFILE2;     # Prints each element of $data2
    }
}
print OUTFILE2 "\n";

} #End of while
close(INFILE);

```

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999.
- [2] Huan Liu and Hiroshi Motoda. *Feature Selection for Knowledge Discovery and Data Mining*. Kluwer Academic Publishers, 1998.
- [3] Marie Coffin and Matthew J. Saltzman. Statistical analysis of computational tests of algorithms and heuristics. *INFORMS Journal on Computing*, 12(1):24 – 44, Winter 2000.
- [4] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
- [5] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1993.
- [6] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356 – 372, April 1995.
- [7] Graham D. Riley and John R. Gurd. Requirement for automatic performance analysis APART. Technical Report FZJ-ZAM-IB-9919, Central Institute for Applied Mathematics, Research Centre Jülich, November 1999.
- [8] John Mellor-Crummey, Robert J. Fowler, Gabriel Marin, and Nathan Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23(1):81 – 104, August 2002.
- [9] Sam Kash Kachigan. *Statistical Analysis: An Interdisciplinary Introduction to Univariate & Multivariate Methods*. Radius Press, Inc., 1986.
- [10] R. Bruce Irvin. *Performance Measurement Tools for High-Level Parallel Programming Languages*. PhD thesis, University of Wisconsin - Madison, 1995.
- [11] M. Alabdulkareem, S. Lakshminarayanan, and S.K. Dhall. Scalability analysis of large codes using factorial designs. *Parallel Computing*, 27(9):1145 – 1171, August 2001.
- [12] Xian-He Sun, Dongmei He, Kirk W. Cameron, and Yong Luo. Adaptive multivariate regression for advanced memory system evaluation: Application and experience. *Performance and Evaluation: An International Journal*, 45(1):1 – 18, 2001.

- [13] Oleg Y. Nickolayev, Philip C. Roth, and Daniel A. Reed. Real-time statistical clustering for event trace reduction. *International Journal of Supercomputing Applications and High Performance Computing*, 11(2):144 – 159, Summer 1997.
- [14] Jeffrey S. Vetter and Daniel A. Reed. Managing performance analysis with dynamic statistical projection pursuit. In *Proceedings of Supercomputing '99*, November 1999.
- [15] Dong H. Ahn and Jeffrey S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of Supercomputing '02*, November 2002.
- [16] Thomas Fahringer, Michael Gerndt, Graham Riley, and Jesper Larsson Täff. Formalizing OpenMP performance properties with ASL. In *Proceedings of the International Workshop on OpenMP: Experiences and Implementations (WOMPEI)*, Lecture Notes in Computer Science, pages 428 – 439, Tokyo, Japan, 2000. Springer.
- [17] R. Bruce Irvin and Barton P. Miller. Mechanisms for mapping high-level parallel performance data. In *Proceedings of the 1996 International Conference on Parallel Processing Workshop on Challenges for Parallel Processing*, pages 10 – 19, August 1996.
- [18] R. Bruce Irvin and Barton P. Miller. Mapping performance data for high-level and data views of parallel program performance. In *Proceedings of the 10th ACM International Conference on Supercomputing, ICS96*, pages 69 – 77, May 1996.
- [19] Xian-He Sun and Kirk W. Cameron. A statistical-empirical hybrid approach to hierarchical memory analysis. In *Proceedings of the 6th International Euro-Par Conference*, pages 141 – 148, August 2000.
- [20] Xian-He Sun, Dongmei He, Kirk W. Cameron, and Yong Luo. A factorial performance evaluation for hierarchical memory systems. In *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, April 1999.
- [21] Thomas Fahringer, Michael Gerndt, Bernd Mohr, Felix Wolf, Graham Riley, and Jesper Larsson Täff. Knowledge specification for automatic performance analysis APART. Technical Report FZJ-ZAM-IB-2001-08, Central Institute for Applied Mathematics, Research Centre Jülich, August 2001.
- [22] Thomas Fahringer, Michael Gerndt, Graham Riley, and Jesper Larsson Täff. Specification of performance problems in MPI programs with ASL. In *Proceedings of the 2000 International Conference on Parallel Processing, ICIP'00*, Montreal, Canada, August 2000.

- [23] Thomas Fahringer, Michael Gerndt, Graham Riley, and Jesper Larsson Täff. On performance modeling for HPF applications with ASL. In *Proceedings of the International Symposium on High Performance Computing*, Lecture Notes in Computer Science, Tokyo, Japan, 2000. Springer.
