

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Performance Prediction and Scheduling for
Parallel Applications on Multi-User Clusters

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Jennifer Melinda Schopf

Committee in charge:

Professor Francine Berman, Chairperson
Professor Larry Carter
Professor Mark Ellisman
Professor William G. Griswold
Professor Peter Taylor

1998

Copyright

Jennifer Melinda Schopf, 1998

All rights reserved.

The dissertation of Jennifer Melinda Schopf is approved,
and it is acceptable in quality and form for publication
on microfilm:

Chair

University of California, San Diego

1998

TABLE OF CONTENTS

	Signature Page	iii
	Table of Contents	iv
	List of Figures	vii
	List of Tables	xvii
	Acknowledgments	xx
	Vita, Publications, and Fields of Study	xxii
	Abstract	xxiii
1	Introduction	1
	A. High Performance Distributed Parallel Computing	3
	B. Performance Prediction Models	4
	C. Point Values and Stochastic Values	7
	D. Stochastic Scheduling	9
	E. Summary	12
	F. Dissertation Outline	12
2	Experimental Environment	13
	A. Experimental Platforms	13
	B. Workload Characterization	15
	C. Applications	16
	D. Master-Slave Applications	18
	1. Genetic Algorithm Application	21
	2. High Temperature Superconductor Kernel	21
	3. N-body Code	23
	E. Regular SPMD Applications	24
	1. Successive Over-Relaxation Code	25
	2. NPB LU Benchmark	25
	3. SP Benchmark	27
	4. EP Kernel	28
3	Structural Modeling	31
	A. Definitions	32
	1. Accuracy	34
	2. Application Profile	35
	3. Building Structural Models	36
	4. Practical Issues	36

B.	Modeling the Genetic Algorithm Code	42
1.	GA Recap	42
2.	Top-level Model for GA	43
3.	Modeling GA Computation Components	46
4.	Modeling GA Communication Components	47
5.	GA Application on the PCL Cluster	50
6.	GA on Linux Cluster	57
7.	GA on Alpha Farm Cluster	59
8.	Discussion	60
C.	Modeling the SOR Benchmark	63
1.	Top-level Model for SOR	63
2.	Component Models for SOR	65
3.	SOR on the PCL Cluster	67
4.	SOR on Linux Cluster and SDSC Alpha Farm	69
5.	Discussion	71
D.	Other Applications	75
1.	Modeling the N-body Code	75
2.	Modeling the Hi-Temp Code	81
3.	Modeling the LU Benchmark	87
4.	Modeling the SP Benchmark	92
5.	Modeling the EP Benchmark	94
6.	Discussion	97
E.	Related Work	97
1.	Models for Parallel Applications	97
2.	Compositional Approaches	98
3.	Component Models	99
4.	Related Petri Net Work	101
5.	Application Profile	101
6.	Model Selection	102
F.	Summary	103
4	Stochastic Values and Predictions	105
A.	Motivation	106
1.	Statistics and Notation	108
2.	Outline	109
B.	Defining Stochastic Values using Normal Distributions	110
1.	Time Frame	111
2.	Arithmetic Operations over Normal Distributions	112
3.	Using Normal Distributions for Non-normal Characteristics	117
4.	Discussion	121
C.	Representing Stochastic Values using Intervals	123

	1. Defining Intervals	124
	2. Arithmetic over Intervals	124
	3. Discussion	126
	D. Representing Stochastic Values using Histograms	128
	1. Defining Histograms	128
	2. Arithmetic over Histograms	130
	3. Discussion	134
	E. Comparison of Stochastic Representations	135
	F. Experimental Verification	135
	1. SOR on PCL	136
	2. GA on PCL	142
	3. NBody on PCL	151
	4. LU on PCL	163
	5. Summary	169
	G. Related Work	169
	H. Summary	171
5	Stochastic Scheduling	173
	A. Scheduling Overview	173
	1. Motivation	174
	2. Assumptions and Outline	176
	B. Stochastic Time Balancing	177
	1. Using Stochastic Information	178
	2. The Tuning Factor	179
	C. Stochastic Scheduling Experiments	184
	1. Performance Metrics	185
	2. SOR on Linux Cluster	187
	3. SOR on PCL Cluster	196
	4. GA on Linux Cluster	202
	5. GA on PCL	209
	6. N-body Code on Linux Cluster	218
	7. Summary	227
	D. Related Work	227
	1. Scheduling	227
	2. Data Allocation	227
	3. Other Related Approaches	228
	E. Summary	229
6	Conclusion	231
	A. Critique and Extensions	231
	B. Conclusions and Contributions	233
	Bibliography	235

LIST OF FIGURES

2.1	Graph of one iteration of an Assign Master-Slave application.	19
2.2	Pseudocode for an Assign Master-Slave application.	19
2.3	Graph of one iteration of a Request Master-Slave application.	20
2.4	Pseudocode for a Request Master-Slave application.	20
2.5	Pseudocode for GA Application	22
2.6	Pseudocode for HiTemp kernel	23
2.7	Pseudocode for the N^2 N-body Code.	24
2.8	Strip decomposition for Red-Black SOR	25
2.9	Pseudocode for the SOR benchmark.	26
2.10	Wavefront execution of the LU benchmark, shown for a single z plane, labeled with order of execution possible.	27
3.1	Pseudocode for the Genetic Algorithm application.	42
3.2	Graphic of the top-level model in Equation 3.1 for the Genetic Algorithm Master-Slave application. The lefthand-side is a dependency graph of the application, and the righthand-side is a graphic for the structural model, with the long bar representing the top-level model, and each of the shaded pieces on the far right representing individual component models.	45
3.3	Graphic of model options for Genetic Algorithm code. The long white bar represents the top-level model, and each of the patterned components on the right represent possible component models. There are multiple component models available for each task defined in the top-level model. Two of the component models for both the Scatter and Gather include an additional component model for PtToPt . Each of these models can be selected independently.	51
3.4	Actual times versus model times for GA Code on PCL platform using operation count and benchmark models when communication is set to zero.	54
3.5	Actual times versus model times for GA Code on Linux cluster platform using benchmark slave model with and without communication models set to zero.	59
3.6	Actual times versus model times for GA Code on Alpha Farm platform using benchmark models where Scatter =0 and Gather = 0.	61
3.7	Skew can be generated by un-balanced processors sharing data. In this figure, the dashed lines indicated message wait time in the message queue. Note that as P1 affects P2. then P2's wait times affect the start times for P3.	65
3.8	Graph of SOR on dedicated PCL Cluster showing results of actual execution times and models. Note that this graph is on a log-linear scale to show the error in the highest and lowest problem sizes.	70

3.9	Actual times versus benchmark and operation count model times for SOR benchmark on dedicated Linux cluster.	73
3.10	Actual times versus model times for SOR Code on Alpha Farm platform using benchmark model and operation count model (communication = 0).	74
3.11	Graph showing benchmark and operation count models without communication, and actual values and means for N-Body code on PCL Cluster.	78
3.12	Graph showing benchmark and operation count models with communication models, with actual values and means for N-Body code on PCL Cluster.	79
3.13	Graph of N-body application on Linux cluster, modeled versus actual execution times.	80
3.14	Graph of actual versus modeled execution times for Hi-Temp code on Linux cluster.	85
3.15	Graph showing results of benchmark model and actual execution times for Hi-Temperature code on PCL cluster.	86
3.16	Modeled versus actual execution times for LU benchmark on Linux Cluster. Note this is a log-linear graph.	90
3.17	Graph of an simple model for LU on PCL cluster and actual times. Note that this graph is on a log-linear scale.	91
3.18	Actual times versus modeled times for SP on Alpha Farm.	93
3.19	Actual times versus modeled times for EP on Alpha Farm. Note log-linear graph.	95
3.20	Actual times versus modeled times for EP on Linux cluster. Note log-linear graph.	96
4.1	Stochastic predictions are created by parameterizing structural models with stochastic values.	107
4.2	Graphs showing the pdf and cdf of SOR benchmark with normal distribution based on data mean and standard deviation.	111
4.3	Graphs showing the pdf and cdf for bandwidth between two workstations over 10Mbits ethernet with long-tailed distribution and corresponding normal distribution.	118
4.4	Graph showing histogram of dedicated behavior of GA code for problem size 800 on a Sparc 10.	119
4.5	Available CPU on a production workstation.	120
4.6	Time series showing temporal locality of CPU data.	121
4.7	Time series showing non-temporal locality of CPU data.	122
4.8	Four sample stochastic values, each with the same interval representation.	127
4.9	Graph showing example of a histogram.	129
4.10	Histograms for sample parameters D and E	132
4.11	Histogram of intermediate results for D + E	132

4.12	Result of D + E using Lüthi’s pdf construction techniques.	133
4.13	Comparison of point value prediction and stochastic value prediction represented by normal distributions for the SOR benchmark on the PCL cluster with CPU loads shown in Figures 4.15 through 4.18.	136
4.14	Comparison of point value prediction and stochastic value prediction represented by intervals for the SOR benchmark on the PCL cluster with CPU loads shown in Figures 4.15 through 4.18.	137
4.15	CPU values during runtime for prediction experiments depicted in Figure 4.13 and Figure 4.14 on Lorax, in the PCL cluster.	138
4.16	CPU values during runtime for prediction experiments depicted in Figure 4.13 and Figure 4.14 on Picard, in the PCL cluster.	138
4.17	CPU values during runtime for prediction experiments depicted in Figure 4.13 and Figure 4.14 on Thing1, in the PCL cluster.	138
4.18	CPU values during runtime for prediction experiments depicted in Figure 4.13 and Figure 4.14 on Thing2, in the PCL cluster.	138
4.19	Comparison of point value prediction and stochastic value prediction represented by normal distributions for the SOR benchmark on the PCL cluster with CPU loads shown in Figures 4.21 through 4.24.	139
4.20	Comparison of point value prediction and stochastic value prediction represented by intervals for the SOR benchmark on the PCL cluster with CPU loads shown in Figures 4.21 through 4.24.	140
4.21	CPU values during runtime for prediction experiments depicted in Figure 4.19 and Figure 4.20 on Lorax, in the PCL cluster.	141
4.22	CPU values during runtime for prediction experiments depicted in Figure 4.19 and Figure 4.20 on Picard, in the PCL cluster.	141
4.23	CPU values during runtime for prediction experiments depicted in Figure 4.19 and Figure 4.20 on Thing1, in the PCL cluster.	141
4.24	CPU values during runtime for prediction experiments depicted in Figure 4.19 and Figure 4.20 on Thing2, in the PCL cluster.	141
4.25	Comparison of point value prediction and stochastic value prediction represented by normal distributions for the GA benchmark on the PCL cluster with CPU loads shown in Figures 4.27 through 4.30.	142
4.26	Comparison of point value prediction and stochastic value prediction represented by intervals for the GA benchmark on the PCL cluster with CPU loads shown in Figures 4.27 through 4.30.	143
4.27	CPU values during runtime for prediction experiments depicted in Figure 4.25 and Figure 4.26 on Lorax, in the PCL cluster.	144
4.28	CPU values during runtime for prediction experiments depicted in Figure 4.25 and Figure 4.26 on Picard, in the PCL cluster.	144
4.29	CPU values during runtime for prediction experiments depicted in Figure 4.25 and Figure 4.26 on Thing1, in the PCL cluster.	144
4.30	CPU values during runtime for prediction experiments depicted in Figure 4.25 and Figure 4.26 on Thing2, in the PCL cluster.	144

4.31	Comparison of point value prediction and stochastic value prediction represented by normal distributions for the GA benchmark on the PCL cluster with CPU loads shown in Figures 4.33 through 4.36.	145
4.32	Comparison of point value prediction and stochastic value prediction represented by intervals for the GA benchmark on the PCL cluster with CPU loads shown in Figures 4.33 through 4.36.	146
4.33	CPU values during runtime for prediction experiments depicted in Figure 4.31 and Figure 4.32 on Lorax, in the PCL cluster.	147
4.34	CPU values during runtime for prediction experiments depicted in Figure 4.31 and Figure 4.32 on Picard, in the PCL cluster.	147
4.35	CPU values during runtime for prediction experiments depicted in Figure 4.31 and Figure 4.32 on Thing1, in the PCL cluster.	147
4.36	CPU values during runtime for prediction experiments depicted in Figure 4.31 and Figure 4.32 on Thing2, in the PCL cluster.	147
4.37	Comparison of point value prediction and stochastic value prediction represented by normal distributions for the GA benchmark on the PCL cluster with CPU loads shown in Figures 4.39 through 4.42.	148
4.38	Comparison of point value prediction and stochastic value prediction represented by intervals for the GA benchmark on the PCL cluster with CPU loads shown in Figures 4.39 through 4.42.	149
4.39	CPU values during runtime for prediction experiments depicted in Figure 4.37 and Figure 4.38 on Lorax, in the PCL cluster.	150
4.40	CPU values during runtime for prediction experiments depicted in Figure 4.37 and Figure 4.38 on Picard, in the PCL cluster.	150
4.41	CPU values during runtime for prediction experiments depicted in Figure 4.37 and Figure 4.38 on Thing1, in the PCL cluster.	150
4.42	CPU values during runtime for prediction experiments depicted in Figure 4.37 and Figure 4.38 on Thing2, in the PCL cluster.	150
4.43	Comparison of point value prediction and stochastic value prediction represented by normal distributions for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.45 through 4.48.	151
4.44	Comparison of point value prediction and stochastic value prediction represented by intervals for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.45 through 4.48.	152
4.45	CPU values during runtime for prediction experiments depicted in Figure 4.43 and Figure 4.44 on Lorax, in the PCL cluster.	153
4.46	CPU values during runtime for prediction experiments depicted in Figure 4.43 and Figure 4.44 on Picard, in the PCL cluster.	153
4.47	CPU values during runtime for prediction experiments depicted in Figure 4.43 and Figure 4.44 on Thing1, in the PCL cluster.	153
4.48	CPU values during runtime for prediction experiments depicted in Figure 4.43 and Figure 4.44 on Thing2, in the PCL cluster.	153

4.49	Comparison of point value prediction and stochastic value prediction represented by normal distributions for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.51 through 4.54.	154
4.50	Comparison of point value prediction and stochastic value prediction represented by intervals for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.51 through 4.54.	155
4.51	CPU values during runtime for prediction experiments depicted in Figure 4.49 and Figure 4.50 on Lorax, in the PCL cluster.	156
4.52	CPU values during runtime for prediction experiments depicted in Figure 4.49 and Figure 4.50 on Picard, in the PCL cluster.	156
4.53	CPU values during runtime for prediction experiments depicted in Figure 4.49 and Figure 4.50 on Thing1, in the PCL cluster.	156
4.54	CPU values during runtime for prediction experiments depicted in Figure 4.49 and Figure 4.50 on Thing2, in the PCL cluster.	156
4.55	Comparison of point value prediction and stochastic value prediction represented by normal distributions for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.57 through 4.60.	157
4.56	Comparison of point value prediction and stochastic value prediction represented by intervals for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.57 through 4.60.	158
4.57	CPU values during runtime for prediction experiments depicted in Figure 4.55 and Figure 4.56 on Lorax, in the PCL cluster.	159
4.58	CPU values during runtime for prediction experiments depicted in Figure 4.55 and Figure 4.56 on Picard, in the PCL cluster.	159
4.59	CPU values during runtime for prediction experiments depicted in Figure 4.55 and Figure 4.56 on Thing1, in the PCL cluster.	159
4.60	CPU values during runtime for prediction experiments depicted in Figure 4.55 and Figure 4.56 on Thing2, in the PCL cluster.	159
4.61	Comparison of point value prediction and stochastic value prediction represented by normal distributions for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.63 through 4.66.	160
4.62	Comparison of point value prediction and stochastic value prediction represented by intervals for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.63 through 4.66.	161
4.63	CPU values during runtime for prediction experiments depicted in Figure 4.61 and Figure 4.62 on Lorax, in the PCL cluster.	162
4.64	CPU values during runtime for prediction experiments depicted in Figure 4.61 and Figure 4.62 on Picard, in the PCL cluster.	162
4.65	CPU values during runtime for prediction experiments depicted in Figure 4.61 and Figure 4.62 on Thing1, in the PCL cluster.	162
4.66	CPU values during runtime for prediction experiments depicted in Figure 4.61 and Figure 4.62 on Thing2, in the PCL cluster.	162

4.67	Comparison of point value prediction and stochastic value prediction represented by normal distributions for the LU benchmark on the PCL cluster with CPU loads shown in Figures 4.69 through 4.72.	163
4.68	Comparison of point value prediction and stochastic value prediction represented by intervals for the LU benchmark on the PCL cluster with CPU loads shown in Figures 4.69 through 4.72.	164
4.69	CPU values during runtime for prediction experiments depicted in Figure 4.67 and Figure 4.68 on Lorax, in the PCL cluster.	165
4.70	CPU values during runtime for prediction experiments depicted in Figure 4.67 and Figure 4.68 on Picard, in the PCL cluster.	165
4.71	CPU values during runtime for prediction experiments depicted in Figure 4.67 and Figure 4.68 on Thing1, in the PCL cluster.	165
4.72	CPU values during runtime for prediction experiments depicted in Figure 4.67 and Figure 4.68 on Thing2, in the PCL cluster.	165
4.73	Comparison of point value prediction and stochastic value prediction represented by normal distributions for the LU benchmark on the PCL cluster with CPU loads shown in Figures 4.75 through 4.78.	166
4.74	Comparison of point value prediction and stochastic value prediction represented by intervals for the LU benchmark on the PCL cluster with CPU loads shown in Figures 4.75 through 4.78.	167
4.75	CPU values during runtime for prediction experiments depicted in Figure 4.73 and Figure 4.74 on Lorax, in the PCL cluster.	168
4.76	CPU values during runtime for prediction experiments depicted in Figure 4.73 and Figure 4.74 on Picard, in the PCL cluster.	168
4.77	CPU values during runtime for prediction experiments depicted in Figure 4.73 and Figure 4.74 on Thing1, in the PCL cluster.	168
4.78	CPU values during runtime for prediction experiments depicted in Figure 4.73 and Figure 4.74 on Thing2, in the PCL cluster.	168
5.1	Possible work allocation schemes and associated scheduling policies.	176
5.2	Diagram depicting the relative values of Tuning Factors for different configurations of power and variability.	181
5.3	Algorithm to Compute Tuning Factor.	182
5.4	Comparison of Mean , VTF , and 95TF policies, for the SOR benchmark on the Linux cluster with CPU loads shown in Figures 5.5 through 5.8.	188
5.5	CPU values during runtime for scheduling experiments depicted in Figure 5.4 on Mystere, in the Linux cluster.	189
5.6	CPU values during runtime for scheduling experiments depicted in Figure 5.4 on Quidam, in the Linux cluster.	189
5.7	CPU values during runtime for scheduling experiments depicted in Figure 5.4 on Saltimbanco, in the Linux cluster.	189
5.8	CPU values during runtime for scheduling experiments depicted in Figure 5.4 on Soleil, in the Linux cluster.	189

5.9	Comparison of Mean , VTF , and 95TF policies, for the SOR benchmark on the Linux cluster with CPU loads shown in Figures 5.10 through 5.13.	191
5.10	CPU values during runtime for scheduling experiments depicted in Figure 5.9 on Mystere, in the Linux cluster.	192
5.11	CPU values during runtime for scheduling experiments depicted in Figure 5.9 on Quidam, in the Linux cluster.	192
5.12	CPU values during runtime for scheduling experiments depicted in Figure 5.9 on Saltimbanco, in the Linux cluster.	192
5.13	CPU values during runtime for scheduling experiments depicted in Figure 5.9 on Soleil, in the Linux cluster.	192
5.14	Comparison of Mean , VTF , and 95TF policies, for the SOR benchmark on the Linux cluster with CPU loads shown in Figures 5.15 through 5.18.	194
5.15	CPU values during runtime for scheduling experiments depicted in Figure 5.14 on Mystere, in the Linux cluster.	195
5.16	CPU values during runtime for scheduling experiments depicted in Figure 5.14 on Quidam, in the Linux cluster.	195
5.17	CPU values during runtime for scheduling experiments depicted in Figure 5.14 on Saltimbanco, in the Linux cluster.	195
5.18	CPU values during runtime for scheduling experiments depicted in Figure 5.14 on Soleil, in the Linux cluster.	195
5.19	Comparison of Mean , VTF , and 95TF policies, for the SOR benchmark on the Linux cluster with CPU loads shown in Figures 5.20 through 5.23.	197
5.20	CPU values during runtime for scheduling experiments depicted in Figure 5.19 on Lorax, in the PCL cluster.	198
5.21	CPU values during runtime for scheduling experiments depicted in Figure 5.19 on Picard, in the PCL cluster.	198
5.22	CPU values during runtime for scheduling experiments depicted in Figure 5.19 on Thing1, in the PCL cluster.	198
5.23	CPU values during runtime for scheduling experiments depicted in Figure 5.19 on Thing2, in the PCL cluster.	198
5.24	Comparison of Mean , VTF , and 95TF policies, for the SOR benchmark on the Linux cluster with CPU loads shown in Figures 5.25 through 5.28.	200
5.25	CPU values during runtime for scheduling experiments depicted in Figure 5.24 on Lorax, in the PCL cluster.	201
5.26	CPU values during runtime for scheduling experiments depicted in Figure 5.24 on Picard, in the PCL cluster.	201
5.27	CPU values during runtime for scheduling experiments depicted in Figure 5.24 on Thing1, in the PCL cluster.	201

5.28	CPU values during runtime for scheduling experiments depicted in Figure 5.24 on Thing2, in the PCL cluster.	201
5.29	Run times for GA application with even data decomposition on Linux cluster.	202
5.30	Comparison of Mean , VTF , and 95TF policies, for the GA code on the Linux cluster with CPU loads shown in Figures 5.31 through 5.34.	204
5.31	CPU values during runtime for scheduling experiments depicted in Figure 5.30 on Mystere, in the Linux cluster.	205
5.32	CPU values during runtime for scheduling experiments depicted in Figure 5.30 on Quidam, in the Linux cluster.	205
5.33	CPU values during runtime for scheduling experiments depicted in Figure 5.30 on Saltimbanco, in the Linux cluster.	205
5.34	CPU values during runtime for scheduling experiments depicted in Figure 5.30 on Soleil, in the Linux cluster.	205
5.35	Comparison of Mean , VTF , and 95TF policies, for the GA code on the Linux cluster with CPU loads shown in Figures 5.36 through 5.39.	207
5.36	CPU values during runtime for scheduling experiments depicted in Figure 5.35 on Mystere, in the Linux cluster.	208
5.37	CPU values during runtime for scheduling experiments depicted in Figure 5.35 on Quidam, in the Linux cluster.	208
5.38	CPU values during runtime for scheduling experiments depicted in Figure 5.35 on Saltimbanco, in the Linux cluster.	208
5.39	CPU values during runtime for scheduling experiments depicted in Figure 5.35 on Soleil, in the Linux cluster.	208
5.40	Comparison of Mean , VTF , and 95TF policies, for the GA code on the PCL cluster with CPU loads shown in Figures 5.41 through 5.44.	210
5.41	CPU values during runtime for scheduling experiments depicted in Figure 5.40 on Lorax, in the PCL cluster.	211
5.42	CPU values during runtime for scheduling experiments depicted in Figure 5.40 on Picard, in the PCL cluster.	211
5.43	CPU values during runtime for scheduling experiments depicted in Figure 5.40 on Thing1, in the PCL cluster.	211
5.44	CPU values during runtime for scheduling experiments depicted in Figure 5.40 on Thing2, in the PCL cluster.	211
5.45	Comparison of Mean , VTF , and 95TF policies, for the GA code on the PCL cluster with CPU loads shown in Figures 5.46 through 5.49.	213
5.46	CPU values during runtime for scheduling experiments depicted in Figure 5.45 on Lorax, in the PCL cluster.	214
5.47	CPU values during runtime for scheduling experiments depicted in Figure 5.45 on Picard, in the PCL cluster.	214
5.48	CPU values during runtime for scheduling experiments depicted in Figure 5.45 on Thing1, in the PCL cluster.	214

5.49	CPU values during runtime for scheduling experiments depicted in Figure 5.45 on Thing2, in the PCL cluster.	214
5.50	Comparison of Mean , VTF , and 95TF policies, for the GA code on the PCL cluster with CPU loads shown in Figures 5.51 through 5.54.	216
5.51	CPU values during runtime for scheduling experiments depicted in Figure 5.50 on Lorax, in the PCL cluster.	217
5.52	CPU values during runtime for scheduling experiments depicted in Figure 5.50 on Picard, in the PCL cluster.	217
5.53	CPU values during runtime for scheduling experiments depicted in Figure 5.50 on Thing1, in the PCL cluster.	217
5.54	CPU values during runtime for scheduling experiments depicted in Figure 5.50 on Thing2, in the PCL cluster.	217
5.55	Comparison of Mean , VTF , and 95TF policies, for the NBody benchmark on the Linux cluster with CPU loads shown in Figures 5.56 through 5.59.	219
5.56	CPU values during runtime for scheduling experiments depicted in Figure 5.55 on Mystere, in the Linux cluster.	220
5.57	CPU values during runtime for scheduling experiments depicted in Figure 5.55 on Quidam, in the Linux cluster.	220
5.58	CPU values during runtime for scheduling experiments depicted in Figure 5.55 on Saltimbanco, in the Linux cluster.	220
5.59	CPU values during runtime for scheduling experiments depicted in Figure 5.55 on Soleil, in the Linux cluster.	220
5.60	Comparison of Mean , VTF , and 95TF policies, for the NBody benchmark on the Linux cluster with CPU loads shown in Figures 5.61 through 5.64.	222
5.61	CPU values during runtime for scheduling experiments depicted in Figure 5.60 on Mystere, in the Linux cluster.	223
5.62	CPU values during runtime for scheduling experiments depicted in Figure 5.60 on Quidam, in the Linux cluster.	223
5.63	CPU values during runtime for scheduling experiments depicted in Figure 5.60 on Saltimbanco, in the Linux cluster.	223
5.64	CPU values during runtime for scheduling experiments depicted in Figure 5.60 on Soleil, in the Linux cluster.	223
5.65	Comparison of Mean , VTF , and 95TF policies, for the NBody benchmark on the Linux cluster with CPU loads shown in Figures 5.66 through 5.69.	225
5.66	CPU values during runtime for scheduling experiments depicted in Figure 5.65 on Mystere, in the Linux cluster.	226
5.67	CPU values during runtime for scheduling experiments depicted in Figure 5.65 on Quidam, in the Linux cluster.	226
5.68	CPU values during runtime for scheduling experiments depicted in Figure 5.65 on Saltimbanco, in the Linux cluster.	226

5.69 CPU values during runtime for scheduling experiments depicted in Figure 5.65 on Soleil, in the Linux cluster.	226
---	-----

LIST OF TABLES

1.1	Predicted and actual execution times for a unit of work in dedicated mode on two machines.	5
1.2	Execution times for a unit of work in single-user and multi-user modes on two machines. Multi-user stochastic reports a mean and standard deviation, abbreviated sd.	8
1.3	Conservative and optimistic scheduling assignments.	11
3.1	Profile of GA Code on PCL cluster.	52
3.2	Times in seconds for arithmetic operations on PCL Cluster Machines, means over 25 runs of 1,000,000 operations.	53
3.3	GA Benchmarks on PCL cluster for problem size of 400, mean of 25 values.	53
3.4	Bandwidth between processors of the PCL Cluster in MBytes/Sec. These values were generated using a benchmark that sends 100 messages of a specified message size from processor x to processor y , and a single message consisting of a one byte acknowledgment from processor y to processor x after 100 messages.	56
3.5	Startup costs for messages on PCL Cluster in μ Sec, equal to Latency plus Packing time for message of n bytes.	56
3.6	Profile of GA Code on Linux cluster.	57
3.7	Profile of GA Code on Alpha Farm cluster.	60
3.8	Summary of the computation component models for SlaveBM used for the GA application on three dedicated platforms.	62
3.9	Summary of the computation component model for SlaveOpCt used for the GA application.	62
3.10	Summary of the Scatter component model used for the GA application on two dedicated machines.	62
3.11	Summary of the Gather component model used for the GA application on two dedicated machines.	62
3.12	Application Profile for SOR benchmark on PCL cluster.	67
3.13	Benchmark for RedComp and BlackComp based on SOR benchmark using no communication version of SOR for problem size of 2000 over 4 processors, with a mean of 25 runs.	69
3.14	Benchmark for RedComp based on SOR benchmark for one processor at problem size of 500.	69
3.15	Application Profile for SOR benchmark on Linux cluster and Alpha Farm.	71
3.16	Summary of the computation component models for benchmark models for RedComp used in defining the SOR structural model on two dedicated platforms, exact component model definitions are given in Equations 3.41 and 3.42.	71

3.17	Summary of the computation component models for benchmark models for RedComp used in defining the SOR structural model on two dedicated platforms, exact component model definitions are given in Equations 3.39 and 3.40.	72
3.18	Summary of the computation component models for benchmark models for RedComm used in defining the SOR structural model on the dedicated Linux cluster, exact component model definitions are given in Equations 3.34 through 3.38.	72
3.19	Profile for N-Body code on the PCL cluster and the Linux cluster. .	76
3.20	Summary of the computation component model for operation count used for the N-body application.	76
3.21	Summary of the computation component models for benchmark models defining the N-body structural model on two dedicated platforms.	76
3.22	Summary of the Scatter component model used for the N-body application on two dedicated machines.	77
3.23	Summary of the Gather component model used for the N-body application on two dedicated machines.	77
3.24	Benchmarks for one processor N-body code on the PCL cluster at 750 problem size, mean over 20 runs.	77
3.25	Profile for Hi-Temp code on the Linux and PCL Clusters.	82
3.26	Component model for CompJob (S_i) for Linux and PCL clusters. .	82
3.27	Benchmarks for Hi-Temp code on PCL cluster, mean of 20 runs for problem size 75.	83
3.28	Chart showing first seconds of execution of HiTemp application on PCL cluster, using benchmarks given in Table 3.27. Numbers represent job number started on processor i at time t	84
3.29	Chart showing first seconds of execution of HiTemp application on PCL cluster, using benchmarks given in Table 3.27. Numbers represent start time for job on processor i at time t , and “done” indicates when a processor is finished computing.	84
3.30	Application Profile for LU benchmark on Linux cluster	87
3.31	Communication component models for LU benchmark on Linux cluster	88
3.32	Computation component models for LU benchmark on Linux cluster and PCL cluster	88
3.33	Application Profile for SP on the Alpha Farm	92
3.34	Model for the SP benchmark on the Alpha farm.	92
3.35	Summary of the computation component models for benchmark models used in defining the EP structural model.	94
3.36	Summary of the computation component models for operation count models used in defining the EP structural model.	94

4.1	Arithmetic Combinations of a Stochastic Value with a Point Value and with other Stochastic Values [Bar78].	114
4.2	Addition and Subtraction over interval values, $x = [\underline{x}, \bar{x}]$, $y = [\underline{y}, \bar{y}]$	124
4.3	Multiplication of interval values $x = [\underline{x}, \bar{x}]$ and $y = [\underline{y}, \bar{y}]$	125
4.4	Division of interval values $x = [\underline{x}, \bar{x}]$ and $y = [\underline{y}, \bar{y}]$. Note that interval division of $\frac{x}{y}$ is only defined if 0 is not in y	125
5.1	Execution times for a unit of work in single-user and multi-user modes on two machines.	175
5.2	Conservative and optimistic scheduling assignments.	176
5.3	First 10 execution times for experiments pictured in Figure 5.4.	186
5.4	Summary statistics using Compare evaluation for experiment pictured in Figure 5.4.	186
5.5	Summary statistics using Compare evaluation for experiment pictured in Figure 5.9.	190
5.6	Summary statistics using Compare evaluation for experiment pictured in Figure 5.14.	193
5.7	Summary statistics using Compare evaluation for experiment pictured in Figure 5.19.	196
5.8	Summary statistics using Compare evaluation for experiment pictured in Figure 5.24.	199
5.9	Summary statistics using Compare evaluation for experiment pictured in Figure 5.30.	203
5.10	Summary statistics using Compare evaluation for experiment pictured in Figure 5.35.	206
5.11	Summary statistics using Compare evaluation for experiment pictured in Figure 5.40.	209
5.12	Summary statistics using Compare evaluation for experiment pictured in Figure 5.45.	212
5.13	Summary statistics using Compare evaluation for experiment pictured in Figure 5.50.	215
5.14	Summary statistics using Compare evaluation for experiment pictured in Figure 5.55.	218
5.15	Summary statistics using Compare evaluation for experiment pictured in Figure 5.60.	221
5.16	Summary statistics using Compare evaluation for experiment pictured in Figure 5.65.	224

ACKNOWLEDGMENTS

I would like to thank my advisor, Fran Berman, for all her support, encouragement and guidance over the last 5 years. Fran's advice and insight have made an enormous contribution to this thesis, and life in general. Without her help, I'd be working for Rolling Stone right now.

Second only to Fran has been Rich Wolski's influence on this work. Rich has commented, complained, suggested, and clarified many many things over the last several years, and he always had a time to chat (or drink coffee) when it was most needed.

Additionally, I thank the members of my committee: Larry Carter, Bill Griswold, Mark Ellisman, Peter Taylor, Reagan Moore and Charlie Brooks for their interest in this work and much needed feedback. And additional thanks to Professors Allen Downey and Keith Maruzllo who gave invaluable assistance along the way.

Special thanks to the other members of the lab, whose friendship and support helped make my graduate career if not fun, then at least interesting. In particular, thanks for all the late nights with Steve Fink, Neil Spring, Silvia Figueira, Graziano Obertelli, and Dmitrii Zagorodnov.

A big thanks to everyone who helped me with access to the various platforms used in this thesis: Phil Andrews and Frank Dwyer, both at SDSC, for use of the Alpha Farm; Neil Spring, SDSC and UCSD, Jim Hayes, UCSD and Graziano Obertelli, UCSD, for the Linux Cluster (aka the Circus); and the entire PCL for giving me dedicated time on the machines there.

Also, thanks to the people who donated or developed codes that were part of the test suite I used: Silvia Figueira, UCSD and SCU, for the original implementation of the SOR code; Karan Bhatia, UCSD, for the original implementation of the GA code; Kelsey Anderson, Harvey Mudd College, for her analysis of the NAS Parallel Benchmark during her CRA Distributed Mentoring summer; Alex Cereghini and Wes Ingalls for the N-body implementation, part of their CS 160

class project; James Kohl, ORNL, for the HiTemp code.

Thanks for the use of the Network Weather Service, originally developed by Rich Wolski, implemented by Neil Spring, and maintained by Jim Hayes and Graziano Obertelli. And to Neil and Dmitrii for help with data analysis.

This research was supported in part by NASA GSRP Grant #NGT-1-52133, under the guidance of David Rudy, and DARPA Contract N66001-97-C-8531, many thanks to Fredricka Darema.

To my family, for all your support: Mom, Ken, Tracy, Carl and Erin. And thanks to my non-computer science friends who kept me sane and whole: Dan, Mike, Heather, Aaron, Kacy, Coryne, and everyone else who made sure I ate and slept at semi-regular intervals. And of course, to Tiger, for being Tiger.

VITA

August 3, 1970	Born, Woods Hole, MA
1988	B.A. Mathematics/Computer Science Vassar College, Poughkeepsie, NY
1994	M.S. Computer Science University of California, San Diego
1998	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

Schopf, J.M., and Berman, F., Performance Prediction in Production Environments, *Proceedings of IPPS/SPDP '98*, April 1998. Also available as UCSD Technical Report #CS97-558, September 1997.

Schopf, J.M., Structural Prediction Models for High-Performance Distributed Applications, *Proceedings of the Cluster Computing Conference (CCC '97)*, March 1997. Also available as UCSD Technical Report #CS97-528.

Berman F., Wolski, R.M., Figueira, S., Schopf, J., and Shao, G., Application-Level Scheduling on Distributed Heterogeneous Networks, *Proceedings of Supercomputing '96*, November 1996.

Wolski, R.M., Anglano, C., Schopf, J., and Berman, F., Developing Heterogeneous Applications Using Zoom and HeNCE, *Proceedings of the Heterogeneous Processing Workshop, International Parallel Processing Symposium (IPPS)*, April 1995.

Schopf, J.M., and Berman, F., Performance Prediction Using Intervals, UCSD Technical Report #CS97-541, May 1997.

Anglano, C., Schopf, J., Wolski, R., and Berman, F., Zoom: A Hierarchical Representation for Heterogeneous Applications, UCSD Technical Report #CS95-451, October, 1995.

FIELDS OF STUDY

Major Field: Computer Science
Studies in Distributed Parallel Computing.
Professor Francine Berman

ABSTRACT OF THE DISSERTATION

Performance Prediction and Scheduling for
Parallel Applications on Multi-User Clusters

by

Jennifer Melinda Schopf

Doctor of Philosophy in Computer Science

University of California, San Diego, 1998

Professor Francine Berman, Chair

Current distributed parallel platforms can provide an efficient set of resources on which to execute many scientific applications. However, when these platforms are shared by multiple users, the performance of an application may be impacted in dynamic and unpredictable ways. In particular, it becomes challenging to achieve good performance for any single application in the system.

To efficiently use distributed parallel platforms, adaptive scheduling techniques are needed to leverage the dynamic performance characteristics of shared resources. A fundamental problem in developing adaptive schedulers is the inability to accurately predict application execution performance.

One promising approach to modeling distributed parallel applications is **structural modeling**. This approach decomposes application performance with respect to its functional structure into a top-level model and interacting component models, representing application tasks. Component models reflect the dynamic, time-dependent changes in effective capacities by allowing application or model developers to choose their parameterizations.

Many conventional models are parameterized by single (point) values. However, in shared environments, point values may provide an inaccurate representation of application behavior. An alternative to using point values is the use

of **stochastic values**, or distributions. Whereas a point value provides a single value representation of a quantity, a stochastic value provides a set of possible values weighted by probabilities to represent a range of likely behavior.

Parameters for structural models can be stochastic values in order to accurately reflect a range of observed behavior, and lend themselves to advanced scheduling techniques. Performance models that cannot adapt to changing resource conditions are often insufficient for the resource selection and scheduling needs of production distributed parallel applications. We take advantage of stochastic information and develop a **stochastic scheduler** to maximize application performance on shared clusters of workstations.

This thesis presents three main contributions: an approach to define distributed parallel application performance models called **structural modeling**, the ability to parameterize these models with **stochastic values** in order to meet the prediction needs of multiple user clusters of workstations, and a **stochastic scheduling policy** that can make use of the resulting stochastic prediction to achieve more performance efficient application execution times and more predictable application behavior.

Chapter 1

Introduction

Current distributed parallel platforms can provide the resources required to execute a scientific application efficiently. However, when these platforms are shared by multiple users, the performance of the applications using the system may be impacted in dynamic and often unpredictable ways. In particular, it becomes challenging to achieve good performance for any single application using the shared resources [SW98b, BCF⁺98, GLFK98, Lue98].

To use distributed parallel platforms efficiently, adaptive scheduling techniques can be developed to make use of the dynamic performance characteristics of shared resources. A fundamental problem in developing adaptive schedulers is the inability to accurately predict application execution performance.

Accurate performance predictions are necessary both to select a set of target resources and to schedule the application and its data upon them. When selecting resources, a prediction of application behavior may be used to evaluate, and eventually rank, different resource sets according to their likely performance. Application scheduling uses predictions of application behavior to make decisions regarding task and data mapping, as well as to estimate a program or task completion time. If any of these predictions are inaccurate, the resulting schedule may not efficiently use the resources, and application execution times may be unnecessarily increased.

One promising approach to modeling distributed parallel applications is **structural modeling**, which decomposes performance with respect to the functional structure of the application into a top-level model and interacting component models. Component models represent application tasks and application or model developers can select the parameters of a component model (representing both application and system characteristics) to reflect the dynamic time-dependent changes in effective capacities of the resources.

Performance models that do not reflect changing resource conditions are often insufficient for the resource selection and scheduling needs of shared distributed parallel applications. Many conventional models are parameterized by single (point) values. However, in shared environments, point values may provide an inaccurate representation of application behavior due to variations in resource performance. An alternative to using point values is the use of **stochastic values**, or distributions. Whereas a point value provides a single value representation of a quantity, a stochastic value provides a set of possible values weighted by probabilities to represent a range of likely behavior.

Parameters for structural models can be stochastic values (or include other additional information) in order to accurately reflect a range of observed behavior. These parameters, as well as stochastic performance predictions, can be used by schedulers to adapt to the dynamic behavior of shared distributed parallel environments. We take advantage of stochastic information and develop a **stochastic scheduler** to maximize the performance of shared clusters of workstations for parallel applications

*This thesis presents three main contributions: an approach to define distributed parallel application performance models called **structural modeling**, the ability to parameterize these models with **stochastic values** in order to meet the prediction needs of shared clusters of workstations, and a **stochastic scheduling policy** that can make use of stochastic predictions to achieve better application execution times and more predictable application behavior.*

This chapter briefly defines key terms, motivates the need for this research, and outlines the approaches for the following chapters.

1.A High Performance Distributed Parallel Computing

As high performance applications have grown in size and complexity, computational resources have adapted to execute them efficiently as well. One way this has been achieved is through the parallelization of applications and the use of distributed resources. The area that we call **distributed parallel computing** is also known as heterogeneous computing, metacomputing, or computing on a computational grid. Basically, it involves the cooperative use of multiple distributed resources connected by networks to solve a single, resource-intensive application.

Distributed parallel systems exist on a number of scales, from a group of workstations in a lab to collections of high-performance computers and other resources linked across the nation or the world. We include in this category clusters of uniform workstations that may exhibit heterogeneous performance characteristics when shared by multiple users. Distributed parallel systems are often characterized by resources with variable performance and slow or undependable networks. Applications that execute on these networks may be resource-intensive, and have often been adapted from parallel applications previously run on a single MPP.

In order to successfully use distributed platforms for parallel applications, several basic issues must be addressed. First, hardware and software infrastructure is required to provide dependable, consistent, and inexpensive access to computation sites, remote resources, and data archives. For the purposes of this thesis, we assume that the underlying hardware is dependable, that is, we do not explicitly address the possibility of failure except as the perception of inadequate performance of a resource by the application. We also assume that an underlying communication substrate is available. For example, our experiments use either

PVM or MPI as a communication substrate, and target applications running over clusters of shared workstations. Secondly, monitoring tools are needed to observe the use of the system. For example, we make extensive use of the Network Weather Service [Wol96, Wol97, WSP97], a tool that can supply dynamic values for bandwidth, available CPU, and memory on a given system. Thirdly, schedulers are required to best use the available shared resources. That is the issue this thesis addresses.

1.B Performance Prediction Models

In order to build efficient schedules, we must have accurate **performance prediction models** for distributed parallel systems, that is, models that represent the wall-clock execution time of an application or part of an application. A **model** is a mathematical representation of a process, device or concept. Models are used to represent a variety of entities throughout computer science—from architectures to forecasts of resource usage to algorithm costs.

One way predictions and prediction models are used is as part of scheduling to assign work to processors. To illustrate the need for accurate prediction models, we present the following example. Consider a simple two-machine system, consisting of machines A and B, executing an embarrassingly parallel application with 30 units of identical work to be completed. In order to allocate work efficiently, we need an accurate estimate of the execution time per unit of work for each machine. This value is often the result of a prediction model.

If the estimate of time per unit of work for a machine is inaccurate, the execution time of the application may be unnecessarily extended. If we assume that the fastest execution time is achieved by having all the processors begin at the same time and finish at approximately the same time, we can achieve an efficient schedule by balancing the number of units of work each processor is assigned accordingly. A poor prediction model might report that Machine A will take 6

seconds and Machine B might take 9 seconds per unit of work. We would then assign 18 units to A and 12 units to B, for a predicted execution time of 108 seconds, as shown in Table 1.1. If Machine A had an actual execution time of 10 seconds per unit and Machine B took 5 seconds per unit, the execution time would be 180 seconds, since the load during execution would be unbalanced. A better data allocation, based on an accurate prediction of work, would be 10 units for Machine A and 20 units for Machine B.

	Machine A	Machine B	Assign A	Assign B	Total Time (Predicted)	Total Time (Actual)
Inaccurate Prediction	6 sec	9 sec	18 units	12 units	108 sec	180 sec
Actual Time	10 sec	5 sec	10 units	20 units	100 sec	100 sec

Table 1.1: Predicted and actual execution times for a unit of work in dedicated mode on two machines.

In Chapter 3, we explore **structural modeling** as an approach to accurate performance modeling. Structural modeling uses the functional structure of the application to define a set of equations that reflect the time-dependent, dynamic mix of application tasks occurring during execution in a distributed parallel environment. We define **tasks** to be atomic units of either communication or computation¹, since computation and communication comprise the major performance activities associated with distributed parallel applications. In the case of computation, we consider a task to be an atomic unit of computation to be executed on a single machine. For communication, we consider a task to be an atomic grouping of communication calls, typically over short time period. The **functional structure** of an application can be thought of as a program dependence graph consisting of nodes that are communication and computation tasks, and edges that describe their relationship to one another. Also in Chapter 3, we present the steps

¹Note that we generalize the conventional notion of “task” to include communication.

involved in building a structural model for a given application, and discuss the verification and usage of structural models for generating performance predictions for a variety of applications on three cluster platforms.

We do not model an application and a system as separate entities. Rather, we examine **implementations** of an application for a particular resource environment. Parallel distributed applications are typically not developed for a generic platform; they are tuned to the particular performance characteristics and architecture model of the resources available. For example, if an application includes a rendering routine, there may be several choices of rendering implementations available, each tuned in terms of the cache size, available math libraries, the precision needed for a given set of application data and/or the programming model supported by the resource. The development of a specific implementation will depend on characteristics of the application, its data, and the underlying platform. Therefore a structural model should represent implementations of an application with respect to particular platforms.

A good performance model, like a good scientific theory, is able to explain available observations and predict future events, while abstracting away unimportant details [Fos95]. Our models are represented by equations that are parameterized by both application and system characteristics. These equations allow us to abstract away details into the top level models and the parameters, but still accurately model the performance by showing the inter-relation of application and system factors through the component models and their parameterizations. If the predictions are for use in a timely manner – for example, as part of a runtime scheduling approach – they must be able to be evaluated quickly and with minimal computational overhead as well, a factor that we address throughout.

1.C Point Values and Stochastic Values

Most performance prediction models use parameters to describe system and application characteristics such as bandwidth, available CPU, message size, operation counts, etc. Model parameters are generally represented as a single likely value, which we refer to as a **point value**. For example, a point value for bandwidth might be 7 Mbits/second.

In practice, point values are often a best guess, an estimate under ideal circumstances, or a value that is accurate only for a given time-frame. In some situations it may be more accurate to represent system and application characteristics as a *distribution* of possible values over a range; for example, bandwidth might be reported as a normal distribution having a mean of 7 Mbits/second with a standard deviation of 1 Mbit/second. We refer to values represented as distributions as **stochastic values**. Whereas a point value gives a single value for a quantity, a stochastic value gives a set of possible values weighted by probabilities to represent a range of likely behavior [TK93].

The need for stochastic values in our example system can be seen when comparing predictions in a dedicated (single-user) environment with predictions in a multi-user (shared) environment. In a single-user environment, the time for a machine to perform one unit of work can often be represented as a point value since there is usually a negligible variance in resource availability and runtimes on dedicated machines.² For example, to balance the execution times of the example machines in a dedicated setting, where Machine A takes 10 seconds to complete a unit of work and Machine B takes 5 seconds (as shown in Table 1.2), Machine B should receive twice as much work as Machine A.

In a multi-user distributed environment, contention for processors and memories will cause the unit execution times to vary over time in a machine-dependent fashion; hence, simply having a good point value prediction of execution time may not be sufficient. If we determine unit execution time using a mean

²This may not be the case if an application is data dependent or non-deterministic.

	Machine A	Machine B	Assign A	Assign B
Single-user	10 sec	5 sec	10 units	20 units
Multi-user (point)	12 sec	12 sec	15 units	15 units
Multi-user (stochastic)	12 sec mean 0.3 sec sd	12 sec mean 1.8 sec sd	?	?

Table 1.2: Execution times for a unit of work in single-user and multi-user modes on two machines. Multi-user stochastic reports a mean and standard deviation, abbreviated sd.

capacity measure over a 24-hour period for a fixed amount of work, it is possible that A and B will have the same unit execution time on average, say 12 seconds per unit of work. In this case, it would make sense to equally balance the work between the machines.

However, a mean value is only summary statistic for a range of actual values, and may neglect critical information about the distribution of work over time. A stochastic value includes information about the distribution of values over the performance range. For example, it may be that because Machine B is much faster than Machine A, it has more users and therefore a more dynamic load. Because of this, at any given time the unit execution time for Machine B might be a mean of 12 seconds per unit of work, with a standard deviation of 1.8 seconds. If we assume the distributions are normal, then two standard deviations will cover approximately 95% of the values, so the time to complete a unit of work will usually vary over an interval from 8.4 seconds to 15.6 seconds. On the other hand, since Machine A is a slower machine without as many users contending for its resources, the actual unit execution time for this machine might be a mean of 12 seconds per unit of work and a standard deviation of 0.3 seconds (or 11.4 to 12.6 seconds per unit of work in most cases). On average, both machines perform the same, however at any given time their performance may differ radically. The amount of data to be assigned to each machine in this case is no longer a simple decision, but will depend on the goals of the user or application developer and the

optimization function of the scheduling policy.

Conventional performance prediction models typically cannot take advantage of knowledge about the distribution of a model parameter value. *In Chapter 4, we define an extension to structural modeling to allow for model parameters that are **stochastic values**, resulting in application performance predictions that are **stochastic predictions**.* We demonstrate that stochastic values can enhance the information conveyed by a performance prediction. In order to make use of stochastic values in prediction models, we need to know how the value can be represented and how to combine the values arithmetically. We examine three representations of distributional data: summarizing them as normal distributions; representing them as a simple interval of values; and using histograms. Arithmetic formulas for each representation are discussed, as well as the advantages and disadvantages of each approach. Chapter 4 also gives an overview of related approaches and additional research on the use of statistical approaches for pragmatic performance predictions.

1.D Stochastic Scheduling

With additional information provided by a stochastic prediction it is possible to develop more sophisticated application schedulers. As stated in [Ber98], an **application scheduler** may:

1. Select a set of resources on which to run the tasks of the application.
2. Assign application task(s) to compute resources.
3. Distribute data or co-locate data and computation.
4. Order tasks on a compute resource.
5. Order communication between tasks.

In the literature, the first two items are often referred to as **mapping**, the third item is referred to as **allocation**, and items 4, 5, or 1-5 are referred to as **scheduling**. Each of these actions uses some form of prediction.

When a scheduler makes use of stochastic information to determine a schedule for the application, we refer to it as a **stochastic scheduler**. Just as stochastic predictions predict a range of possible execution times, stochastic schedulers must choose from a set of possible schedules. Therefore schedulers of this nature will have the added actions of both defining the set of possible schedules and deciding which schedule in that set is appropriate for a given application, environment and user.

To illustrate using our example, if the application developer required the application to finish by a given time, a scheduler may choose to conservatively assign the data and assign more work to the smaller variance machine (Machine A). If the prediction supplied to the scheduler was a stochastic value represented using a normal distribution, one way to do this might be to use a 95% confidence interval³ and assign the data using the mean plus two standard deviations as the time per unit of work. This value is chosen for normal distributions since it equates to approximately 95% of the values in the predicted range. In the example setting, we would assign 17 units of work (rounded up from 16.59) for Machine A and 13 units for Machine B, as shown in Table 1.3. If the execution times do indeed fall within the predicted interval (that is, machine A actually varied in performance from 11.4 to 12.6 seconds per unit of work and machine B actually varied in performance from from 8.4 seconds to 15.6 seconds). this data assignment would have a best case completion time of 148.2, but a worse case completion time of only 214.2, based on the given range of possible performance. We call this a **95% conservative schedule** since it corresponds to a 95% probability that each of the tasks will finish in the predicted range of time on their respective processors.

³A confidence interval is an interval of plausible values for a characteristic constructed so that the value of the characteristic will be captured inside the interval [DP96a].

	Assign A	Assign B	Best time	Worst time
95% Conservative Prediction	17 units	13 units	148.2 sec	214.2 sec
95% Optimistic Prediction	13 units	17 units	152.1 sec	265.2 sec

Table 1.3: Conservative and optimistic scheduling assignments.

If there was a small penalty for poor predictions, a scheduler might optimistically assign a greater portion of the work to the machine with the maximum potential performance, Machine B. Likewise, if the application developer had reason to believe that the previous distribution of data was too conservative, he/she might want to assign more units of work to Machine B. For example, the opposite of a 95% conservative schedule would be a **95% optimistic schedule**, as shown in line 2 of Table 1.3, corresponding to a 5% conservative schedule, or a 5% confidence interval that the task on a processor will complete in a given time. The 95% optimistic schedule would have a best case completion time of 152.1, but a worse case completion time of 265.2 in this setting.

As third option, we could consider allowing the percentage of conservatism to vary for the two platforms, that is, instead of a 95% optimistic or 95% conservative schedule, we could vary these percentages along the entire spectrum. Knowing the environment in order to determine the percentage of conservatism needed is as important as having valid information when scheduling with stochastic data, as the possible variations are extensive. A scheduling approach that can make these decisions and provide good execution performance is the ultimate goal of the work described in this thesis.

*In Chapter 5, we present an approach to developing a **stochastic scheduler**.* The stochastic scheduling policy we present is based on time balancing and uses a system of benefits and penalties to define the values to use in the set of possible schedules. This approach is similar to the scheduling approach of Sih and Lee [SL90b], in which the amount of work is increased for a processor with desirable characteristics (such as a light load, fast processor speed, etc.), called

benefits, or decreased when a machine has undesirable properties (such as poor network connectivity, small memory capacity, etc.), called **penalties**. The resulting schedules often achieve a faster application execution time than comparative schedules built without the additional stochastic data, and frequently result in more predictable application execution behavior for the experimental applications.

1.E Summary

The three main research contributions presented in this thesis are: an approach to performance modeling called structural modeling, the ability to parameterize these models using stochastic information, and a stochastic scheduling policy that adapts to the performance characteristics available resources of shared clusters of workstations. The goal of this dissertation is to present a technique for modeling distributed parallel applications and a scheduling approach that can utilize enhanced modeling information to determine performance-efficient application schedules.

1.F Dissertation Outline

The thesis is organized as follows: Chapter 2 presents the experimental environment used for the thesis. It details the platforms used, the workload characterization, and the test suite of applications. Chapter 3 defines structural modeling. Chapter 4 presents stochastic parameters and predictions. Chapter 5 describes the use of the stochastic predictions in a stochastic scheduling policy based on benefits and penalties. Chapter 6 presents conclusions and contributions, and discusses the extensive possible future work.

Chapter 2

Experimental Environment

In order to experimentally validate the claims in this thesis, we define an experimental space with a three axes: platform, workload and application. This chapter defines these axes, and details the domain for each of them.

2.A Experimental Platforms

We focus on environments where resources available to a user will have variable, possibly unpredictable, performance due to the presence of other users on the system. Although performance variation can occur on a single machine or on a space-shared parallel platform (where sets of nodes are assigned to a single user at a time), we focus specifically on shared clusters of workstations. In particular, our experiments were run on the following platforms:

UCSD Parallel Computation Lab (PCL) Cluster. This cluster consists of four Sparc workstations located in the UCSD Parallel Computation Lab. The machines are connected with either 10 Mbit (slow) or 100 Mbit (fast) ethernet over an Intel switch, and all run Solaris. The machines in the PCL cluster are:

Thing1: 200MHz Sun Ultra 1, 128MBytes RAM, fast ethernet connection

Thing2: 200MHz Sun Ultra 1, 128MBytes RAM, fast ethernet connection

Picard: 40MHz Sparc 10, 32MBytes RAM, slow ethernet connection

Lorax: 85MHz Sparc 5, 32MBytes RAM, slow ethernet connection

These machines are workstations on student's desks, and are used for both day-to-day activities like reading email, using Netscape, and running compilers, as well as for more compute-intensive experimental work.

Application-Level Scheduling (AppLeS) Research Group Linux Cluster.

This cluster consists of four PC's located in the Parallel Computation Lab. They are connected using full-duplex, 100 megabit (fast) switched ethernet.

This cluster is made up of:

Soleil: 400 MHz Pentium II, Debian GNU/Linux, 384 MBytes RAM

Saltimbanco: 400 MHz Pentium II, Debian GNU/Linux, 256 MBytes RAM

Mystere: 400 MHz Pentium II, Debian GNU/Linux, 384 MBytes RAM

Quidam: 400 MHz Pentium II, Debian GNU/Linux, 256 MBytes RAM

The Linux Cluster is primarily used as a computation resource, and not used on an individual basis. In addition, this is a lab-administered machine, so extensive dedicated experiments were possible on this platform.

San Diego Supercomputer Center (SDSC) Alpha Farm. This platform consists of eight DEC Alpha 3000/400 workstations running OSF Unix located at the San Diego Supercomputer Center. They each run at 133 MHz, and have an 8K primary cache and a 512K secondary cache, along with 128MBytes of RAM. They are connected via high-speed ethernet.

This shared platform has been used by various researchers in a compute-intensive manner, usually as a system, but at times individually by machine.

These three platforms span a wide range of cluster system characteristics. In terms of performance, the Linux cluster machines were significantly faster than the PCL cluster machines. The PCL cluster machines were faster than the Alphas, although there were twice as many machines in the Alpha Farm as the PCL cluster. In terms of networking, both the Alpha Farm and the Linux Cluster used high speed ethernet connections, but communication costs were typically much more a factor in the performance on the Linux cluster due to its faster workstations. In addition, while the Alpha Farm and Linux Cluster both have homogeneous processors, allowing us to make some simplifying modeling assumptions, the PCL platform had three types of machines and different network connections as well. Also, although the machines in the Alpha Farm and the Linux cluster were homogeneous, they were not used uniformly, as each had a single workstation used more often than the others to connect to the system. Finally, these systems all had different operating systems, different C and Fortran compilers, etc.