- [24] Graham E. Searle, Julian W. Gardner, Michael J. Chappell, Keith R. Godfrey, and Michael J. Chapman. System identification of electronic nose data from cyanobacteria experiments. *IEEE Sensors Journal*, 2(3):218 – 229, June 2002.
- [25] Douglas C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, Inc., 1997.
- [26] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, Inc., 1991.
- [27] David J. Lilja. *Measuring Computer Performance: A practitioner's guide*. Cambridge University Press, 2000.
- [28] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, 274(1 - 2):43 – 87, March 2002.
- [29] Jennifer G. Dy and Carla E. Brodley. Feature subset selection and order identification for unsupervised learning. In *Proceedings of the 17th International Conference on Machine Learning*, June-July 2000.
- [30] Anil Jain and Douglas Zongker. Feature selection: Evaluation, application, and small sample performance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(2):153 – 158, February 1997.
- [31] Karl W. Pettis, Thomas A. Bailey, Anil K. Jain, and Richard C. Dubes. An intrinsic dimensionality estimator from near-neighbor information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(1):25 – 37, January 1979.
- [32] Neal Wyse, Richard Dubes, and Anil K. Jain. *Pattern Recognition in Practice*, chapter A Critical Evaluation of Intrinsic Dimensionality Algorithms, pages 415 – 425. North-Holland Publishing Company, 1980.
- [33] Moshe F. Rubinstein, editor. *Patterns of Problem Solving*. Prentice-Hall, Inc., 1975.
- [34] Harry I. Forsha, editor. *Show Me: the Complete Guide to Storyboarding and Problem Solving*. ASQC Quality Press, 1995.
- [35] C. T. H. Everaars, F. Arbab, and F. J. Burger. Restructuring sequential fortran code into a parallel/distributed application. In *Proceedings of the International Conference on Software Maintenance 1996*, pages 13 – 22, November 1996.

- [36] John L. Volakis and Leo C. Kempel. Electromagnetics: Computational methods and considerations. *IEEE Computational Science and Engineering*, 2(1):42 – 57, Spring 1995.
- [37] R. Nilavalan, I. J. Craddock, D. L. Paul, and C. J. Railton. Conformal antenna array modeling using a locally nonorthogonal FDTD. *Microwave and Optical Technology Letters*, 30(4):238 – 240, August 2001.
- [38] Leo C. Kempel and John L. Volakis. A finite element-boundary integral method for cavities in a circular cylinder. In *Proceedings of the 1993 IEEE Antennas and Propagation Society Symposium*, pages 292 – 295, June 1993.
- [39] Jr. Richard C. Booton. *Computational Methods for Electromagnetics and Microwaves*. John Wiley & Sons, Inc., 1992.
- [40] Kosmo D. Tatalias and James M. Bornholdt. Mapping electromagnetic field computations to parallel processors. *IEEE Transactions on Magnetics*, 25(4):2901 – 2906, July 1989.
- [41] Ali R. Baghai-Wadji. An introduction to the fast-MoM in computational electromagnetics. In *IEEE 6th Topical Meeting on Electrical Performance in Electronic Packaging*, page 231, October 1997.
- [42] John L. Volakis, Arindam Chatterjee, and Leo C. Kempel. *Finite Element Method for Electromagnetics: Antennas, Microwave Circuits, and Scattering Applications*. IEEE Press, 1998.
- [43] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial*. Siam, second edition, 2000.
- [44] John L. Volakis, Tayfun Özdemir, and Jian Gong. Hybrid finite-element methodologies for antennas and scattering. *IEEE Transactions on Antennas and Propagation*, 45(3):493 – 507, March 1997.
- [45] Young W. Kwon and Hyochoong Bang. *The Finite Element Method Using MATLAB*. CRC Press, second edition, 2000.
- [46] Leo C. Kempel. Implementation of various hybrid finite element-boundary integral methods: Bricks, prisms, and tets. In *Proceedings of the 1999 ACES Meeting*, pages 242 – 249, 1999.
- [47] Gary Goldman and Partha Tirumalai. UltraSPARC-IITM: The advancement of UltraComputing. In *Proceedings of the 41st IEEE Computer Society International Conference (CompCon '96): Technologies for the Information Superhighway*, pages 417 – 423, Santa Clara, CA, 1996.

- [48] UltraSPARCTM-II data sheet, July 1997. Sun Microsystems.
- [49] The UltraTM 450 workstation architecture. Technical White Paper, 1998. Sun Microsystems.
- [50] Adrian Cockcroft and Richard Pettit. *Sun Performance and Tuning: Java and the Internet*. Sun Microsystems Press, second edition, 1998.
- [51] Jim Mauro and Richard McDougall. *Solaris Internals: Core Kernel Components*. Sun Microsystems Press, 2001.
- [52] Linda Trocine and Linda C. Malone. Finding important independent variables through screening designs: A comparison of methods. In *Proceedings of the 2000 Winter Simulation Conference*, volume 1, pages 749 – 754, December 2000.
- [53] Linda Trocine and Linda C. Malone. An overview of newer, advanced screening methods for the initial phase in an experimental design. In *Proceedings of the 2001 Winter Simulation Conference*, volume 1, pages 169 – 178, December 2001.
- [54] Bernd Mohr. Design of automatic performance analysis systems: APART technical report. Technical Report Draft Report, <http://www.fz-juelich.de/apart-1/wp3/index.html>, Central Institute for Applied Mathematics, Research Centre Jülich, May 2000.
- [55] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The John Hopkins University Press, 1989.
- [56] Abdul Waheed, Diane T. Rover, and Jeffrey K. Hollingsworth. Modeling and evaluating design alternatives of an on-line instrumentation system: A case study. *IEEE Transactions on Software Engineering*, 24(6):451 – 470, June 1998.
- [57] Jack Dongarra, Allen Malony, Shirley Moore, Philip Mucci, and Sameer Shende. Performance instrumentation and measurement for terascale systems. In *Proceedings of the International Conference on Computational Science, ICCS 2003, Terascale Performance Analysis Workshop*, June 2003.
- [58] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A message passing standard for mpp and workstations. *Communications of the ACM*, 39(7):84 – 90, July 1996.
- [59] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. *SIGPLAN Notices*, 29(6):196 – 205, June 1994.
- [60] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317 – 329, Winter 2000.

- [61] Inc. Kuck & Associates. *Guide Reference Manual, Version 3.9*, March 2000.
- [62] Sanford Weisberg. *Applied Linear Regression*. John Wiley & Sons, Inc., 1980.
- [63] Rolf Sundberg. *Encyclopedia of Environmetrics*, chapter Collinearity, pages 365 – 366. John Wiley & Sons, Inc., 2002.
- [64] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37 – 54, Fall 1996.
- [65] Ricardo Gutierrez-Osuna and H. Troy Nagle. A method for evaluating data-preprocessing techniques for odor classification with an array of gas sensors. *IEEE Transactions on Systems, Man, and Cybernetics—Part B: Cybernetics*, 29(5):626 – 632, October 1999.
- [66] Anup Mathur. *A Stochastic Process Model for Transient Trace Data*. PhD thesis, Virginia Polytechnic Institute and State University, 1996.
- [67] Jr. John R. Deller, John G. Proakis, and John H. L. Hansen. *Discrete-Time Processing of Speech Signal*. Macmillan Publishing Company, first edition, 1993.
- [68] William R. Dillon and Matthew Goldstein. *Multivariate Analysis: Methods and Applications*. John Wiley, 1984.
- [69] Trevor Hastie, Robert Tibshirani, and Jerome Friedman, editors. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag, 2001.
- [70] Ruby L. Kennedy, Yuchun Lee, Benjamin Van Roy, Christopher D. Reed, and Richard P. Lippmann, editors. *Solving Data Mining Problems Through Pattern Recognition*. Prentice Hall, PRT, 1997.
- [71] Ricardo Gutierrez-Osuna. Pattern analysis for machine olfaction: A review. *IEEE Sensors Journal*, 2(3):189 – 202, June 2002.
- [72] R. Gutierrez-Osuna, T. Nagle, B. Kermani, and S. Schiffman. *Handbook of Machine Olfaction: Electronic Nose Technology*, chapter 7: Signal Conditioning and Pre-processing. Wiley - VCH, 2002.
- [73] Pierre A. Devijver and Josef Kittler, editors. *Pattern Recognition: A Statistical Approach*. Prentice Hall International, 1982.
- [74] Huan Liu, Hongjun Lu, and Lei Yu. Active sampling: An effective approach to feature selection. In *SIAM International Conference on Data Mining*, May 2003.
- [75] Avrim L. Blum and Pat Langley. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97(1 - 2):245 – 271, December 1997.

- [76] Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1 - 2):273 – 324, December 1997.
- [77] Jennifer G. Dy and Carla E. Brodley. Visualization and interactive feature selection for unsupervised data. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 360 – 364, August 2000.
- [78] M. Dash and H. Liu. Feature selection for classification. *Intelligent Data Analysis*, 1(3):131 – 156, 1997.