2.B Workload Characterization

The second axis of our experimental space is workload. By **workload** we mean the set of applications running on a given platform within a specific period. Workload is typically measured in a performance model by the effect it has on the platform resource capacities by such measures as *bandwidth*, *available CPU*, *latency*, etc. One of the primary sources of performance variation on a production platform is the fluctuation of available CPU and bandwidth at a given time. By *available CPU* we mean the percentage of a machine available for use by a given application, as used by Wolski [Wol96]. These are also the parameters that are the most difficult to control in an environment where a user cannot limit the access of other users to the machines or networks of interest. Note that a “typical” workload can vary radically from one site to another, from one time interval to another, and from one resource to another, so there is no universal typical workload.

We addressed the lack of a typical workload in two ways. First, we ran experiments on platforms where we could control the outside use of the resources – unlike the average user – and could simulate additional workloads on them. This helped in validating and verifying the performance prediction strategies and models by providing repeatable results. Second, we ran experiments on production platforms in competition with other users to validate and verify our predictive and scheduling techniques in a realistic setting. Details on the workload simulations and production workloads are discussed as part of the experimental results in Chapters 4 and 5.

2.C Applications

The third axis of our experimental space is the application. Distributed parallel applications have some similarity with both parallel and distributed applications. Two classes of distributed parallel applications, **Master-Slave** and **Regular SPMD**, are detailed in the next two sections, along with the representative applications and implementations for each class that were used as a test suite for this thesis.

Parallel applications targeted to distributed resources differ from parallel applications targeted to MPPs in several ways. Characteristics of parallel distributed applications include:

Coarse granularity. In order to amortize the high communication costs of commercial networks and off-the-shelf operating systems, distributed parallel applications are typically coarse-grained. Many MIMD or SPMD applications that were originally developed for MPP’s port well to distributed parallel platforms, but SIMD applications require synchronization that cannot be performed efficiently in this environment. Many vector codes have been reworked to suit networked resources, but this can be a difficult task.

Small number of tasks. Distributed parallel applications generally consist of a

small number of tasks [CS95, DS95, DTMH96, PRW94, MMFM93, WK93, YB95]. The size of individual tasks vary from application to application.

Modular Implementation. Many distributed parallel applications are actually several parallel applications joined together to solve a larger problem. For example, the Global Climate Modeling group at UCLA combined an ocean physics code [Cox84], an atmospheric physics code [AL77, MMFM93] and an atmospheric chemistry code [DS95] to more accurately investigate global weather patterns. Each of these codes were developed for separate platforms and but are executed in tandem over a wide-area high-performance distributed parallel platform to solve a larger problem [MFS⁺95]. Distributed parallel applications such as this may also use a mixture of languages and computational paradigms (data parallel, sequential, vector), and may even use different underlying communication substrates (PVM, MPI, NEXUS, etc.).

Tasks may have multiple implementations. A task in a parallel distributed application may have multiple implementations, each perhaps tuned to a different platform or performance characteristic. For example, if an application uses a rendering routine, there may be several choices of rendering routines targeted to different possible platforms, each differing in performance capability and algorithmic approach.

Many parallel distributed applications fall within the classes of Master-Slave and Regular SPMD applications, described in the following sections. Within these classes, we attempted to vary other characteristics as well. The applications in our test suite are written in C, Fortran, or a mix of both. Some run over PVM, some use MPI. Some were developed by students, some are kernels of scientific codes, and others are part of scientific benchmarking suites. The communication and computation ratios vary from computation-intensive to communication-intensive, although most are coarse-grained, with a high computation to commu-

nication ratio as seen in most practical codes for distributed platforms. The codes we are using have different synchronization styles, and address data and work distribution issues differently. Future work includes extending the techniques in this thesis for other application classes, such as pipelined and multi-level codes.

2.D Master-Slave Applications

One class of application that achieves good performance on clusters of workstations is the **Master-Slave** class of applications. These applications are structured so that a Master process controls the data for multiple Slave processes, with all the Slaves running the same code. The Master-Slave paradigm is widely used in the cluster computing setting because it is easy for the Master to dynamically adjust the data decomposition across the Slave computations, and to achieve good performance on systems with changing characteristics. This is especially important for multi-user clusters of heterogeneous workstations. Programming approaches vary for Master-Slave computations. We denote the two most common Master-Slave paradigms as **Assign** and **Request**, and characterize their programming approaches below.

Assign. In an Assign-style Master-Slave application, the Master process assigns all of the data to the Slave processes before they begin to execute. The Slaves run in parallel, and send their results back to the Master process. This usually repeats for some number of iterations.

An example of an Assign Master-Slave application is shown in Figure 2.1 and the pseudocode is given in Figure 2.2. This type of application consists of a Master task that determines how much work each of P Slave tasks performs. The Master communicates to the Slaves generally using a Scatter routine, perhaps a multicast, and receives information back from the Slaves with a Gather routine, perhaps a series of receives. Synchronization occurs every

iteration at the Master task, but may also occur for each communication or computation task depending on the implementation.

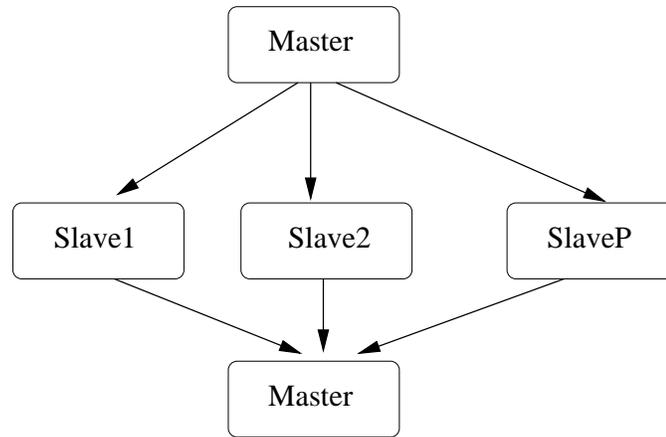


Figure 2.1: Graph of one iteration of an Assign Master-Slave application.

```

Master Computation:
  Start Slaves
  For i = 1 to MaxIterations
    Broadcast Data to Slaves
    Receive Data from all Slaves
  End of iteration computation
  Kill Slaves

Slave Computation:
  While (True)
    Receive Data
    Compute Slave Work
    Send Data to Master
  
```

Figure 2.2: Pseudocode for an Assign Master-Slave application.

Request. In a Request-style Master-Slave application, the Master task holds the data and Slave tasks request portions of it. Each piece of work can be thought of as a **job**. Output from the job can either be returned a piece at a time when the Slave requests another job, or all at once when the pool of jobs is

empty. There is no synchronization between the Slaves, but there may be a synchronization between iterations as the Master determines a new set of jobs to be handed out. This style of Master-Slave application is also called *work queue*, *task farming*, or *self-scheduling* [Wil95]. This is characterized by the graph in Figure 2.3 and the pseudocode in Figure 2.4.

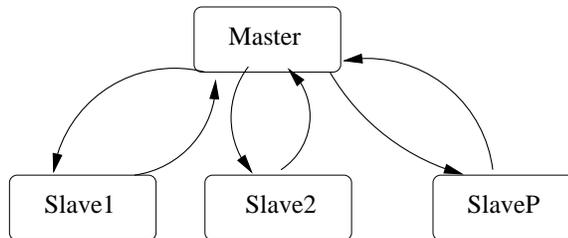


Figure 2.3: Graph of one iteration of a Request Master-Slave application.

```

Master Computation:
  Start Slaves
  For i = 1 to MaxIterations
    Send initial Data to each Slave
    Receive Data from all Slaves
    While there is Data uncomputed
      Send Data set to Slave
      Receive Data from all Slaves
    End of iteration computation
  Kill Slaves

Slave Computation:
  While (True)
    Receive Data
    Compute Slave Work
    Send Data to Master
    Ask Master for more Data
  
```

Figure 2.4: Pseudocode for a Request Master-Slave application.

In the following subsections we present the Master-Slave distributed applications that are used as the test suite codes for this thesis.

2.D.1 Genetic Algorithm Application

Genetic algorithms are search algorithms that model survival over generations of populations. In a typical genetic algorithm, there is a “population” of possible solutions that are “bred” with one another to create “child” solutions. The child solutions are evaluated, and the “fittest” (best) of the new “generation” of solutions are used as “parents” for the next generation. Genetic algorithms have been used extensively by the artificial intelligence community as an optimization technique for NP-complete and NP-hard problems. They are now being used by several groups of computational scientists [DTMH96, SK95, PM94] to address problems such as protein folding.

We implemented a genetic algorithm (GA) heuristic for the Traveling Salesman Problem (TSP) [LLKS85, WSF89]. Our distributed implementation of this genetic algorithm uses a global population and synchronizes between generations [Bha96]. It was written in C using PVM. This application has a typical Assign Master-Slave structure. All of the Slaves operate on a global population (each member of the population is a solution to the TSP for a given set of cities) that is broadcast to them by the Master using a PVM multicast routine. Each Slave works in isolation to create a specified number of children (representing new tours), and to evaluate them. This data is sent back to the Master. Once all the sets of children are received by the Master, they are sorted (by efficiency of the tour), some percentage are chosen to be the next generation, and the cycle begins again. Figure 2.5 shows the pseudocode for this application. In our implementation, a Slave process runs on the Master processor. Data assignment is done statically at runtime.

2.D.2 High Temperature Superconductor Kernel

The High Temperature Superconductor Kernel (HiTemp) is used as part of the suite of applications used to test new versions of PVM, and is a simplified

```

Master Computation:
  Start Slaves
  For i = 1 to MaxIterations
    Broadcast entire population to Slave
    Receive Data from all Slaves
    Sort children
    Select next generation
  Kill Slaves
  Return best child

Slave Computation:
  While (True)
    Receive population from Master
    For j = 1 to NumberofChildren
      Randomly pick two parents
      Cross parents to create child
      Evaluate child
    Send all children to Master

```

Figure 2.5: Pseudocode for GA Application

kernel of a larger superconductor code [Koh97]. The Master in this application spawns Slaves that do matrix inverses of various sizes using specially-tuned BLAS routines.

HiTemp uses a “pool of jobs” approach, typical of the Request-style Master-Slave applications. The Master is started, and it spawns a Slave task on each available host. Each Slave then requests individual jobs from the available work pool held by the Master, in this case the matrix information, until the pool of jobs is empty. In this application, the actual jobs vary in size nondeterministically to simulate superconductor environmental data. This code is used as a test code for PVM, and is written in C and Fortran using PVM library calls. Data decomposition is done dynamically at runtime by the Slaves, and the only synchronization is between iterations. A Slave process runs on the Master process, but this interferes at most minimally with the Slave execution. Pseudocode for this application is given in Figure 2.6.

```

Master Computation:
  Start Slaves
  For i = 1 to MaxIterations
    Send one job to each Slave
    While there are jobs in work pool
      Receive result of job from Slave
      Send job to Slave
    Receive result of last job from each Slave
  Kill Slaves

Slave Computation:
  While (True)
    Receive job
    Compute matrix inverse
    Send job result to Master

```

Figure 2.6: Pseudocode for HiTemp kernel

2.D.3 N-body Code

N-body simulations, also known as particle simulations, arise in astrophysics and celestial mechanics (gravity is the force), plasma simulation and molecular dynamics (electrostatic attraction or repulsion is the force), and computational fluid dynamics (using the vortex method). These problems were some of the first problems addressed in parallel computation and continue to be studied in distributed parallel environments. The papers collected in the SIAM minisymposium [HT97] and its predecessor [Bai95] offer ample evidence of the breadth and importance of N-body methods.

The problem our N-body implementation addresses is the tracking of the motion of planetary bodies. We scale the number of bodies involved, using a two-dimensional space with boundaries. Our implementation is an exact N^2 version that calculates the force of each planet on every other, and then updates their velocities and locations accordingly for each time step. It is based on an Assign-style Master-Slave approach [CI98] and written in C using PVM. For each iteration, the Master sends the entire set of bodies to each Slave, and also assigns a portion

of the bodies to each processor to calculate the next iteration of values. The Slaves calculate the new positions and velocities for their assigned bodies, and then send this data back to the Master process. Data decomposition is done statically at runtime by the Master, which also reads in an initial data file. Figure 2.7 shows pseudocode for this application.

```

Master Computation:
  Start Slaves
  For i = 1 to MaxIterations
    Broadcast all body data to Slaves
    Send each slave sector information
    Receive data for each sector from Slaves
    Sort bodies according to new positions
  Kill Slaves

Slave Computation:
  While (True)
    Receive velocity, force, position data
    Compute new data for bodies in sector
    Send sector data to Master

```

Figure 2.7: Pseudocode for the N^2 N-body Code.

2.E Regular SPMD Applications

In conjunction with the class of Master-Slave applications, we also focused on the class of **Regular SPMD** applications. Regular SPMD applications are comprised of multiple identical tasks (with no “lead” task) working on different data sets. We focus on applications that have regular, well-structured communication patterns. Such codes are widely used in the scientific community. A few examples include: NASA’s High Speed Civil Transport design code [GW], an application to model the San Diego Bay [HT], as well as matrix codes and other mathematical functions at the heart of many scientific codes, including successive over-relaxation codes, multigrid codes and Fourier transforms [BHS⁺95].

We implemented a test suite of Regular SPMD applications as examples of this class. In the following subsections, we address the methods of data distribution, the communication patterns, and synchronization issues for each.

2.E.1 Successive Over-Relaxation Code

Successive Over-Relaxation (SOR) is a Regular SPMD code that solves Laplace’s equation. Our implementation uses a red-black stencil approach where the calculation of each point in a grid at time t is dependent on the values in a stencil around it at time $t - 1$, as depicted in Figure 2.8. The application is divided into “red” and “black” phases, with communication and computation alternating for each. In our implementation, these two phases repeat for a predefined number of iterations. Pseudocode for the SOR is given in Figure 2.9.

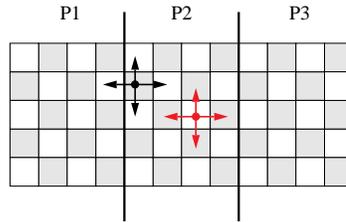


Figure 2.8: Strip decomposition for Red-Black SOR

We used a strip decomposition to partition the code onto the workstation clusters. In this implementation, all the processors are equal peers, as opposed to the Master and Slave set-up, and each is responsible for an individual data set. Communication is done twice an iteration, and there is only loose synchronization between neighboring processors.

2.E.2 NPB LU Benchmark

We implemented distributed forms of several of the NAS Parallel Benchmarks (NPB) [BHS⁺95]. These benchmarks were developed to embody key compu-

```

Setup, including decomposition
For i = 1 to MaxIterations
  RED PHASE
  Receive right and left data
  For j = FirstRow to LastRow
    For k = FirstCol to LastCol
      Compute red values

  Share left column with left neighbor
  Share right column with right neighbor
  BLACK PHASE
  Receive right and left data
  For j = FirstRow to LastRow
    For k = FirstCol to LastCol
      Compute black values

  Share left column with left neighbor
  Share right column with right neighbor

```

Figure 2.9: Pseudocode for the SOR benchmark.

tational and data-movement characteristics of typical processing in computational fluid dynamics (CFD) calculations, and have gained wide acceptance as a standard indicator of supercomputer and parallel distributed platform performance. They are carefully coded to use modern algorithms, avoid unnecessary computation, and generally represent real-world codes. They perform reasonably well across many platforms.

The **LU benchmark** is a simulated CFD application that solves a block-lower-triangular/block-upper-triangular system of equations. This system of equations is the result of an unfactored implicit finite-difference discretization of the Navier-Stokes equations in three dimensions. The LU benchmark finds lower triangular and upper triangular matrixes such that $L \cdot U = A$ for an original matrix A . The NPB version of the LU benchmark is based on the NX reference implementation from 1991 [BBB⁺91]. It consists of a startup phase, a lower/upper triangular-solving function that is iterated, and then a verification phase. The lower-upper triangular solving function is an iterative implicit method that partitions the left-hand side matrix into a lower triangular matrix and an upper triangular matrix.

A 2-D partitioning of the grid onto processors occurs by halving the grid repeatedly in the first two dimensions, alternately x and then y , until all power-of-two processors are assigned, resulting in vertical columns of data assigned to the individual processors. The computation then proceeds to compute from one corner of a given z plane, on diagonals, to the opposite corner of the same z plane, and then proceeds to the next z plane. Communication of boundary data occurs after completion of the computation on all diagonals that contact an adjacent partition. This constitutes a diagonal pipelining method and is called a “wavefront” method by its authors [BFVW93]. This is depicted for one Z plane in Figure 2.10. It results in a relatively large number of small communications of 5 words each.

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

Figure 2.10: Wavefront execution of the LU benchmark, shown for a single z plane, labeled with order of execution possible.

Although this algorithm is not an optimal solution for the problem, it is used as a NPB benchmark because it is very sensitive to the small-message communication performance of an MPI implementation. It is the only benchmark in the NPB 2.0 suite that sends large numbers of very small (40 byte) messages. For this reason we included it in our Regular SPMD test suite.

2.E.3 SP Benchmark

The SP Benchmark, part of the NPB suite [BHS⁺95], is a simulated CFD application that solves systems of equations resulting from an approximately

factored implicit finite-difference discretization of the Navier-Stokes equations. The SP code solves scalar penta-diagonal systems resulting from full diagonalization of the approximately factored scheme. It solves three sets of un-coupled systems of equations, first in the x , then in the y , then in the z direction. It uses a multi-partition algorithm [BC88] in which each processor is responsible for several disjoint sub-blocks of points (“cells”) of the grid. The cells are arranged such that for each direction of the “line solve phase”, the cells belonging to a certain processor will be evenly distributed along the direction of solution. This allows each processor to perform useful work throughout a line solve, instead of being forced to wait for the partial solution to a line from another processor before beginning work. Additionally, the information from a cell is not sent to the next processor until all sections of linear equation systems handled in this cell have been solved. Therefore the granularity of communications is kept large and fewer messages are sent.

Structurally, this code has a small initialization phase, followed by a broadcast of the data to all the processors. Then the bulk of the work is done in a phase that repeats for ten iterations and consists of copying faces (overlapping data) between each processor, solving in the x direction, the y direction, and the z direction, and an update task. There is loose synchronization between nodes for each iteration.

2.E.4 EP Kernel

We include the NPB Embarrassingly Parallel (EP) Kernel [BHS⁺95] as an example of a Regular SPMD application with no synchronization and minimal communication.

The NPB EP benchmark generates 2^n pairs of random numbers, and tests them to see whether Gaussian random deviates can be constructed from them. The pseudo-random number generator used in this is of the linear congruential recursion type. This problem is typical of many Monte Carlo simulation applications. The only requirement for communication is a small gathering of information at the end.

There is no synchronization. The data decomposition, in this case the assignment of a number of pairs to generate, is done statically at compile time.

Chapter 3

Structural Modeling

In this chapter we present a performance modeling technique called **structural modeling** that provides a prediction of an application's execution time on a set of distributed resources. A structural model abstracts application performance as a set of component models, one for each task, and an over-reaching top-level model that describes their relations. Each component model can be selected independently, providing the flexibility to substitute distinct models for each computation and communication task as required. Structural modeling was developed for distributed parallel applications, especially those targeted to production clusters of workstations, but this approach is fully applicable to a wider environment.

This chapter is organized as follows: Section 3.A defines the terms used throughout and addresses several practical issues for using structural models in current distributed parallel environments. We fully derive structural models for two representative applications, the Genetic Algorithm Master-Slave code in Section 3.B and the Regular SPMD Successive Over-Relaxation benchmark in 3.C. Modeling results for additional applications in each class are summarized in Section 3.D to demonstrate the accuracy and flexibility of this approach. Related work is presented in Section 3.E, and Section 3.F gives a summary of the approach and results presented in this chapter.

3.A Definitions

One promising approach to modeling distributed parallel applications is **structural modeling**. This approach decomposes application performance with respect to its functional structure into a top-level model and interacting component models representing application tasks. Unlike many other performance modeling approaches [TB86, Moh84, KME89, ML90, YZS96, SW96a, SW96b, DDH⁺98, Adv93, CQ93], structural models do not represent an application and a system as separate entities. In structural modeling, **implementations** of an application are represented for a particular resource environment. This is important because in distributed parallel environments the platform and the application are not independent and cannot be altered in an independent manner in the model.

A **structural model** consists of a top-level model, component models, and possibly input parameters. The inputs to a structural model are **input parameters** that represent relevant system and application characteristics. Input parameters can be benchmarks, constants, dynamic system measurements or application attributes, such as problem size or the number of iterations to be executed. The output of a structural model is a **predicted execution time**. Structural models can also be developed for alternative performance metrics, such as speedup or response time. The inputs and outputs of a structural model may be single, numerical *point* values or may be *stochastic values*, as described in the next chapter. The goal of a structural model is to provide an estimate of execution time for a distributed parallel application.

A **top-level model** is defined to be a performance equation consisting of component models and composition operators. Each **component model** is also defined to be a performance equation, parameterized by platform and application characteristics. We define **composition operators** as functions that compose component models, as described below. Top-level models represent the dynamic mix of application tasks, based on the functional structure of the application. All

interactions between modeled components (including overlap) are reflected in the top-level model.

Every task in a top-level model is represented as a component model. **Component models** are defined as performance equations representing the performance of individual tasks of the application implementation. They are comprised of input parameters (benchmarks, constants, etc.) and/or other component models. We define a **task** to be an atomic unit of communication or computation. The exact definition of task varies between applications in the same way the notion of “basic operation” varies between algorithms. For some applications, a task is a function [CS95] or an inner loop where most of the work is done [DS95]. For other applications, a task may be another entire application that can stand on its own [MMFM93].

In this thesis, we use the term **computation task** to denote an atomic unit of computation that is not functionally split between two machines. For example, a data-parallel slave task in a Master-Slave application can be performed on multiple machines simultaneously, but will not be split into two or more constituent sub-tasks, each running on a different machine. We denote a **communication task** to be a logically related set of communication activities. An example of this would be a communication task that enforces the sharing of data between computation phases.

Component models and input parameters are combined using composition operations. We define **composition operators** to be functions that compose the results of component models and input parameters. They represent all interactions between component models. For example, the operator $+$ is used as a combine operation. Compositional operators must be defined for whatever domain the input values of the component models represent. This is discussed in detail when the domain of input parameters are extended to allow for stochastic valued parameters in Chapter 4.

Structural models provide a compromise between a general “one size fits

all” approach and a completely specific “tailor made” approach to developing performance models. “One size fits all” performance models are general, and can be used for a large set of applications, but such models may not be especially accurate for any single application. Alternatively, a “tailor made” model is likely to have a high degree of accuracy for the application it is targeted to, but will be difficult to adapt to fit multiple implementations. The structural modeling approach is meant to provide the flexibility of a general solution through the top-level model, but the accuracy of a specified model via the instantiation of the component models.

3.A.1 Accuracy

The issue of accuracy is fundamental to any modeling approach. Following the literature, we define the **error** of a predictive model to be the absolute difference between the prediction and an actual execution time (wall-clock time). Other possible error metrics to determine accuracy include mean square error or mean percentage error. A model is **accurate** when the error is “small”, where small is a subjective term based on the goals of the user, and generally defined by a threshold value.

The accuracy required from a predictive model varies depending on the use of the resulting prediction. We characterize predictions as accurate in a “relative” sense or an “absolute” sense. A **relatively accurate set of predictions** is a set of predictions that may have a large error associated with them, but that have a valid ranking between them. That is, for a set implementations $\{A_1, A_2, \dots, A_n\}$, predictions $\text{Pred}(A_1), \text{Pred}(A_2), \dots, \text{Pred}(A_n)$, and actual execution times $\text{Act}(A_1), \text{Act}(A_2), \dots, \text{Act}(A_n)$, the predictions are relatively accurate if $\text{Pred}(A_1) > \text{Pred}(A_2) > \dots > \text{Pred}(A_n)$ implies $\text{Act}(A_1) > \text{Act}(A_2) > \dots > \text{Act}(A_n)$. An **absolutely accurate prediction** gives a result within a small error threshold and may or may not result in an accurate ranking. In this case, we have $\text{Pred}(A_i) \pm \text{Threshold} = \text{Act}(A_i)$.

Different degrees of accuracy are appropriate in different instances. For

example, when models are used to generate predictions to select or compare resources, it is enough for them to give a valid relative ranking of resource sets. Using a relative metric has also been effective when comparing different parallel machines for a given application in [WH94] and in finding bottlenecks [MR98]. However, for most scheduling needs, an absolute prediction is a requirement.

In our setting, the accuracy of a prediction can only be determined post-mortem. Without significant additional information, it is difficult to predict the accuracy of the prediction itself. If we know the error contributed by each model component, it may be possible to calculate the overall expected error for the application, however this is not always the case.

3.A.2 Application Profile

For many applications, execution time is concentrated in a subset of the tasks. We use an application profile to prioritize the significance of each task to the overall application execution time. Given an application $APP = \{\text{TASK}_i\}_{i=1}^n$, we define an **application profile** to be a set of tuples of the form: $\{\langle \text{TASK}_i, P_i \rangle\}$, where TASK_i is one task in the set defined in the top-level model, and P_i is the percentage of execution time spent executing in that task, with $\sum P_i \leq 100\%$. This value is less than 100% when some part of the execution time is spent executing part of the application not abstracted as a task, and should be negligible.

The application profile is used to prioritize the tasks according to a user defined metric of “importance”. In many cases, as discussed in the next sections, there is a fairly clear-cut dividing line between tasks that contribute significantly to the application performance and those that do not, according to the application profile. We consider the determination of which tasks are “important” to be a user defined decision.

3.A.3 Building Structural Models

Informally, there are five steps to building structural prediction models. Examples of this method are given in Sections 3.B and 3.C for the GA and SOR codes, respectively.

1. The structure of the application is examined and a **top-level model** is constructed to represent this structure.
2. **Component models** for the set of tasks distinguished in the top-level model are defined.
3. An **application profile** is used to determine which components must be modeled to achieve the required accuracy.
4. Component models are **selected** using available data sources, meta-data and error threshold as guides.
5. The **accuracy** of the model is analyzed and if the model is determined to be inaccurate for its use, a course of action is determined.

3.A.4 Practical Issues

Several issues must be addressed to make it feasible to use structural modeling in a practical setting.

1. How are top-level models defined?

We use an application developer's description of an implementation to construct the top-level model. Often in describing an application, a developer will use a graphical representation such as a program dependency graph [FOW84], a more detailed representation such as Zoom [ASWB95], or even a visual programming language representation such as HeNCE [BDGM93], CODE 2.0 [NB92], Enterprise [SSLP93] or VPE [DN95]. From these graphical representations, defining

a structural model is straightforward. Even when a graphical representation is not available, it is often straightforward to determine the structure of an application at its most coarse-grained level.

2. Where do input parameters come from?

Input parameters are values for application and system characteristics used in the calculation of a component or top-level model. Examples of input parameters are benchmarks or operation counts, an iteration count, a dynamic measurement for bandwidth, or a value for the size of a message. Input parameters for models, both application and system characteristics, can be supplied from several sources. In the MARS system [GR96] this data is obtained by instrumenting the code and using an application monitor to record the application's behavior. System-specific data, such as bandwidth values, CPU capacity, memory sizes, etc., can be supplied by system databases available in most resource management systems (such as the Meta-Computing Information Service (MCIS) for the Globus project [FK97] or the Host Object Database for Legion [Kar96, GWtLt97]) or by on-line tools such as the Network Weather Service [Wol96, Wol97, WSP97] that can supply dynamic values for bandwidth, CPU usage, and memory on a given system. One goal of the structural modeling approach is to use realistically available information when defining the models.

3. What information is generally available for the parameters?

In general, we assume that basic information regarding the bandwidth, latency or startup, and arithmetic operations are available for each system. For each application, we assume that information regarding the structure, task operations and messages is available. Basic system information can be generated using simple benchmarks. For example, on the PCL cluster we generated bandwidth values using a simple PVM benchmark that sent 100 messages from processor x to processor y of a given size, then a 1 byte acknowledgment was sent from processor

y to x . A similar benchmark was constructed to estimate latency and message packing costs on the systems. These benchmarks are given in Table 3.4 and 3.5 for the PCL cluster. Alternatively, an on-line monitoring tool like the Network Weather Service can supply bandwidth values, as was done for the Linux Cluster (see Section 3.B.6).

Similarly, we assume that basic arithmetic operation benchmarks are available. We calculated these using a benchmark that took the average over 25 runs of 10^6 operations of each type of interest, for example, as given in Table 3.2. Both these benchmarks take a minimal effort to run, but provide necessary basic information for modeling.

In addition to the system information, information about the application and its implementation is needed. For the communication portions of the code, we assume that the number and size of the messages is well known. This can often be achieved through static code analysis or dynamic code instrumentation. Likewise, for computation elements, we assume that either there is a basic benchmark for the application on the machines of interest, or an operation count is available. Operation counts of the sort we need can be found using a disassembler like `dis [mp]`, or by using compiler options on some platforms.

If memory usage or disk usage were modeled as well, we would again require some basic information about these properties. One goal of structural modeling was to use easily available information, but without any information we cannot hope to achieve accurate predictions.

4. How do you benchmark a code?

The accuracy of a component model will depend on the accuracy of its input parameters. In general, to benchmark the computation portion of an application in a dedicated environment, we use a single processor version of the code running on a problem size equal to $\frac{\text{medial problem size}}{\text{number of slaves}}$, averaged over 25 runs. The set of problem sizes is determined by the user. We use a benchmark on this prob-

lem size to emulate the conditions an application is likely to experience over the problem size range of interest. Likewise, we take the mean over 25 runs to assure a high degree of accuracy for the input parameter used. Additional benchmarking approaches are presented in Section 3.D.3.

It is possible that adding timing statements to a code in order to benchmark sections of it may perturb the code [MRW92, HM96]. If a large perturbation is indicated, the input parameter for this value should be tagged with meta-data indicating the possible error in the value.

5. How are application profiles defined?

Profiles for applications running on a shared cluster of workstations may be dependent on the architecture, specific implementations of the application, load on relevant machines, problem size, etc. In practice, for a particular environment, these values can be found through profiling tools such as gprof [GKM82] and pixie [Sil], by using static information based on code analysis, or by using dynamic information based on code instrumentation. An estimated value can also be supplied by the application developer. Often, such estimates are accurate enough for the purpose of determining the component models on which to concentrate the modeling effort.

6. How are application models selected?

For most applications, there will be a variety of possible component models for each task defined in the top-level model. Currently all model selection is done by the model developer, but there are several guidelines that can be used. A set of component models must be selected whose composition provides an overall model at the level of accuracy desired. We tried to choose the model with the lowest overhead that will accurately portray an application's behavior. Often, there is a trade-off between the overhead of a model and its accuracy. For example, a task may have two possible component models: one resulting in a high degree of

accuracy that has a high overhead (or a high compute time), and another much simpler (with a lower overhead or compute time) that has a lower degree of accuracy. This trade-off must be evaluated on a per-application and per-model use basis.

One way to reduce the work in modeling is to use the application profile to narrow the number of components to be modeled to those contributing significantly to the execution time. If there are still several choices left, some ways to decide between component models are based on:

- **Available Information.** Component model selection can depend on the sources of information that are available. Different component models are parameterized by different pieces of information and, of course, only the information that is available can be used.
- **Meta-data.** We can associate additional information, meta-data, with an input parameter or component. **Meta-data** is an attribute that describes the determination or content of parameters or predictions [Ber98] and provides a qualitative measure about the data in question. For example, this information could be accuracy, lifetime data, or stochastic(distributional) data, as addressed in the next chapter. These are also called **QoIn**, or quality of information, measures [BW98, BWS97]. Quality of information measures can often help determine which models are more likely to meet a user’s criteria. When developing component models or gathering input parameters, we often “tag”, or associate, meta-data with the values. This is currently done informally, for example, as a comment in the code or benchmark table.
- **Affinity.** A measure or likeness or resemblance between a task and a component model, or **affinity**, can be used to select models. For example, a well structured code with a small inner loop may lend itself to an operation count model.

These are only guidelines. We plan to investigate more specific heuristics in future

work.

7. How can we increase the accuracy of a structural model?

Once an entire structural model is defined, the performance is predicted, and the application is run, a developer can analyze the accuracy of the model and determine a course of action. Depending on the use of the model, different degrees of accuracy may be acceptable. If a model developer determines that a given model is not sufficiently accurate for its use, it may be desirable to modify the model to improve its accuracy. Some possible transformations are:

- **Substitute** a component model. This can be done to avoid using input data that is inaccurate or that has a high variance associated with it.
- **Refine** a component model by using alternative parameters. This can be done when an execution property is not being modeled within a component.
- **Add** other component models. This can be done in accordance with the profile, or when there is evidence that the profile is in error.
- **Re-structure** the top-level model. This can be done when tasks or task interactions are not being captured by the current top-level model approach.

Again, this process is currently done by the model developer until the the overall error and model properties are adequate. Evaluating the exact cause of an error in a prediction is difficult, but we have found that meta-data for the component models and input parameters can be extremely helpful. In addition, if a given application has been modeled previously, there may be information about the component values used and their accuracy. Examples of this process are given in the following sections for several applications.

3.B Modeling the Genetic Algorithm Code

This section describes the development of a structural model for a Genetic Algorithm (GA) Master-Slave code on three different clusters of workstations. To build a structural model, first we build a top-level model, and define the component models it comprises. After determining an application profile, we select component models for the top-level model. Finally, we determine the accuracy of the model. If the model is not accurate enough for the developer's purposes, this process may iterate. We present models for the GA on three dedicated platforms to show the extensibility and flexibility of the structural modeling approach. Multi-user non-dedicated platforms are addressed in the next chapter.

3.B.1 GA Recap

```

Master Computation:
M1  Start Slaves
M2  For i = 1 to MaxIterations
M3    Broadcast entire population to Slave
M4    Receive Data from all Slaves
M5    Sort children
M6    Select next generation
M7  Kill Slaves
M8  Return best child

Slave Computation:
S1  While (True)
S2    Receive population from Master
S3    For j = 1 to NumberofChildren
S4      Randomly pick two parents
S5      Cross parents to create child
S6      Evaluate child
S7    Send all children to Master

```

Figure 3.1: Pseudocode for the Genetic Algorithm application.

A Master-Slave distributed Genetic Algorithm (GA) heuristic for the Traveling Salesman Problem (TSP) was presented in Section 2.C and is reviewed

in the pseudocode in Figure 3.1. The Master task determines (statically) how many children each of P Slave tasks will generate, and then sends the entire population to each of the Slaves using a Scatter routine, in this case a PVM multicast. Each of the Slaves receives the entire population, generates their assigned number of children randomly, and then sends results back to the Master. This sequence iterates for a specified number of generations. In our GA implementation, a Slave process shares a processor with the Master process.

3.B.2 Top-level Model for GA

The top-level model for the GA is structured to reflect the four main components of the application: the Master task, a Scatter, the Slave tasks and a Gather. Therefore, the top-level model has four components. There are a number of possible interactions and synchronizations between component tasks. Three possible top-level models for the GA are:

$$\mathbf{GAExTime1} = \mathbf{Master}(M) + \mathbf{Scatter}(M, P) + \mathit{Max}[\mathbf{Slave}(S_i)] + \mathbf{Gather}(P, M) \quad (3.1)$$

$$\mathbf{GAExTime2} = \mathbf{Master}(M) + \mathbf{Scatter}(M, P) + \mathbf{Slave}(S_i) + \mathbf{Gather}(P, M) \quad (3.2)$$

$$\mathbf{GAExTime3} = \mathbf{Master}(M) + \mathit{Max}[\mathbf{SingleScatter}(M, S_i) + \mathbf{Slave}(S_i) + \mathbf{SingleGather}(S_i, M)] \quad (3.3)$$

where

- **GAExTime** : Execution time for the application.
- M : The processor running the Master process.
- P : The set of Slave processes, S_1, S_2, \dots, S_P .
- S_i : The i th Slave process/processor pair.

- **Master**(M) : Execution time for Master computation (lines M2, M5 and M6 of the pseudocode in Figure 3.1) on processor M .
- **Scatter**(M, P): Execution time for the Master process on M to send data to the set of P Slaves and for it to be received (lines M3 and S2).
- *Max* : A maximum function,
- **Slave**(S_i) : Execution time for Slave computation on processor S_i (line S3 through S6).
- **Gather**(P, M) : Execution time for all P Slaves to send data to the Master on processor M , and for the Master to receive it (lines S7 and M3).
- **SingleScatter**(M, S_i): Execution time for the Master to send data to Slave S_i , and for it to be received.
- **SingleGather**(S_i, M): Execution time for Slave S_i to send data to the Master, and for it to be received.

In a non-dedicated shared environment, each of these may also be parameterized by time in order to reflect the dynamic properties of the system. This chapter addresses models in a dedicated system where time-dependent variations in behavior are typically negligible. In addition, we assume the one-time cost of starting and killing the slaves is insignificant in comparison to the main computation.

Each top-level GA model is based on several assumptions about the interactions of the component tasks. Top-level model 3.1, shown in Figure 3.2, assumes that the time for the Slave tasks is equal to $Max[\mathbf{Slave}(S_i)]$, that is, the amount of time for the total Slave operation is equal to the maximum of all the individual Slave operations. To evaluate this function, all P Slave times must be assessed.

Top-level model 3.2 is an option if the application is load balanced (i.e. each slave processor receives an amount of data so that the computation of each

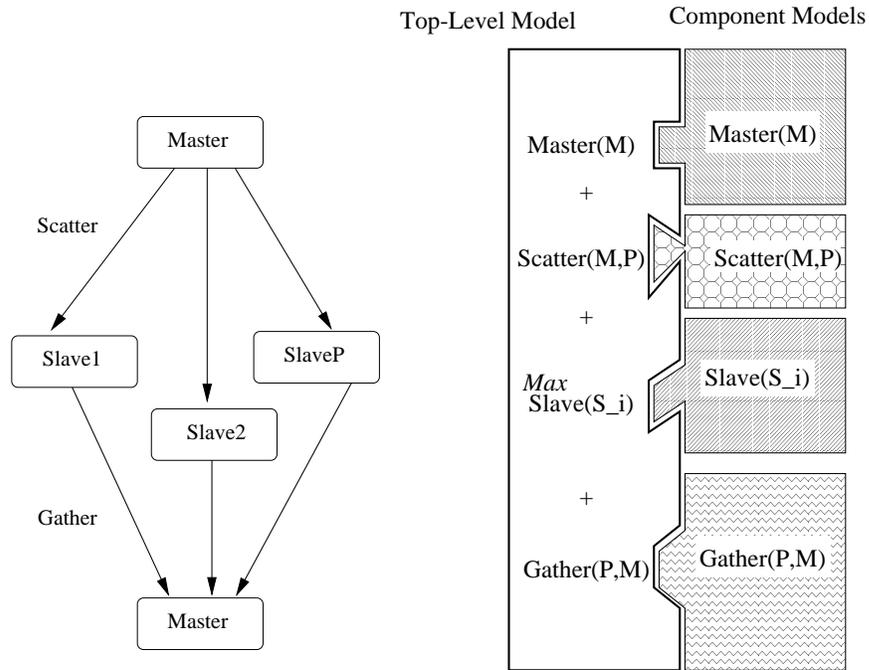


Figure 3.2: Graphic of the top-level model in Equation 3.1 for the Genetic Algorithm Master-Slave application. The lefthand-side is a dependency graph of the application, and the righthand-side is a graphic for the structural model, with the long bar representing the top-level model, and each of the shaded pieces on the far right representing individual component models.

data partition balances the compute times). It has a lower complexity than 3.1 since it only calculates one Slave time.

Both 3.1 and 3.2 assume that there is a synchronization between each task represented by a component model, perhaps implemented by a barrier between each function. This may be the case if the processor running the Master task also has a Slave task, and the application is accurately load balanced. Top-level model 3.3 applies when there is only a synchronization point at the Master component, and the other components overlap. Note that for this top-level model, the Scatter and Gather are defined in terms of a single slave instead of the set.

Each of these top-level models are suitable for Master-Slave applications

with different implementations or environments. Because of the synchronization structure of our GA application, we will use the top-level model given in Equation 3.1 for our implementations.

3.B.3 Modeling GA Computation Components

The second step in building a structural model for the GA is to define possible component models for the four components identified by the top-level model. We first examine the computation component models. Most estimates of computation are based on evaluating some time per data element, and then multiplying by the number of elements being computed for the overall problem. There are two widely used approaches for this: counting the number of operations to compute a single element, and benchmarking. Possible computation component models for the GA based on these approaches are:

$$\mathbf{MasterOpCt}(M) = NumElt(M) * Op(\mathbf{Master}, M) * CPU(M) \quad (3.4)$$

$$\mathbf{SlaveOpCt}(S_i) = NumElt(S_i) * Op(\mathbf{Slave}, S_i) * CPU(S_i) \quad (3.5)$$

$$\mathbf{MasterBM}(M) = NumElt(M) * BM(\mathbf{Master}, M) \quad (3.6)$$

$$\mathbf{SlaveBM}(S_i) = NumElt(S_i) * BM(\mathbf{Slave}, S_i) \quad (3.7)$$

where

- $NumElt(x)$: Number of elements computed by task x , usually available from static or dynamic code analysis, or assigned by the scheduler.
- $Op(\mathbf{TASK}, x)$: Number of operations to compute task \mathbf{TASK} for a single element on processor x , available from using a tool like `dis [mp]`, using compiler options, or static code analysis.
- $CPU(x)$: Dedicated time measurement to perform one operation on processor x , generated with an a priori unit benchmark, such as described in Section 3.A.4, point 3.

- $BM(\text{TASK}, x)$: Time for processor x to compute task **TASK** for one element, usually generated with an a priori benchmark on a dedicated system. Several approaches for benchmarking applications are enumerated in Section 3.A.4, point 4.

The accuracy of a component model depends on the accuracy of its input parameters. To evaluate the $Op(\text{TASK}, x)$ parameters in Equations 3.4 or 3.5, we must obtain an accurate estimate of the operation count. Similarly, to evaluate the $BM(\text{TASK}, x)$ parameters in Equations 3.6 or 3.7, we need accurate benchmarks for the Master and Slave tasks.

If contention on the machines affects the computation times, dedicated time estimates will not suffice. In this case, we may want to refine a component model or its parameters (as described in Section 3.A.4, point 7) by factoring in a value for available CPU. For the Slave task models, we could use the models:

$$\mathbf{ContentionSlaveOpCt}(S_i) = \mathbf{AvailCPU}(S_i) * \mathbf{SlaveOpCt}(S_i) \quad (3.8)$$

$$\mathbf{ContentionSlaveBM}(S_i) = \mathbf{AvailCPU}(S_i) * \mathbf{SlaveBM}(S_i) \quad (3.9)$$

The model for $\mathbf{AvailCPU}(S_i)$ could be a dynamically supplied value, for example from the Network Weather Service [Wol96], or from an analytical model of contention, for example [FB96, LS93, ZY95]. Notice that in this case, $\mathbf{AvailCPU}$ could be a model or a supplied parameter. More adaptations for production systems are discussed in the next chapter.

3.B.4 Modeling GA Communication Components

In addition to the two computational tasks, there are two communication tasks to model, a Scatter and a Gather. On different resource management systems, and on distinct architectures controlled by a single resource management system, both Scatter and Gather routines may be implemented differently.

Examining the Scatter routine first, if we had the top-level model given in Equation 3.1 or 3.2, then we would need to define a model for the Scatter routine

with the entire set of Slaves as the receiving group. There are three straightforward ways to do this. If the Scatter can be represented as a series of sequential sends of equal sizes, then the time it will take to execute will be the sum of P point-to-point sends from the Master to the P Slaves. If the sends are executed in parallel, the time would be the maximum of the P point-to-point sends. A third way to predict the execution time would be to use a benchmark specific to the target system and the number of Slaves. The component models for these three cases are:

$$\mathbf{Scatter1}(M, P) = \sum_{i=1}^P [\mathbf{PtToPt}(M, S_i)] \quad (3.10)$$

$$\mathbf{Scatter2}(M, P) = \mathit{Max} [\mathbf{PtToPt}(M, S_i)] \quad (3.11)$$

$$\mathbf{Scatter3}(M, P) = \mathit{BM}(\mathit{Multicast}, P) \quad (3.12)$$

where

$$\mathbf{PtToPt}(x, y) = \mathit{NumElt}(y) * \frac{\mathit{Size}(Elt)}{\mathit{BW}(x, y)} \quad (3.13)$$

and

- $\mathit{NumElt}(y)$: Number of elements in a message for processor y , usually available from static code analysis.
- $\mathit{Size}(Elt)$: Size of a single element in bytes, available from static code analysis.
- $\mathit{BW}(x, y)$: Bandwidth in bytes per second between processor x and processor y , available from a priori benchmarks or a dynamic measurement system.
- $\mathit{BM}(\mathit{Multicast}, P)$: Benchmark for multicast, parameterized by P , the number of processors in the receiving group (the number of Slaves in this case). This value might be available from benchmark execution experiments or from previous work by the developer.

The component models **Scatter1** (Equation 3.10) and **Scatter2** (Equation 3.11) utilize input parameters, such as the number of Slaves, that must be

supplied by the user or the system, as well as a component model for the point-to-point communication times (**PtToPt**, Equation 3.13). This nested structure is common in structural modeling. **Scatter3** (Equation 3.12) is defined only in terms of the input benchmark for the multicast.

If instead of the top-level model given in Equation 3.1, we were using a top-level model such as Equation 3.3, we would need the values for **SingleScatter**(M, S_i), that is, a single point-to-point message for each Slave. In this case we could use the **PtToPt** model in Equation 3.13:

$$\mathbf{SingleScatter}(M, S_i) = \mathbf{PtToPt}(M, S_i) \quad (3.14)$$

One advantage of the structural modeling approach is the ability to refine a component model and add lower level details without affecting the higher level structure. For any of the communication models, we might want to include other factors that can contribute to execution time, such as startup time or contention. In this case, we can extend the point-to-point communication model, **PtToPt**, with various (non-exclusive) forms of additional information, e.g.:

$$\mathbf{PtToPt2}(x, y) = \mathit{Startup}(x) + \mathbf{PtToPt}(x, y) \quad (3.15)$$

$$\mathbf{PtToPt3}(x, y) = \mathit{AvailBW}(x, y, t) * \mathbf{PtToPt}(x, y) \quad (3.16)$$

where:

- $\mathit{Startup}(x)$: Startup or latency costs for a message on processor x , and
- $\mathit{AvailBW}(x, y, t)$: Fraction of bandwidth available between processor x and processor y at time t .

Again, hierarchical definitions are common when building component models as they allow both flexibility and extensibility as needed for different situations. It should be noted that users can define additional component models as needed.

Similar models could be defined for the Gather as well. For example:

$$\mathbf{Gather1}(P, M) = \sum_{i=1}^P [\mathbf{PtToPt}(S_i, M)] \quad (3.17)$$

$$\mathbf{Gather2}(P, M) = \mathit{Max}[\mathbf{PtToPt}(S_i, M)] \quad (3.18)$$

$$\mathbf{SingleGather}(S_i, M) = \mathbf{PtToPt}(S_i, M) \quad (3.19)$$

Since the top-level model indicated that the Gather and Scatter are collective operations, we can use either Equation 3.17 or 3.18 for the Gather and either Equation 3.10, 3.11 or 3.12 for the Scatter. Combining these possible component models with the two choices for the Master task and the two choices for the Slave task is shown in Figure 3.3. Note that the selection of each component model is independent.

The next three subsections address how the GA's component models are selected for implementations on three dedicated platforms: the PCL Cluster, the Linux Cluster and the SDSC Alpha Farm.

3.B.5 GA Application on the PCL Cluster

In this subsection we demonstrate how component models are selected for the GA application on the PCL cluster. The top-level model on each platform will remain the same, but the underlying component models and input parameters will change for each distinct platform. We verify the validity of each model for its platform experimentally on **dedicated** systems. The next chapter addresses adaptations that must be made for the model to represent application behavior in multi-user production environments.

The goal of these experiments is to show that the structural modeling approach provides the extensibility and flexibility required to accurately represent distributed parallel applications on distinct platforms. Problem sizes for each platform are chosen to be in a range similar to those used at the present time by the application developers. The lower end of the range was generally the smallest problem that could run in 20-30 seconds, as problem sizes smaller than that are not typically used over networks of workstations. The upper end of the range was generally determined by the largest problem size that could fit in core memory for

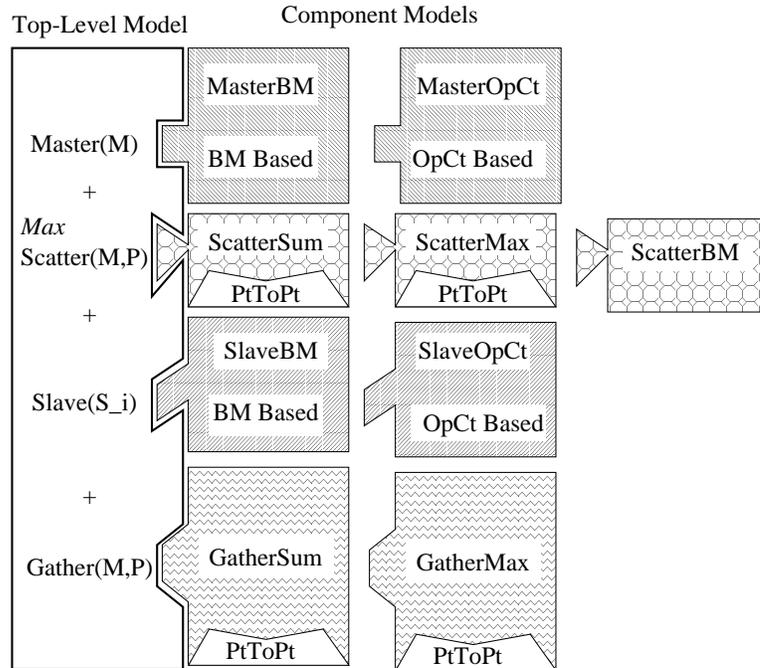


Figure 3.3: Graphic of model options for Genetic Algorithm code. The long white bar represents the top-level model, and each of the patterned components on the right represent possible component models. There are multiple component models available for each task defined in the top-level model. Two of the component models for both the Scatter and Gather include an additional component model for **PtToPt**. Each of these models can be selected independently.

the platform.

We first implemented the GA application on networked resources in the UCSD Parallel Computation Lab. The configuration includes a Sparc 5, a Sparc 10 and two Ultra Sparc's, all running Solaris and connected over Ethernet as described in Section 2.A.

The application developer [Bha96] supplied us with an application profile for this application and platform, given in Table 3.1. It indicated that the Slave task would have the greatest impact on performance, in part due to the slow processor speed of several of the machines, and the fact that the message sizes

Master (M)	2%
Scatter (M, P)	2%
Slave (S_i)	94%
Gather (P, M)	2%

Table 3.1: Profile of GA Code on PCL cluster.

were small and the networks were unloaded. This is a clear cut case of a user deciding on the importance of tasks based on the application profile, as introduced in Section 3.A.2. Therefore, we concentrated on modeling the Slave component (a computation task model).

We analyzed two choices for the Slave component: an operation count model and a benchmark model.

$$\begin{aligned}
 \mathbf{SlaveOpCt}(S_i) &= NumElt(S_i) * Op(\mathbf{Slave}, i) * CPU(S_i) \\
 &= \frac{N}{P} * OP * CPU(S_i)
 \end{aligned}
 \tag{3.20}$$

$$\begin{aligned}
 \mathbf{SlaveBM}(S_i) &= NumElt(S_i) * BM(\mathbf{Slave}, S_i) \\
 &= \frac{N}{P} * BM(\mathbf{Slave}, S_i)
 \end{aligned}
 \tag{3.21}$$

where

- N : The problem size, in this case the size of the population.
- P : The number of processors.
- $NumElt(S_i)$: The number of elements assigned to each slave, in this case $\frac{N}{P}$, as ascertained from static code analysis.
- OP : The number of operations in the Slave routine, 3,745,293 real operations, supplied by the application developer, and tagged with the meta-data that this value may be incorrect due to the non-determinism and irregular structure of the inner loops of the GA.
- $CPU(S_i)$: The time per arithmetic operation. This is given in Table 3.2 for the machines in the PCL cluster,

- $BM(\text{Slave}, S_i)$: Benchmark for the Slave routine, given in Table 3.3.

Machine Name	Integer Addition	Integer Multiplication	Real Addition	Real Multiplication
Thing1	0.8689 sec	2.042 sec	0.8610 sec	0.8746 sec
Thing2	0.854 sec	2.007 sec	0.8541 sec	0.8535 sec
Lorax	1.667 sec	5.850 sec	2.262 sec	2.974 sec
Picard	2.255 sec	6.765 sec	2.755 sec	2.76 sec

Table 3.2: Times in seconds for arithmetic operations on PCL Cluster Machines, means over 25 runs of 1,000,000 operations.

Machine Name	$BM(\text{Slave}, S_i)$
Thing1	$9.9019 * 10^{-3} \frac{\text{sec}}{\text{element}}$
Thing2	$9.8433 * 10^{-3} \frac{\text{sec}}{\text{element}}$
Lorax	$2.4221 * 10^{-2} \frac{\text{sec}}{\text{element}}$
Picard	$3.38396 * 10^{-2} \frac{\text{sec}}{\text{element}}$

Table 3.3: GA Benchmarks on PCL cluster for problem size of 400, mean of 25 values.

Our first choice, driven by the availability of input information, was to model the Slave computation using an operation count model, however, this model was determined to be inaccurate in comparison with actual values as shown in Figure 3.4. It is possible that this was due to inaccurate data for the operation count, as indicated by the meta-data tagged to that input parameter.

Therefore, we chose another component model for the Slave component that avoided the potentially faulty data, and used a benchmark model. The benchmark model achieved results within 7.7% of the actual run times until the problem size began to spill out of core memory at a problem size of 2000. At this point, the error is 25% and increasing. Figure 3.4 shows a comparison of these two models (an operation count model, and a benchmark model) as compared to actual execution time of the GA code. The problem size in the figure is the number of tours.

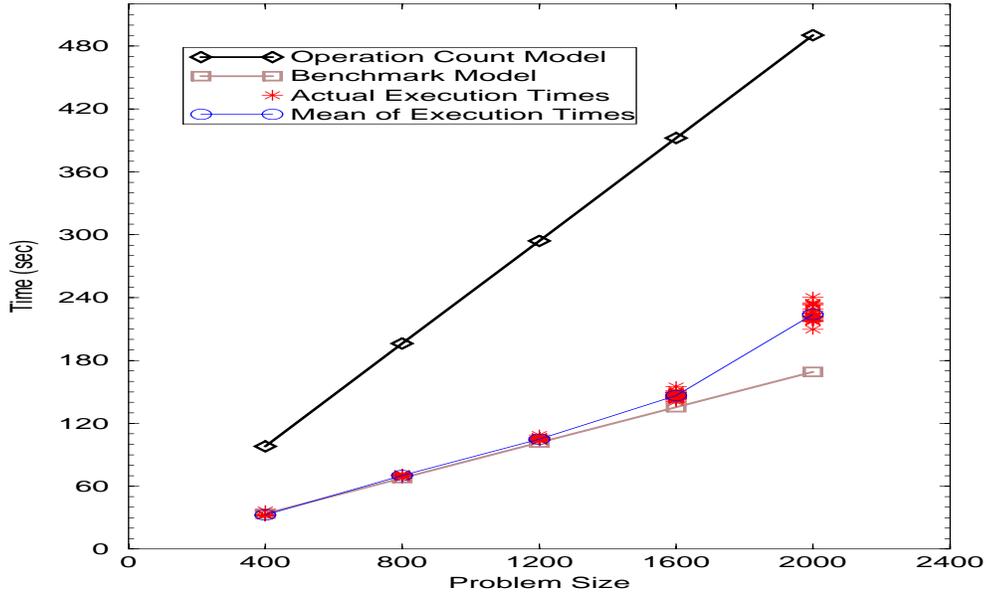


Figure 3.4: Actual times versus model times for GA Code on PCL platform using operation count and benchmark models when communication is set to zero.

If we wanted to accurately model larger (or smaller [Ger98]) problem sizes, we would need more detailed models that address how the execution time is being spent. For example, we might want a different component model for the Slave task when the problem size spills from core memory that would calculate the time to compute the elements that fit within memory, and then the time to compute the rest, which would include a memory access time, for example:

$$\begin{aligned} \mathbf{LargeSlaveBM}(S_i) &= \left(\frac{MemSize(S_i)}{Size(Elt)} * \mathbf{SlaveBM}(S_i) \right) + \\ &\left(\left[\frac{NumElt(S_i)}{Size(Elt)} - \frac{MemSize(S_i)}{Size(Elt)} \right] * [MemAccess(S_i) + \mathbf{SlaveBM}(S_i)] \right) \end{aligned} \quad (3.22)$$

Other formulas are possible as well. While more accurate, use of this version for the component model would add overhead and complexity to the computation of the top-level model and is unnecessary for the problem sizes that fit within core memory.