- [79] Douglas Zongker and Anil Jain. Algorithms for feature selection: An evaluation. In *Proceedings of the 13th International Conference on Pattern Recognition*, pages 18 – 22, 1996.
- [80] Manoranjan Dash, Kiseok Choi, Peter Scheuermann, and Huan Liu. Feature selection for clustering - a filter solution. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, December 2002.
- [81] M. Dash, H. Liu, and J. Yao. Dimensionality reduction of unsupervised data. In *Proceedings of the 9th International Conference on Tools with Artificial Intelligence*, pages 532 – 539, November 1997.
- [82] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379 – 656 and 623 – 656, July, October 1948.
- [83] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc., 1991.
- [84] Robert S. Bennett. The intrinsic dimensionality of signal collections. *IEEE Transactions on Information Theory*, IT-15:517 – 525, September 1969.
- [85] Keinosuke Fukunaga. *Handbook of Statistics*, volume 2, chapter Intrinsic Dimensionality Extraction, pages 347 – 360. North-Holland Publishing Company, 1982.
- [86] Michael Kirby. *Geometric Data Analysis: An Empirical Approach to Dimensionality Reduction and the Study of Patterns*. John Wiley & Sons, Inc., 2001.
- [87] I. T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, Inc., 1986.
- [88] A. Ralph Hakstian, W. Todd Rogers, and Raymond B. Cattell. The behavior of number-of-factors rules with simulated data. *Multivariate Behavioral Research*, 17:193 – 219, April 1982.
- [89] Jennifer G. Dy, Carla E. Brodley, Avi Kak, Lynn S. Broderick, and Alex M. Aisen. Unsupervised feature selection applied to content-based retrieval of lung images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(3):373 – 378, March 2003.

- [90] Miguel Vélez-Reyes and Luis O. Jiménez. Subset selection analysis for the reduction of hyperspectral imagery. In *Proceedings of the Geoscience and Remote Sensing Symposium, IGARRS '98*, pages 1577 – 1581 Vol. 3, 1998.
- [91] Charng da Lu and Daniel A. Reed. Compact application signatures for parallel and distributed scientific codes. In *Proceedings of Supercomputing '02*, November 2002.
- [92] Antonio Espinosa, Tomás Margalef, and Emilio Luque. Automatic performance evaluation of parallel programs. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing, PDP '98*, pages 43 – 49, January 1998.
- [93] M. Gerndt, B. Mohr, M. Pantano, and F. Wolf. Performance analysis on CRAY T3E. In *Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing, PDP '99*, pages 241 – 248, February 1999.
- [94] Domingo Rodríguez, Nayda G. Santiago, and Carlos Vélez. Implementation of a new class of FFT algorithms on transputer computational structures. In *Proceedings of the 36th Midwest Symposium on Circuits and Systems*, volume 2, pages 1105 – 1108, 1993.
- [95] Nayda G. Santiago, Diane T. Rover, and Domingo Rodríguez. A statistical approach for the analysis of the relation between low-level performance information, the code, and the environment. In *Proceedings of the International Conference on Parallel Processing Workshops, HPSECA 02*, pages 282 – 289, August 2002.
- [96] Nayda G. Santiago, Diane T. Rover, and Domingo Rodríguez. A statistical approach for the analysis of the relation between low-level performance information, the code, and the environment. To appear, *Journal Parallel and Distributed Computing Practice*.
- [97] Nayda G. Santiago, Diane T. Rover, and Domingo Rodríguez. Subset selection of performance metrics describing system-software interactions. *Supercomputing 2002, SC'02*. Poster.
- [98] Steven G. Krantz. *Real Analysis and Foundations*. CRC Press, Inc., 1991.
- [99] Henry Stark and John W. Woods. *Probability, Random Processes, and Estimation Theory for Engineers*. Prentice-Hall, Inc., second edition, 1994.
- [100] Richard Barret, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1993.
- [101] Alan Jennings and J.J. McKeown. *Matrix Computation*. John Wiley & Sons, Inc., second edition, 1992.

- [102] Michael W. Frazier. *An Introduction to Wavelets Through Linear Algebra*. Springer, 1999.
- [103] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 1990.
- [104] Gene Golub and James M. Ortega. *Scientific Computing: An Introduction with Parallel Computing*. Academic Press, Inc., 1993.
- [105] Paul Messina, David Culler, Wayne Pfeiffer, William Martin, J. Tinsley Oden, and Gary Smith. Architecture. *Communications of the ACM*, 41(11):36 – 44, November 1998.