We also evaluated several models for the communication components:

$$\begin{aligned}
\mathbf{Scatter1}(M, P) &= \sum_{i=1}^P [\mathbf{PtToPt}(M, S_i)] \\
&= \sum_{i=1}^P \left[NumElt(S_i) * \frac{Size(Elt)}{BW(M, S_i)} \right] \\
&= \sum_{i=1}^P \left[N * \frac{24bytes}{BW(M, S_i)} \right]
\end{aligned} \tag{3.23}$$

$$\begin{aligned}
\mathbf{Scatter2}(M, P) &= \sum_{i=1}^P [\mathbf{PtToPt2}(M, S_i)] \\
&= \sum_{i=1}^P \left[\left\{ NumElt(S_i) * \frac{Size(Elt)}{BW(M, S_i)} \right\} + Startup(M) \right] \\
&= \sum_{i=1}^P \left[\left\{ N * \frac{24bytes}{BW(M, S_i)} \right\} + Startup(M) \right]
\end{aligned} \tag{3.24}$$

$$\begin{aligned}
\mathbf{Gather1}(P, M) &= \sum_{i=1}^P [\mathbf{PtToPt}(S_i, M)] \\
&= \sum_{i=1}^P \left[NumElt(S_i) * \frac{Size(Elt)}{BW(S_i, M)} \right] \\
&= \sum_{i=1}^P \left[\frac{N}{P} * \frac{24bytes}{BW(S_i, M)} \right]
\end{aligned} \tag{3.25}$$

$$\begin{aligned}
\mathbf{Gather2}(P, M) &= \sum_{i=1}^P [\mathbf{PtToPt2}(S_i, M)] \\
&= \sum_{i=1}^P \left[\left\{ NumElt(S_i) * \frac{Size(Elt)}{BW(S_i, M)} \right\} + Startup(S_i) \right] \\
&= \sum_{i=1}^P \left[\left\{ \frac{N}{P} * \frac{24bytes}{BW(S_i, M)} \right\} + Startup(S_i) \right]
\end{aligned} \tag{3.26}$$

where

- N : Problem size.
- P : Number of processors.
- $NumElt(S_i)$: Number of elements in a message. For Scatter this is the entire population, or N ; for the Gather, each message has the updated portion of the population or $\frac{N}{P}$.

- $Size(Elt)$: The size of an element, in this case 3 doubles, or 24 bytes, from static code analysis.
- $BW(x, y)$: Bandwidth between processor x and y , shown in Table 3.4, from an a priori benchmark.
- $Startup(x)$: The startup message time on processor x , in this case including both latency and packing cost, given in Table 3.5, from an a priori benchmark.

From / To	Thing1	Thing2	Lorax	Picard
Thing1	11.94	6.7854	0.9633	0.9871
Thing2	7.1556	12.50	0.9564	0.9885
Lorax	0.8219	0.8240	2.5205	0.831
Picard	0.9817	0.9864	0.8994	3.4919

Table 3.4: Bandwidth between processors of the PCL Cluster in MBytes/Sec. These values were generated using a benchmark that sends 100 messages of a specified message size from processor x to processor y , and a single message consisting of a one byte acknowledgment from processor y to processor x after 100 messages.

	Thing1	Thing2	Lorax	Picard
Thing1	354+0.0065 n	439+0.00588 n	1877+0.00613 n	1142+ 0.00550 n
Thing2	436+0.00588 n	352+0.0060 n	1710+0.0060 n	1153+0.00575 n
Lorax	1596+0.0523 n	1584+ 0.0546 n	1922+0.0538 n	2978+0.0609 n
Picard	1140+0.0429 n	1148+0.0391 n	3203+0.0406 n	1489+0.0433 n

Table 3.5: Startup costs for messages on PCL Cluster in μ Sec, equal to Latency plus Packing time for message of n bytes.

Using distinct communication components in the top-level model had no effect on the accuracy of the prediction, most likely because the application spends very little of the overall time communicating in this model.

3.B.6 GA on Linux Cluster

Moving the GA application to the Linux cluster involved changing the component models. On this cluster, the application had a different profile, as shown in Table 3.6. Since the machines on this cluster are much faster than those in the PCL cluster, but the bandwidth does not change as much, the communication portions of the code became much more critical to the performance of the application. With the faster processors, only about two-thirds of the time was spent doing computation, almost entirely in the Slave tasks, and the rest was spent on communication. Of the communication costs, two thirds of the time was spent in the Scatter, and one third is spent in the Gather.

Master (M)	2%
Scatter (M, P)	22%
Slave (S_i)	64%
Gather (P, M)	11%

Table 3.6: Profile of GA Code on Linux cluster.

The top-level model did not change, however the available component models did. We no longer had an operation count model to base **SlaveOpCt** on, due to the change in system (and therefore, change in the number of operations an inner loop would have) and the decision to not re-evaluate this value on a new platform due to the large error on the previous platform. The following component models were used for the Linux cluster:

$$\begin{aligned}
 \mathbf{Scatter}(M, S_i) &= P * \mathbf{PtToPt}(M, S_i) \\
 &= P \left[\mathit{NumElt}(S_i) * \frac{\mathit{Size}(Elt)}{\mathit{BW}(M, S_i)} \right] \\
 &= P \left[N * \frac{3 \text{ doubles} * (8 \text{ bytes/double})}{2.6125 \text{ MBytes/sec}} \right]
 \end{aligned} \tag{3.27}$$

$$\begin{aligned}
 \mathit{Max}[\mathbf{Slave}(S_i)] &= \mathit{NumElt}(S_i) * \mathit{BM}(\mathbf{Slave}, S_i) \\
 &= \frac{N}{P} * (2.5987 * 10^{-3}) \text{sec}
 \end{aligned} \tag{3.28}$$

$$\begin{aligned}
\mathbf{Gather}(S_i, M) &= P * \mathbf{PtToPt}(S_i, M) \\
&= P \left[NumElt(M) * \frac{Size(Elt)}{BW(S_i, M)} \right] \\
&= P \left[\frac{N}{P} * \frac{3 \text{ doubles} * (8 \text{ bytes/double})}{2.6125 \text{ MBytes/sec}} \right]
\end{aligned} \tag{3.29}$$

where

- N : The problem size.
- P : The number of processors.
- $NumElt(S_i)$: The number of elements. For Scatter this is N , for the Slave task this is $\frac{N}{P}$, for the Gather this is $\frac{N}{P}$. This came from static code analysis.
- $Size(Elt)$: Size of a single element, 3 doubles or 24 bytes.
- $BW(x, y)$: Bandwidth between processor x and y . We used a value of 2.6125 MBytes/sec for this since all the processors were the same. This value was supplied by the Network Weather Service as an average for a quiescent system.
- $BM(\text{Slave}, S_i)$: Benchmark for Slave computation on processor S_i . This was supplied by single processor runs of the code on a problem size of 1250 (since the medial problem size of interest is 5000 and the number of processors is 4), equal to $2.5987 * 10^{-3}$ seconds per element.

Figure 3.5 shows actual times versus modeled times using the benchmark slave model with and without communication models models set to zero. With the communication models, we achieve predictions within 8.5% of the mean actual values until the large increase in execution time, most likely due to a spill from in-core memory after the 8,000 problem size. At this point, the error is almost 20% and increasing.

In summary, on the Linux platform we were able to use a similar benchmark model for the slave component to the one used for the PCL cluster with

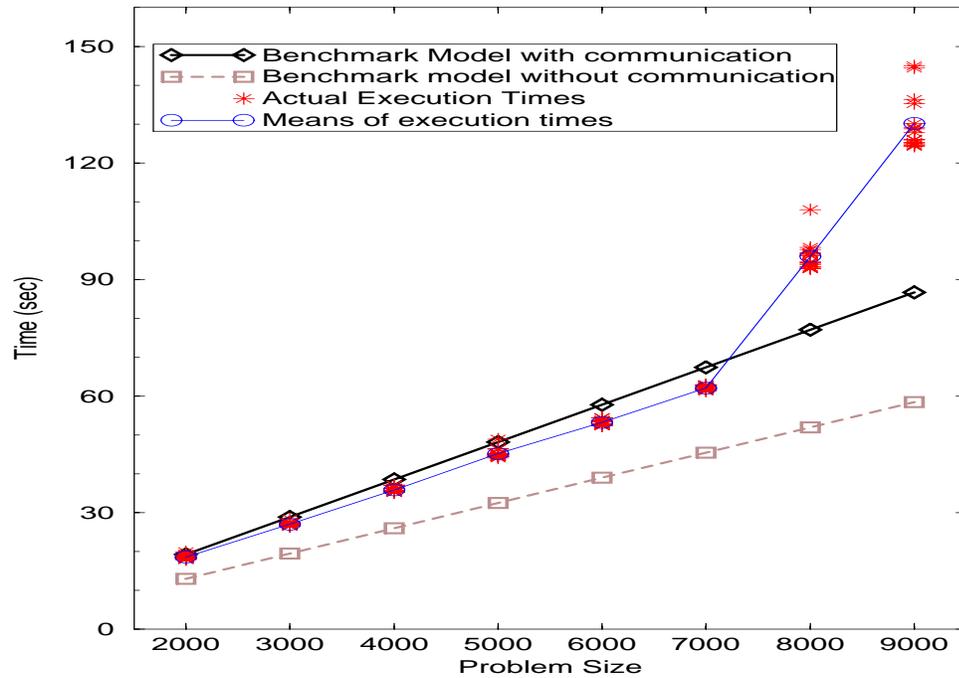


Figure 3.5: Actual times versus model times for GA Code on Linux cluster platform using benchmark slave model with and without communication models set to zero.

modified benchmarks, but it was necessary to analyze the communication components as well (as opposed to setting them equal to zero), as indicated by the application profile.

3.B.7 GA on Alpha Farm Cluster

Moving to the Alpha farm, we were back to the situation of having slower processors relative to the network speeds, and the application profile changed, as shown in Table 3.7. Like on the PCL cluster, most of the time is spent computing the Slave tasks. The top-level model did not change. We used the component model given in Equation 3.30.

Master (M)	4%
Scatter (M,P)	3%
Slave (S_i)	91%
Gather (P,M)	2%

Table 3.7: Profile of GA Code on Alpha Farm cluster.

$$\begin{aligned}
Max[\mathbf{Slave}(\mathbf{S}_i)] &= Max[NumElt(S_i) * BM(\mathbf{Slave}, S_i)] \\
&= Max[\frac{N}{P} * 2.0369 * 10^{-2} sec]
\end{aligned}
\tag{3.30}$$

where

- N : The problem size.
- P : The number of processors.
- $NumElt(S_i)$: The number of elements, for the Slave task this is $\frac{N}{P}$. This came from static code analysis.
- $Size(Elt)$: Size of a single element, $20 * N$ bytes.
- $BM(\mathbf{Slave}, S_i)$: A benchmark for the GA, supplied by single processor runs of the code on a problem size of 312, which is the medial problem size divided by 8 slaves, equal to $2.0369 * 10^{-2}$ seconds per element.

Using these models, we achieved predictions within 6.5%, as shown in Figure 3.6.

3.B.8 Discussion

This section presented the full development of a structural model for a Master-Slave GA application. Choices for the top-level model were analyzed and the top-level model in Equation 3.1 was chosen based on the interactions and synchronization it represented. Component models for the computation and communication tasks depicted in the top-level model were outlined, and then fully

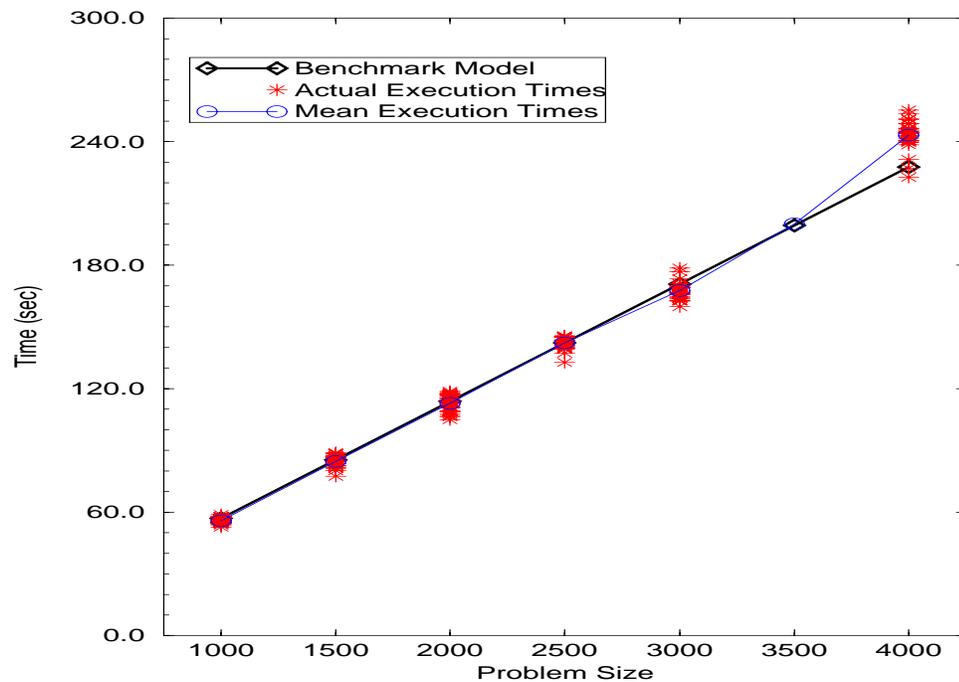


Figure 3.6: Actual times versus model times for GA Code on Alpha Farm platform using benchmark models where **Scatter**=0 and **Gather** = 0.

defined for each of three dedicated platforms. Tables 3.8, 3.9, 3.10 and 3.11 show a summary of the component models and their input parameters. In the following sections we will show only the tables and not the model derivation for each platform.

The goal of the experiments in these subsections was to show the flexible and adaptive nature of structural modeling. We adapted models for three platforms easily, and in each case, achieved good predictions of the actual values for a wide range of problem sizes. The same top-level model was used in each case, only the lower level details in the component models were adjusted to fit the new implementations.

	SlaveBM(S_i)	
	$NumElt(S_i)$	$BM(Slave, S_i)$
PCL	$\frac{N}{P}$	Table 3.3
Linux Cluster	$\frac{N}{P}$	$2.5987 * 10^{-3}$ sec
Alpha Farm	$\frac{N}{P}$	$2.0369 * 10^{-2}$ sec

Table 3.8: Summary of the computation component models for **SlaveBM** used for the GA application on three dedicated platforms.

	SlaveOpCt(S_i)		
	$NumElt(S_i)$	$Op(Slave, S_i)$	$CPU(S_i)$
PCL	$\frac{N}{P}$	OP	Table 3.2

Table 3.9: Summary of the computation component model for **SlaveOpCt** used for the GA application.

	Scatter		
	$NumElt(S_i)$	$Size(Elt)$	$BW(M, S_i)$
PCL	N	24 bytes	Table 3.4
Linux	N	24 bytes	2.6125 MBytes/sec

Table 3.10: Summary of the **Scatter** component model used for the GA application on two dedicated machines.

	Gather		
	$NumElt(S_i)$	$Size(Elt)$	$BW(S_i, M)$
PCL	$\frac{N}{P}$	24 bytes	Table 3.4
Linux	$\frac{N}{P}$	24 bytes	2.6125 MBytes/sec

Table 3.11: Summary of the **Gather** component model used for the GA application on two dedicated machines.

3.C Modeling the SOR Benchmark

This section details building a model for a Successive Over-Relaxation (SOR) benchmark, a member of the Regular SPMD application class. In the SOR implementation, the application is divided into “red” and “black” phases, with communication and computation alternating for each [Bri87]. This repeats for a predefined number of iterations, as described in Section 2.E.1. For this application, like the GA, we implemented the code in three dedicated settings.

3.C.1 Top-level Model for SOR

Given the four-task structure of the SOR code, possible top-level models for this application include:

$$\begin{aligned} \mathbf{SORExTime1} &= \\ & \mathit{NumIts} (\mathit{Max} [\mathbf{RedComp}(i) + \mathbf{RedComm}(i) + \\ & \qquad \qquad \qquad \mathbf{BlackComp}(i) + \mathbf{BlackComm}(i)]) \end{aligned} \quad (3.31)$$

$$\begin{aligned} \mathbf{SORExTime2} &= \\ & \mathit{NumIts} (\mathit{Max}[\mathbf{RedComp}(i)] + \mathit{Max}[\mathbf{RedComm}(i)] + \\ & \qquad \qquad \mathit{Max}[\mathbf{BlackComp}(i)] + \mathit{Max}[\mathbf{BlackComm}(i)]) \end{aligned} \quad (3.32)$$

$$\begin{aligned} \mathbf{SORExTime3} &= \sum_{t=1}^{\mathit{NumIts}} [\mathbf{Iteration}(t)] \quad \textit{where} \\ \mathbf{Iteration}(t) &= \\ & \mathit{Max}[\mathbf{RedComp}(i, t)] + \mathit{Max}[\mathbf{RedComm}(i, t + \Delta_1)] + \\ & \mathit{Max}[\mathbf{BlackComp}(i, t + \Delta_2)] + \mathit{Max}[\mathbf{BlackComm}(i, t + \Delta_3)] \end{aligned} \quad (3.33)$$

and where

- NumIts : The number of iterations.
- Max : A maximum function.

- **RedComp**(i): Execution time for the Red Computation phase on processor i .
- **RedComm**(i): Time to send and receive receive data between processor i and its neighbors during the Red phase.
- **BlackComp**(i): Execution time for the Black Computation phase on processor i .
- **BlackComm**(i): Time to send and receive receive data between processor i and its neighbors during the Black phase.

In the top-level equation given in Equation 3.31, the time for a single iteration is equivalent to the time for the slowest processor to complete that iteration, assuming that there is synchronization only once an iteration.

In our implementation, this code is decomposed into strips and employs a five-point stencil communication, so synchronization occurs only between processors sharing data, not over the entire cluster. Note that it is possible for processors that are not neighbors to become unsynchronized. For example, when data is decomposed into strips, delays in communication between a processor executing data strip S_i and its neighbor executing S_{i+1} can retard communication between the processor executing strip S_{i+1} and the processor executing strip S_{i+2} . In particular, accumulating communication delays can create a kind of “skew” that can delay execution over all strips for each iteration. This is depicted in Figure 3.7. If the work is load balanced across all processors to fit their execution capacities, the skew between processors may be reduced.

If skew were to become an issue for an application, we could use the top-level model given in Equation 3.32 instead of 3.31. This would allow for the delays due to skew to be accounted for by calculating the maximum time for each component individually.

If skew was very severe or unpredictable, due to contention or poor load-balancing for example, an application developer might try to use predictions of

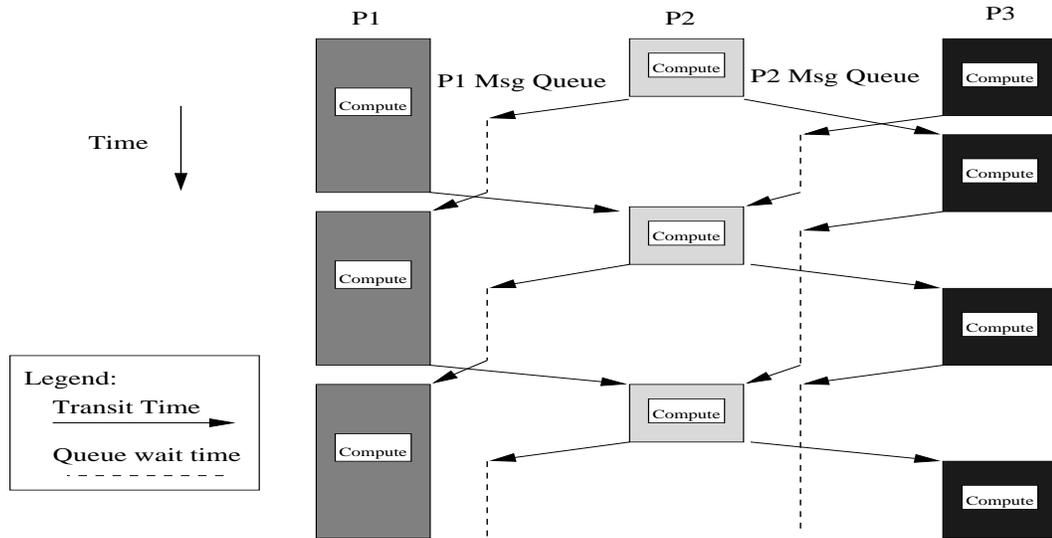


Figure 3.7: Skew can be generated by un-balanced processors sharing data. In this figure, the dashed lines indicated message wait time in the message queue. Note that as P1 affects P2, then P2's wait times affect the start times for P3.

this contention from an on-line monitor such as the Network Weather Service. In this case, the top-level model given in Equation 3.33 with its additional time parameterization might be used.

In the following, we demonstrate the development of a model for SOR using the top-level model from Equation 3.31 which is well suited to our implementations and environment.

3.C.2 Component Models for SOR

Once we decide on a top-level model, we need to define the underlying component models for the computation and communication. The two communication models, unlike the Gather and Scatter for a Master-Slave application, contain both sends and receives. Component models for them are:

$$\mathbf{RedComm}(i) = \mathbf{SendLR}(i) + \mathbf{ReceLR}(i) \quad (3.34)$$

$$\mathbf{BlackComm}(i) = \mathbf{SendLR}(i) + \mathbf{ReceLR}(i) \quad (3.35)$$

where

$$\mathbf{SendLR}(i) = \mathbf{PtToPt}(i, i + 1) + \mathbf{PtToPt}(i, i - 1) \quad (3.36)$$

$$\mathbf{ReceLR}(i) = \mathbf{Process}(i + 1) + \mathbf{Process}(i - 1) \quad (3.37)$$

and

$$\mathbf{Process}(x) = \mathit{NumElt}(x) * \mathit{Size}(Elt) * \mathit{Unpack}(x) \quad (3.38)$$

where

- **PtToPt**: as defined in Equation 3.13.
- $\mathit{Unpack}(x)$: Time to unpack a byte for processor x .

For computation, we can use models similar to those from the GA application:

$$\mathbf{RedCompOpCt}(i) = \mathit{NumElts}(i) * \mathit{Op}(\mathbf{Red}, i) * \mathit{CPU}(i) \quad (3.39)$$

$$\mathbf{BlackCompOpCt}(i) = \mathit{NumElts}(i) * \mathit{Op}(\mathbf{Black}, i) * \mathit{CPU}(i) \quad (3.40)$$

$$\mathbf{RedCompBM}(i) = \mathit{NumElts}(i) * \mathit{BM}(\mathbf{Red}, i) \quad (3.41)$$

$$\mathbf{BlackCompBM}(i) = \mathit{NumElts}(i) * \mathit{BM}(\mathbf{Black}, i) \quad (3.42)$$

where

- $\mathit{NumElt}(i)$: Number of elements calculated by processor i .
- $\mathit{Op}(\mathbf{TASK}, i)$: Number of operations to compute task **TASK** for a single element on processor i .
- $\mathit{CPU}(i)$: Time to perform one operation on processor i .
- $\mathit{BM}(\mathbf{TASK}, i)$: Time for processor i to compute task **TASK** for one element.

3.C.3 SOR on the PCL Cluster

Our initial application profile for the PCL cluster, shown in Table 3.12, suggested the SOR would spend most of its execution time computing the RedComp and BlackComp tasks on the network of machines due to the relative slowness of several workstations compared to the speed of the unloaded network. Therefore component models for the two computation components are particularly important to the overall performance of the top-level model.

RedComp (i)	47%
RedComm (i)	3%
BlackComp (i)	47%
BlackComm (i)	3%

Table 3.12: Application Profile for SOR benchmark on PCL cluster.

In order to estimate the computation for the slave task, we had several choices. We could use an operation count model (as depicted in Equation 3.39 and shown parameterized in Equation 3.45) or a benchmark model (as depicted in 3.41, and shown parameterized in Equation 3.45). Using the benchmark model, we have two choices for benchmarking the SOR code: $BM(\text{Red1P},i)$ in Equation 3.44, a benchmark based on running the SOR for one processor at a problem size of $\frac{\text{medial problem size}}{\text{number of slaves}}$ number of slaves, and $BM(\text{RedNC},i)$ in Equation 3.43. a benchmark based on running a version of the SOR code that had no communication in it for the medial problem size of the entire application space. Using the RedComp as an example, the models are:

$$\begin{aligned}
 \text{RedCompBMNC}(i) &= \text{NumElt}(i) * BM(\text{RedNC}, i) \\
 &= (N * \frac{N}{P}) * BM(\text{RedNC}, i)
 \end{aligned}
 \tag{3.43}$$

$$\begin{aligned}
\mathbf{RedCompBM1P}(i) &= NumEltts(i) * BM(\mathbf{Red1P}, i) \\
&= (N * \frac{N}{P}) * BM(\mathbf{Red1P}, i)
\end{aligned} \tag{3.44}$$

$$\begin{aligned}
\mathbf{RedCompOpCt}(i) &= NumEltts(i) * Op(\mathbf{Red}, i) * CPU(i) \\
&= (N * \frac{N}{P}) * OP * CPU(i)
\end{aligned} \tag{3.45}$$

where

- N : The problem size.
- $NumEltts(i)$: The number of elements calculate by processor i . This is equal to $N * \frac{N}{P}$, or $\frac{N}{P}$ columns each of size N .
- $BM(\mathbf{RedNC}, i)$: Benchmark for Red Computation. This is based on N processor with the no communication benchmark on processor i , given in Table 3.13. mean over 25 runs.
- $BM(\mathbf{Red1P}, i)$: Benchmark for Red Computation. This is based on the 1-processor benchmark on processor i , given in Table 3.14.
- $Op(\mathbf{Red}, i)$: Operation count for Red Computation. This is obtained from dynamic code analysis.
- $CPU(i)$: Time per operation on processor i . This is given in Table 3.2 in the last section, from a priori benchmark runs.

For this set of experiments, shown in Figure 3.8, the one-processor benchmark model was within 5% of actual values, the zero-communication benchmark model was within 11% of actual values, and the operation count model achieved predictions within 7.5% of actual execution time, all until the problem size spilled from memory. In order to model larger problem sizes, the model would need to be

Machine	Time
Thing1	$1.4466 * 10^{-6} \frac{sec}{point}$
Thing2	$1.4319 * 10^{-6} \frac{sec}{point}$
Lorax	$4.2722 * 10^{-6} \frac{sec}{point}$
Picard	$3.9678 * 10^{-6} \frac{sec}{point}$

Table 3.13: Benchmark for RedComp and BlackComp based on SOR benchmark using no communication version of SOR for problem size of 2000 over 4 processors, with a mean of 25 runs.

Machine	Time
Thing1	$1.0714 * 10^{-6} \frac{sec}{point}$
Thing2	$1.0423 * 10^{-6} \frac{sec}{point}$
Lorax	$3.0973 * 10^{-6} \frac{sec}{point}$
Picard	$3.8313 * 10^{-6} \frac{sec}{point}$

Table 3.14: Benchmark for RedComp based on SOR benchmark for one processor at problem size of 500.

modified to included memory. Note that this also demonstrates the error in assuming a flat memory, as Gustafson described in [Gus90]. This model is also inaccurate for the smallest problem sizes where factors that are amortized for larger problem sizes become much more apparent [Ger98]. As such, this application reinforces the usability of this approach in identifying models within a given problem range, and allowing flexibility as needed over a range of problem sizes and/or implementations.

3.C.4 SOR on Linux Cluster and SDSC Alpha Farm

We ported the SOR benchmark from the PCL cluster to the Linux cluster and Alpha farm platforms. In these setting, the models were roughly the same as in the PCL setting. The application profiles for these platforms are given in Table 3.15. The task component models for the top-level SOR model repre-

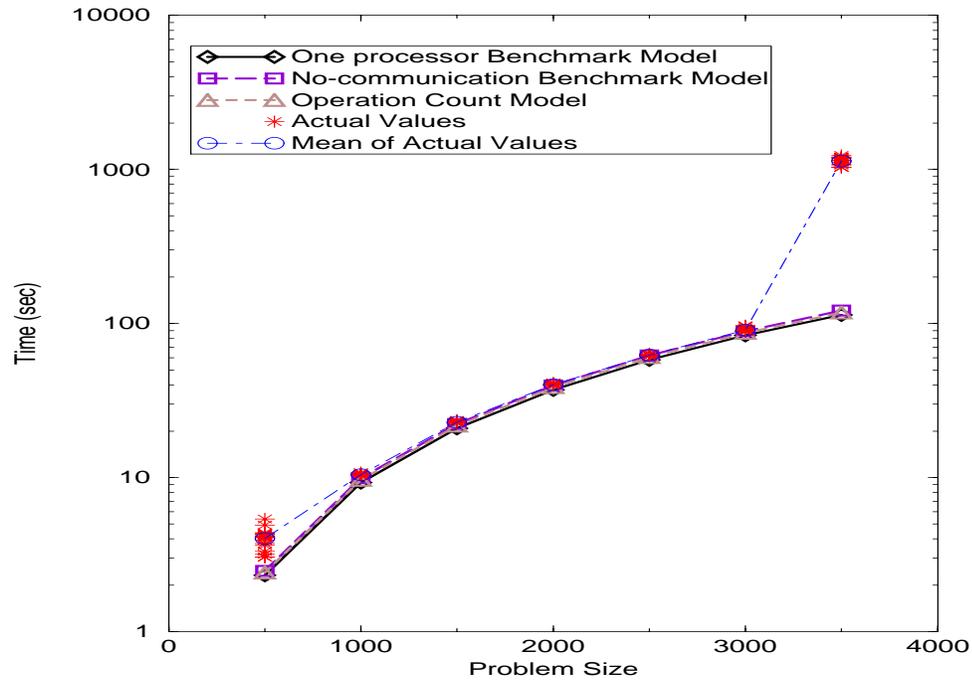


Figure 3.8: Graph of SOR on dedicated PCL Cluster showing results of actual execution times and models. Note that this graph is on a log-linear scale to show the error in the highest and lowest problem sizes.

sented in Equation 3.31, are given with platform-specific parameterizations in Tables 3.16, 3.17, and 3.18, referring to Equations 3.39 through 3.42 for computation, and referring to Equations 3.34 through 3.38 for communication.

For these systems, benchmark values for the computation components were calculated by running the SOR on a single processor on a problem size of $\frac{\text{medial value}}{\text{number of slaves}}$, and then averaging 25 runs. Since the processors within the Linux cluster and the Alpha farm are identical, the same value can be used for all of them. Values for the operation count models were found using the `-s` option when compiling the code on their respective platforms. Benchmarks for the time per operation were found by running a simple benchmark that computed 10^6 operations, and averaging them. Bandwidth numbers for the Linux cluster were supplied by the Network Weather Service running on a quiescent system, and values for $Unpack(i)$

	Linux Cluster	Alpha Farm
RedComp (i)	38%	49%
RedComm (i)	12%	1%
BlackComp (i)	38%	49%
BlackComm (i)	12%	1%

Table 3.15: Application Profile for SOR benchmark on Linux cluster and Alpha Farm.

	CompBM (i)	
	$NumElt(S_i)$	$BM(\mathbf{Red}, i)$
Linux Cluster	$N * \frac{N}{P}$	$4.68345 * 10^{-6}$
Alpha Farm	$N * \frac{N}{P}$	$4.53196 * 10^{-6}$

Table 3.16: Summary of the computation component models for benchmark models for RedComp used in defining the SOR structural model on two dedicated platforms, exact component model definitions are given in Equations 3.41 and 3.42.

were found using a simple PVM benchmark to measure the time to unpack a message.

Figure 3.9 shows the resulting models and actual execution times for each problem size on the Linux cluster, and Figure 3.6 shows the same for the Alpha farm. The former achieved results within 4% for the benchmark model, but the operation count model was off by 30%. Both Alpha farm models were within 6.5% of the actual values.

3.C.5 Discussion

In this section we derived component models for the Regular SPMD Successive Over-Relaxation code on three dedicated platforms. The flexibility of structural modeling was shown by the ease of adaption from one platform to another, adding component models when indicated by the application profile. Several types of benchmarks were used.

CompOpCt(i)					
	$NumElt(S_i)$	$Op(\text{Red}, i)$		$CPU(i)$	
	$NumElt(S_i)$	RealAdds	RealMults	RealAdds	RealMults
Linux	N	3	2	$2.896 * 10^{-6}$	$0.1997 * 10^{-6}$
Alpha	N	5	3	$4.532 * 10^{-6}$	$0.2281 * 10^{-6}$

Table 3.17: Summary of the computation component models for benchmark models for RedComp used in defining the SOR structural model on two dedicated platforms, exact component model definitions are given in Equations 3.39 and 3.40.

RedComm(i)					
SendLR(i)				ReceLR(i)	
PtToPt(i, i+1)		PtToPt(i, i+1)		Process(i)	Process(i)
NumElt(i)	Size(Elt)	BW(i, i+1)	BW(i, i-1)	UnPack(i)	
N	8 Bytes	$2.6125 \frac{MBytes}{sec}$	$2.6125 \frac{MBytes}{sec}$	$3.713 * 10^{-3} \frac{sec}{Mbyte}$	

Table 3.18: Summary of the computation component models for benchmark models for RedComm used in defining the SOR structural model on the dedicated Linux cluster, exact component model definitions are given in Equations 3.34 through 3.38.

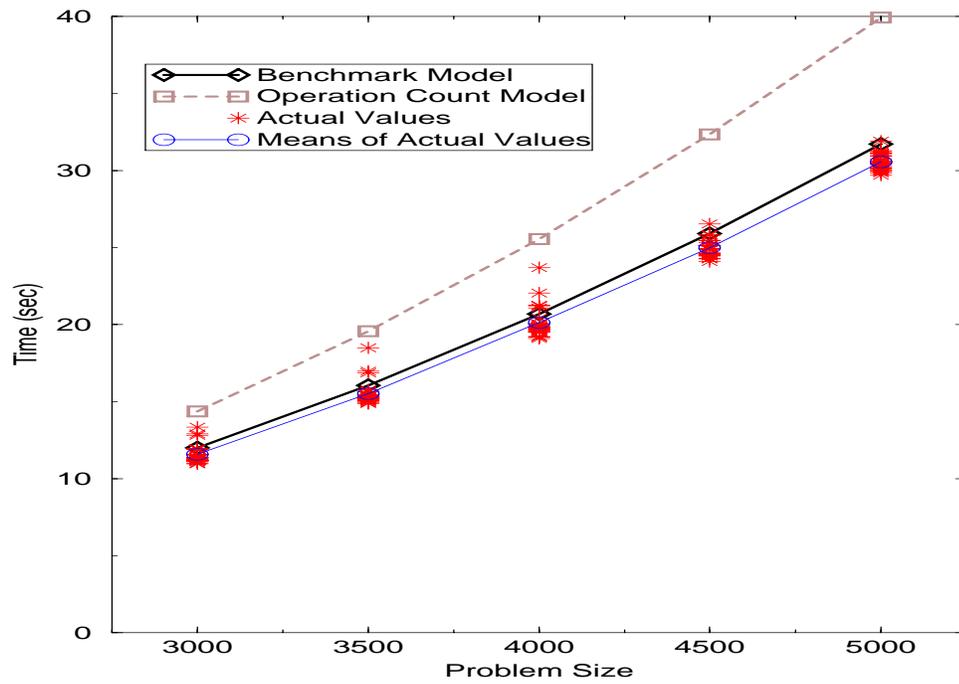


Figure 3.9: Actual times versus benchmark and operation count model times for SOR benchmark on dedicated Linux cluster.

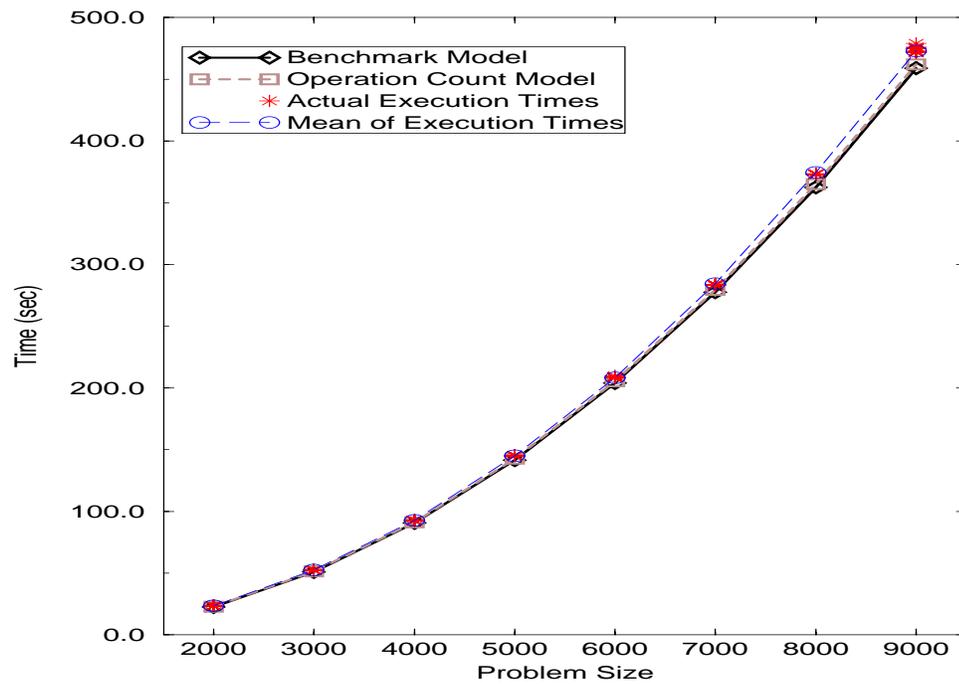


Figure 3.10: Actual times versus model times for SOR Code on Alpha Farm platform using benchmark model and operation count model (communication = 0).

3.D Other Applications

To show the flexibility and accuracy of the structural modeling approach, we implemented a test suite of applications over the three dedicated platforms. In addition to demonstrating the flexibility of the structural modeling approach, we use these applications to demonstrate courses of action to take when a model does not meet the accuracy requirement of the model developer, for example in the case of the N-Body code, presented in Section 3.D.1. In section 3.D.2 we demonstrate adjustments needed for dynamic load balancing using the HiTemp code. The LU code (Section 3.D.3) and the SP code (Section 3.D.4) also demonstrate additional benchmarking techniques.

3.D.1 Modeling the N-body Code

As part of the class of Master-Slave codes, we implemented a simple N-body simulation, also known as a particle simulation, to simulate the motion of planetary bodies, discussed in Section 2.C. For each iteration, the Master sends the entire set of particles to each Slave, and also assigns a portion of the particles to each processor to calculate the next iteration of values. The Slaves calculate the new positions and velocities for their assigned particles, and then send this data back to the Master process.

This implementation had a Slave process running on the Master processor, and the top-level model was:

$$\begin{aligned} \mathbf{NBodyExTime} = \\ \mathbf{Master}(M) + \mathbf{Scatter}(M, P) + \mathit{Max}[\mathbf{Slave}(S_i)] + \mathbf{Gather}(M, P) \end{aligned} \quad (3.46)$$

As with the GA application, component models were determined for each platform as needed. The basic models possible were similar to those already described in the previous section, and are given in Tables 3.20 through 3.23. The application profile for the PCL and Linux clusters is presented in Table 3.19.

	PCL Cluster	Linux Cluster
Master (M)	5%	3%
Scatter (M,P)	8%	29%
<i>Max</i> [Slave (S_i)]	85%	55%
Gather (M,P)	2%	13%

Table 3.19: Profile for N-Body code on the PCL cluster and the Linux cluster.

SlaveOpCt(S_i)					
	$NumElt(S_i)$	$Op(Red, i, Elt)$		$CPU(i)$	
	$NumElt(S_i)$	RealAdds	RealMults	RealAdds	RealMults
PCL	$\frac{N*N}{P}$	9	12	Table 3.2	

Table 3.20: Summary of the computation component model for operation count used for the N-body application.

The application profile on the PCL cluster indicated that communication costs for the N-body code were low. Consequently it seemed feasible to set the communication component models to zero without too much adverse affect on the over all prediction, at a savings of model development time. However, the zero communication model resulted in predictions that were off by 25% (when computation was modeled using a benchmark), and off by 16% (when computation was modeled using an operation count), as shown in Figure 3.11.

It was determined that additional components needed to be modeled to reflect the communication time (see Section 3.A.4), as the model developer believed

SlaveBM(i)		
	$NumElt(S_i)$	$BM(Slave, S_i)$
PCL Cluster	$\frac{N*N}{P}$	Table 3.24
Linux Cluster	$\frac{N*N}{P}$	$1.01798 * 10^{-7}$

Table 3.21: Summary of the computation component models for benchmark models defining the N-body structural model on two dedicated platforms.

	Scatter		
	$NumElt(S_i)$	$Size(Elt)$	$BW(M, S_i)$
PCL	N	208 Bytes	Table 3.4
Linux	N	208 Bytes	6.610 MBytes/sec

Table 3.22: Summary of the **Scatter** component model used for the N-body application on two dedicated machines.

	Gather		
	$NumElt(S_i)$	$Size(Elt)$	$BW(S_i, M)$
PCL	$\frac{N}{P}$	208 Bytes	Table 3.4
Linux	$\frac{N}{P}$	208 Bytes	2.6125 MBytes/sec

Table 3.23: Summary of the **Gather** component model used for the N-body application on two dedicated machines.

both **SlaveComp** models to be accurate, a decision reinforced by the fact that they both calculated similar results for the computation only. Using the **Gather** and **Scatter** models shown in Tables 3.22 and 3.23, we achieved predictions within 9% for the benchmark model and 8% for the operation count model on the PCL cluster, as shown in Figure 3.12. This application on the PCL illustrated actions by a model developer when a profile might be in error, and additional component models are needed. The exact course of action is a judgment call. Instead of adding component models, the model developer could have decided to refine the

Machine	Time (sec)
Thing1	$1.34277 * 10^{-5}$
Thing2	$1.33701 * 10^{-5}$
Lorax	$5.23152 * 10^{-5}$
Picard	$4.46283 * 10^{-5}$

Table 3.24: Benchmarks for one processor N-body code on the PCL cluster at 750 problem size, mean over 20 runs.

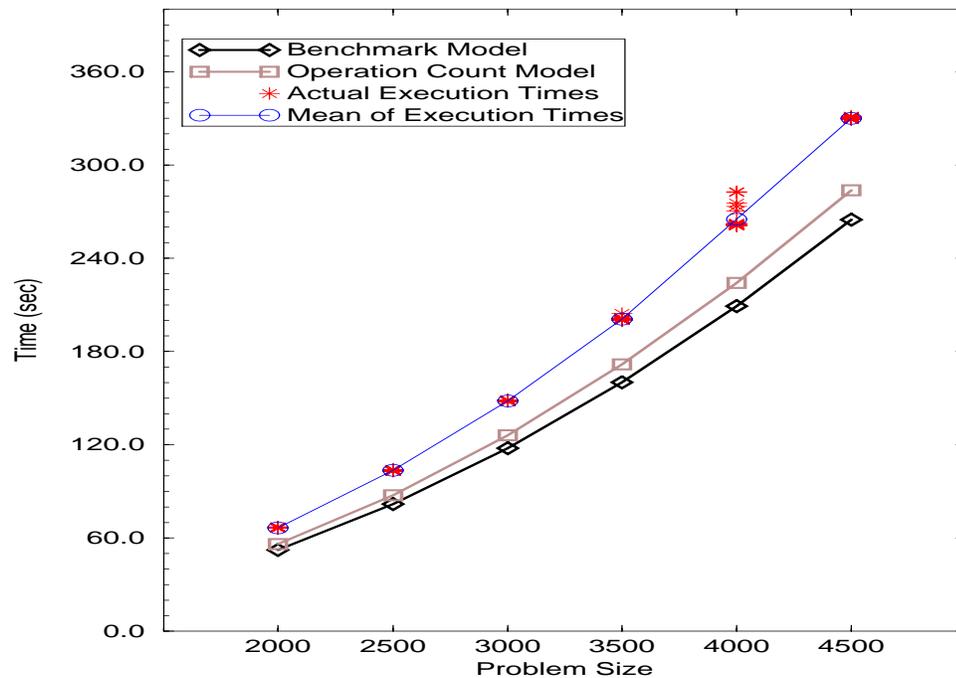


Figure 3.11: Graph showing benchmark and operation count models without communication, and actual values and means for N-Body code on PCL Cluster.

computation benchmarks, or to examine any input parameters in common for their validity.

On the Linux cluster, we had only a benchmark model available, and achieved predictions within 17% for most of the range of problem sizes, the exception being for the smallest problem size where we were off almost 100% in relative terms, and by 4.8 seconds in absolute terms. Also of interest for this application is the noticeable behavior of the benchmark approach - the error for the medial value of 4000 is practically nil, but increases in either direction, as shown in Figure 3.13.

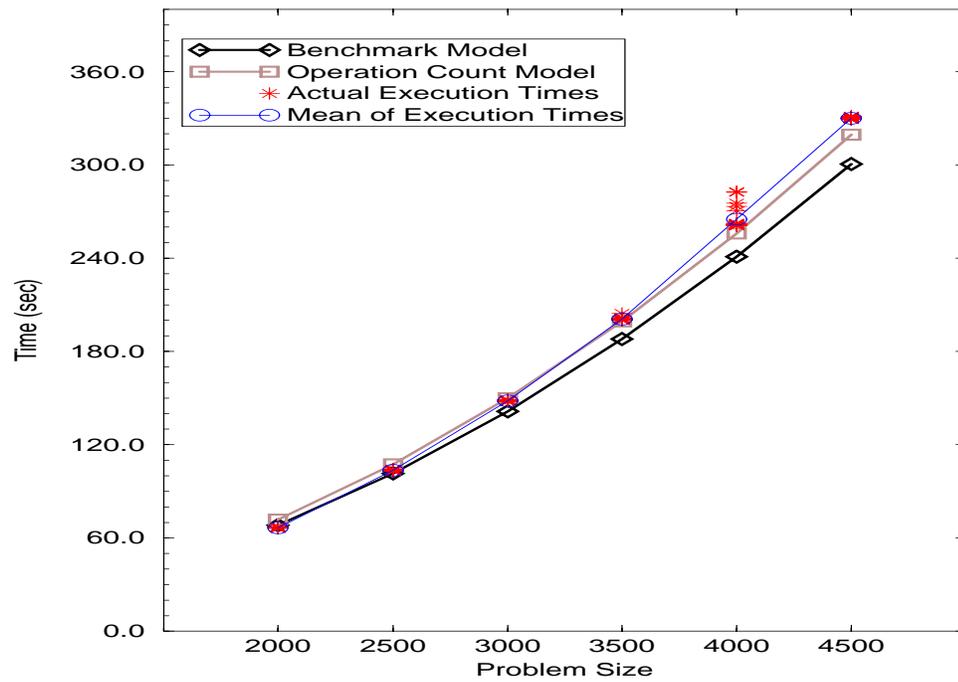


Figure 3.12: Graph showing benchmark and operation count models with communication models, with actual values and means for N-Body code on PCL Cluster.

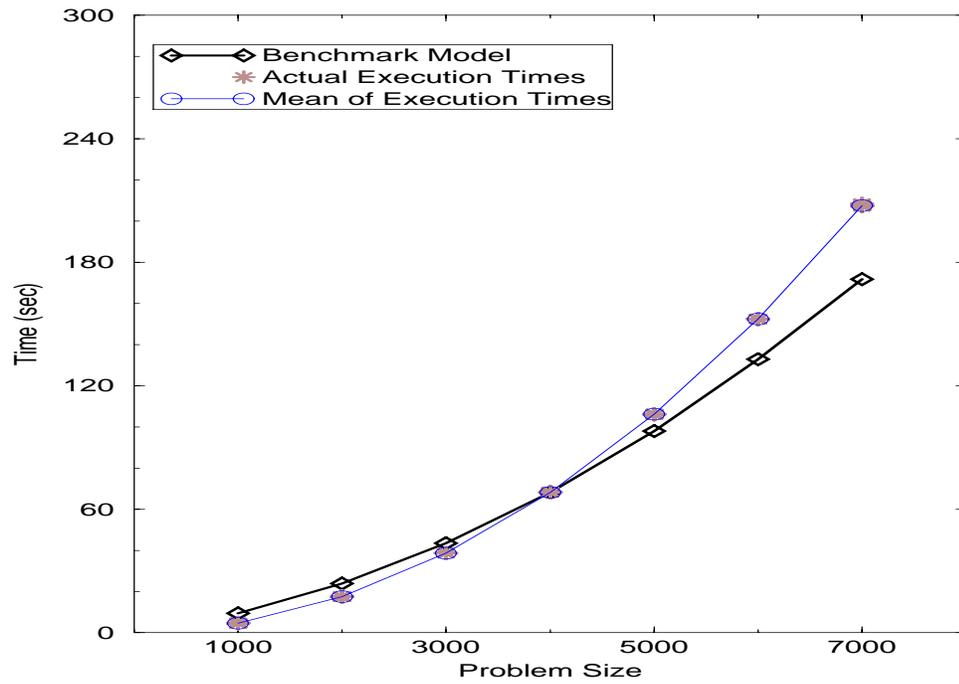


Figure 3.13: Graph of N-body application on Linux cluster, modeled versus actual execution times.

3.D.2 Modeling the Hi-Temp Code

The Hi-Temperature Superconductor code, introduced in Section 2.C, is a Master-Slave Request style code. It uses a “pool of jobs” approach: The Master is started, and it spawns a Slave task on each available host. Each Slave then requests a job from the available job pool held by the Master, does the computation needed for that job, and returns the result. This continues until the pool of jobs is empty.

The top-level model used for this code reflected its Request-style implementation. It consists of the setup time for the Master process, then the time for one slave process to receive a job, compute that job, and then return the results of that job to the Master for its entire set of jobs. This is:

$$\begin{aligned}
 \mathbf{HiTempExTime} &= \mathbf{MasterSetup}(M) \\
 &+ \mathit{Max}\left\{ \sum_{j=1}^{NumJobsOnS_i} [\mathbf{ReceiveJob}(M, S_i) + \mathbf{CompJob}(S_i) \right. \\
 &\qquad \qquad \qquad \left. + \mathbf{ReturnJob}(S_i, M)] \right\}
 \end{aligned} \tag{3.47}$$

Unlike the Master-Slave codes previously considered in this chapter, the exact number of jobs that get assigned to a processor, parameter $NumJobsOnS_i$, is not determined by the Master, rather, the jobs are requested one at a time by the slaves. In addition, this implementation dynamically requests jobs, although in a dedicated setting the difference between dynamic and static requesting isn’t as large as it would be in a production system. To predict the execution time of a Request-style application we still need to have an estimate for the number of jobs a processor is likely to request. As we’ll see for the heterogeneous PCL cluster, this value can be determined using a benchmark in a dedicated setting, but is much more difficult to estimate for production environments.

Another way to estimate the execution time of a dynamically allocated job [SW98b] is to estimate the maximum run time for a problem size of $n - 1$, and assume a worst case scenario in which the last job is started at that time on the slowest processor. A top-level model for this scenario is:

$$\begin{aligned}
\mathbf{HiTempExTime} &= \mathbf{MasterSetup}(M) \\
&+ \mathit{Max}\left\{ \sum_{j=1}^{NumJobsOnS_i} [\mathbf{ReceiveJob}(M, S_i) + \mathbf{CompJob}(S_i) \right. \\
&\quad \left. + \mathbf{ReturnJob}(S_i, M)] \right\} \\
&+ \mathit{Max}\{\mathbf{ReceiveJob}(M, SlowestSlave) \\
&\quad + \mathbf{CompJob}(SlowestSlave) + \mathbf{ReturnJob}(SlowestSlave, M)\}
\end{aligned} \tag{3.48}$$

This simplifies to the top-level model in Equation 3.47 in the implicit worst case. In many environments, such as the PCL environment detailed below, using Equation 3.48 will result in more accurate predictions.

	Linux Cluster	PCL Cluster
Master(M)	2%	5%
ReceiveJob(M, S_i)	4%	3%
CompJob(S_i)	90%	89%
ReturnJob(S_i, M)	4%	3%

Table 3.25: Profile for Hi-Temp code on the Linux and PCL Clusters.

	CompJob(S_i)
Linux Cluster	$8.649 * 10^{-3} \frac{sec}{element}$
PCL Cluster	Table 3.27

Table 3.26: Component model for **CompJob(S_i)** for Linux and PCL clusters.

The application profile for the HiTemp code on the Linux cluster and PCL cluster is given in Table 3.25, and the predictions for component model used are summarized in Table 3.26. The other element in the top-level model that must be determined is a value for the input parameter $NumJobsOnS_i$.

In order to predict the execution time of this application we first need to predict how many jobs each processor would calculate, and then determine which processor would be calculating the last job. For the dedicated Linux cluster, the

Machine	Time
Thing1	0.89105 $\frac{sec}{element}$
Thing2	0.87058 $\frac{sec}{element}$
Lorax	3.6618 $\frac{sec}{element}$
Picard	3.4382 $\frac{sec}{element}$

Table 3.27: Benchmarks for Hi-Temp code on PCL cluster, mean of 20 runs for problem size 75.

work is divided evenly over the 4 slaves, so this value is simply $\frac{N}{P}$. For the PCL cluster, this is a harder value to determine. From the benchmark given in Table 3.27, it can be shown that the last running job would be computed by either Lorax or Picard, since they would request their last job before either Thing1 or Thing2 would. However, calculating this depended on having very exact information. In Table 3.28 we chart the first few seconds of execution time on the PCL cluster for the HiTemp application, and in Table 3.29 we chart the last few seconds, from $t = 68$ seconds to completion time for a problem size of 200. Both these tables are based on the benchmarks given in Table 3.27 and assume that they are 100% accurate. From Table 3.27, we can calculate that Picard and Lorax would compute approximately 10% of the overall jobs each, while each of Thing1 and Thing2 would compute approximately 40% of the jobs.

Using this model and benchmark, we achieved predictions within 4% on the Linux cluster and 3% on the PCL cluster, as shown in Figures 3.14 and 3.15. One of the interesting features of this application was the determination of the input parameters $NumJobsOnS_i$. In a production environment, or one in which the benchmark information was not so carefully calculated as ours, we might want to use the more conservative top-level model given in Equation 3.48, and estimate the runtime for $n - 1$ jobs, then add on the time for the n^{th} job to be run on the slowest processor.

Time	Thing1	Thing2	Lorax	Picard
0.0	1	2	3	4
0.871		5		
0.891	6			
1.74		7		
1.78	8			
2.61		9		
2.67	10			
3.44				11
3.48		12		
3.56	13			
3.66			14	
4.35		15		
4.46	16			

Table 3.28: Chart showing first seconds of execution of HiTemp application on PCL cluster, using benchmarks given in Table 3.27. Numbers represent job number started on processor i at time t .

Time	Thing1	Thing2	Lorax	Picard
68	194	195	190	189
68.61	196			
68.76				197
68.77		198		
69.57			199	
69.50	200			
69.64		done		
70.39	done			
72.20				done
73.23			done	

Table 3.29: Chart showing first seconds of execution of HiTemp application on PCL cluster, using benchmarks given in Table 3.27. Numbers represent start time for job on processor i at time t , and “done” indicates when a processor is finished computing.

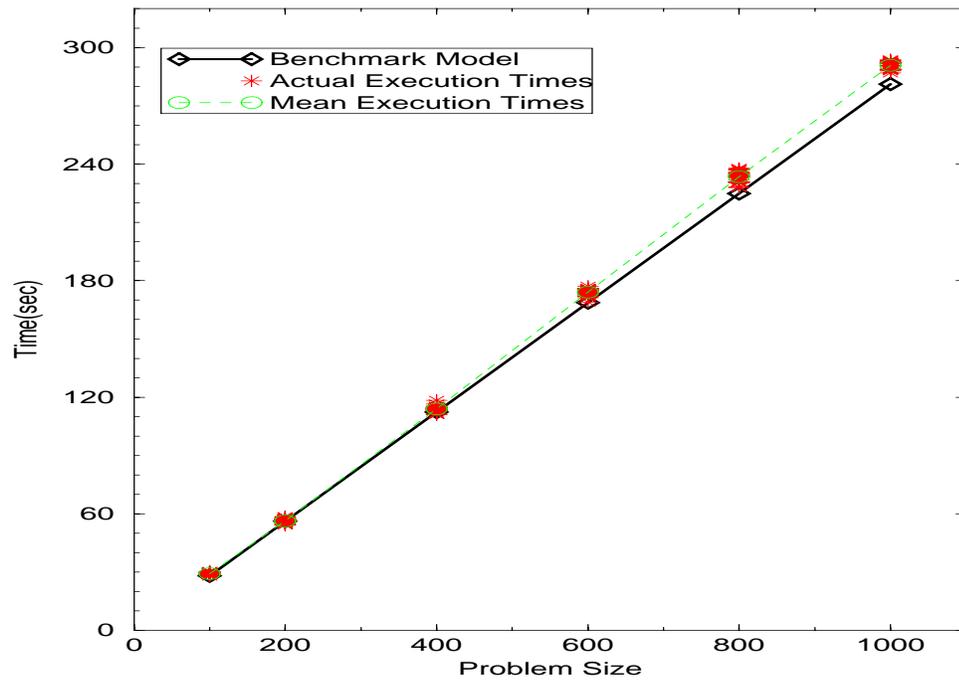


Figure 3.14: Graph of actual versus modeled execution times for Hi-Temp code on Linux cluster.

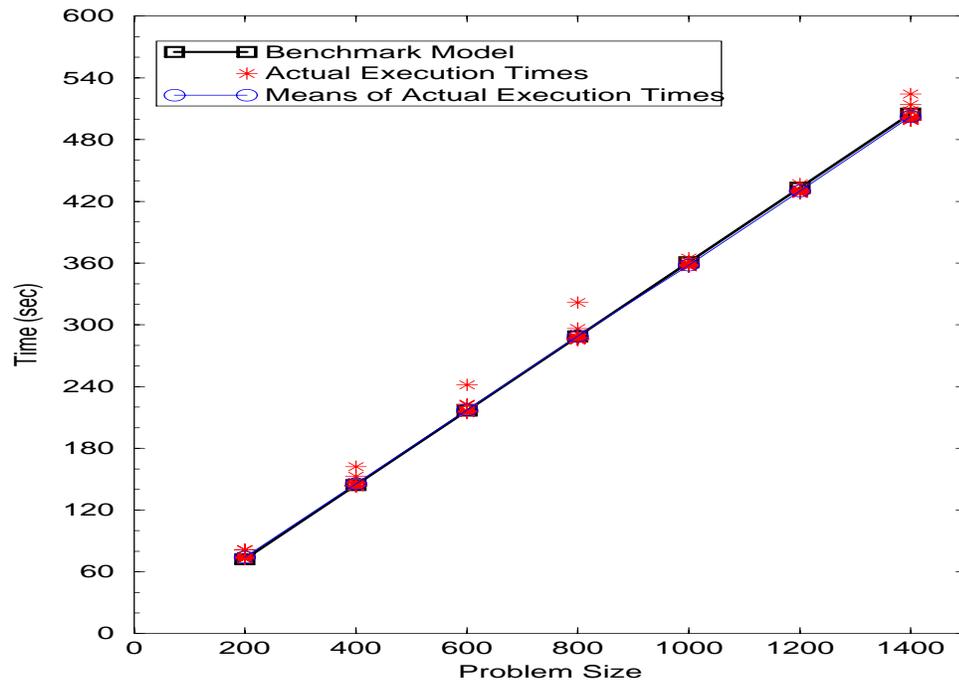


Figure 3.15: Graph showing results of benchmark model and actual execution times for Hi-Temperature code on PCL cluster.

3.D.3 Modeling the LU Benchmark

In addition to the Master-Slave applications, we examined several Regular SPMD benchmarks, one of which was an LU decomposition. The LU algorithm finds a lower triangular and an upper triangular matrix such that $L \cdot U = A$ for an original matrix A . It works on diagonals that progressively sweep from one corner on a given z plane to the opposite corner of the same z plane, thereupon proceeding to the next z plane. There are a relatively large number of small communications of 5 words each.

This code is structured so that the natural division isn't at the iteration, it is an entire application run. This code has an easy-to-calculate number of messages per entire application run, and a computation size that depends on the size of the area under work, that changes for every iteration. Therefore, the application developer decided to examine the execution time as just communication and computation [And97]. The top-level model used is given in Equation 3.49. The application profile for the Linux and PCL cluster are shown in Table 3.30. The component models used are given in Tables 3.31 and 3.32.

$$\mathbf{LUExTime} = \mathbf{Comp} + \mathbf{Comm} \quad (3.49)$$

	Linux Cluster	PCL Cluster
Comp	65%	80%
Comm	35%	20%

Table 3.30: Application Profile for LU benchmark on Linux cluster

This is a fairly large code, consisting of 33 Fortran files and over 6,000 lines of code. Because of this, it was non-trivial to calculate operation counts for the computation as a separate entity. Benchmarking the code presented a more efficient approach. We examined three ways to benchmark the code:

1. Run a one-processor version of the code. This resulted in a large error for this

Comm	
NumMsgs	Max [Lat(x)]
$(P[(n-1)+(n-2)]+10)$	Max [Lat(x)]
$8n-2$	0.2 sec

Table 3.31: Communication component models for LU benchmark on Linux cluster

Comp	
Linux Cluster	$\frac{BM(P,N)-Comm}{N^3}$
	$2.5755 * 10^{-3} \frac{sec}{elt}$
PCL Cluster	$\frac{BM(P,N)}{N^3}$
	$3.7522 * 10^{-3} \frac{sec}{elt}$

Table 3.32: Computation component models for LU benchmark on Linux cluster and PCL cluster

code, possibly due to scaling issues associated with the code. The problem size for the application scales as N^3 . To approximate a 4 processor version of the $n = 30$ problem size on one processor required running a problem size of 19 on one processor. In general, this did not scale well for the application.

2. Comment out the communication routines, as we did in the SOR code (Equation 3.43), and benchmark the resulting computation routines. This was not an option because of the way the code was structured.
3. Benchmark the code as a whole, and subtract the communication time to achieve a computation-only estimate. We estimated the communication by modeling it as shown in Table 3.31, and subtracted this from the overall time of a medial problem size (30) to achieve a computation benchmark for the Linux cluster.

It is also possible to model the application as one large benchmark (with computation and communication together), and scale that by the problem size. In the graph given in Figure 3.16 we compare this approach with benchmarking

approach 3 above, and can see that this one-part model is both greater than or less than the medial our approximations are increasingly in error, and growing on both ends away from the medial problem size. This is due to the fact that we scale the benchmark by the problem size cubed, but the communication part of the execution time scales by n , which is not reflected in the one-part model.

On the PCL cluster, due to the larger computation/communication ratio exhibited by the platform, we could use the simple one part model based solely on the benchmark for the entire application at a medial problem size. This is similar to the use of communication components in previous applications in which we could set the communication component models to zero. Using this approach resulted in a model on the Linux cluster that estimated the actual execution time of various problems sizes within 15%, as shown in Figure 3.16, and on the PCL cluster within 10% for the problem sizes we examined, as shown in Graph 3.17. This application shows that sometimes the simplest structure is just that of a single task, and lower level considerations can sometimes be ignored, as their costs are amortized for problem sizes of interest.

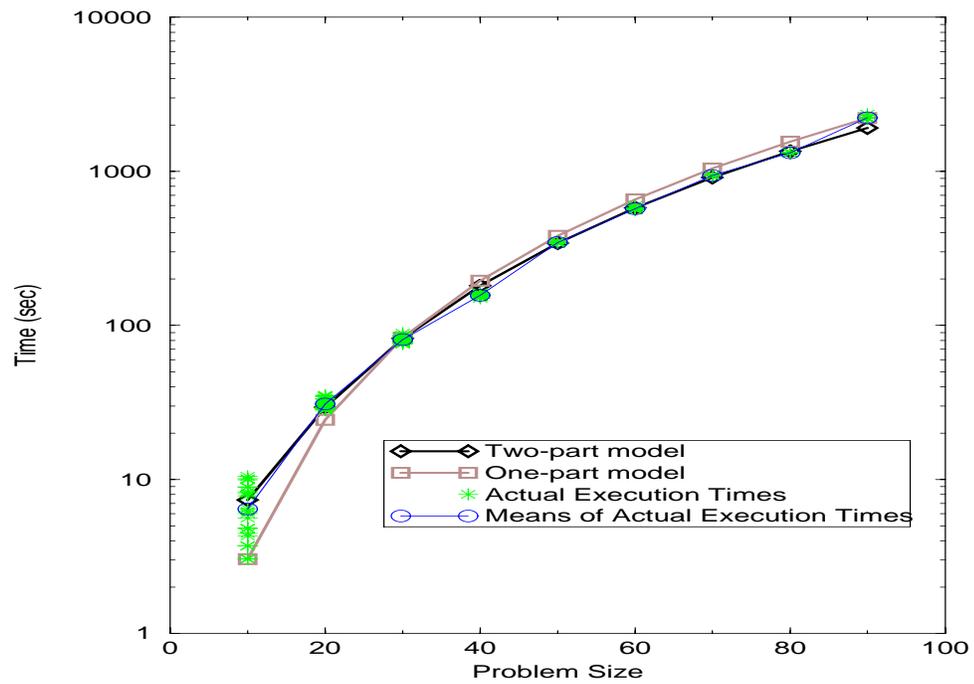


Figure 3.16: Modeled versus actual execution times for LU benchmark on Linux Cluster. Note this is a log-linear graph.

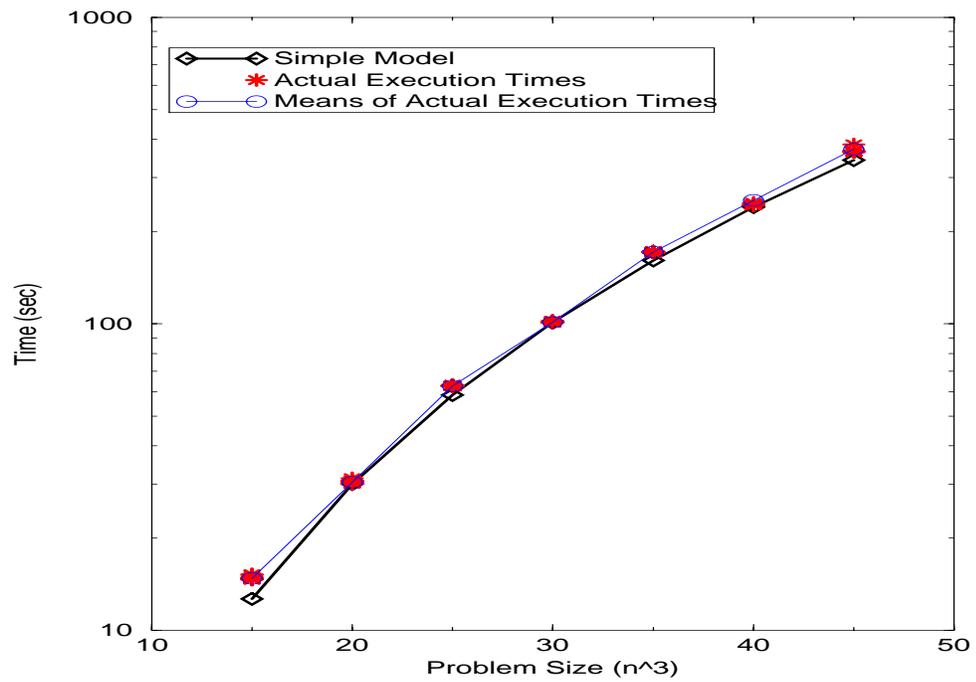


Figure 3.17: Graph of an simple model for LU on PCL cluster and actual times. Note that this graph is on a log-linear scale.

3.D.4 Modeling the SP Benchmark

SP is another Regular SPMD benchmark from the NPB suite. It solves three sets of un-coupled equations in three dimensions using a scalar penta-diagonal algorithm, first in the x direction, then in the y , and finally in the z direction.

To model SP, we had very little information about the internals of the code, which was approximately 5,000 lines of code spread over 30 files. We had no meta-data about possible input parameters, but we did have the information that the application profile (given in Table 3.33) was extremely accurate for a wide range of problem sizes [And97]. Because of our confidence in the application profile, we used only a benchmark for computation and this percentage to model the performance of the application. The models we used were are given in Table 3.34.

Computation	75%
Communication	25%

Table 3.33: Application Profile for SP on the Alpha Farm

SPEXTime(i)		
Comp		Comm
NumElt(i)	BM(SP, i)	$\frac{25}{75}$ Comp
$\frac{N^3}{P}$	$0.6947 * 10^{-3}$	

Table 3.34: Model for the SP benchmark on the Alpha farm.

This resulted in predictions within 7.5%, as shown in Figure 3.18. This application demonstrated yet another way to adapt structural models based on the level of information available.

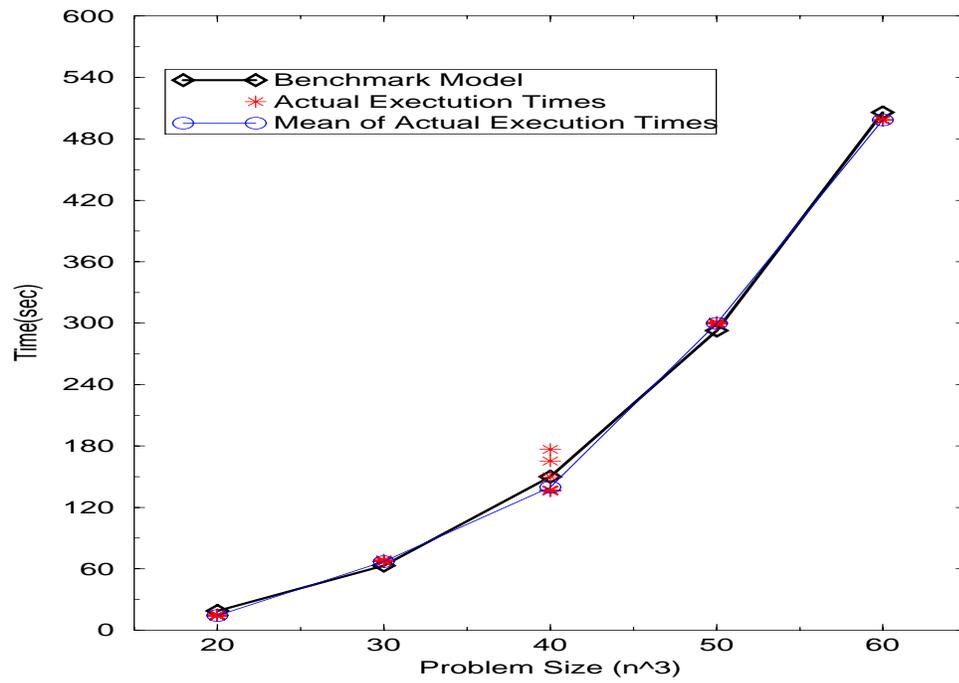


Figure 3.18: Actual times versus modeled times for SP on Alpha Farm.

3.D.5 Modeling the EP Benchmark

The EP (Embarrassingly Parallel) benchmark is also part of the NAS benchmark suite. This code generates random numbers based on generating Gaussian pairs.

Since this code is embarrassingly parallel, with each processor running it's own task in isolation, the execution time is merely the maximum computation time for each processor:

$$\mathbf{EPExTime} = \mathit{Max}[\mathbf{Computation}(i)] \quad (3.50)$$

We used the following component models in Tables 3.35 and 3.36.

	BMComp(i)	
	NumElt(i)	BM(EP, i)
Alpha Farm	$\frac{2^N}{P}$	$3.30658 * 10^{-7}$
Linux Cluster	$\frac{2^N}{P}$	$3.1296 * 10^{-6}$

Table 3.35: Summary of the computation component models for benchmark models used in defining the EP structural model.

	OpCtComp(i)		
	NumElt(i)	Op(EP, i)	CPU(i)
Alpha Farm	$\frac{2^N}{P}$	29 real mults	$2.28024 * 10^{-7}$

Table 3.36: Summary of the computation component models for operation count models used in defining the EP structural model.

This resulted in two models that predicted the execution times within 8% of the mean values for the EP on the Alpha Farm, as shown in Figure 3.19, and within 3% on the Linux cluster, as shown in Figure 3.20.

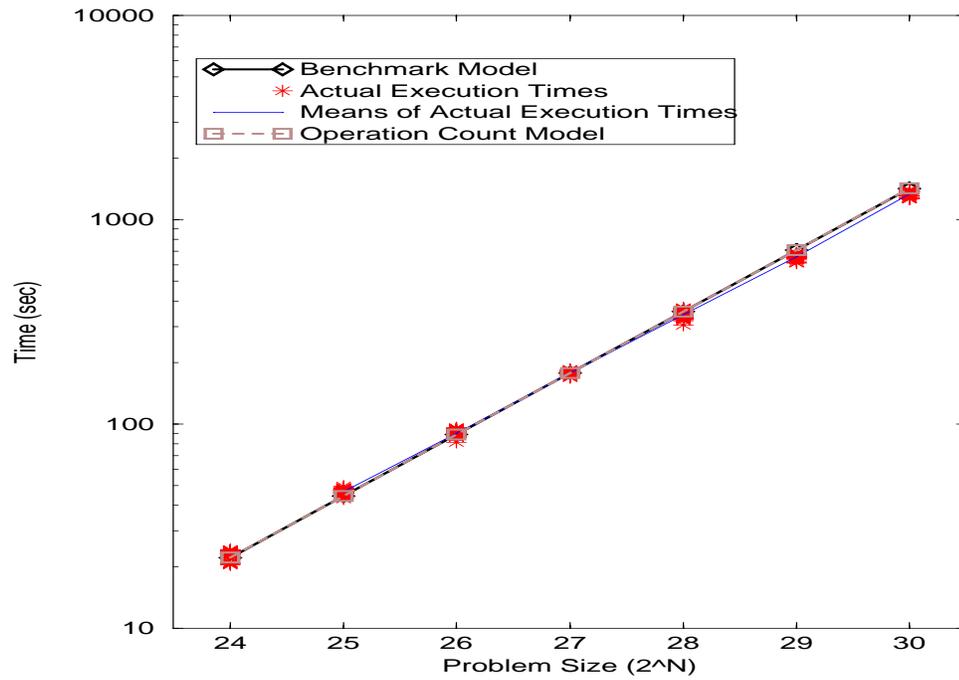


Figure 3.19: Actual times versus modeled times for EP on Alpha Farm. Note log-linear graph.

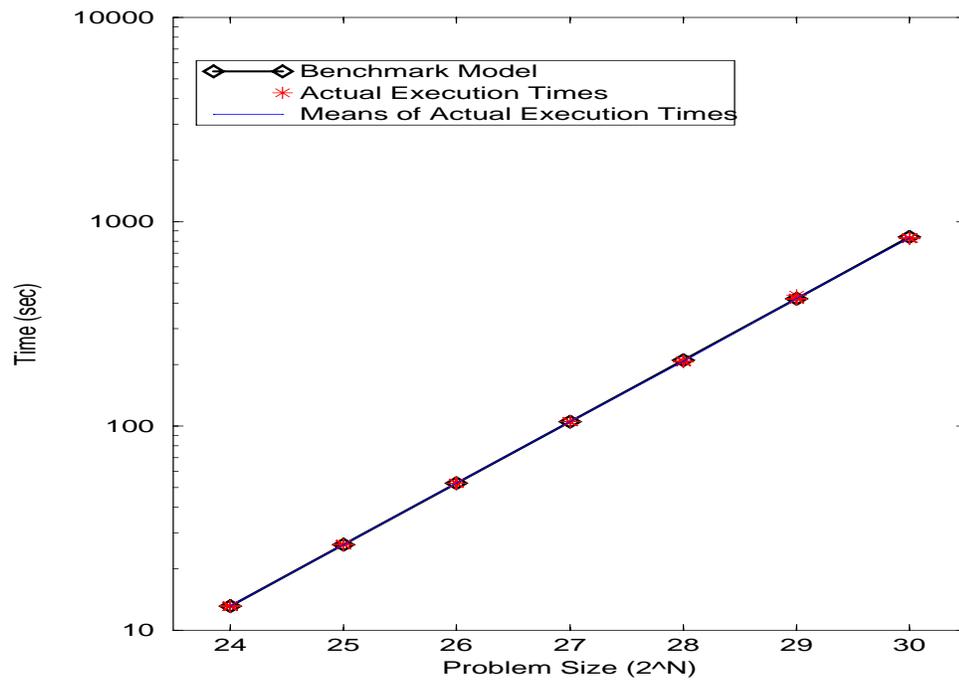


Figure 3.20: Actual times versus modeled times for EP on Linux cluster. Note log-linear graph.

3.D.6 Discussion

In this section we presented five applications belonging to the Master-Slave and Regular SPMD classes. Structural models were constructed for each to validate the approach and demonstrate the flexibility of each.

The N-body code demonstrated a course of action to take when the application profile might be in error, in this case, by adding additional communication component models. The Hi-temp code demonstrated how to model an application with dynamic load balancing on both homogeneous and heterogeneous platforms. The LU benchmark demonstrated additional benchmarking techniques possible for a platform. The SP showed how the profile could be used as part of the model, and the EP showed a simple approach using only computation. All the applications were modeled to a high degree of accuracy, and the flexibility of adapting structural models for applications and platforms was illustrated.

3.E Related Work

Many researchers have focused on the problem of predicting performance in various environments. This section surveys related work to our approach.

3.E.1 Models for Parallel Applications

One common approach to modeling parallel applications is to separate the performance into application characteristics and system characteristics [TB86, Moh84, KME89, ML90]. This approach may also be adequate when the underlying resources are very similar. For example, Zhang [YZS96] considers only workstations that do floating point arithmetic identically. Similarly, the WARP project [SW96a, SW96b] uses an architectural model based on a task graph weighted by relative execution times of a generic benchmark. Neither of these approaches allow for the fact that different applications will have better performance on different machines,

and that a single relative ranking is not sufficient when dealing with heterogeneous resources.

In the cluster environment the line between application and resource is not easily drawn. We do not separate application and system characteristics as part of the top-level model, or in structural modeling in general, because developers address their codes as implementations—or combinations of both application and system. For example, a task will be tuned to fit an architecture’s specific cache size or to use an architecture-specific library routine.

Another closely related approach in parallel computing is Lost Cycles Analysis [Cro94]. Lost cycles analysis involves measurement and modeling of all sources of overhead in a parallel program: synchronization, load imbalance, communication, insufficient parallelism, etc. Crovella et. al have built a tool for measuring overheads in parallel programs, then fit those measurements to analytic forms. These tools enable programmers to develop accurate performance models of parallel applications without requiring extensive performance modeling expertise, but may require a great deal of low-level information about the platform of concern.

3.E.2 Compositional Approaches

The structural modeling approach is similar in spirit to the use of *skeletons* for parallel programming [Col89, DFH⁺93]. With skeletons, useful patterns of parallel computation and interactions are packaged together as a construct, and then parameterized with other pieces of code. Such constructs are skeletons in that they have structure, but lack detail, much as the top-most structural model shows the structure of the application with respect to its constituent task implementations, but the details of the tasks themselves are supplied by individual component models. Most research on skeletons has concentrated on the use of functional languages to implement skeletons as opposed to using them to predict application execution performance.

Fersha [FJ98] extended the skeleton approach to use petri nets for performance prediction of developing programs from a software engineering or performance engineering point of view. However, to use this approach, all tasks, communication packets, and processes (virtual processors) and their ordering must be defined. This approach is a step forward, but by tying it to Petri nets, the computational overhead may be excessive for larger, complex applications.

A similar approach is called *templates*, defined in [BBC⁺94]. Templates are a description of a general algorithm rather than the executable object code or the source code more commonly found in a conventional software library. Nevertheless, although templates are general descriptions of key algorithms, they offer whatever degree of customization the user may desire. For example, they can be configured for the specific data structure of a problem or for the specific computing system on which the problem is to run. The template work focuses on the use of iterative methods for solving large sparse systems of linear equations, where an iterative method is an algorithm that produces a sequence of approximations to the solution of a linear system of equations. Templates for other classes of applications have not yet been addressed. They are not currently being used for performance prediction in a practical setting.

Similarly, Chandy [Cha94] addresses *programming archetypes*, abstractions from a class of programs with a common structure. Archetypes include class-specific design strategies and a collection of example program designs and implementations, optimized for a collection of target machines. This work concentrates on deriving program specifications for reasoning about correctness and performance, not developing models for the same.

3.E.3 Component Models

In some sense, all modeling work grounded in practical methods can be considered related work to the component model approach. We hope to leverage existing modeling approaches to build component models, and to make selections

between them.

One of the problems we have encountered is that many approaches to modeling assume the availability of pieces of data that we do not have for our system. Thus, we have grounded this work in models used for current applications [CS95, DS95, DTMH96, MMFM93, PRW94, WK93, YB95]. These models all take into account complexity of computation, information availability, and accuracy needed for predictions in a parallel distributed environment.

POEMS [DDH⁺98] is an end-to-end modeling environment consisting of compiler driven task-graph generation and multiple modeling and simulation approaches to verify and extend applications. This is an ambitious project that fits nicely with the structural modeling approach philosophy. When the components are assembled, the POEM-version of top-level models and component models will be automatically generated by a variation of the Fortran-D compiler. This data will be fed to a variety of modeling approaches, both theoretical and analytical, where different parts of the application and system can be modeled at differing levels of detail using an object approach. At this point, the models can either be symbolically solved or simulated to make estimates of execution time. The current focus of the POEMS project is the development of theoretical modeling approaches and simulation. Task graphs are currently constructed by hand, and models are selected by hand as well. The POEMS approach will eventually allow different components of the application, as well as the operating system and the hardware to be modeled at varying levels of detail.

Examples of instrumentation are seen in the methods presented by the CLIP and CLASP work [AHWC98]. CLIP (Common Lisp Instrumentation Package) aids the researcher in instrumenting Lisp code to define and run experiments and collect data about runtime conditions. This data can be analyzed by a number of statistical packages, but is meant to be used by CLASP (Common Lisp Analytical Statistics Package) to analyze the data using graphics, statistical tests and various kinds of data manipulation. CLIP is only compatible with Common Lisp

codes, thereby limiting its usefulness in the scientific community, but CLASP can work with most data files, although it was targeted to AI simulations in development.

3.E.4 Related Petri Net Work

PAPS [WH94] is a parallel program prediction toolset based on a set of Petri net-driven performance prediction tools. Parallel systems are described by acyclic task graphs representing typical workload of parallel programs, by processor graphs describing the hardware, and a mapping function. These three specifications are independent and changeable. The performance model generator reads in these three pieces of information and automatically constructs a Petri net performance model for it. This work uses a relative measure of performance to compare applications on different parallel machines with different mapping functions.

Unfortunately, in parallel distributed computing it would be very time consuming to evaluate the changing resource sets and data mappings using this approach. In addition, large programs often cannot be specified to a useful level of detail, in part because the resulting petri net models would be too large to fit into memory (problems of this nature occur graphs of more than 10,000 nodes, a size well within reason for even simple benchmarks). In addition, the specification can be tedious, and is error prone for large systems.

3.E.5 Application Profile

Various application profiling strategies have been addressed in the literature. A detailed application profile is of common use in compiler optimizations [Wal90, CGS95]. For program comprehension and tuning, systems tools such as `gprof` [GKM82] and `pixie` [Sil], can supply application profile tables of data to help determine the percentage of time spent in a function, how many times a basic block was run, how often a global variable was accessed, and for some applications

accesses and how many misses occur at each level of memory.

Many scheduling approaches use what can be thought of as an application profile for the performance prediction portion of their scheduling methodology. They require information about the frequency and average execution time for each I/O operation, memory read or write, and floating point operation throughout the application. Obtaining this level of information is difficult and often time-consuming if even possible. Zhang et al. [YZS96] have developed a tool to measure the basic timing results needed for their approach, but they avoid the problem of having the profile change with respect to problem size by analyzing a single problem size over a range of possible workstation networks. Others, like Simon and Wierum [SW96a], require complex microbenchmarks to be run on each system.

Ken Sevcik [Sev89] uses a parallelism profile to build schedules for space-shared machines. His profile is determined experimentally by running the application in the presence of an ample supply of processors. It indicates, as a function of time, the number of processors in use throughout the execution of the application.

3.E.6 Model Selection

Related work to component model selection approach can be found in the area of algorithm and platform classifications. Jamieson [Jam87] examines the relationship between architecture and algorithm characteristics for signal processing applications. This work attempts to identify the influence of specific application characteristics on architecture characteristics. Saavedra-Barrera [SBSM89] defined micro-benchmarks to characterize machine performance, and defined a visualization technique called a *pershape* to provide a quantitative way of measuring the performance similarities of multiple machines. Both of these efforts provide some quantification of application/architecture affinity.

Many modeling approaches assume that considerable, low-level information is available. For example, Adve [Adv93] assumes that all execution times of sub-task components are known, that they are deterministic in nature, and

that all communication, contention and other overheads are negligible. Likewise, [CQ93, YZS96] assume that operation counts for various procedures or application components are available for modeling. These approaches cannot extend to heterogeneous platforms as operation counts and their meanings will change from platform to platform. It is well documented [Cro94] that while one machine or suite of machines will perform best on a given application, another set will perform best on a different application. We have an additional problem in that often different implementations are targeted to specific architectures, and these may not even be the same algorithm for a problem, let alone have the same operation count.

3.F Summary

In this chapter we have presented a technique to model applications targeted to clusters of workstations called **structural modeling**. Application execution behavior is decomposed in accordance with the functional structure of the application, and individual tasks are modeled with flexible, extensible component models. Results were presented for several Master-Slave and Regular SPMD applications on three different single-user platforms to show not only the accuracy of this approach, but the ease in adaption to new platforms. The next chapter shows how this work can be extended to predict the behavior of systems shared by multiple users.

Some of the material in this chapter has been previously published [Sch97]. The dissertation author was the primary investigator and single author of this paper.

Chapter 4

Stochastic Values and Predictions

In the last chapter we showed that it was possible to model a diverse set of applications on a variety of single-user (dedicated) cluster platforms using the structural modeling approach. One of the main differences between a single-user platform and a multi-user (production) platform is that system and application characteristics, especially available CPU and bandwidth, may exhibit variable behavior during the execution of an application. Program execution can also vary based on data set characteristics or non-determinism as well.

This chapter describes how to extend the structural modeling approach to incorporate information about varying system and application behavior. We define stochastic values, or values that represent distributional data, to reflect fluctuations in behavior. We describe three possible representations for stochastic values and the associated arithmetic for each, discuss the advantages and disadvantages of each approach, and show experiments that support this approach. We begin this chapter by assuming that accurate structural models are available for the application on the system in question. Section 4.H addresses the case in which a structural model does not accurately capture the behavior of the application.

4.A Motivation

Most performance prediction models use parameters to describe system and application characteristics such as bandwidth, available CPU, message size, operation counts, etc. Model parameters are generally assumed to have a single likely value, which we refer to as a **point value**. For example, a point value for bandwidth might be 7 Mbits/second.

However, in practice, point values are often a best guess, an estimate under ideal circumstances, or a value that is accurate only for specific time and load conditions. Frequently, it is more useful to represent system and application characteristics as a **distribution** of possible values over a range; for example, bandwidth might be reported as having a mean of 7 Mbits/second and a standard deviation of 1 Mbit/second. We refer to values that can be represented by distributional data as **stochastic values**. Whereas a point value gives a single value for a quantity, a stochastic value gives a set of possible values weighted by probabilities to represent a range of likely behavior [TK93]. More precisely, we define a stochastic value X to be the set of values $\{x_1, \dots, x_n\}$ with the associated function $F(x_i) = p_i$, where p_i is the probability associated with x_i , for all values $x_i \in X$. Note that $\sum_i p_i = 1$.

We call a prediction that can be represented as a distribution of execution times a **stochastic prediction**. Such predictions are calculated by using stochastic values to parameterize structural models. This is shown graphically in Figure 4.1.

There are two problems that must be addressed to make it feasible to use stochastic values for performance predictions. The first problem is to determine a representation for the distributional data. In this chapter we present three ways to represent stochastic values: using *normal distributions*, using an *interval* representation, and using *histograms*. Each of these methodologies has advantages and disadvantages that will be discussed.

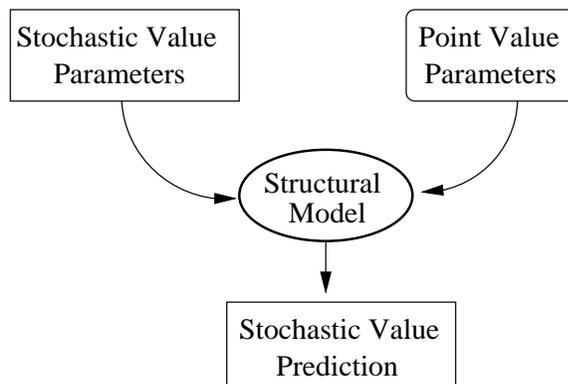


Figure 4.1: Stochastic predictions are created by parameterizing structural models with stochastic values.

In addition, to make use of stochastic information in a structural model, we need to define all arithmetical and collective operation functions that are used as composition operations. In defining composition operators for structural models in the last chapter, the need for functionality to allow for additional information was addressed briefly. If the parameters to a component models are single (point) valued execution times, then, for example, the composition operator $+$ will be basic addition, that is $f(x, y) = x + y$. However, if the parameters for a component model are stochastic values, the composition operator for $+$ would need to be defined over stochastic values as well as result in stochastic values. In order to use stochastic values to parameterize structural models, we must define not only the representation of a stochastic value and how to derive it from the available data, but the needed arithmetic as well.

Stochastic values are only one form of additional information that can be incorporated into structural models. Other forms of data that could be incorporated are so-called quality of information (“QoIn”) measures [BW98, BWS97]. QoIn measures are quantifiable attributes which characterize the quality of a value. Such attributes might include the lifetime of the prediction, its accuracy, the overhead of computing the prediction, etc. In order to generate a quality of information

metric for the overall prediction, the relevant QoIn measures would need to be combined as part of the function of the composition operators, similar to the arithmetic defined for stochastic values in this chapter.

4.A.1 Statistics and Notation

Throughout this chapter some basic statistical terms and notation are used. We review them in this section.

The **mean** of a set of data is the arithmetical average over all values in the set. Given a set of n values, x_1, \dots, x_n , the mean is defined to be:

$$\frac{\sum_{i=1}^n x_i}{n} \quad (4.1)$$

The mean of a set of data samples is often notated \bar{x} but we denote it m to differentiate it from an upper bound (defined below).

The primary measure of variability in statistics depends on the extent to which each sample observation (value) deviates from the mean of the samples. Subtracting the mean from each value gives us the **set of deviations from the mean** m . The n deviations from the mean are the differences

$$x_1 - m, \quad x_2 - m, \quad \dots, \quad x_n - m \quad (4.2)$$

The larger the magnitude of the deviations, the greater the amount of variability in the sample.

Since deviations can be both positive and negative, and their sum is zero, simply adding them will not give a measure of the variance over a set of data. To report a summary statistic for the deviation over the data set we use the **sample variance**, sd^2 , equal to the sum of the squared deviations from the mean, divided by the size of the data set minus one:

$$sd^2 = \frac{\sum_{i=1}^n (x_i - m)^2}{n - 1} \quad (4.3)$$

Another useful metric is the **sample standard deviation** defined as the square root of the variance, denoted sd

$$sd = \sqrt{\frac{\sum_{i=1}^n (x_i - m)^2}{n - 1}} \quad (4.4)$$

We denote the mean and standard deviation of a data set as the tuple (m, sd) .

Two other important values for a set of data are the upper bound and the lower bound. The **upper bound** of x_1, \dots, x_n is the value $\bar{x} \geq x_i$ for all i . Likewise, the **lower bound** of a set is the value $\underline{x} \leq x_i$ for all i . We denote the upper bound and lower bound of a data set as the tuple $[\underline{x}, \bar{x}]$.

4.A.2 Outline

This chapter is organized as follows: In Section 4.B we address representing stochastic values as normal distributions, and describe how to combine such values arithmetically. In Sections 4.C and 4.D, we detail two alternative representations of stochastic values, intervals and histograms. We discuss the trade-offs associated with using each of these representations, including the expressiveness of the representation technique, availability of the data to define the representation, complexity of the arithmetic needed to combine stochastic values of a given representation, the error such combinations may introduce, and practicality issues. In addition, the efficiency of computing a prediction must be compared to the quality of the result, a continuation of the accuracy-overhead trade-off discussed last chapter.

In Section 4.F, we demonstrate how stochastic values can be used to improve performance predictions using several applications in various multi-user environments. Related work is presented in 4.G. We conclude with a summary and a list of possible extensions in Section 4.H. In the next chapter we show how to use these predictions in stochastic scheduling policies.

4.B Defining Stochastic Values using Normal Distributions

The first representation we considered for stochastic values was a general distribution. General distributions are difficult to use because they have no unifying characteristics. It is often appropriate to summarize or approximate a general distribution by associating it with a member of a known family of distributions that is similar to the given data.

One common family used to approximate distributions when appropriate is the family of **normal distributions**. The family of normal distributions is symmetric and bell-shaped, and approximates the distribution of many random variables (as the proportion of outcomes of a large number of independent repetitions of an experiment in which the probabilities remain constant from trial to trial). Normal distributions are defined using two values: a **mean**, which gives the “center” of the range of the distribution, and a **standard deviation**, which describes the variability in the distribution and gives a range around the mean. For normal distributions, a range equal to two standard deviations includes approximately 95% of the possible values of the data.

Many real phenomena in computer systems generate distributions that are close¹ to normal [Adv93, AV93, Das97]. For example, some benchmarks on dedicated systems may exhibit execution time values with normal distributions. Figure 4.2 shows a histogram of runtimes for an SOR benchmark on a single workstation with no other users present, and the normal distribution based on the data mean, m , and standard deviation, sd , as defined in Section 4.A.1. Distributions are represented graphically in two common ways: the **probability density function** (pdf), as shown on the left in Figure 4.2, which graphs values against their probabilities, similar to a histogram, and the **cumulative distribution function** (cdf), as shown on the right in Figure 4.2, which illustrates the probability that a

¹See Section 4.B.4 for a discussion on ways to determine “close”.

point in the range is less than or equal to a particular value.

To calculate predictions using a structural model, parameters must be combined arithmetically. To do this in the timely manner needed for on-the-fly scheduling, the arithmetic must be tractable. Normal distributions have well defined arithmetic properties and are closed under linear combinations [LM86]. Therefore, summarizing stochastic information using normal distributions enables the needed arithmetic combinations.

We will assume that when representing stochastic values by distributions that the distribution is normal, and based on the data mean and standard deviation. In subsection 4.B.2 we discuss the needed arithmetic to use normal distributions in structural models, and in Section 4.B.4 we discuss alternatives to consider when the assumption of a normal distribution does not hold true.

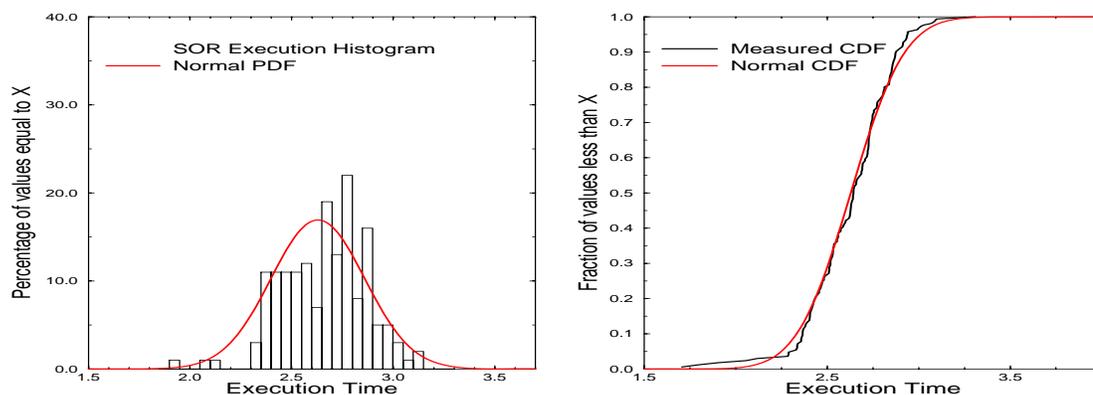


Figure 4.2: Graphs showing the pdf and cdf of SOR benchmark with normal distribution based on data mean and standard deviation.

4.B.1 Time Frame

Often, stochastic values for a system characteristic are determined by examining a time series of previous values for that characteristic. The values in the time series can be used to make predictions about the future behavior of the characteristic. There are a variety of possible approaches to generate a prediction

based on a time series [Wol97, Wol96]. The period over which the time series is taken (T) may influence the set of values used to make a prediction.

Determining a good value for T is important. If too many values are included in T , then current changes in the time series will not be adequately represented in the final prediction. However, if the time frame is too small then recently occurring out-lier values may influence the final prediction overly much. As there are many ways to predict the future behavior of a value given a time series, there are many ways to select the range of values of interest [Wol97, WSH98].

In this work, we determined the value of T experimentally. In particular, in defining bandwidth and CPU distributions, we used an auto-correlation technique for the Aggregate Mean and Aggregate Standard Deviation (defined below), which indicated that a window of $T=5$ minutes gave a high correlation (above 0.7 using a t-test) [Wol98]. That is, for our systems it was very likely that the behavior seen in the previous 5 minutes would be close to the behavior for the next five minutes. Note that this T is specific to, and was verified for, each experimental environment. The fact that the same value of T held for all three cases under a variety of workloads is promising for a general definition. This is further supported by a study run to determine the self-similarity of CPU experiments [WSH98] which were run at the same time as the experiments presented in this thesis. In addition, Dinda and O'Hallaron [Din98, DO98] found CPU behavior to be stable for an epoch of length 150-450 seconds, further supporting our choice of 300 seconds. In addition, users can easily evaluate this value for their own platforms as well. Unless stated explicitly otherwise, we use $T=5$ minutes in the following.

4.B.2 Arithmetic Operations over Normal Distributions

In order to extend structural prediction models for use with stochastic values, we need to provide a way to combine them arithmetically. In this subsection we define common arithmetic interaction operators for stochastic values represented by normal distributions by taking advantage of the fact that normal

distributions are closed under linear combinations [LM86].

For each arithmetic operation, we define a rule for combining stochastic values based on standard statistical error propagation methods [Bar78]. In the following, we assume that point values are represented by P and all stochastic values are of the form (m_i, sd_i) and represent normal distributions, where m_i is the mean and sd_i is the standard deviation.

When combining two stochastic values, two cases must be considered: when the distributions are correlated and when they are uncorrelated. Two distributions are **correlated** when there is an association between them, that is, they jointly vary in a similar manner [DP96b]. For example, when network traffic is heavy, available bandwidth tends to be low and latency tends to be high. When network traffic is light, available bandwidth tends to be high and latency tends to be low. We say that the distributions of latency and bandwidth are correlated in this case.

When two stochastic values are **uncorrelated** they do not jointly vary in a similar manner. This may be the case when the time between measurements of a single quantity is large, or when the two stochastic values represent distinct characteristics. For example, available CPU on two machines not running any applications in common. For arithmetic on uncorrelated stochastic values, we use a probability-based square root error computation to sharpen² the intervals where possible.

Note that correlation and non-independence are not equivalent. Two correlated variables are not independent, however two non-independent variables may be uncorrelated because “the correlation coefficient ρ only measures the strength of the linear relationship between [random variables] X and Y” [Ric95]. For our purposes, we examine only correlation. This may be a source of error in the final predictions if two parameters are not independent but are uncorrelated to one

²**Sharp** [Neu90], or tight, bounds on the resulting intervals of a computation using stochastic values are especially important for predictions to be used in scheduling where tighter bounds on a predicted execution time can lead to a more efficient execution time. A set of bounds $[l, u]$ are sharper than another set $[l', u']$ if $l' \leq l$ and $u' \geq u$.

another.

Table 4.1 summarizes the arithmetic operations between a stochastic value and a point value, two stochastic values from correlated distributions, and two stochastic values from uncorrelated distributions. In the following subsections we give an example of how each is used in structural models.

	Addition	Multiplication
Point Value and Stochastic Value	$(m_i, sd_i) + P = ((m_i + P), sd_i)$	$P(m_i, sd_i) = (Pm_i, Psd_i)$
Stochastic Values with Correlated Distributions	$\sum_{i=1}^n (m_i, sd_i) = \left(\sum_{i=1}^n m_i, \sum_{i=1}^n sd_i \right)$	$(m_i, sd_i)(m_j, sd_j) = (m_i m_j, (sd_i m_j + sd_j m_i + sd_i sd_j))$
Stochastic Values with Uncorrelated Distributions	$\sum_{i=1}^n (m_i, sd_i) \approx \left(\sum_{i=1}^n m_i, \sqrt{\sum_{i=1}^n sd_i^2} \right)$	$(m_i, sd_i)(m_j, sd_j) \approx \left(m_i m_j, \left(m_i m_j \sqrt{\left(\frac{sd_i}{m_i}\right)^2 + \left(\frac{sd_j}{m_j}\right)^2} \right) \right)$

Table 4.1: Arithmetic Combinations of a Stochastic Value with a Point Value and with other Stochastic Values [Bar78].

Addition/Subtraction by a Stochastic Value

The communication component model for the SOR given in Equation 3.34 provides an example of the addition of two stochastic values. Another example of a model in which two stochastic values must be added is

$$\mathbf{Comm} = Latency + \frac{MsgSize}{Bandwidth} \quad (4.5)$$

where both *Latency* and *Bandwidth* are stochastic values.

We define the sum of two correlated stochastic values to be the sum of their means and the sum of their standard deviations:

$$\sum_{i=1}^n (m_i, sd_i) = \left(\sum_{i=1}^n m_i, \sum_{i=1}^n sd_i \right) \quad (4.6)$$

When two stochastic values are uncorrelated, their values are independent, so we use a probability-based square root error computation:

$$\sum_{i=1}^n (m_i, sd_i) \approx \left(\sum_{i=1}^n m_i, \sqrt{\sum_{i=1}^n sd_i^2} \right) \quad (4.7)$$

Subtraction of two stochastic values would have the same form as addition, only with a negative value for one of the m_i . Since normal distributions are closed under addition and subtraction, the resulting stochastic value will also have a normal distribution.

Multiplication/Division by a Stochastic Value

While less common in structural modeling than addition, there are times when it is necessary to multiply two stochastic values. For example, in Equation 3.9, we multiply $\mathbf{AvailCPU}(S_i)$ by a value generated by a dedicated benchmark. If the application is data dependent or nondeterministic, such as is the case with the Genetic Algorithm Code (Section 2.D.1), then it is likely that the dedicated benchmark may be represented by a stochastic value. If both $\mathbf{AvailCPU}(S_i)$ and the benchmark are stochastic values, we would need to calculate their product.

As with the addition of two stochastic values, there are two cases for multiplying stochastic values³. When the distributions are correlated we can use a formula similar to standard statistical error propagation

$$(m_i, sd_i)(m_j, sd_j) = (m_i m_j, (sd_i m_j + sd_j m_i + sd_i sd_j)) \quad (4.8)$$

³We treat division of (m_i, sd_i) by (m_j, sd_j) as multiplication of (m_i, a_i) by (m_j^{-1}, sd_j^{-1}) for $m_j \neq 0$.

In cases where the distributions are uncorrelated, or when $sd_i sd_j$ is very small compared to the other terms (which should be common for high quality [low variance] information), multiplication is much like the rule for addition of uncorrelated stochastic values, and given by the following formula

$$(m_i, sd_i)(m_j, sd_j) \approx \left(m_i m_j, \left(m_i m_j \sqrt{\left(\frac{sd_i}{m_i}\right)^2 + \left(\frac{sd_j}{m_j}\right)^2} \right) \right) \quad (4.9)$$

In the case where either m_i or m_j is equal to zero, we define their product to be zero.

Note that the product of stochastic values with normal distributions does not itself have a normal distribution. Rather, it is long-tailed. In many circumstances, we can approximate the long-tailed distribution with a normal distribution and ignore the tail as discussed below in Section 4.B.3.

Collective Operations over Stochastic Values

Often structural models will combine components using collective operations such as *Max*, *Min*, or other operators. The combination of stochastic values for operations over a group must often be addressed in a situation-dependent manner.

For example, consider the calculation of the *Max* operator. Depending on the penalty for an incorrect prediction, different approaches may be taken. *Max* could be calculated by selecting the largest mean of the stochastic value inputs, or by selecting the stochastic value with the largest magnitude value in its entire range. For example, to compute the *Max* of three possible stochastic values $\{A = (4, 0.5), B = (3, 2), C = (3, 1)\}$, *A* has the largest mean, and *B* has the largest value within its range. However on average, the values of *A* are likely to be higher than the values of *B*. The usage of the resulting *Max* value, and the quality of the result required, will determine how *Max* should be calculated. This information would need to be supplied by the model builder, scheduler or user.

4.B.3 Using Normal Distributions for Non-normal Characteristics

In this section, we provide examples of stochastic values for system and application characteristics which are not normal, but can be represented by normal distributions in many cases.

Long-tailed Data

Not all system characteristics can be cleanly represented by normal distributions. Figure 4.3 shows the pdf and cdf for bandwidth data between two workstations over 10 Mbit ethernet. This is a typical graph of a **long-tailed distribution**, that is, the data has a threshold value, and varies monotonically from that point, generally with the median value several points below (or above) the threshold. A similarly shaped distribution, shown in Figure 4.4, may be found in data resulting from dedicated runs of the Genetic Algorithm code.

Neither of these distributions are normal, however, it may be useful to approximate them using normal distributions. Note that context is crucial. That is, it is not always appropriate to approximate long-tail distributions in this way. For example, if a prediction were to be used to make an absolute guarantee of service, we could not approximate long-tailed distributions with a normal distribution because crucial values could be lost. In that case, it might be better to represent the data using a lower and upper bound or a histogram, as described in the following sections.

There are important tradeoffs in this work between efficiency of computing the prediction and the quality of the resulting prediction. Normal distributions are a good substitution for long-tailed distributions only when inaccuracy in the data represented by the normal distribution can be tolerated by the scheduler, performance model, or other mechanism that uses the data. Future work may include examining other possible distributions and the feasibility of combining them for

this setting.

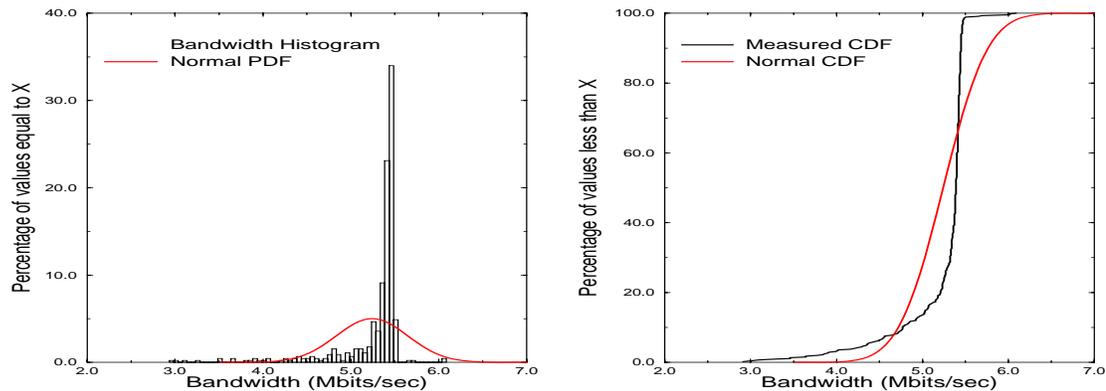


Figure 4.3: Graphs showing the pdf and cdf for bandwidth between two workstations over 10Mbits ethernet with long-tailed distribution and corresponding normal distribution.

Modal Distributions

For some application or system characteristics, the data is multi-modal. For example, Figure 4.5 shows a histogram of available CPU data for an Ultra Sparc workstation (Thing2 in the PCL Cluster) running Solaris taken over 12 hours using `vmstat`. The Unix tool `vmstat` reports the exact CPU activity at a given time, in terms of the processes in the run queue, the blocked processes, and the swapped processes. Because it takes a snapshot of the system every n seconds (where for our trace, $n = 5$), it reports what you would expect to see on a round-robin scheduled system at a fine grain.

The majority of the data lies in three modes: a mode centered at 0.94, a mode centered at 0.49, and another mode centered at 0.33. In the case of available CPU, the modes are often the result of the scheduling algorithm of the operating system. For example, most Unix-based operating systems use a round-robin algorithm to schedule CPU bound processes. When a single process is running, it receives all of the CPU. When two processes are running, each uses approxi-

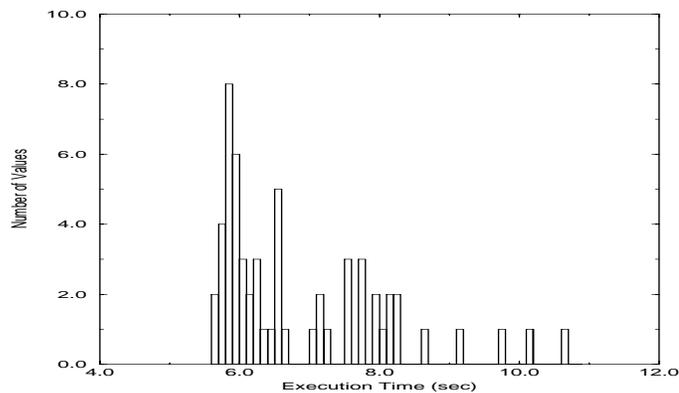


Figure 4.4: Graph showing histogram of dedicated behavior of GA code for problem size 800 on a Sparc 10.

mately half of the CPU, when there are three, each gets a third, etc. This is the phenomenon exhibited in Figure 4.5.

Recall that we used a time frame of $T=5$ minutes for these experiments, as explained in Section 4.B.1. Examining data on this scale, we can characterize modal data into one of two classes: data that shows temporal locality, and data that does not show temporal locality. If the value of the characteristic remains within a single mode during the time frame of interest, we say that the value shows **temporal locality**. An example of this (from the 24 CPU hour trace) is shown as a time series in Figure 4.6. When this occurs, we can approximate the available CPU as a normal distribution based on the data mean and standard deviation of the appropriate mode without excessive loss of information.

If the data changes modes frequently or unpredictably, we say that it that it exhibits **temporal non-locality**. An example of this, again taken from the 24 hour CPU trace, is shown as a time series in Figure 4.7. In this case, some way of deriving a prediction must be devised which takes into account the fluctuation of the data between multiple modes.

One approach is to simply ignore the multi-modality of the data and represent the stochastic value as a normal distribution based on the mean and

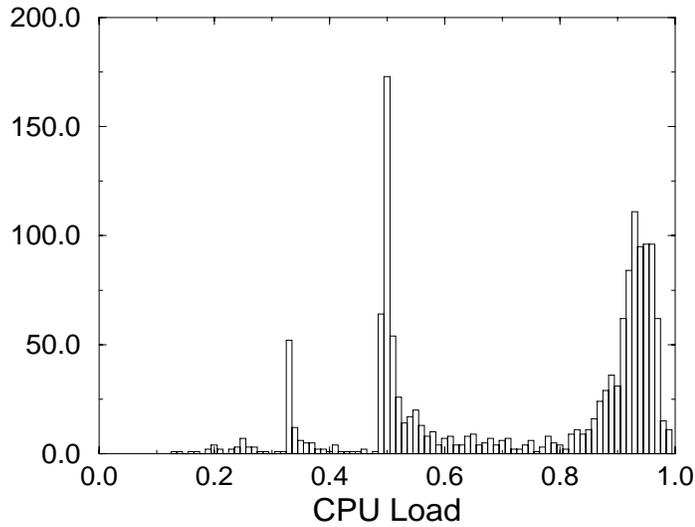


Figure 4.5: Available CPU on a production workstation.

standard deviation of the data as a whole. However, in the case of temporal non-locality, this approximation may no longer accurately capture the relevant behavior characteristics of the data. Because of the multi-modal behavior, a system characteristic with a small variance in actuality is now represented by a normal distribution with a large variance (and a large standard deviation). For the data in Figure 4.7, the mean was 0.66 and the standard deviation was 0.24.

It was observed that while the standard deviation for the data set as a whole was quite large, the standard deviation for each mode was very small in comparison. To try to take into consideration the multi-modal characteristic of the data, one approach is to calculate an aggregate mean and standard deviation for the value based on the mean and standard deviation for each mode. Let (m_i, sd_i) represent the mean and the standard deviation for the data in mode i . We define the **Aggregate Mean (AM)** and the **Aggregate Standard Deviation (ASD)** of a multi-modal distribution by:

$$AM = \sum p_i(m_i) \quad (4.10)$$

$$ASD = \sum p_i(sd_i) \quad (4.11)$$

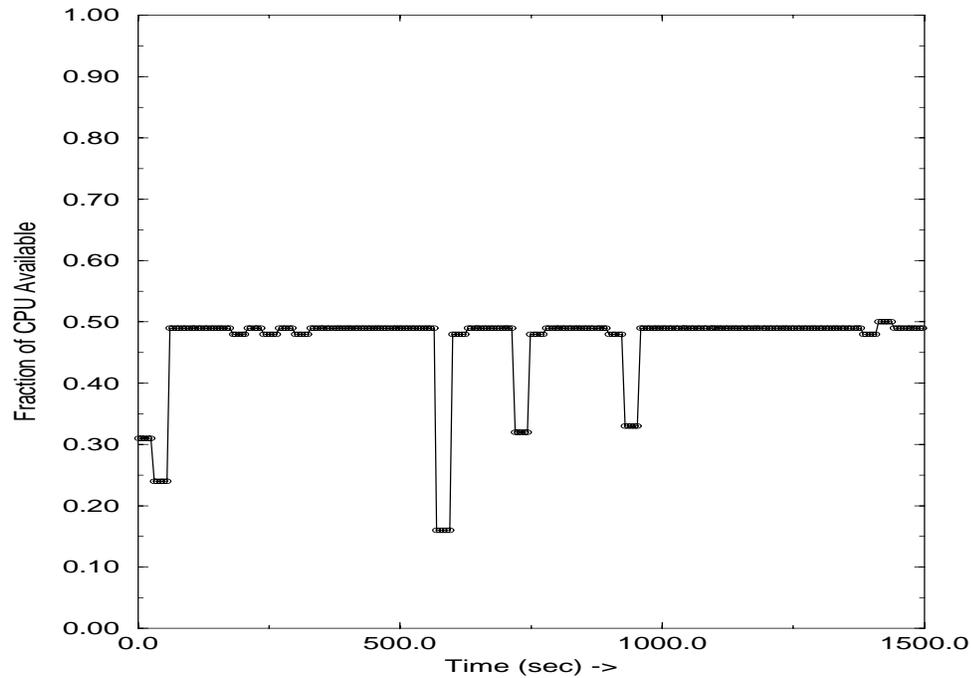


Figure 4.6: Time series showing temporal locality of CPU data.

where p_i is the percentage of data in mode i . Since we represent each individual mode in term of a normal distribution, (AM, ASD) will also have a normal distribution. For the data in Figure 4.7, $AM = 0.68$ and $ASD = 0.031$.

4.B.4 Discussion

In this section we made a key assumption that the values in the distribution were “close” to (could be adequately represented by) normal distributions. In order to define “close” we can consider several methods for determining the similarity between a given data set and the normal distribution represented by its data mean and its data standard deviation.