- [106] Chris Herring. Microprocessors, microcontrollers, and systems in the new millennium. *IEEE Micro*, 20(6):45 – 51, December 2000.
- [107] Michael Slater. The microprocessor today. In Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, editors, *Readings in Computer Architecture*, pages 668 – 680. Morgan Kaufmann Publishers, 2000.
- [108] Bruce Greer, John Harrison, Grep Henry, Wei Li, and Peter Tang. Scientific computing on the itaniumTM processor. In *Proceedings of Supercomputing '01*, November 2001.
- [109] INTEL. *Intel Itanium 2 Processor Reference Manual For Software Development and Optimization*, April 2003. Order Number 251110-002.
- [110] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter Computational Grids. Morgan Kaufmann Publishers, Inc., 1999.
- [111] Matthew Shields, Omer F. Rana, David W. Walker, and David Golby. A collaborative code development environment for computational electro-magnetics. In *Proceedings of the 8th Working Conference on Software Architectures for Scientific Computing*, pages 119 – 141, October 2001.
- [112] Jack Dongarra and David W. Walker. The quest for petascale computing. *Computing in Science and Engineering*, 3(3):32 – 39, May/June 2001.
- [113] Abdul Waheed and Diane T. Rover. *Modeling and Simulation of Advanced Computer Systems*, chapter Instrumentation Systems for Parallel Tools, pages 35 – 54. Gordon and Breach Publishers, Inc., 1996.
- [114] Jeffrey K. Hollingsworth and Bart Miller. *The Grid: Blueprint for a New Computing Infrastructure*, chapter Instrumentation and Measurement. Morgan Kaufmann Publishers, Inc., 1999.

- [115] Daniel A. Reed. *Models and Techniques for Performance Evaluation of Computer and Communication Systems.*, chapter Performance Instrumentation Techniques for Parallel Systems, pages 463 – 490. Springer-Verlag Lecture Notes in Computer Science, 1993.
- [116] Jonathan Geisler and Valerie Taylor. *Performance Evaluation and Benchmarking with Realistic Applications.* Rudolf Eigenmann, editor, chapter Performance Coupling: Case Studies for Measuring the Interactions of Kernels in Modern Applications. MIT Press, 2001.
- [117] Jeffrey Brown, Al Geist, Cherri Pancake, and Diane Rover. Software tools for developing parallel applications. part 1: Code development and debugging. In *SIAM Conference on Parallel Processing for Scientific Computing Proceedings*, March 1997.
- [118] Jeffrey Brown, Al Geist, Cherri Pancake, and Diane Rover. Software tools for developing parallel applications. part 2: Interactive control and performance tuning. In *SIAM Conference on Parallel Processing for Scientific Computing Proceedings*, March 1997.
- [119] Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley Publishing Company, Inc., 1995.
- [120] Michael T. Heath, Allen D. Malony, and Diane T. Rover. The visual display of parallel performance data. *Computer*, 28(11):21 – 28, November 1995.
- [121] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29 – 39, September 1991.
- [122] Jerry C. Yan and Sekhar R. Sarukkai. Analyzing parallel program performance using normalized performance indices and trace transformation techniques. *Parallel Computing*, 22(9):1215 – 1237, November 1996.
- [123] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. *An Overview of the Pablo Performance Analysis Environment.* Department of Computer Science, University of Illinois - Urbana, November 1992. Pablo Documentation.
- [124] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37 – 46, November 1995.
- [125] Hong-Linh Truon and Thomas Fahringer. SCALEA: A performance analysis tool for distributed and parallel programs. In *Proceedings of the 8th International Euro-Par Conference, EUROPAR 2002*, August 2002.

- [126] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences*. Duxbury Thomson Learning, 2000.
- [127] Ronald P. Cody and Jeffrey K. Smith. *Applied Statistics and the SAS Programming Language*. Prentice Hall, fourth edition, 1997.
- [128] Robert G. D. Steel, James H. Torrie, and David A. Dickey. *Principles and Procedures of Statistics: A Biometrical Approach*. The McGraw-Hill Companies, Inc., third edition, 1997.
- [129] Paul R. Cohen. *Empirical Methods for Artificial Intelligence*. The MIT Press, 1995.
- [130] Rudolf J. Freund and William J. Wilson. *Statistical Methods*. Academic Press, Inc., 1993.
- [131] Stephen A. Ward and Jr. Robert H. Halstead. *Computation Structures*. The MIT Press and the McGraw-Hill Book Company, 1990.