One common measurement of goodness-of-fit is the chi-squared (χ^2) technique [DP96c]. This is a quantitative measure of the extent to which observed counts differ from the expected counts over a given range, called a cell. The value

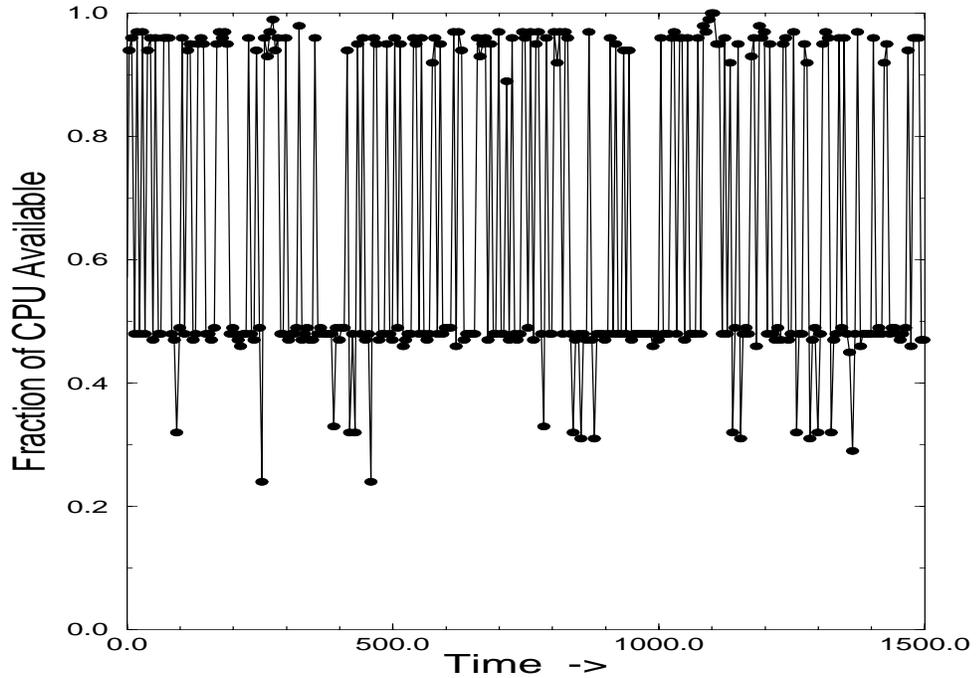


Figure 4.7: Time series showing non-temporal locality of CPU data.

for χ^2 is the sum of a goodness-of-fit for all quantities:

$$\chi^2 = \sum_{\text{all cells}} \frac{(\text{observed cell count} - \text{expected cell count})^2}{\text{expected cell count}} \quad (4.12)$$

for the observed data and the expected data resulting from the normal distribution. The value of the χ^2 statistic reflects the magnitude of the discrepancies between observed and expected cell counts, a larger value indicates larger discrepancies.

Another metric of closeness in the literature is called 1-distance between pdf's [MLH95] where:

$$\|f_n - f_d\|_1 = \int_{-\infty}^{\infty} |f_n(x) - f_d(x)| dx \quad (4.13)$$

for function f_d for the data and f_n for the normal distribution based on the data mean and standard deviation. This corresponds to a maximal error between the functions.

For both of these metrics, a user or scheduler would need to determine a threshold for closeness acceptable for their purposes.

If we approximate non-normal data using a normal distribution there may be several effects. When the distribution of a stochastic value is represented by a normal distribution but is not actually normal, arithmetic operations might exclude values that they should not. By performing arithmetic on the mean and standard deviation we are able to use optimistic formulas for uncorrelated values in order to narrow the range that is considered in the final prediction. If the distributions of stochastic values were actually long-tailed, for example, this might cut off values from the tail in an unacceptable way.

Normal distributions are closed under linear combinations [LM86], but general distributions are not. If we use arithmetic rules defined for normal distributions on non-normal data, we have no information about the distribution of the result. This implies that the intermediate and final results generated when calculating a prediction may have a distribution of unknown shape and characteristic.

Finally, it may not be possible to ascertain the distribution of a stochastic value, or the distribution may not be sufficiently “close” to normal. In such cases, other representations must be used. The next sections we explore alternatives for representing stochastic values, namely that of using intervals and using histograms.

4.C Representing Stochastic Values using Intervals

This section offers an alternative to representing stochastic values as normal distributions by representing them as intervals. The definition of an interval is given, and the rules for interval arithmetic are explained, followed by a discussion of the necessary assumptions as well as the advantages and disadvantages of this approach.

4.C.1 Defining Intervals

The simplest way to represent the variability of a stochastic value is as an interval of values. A similar approach was used for queuing network models in [LH95]. We define the **interval** of a stochastic value X to be the tuple

$$X = [\underline{x}, \bar{x}] \quad (4.14)$$

where

$$\{x \in X | \underline{x} \leq x \leq \bar{x}\} \quad (4.15)$$

The values \underline{x} and \bar{x} are called the **endpoints** of the interval. The value \underline{x} is the minimum value over all $x \in X$ and is called the **lower bound**, and the value \bar{x} is the maximum value, called the **upper bound**. The **length** of an interval $X = [\underline{x}, \bar{x}]$ is defined as $|X| = \bar{x} - \underline{x}$.

The main advantage of using intervals is the simplicity and intuitiveness of the approach. Determining the maximum and minimum value of the interval for a stochastic value is always possible, and correlates to the intuition behind defining a lower and upper bound for a value.

4.C.2 Arithmetic over Intervals

Combining stochastic values represented as intervals involves obeying well defined rules for computing interval arithmetic [Neu90]. Formulas are given in Tables 4.2, 4.3, and 4.4. The result of arithmetic on intervals in structural models is an interval which provides an upper bound and lower bound on the prediction.

Addition	$x + y = [\underline{x} + \underline{y}, \bar{x} + \bar{y}]$
Subtraction	$x - y = [\underline{x} - \bar{y}, \bar{x} - \underline{y}]$

Table 4.2: Addition and Subtraction over interval values, $x = [\underline{x}, \bar{x}]$, $y = [\underline{y}, \bar{y}]$.

	$y \geq 0$	$y \ni 0$	$y \leq 0$
$x \geq 0$	$[\underline{x}y, \bar{x}\bar{y}]$	$[\bar{x}y, \bar{x}\bar{y}]$	$[\bar{x}y, \underline{x}\bar{y}]$
$x \ni 0$	$[\underline{x}\bar{y}, \bar{x}\bar{y}]$	$[\min(\underline{x}\bar{y}, \bar{x}y), \max(\underline{x}y, \bar{x}\bar{y})]$	$[\bar{x}y, \underline{x}y]$
$x \leq 0$	$[\underline{x}\bar{y}, \bar{x}y]$	$[\underline{x}\bar{y}, \underline{x}y]$	$[\bar{x}\bar{y}, \underline{x}y]$

Table 4.3: Multiplication of interval values $x = [\underline{x}, \bar{x}]$ and $y = [\underline{y}, \bar{y}]$.

	$y > 0$	$y < 0$
$x \geq 0$	$[\underline{x}/\bar{y}, \bar{x}/y]$	$[\bar{x}/\bar{y}, \underline{x}/y]$
$x \ni 0$	$[\underline{x}/y, \bar{x}/\bar{y}]$	$[\bar{x}/\bar{y}, \underline{x}/\bar{y}]$
$x \leq 0$	$[\underline{x}/y, \bar{x}/\bar{y}]$	$[\bar{x}/y, \underline{x}/\bar{y}]$

Table 4.4: Division of interval values $x = [\underline{x}, \bar{x}]$ and $y = [\underline{y}, \bar{y}]$. Note that interval division of $\frac{x}{y}$ is only defined if 0 is not in y .

The correlation of stochastic values is also an issue when using arithmetic over intervals. Interval arithmetic is defined assuming two values may be correlated. This may lead to the overestimation of apparently simple expressions, since each occurrence of a single variable in an arithmetic expression is treated as a different variable. For example, if $A = [\underline{a}, \bar{a}]$, then $A - A = [\underline{a} - \bar{a}, \bar{a} - \underline{a}]$, instead of $[0, 0]$. This is also called the *dependence* or *simultaneity* problem [Neu90].

In some cases, this problem can be mitigated by rewriting the expression to avoid multiple occurrences of arguments [LH95]. For example, the expression $f(X, Y) = \frac{X - Y}{X + Y}$ for interval values X and Y may be rewritten as $1 - \frac{2}{1 + X/Y}$ if $0 \notin Y$. If the expression is evaluated in the latter form, it results in the exact range of $f(x, y)$ for $x \in X$ and $y \in Y$. Evaluating this expression with the intervals $X = [4, 5]$, $Y = [1, 2]$ results in $[2/7, 4/5]$ using the first formula, and $[1/3, 2/3]$ using the second.

To further reduce the effect of the simultaneity problem, Hansen [Han75] has developed a generalized interval arithmetic, but this requires additional assumptions about the nature of the intervals not available in more practical set-

tings. To obtain arbitrarily sharp results without additional assumptions usually requires using specialized techniques such as interval splitting [MR95, Ske74]. Interval splitting is a procedure that divides the interval into sub-intervals, each of which more closely approximates a uniform distribution, and results in a histogram. Other methods to obtain sharp results include recursive differentiation [Neu90], the use of Taylor expansions [Han92], or other optimization techniques. These methods may greatly increase the computational complexity of calculating an interval.

A simpler extension is possible if it can be assumed that the probabilities associated with the values in an interval are monotonic (increases or decreases throughout the entire range). When this assumption holds, a sharp interval evaluation can be computed simply by evaluating the function at the endpoints of the argument interval [LH95]. That is, instead of using interval arithmetic to solve the structural model parameterized by stochastic values $X_i = [\underline{x}_i, \overline{x}_i]$ and resulting in the prediction $P = [\underline{p}, \overline{p}]$, we solve the model twice: first using the \underline{x}_i values to evaluate \underline{p} , then solving the model using the \overline{x}_i values to evaluate \overline{p} .

4.C.3 Discussion

The primary disadvantage of using intervals is that a given interval may need to be a very large to account for outlying values. Figure 4.8 shows four possible stochastic values, all with very different distributions, all represented by the same interval. Also, this figure demonstrates that a simple interval doesn't describe the "shape" of the distribution of the stochastic value. For example, a value with two or more modes would be represented by the same interval as a value with a single mode.

Another disadvantage when using intervals is the fact that interval arithmetic assumes that the distribution of values over an interval is uniform. However, when this assumption is invalid, errors can be reduced using interval splitting [MR95, Ske74]. In general, however, these disadvantages are outweighed by the simplicity of the approach, as demonstrated in the experiments in Section 4.F.

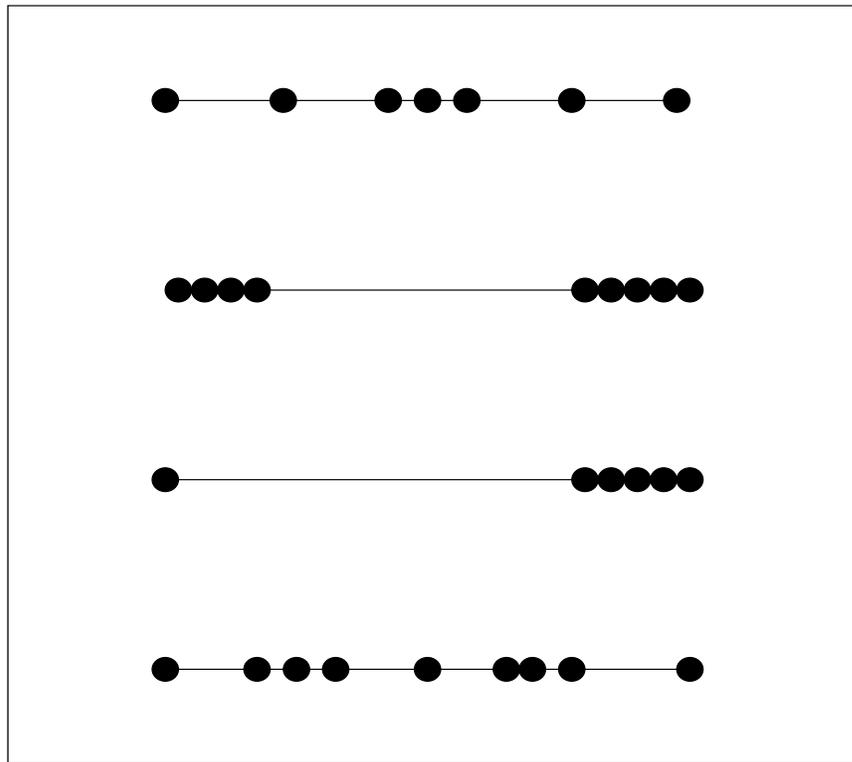


Figure 4.8: Four sample stochastic values, each with the same interval representation.

4.D Representing Stochastic Values using Histograms

A third way to represent stochastic values is by using histograms. **Histograms**, also called VU-lists [Lue98] or frequency distributions [Bla80], consist of a set of intervals with associated probabilities. In this section we define histograms and discuss alternative ways to construct them. In Section 4.D.2 we discuss the arithmetic functions needed to combine them, as well as some summary techniques presented by Lüthi [Lue98]. In Section 4.D.3 we discuss the assumptions needed for this approach, and its advantages and disadvantages.

4.D.1 Defining Histograms

A stochastic value X may be specified using a one-dimensional histogram $H(X)$ as follows:

$$\begin{aligned}
 H(X) = & X_1, X_2, \dots, X_m \\
 X_1 = & [\underline{x}_1, \overline{x}_1] : p_1 \\
 X_2 = & [\underline{x}_2, \overline{x}_2] : p_2 \\
 & \vdots \\
 X_m = & [\underline{x}_m, \overline{x}_m] : p_m
 \end{aligned} \tag{4.16}$$

where $\sum_{i=1}^m p_i = 1$. Each X_i entry in the definition of $H(X)$ is a pair consisting of an interval $[\underline{x}_i, \overline{x}_i]$, and an associated probability, p_i . Graphically, this is depicted in Figure 4.9. The histogram approach provides more information about a set of values than a single interval, but less information than a distribution since the distribution of probabilities over each sub-interval in the histogram is assumed to be uniform.

One of the primary difficulties in defining histograms for stochastic values is the determination of the appropriate intervals and probabilities for each histogram. There are a number of guidelines for defining histograms in the liter-

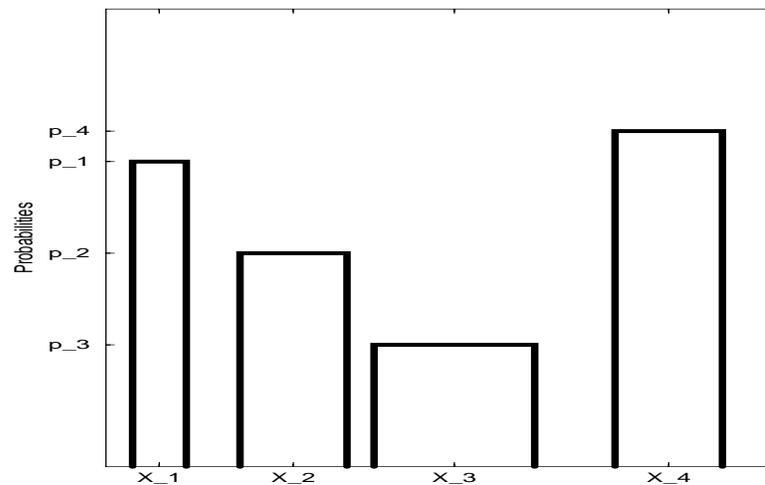


Figure 4.9: Graph showing example of a histogram.

ature. In general, the number of intervals as well as their width depends on the application and system characteristics being measured, as well as the granularity of the measurement [LMH96]. The following are “rules of thumb” used by researchers for defining histograms. The parameter v represents the quantity of data in the set of values.

- The number of intervals should be:
 - between 5 to 20, using the larger number of intervals for larger quantities of data [MSW81].
 - between 10 and 20 for $v > 50$ and between 6 and 10 for $v < 50$ [Bla80, Cra68].
 - approximately equal to \sqrt{v} [Bai72].
 - equal to $1 + 3.3 \log_{10}(v)$ [KRB83, Stu26]
- The width of all intervals should be
 - equal [Bla80, Cra68, Bai72].

– computed using [Bla80]

$$width = \frac{\text{maximum value} - \text{minimum value}}{\text{number of intervals}} \quad (4.17)$$

– calculated by $3.49\sigma v^{-1/3}$ for σ the sample standard deviation [KRB83, Sco79].

- Points of subdivision of the axis of measurement should be chosen so that it is impossible for a measurement to fall on a point of division [MSW81].

Although “rules of thumb” do not provide a rigorous approach to choosing the number of intervals in a histogram and are in fact contradictory at times, the collective experience of these researchers indicates that the “best” number of intervals to select is data-dependent, application-dependent and system-dependent, as well as dependent on how the histogram will be used.

In the most closely related work to our research [LH95], histograms are defined initially based on the number of intervals provided by the user. The intervals are then split, using interval splitting [MR95, Ske74] and a user supplied procedure to produce more subintervals for intervals with high probabilities and fewer for intervals of the same width but lower probability. This procedure is iterated until the changes that result by splitting the intervals no longer affects the resulting prediction. For their purposes, “Computational complexity is reasonable given that the number of parameters specified as histograms is not too high.” -[LMH96]

4.D.2 Arithmetic over Histograms

Given a set of k stochastic valued parameters, $D_1 \dots D_k$, represented by histograms, assume that parameter D_i has m_i intervals $\{D_{i,1}, \dots, D_{i,m_i}\}$ with

$D_{i,j} = [\underline{d}_{i,j}, \bar{d}_{i,j}] : p_{i,j}$, i.e.:

$$\begin{aligned}
 D_i &= \{D_{i,1}, D_{i,2}, \dots, D_{i,m_i}\} \quad \text{and} \\
 D_{i,1} &= [\underline{d}_{i,1}, \bar{d}_{i,1}] : p_{i,1} \\
 D_{i,2} &= [\underline{d}_{i,2}, \bar{d}_{i,2}] : p_{i,2} \\
 &\vdots \\
 D_{i,m_i} &= [\underline{d}_{i,m_i}, \bar{d}_{i,m_i}] : p_{i,m_i} \quad \text{for } i = 1 \dots k
 \end{aligned} \tag{4.18}$$

Since histograms are just sets of intervals, the interval arithmetic rules defined in the last section can be used to combine them arithmetically. However, every interval has an associated probability. Hence, when calculating $F = D \odot E$ for some arithmetic function \odot and histogram values D and E , it is necessary to calculate intermediate results for all possible combinations of intervals for each histogram. For example, given the histograms pictured in Figure 4.10:

$$\begin{aligned}
 D &\rightarrow D_1 = [1, 3] : 0.25 \\
 &\quad D_2 = [3, 5] : 0.75 \\
 E &\rightarrow E_1 = [2, 4] : 0.6 \\
 &\quad E_2 = [5, 8] : 0.4
 \end{aligned} \tag{4.19}$$

$$\begin{aligned}
 F = D + E &\rightarrow [1 + 2, 3 + 4] : (0.25 * 0.6), \\
 &\quad [1 + 5, 3 + 5] : (0.25 * 0.4), \\
 &\quad [3 + 2, 5 + 4] : (0.75 * 0.6), \\
 &\quad [3 + 5, 5 + 8] : (0.75 * 0.4)
 \end{aligned} \tag{4.20}$$

Giving intermediate results (depicted in Figure 4.11):

$$\begin{aligned}
 [3, 7] &: 0.15 \\
 [6, 8] &: 0.1 \\
 [5, 9] &: 0.45 \\
 [8, 13] &: 0.3
 \end{aligned} \tag{4.21}$$

Using the method presented below, we can combine these intermediate results into

a pdf, depicted in 4.12, with:

$$\begin{aligned}
 [3, 5] & : 0.15 \\
 [5, 6] & : 0.6 \\
 [6, 7] & : 0.7 \\
 [7, 8] & : 0.55 \\
 [8, 9] & : 0.75 \\
 [9, 13] & : 0.3
 \end{aligned}
 \tag{4.22}$$

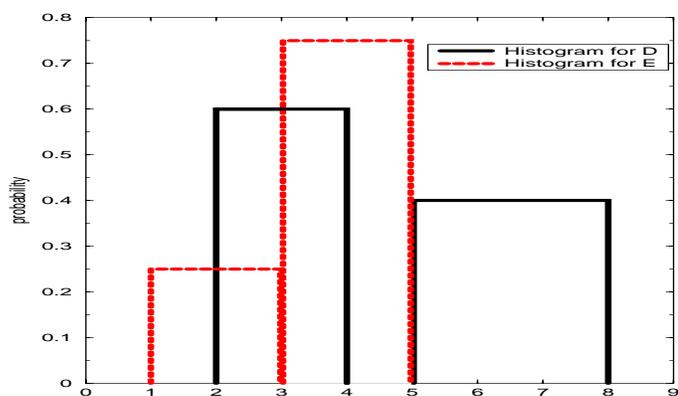


Figure 4.10: Histograms for sample parameters D and E

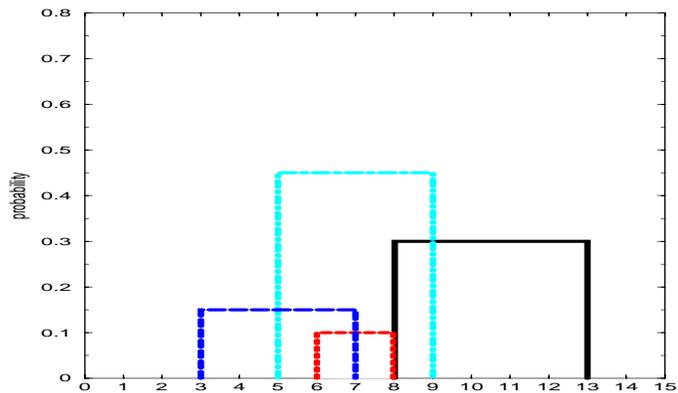


Figure 4.11: Histogram of intermediate results for $D + E$

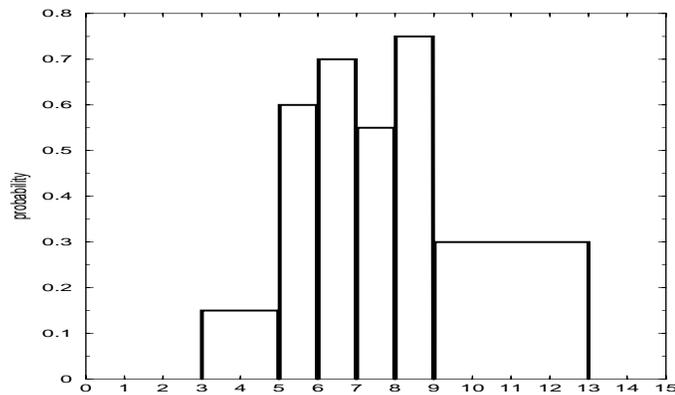


Figure 4.12: Result of D + E using Lüthi's pdf construction techniques.

Once the intermediate results have been calculated, they must be aggregated to form the result histogram, as pictured in Figure 4.12. We review the analysis given in [Lue98] to define pdf's over histograms.

Lüthi [Lue98] derives a pdf for sets of intermediate results given the assumption that none of the intermediate results are of width zero, and that each interval has a uniform distribution. The end result is the following equation deriving pdf's for the intermediate results (denoting the probability of the i th intermediate interval by f_{X_i}):

$$\begin{aligned}
 f_{X_i} &= \frac{\chi_{X_i}(x)}{\bar{x}_i - \underline{x}_i} \\
 &= \begin{cases} 0 & \text{if } x \notin X_i \\ \frac{1}{\bar{x}_i - \underline{x}_i} & \text{if } x \in X_i \end{cases} \quad (4.23)
 \end{aligned}$$

where

$$\chi_A(x) = \begin{cases} 0 & \text{if } x \notin A \\ 1 & \text{if } x \in A \end{cases} \quad (4.24)$$

These intermediate probabilities can be combined to obtain an approximation of the actual probability of the performance measure of interest by a

weighted summation. The aggregated approximation function is $f_X^{(app)}$:

$$\begin{aligned} f_X^{(app)} &= \sum_{i=1}^I p_i f_{X_i}(x) \\ &= \sum_{i=1}^I \frac{p_i}{\bar{d}_i - \underline{d}_i} \cdot \chi_{X_i}(x) \end{aligned} \tag{4.25}$$

The construction of $f_X^{(app)}$ is shown in Figure 4.12. Lüthi also presents equations for constructing cdf's.

4.D.3 Discussion

When combining stochastic values represented as histograms, it is assumed that each partial result is uniformly distributed. If this assumption does not hold, the intervals can be subdivided using interval splitting [MR95, Ske74], creating smaller intervals that are more likely to fit this assumption. Even if the intervals are relatively small, uniformity may not accurately capture the distribution of the target data.

In addition, the histogram representation assumes that each stochastic value is uncorrelated in order to determine the probabilistic percentages associated with each partial result. If the stochastic values are correlated, weights reflecting the correlation must be supplied, a difficult task in practice.

A serious drawback to using histograms in a practical setting is the lack of an agreed upon and practical definition of the number and width of the intervals to use. In our setting, predictions must be made at execution time, therefore it must be computationally efficient to determine the number and content of intervals. The number of free parameters in Lüthi's approach can make the determination of a useful histogram for the data computationally inefficient, and in this case, another approach should be used. Due to the lack of a practical definition, the exponential nature of the arithmetic, and the assumption of non-correlated values, we leave the practical extension to this approach as future work.

4.E Comparison of Stochastic Representations

The previous sections presented three representations for stochastic values: distributions, intervals and histograms. Each approach has advantages and disadvantages. Intervals are the most easily defined since the minimum and maximum value in a data set for a stochastic value are easy to determine. However, outliers in the data can affect the size of the interval, and no details about the shape of the data are included in a simple range. Histograms allow the shape of a stochastic value to be elucidated, and lend themselves to the grouping of values, but are difficult to define in a practical setting. Distributions can be defined using well understood metrics for the data, but in order to be tractable arithmetically, we must assume that the associated data fits a well-defined and computationally efficient family of distributions, such as the family of normal distributions, which is not always a valid assumption.

In the following sections, we present experimental data and results demonstrating the use of each type of representation for stochastic values as part of the calculation of a stochastic prediction.

4.F Experimental Verification

To demonstrate the usefulness of stochastic values as performance predictions, we derive stochastic execution time predictions for several applications on our distributed platforms. We use a stochastic value to represent the available CPU, and compare results representing the values as intervals and normal distributions to point value predictions. Workstations for all clusters were shared by multiple users and exhibited diverse processor speeds, available physical memory, and CPU load. The network was also shared by other users. The Network Weather Service supplied us with accurate run-time information about the CPU load on our machines as well as the variance of those values at 10 second intervals.

4.F.1 SOR on PCL

In our first set of experiments we examined the SOR code, presented in Section 2.E.1, on the PCL cluster. Figure 4.13 shows the normal distribution stochastic value predictions versus the actual time, Figure 4.14 shows the interval stochastic value predictions versus the actual time, run when the PCL cluster had CPU loads shown in Figures 4.15 through 4.18. Two machines had fairly constant load while the other two varied significantly. Both approaches do well in this situation: Using normal distribution representations, we capture 20 of 27 runs; Using interval representations, we capture 26 of 27. However, this statistic can be misleading since one reason the interval representation capture so many values is due to the fact that the bounds are rather large, most likely due to outlier values in the CPU data.

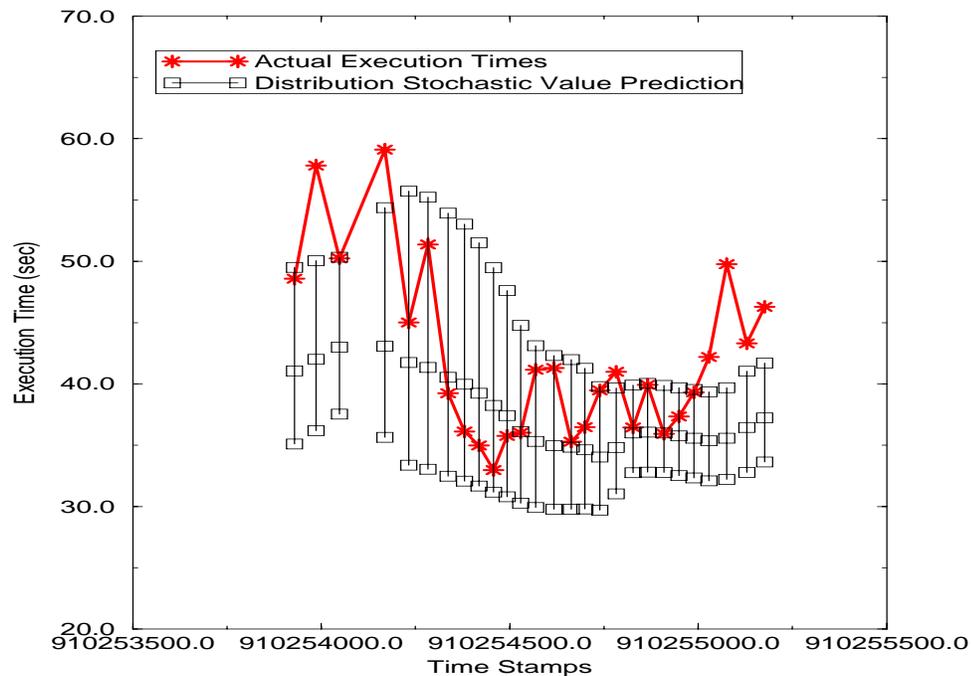


Figure 4.13: Comparison of point value prediction and stochastic value prediction represented by normal distributions for the SOR benchmark on the PCL cluster with CPU loads shown in Figures 4.15 through 4.18.

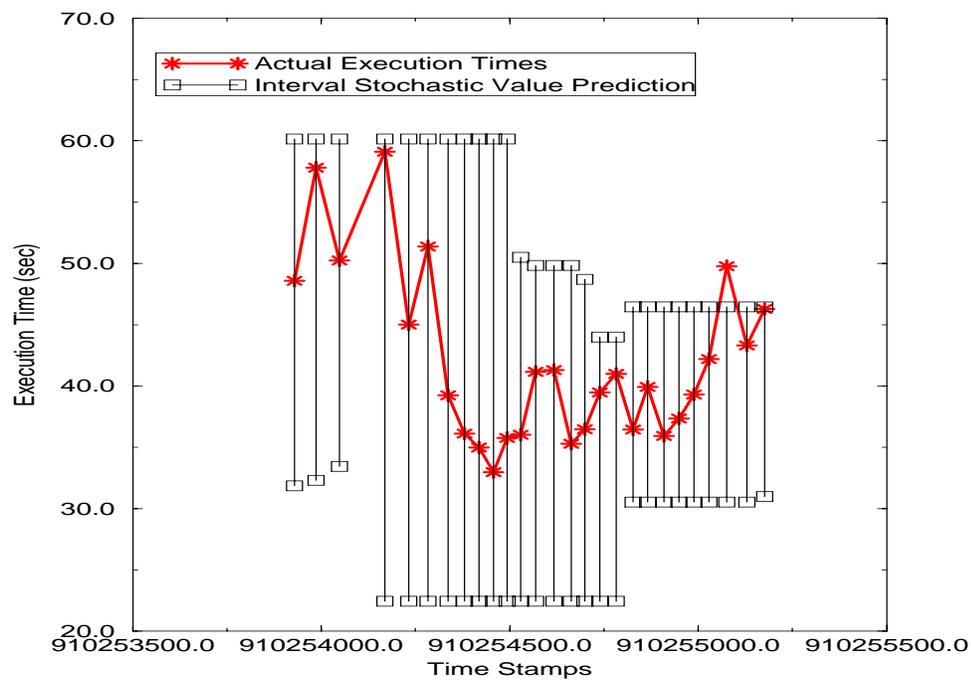


Figure 4.14: Comparison of point value prediction and stochastic value prediction represented by intervals for the SOR benchmark on the PCL cluster with CPU loads shown in Figures 4.15 through 4.18.

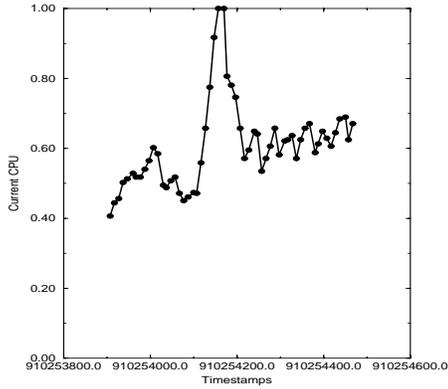


Figure 4.15: CPU values during runtime for prediction experiments depicted in Figure 4.13 and Figure 4.14 on Lorax, in the PCL cluster.

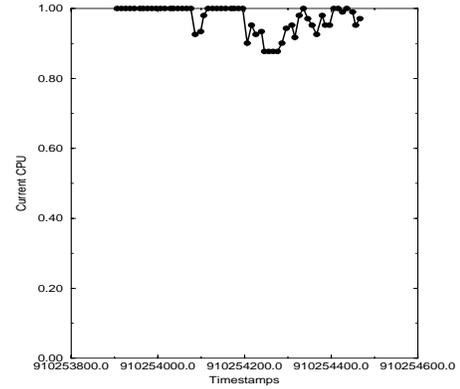


Figure 4.16: CPU values during runtime for prediction experiments depicted in Figure 4.13 and Figure 4.14 on Picard, in the PCL cluster.

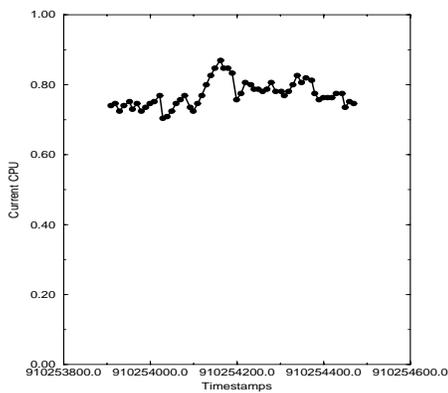


Figure 4.17: CPU values during runtime for prediction experiments depicted in Figure 4.13 and Figure 4.14 on Thing1, in the PCL cluster.

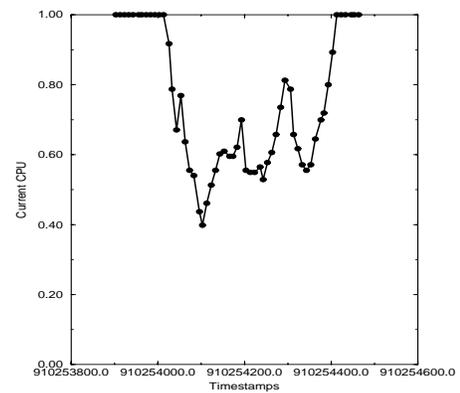


Figure 4.18: CPU values during runtime for prediction experiments depicted in Figure 4.13 and Figure 4.14 on Thing2, in the PCL cluster.

A second set of experiments for the SOR code on the PCL cluster are presented. Figure 4.19 shows the normal distribution stochastic value predictions versus the actual time, Figure 4.20 shows the interval stochastic value predictions versus the actual time, run when the PCL cluster had CPU loads shown in Figures 4.21 through 4.24. This example illustrates the effect of a high variance production system (the execution time varied over 300%). For these runs, using a normal distribution representation captured 13 of 27 runs, while the interval representation captured 24 of the 27.

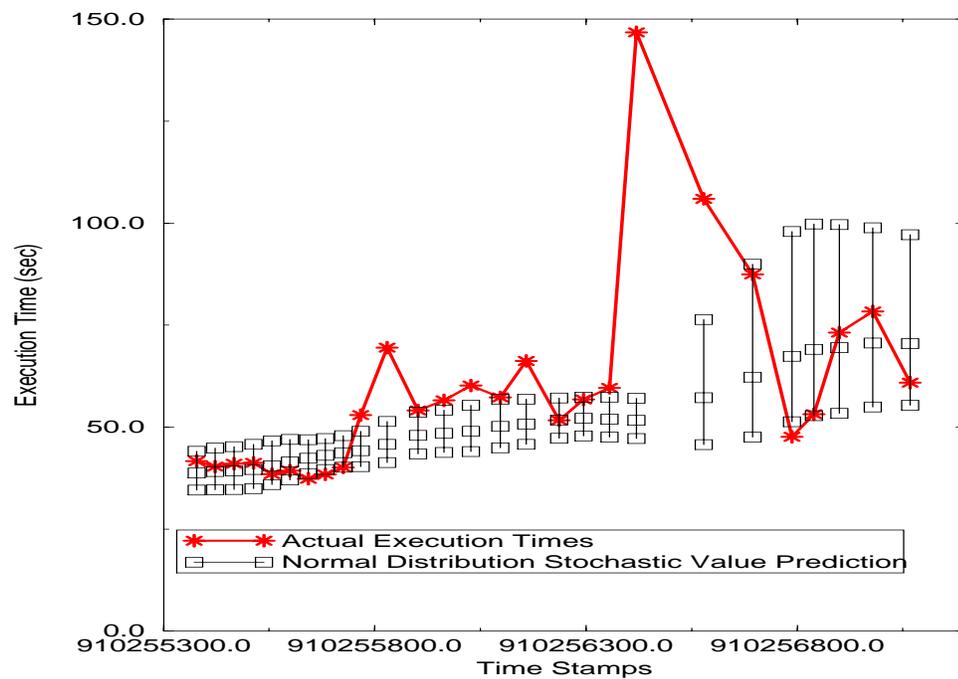


Figure 4.19: Comparison of point value prediction and stochastic value prediction represented by normal distributions for the SOR benchmark on the PCL cluster with CPU loads shown in Figures 4.21 through 4.24.

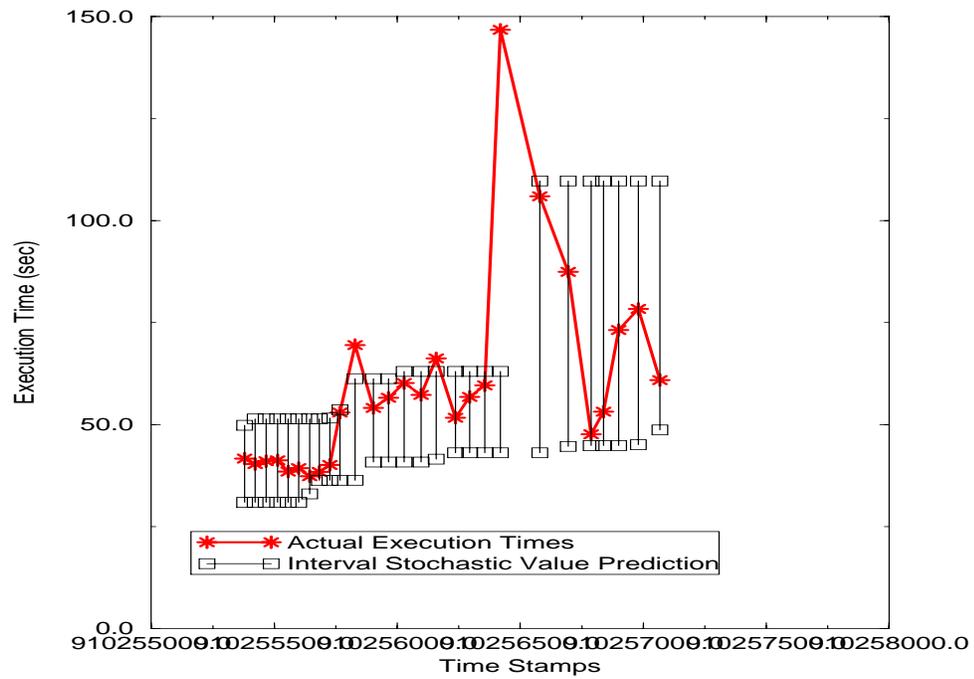


Figure 4.20: Comparison of point value prediction and stochastic value prediction represented by intervals for the SOR benchmark on the PCL cluster with CPU loads shown in Figures 4.21 through 4.24.

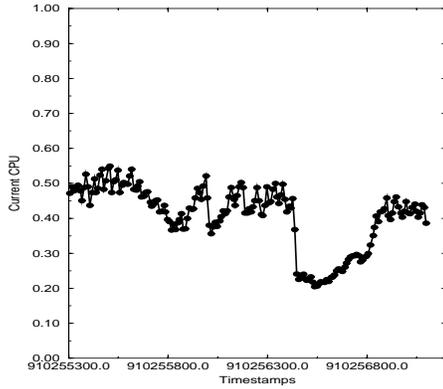


Figure 4.21: CPU values during runtime for prediction experiments depicted in Figure 4.19 and Figure 4.20 on Lorax, in the PCL cluster.

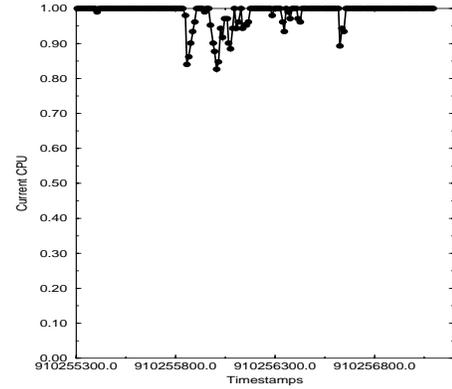


Figure 4.22: CPU values during runtime for prediction experiments depicted in Figure 4.19 and Figure 4.20 on Picard, in the PCL cluster.

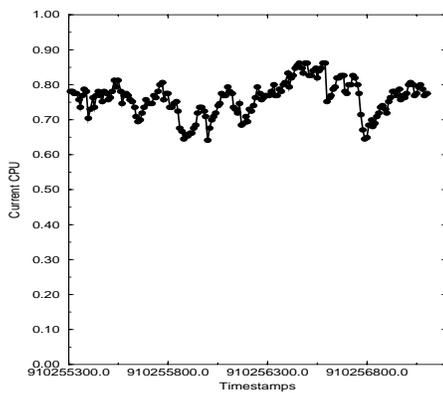


Figure 4.23: CPU values during runtime for prediction experiments depicted in Figure 4.19 and Figure 4.20 on Thing1, in the PCL cluster.

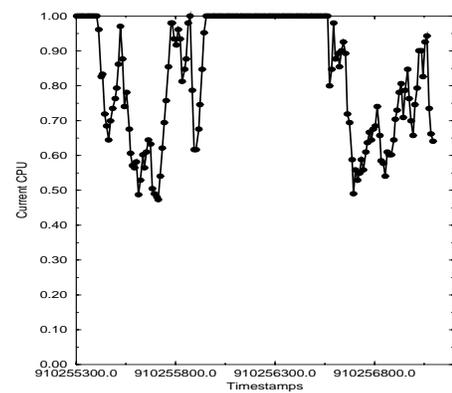
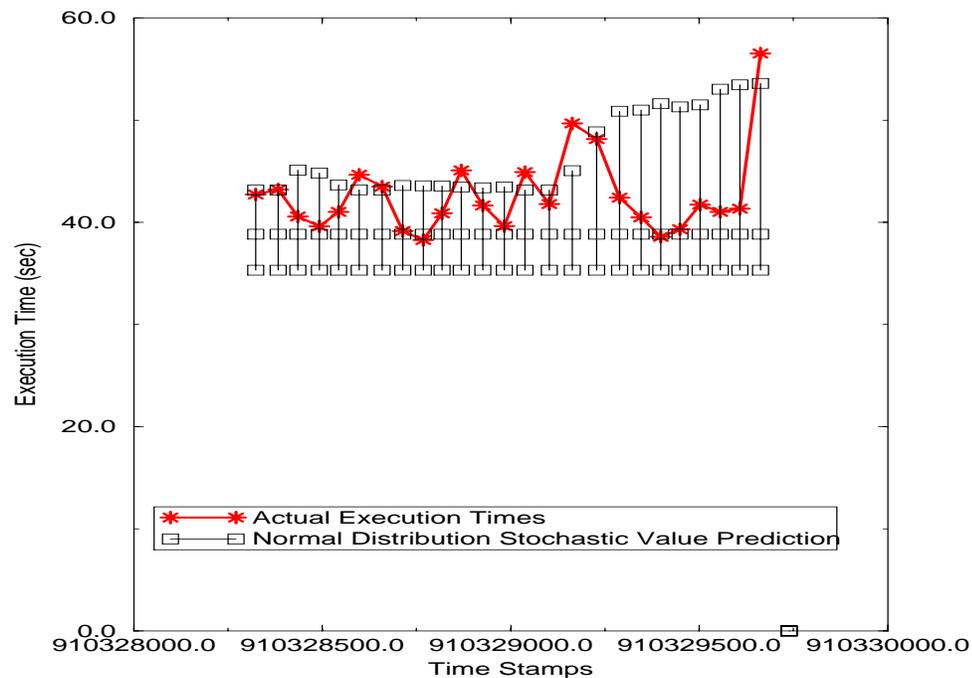


Figure 4.24: CPU values during runtime for prediction experiments depicted in Figure 4.19 and Figure 4.20 on Thing2, in the PCL cluster.

4.F.2 GA on PCL

In this set of experiments we examined the GA code, presented in Section 2.D.1, on the PCL cluster. Figure 4.25 shows the normal distribution stochastic value predictions versus the actual time, Figure 4.26 shows the interval stochastic value predictions versus the actual time, run when the PCL cluster had CPU loads shown in Figures 4.27 through 4.30. The normal distribution representation captured 18 of 25 runs, while the interval representation captured 19. Note that in this set of experiments it is especially noticeable the generally tighter bounds obtained using normal distributions, as compared to the interval representation.



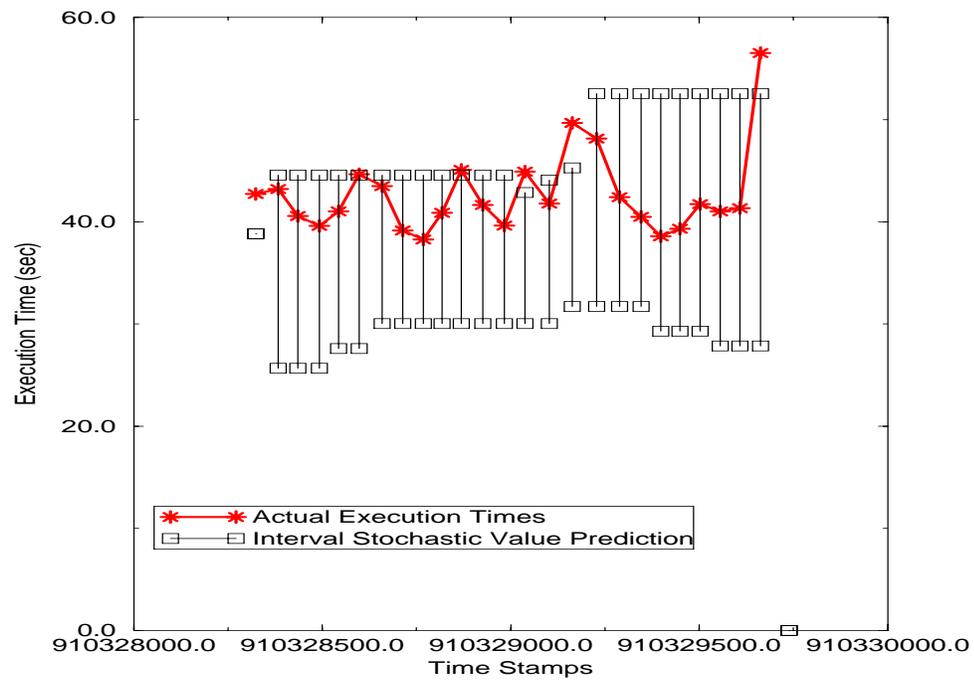


Figure 4.26: Comparison of point value prediction and stochastic value prediction represented by intervals for the GA benchmark on the PCL cluster with CPU loads shown in Figures 4.27 through 4.30.

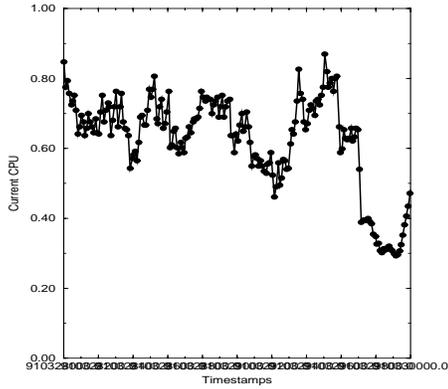


Figure 4.27: CPU values during runtime for prediction experiments depicted in Figure 4.25 and Figure 4.26 on Lorax, in the PCL cluster.

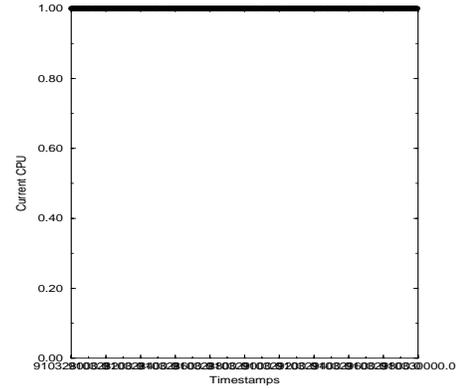


Figure 4.28: CPU values during runtime for prediction experiments depicted in Figure 4.25 and Figure 4.26 on Picard, in the PCL cluster.

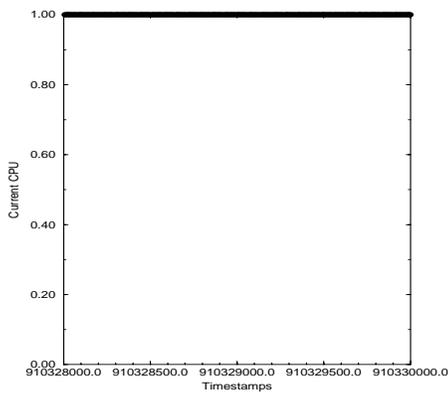


Figure 4.29: CPU values during runtime for prediction experiments depicted in Figure 4.25 and Figure 4.26 on Thing1, in the PCL cluster.

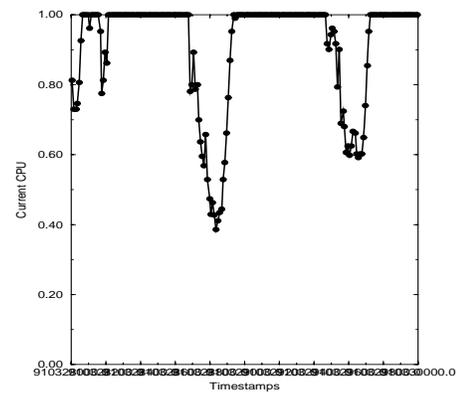


Figure 4.30: CPU values during runtime for prediction experiments depicted in Figure 4.25 and Figure 4.26 on Thing2, in the PCL cluster.

Figure 4.31 shows the normal distribution stochastic value predictions versus the actual time, Figure 4.32 shows the interval stochastic value predictions versus the actual time, run when the PCL cluster had CPU loads shown in Figures 4.33 through 4.36. The normal distribution representation captured 21 of 25 runs, while the interval representation captured 16. This is an example of when the normal distribution approach may have larger bounds than the interval approach.

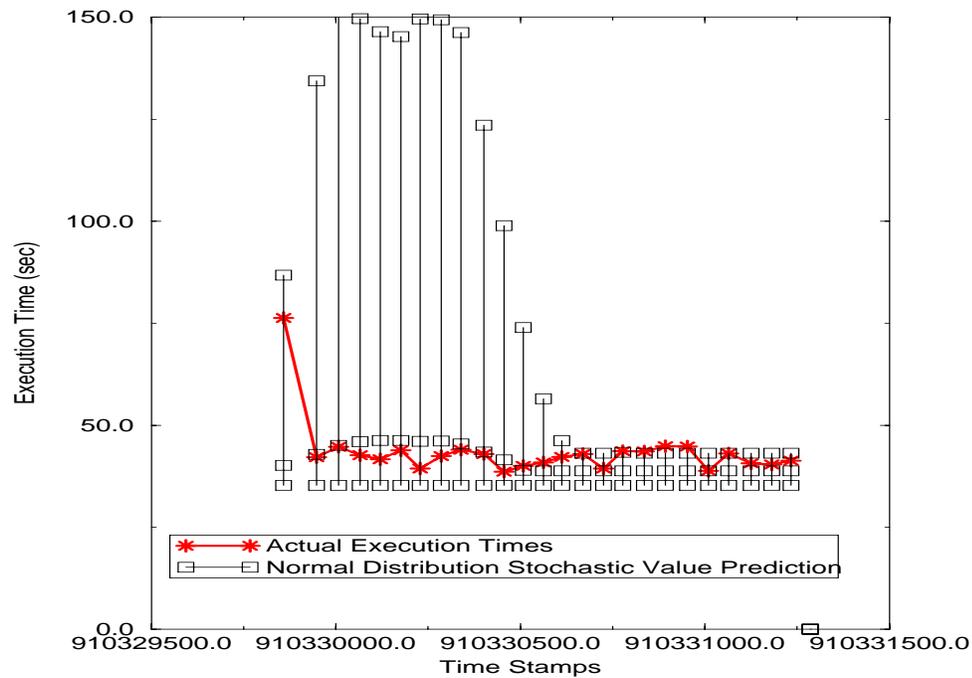


Figure 4.31: Comparison of point value prediction and stochastic value prediction represented by normal distributions for the GA benchmark on the PCL cluster with CPU loads shown in Figures 4.33 through 4.36.

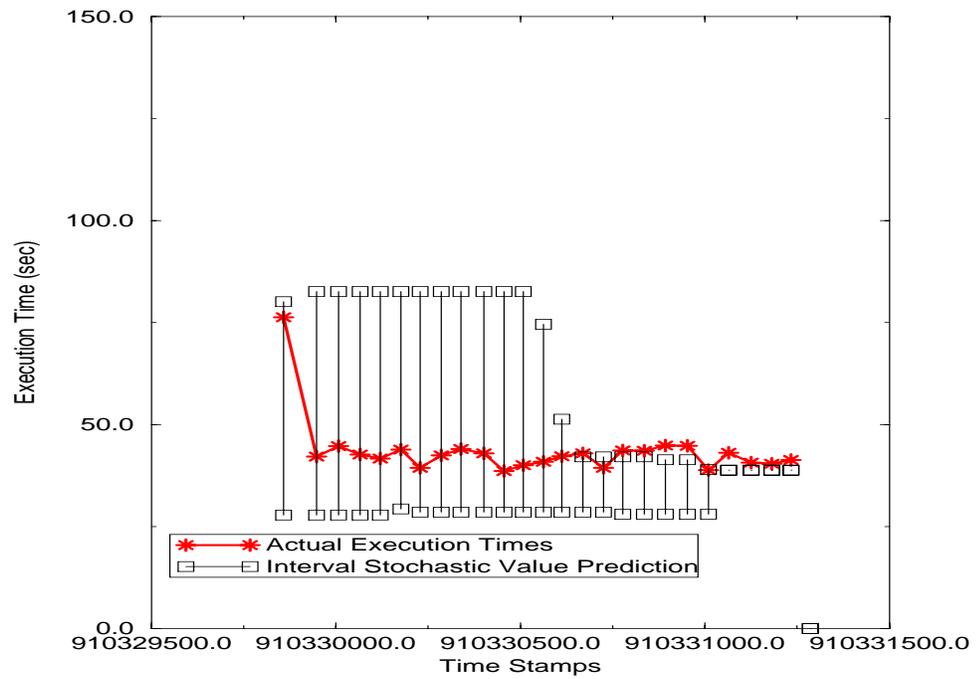


Figure 4.32: Comparison of point value prediction and stochastic value prediction represented by intervals for the GA benchmark on the PCL cluster with CPU loads shown in Figures 4.33 through 4.36.

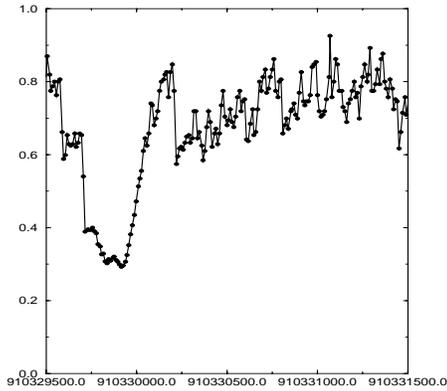


Figure 4.33: CPU values during runtime for prediction experiments depicted in Figure 4.31 and Figure 4.32 on Lorax, in the PCL cluster.

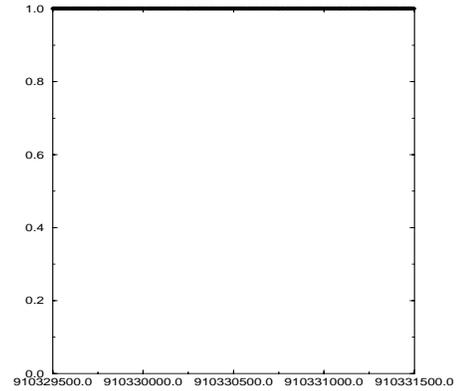


Figure 4.34: CPU values during runtime for prediction experiments depicted in Figure 4.31 and Figure 4.32 on Picard, in the PCL cluster.

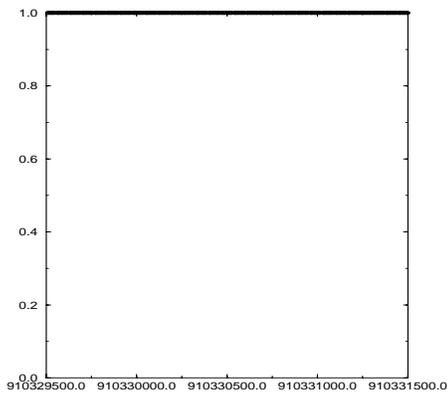


Figure 4.35: CPU values during runtime for prediction experiments depicted in Figure 4.31 and Figure 4.32 on Thing1, in the PCL cluster.

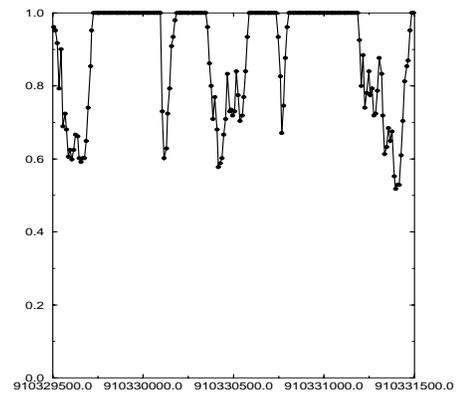


Figure 4.36: CPU values during runtime for prediction experiments depicted in Figure 4.31 and Figure 4.32 on Thing2, in the PCL cluster.

Figure 4.37 shows the normal distribution stochastic value predictions versus the actual time, Figure 4.38 shows the interval stochastic value predictions versus the actual time, run when the PCL cluster had CPU loads shown in Figures 4.39 through 4.42. The normal distribution representation captured 17 of 25 runs, while the interval representation captured 9.

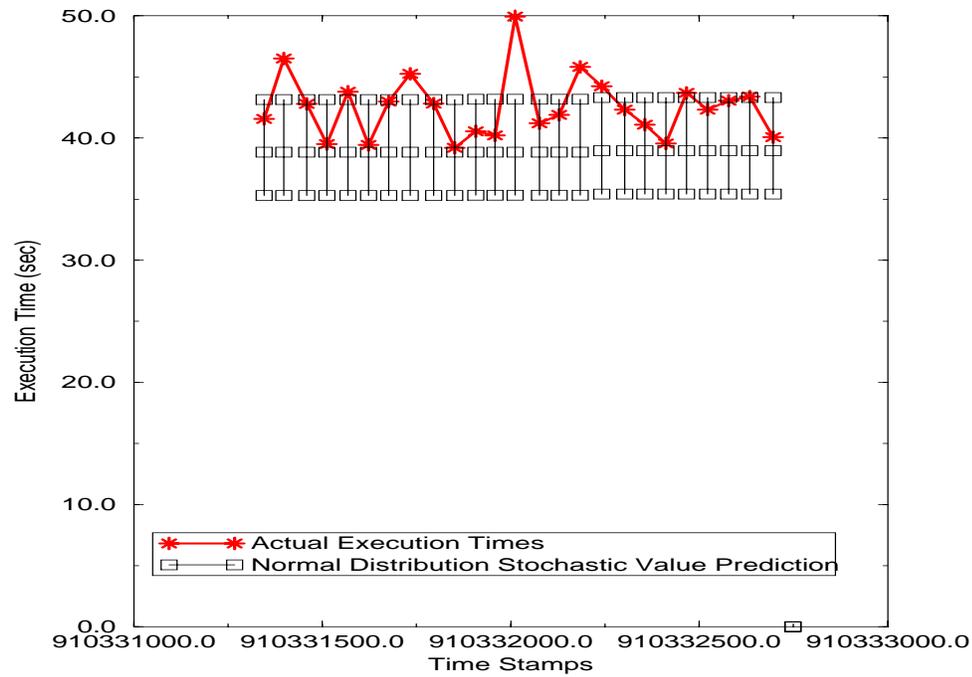


Figure 4.37: Comparison of point value prediction and stochastic value prediction represented by normal distributions for the GA benchmark on the PCL cluster with CPU loads shown in Figures 4.39 through 4.42.

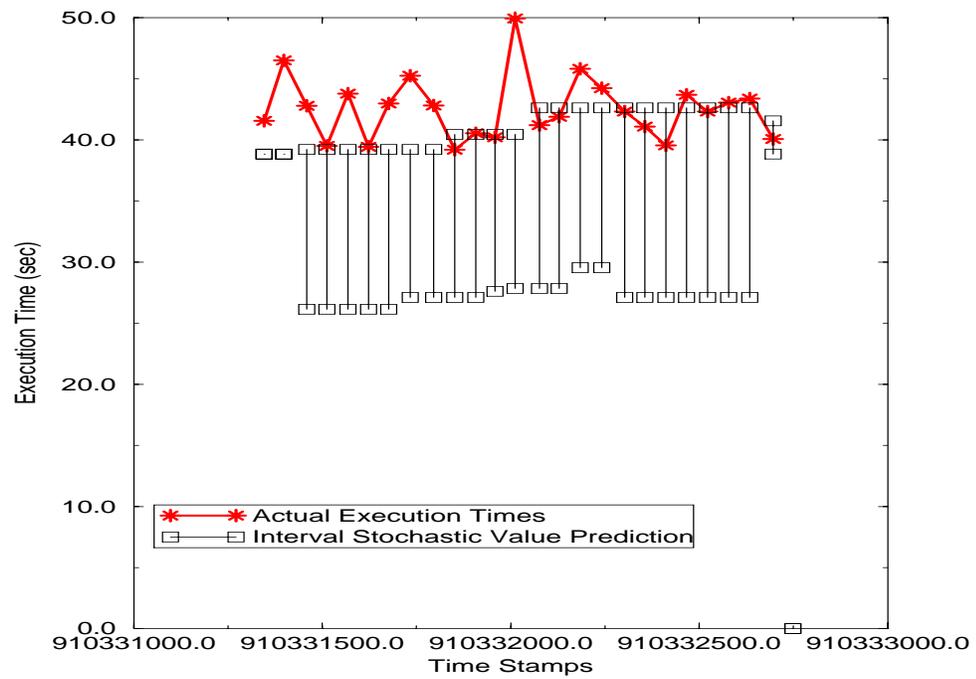


Figure 4.38: Comparison of point value prediction and stochastic value prediction represented by intervals for the GA benchmark on the PCL cluster with CPU loads shown in Figures 4.39 through 4.42.

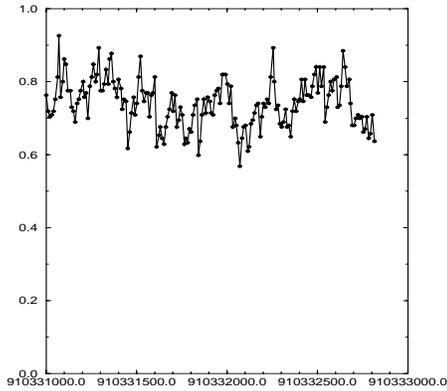


Figure 4.39: CPU values during runtime for prediction experiments depicted in Figure 4.37 and Figure 4.38 on Lorax, in the PCL cluster.

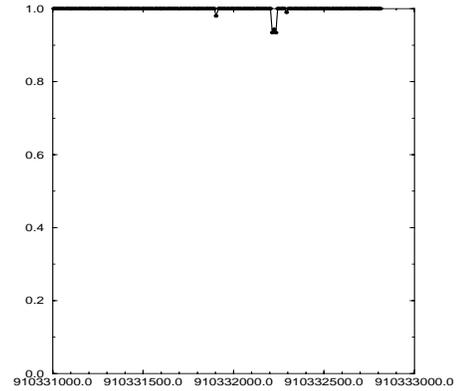


Figure 4.40: CPU values during runtime for prediction experiments depicted in Figure 4.37 and Figure 4.38 on Picard, in the PCL cluster.

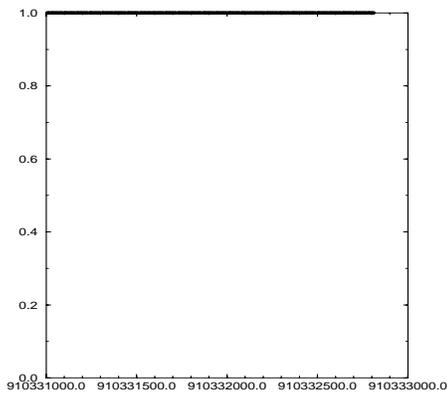


Figure 4.41: CPU values during runtime for prediction experiments depicted in Figure 4.37 and Figure 4.38 on Thing1, in the PCL cluster.

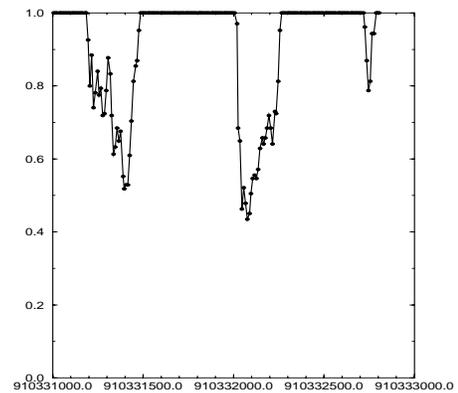


Figure 4.42: CPU values during runtime for prediction experiments depicted in Figure 4.37 and Figure 4.38 on Thing2, in the PCL cluster.

4.F.3 NBody on PCL

In our first set of experiments we examined the NBody code, presented in Section 2.D.3, on the PCL cluster. Figure 4.43 shows the normal distribution stochastic value predictions versus the actual time, Figure 4.44 shows the interval stochastic value predictions versus the actual time, run when the PCL cluster had CPU loads shown in Figures 4.45 through 4.48. This is an interesting set of experiments because it shows how the size of the prediction range changes with respect to the changing loads on the machines. Using normal distributions, we capture 24 of the 25 values, and using intervals we capture all 25.

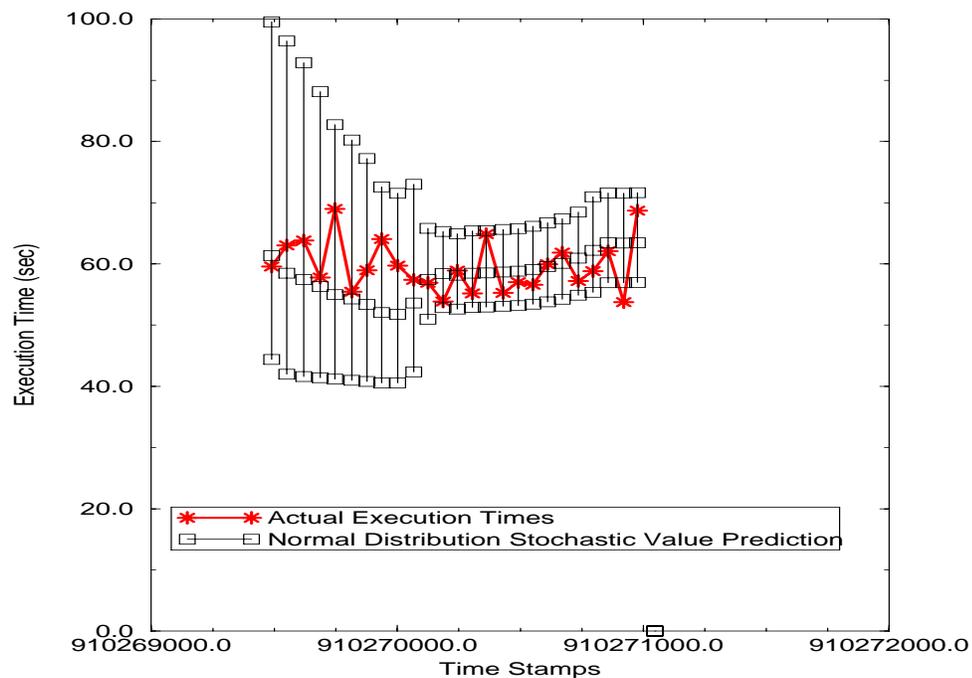


Figure 4.43: Comparison of point value prediction and stochastic value prediction represented by normal distributions for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.45 through 4.48.

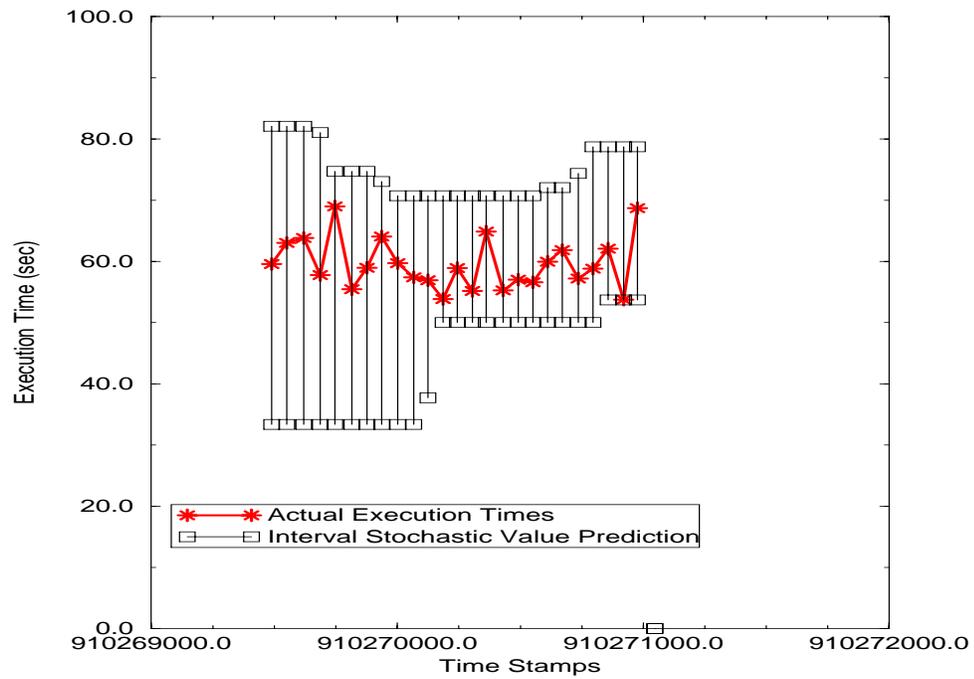


Figure 4.44: Comparison of point value prediction and stochastic value prediction represented by intervals for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.45 through 4.48.

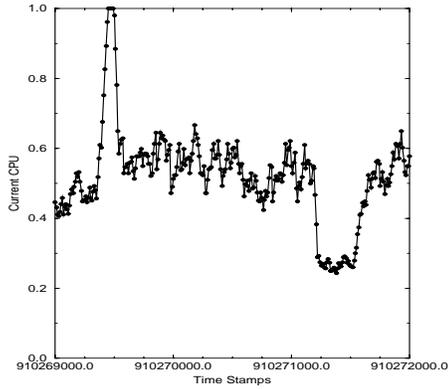


Figure 4.45: CPU values during runtime for prediction experiments depicted in Figure 4.43 and Figure 4.44 on Lorax, in the PCL cluster.

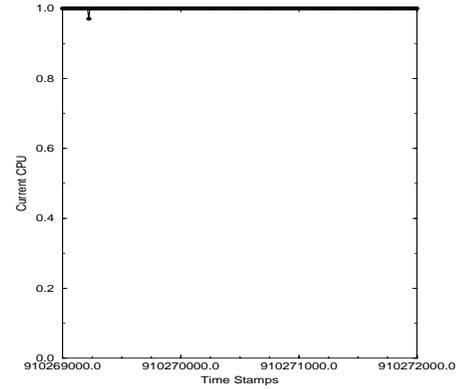


Figure 4.46: CPU values during runtime for prediction experiments depicted in Figure 4.43 and Figure 4.44 on Picard, in the PCL cluster.

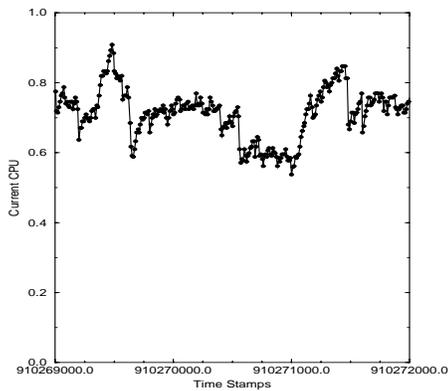


Figure 4.47: CPU values during runtime for prediction experiments depicted in Figure 4.43 and Figure 4.44 on Thing1, in the PCL cluster.

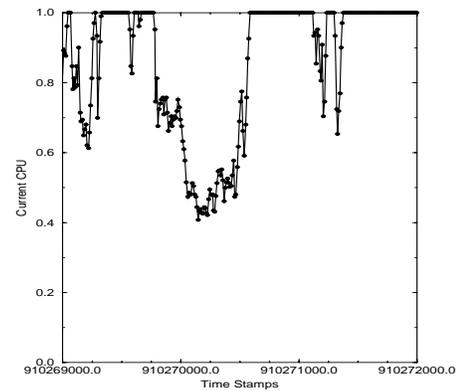


Figure 4.48: CPU values during runtime for prediction experiments depicted in Figure 4.43 and Figure 4.44 on Thing2, in the PCL cluster.

Figure 4.49 shows the normal distribution stochastic value predictions versus the actual time, Figure 4.50 shows the interval stochastic value predictions versus the actual time, run when the PCL cluster had CPU loads shown in Figures 4.51 through 4.54. Using normal distributions, we capture 16 of the 25 values, and using intervals we capture 23.

This set of experiments shows the unfortunate side-effects of using past behavior to predict future CPU usage. It is clear from figures 4.49 and 4.50 that if the predictions were shifted 5 runs left we could capture 100% of the execution times. This is an area for future work.

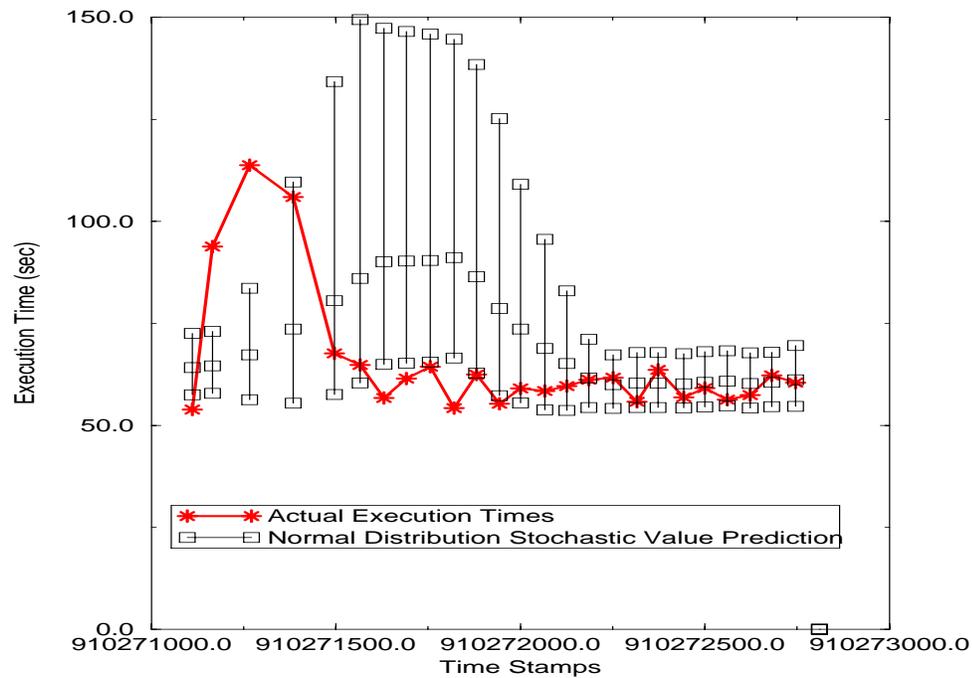


Figure 4.49: Comparison of point value prediction and stochastic value prediction represented by normal distributions for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.51 through 4.54.

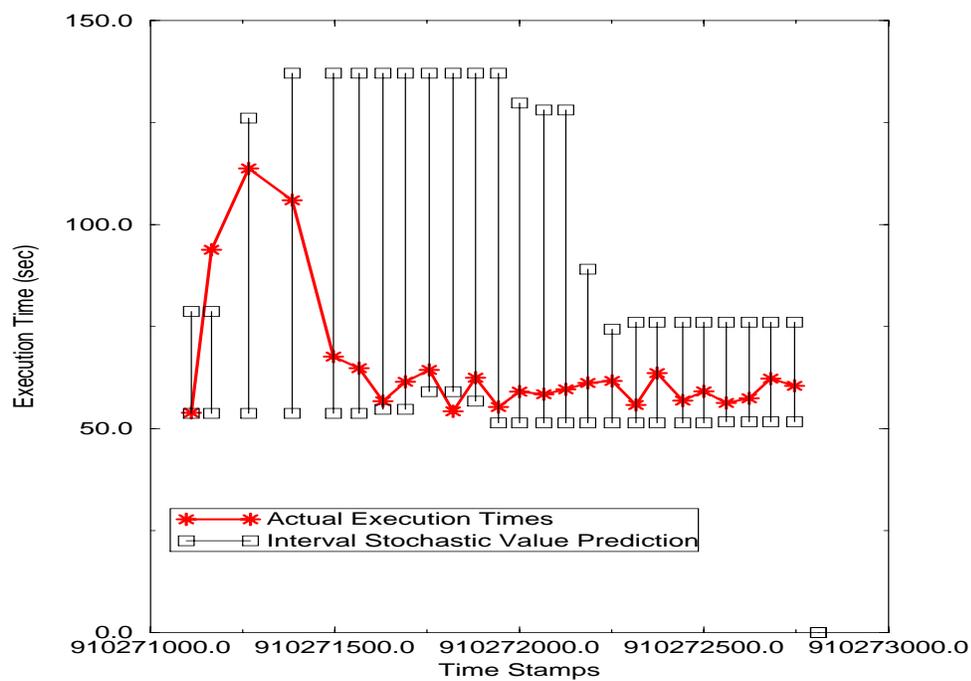


Figure 4.50: Comparison of point value prediction and stochastic value prediction represented by intervals for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.51 through 4.54.

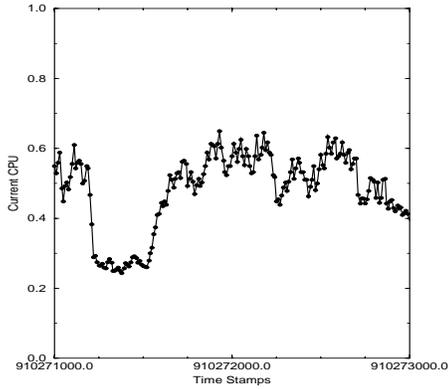


Figure 4.51: CPU values during runtime for prediction experiments depicted in Figure 4.49 and Figure 4.50 on Lorax, in the PCL cluster.

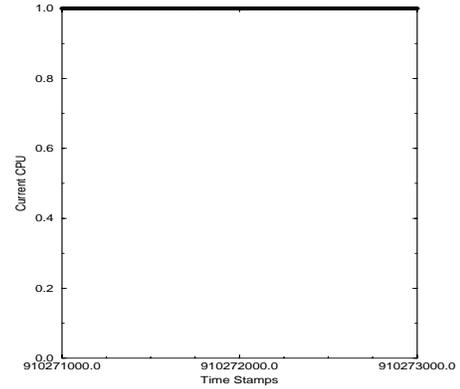


Figure 4.52: CPU values during runtime for prediction experiments depicted in Figure 4.49 and Figure 4.50 on Picard, in the PCL cluster.

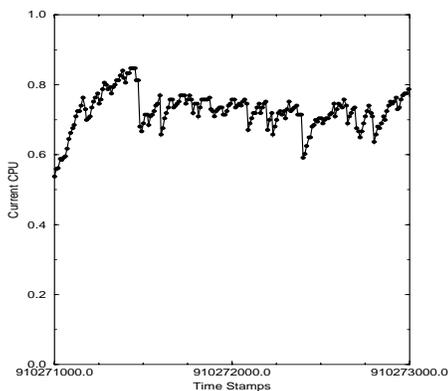


Figure 4.53: CPU values during runtime for prediction experiments depicted in Figure 4.49 and Figure 4.50 on Thing1, in the PCL cluster.

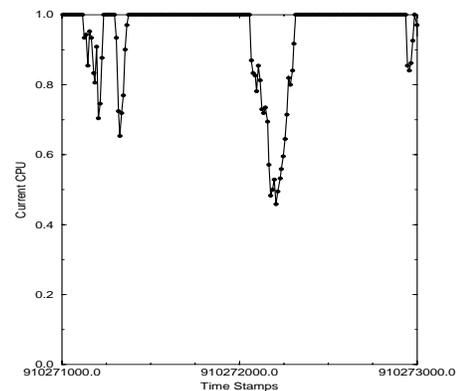


Figure 4.54: CPU values during runtime for prediction experiments depicted in Figure 4.49 and Figure 4.50 on Thing2, in the PCL cluster.

Figure 4.55 shows the normal distribution stochastic value predictions versus the actual time, Figure 4.56 shows the interval stochastic value predictions versus the actual time, run when the PCL cluster had CPU loads shown in Figures 4.57 through 4.60. Using normal distributions, we capture 17 of the 25 values, and using intervals we capture 25. These graphs show how the different approaches react differently to the changing loads.

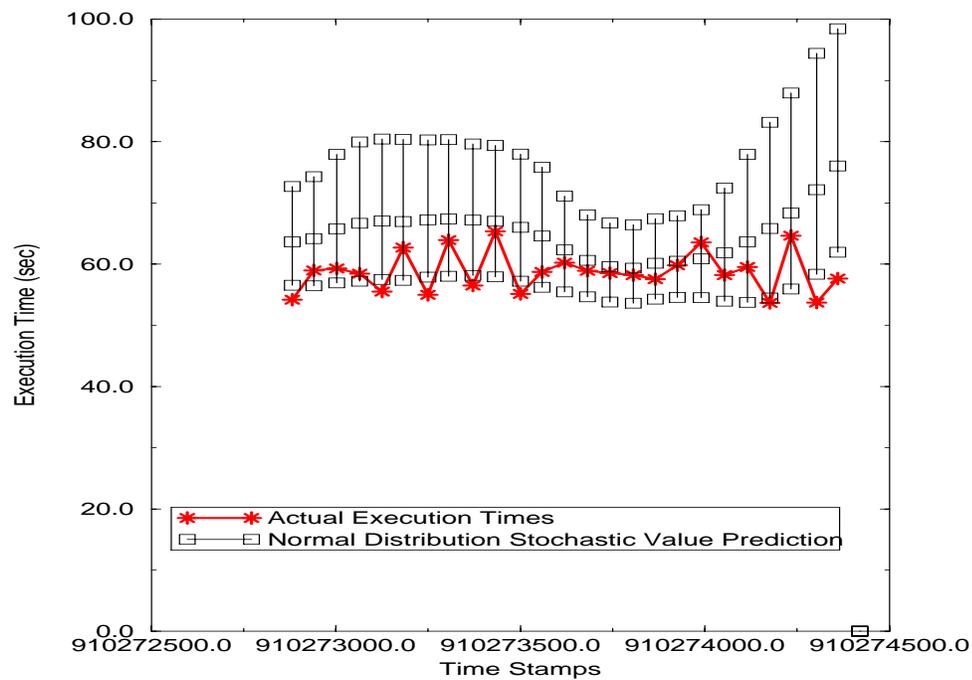


Figure 4.55: Comparison of point value prediction and stochastic value prediction represented by normal distributions for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.57 through 4.60.

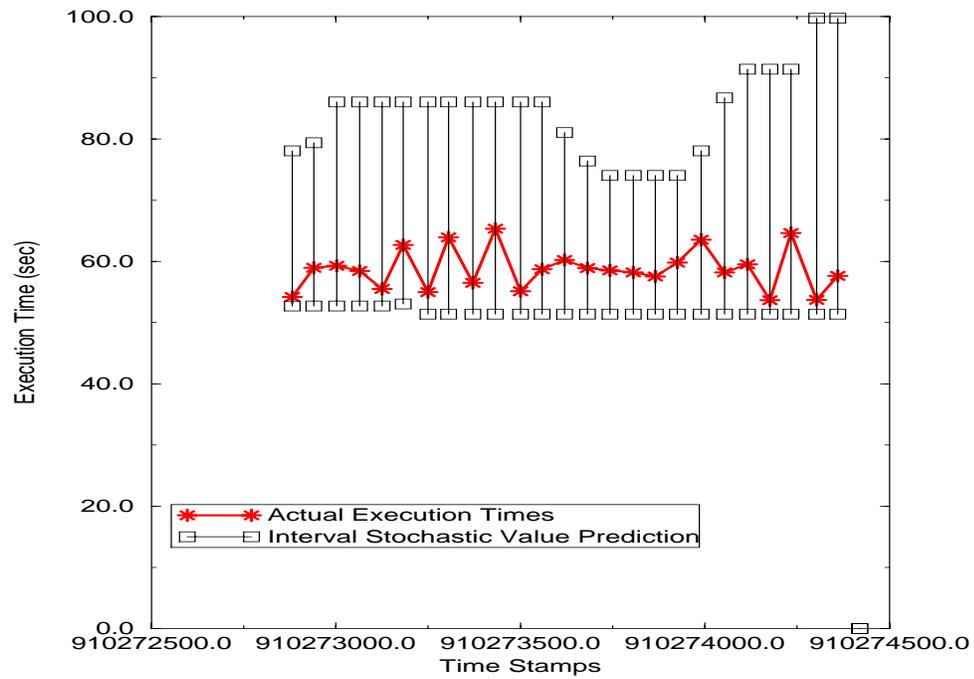


Figure 4.56: Comparison of point value prediction and stochastic value prediction represented by intervals for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.57 through 4.60.

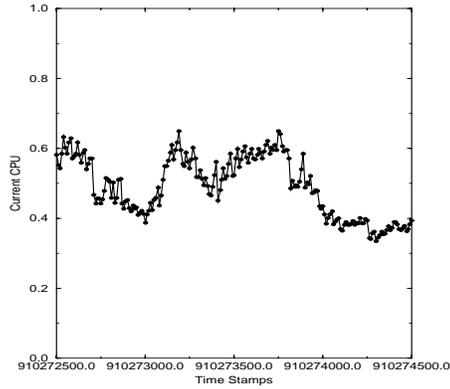


Figure 4.57: CPU values during runtime for prediction experiments depicted in Figure 4.55 and Figure 4.56 on Lorax, in the PCL cluster.

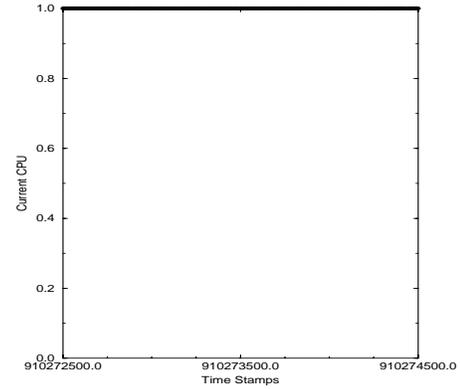


Figure 4.58: CPU values during runtime for prediction experiments depicted in Figure 4.55 and Figure 4.56 on Picard, in the PCL cluster.

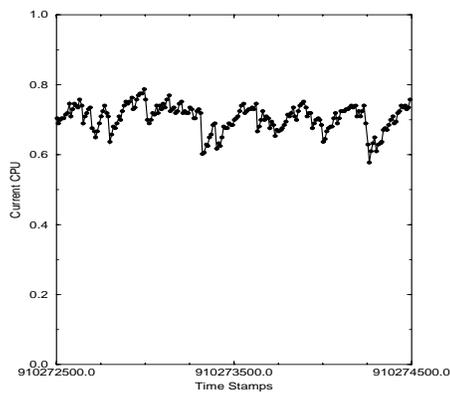


Figure 4.59: CPU values during runtime for prediction experiments depicted in Figure 4.55 and Figure 4.56 on Thing1, in the PCL cluster.

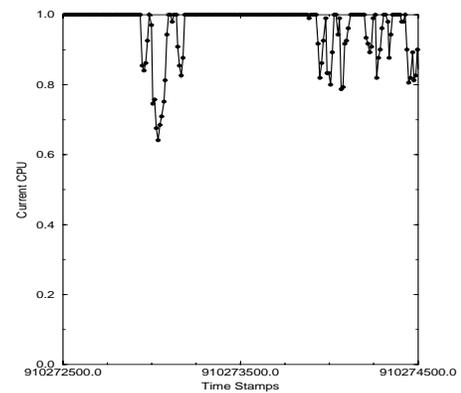


Figure 4.60: CPU values during runtime for prediction experiments depicted in Figure 4.55 and Figure 4.56 on Thing2, in the PCL cluster.

Figure 4.61 shows the normal distribution stochastic value predictions versus the actual time, Figure 4.62 shows the interval stochastic value predictions versus the actual time, run when the PCL cluster had CPU loads shown in Figures 4.63 through 4.66. Using normal distributions, we capture 22 of the 25 values, and using intervals we capture 25.

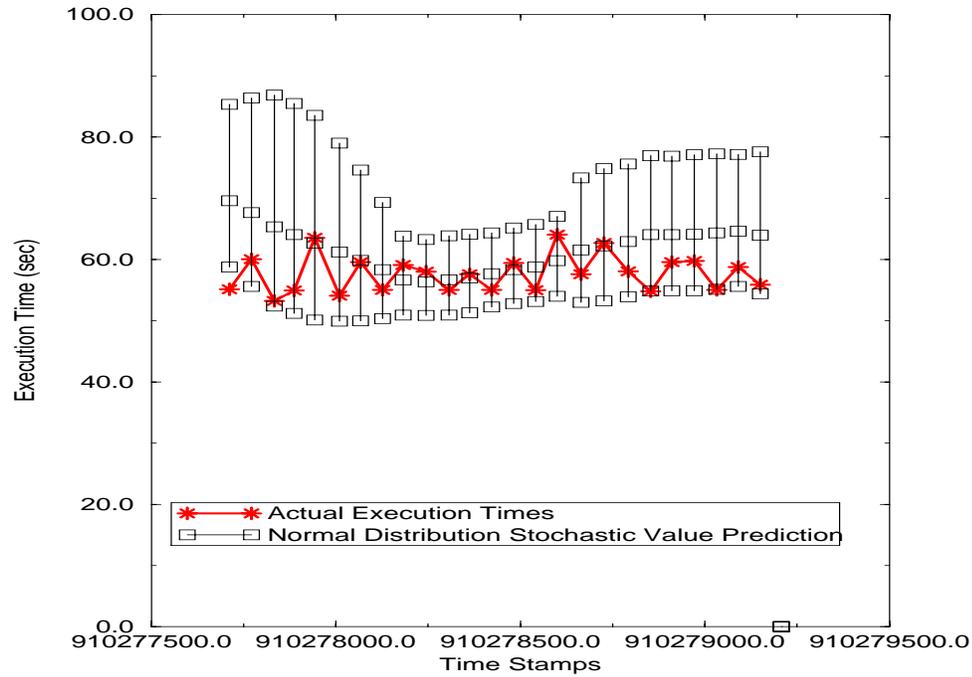


Figure 4.61: Comparison of point value prediction and stochastic value prediction represented by normal distributions for the Nbody benchmark on the PCL cluster with CPU loads shown in Figures 4.63 through 4.66.

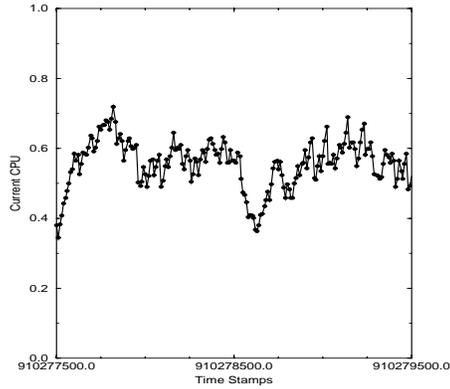


Figure 4.63: CPU values during runtime for prediction experiments depicted in Figure 4.61 and Figure 4.62 on Lorax, in the PCL cluster.

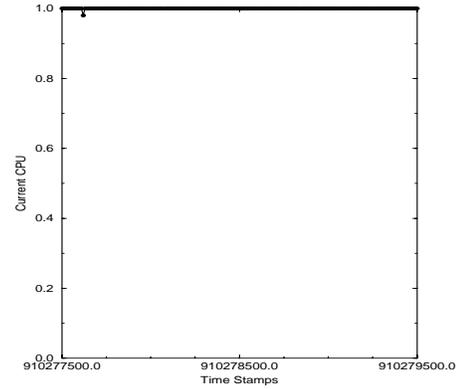


Figure 4.64: CPU values during runtime for prediction experiments depicted in Figure 4.61 and Figure 4.62 on Picard, in the PCL cluster.

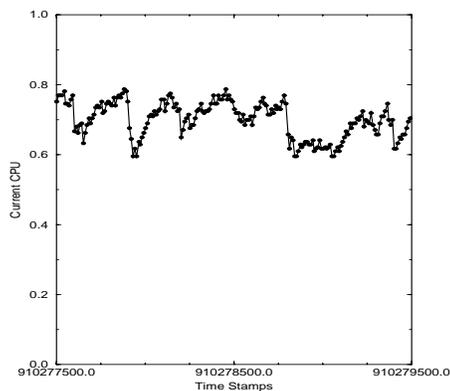


Figure 4.65: CPU values during runtime for prediction experiments depicted in Figure 4.61 and Figure 4.62 on Thing1, in the PCL cluster.

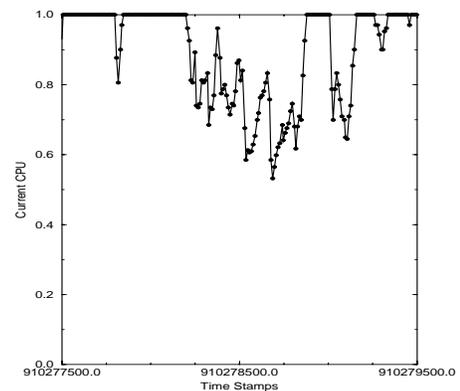


Figure 4.66: CPU values during runtime for prediction experiments depicted in Figure 4.61 and Figure 4.62 on Thing2, in the PCL cluster.

4.F.4 LU on PCL

In this of experiments we examined the LU code, presented in Section 2.E.2, on the PCL cluster. Figure 4.67 shows the normal distribution stochastic value predictions versus the actual time, Figure 4.68 shows the interval stochastic value predictions versus the actual time, run when the PCL cluster had CPU loads shown in Figures 4.69 through 4.72. The normal distribution representation captured 17 of 25 runs, while the interval representation captured 21s, but the normal distributions had significantly tighter ranges.

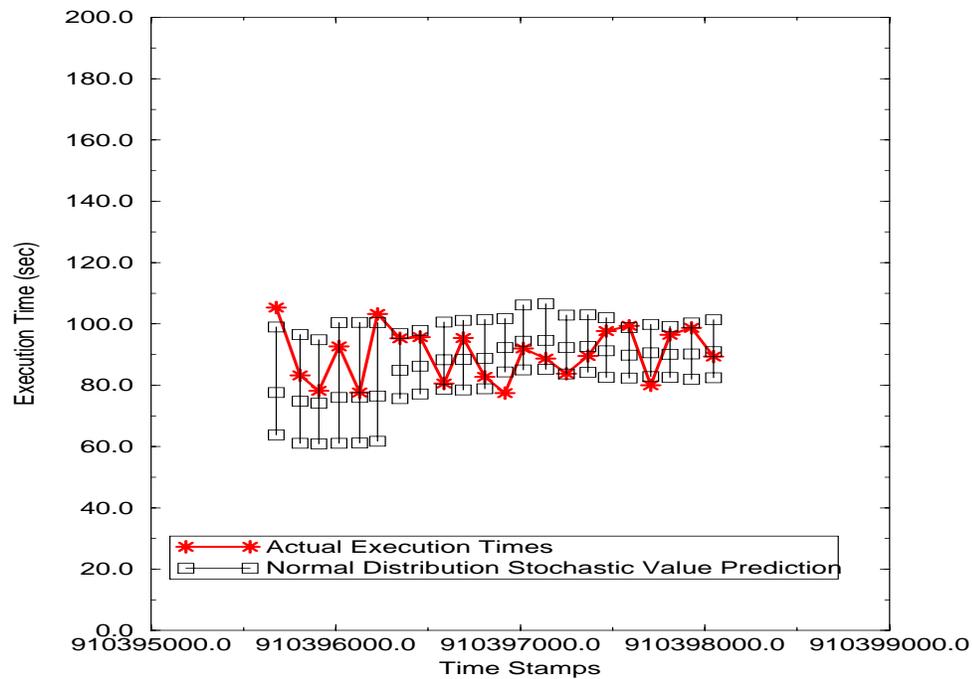


Figure 4.67: Comparison of point value prediction and stochastic value prediction represented by normal distributions for the LU benchmark on the PCL cluster with CPU loads shown in Figures 4.69 through 4.72.

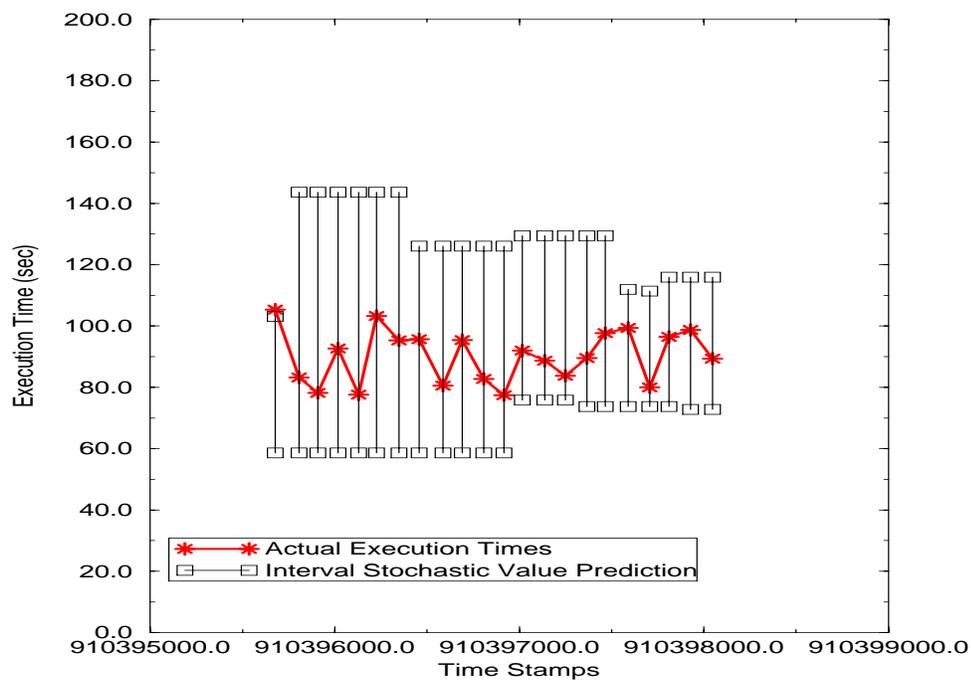


Figure 4.68: Comparison of point value prediction and stochastic value prediction represented by intervals for the LU benchmark on the PCL cluster with CPU loads shown in Figures 4.69 through 4.72.

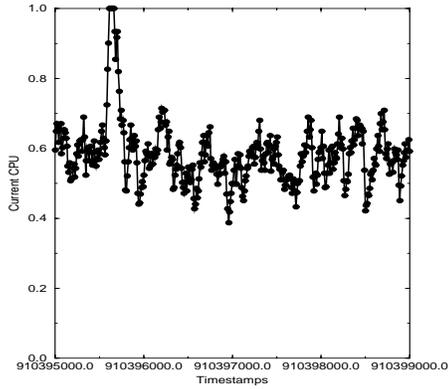


Figure 4.69: CPU values during runtime for prediction experiments depicted in Figure 4.67 and Figure 4.68 on Lorax, in the PCL cluster.

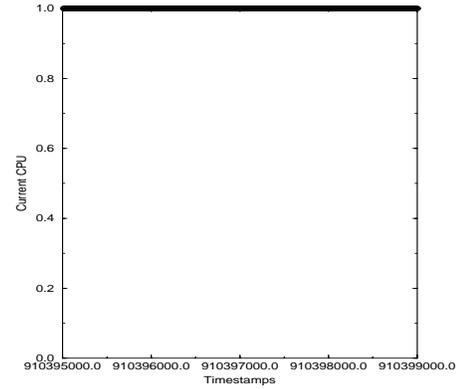


Figure 4.70: CPU values during runtime for prediction experiments depicted in Figure 4.67 and Figure 4.68 on Picard, in the PCL cluster.

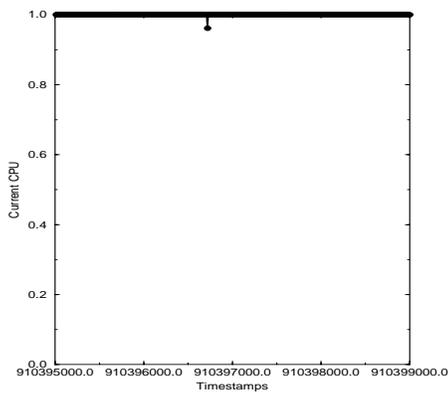


Figure 4.71: CPU values during runtime for prediction experiments depicted in Figure 4.67 and Figure 4.68 on Thing1, in the PCL cluster.

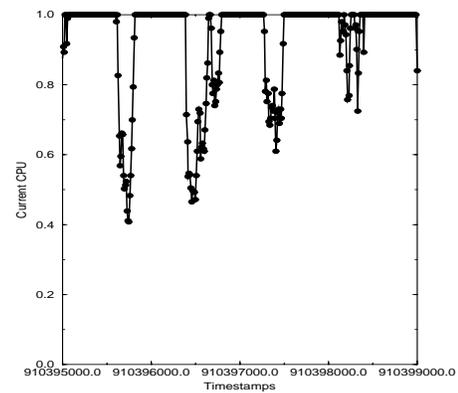


Figure 4.72: CPU values during runtime for prediction experiments depicted in Figure 4.67 and Figure 4.68 on Thing2, in the PCL cluster.

Figure 4.73 shows the normal distribution stochastic value predictions versus the actual time, Figure 4.74 shows the interval stochastic value predictions versus the actual time, run when the PCL cluster had CPU loads shown in Figures 4.75 through 4.78. The normal distribution representation captured 17 of 25 runs, while the interval representation captured 22.

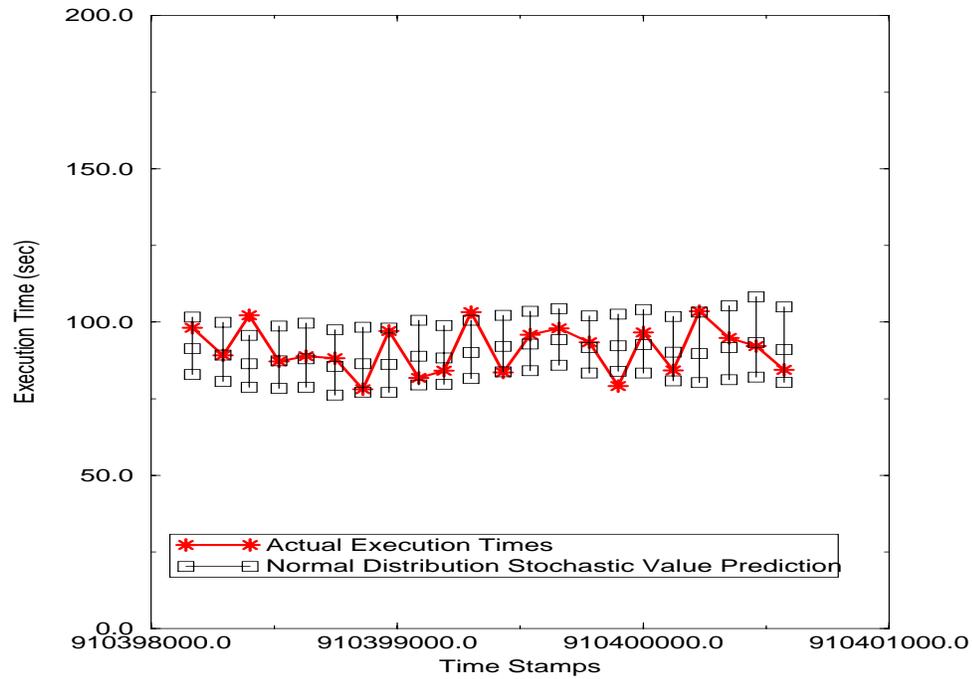


Figure 4.73: Comparison of point value prediction and stochastic value prediction represented by normal distributions for the LU benchmark on the PCL cluster with CPU loads shown in Figures 4.75 through 4.78.

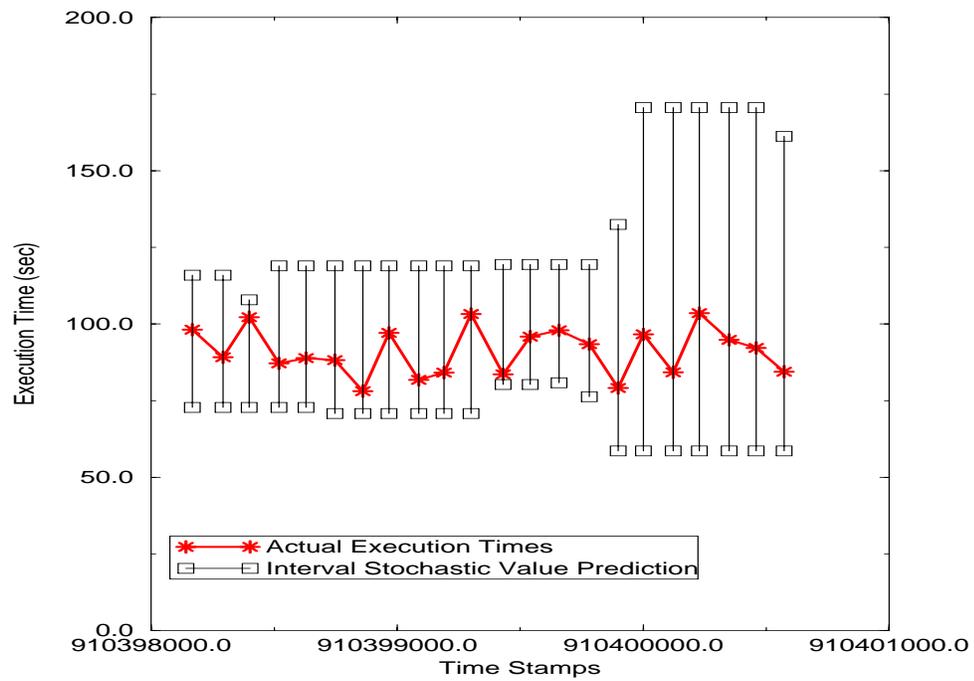


Figure 4.74: Comparison of point value prediction and stochastic value prediction represented by intervals for the LU benchmark on the PCL cluster with CPU loads shown in Figures 4.75 through 4.78.

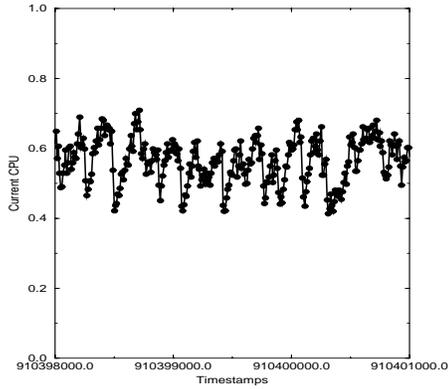


Figure 4.75: CPU values during runtime for prediction experiments depicted in Figure 4.73 and Figure 4.74 on Lorax, in the PCL cluster.

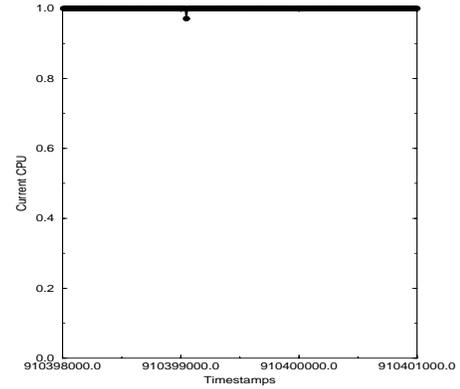


Figure 4.76: CPU values during runtime for prediction experiments depicted in Figure 4.73 and Figure 4.74 on Picard, in the PCL cluster.

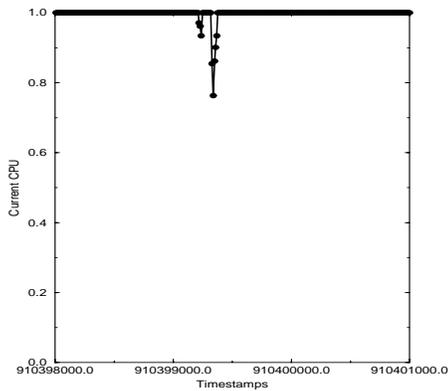


Figure 4.77: CPU values during runtime for prediction experiments depicted in Figure 4.73 and Figure 4.74 on Thing1, in the PCL cluster.

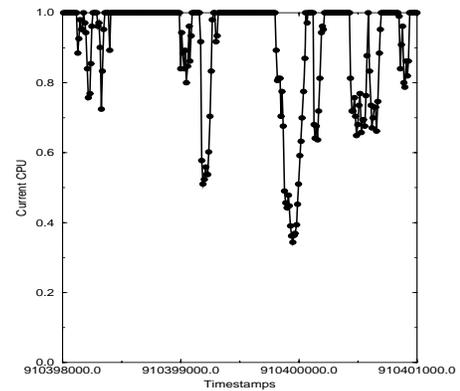


Figure 4.78: CPU values during runtime for prediction experiments depicted in Figure 4.73 and Figure 4.74 on Thing2, in the PCL cluster.

4.F.5 Summary

In summary, for the majority of the experiments, we achieved predictions that captures the majority of the execution behavior using either normal distribution representations or interval representations for the stochastic information. This was the case even when a point-valued prediction was off by 100%. We saw that predictions using normal distributions often resulted in sharper intervals, while predictions using interval representations captured more data. The right choice of a representation will be dependent on the use of the prediction.

4.G Related Work

There are several related approaches of note, however stochastic values as defined here are not related to Petri net models [MBC84, Mol90], also called “stochastic models”, in any way except through the application of conventional statistical techniques. Some researchers are beginning to use probabilistic techniques to represent data for predictions of application performance in distributed environments. Brasileiro et al. [BFM91] use probability distribution functions to calculate wait times on a token ring network. This work borrows heavily from queuing theory in a much more theoretical setting than our production setting. Formulas are provided to determine the suitability of an application for a given network based on queuing models, but no practical experiments are shown. In addition, Das’ group at Penn State has been studying the effect of assuming normal distributions in simulations [Das97]. In addition, the WARP project [SW96a, SW96b] models resource contention using queuing models.

The most closely related work to our approach is by Mendes as part of the Rice Fortran D95 compiler [MR98]. They generate symbolic performance predictions using a data parallel compiler. The compiler translates data parallel code and generates a symbolic cost model for program execution time, predicting the scalability of each fragment of the original code. The compiler generates upper

and lower bounds for predicted execution time by considering extrema of system-dependent constants (eg. memory references times) [MR98]. They extended the Fortran D95 compiler with appropriate functionality to extract information regarding the execution cost, in symbolic form, of all the loops and message passing activity in the translated program, as a function of the number of processors (P) and the problem size (N). To predict the program's performance one has simply to evaluate this cost model for specific values of P and N . This work, as we do, assumes some baseline system benchmarks, for example the time to send a message of size X , the time to receive a message of size X , the time to do an assignment and the time to do an arithmetic operation. However, some terms in the performance model may not be able to be determined at compile time, and values may change as the problem size scales. In particular, memory or cache accesses may become a factor. Mendes and colleagues allow the use of an estimated lower and upper bound for these values and build two models - one for the lower bound and one for the upper. The range between the two can be very large, and no attempt is made to minimize it. This work also stresses the need for models to be tied to the compiled code, not the source code, for analysis, much as we insist that implementations must be analyzed, not machine and application separately.

Another approach to using uncertainties in models is to build upon fuzzy logic [Kas96], where a fuzzy number is represented by a set of real numbers and an associated membership function. Autopilot [RSR97] couples a fuzzy logic rule base for distributed decision making with wide area performance sensors and policy control actuators. The rule base embodies common sense rules for resource management which are more amenable to fuzzy logic rules due to the conflicting goals involved and poorly understood optimization spaces involved. Lüthi also examined approaches for using fuzzy logic for network queuing models [LH97] by replacing single value parameters of open as well as closed queuing models by fuzzy numbers, and discusses asymptotic results for these models.

Karlin's work in competitive analysis using partial information also bears

relevance to our work [KMMO94, IKP96]. Competitive analysis is concerned with comparing the performance of on-line algorithms with that of optimal off-line algorithms. Often work in this area is faced with the concern of only partial information being available, and decisions being made appropriately in that environment.

Our work is somewhat related to “quality of information” such as timeliness and accuracy is used to parameterize business decisions in the area of Business Information Systems [Ver]. In addition, other quality of information efforts in the area of metacomputing [BWS97, KB97] have begun to address the need for additional information in order to make educated decisions about performance in a cluster environment with dynamic load. Using a stochastic value to represent the range of values possible from some system or application characteristic is one step in this direction.

4.H Summary

In this chapter, we describe a new approach to application performance prediction in multi-user (production) environments. We have defined stochastic values to reflect a likely range of behavior for model parameters, and extended the definition of structural models to allow for stochastic parameters as well as stochastic performance predictions. We allow alternative representations of the stochastic values as well. Our experiments demonstrate that in production settings, stochastic values can accurately identify the range of application execution behavior, providing more comprehensive and more accurate information about application execution than point values. The next chapter examines a sophisticated strategy that can take advantage of this stochastic information.

Some of the material in this chapter has been previously published [SB97a, SB97b]. The dissertation author was the principal researcher/author on these papers, and the co-author listed in these publications directed and supervised the research which forms the basis for this chapter.

Chapter 5

Stochastic Scheduling

In the previous chapters we discussed ways to model and predict the performance of applications executing on shared clusters of workstations. We developed compositional structural models, and extended them to use stochastic system and application information, as well as to provide stochastic execution time predictions.

In this chapter, we define a scheduling approach that can take advantage of stochastic predictions. In particular, we outline a strategy for defining and choosing among the set of schedules that can occur when using scheduling policies determined by stochastic information. We use this strategy to develop a stochastic time balancing scheduling policy for data parallel applications when stochastic information is represented by normal distributions. We present experimental results that indicate the promise of this approach, and discuss the generalization of this approach beyond the domain of our instantiation.

5.A Scheduling Overview

In this section, we provide a brief overview of terms that will be useful in defining our scheduling policy. An **application scheduler** is a process that allocates tasks to resources, allocates data to task/processor pairs, and orders tasks

in time on a particular resource or set of resources [Ber98]. Schedulers implement a **scheduling policy** (algorithm) that determines the **schedule** (assignment of tasks and data to resources) to be used.

Scheduling policies may use one of a variety of cost functions, or **performance measures**, to rank schedules and determine a “best” schedule. A common performance measure for application schedulers (and the one that we use) is **application execution time**. The goal of many scheduling policies is to minimize application execution time. Often, there are secondary performance measures of interest. For example, it can be highly desirable to reduce the variation of the execution times.

5.A.1 Motivation

To illustrate the need for stochastic scheduling we re-examine the example application from Chapter 1. The example system consists of two machines, A and B, executing an embarrassingly parallel application with 30 units of identical work to be completed. To schedule this application when the system is dedicated (i.e. no other users are present) it would be appropriate to use point-valued predictions for the time per unit in order to compute the work allocation. If Machine A completed one unit of work in 10 seconds, and Machine B completed one unit of work in 5 seconds, B should be assigned twice as much work as A, as shown in Table 5.1.

In a multi-user environment, contention for the processors and memories will cause the unit execution times to vary over time in a machine-dependent fashion. Using, for example, a 24-hour average, the machines may perform identically, taking 12 seconds per unit of work. If this were the only information available, we would balance the work evenly between the two machines.

It may be the case, though that one machine has a higher variation in behavior than the other, as shown in line 3 of Table 5.1. In the last chapter we demonstrated the usefulness of stochastic predictions over point-valued predictions. We would like our scheduling policy to take this behavior into account. For

example, if the application developer required the application to finish by a given time, we may choose to conservatively assign the work by assigning more work to the machine with a smaller variation in performance (Machine A).

	Machine A	Machine B	Assign A	Assign B
Single-user	10 sec	5 sec	10 units	20 units
Multi-user (point)	12 sec	12 sec	15 units	15 units
Multi-user (stochastic)	12 sec mean 0.3 sec sd	12 sec mean 1.8 sec sd	?	?

Table 5.1: Execution times for a unit of work in single-user and multi-user modes on two machines.

If the prediction is a stochastic value represented using a normal distribution, we can engineer the assignment of work to processors to be “conservative” by assigning work to processors using the mean and two standard deviations as the time per unit of work estimate. In the example setting, using a time per work estimate of 12.6 for Machine A and 15.6 for Machine B, and solving for

$$TimeOnA * DataOnA = TimeOnB * DataOnB \quad (5.1)$$

we would assign 17 units of work (rounded up from 16.59) for Machine A and 13 units for Machine B, as shown in Table 5.2. We call this a **95% conservative schedule** since it corresponds to a 95% confidence interval¹ for the completion time of a unit of work. Note that this does not necessarily equate to a 95% confidence interval for the completion of an application as a whole [Car98].

If there was a small penalty for poor predictions, for example if time was less constrained, the user might optimistically assign a greater portion of the work to the machine with the maximum potential performance, Machine B. Likewise, if the user had reason to believe that the previous distribution of work was too conservative, it might be desirable to assign more units of work to Machine B. For

¹A confidence interval is an interval of plausible values for a characteristic constructed so that the value of the characteristic will be captured inside the interval [DP96a].

example, the opposite of a 95% conservative schedule would be a **95% optimistic schedule**, as show in line 2 of Table 5.2, corresponding to a 5% conservative schedule, or a 5% confidence interval that the task on a processor will complete in a given time.

	Assign A	Assign B	Best time	Worst time
95% Conservative Prediction	17 units	13 units	148.2 sec	214.2 sec
95% Optimistic Prediction	13 units	17 units	152.1 sec	265.2 sec

Table 5.2: Conservative and optimistic scheduling assignments.

Of course, nothing limits the policies to only these two bounds. Other possible schedules are shown in Figure 5.1. The difficulty is how to determine the most performance-efficient schedule for a given application, platform, workload and user. That is the problem this chapter addresses.

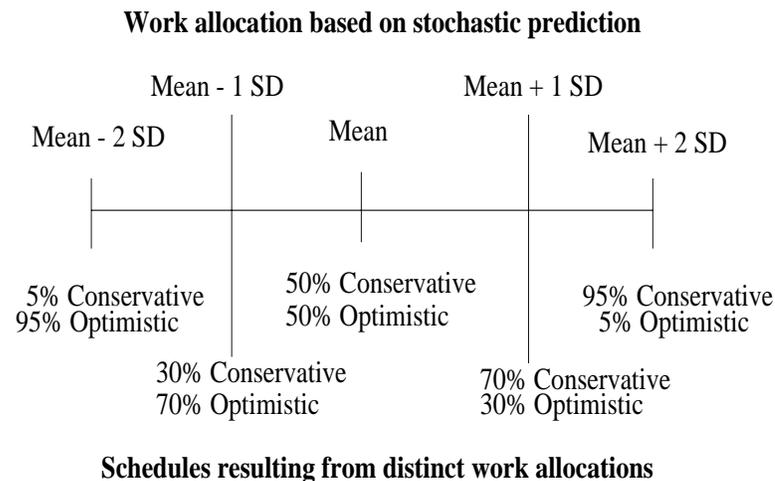


Figure 5.1: Possible work allocation schemes and associated scheduling policies.

5.A.2 Assumptions and Outline

For the purposes of this chapter, we assume that the target set of resources is a fixed set of shared heterogeneous workstations. We will consider the set of data

parallel applications presented previously in the Master-Slave and Regular SPMD classes of applications: SOR, GA and N-Body². Since the applications are data parallel, we also assume each processor executes the same task on its own data. Given these assumptions, scheduling simplifies to the data allocation problem: *How much data should be assigned to each task/processor pair, and how should that decision be made?*

Data allocation using point value information is a well-studied problem [HS91, ZLP96, Geh95, BK91, DMP97, HK97]. The goal of this chapter is to define a data allocation policy that can take advantage of stochastic information. We define a prototype scheduler that uses stochastic application execution time predictions to define a more sophisticated time balancing scheduling strategy than is possible with point-valued information. This scheduler adjusts the amount of data assigned to a task/processor pair in accordance with the variability of the system and the user's performance goals.

This chapter is organized as follows: Section 5.B reviews the time balancing scheduling policy and defines a stochastic extension, along with an algorithm for adjusting the stochastic schedule in accordance with the environment. Scheduling results using this initial approach are presented in Section 5.C. Related work is presented in 5.D, and we conclude in section 5.E.

5.B Stochastic Time Balancing

In this section we describe a strategy for stochastic time balancing. **Time balancing** is a common scheduling policy for data parallel applications that attempts to minimize the execution time of a data parallel application by assigning data so that each processor finishes executing at roughly the same time. This is generally accomplished by solving a set of equations, such as those given in Equa-

²We also assume that accurate (within a user-defined threshold) structural models and stochastic predictions are available to the scheduler, and that the stochastic information can be represented using normal distributions. See Section 3.A.4 for a further discussion of accuracy.

tion 5.2, to determine the data assignments ($\{ D_i \}$ in the following equations).

$$\begin{aligned} D_i u_i + C_i &= D_j u_j + C_j \quad \forall i, j \\ \sum D_i &= D_{Total} \end{aligned} \tag{5.2}$$

where

- u_i : Time to compute one unit of data on task/processor pair i .
- D_i : Amount of data assigned to task/processor pair i .
- D_{Total} : Total amount of data for the application.
- C_i : Time to distribute the data.

To solve these equations we assume that all variables are point-valued. We can assume that C_i is independent of the data assignment, as it is for the SOR application. If the communication time is a function of the amount of data assigned to a processor, such as it is for the GA, this value can be added in to the u_i factor. A good example of developing and solving time balancing equations to solve for a performance efficient data allocation is shown in work by the AppLeS Group [BWF⁺96].

5.B.1 Using Stochastic Information

One way to extend time balancing is to adjust the u_i value to alter the predicted completion time on task/processor pair i . Instead of balancing the time using the result of the mean of a stochastic prediction (or the result of a point-valued prediction) of execution time for a task/processor pair, we can use any (point) value in the range given by a stochastic prediction. This would allow the flexibility to choose a larger u_i , resulting in an assignment of less data to possibly mitigate the effects of variability on the overall application behavior (by assigning less data to a high variability machine). One approach to determine which value to choose for u_i is to add a specified number of standard deviations to the mean

of a stochastic prediction. Since the standard deviation is a value specific to the performance of a given machine, each task/processor pair would have its data allocation adjusted in a machine-dependent manner. We call the multiplier that determines the number of standard deviations to add to (or subtract from) the mean the **Tuning Factor**. The Tuning Factor can be defined in a variety of ways, and will determine the percentage of “conservatism” of the scheduling policy.

With this in mind, we can define u_i as

$$u_i = m_i + sd_i * TF \quad (5.3)$$

where

m_i : The *mean* of the predicted completion time for task/processor pair i .

sd_i : The *standard deviation* of the predicted completion time for task/processor pair i .

TF : The *Tuning Factor*, used to determine the number of standard deviations to add to (or subtract from) the mean value to determine how conservative the data allocation should be.

Given the definition of u_i in Equation 5.3, the set of equations to solve over is now:

$$\begin{aligned} D_i(m_i + TF * sd_i) + C_i &= D_j(m_j + TF * sd_j) + C_j \quad \forall i, j \\ \sum D_i &= D_{Total} \end{aligned} \quad (5.4)$$

5.B.2 The Tuning Factor

The Tuning Factor represents the variability of the system as a whole, defined by the amount of variation in the available CPU, the variation in the available bandwidth between each pair of processors, a measure of nondeterminism of the application, etc. The Tuning Factor is a key element in defining a stochastic time balancing policy. It provides the “knob” to turn to make the scheduling policy more or less conservative. The Tuning Factor can be provided by the user or

calculated by the scheduler. The challenge lies in choosing or calculating a Tuning Factor that promotes a performance-efficient schedule for a given application, environment, and user.

Our computational method for determining the Tuning Factor uses a system of benefits and penalties calculated using stochastic prediction information. This benefit and penalty approach to determining the value of the Tuning Factor draws from the scheduling approach of Sih and Lee [SL90b] where the amount of work is increased for a processor with desirable characteristics (such as a light load, fast processor speed, etc.), considered as **benefits**, and decreased when a machine has undesirable properties (such as poor network connectivity, small memory capacity, etc.), considered as **penalties**. The basic idea is to assign more optimistic scheduling policies to platforms with a smaller variability in performance.

We use three values that can provide information about the variability of the system as a whole: the *number* of machines that exhibit fluctuating performance behavior in the platform; the *variability* of performance exhibited of each machine, represented as the standard deviation, *sd*; and a measure of the available computing capacity of the machines, which we call *power* below. We want to *benefit* (give a more optimistic schedule to) the platforms that have a smaller variability associated with them. Specifically, we want to benefit:

Platforms with fewer varying machines. If only one machine on a platform has varying behavior, the platform should have a more optimistic schedule than another platform that has the majority of the machines exhibiting variable behavior.

Low variability machines, especially those with lower powers. We want to benefit a platform more for having a slow machine with a high variability than having a fast machine with a high variability. The faster machine may have more data assigned to it, so the high variability is likely to have a greater impact on the overall application execution time.

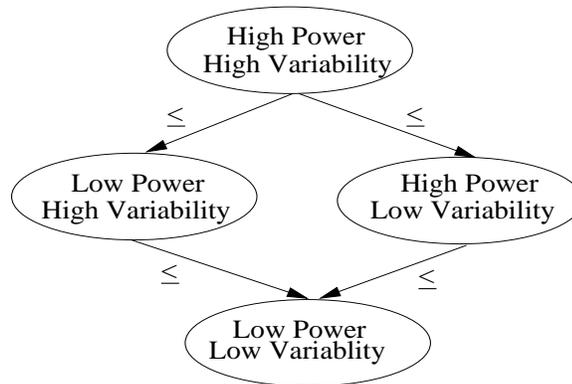


Figure 5.2: Diagram depicting the relative values of Tuning Factors for different configurations of power and variability.

We identify the partial ordering in Figure 5.2 for this requirement. In Figure 5.3 we define an algorithm to calculate the Tuning Factor for stochastic values represented as normal distributions and obeys the partial ordering in Figure 5.2. Note that many other functions satisfy the relative ranking in Figure 5.2 and are also viable.

The algorithm given in Figure 5.3 requires the definition of a number of parameters that are specific to the execution environment:

Power(TP_i): A measure of the communication and computation power of the machine with respect to the resource requirements of the application. This can be determined by an application-specific benchmark.

HighPower: A value that identifies machines with “high” power. For our initial experiments, we calculated an average power for the machines in the platform, and any machine with a greater value for power than the average was considered a High Power machine.

Variability(TP_i): A measure of the variability of performance delivered by a machine. For our experiments we set Variability(TP_i) equal to the standard deviation of the available CPU measurements for processor i .

```

For each task/processor pair, TP_i
  if Power (TP_i) > HighPower
    if Variability(TP_i) > HighVariability
      Value_i = MaxTuningFactor
    else
      Value_i = MidTuningFactor
  else
    if Variability(TP_i) > HighVariability
      Value_i = MidTuningFactor
    else
      Value_i = MinTuningFactor

TuningFactor =  $\frac{\sum \text{Value } i}{\text{number of machines}}$ 

```

Figure 5.3: Algorithm to Compute Tuning Factor.

HighVariability: A value that identifies when a machine has a high variability. There are many ways to chose this parameter. For our experiments, we defined a HighVariability to be a standard deviation for the available CPU measurements greater than one-quarter of the width of the average mode from previous experimental data. In particular, for our experiments, HighVariability = 0.07 for the available CPU, since this was roughly equivalent to having modes that were 0.3 wide. This metric should be adjusted to the environment and can be defined by the user.

Tuning Factor Range: The Tuning Factor Range defines the range of values for the Tuning Factor. For the algorithm defined in Figure 5.3, three values within the Tuning Factor Range must be defined: *MinTuningFactor*, *MidTuningFactor*, and *MaxTuningFactor*. These values can be used to determine the percentage of conservatism for the schedule that impacts the data allocation.

In this initial work, we focused on stochastic values represented using normal

distributions and consequently focused in particular on predictions at the mean, the mean plus one standard deviation, and the mean plus two standard deviations. Therefore, we set $MinTuningFactor = 0$, $MidTuningFactor = 1$, and $MaxTuningFactor = 2$.

Note that using the algorithm given in Figure 5.3, schedules range only from a 50% to a 95% conservative schedule as illustrated in Figure 5.1. This is because on a single-user (dedicated) platform, we expect mean values to achieve the best performance. Adding variability to the system (by adding other users) typically impacts performance in a negative way, leading to a more conservative schedule to mitigate the effect of the variability. However, one feature of the stochastic scheduling approach is that the scheduling policy can be adjusted to reflect the socio-political factors (promoting a more or less conservative schedule) that may constrict the application of a scheduling policy in a production environment [LG97].

Several other factors need to be considered when defining a Tuning Factor or evaluating the system variability. The time frame over which the variability information is gathered must be defined a priori, and can impact the data greatly. For our experiments, we used a time frame, T , for the stochastic predictions as defined in Section 4.B.1. See this section for a further discussion of time.

Note that the algorithm presented in Figure 5.3 is used to compute the Tuning Factor, this parameter then has the property that a value close to the threshold and far from threshold are treated as equivalent. It is possible to analyze the partial ordering in such a way that the values are continuous given additional information, thus negating the need for the discrete Tuning Factor Range values and this unfortunate side effect. This is a topic for future work.

Our first attempt at defining a Tuning Factor does not include information pertaining to network performance. Future work includes adding this factor and analyzing the needed partial ordering as well. Other factors such as memory size, available disk space, etc. could also be considered.

Note that the approach discussed here is one of many possible ways to define the Tuning Factor, the heart of our scheduling heuristic. Other choices are possible, and we plan to explore additional approaches that may better suit other particular environments and applications. Finally, similar extensions could be evaluated for other scheduling policies than time balancing as well.

5.C Stochastic Scheduling Experiments

In this section we compare a time balancing scheduling policy parameterized by a mean-valued u_i (i.e. when the Tuning Factor is always 0) with two time balancing policies where u_i is determined by a non-zero Tuning Factor. We evaluate two scheduling policies that use non-zero Tuning Factors. The first fixes the Tuning Factor to a 95% conservative schedule (where two standard deviations are added to the mean for the u_i value for every run), called **95TF**. The second determines a Tuning Factor at run-time based on environmental data calculated using the system of benefits and penalties described previously by the algorithm given in Figure 5.3, and is called the Variable Tuning Factor, or **VTF**. We compare both policies that use stochastic prediction information to a scheduling policy that uses only a point-valued prediction, called **Mean**. *Our goal is to experimentally determine in which situations it was advantageous to use non-zero Tuning Factors.*

All of the scheduling policies in our experiments use structural performance prediction models parameterized by run-time information provided by the Network Weather Service. In the experiments, we ran applications using the data distribution determined by each policy at runtime and compared their execution times. In order to compare the policies fairly we alternate scheduling policies for a single problem size of the application, so any two adjacent runs could experience similar workloads and variation in environment. (Each run was short, between 30 and 90 seconds.)

The experiments were run on the PCL cluster, consisting of 4 heteroge-

neous Sparc workstations connected by a mixture of slow and fast ethernet, and the Linux Cluster, consisting of 4 PC's running Linux connected by fast ethernet. Both platforms were shared among other users at the time of the experiments. We show CPU traces to illustrate the load conditions during the scheduling experiments.

Three applications were used in the experiments: the Regular SPMD SOR code (described in Section 2.E.1), and two data parallel Master-Slave applications: a Genetic Algorithm (GA) described in Section 2.D.1, and an N-Body Code, described in Section 2.D.3.

5.C.1 Performance Metrics

We consider a scheduling policy to be “better” than others if it exhibits the lowest execution time on a given run. Note that for our experiments, the “best” scheduling policy varies over time and with different load conditions. To compare these policies, we define two summary statistics. The first, called **Window**, is the proportion of runs by a scheduling policy that has the minimal execution time compared to the policy run before it and the policy run after it, or a window of three runs. For example, given the first few run times for the experiment in Figure 5.4, shown in Table 5.3, in the first set of three (Mean at 2623, VTF at 2689 and 95TF at 2758) the minimal time is achieved using the VTF policy. For the next set of three (VTF at 2689, 95TF at 2758 and Mean at 2821) VTF again achieved the minimal execution time, etc.

Using this metric for the runs shown in Figure 5.4, the lowest execution time was achieved by the Mean 9 times of 58 windows, for VTF 27 times of 58 windows and for 95TF 22 times of 58 windows, indicating that the VTF policy was almost three times as likely to achieve a minimal execution time than Mean.

The second summary statistic, which we call **Compare**, evaluates how often each run achieves a minimal execution time. There are three possibilities: it can have a *better* execution time than both the run before and the run after, it can have a *worse* execution time than both, or it can be *mixed* – better than one,

Time stamp	Execution Time	Policy
2623	38.534076	Mean
2689	32.802351	VTF
2758	34.633066	95TF
2821	33.985560	Mean
2886	33.982391	VTF
2951	34.711265	95TF
3023	38.724127	Mean
3087	33.523844	VTF
3152	33.553239	95TF
3221	37.223270	Mean

Table 5.3: First 10 execution times for experiments pictured in Figure 5.4.

and worse than the other. Referring to the execution times given in Table 5.3, for VTF at 2689, it achieves a better execution time than both the Mean run before it or the 95TF run after it; the 95TF run at 2758 does worse than either the VTF run before it or the Mean run after it, the Mean run at 2821 is mixed, it does better than the 95TF run before it, but worse than the VTF run after it. These results are given in Table 5.4 and indicate that VTF is three times as likely to have a faster execution time than the runs before or after it than the mean, and one-fourth as likely to be worse than both than the Mean.

Policy	Better	Mixed	Worse
Mean	3	4	12
VTF	10	7	3
95TF	6	9	4

Table 5.4: Summary statistics using Compare evaluation for experiment pictured in Figure 5.4.

The third metric we use is a mean and standard deviation for the set of runtimes as a whole, for each policy. For the data given in Figure 5.4, the Mean policy runs had a mean of 32.87 and a standard deviation of 3.780, the VTF policy

runs had a mean 30.91 and a standard deviation of 3.34, and for the 95TF policy runs had a mean 30.66 and a standard deviation of 2.75. This indicates that over the entire run, the VTF policy exhibited a 5-8% improvement in overall execution time, and less variation in execution time than the Mean policy as well.

5.C.2 SOR on Linux Cluster

In our first set of experiments we examined the SOR code, presented in Section 2.E.1, on the Linux cluster. Figure 5.4 shows a comparison of the three scheduling policies when the Linux cluster had CPU loads shown in Figures 5.5 through 5.8. In this set of experiments, the load on two machines is fairly constant, while the load on a third has a small variation, and the available CPU on a fourth has a very large variation. Of note especially is the dramatic drop in run times at time stamp 910005000, likely due to the cessation of a job on Soleil. The metrics for this graph are given in Section 5.C.1 and indicate a significant improvement when using stochastic information.

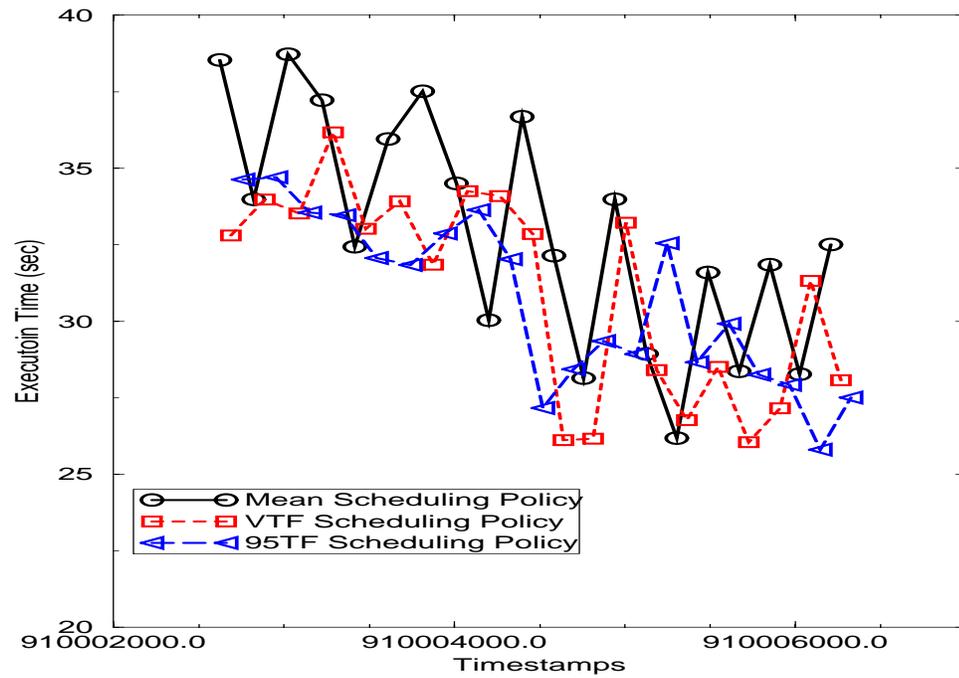


Figure 5.4: Comparison of **Mean**, **VTF**, and **95TF** policies, for the SOR benchmark on the Linux cluster with CPU loads shown in Figures 5.5 through 5.8.

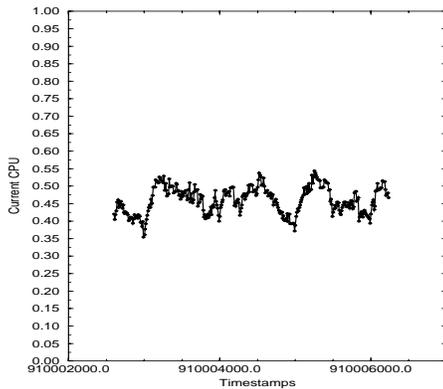


Figure 5.5: CPU values during runtime for scheduling experiments depicted in Figure 5.4 on Mystere, in the Linux cluster.

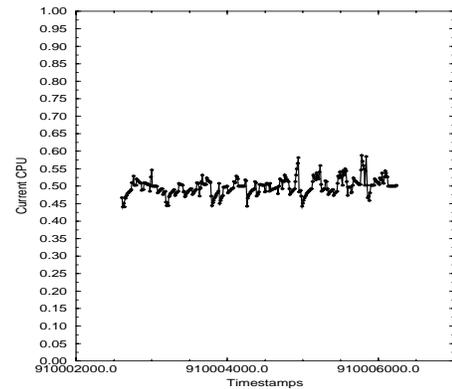


Figure 5.6: CPU values during runtime for scheduling experiments depicted in Figure 5.4 on Quidam, in the Linux cluster.

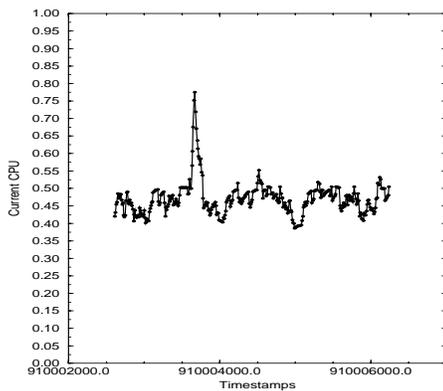


Figure 5.7: CPU values during runtime for scheduling experiments depicted in Figure 5.4 on Saltimbanco, in the Linux cluster.

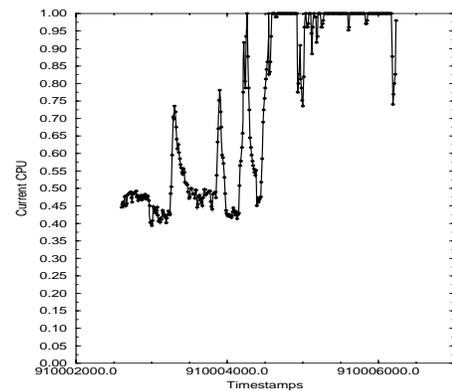


Figure 5.8: CPU values during runtime for scheduling experiments depicted in Figure 5.4 on Soleil, in the Linux cluster.

SOR on Linux, Constant Available CPU

For this set of experiments, there was a fairly constant load over the four Linux machines, as shown in Figures 5.10 through 5.13. Figure 5.9 shows a comparison of the three scheduling policies. Using the Window metric, the lowest execution time was achieved by the Mean 24 times of 58 windows, 24 times for VTF and 10 times for 95TF. The Mean policy runs had a mean of 34.82 and a standard deviation of 3.63, the VTF policy runs had a mean 34.52 and a standard deviation of 1.22, and for the 95TF policy runs had a mean 35.23 and a standard deviation of 1.58. The results of the Compare metric are given in Table 5.5.

Policy	Better	Mixed	Worse
Mean	8	6	5
VTF	8	7	5
95TF	3	6	10

Table 5.5: Summary statistics using Compare evaluation for experiment pictured in Figure 5.9.

For this set of experiments, the augmented scheduling policies did not overly benefit the performance. We feel this is because of the low variation seen on the system. However, it should be noted that even if the overall execution time did not improve significantly, the variation of the execution time was reduced by half.

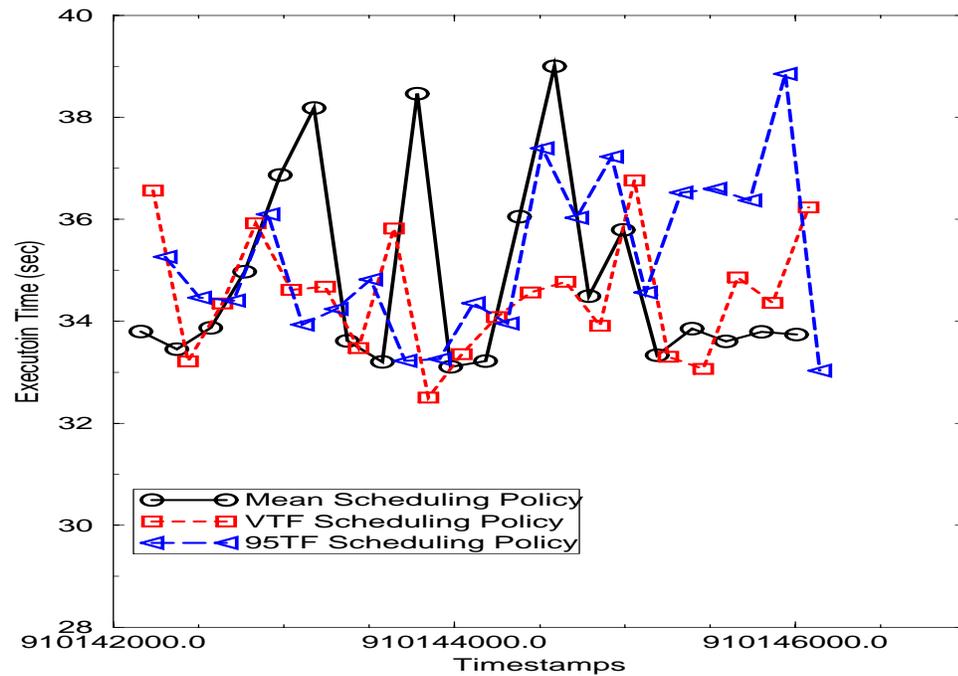


Figure 5.9: Comparison of **Mean**, **VTF**, and **95TF** policies, for the SOR benchmark on the Linux cluster with CPU loads shown in Figures 5.10 through 5.13.

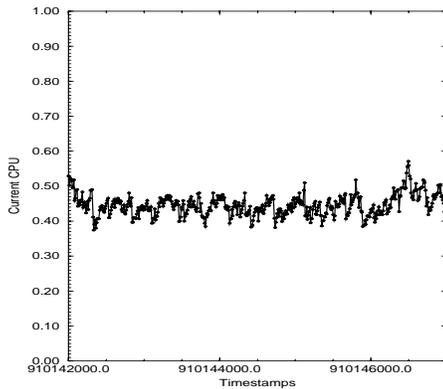


Figure 5.10: CPU values during runtime for scheduling experiments depicted in Figure 5.9 on Mystere, in the Linux cluster.

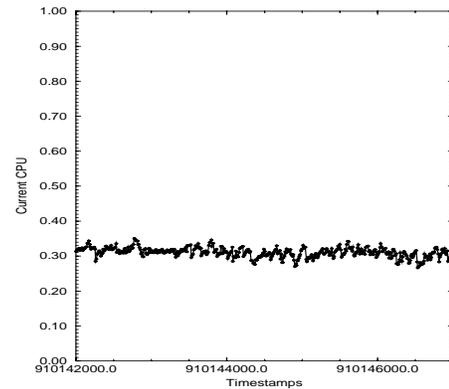


Figure 5.11: CPU values during runtime for scheduling experiments depicted in Figure 5.9 on Quidam, in the Linux cluster.

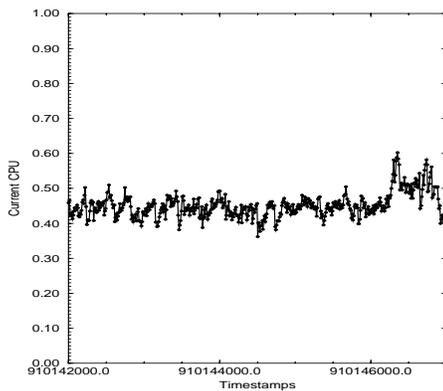


Figure 5.12: CPU values during runtime for scheduling experiments depicted in Figure 5.9 on Saltimbanco, in the Linux cluster.

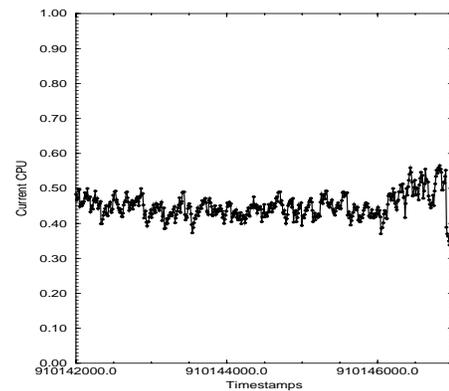


Figure 5.13: CPU values during runtime for scheduling experiments depicted in Figure 5.9 on Soleil, in the Linux cluster.

SOR on Linux, High Variation in Available CPU

Figure 5.14 shows a comparison of the three scheduling policies when the Linux cluster had CPU loads shown in Figures 5.15 through 5.18. Using the Window metric, the lowest execution time was achieved by the Mean 17 times of 58 windows, 17 times for VTF and 24 times for 95TF. The Mean policy runs had a mean of 35.4 and a standard deviation of 4.57, the VTF policy runs had a mean 35.29 and a standard deviation of 3.5, and for the 95TF policy runs had a mean 35.49 and a standard deviation of 3.83. The results of the Compare metric are given in Table 5.6.

Policy	Better	Mixed	Worse
Mean	4	9	6
VTF	3	11	6
95TF	8	7	4

Table 5.6: Summary statistics using Compare evaluation for experiment pictured in Figure 5.14.

The summary statistics indicate that over the entire run, although there wasn't a great deal of execution time improvement, there was less variation in the performance with the stochastic scheduling policies. Moreover, both the Compare and Window metrics indicate that for similar load conditions, 95TF was almost twice as likely to achieve a minimal execution time than the Mean scheduling policy.

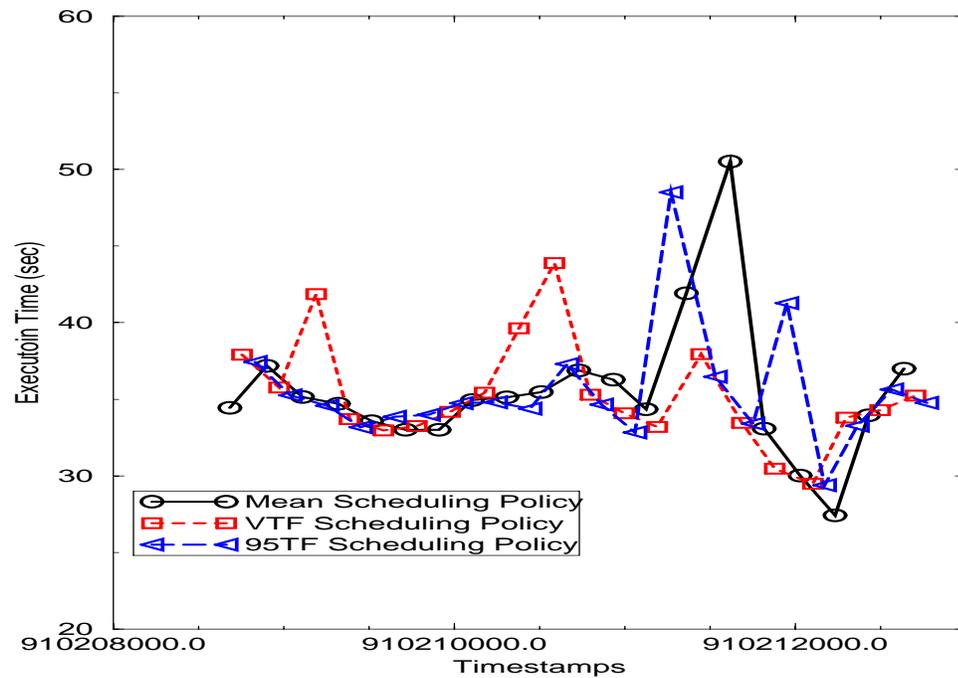


Figure 5.14: Comparison of **Mean**, **VTF**, and **95TF** policies, for the SOR benchmark on the Linux cluster with CPU loads shown in Figures 5.15 through 5.18.

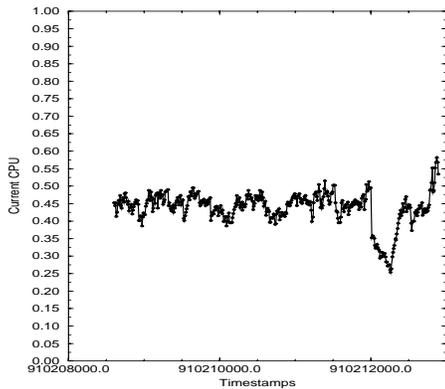


Figure 5.15: CPU values during runtime for scheduling experiments depicted in Figure 5.14 on Mystere, in the Linux cluster.

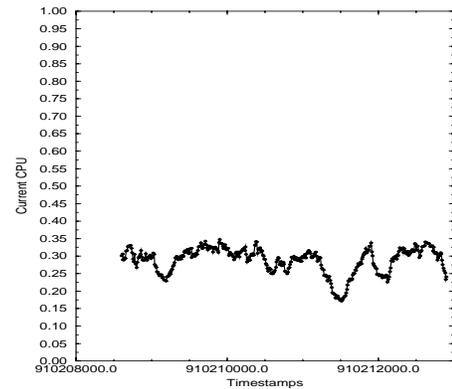


Figure 5.16: CPU values during runtime for scheduling experiments depicted in Figure 5.14 on Quidam, in the Linux cluster.

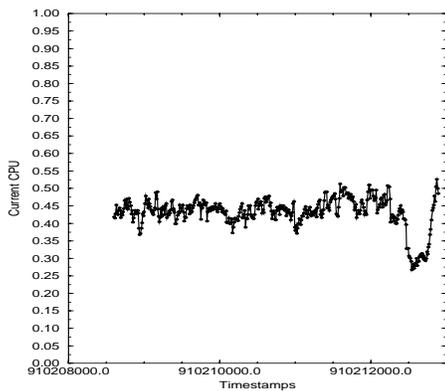


Figure 5.17: CPU values during runtime for scheduling experiments depicted in Figure 5.14 on Saltimbanco, in the Linux cluster.

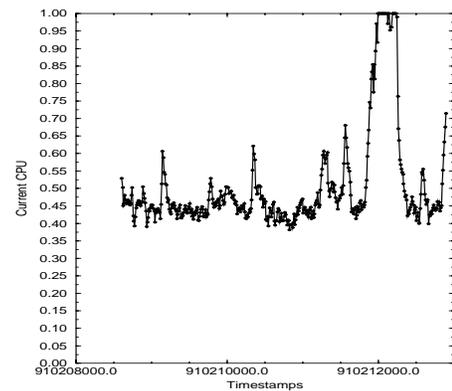


Figure 5.18: CPU values during runtime for scheduling experiments depicted in Figure 5.14 on Soleil, in the Linux cluster.

5.C.3 SOR on PCL Cluster

We ran similar experiments for the SOR code on the PCL cluster. Again, we alternated scheduling policies to compare them fairly. As with the Linux cluster we ran these experiments under a variety of load conditions.

Figure 5.19 shows a comparison of the three scheduling policies when the PCL cluster had CPU loads shown in Figures 5.20 through 5.23. Two of the machines had a very low variability, while two had a high variability in available CPU. Using the Window metric, the lowest execution time was achieved by the Mean 8 times of 58 windows, 39 times for VTF and 11 times for 95TF. The Mean policy runs had a mean of 24.74 and a standard deviation of 2.72, the VTF policy runs had a mean 22.15 and a standard deviation of 2.98, and for the 95TF policy runs had a mean 24.48 and a standard deviation of 2.57. The results of the Compare metric are given in Table 5.7.

Policy	Better	Mixed	Worse
Mean	3	7	9
VTF	15	2	3
95TF	3	8	8

Table 5.7: Summary statistics using Compare evaluation for experiment pictured in Figure 5.19.

These summary statistics indicate that over the entire run, there was a 10% improvement when using VTF as compared to the Mean scheduling policy. Moreover, VTF was almost 5 times as likely to achieve a minimal execution time than Mean in a run-by-run comparison.

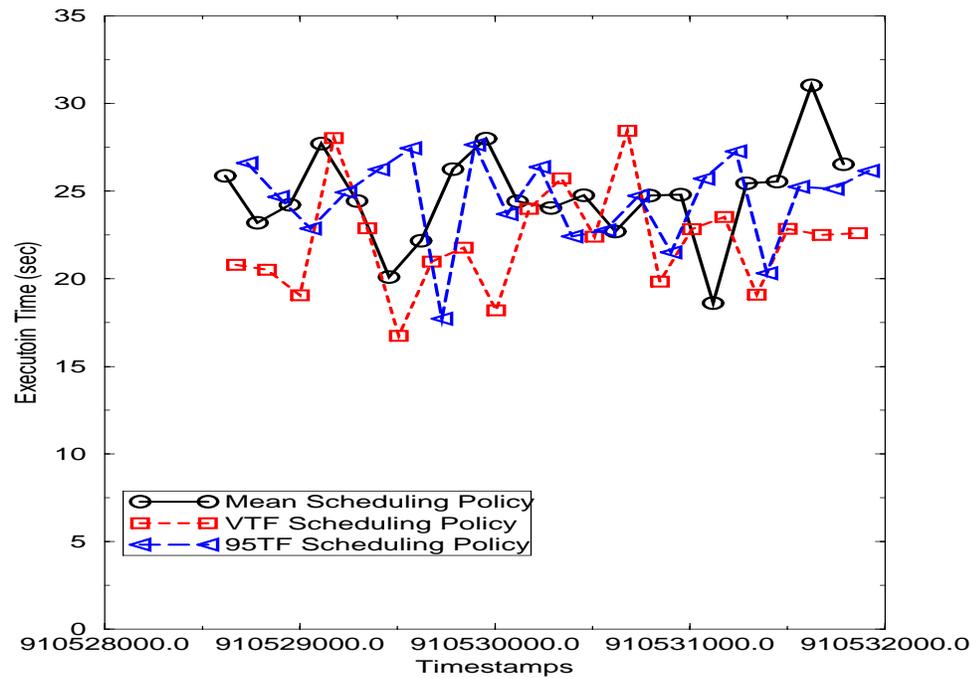


Figure 5.19: Comparison of **Mean**, **VTF**, and **95TF** policies, for the SOR benchmark on the Linux cluster with CPU loads shown in Figures 5.20 through 5.23.

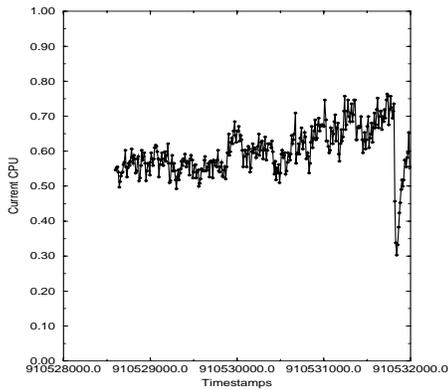


Figure 5.20: CPU values during runtime for scheduling experiments depicted in Figure 5.19 on Lorax, in the PCL cluster.

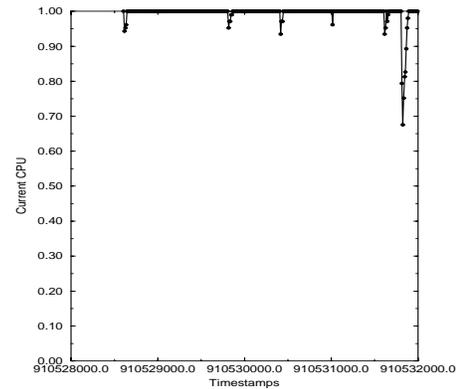


Figure 5.21: CPU values during runtime for scheduling experiments depicted in Figure 5.19 on Picard, in the PCL cluster.

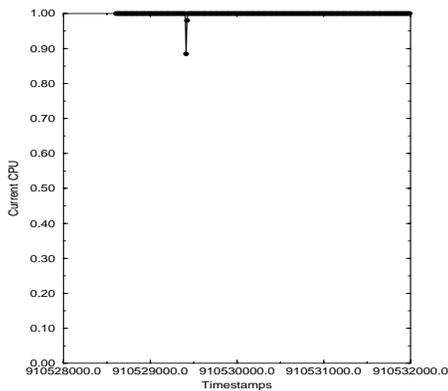


Figure 5.22: CPU values during runtime for scheduling experiments depicted in Figure 5.19 on Thing1, in the PCL cluster.

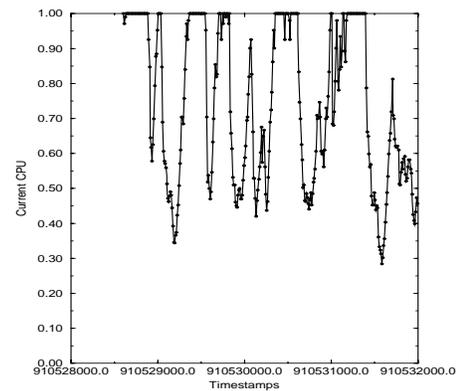


Figure 5.23: CPU values during runtime for scheduling experiments depicted in Figure 5.19 on Thing2, in the PCL cluster.

SOR on PCL, High Variability in Available CPU

Figure 5.24 shows a comparison of the three scheduling policies when the PCL cluster had CPU loads shown in Figures 5.25 through 5.28, in which three machines had a high variability. Using the Window metric, the lowest execution time was achieved by the Mean 25 times of 58 windows, 14 times for VTF and 19 times for 95TF. The Mean policy runs had a mean of 23.31 and a standard deviation of 4.35, the VTF policy runs had a mean 22.78 and a standard deviation of 2.74, and for the 95TF policy runs had a mean 22.31 and a standard deviation of 2.75. The results of the Compare metric are given in Table 5.8.

Policy	Better	Mixed	Worse
Mean	8	4	7
VTF	5	8	7
95TF	6	8	5

Table 5.8: Summary statistics using Compare evaluation for experiment pictured in Figure 5.24.

These results are inconclusive. The execution times for the stochastic scheduling policies show some improvement over the point-valued Mean policy, and a much smaller standard deviation in execution time, but the Window and Compare metrics indicate that on a run-by-run level, the improvement was negligible.

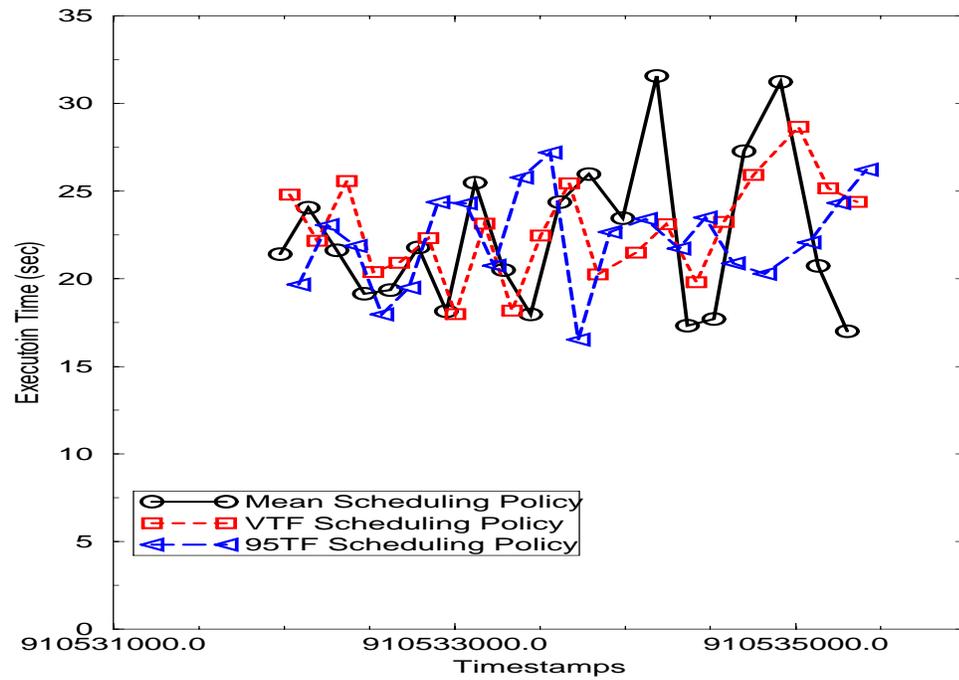


Figure 5.24: Comparison of **Mean**, **VTF**, and **95TF** policies, for the SOR benchmark on the Linux cluster with CPU loads shown in Figures 5.25 through 5.28.

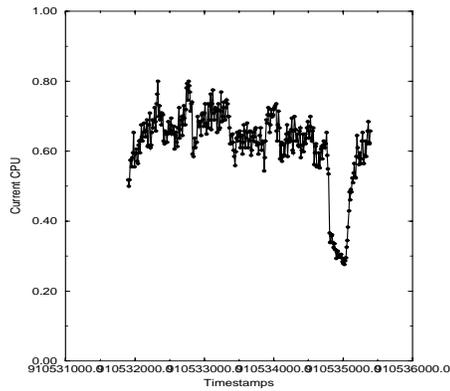


Figure 5.25: CPU values during runtime for scheduling experiments depicted in Figure 5.24 on Lorax, in the PCL cluster.

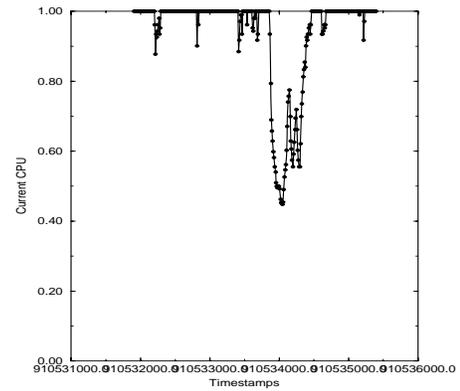


Figure 5.26: CPU values during runtime for scheduling experiments depicted in Figure 5.24 on Picard, in the PCL cluster.

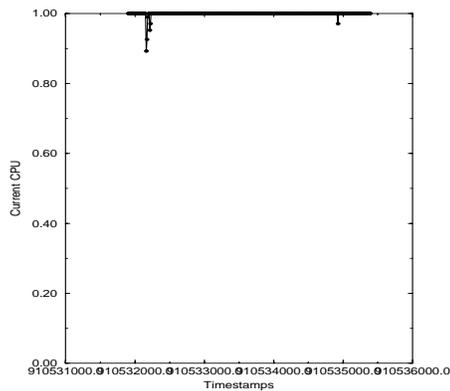


Figure 5.27: CPU values during runtime for scheduling experiments depicted in Figure 5.24 on Thing1, in the PCL cluster.

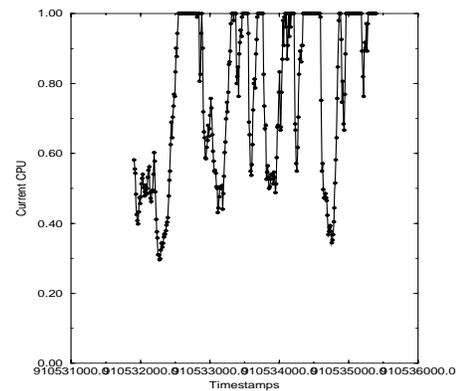


Figure 5.28: CPU values during runtime for scheduling experiments depicted in Figure 5.24 on Thing2, in the PCL cluster.

5.C.4 GA on Linux Cluster

The third group of experiments use the Genetic Algorithm code on the Linux cluster. The GA has the added complication of nondeterministic behavior, making comparisons even more difficult since not only the loads are changing but the actual work being done by the application changes as well. This is shown by Figure 5.29 which shows the run times for the GA on the dedicated Linux cluster when the data was distributed evenly among the four processors, consistently from run to run. For these runs, the mean was 135.48 with a standard deviation of 5.04.

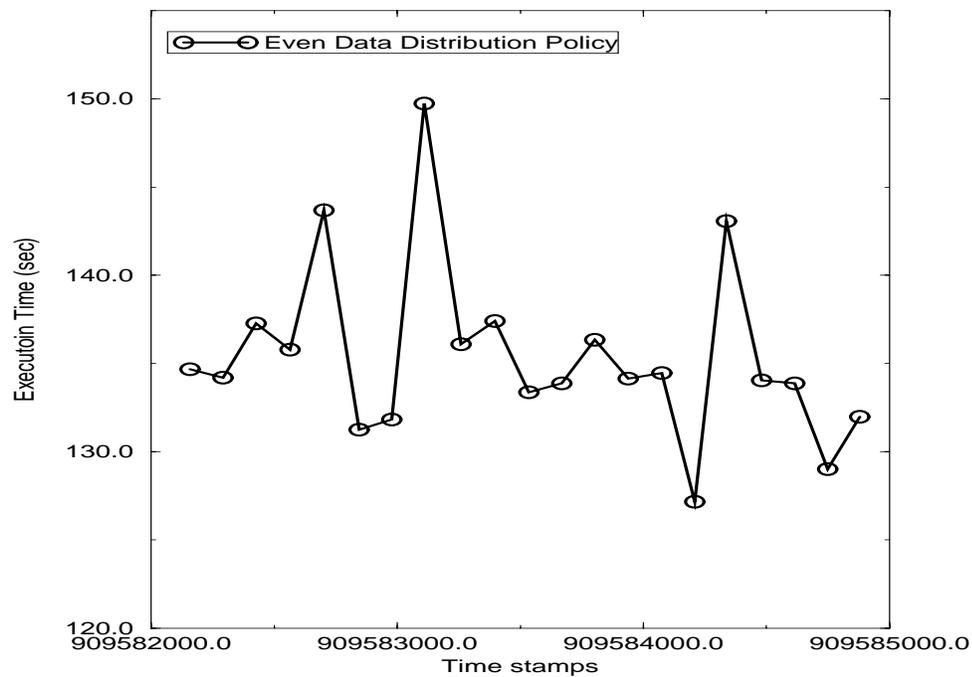


Figure 5.29: Run times for GA application with even data decomposition on Linux cluster.

Figure 5.30 shows a comparison of the three scheduling policies when the Linux cluster had CPU loads shown in Figures 5.31 through 5.34, showing high, but consistent variability for two of the machines. Using the Window metric, the lowest execution time was achieved by the Mean 19 times of 58 windows, 24 times for VTF and 15 times for 95TF. The Mean policy runs had a mean of 66.37 and a standard deviation of 2.2, the VTF policy runs had a mean 65.05 and a standard deviation of 2.0, and for the 95TF policy runs had a mean 66.62 and a standard deviation of 2.01. The results of the Compare metric are given in Table 5.9.

Policy	Better	Mixed	Worse
Mean	9	3	7
VTF	9	5	6
95TF	6	3	10

Table 5.9: Summary statistics using Compare evaluation for experiment pictured in Figure 5.30.

The statistics indicate only a slight reduction in execution time, and less variance in execution time as well, but VTF performed better than the Mean scheduling policy on a run-by-run basis.

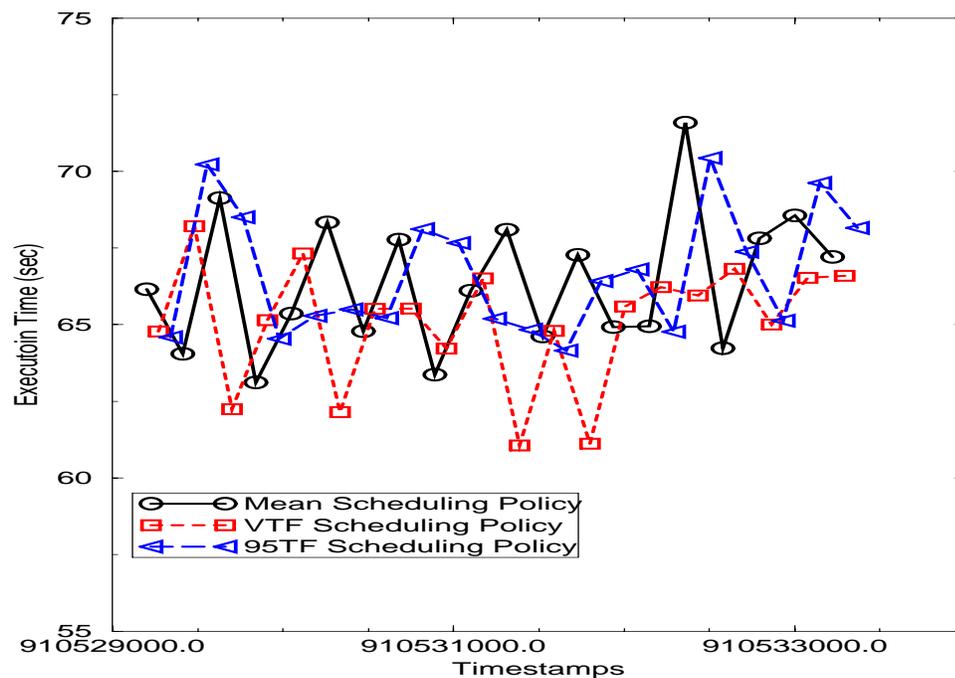


Figure 5.30: Comparison of **Mean**, **VTF**, and **95TF** policies, for the GA code on the Linux cluster with CPU loads shown in Figures 5.31 through 5.34.

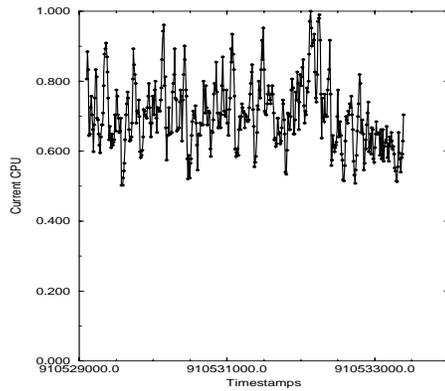


Figure 5.31: CPU values during runtime for scheduling experiments depicted in Figure 5.30 on Mystere, in the Linux cluster.

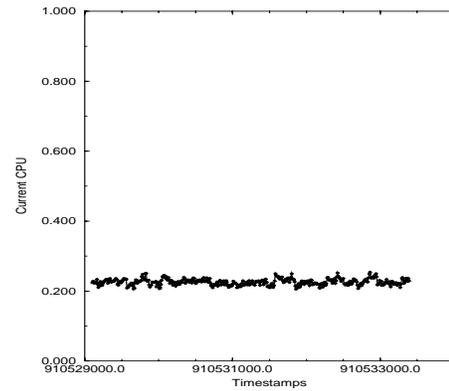


Figure 5.32: CPU values during runtime for scheduling experiments depicted in Figure 5.30 on Quidam, in the Linux cluster.

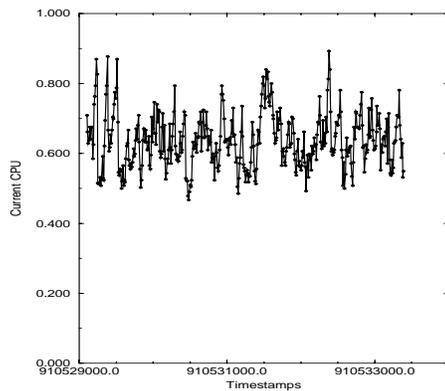


Figure 5.33: CPU values during runtime for scheduling experiments depicted in Figure 5.30 on Saltimbenco, in the Linux cluster.

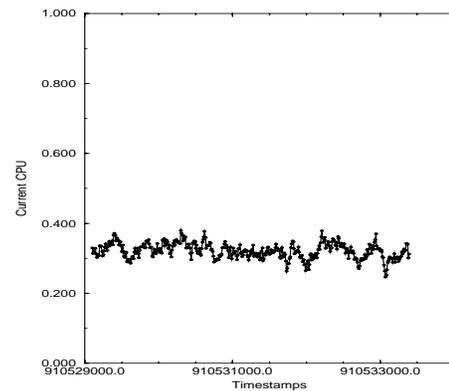


Figure 5.34: CPU values during runtime for scheduling experiments depicted in Figure 5.30 on Soleil, in the Linux cluster.

GA on Linux, High Variability in Available CPU

Figure 5.35 shows a comparison of the three scheduling policies when the Linux cluster had CPU loads shown in Figures 5.36 through 5.39, in which two machines showed a very high variability in available CPU. Using the Window metric, the lowest execution time was achieved by the Mean 16 times of 58 windows, 21 times for VTF and 21 times for 95TF. The Mean policy runs had a mean of 68.18 and a standard deviation of 2.67, the VTF policy runs had a mean 66.926 and a standard deviation of 2.52, and for the 95TF policy runs had a mean 66.93 and a standard deviation of 1.67. The results of the Compare metric are given in Table 5.10.

Policy	Better	Mixed	Worse
Mean	5	5	9
VTF	8	4	8
95TF	9	5	5

Table 5.10: Summary statistics using Compare evaluation for experiment pictured in Figure 5.35.

Again, there was a slight improvement in overall runtimes (2%), but the standard deviation was reduced by almost 40% when comparing Mean and 95TF. On a run-by-run basis, the stochastic policies were almost twice as likely to have a faster execution time.

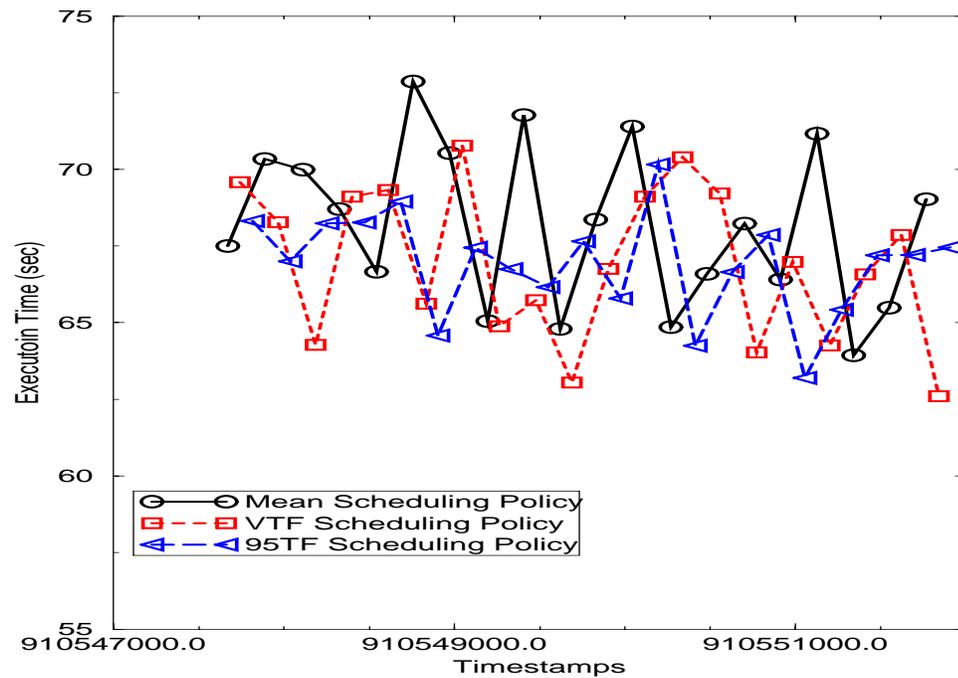


Figure 5.35: Comparison of **Mean**, **VTF**, and **95TF** policies, for the GA code on the Linux cluster with CPU loads shown in Figures 5.36 through 5.39.

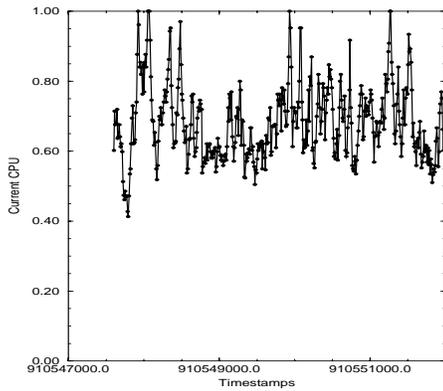


Figure 5.36: CPU values during runtime for scheduling experiments depicted in Figure 5.35 on Mystere, in the Linux cluster.

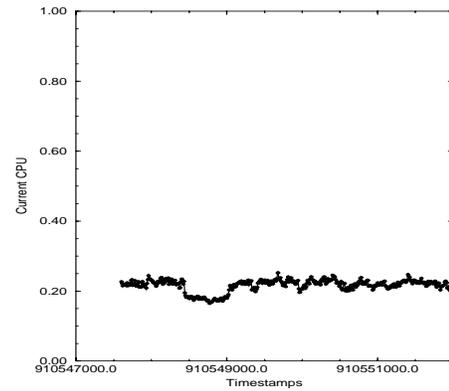


Figure 5.37: CPU values during runtime for scheduling experiments depicted in Figure 5.35 on Quidam, in the Linux cluster.

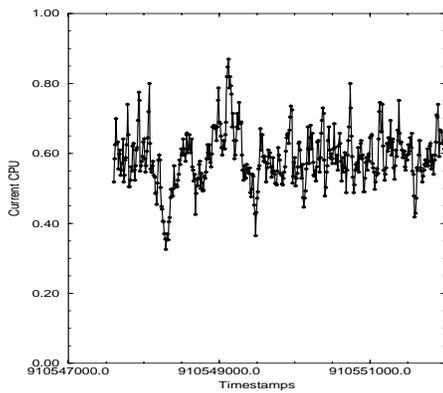


Figure 5.38: CPU values during runtime for scheduling experiments depicted in Figure 5.35 on Saltimbanco, in the Linux cluster.

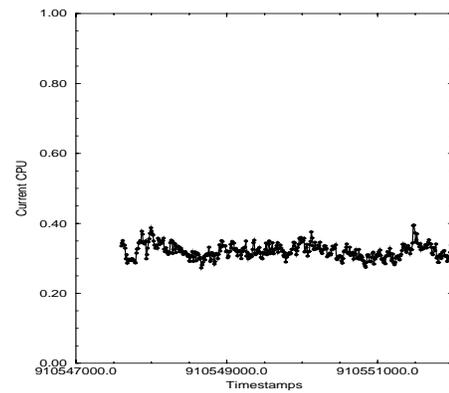


Figure 5.39: CPU values during runtime for scheduling experiments depicted in Figure 5.35 on Soleil, in the Linux cluster.

5.C.5 GA on PCL

Our fourth set of experiments were run for the GA code on the PCL cluster. Figure 5.40 shows a comparison of the three scheduling policies when the PCL cluster had CPU loads shown in Figures 5.41 through 5.44, in which two machines showed a very high variability in available CPU. Using the Window metric, the lowest execution time was achieved by the Mean 17 times of 58 windows, 23 times for VTF and 21 times for 95TF. The Mean policy runs had a mean of 78.93 and a standard deviation of 13.44, the VTF policy runs had a mean 75.60 and a standard deviation of 9.61. and for the 95TF policy runs had a mean 77.31 and a standard deviation of 9.76. The results of the Compare metric are given in Table 5.11.

Policy	Better	Mixed	Worse
Mean	4	8	8
VTF	8	7	6
95TF	7	7	6

Table 5.11: Summary statistics using Compare evaluation for experiment pictured in Figure 5.40.

Of note with this set of experiments is the spike seen at time 909960000, and the fact that both stochastic scheduling policies compensate for the variation significantly better than the Mean scheduling policy. In addition, over the entire set of runs, the VTF policy showed almost a 5% improvement over the Mean policy, and had a much smaller variation as well. The Compare metric also indicates that VTF was almost twice as likely to have the best execution time than Mean.

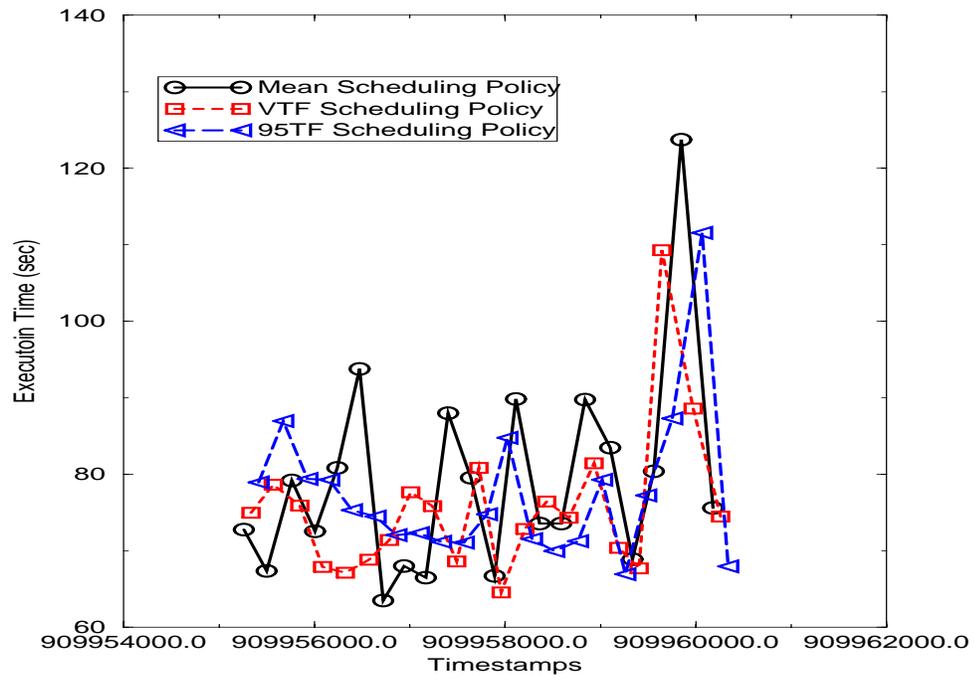


Figure 5.40: Comparison of **Mean**, **VTF**, and **95TF** policies, for the GA code on the PCL cluster with CPU loads shown in Figures 5.41 through 5.44.

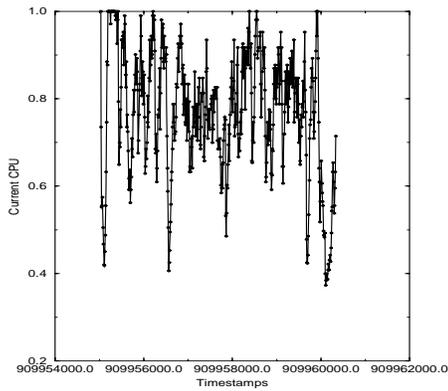


Figure 5.41: CPU values during runtime for scheduling experiments depicted in Figure 5.40 on Lorax, in the PCL cluster.

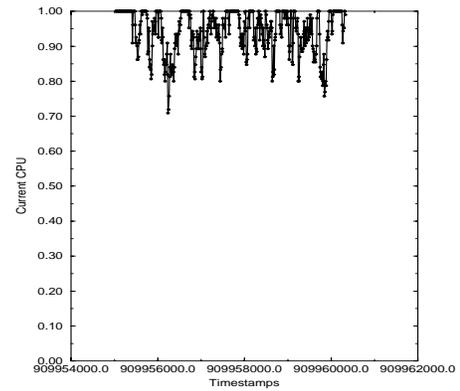


Figure 5.42: CPU values during runtime for scheduling experiments depicted in Figure 5.40 on Picard, in the PCL cluster.

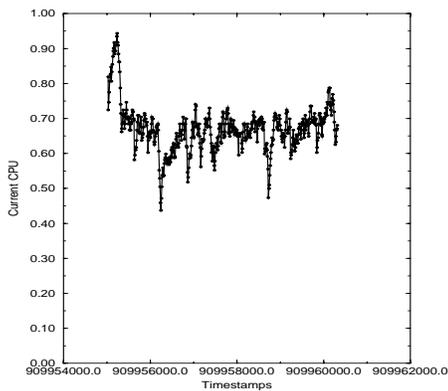


Figure 5.43: CPU values during runtime for scheduling experiments depicted in Figure 5.40 on Thing1, in the PCL cluster.

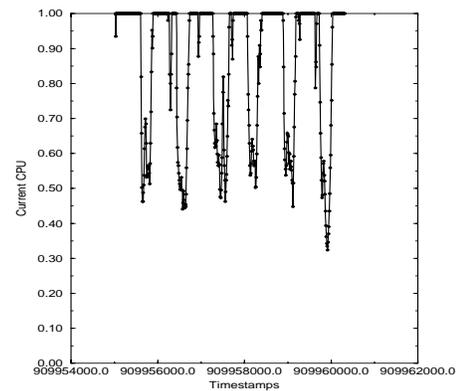


Figure 5.44: CPU values during runtime for scheduling experiments depicted in Figure 5.40 on Thing2, in the PCL cluster.

GA on PCL, High Variability in Available CPU

Figure 5.45 shows a comparison of the three scheduling policies when the PCL cluster had CPU loads shown in Figures 5.46 through 5.49 showing a large variation in available CPU on all four machines. Using the Window metric, the lowest execution time was achieved by the Mean 11 times of 58 windows, 33 times for VTF and 17 times for 95TF. The Mean policy runs had a mean of 76.48 and a standard deviation of 24.89, the VTF policy runs had a mean 72.55 and a standard deviation of 26.9, and for the 95TF policy runs had a mean 75.48 and a standard deviation of 23.3. The results of the Compare metric are given in Table 5.12.

Policy	Better	Mixed	Worse
Mean	2	10	8
VTF	10	9	2
95TF	4	10	6

Table 5.12: Summary statistics using Compare evaluation for experiment pictured in Figure 5.45.

It is unclear why 95TF performed so poorly, however VTF showed a 5% improvement over the mean execution times of the Mean policy. Likewise, VTF was almost 5 times more likely to have a faster execution time than Mean in a run-by-run comparison.

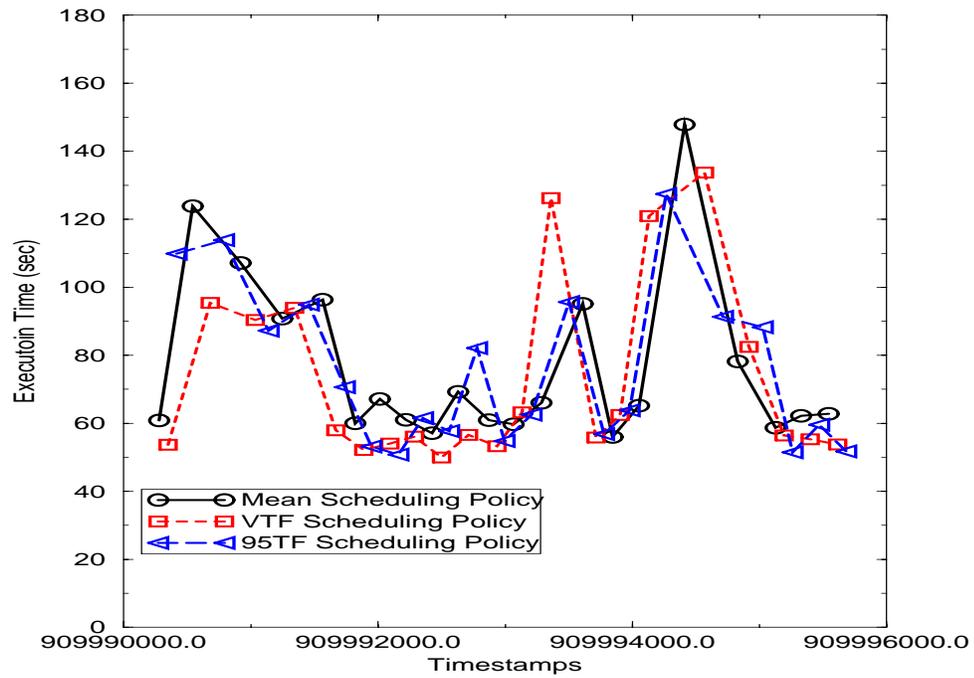


Figure 5.45: Comparison of **Mean**, **VTF**, and **95TF** policies, for the GA code on the PCL cluster with CPU loads shown in Figures 5.46 through 5.49.

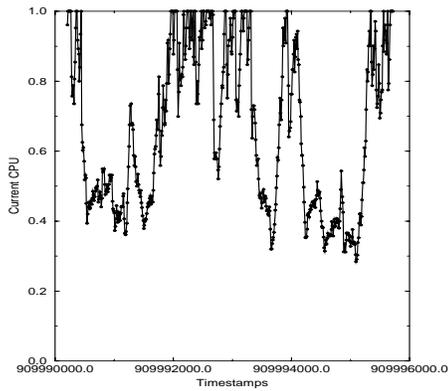


Figure 5.46: CPU values during runtime for scheduling experiments depicted in Figure 5.45 on Lorax, in the PCL cluster.

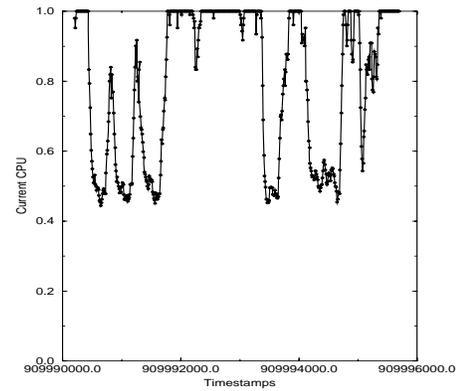


Figure 5.47: CPU values during runtime for scheduling experiments depicted in Figure 5.45 on Picard, in the PCL cluster.

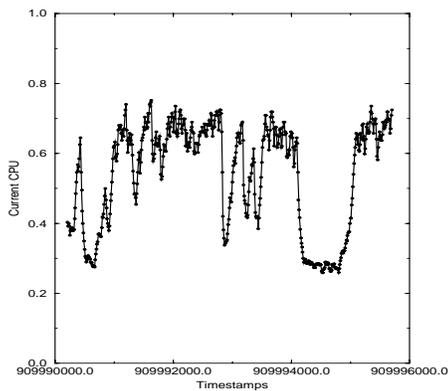


Figure 5.48: CPU values during runtime for scheduling experiments depicted in Figure 5.45 on Thing1, in the PCL cluster.

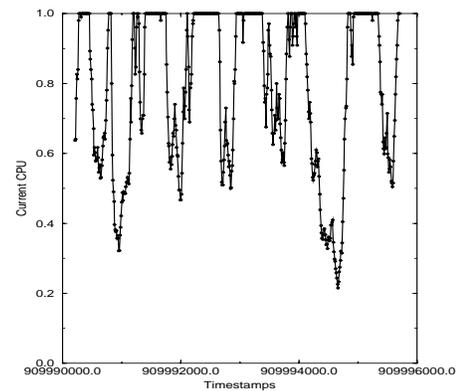


Figure 5.49: CPU values during runtime for scheduling experiments depicted in Figure 5.45 on Thing2, in the PCL cluster.

GA on PCL, Low Variation in Available CPU

Figure 5.50 shows a comparison of the three scheduling policies when the PCL cluster had CPU loads shown in Figures 5.51 through 5.54. Using the Window metric, the lowest execution time was achieved by the Mean 27 times of 58 windows, 17 times for VTF and 17 times for 95TF. The Mean policy runs had a mean of 41.31 and a standard deviation of 5.28, the VTF policy runs had a mean 42.18 and a standard deviation of 4.79, and for the 95TF policy runs had a mean 43.62 and a standard deviation of 6.82. The results of the Compare metric are given in Table 5.13.

Policy	Better	Mixed	Worse
Mean	10	6	4
VTF	5	7	9
95TF	5	7	8

Table 5.13: Summary statistics using Compare evaluation for experiment pictured in Figure 5.50.

Of note with this set of experiments is that 95TF appears to be overly conservative when Thing2 has idle cycles, although VTF appears to adjust more efficiently as one would expect.

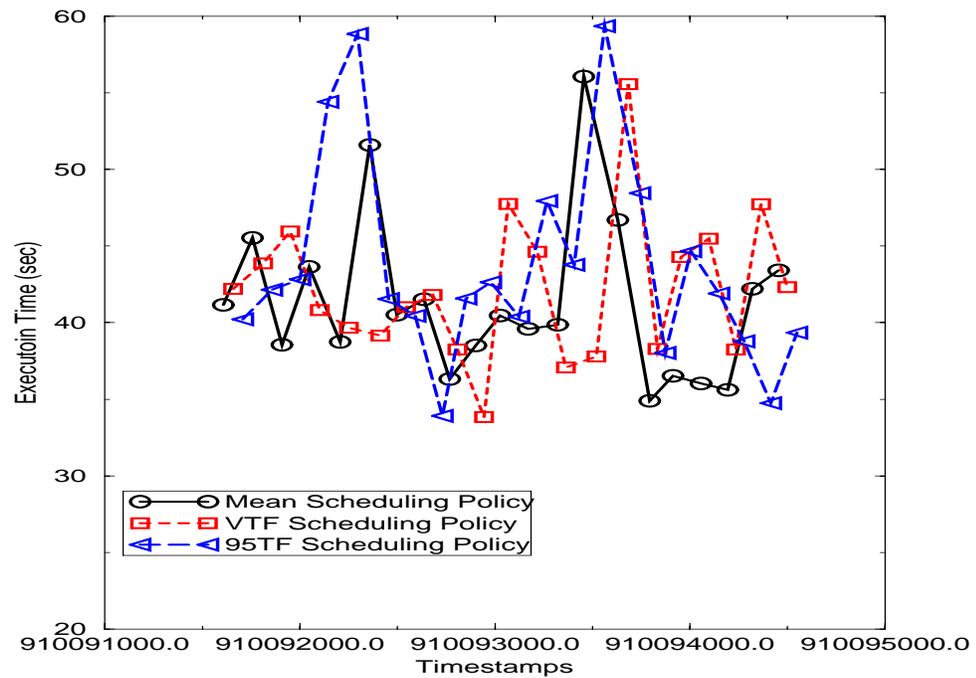


Figure 5.50: Comparison of **Mean**, **VTF**, and **95TF** policies, for the GA code on the PCL cluster with CPU loads shown in Figures 5.51 through 5.54.

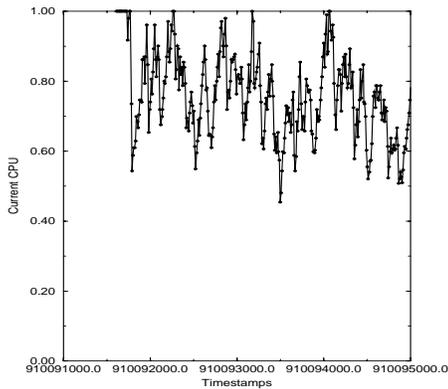


Figure 5.51: CPU values during runtime for scheduling experiments depicted in Figure 5.50 on Lorax, in the PCL cluster.

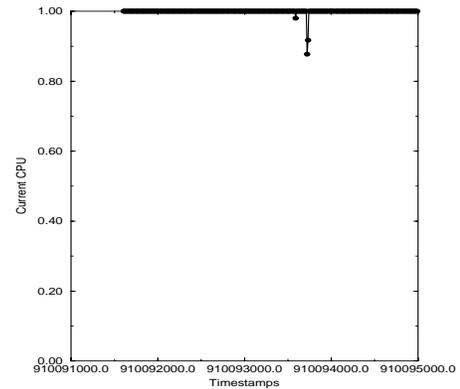


Figure 5.52: CPU values during runtime for scheduling experiments depicted in Figure 5.50 on Picard, in the PCL cluster.

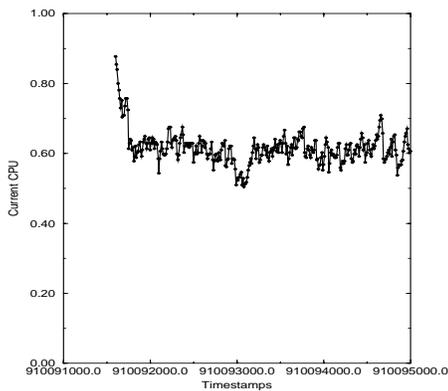


Figure 5.53: CPU values during runtime for scheduling experiments depicted in Figure 5.50 on Thing1, in the PCL cluster.

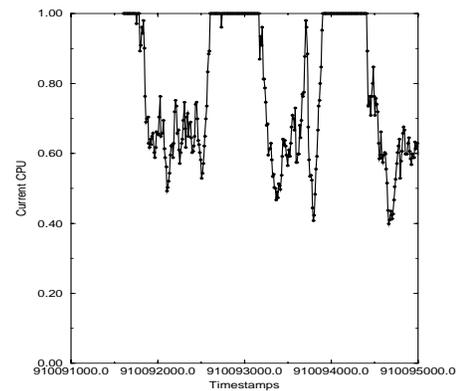


Figure 5.54: CPU values during runtime for scheduling experiments depicted in Figure 5.50 on Thing2, in the PCL cluster.

5.C.6 N-body Code on Linux Cluster

We also ran experiments using the N-Body solver, described in Section 2.D.3. Unlike the GA, this code was deterministic, but communication played a larger role in predicting the execution time.

Figure 5.55 shows a comparison of the three scheduling policies when the Linux cluster had CPU loads shown in Figures 5.56 through 5.59 which show a fairly consistent variation in available CPU. Using the Window metric, the lowest execution time was achieved by the Mean 15 times of 58 windows, 23 times for VTF and 20 times for 95TF. The Mean policy runs had a mean of 61.93 and a standard deviation of 7.48, the VTF policy runs had a mean 59.84 and a standard deviation of 5.22, and for the 95TF policy runs had a mean 59.73 and a standard deviation of 4.81. The results of the Compare metric are given in Table 5.15.

Policy	Better	Mixed	Worse
Mean	5	7	7
VTF	8	5	7
95TF	7	5	7

Table 5.14: Summary statistics using Compare evaluation for experiment pictured in Figure 5.55.

These statistics show a slight improvement in overall execution time (4-5%) when comparing the stochastic policies to the Mean policy, and a reduced variation as well. Moreover, on a run-by-run basis, as shown by the Window and Compare metrics, the stochastic scheduling policies were more likely to achieve a better execution time.

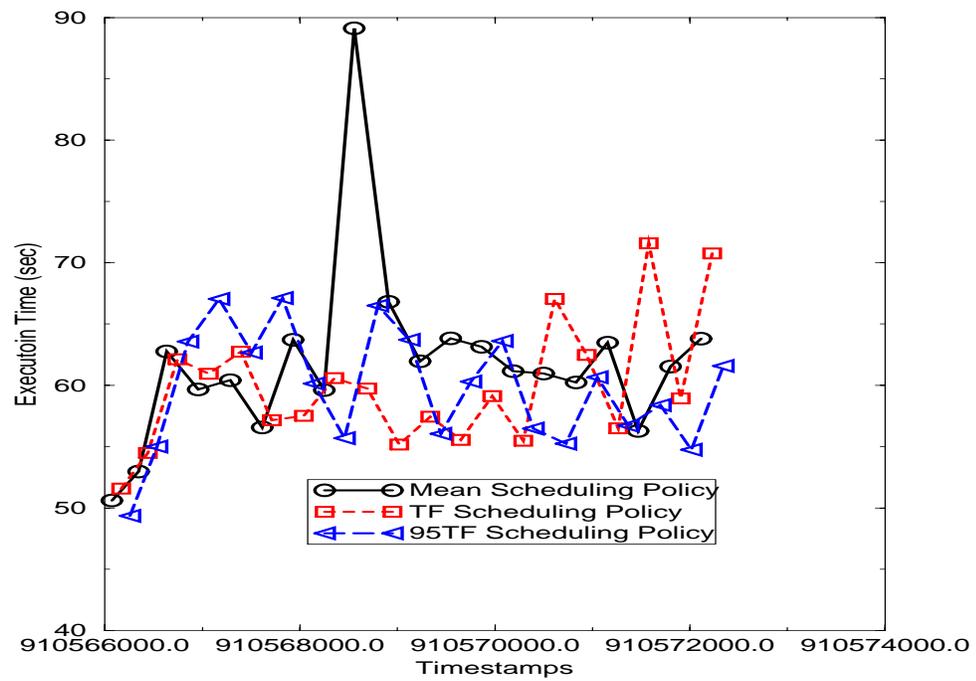


Figure 5.55: Comparison of **Mean**, **VTF**, and **95TF** policies, for the NBody benchmark on the Linux cluster with CPU loads shown in Figures 5.56 through 5.59.

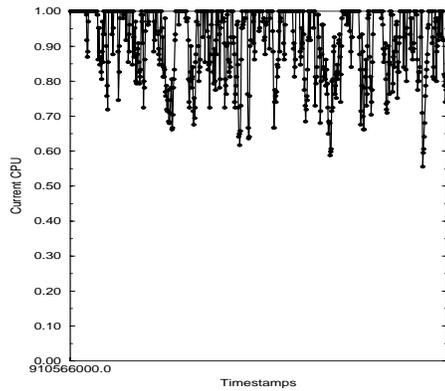


Figure 5.56: CPU values during runtime for scheduling experiments depicted in Figure 5.55 on Mystere, in the Linux cluster.

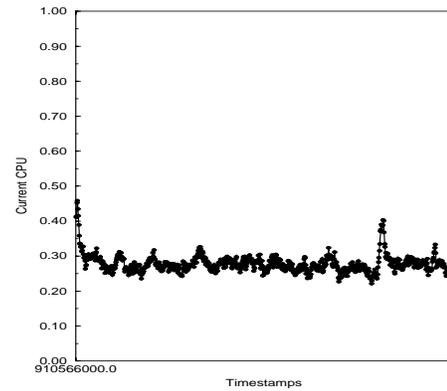


Figure 5.57: CPU values during runtime for scheduling experiments depicted in Figure 5.55 on Quidam, in the Linux cluster.

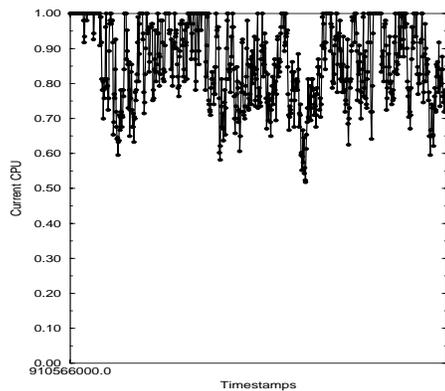


Figure 5.58: CPU values during runtime for scheduling experiments depicted in Figure 5.55 on Saltimbanco, in the Linux cluster.

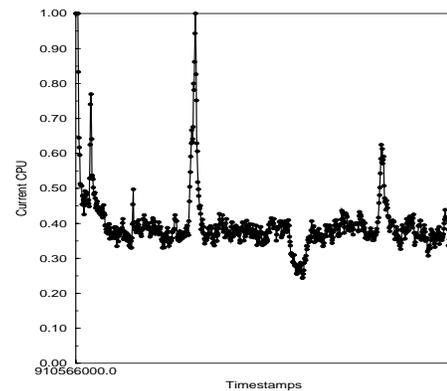


Figure 5.59: CPU values during runtime for scheduling experiments depicted in Figure 5.55 on Soleil, in the Linux cluster.

NBody on Linux, Low Variability in Available CPU

Figure 5.60 shows a comparison of the three scheduling policies when the Linux cluster had CPU loads shown in Figures 5.61 through 5.64, showing a very small variation in available CPU. Using the Window metric, the lowest execution time was achieved by the Mean 8 times of 58 windows, 34 times for VTF and 16 times for 95TF. The Mean policy runs had a mean of 53.96 and a standard deviation of 4.46, the VTF policy runs had a mean 52.44 and a standard deviation of 5.42, and for the 95TF policy runs had a mean 54.49 and a standard deviation of 5.75. The results of the Compare metric are given in Table 5.15.

Policy	Better	Mixed	Worse
Mean	4	6	9
VTF	13	1	6
95TF	6	5	8

Table 5.15: Summary statistics using Compare evaluation for experiment pictured in Figure 5.60.

Using these statistics, we see only a slight improvement in overall execution times when comparing the Mean policy to the two stochastic policies, and not even that when comparing the Mean and 95TF. However, the Compare and Window metrics indicate that on a run-by-run basis, there is a significant improvement, with VTF having a minimal execution time nearly three times as often as Mean.

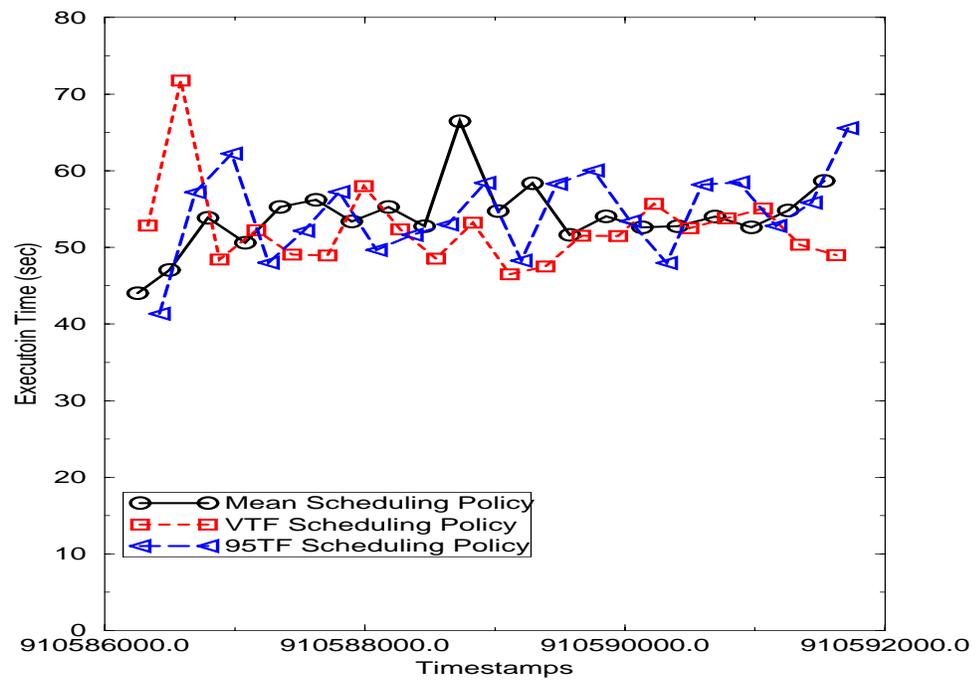


Figure 5.60: Comparison of **Mean**, **VTF**, and **95TF** policies, for the NBody benchmark on the Linux cluster with CPU loads shown in Figures 5.61 through 5.64.

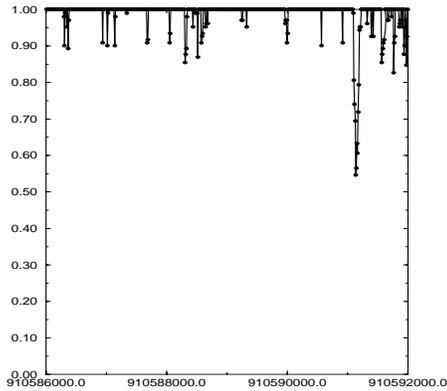


Figure 5.61: CPU values during runtime for scheduling experiments depicted in Figure 5.60 on Mystere, in the Linux cluster.

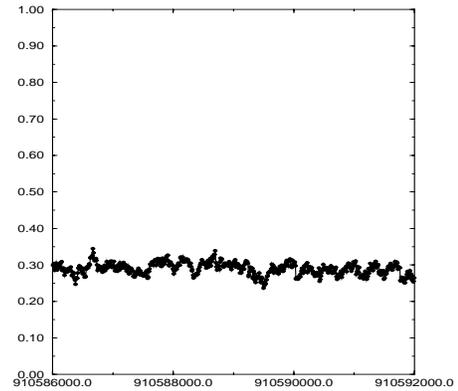


Figure 5.62: CPU values during runtime for scheduling experiments depicted in Figure 5.60 on Quidam, in the Linux cluster.

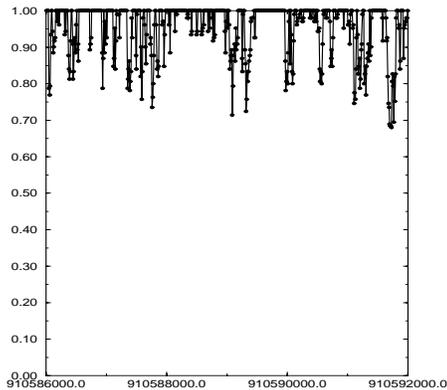


Figure 5.63: CPU values during runtime for scheduling experiments depicted in Figure 5.60 on Saltimbanco, in the Linux cluster.

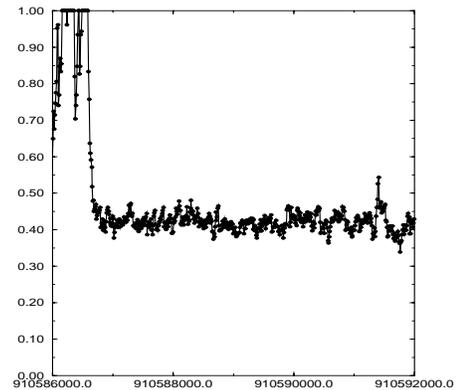


Figure 5.64: CPU values during runtime for scheduling experiments depicted in Figure 5.60 on Soleil, in the Linux cluster.

NBody on Linux, Higher Variation in Available CPU

Figure 5.65 shows a comparison of the three scheduling policies when the Linux cluster had CPU loads shown in Figures 5.66 through 5.69. Using the Window metric, the lowest execution time was achieved by the Mean 9 times of 58 windows, 26 times for VTF and 23 times for 95TF. The Mean policy runs had a mean of 62.64 and a standard deviation of 10.03, the VTF policy runs had a mean 59.93 and a standard deviation of 11.72, and for the 95TF policy runs had a mean 60.49 and a standard deviation of 10.05. The results of the Compare metric are given in Table 5.16.

Policy	Better	Mixed	Worse
Mean	3	8	8
VTF	8	8	4
95TF	7	6	6

Table 5.16: Summary statistics using Compare evaluation for experiment pictured in Figure 5.65.

Using these statistics, we see that there is a 5% improvement when using VTF over the Mean policy, although for these runs there is not an improvement in the standard deviation of the execution times. Of note is the large change in performance seen at time stamp 910596000 when the load on three of the machines increased significantly.

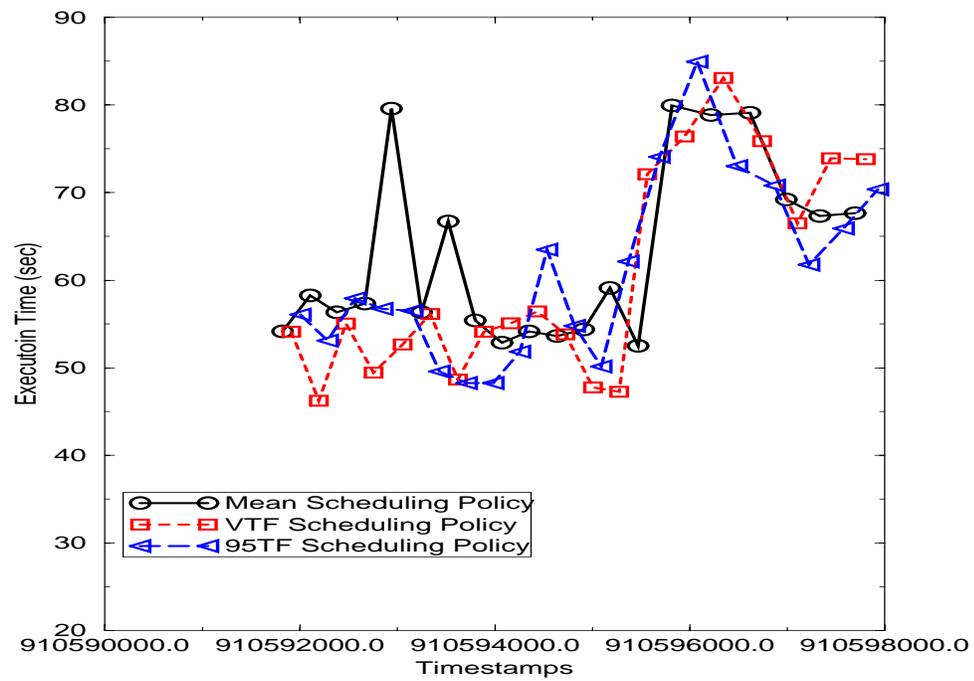


Figure 5.65: Comparison of **Mean**, **VTF**, and **95TF** policies, for the NBody benchmark on the Linux cluster with CPU loads shown in Figures 5.66 through 5.69.

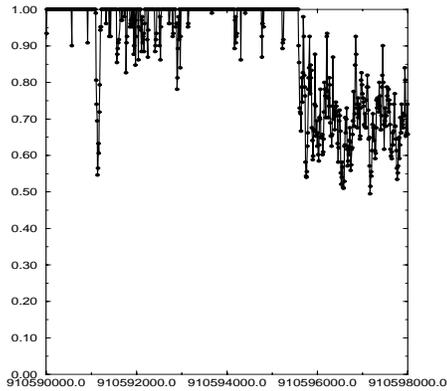


Figure 5.66: CPU values during runtime for scheduling experiments depicted in Figure 5.65 on Mystere, in the Linux cluster.

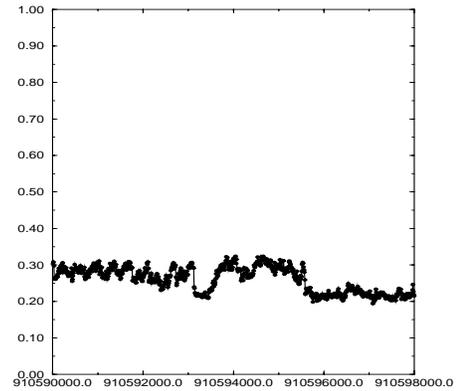


Figure 5.67: CPU values during runtime for scheduling experiments depicted in Figure 5.65 on Quidam, in the Linux cluster.

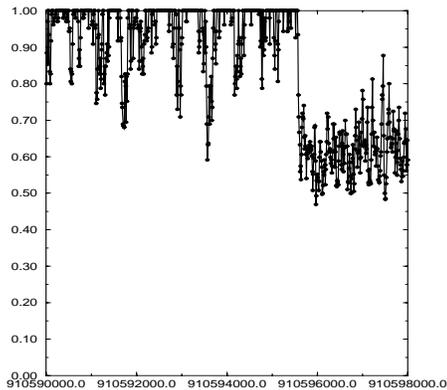


Figure 5.68: CPU values during runtime for scheduling experiments depicted in Figure 5.65 on Saltimbanco, in the Linux cluster.

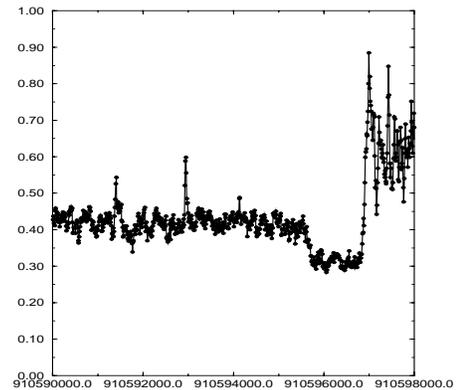


Figure 5.69: CPU values during runtime for scheduling experiments depicted in Figure 5.65 on Soleil, in the Linux cluster.

5.C.7 Summary

The previous subsections presented graphs comparing three scheduling policies for several applications on two shared clusters of workstations. In general, we found that the policies with non-zero Tuning Factors (VTF and 95TF) led to reduced execution times when there was a high variation in available CPU. In addition, in almost all cases the standard deviation (or variation associated with the actual execution times) was greatly reduced when using non-zero Tuning factors, leading to more predictable execution time behavior.

5.D Related Work

5.D.1 Scheduling

The most closely related work, and the primary work this project grew out of, is AppLes – Application Level Scheduler [BW96, BW97, BWF⁺96]. The AppLes approach recognizes that applications require special-purpose tuned schedules to achieve the best possible performance. A scheduling agent, tightly-coupled to the application, is built to maximize the performance of the application using dynamic information and adaptive scheduling techniques.

Prophet [WZ98] uses run-time granularity information to select the best number of processors to apply to the application, and is also aware of the overhead possibly added by a scheduler. Prophet supports the use of highly-shared networks by utilizing dynamic system information about the level of CPU and network usage at runtime supplied by the Network resource Monitoring System (NRMS), much as our scheduling and prediction approach made use of the Network Weather Service.

5.D.2 Data Allocation

Our approach to data allocation was based on a scheduling approach originally presented by Sih and Lee [SL90b, SL90a, SL93]. This work concentrated on

a compile-time scheduling approach for interconnection-constrained architectures, and used a system of benefits and penalties (although not identified as such) for task mapping and data allocation. As a compile-time approach, it did not use current system information, and targeted a different set of architectures than our work as well.

Zaki et al. [ZLP96] also focused on the data allocation aspect of scheduling. This work compared load balancing techniques showing that different schemes were best for different applications under varying program and system parameters. However, the load model used in this work did not accurately portray the available CPU seen on our systems.

5.D.3 Other Related Approaches

Waldspurger and Weihl have examined the use of a randomized resource allocation mechanism called *lottery scheduling* [WW94]. Lottery Scheduling is an operating system level scheduler that provides flexible and responsive control over the relative execution rates of a wide range of applications. It is related to our approach in that they manage highly shared resources using time series information, however this approach is not directly applicable to a higher level scheduler such as ours. In addition, this work allows modular resource management, an approach that may extend into our scheduling in the future.

Remulac [BG98] investigates the design, specification, implementation and evaluation of a software platform for network-aware applications. Such applications must be able to obtain information about the current status of the network resources, provided by Remos (REsource MOnitoring System) [LMS⁺98]. Remos and Remulac address issues in finding a compromise between accuracy (the information provided is best-effort, but includes statistical information if available) and efficiency (providing a query-based interface, so applications incur overhead only when they acquire information) much as we were concerned with the tradeoffs between accuracy and efficiency in stochastic information and scheduling choices.

5.E Summary

In this chapter we have examined one use of stochastic prediction information, namely building stochastic scheduling policies. We extended the standard time balancing data allocation policy to use stochastic information to determine the data allocation when we have stochastic information represented as normal distributions. This extension hinged on the definition of a Tuning Factor for the system which represented the variability of the system as a whole and was used as the “knob” that determines the percentage of conservatism needed for the scheduling policy. Our experimental results demonstrate that it is often possible to obtain faster execution times and more predictable application behavior using stochastic information in schedules.

Chapter 6

Conclusion

The goal of this dissertation has been to present a technique for modeling distributed parallel applications and a scheduling approach that can utilize enhanced modeling information to determine performance-efficient application schedules. We presented structural modeling, an adaptable, flexible approach to compositional modeling, and extended this technique to allow stochastic parameters. We then used structural models with stochastic predictions to tune a time balancing scheduling policy for data parallel applications. Our results showed that stochastic scheduling policies often resulted in faster execution times and more predictable application behavior.

6.A Critique and Extensions

As in any thesis, this work circumscribed a problem space that can be extended in many directions. A few of these areas include:

Extending the applicability of structural models. The structural modeling approach presented in Chapter 3 adequately predicted the behavior of two classes of applications when the performance of the application consisted of communication and computation, but did not account for other execution behaviors. Two execution behaviors we plan to add to the structural modeling

approach are I/O behavior, perhaps building off work by Smirni [SRar, RSSS98], and memory behavior, possibly using work presented in PMH [ACFS93] or the LDA Model [SW98a] as starting points. In addition, we would like to expand the classes of applications that structural models have been shown to apply to.

Expanding the use of structural models. We demonstrated the use of structural models in predictions for scheduling, there are other areas in which such an approach might prove to be useful. Future work includes examining how application developers might use the modular design and extensibility of structural models as part of a large software design project in order to predict the execution time of a partially written application and steer development efforts. In addition, we would like to examine the use of structural models for porting efforts. A very common problem for application scientists is the fact that the platforms continually change as hardware resources are upgraded. Given an application running on soon-to-be-retired platform, applications developers would like to have a pragmatic way to decide where their application would best be ported to, and structural models may provide one way to determine this information.

Using other forms of extended information. We extended structural models, predictions and scheduling to take into account stochastic values. There are other sources of meta-data that might also be beneficial to include, such as error or bias predictions, or estimates of computational complexity. Another common problem seen in predictions is the lack of a complete set of information, so we would like to investigate how to compensate for incomplete information in predictions and scheduling.

Extending the applicability of stochastic predictions. In Chapter 4 we detailed a predictive approach to be used as part of an on-line scheduler. We would like to address the different needs of predictions used for resource selection, as this is also a very important part of achieving performance on parallel distributed systems. In addition, we would like to expand the systems that our predictions apply to, both in terms of being able to add other types of machines, for example

including predictions for queue times so this approach can take into account queued systems, and by examining what factors would need to be added in order to use this approach for larger grid-style computing systems.

Extending stochastic scheduling. By far, the most room for future work can be seen in the stochastic scheduling approach. Some issues we plan to address are:

- Can stochastic data be used in other parts of the scheduling process besides data allocation?
- What other scheduling policies can take advantage of stochastic values, and how?
- Further extensions to time-balancing: definitions for other stochastic value representations; inclusion of networking information, memory data; other Tuning factor definitions
- Dynamic re-balancing approaches

All of these are logical extensions to the work presented.

6.B Conclusions and Contributions

Distributed parallel platforms are becoming more and more prevalent in research environments. As they grow in popularity, so does their user base. Because of this, in part, a growing problem in achieving performance on these platforms is resource contention. Our solution to this problem was the development of a more sophisticated scheduling approach that could factor in the variability of resource performance capabilities.

This thesis presented three main contributions: an approach to define distributed parallel application performance models called **structural modeling**, the ability to parameterize these models with **stochastic values** in order to meet the

prediction needs of shared clusters of workstations, and a **stochastic scheduling policy** that can make use of stochastic predictions to achieve better application execution times and more predictable application behavior. We verified each of these contributions for an extensive test suite of applications, several platforms and a variety of load conditions.

In conclusion, we believe that structural modeling is a promising approach for predicting the behavior of parallel applications running on distributed platforms due to its extensibility and flexibility. This approach can be augmented to allow for a variety of sources and types of data, as we extended structural models to allow for stochastic valued parameters. The resulting predictions from these models can be used in a variety of settings, with out example of an on-line scheduler just one possibility.

We demonstrated that stochastic values contribute valuable additional information about the performance of applications running on shared clusters of workstations. Both using interval and normal distribution representations of stochastic data can be used efficiently, with differing results that will meet different prediction needs.

In addition, we defined a stochastic scheduling approach that defined an initial approach but also opens up a broad new area of possible schedulers that take into account additional available information. Our initial instantiation shows promise in reducing execution time for systems with a high variability in available performance, as seen in many shared cluster environments.

Bibliography

- [ACFS93] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 1993.
- [Adv93] Vikram S. Adve. *Analyzing the Behavior and Performance of Parallel Programs*. PhD thesis, University of Wisconsin-Madison, December 1993. Also available as University of Wisconsin Computer Sciences Technical Report #1201.
- [AHWC98] Scott Anderson, David Hart, David Westbrook, and Paul Cohen. A toolbox for analyzing programs. *International Journal on AI Tools*, 1998. to appear.
- [AL77] A. Arakawa and V. R. Lamb. Computational design of the basic dynamical processes of the ucla general circulation model. *Methods in Comput. Phys.*, 17:173–265, 1977.
- [And97] Kelsey Anderson. Personal communication, Summer 1997.
- [ASWB95] Cosimo Anglano, Jennifer Schopf, Richard Wolski, and Francine Berman. Zoom: A hierarchical representation for heterogeneous applications. Technical Report #CS95-451, University of California, San Diego, Computer Science Department, 1995.
- [AV93] Vikram Adve and Mary Vernon. The influence of random delays on parallel execution times. In *Proceedings of Sigmetrics '93*, 1993.
- [Bai72] A. Baines. Introduction. In Owen L. Davies and Peter L. Goldsmith, editors, *Statistical Methods in Research and Production*, chapter One. Published for Imperial Chemical Industries Limited by Oliver and Boyd, 1972.
- [Bai95] D. H. Bailey, editor. *Seventh SIAM conference on parallel processing for scientific computing*, 1995.
- [Bar78] B. Austin Barry. *Errors in Practical Measurement in Science, Engineering and Technology*. John Wilsey & Sons, 1978.

- [BBB⁺91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T.A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrisnam, and S. K. Weeratunga. The nas parallel benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
- [BBC⁺94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.
- [BC88] J. Bruno and P.R. Cappello. Implementing the beam and warming method on the hypercube. In *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, 1988.
- [BCF⁺98] S. Brunett, K. Czajkowski, S. Fitzgerald, I. Foster, A. Johnson, C. Kesselman, J. Leigh, and S. Tuecke. Application experiences with the globus toolkit. In *Proceedings of the 7th IEEE Symp. on High Performance Distributed Computing (HPDC-7)*, 1998.
- [BDGM93] A. Beguelin, J. Dongarra, A. Geist, and R. Manchek. Hence: A heterogeneous network computing environment. *Scientific Programming*, 3(1):49–60, 1993.
- [Ber98] Francine Berman. *The Grid: Blueprint for a New Computing Infrastructure*, edited by Ian Foster and Carl Kesselman, chapter 12: High-Performance Schedulers. Morgan Kaufmann Publishers, Inc., 1998.
- [BFM91] Marcos A. G. Brasileiro, James A. Field, and Antao B. Moura. Modeling and analysis of time critical applications on local area networks. In *Proc. of Chilean Computer Science Society*, pages 459–71, 1991.
- [BFVW93] E. Barszcz, R. Fatoohi, V. Venkatakrisnam, and S. Weeratunga. Solution of regular, sparse triangular linear systems on vector and distributed-memory multiprocessors. Technical Report Technical Report NAS RNR-93-007, NASA Ames Research Center, Moffett Field, CA, April 1993.
- [BG98] J. Bolliger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE Transactions on Software Engineering (Special Issue on Mobility and Network-Aware Computing)*, 24(5), May 1998.
- [Bha96] Karan Bhatia. Personal communication, 1996.

- [BHS⁺95] David Bailey, Tim Harris, William Saphir, Rob van der Wijn- gaart, Alex Woo, and Maurice Yarrow. The nas parallel bench- marks 2.0. Technical Report Report NAS-95-020, Numerical Aerospace Simulation Facility at NASA Ames Research Center, De- cember 1995. Also available at [http://science.nas.nasa.gov/ Software/NPB/Specs/npb2_0/npb2_0.html](http://science.nas.nasa.gov/Software/NPB/Specs/npb2_0/npb2_0.html).
- [BK91] Scott B. Baden and Scott R. Kohn. A comparison of load balanc- ing strategies for particle methods running on mimd multiprocessors. Technical Report CS91-199, University of California, San Diego, Com- puter Science and Engineering Dept., 1991.
- [Bla80] Leland Blank. *Statistical Procedures for engineering, management, and science*. McGraw-Hill Book Company, 1980.
- [Bri87] William L. Briggs. *A Multigrid Tutorial*. Society for Industrial and Applied Mathematics, Lancaster Press, 1987.
- [BW96] Francine Berman and Richard Wolski. Scheduling from the perspec- tive of the application. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996.
- [BW97] Francine Berman and Richard Wolski. The apples project: A status report. In *Proceedings of the 8th NEC Research Symposium*, 1997.
- [BW98] Francine Berman and Richard Wolski. Performance prediction engineering. DARPA report, available from <http://www.cs.ucsd.edu/groups/hpcl/apples/PPE/report.html>, 1998.
- [BWF⁺96] Francine Berman, Richard Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heteroge- neous networks. In *Proceedings of SuperComputing '96*, 1996.
- [BWS97] Francine Berman, Richard Wolski, and Jennifer Schopf. Performance prediction engineering for metacomputing. <http://www.cs.ucsd.edu/groups/hpcl/apples/PPE/index.html>, 1997.
- [Car98] Larry Carter. Personal communication, 1998.
- [CGS95] Brad Calder, Dirk Grunwald, and Amitabh Srivastava. The pre- dictability of branches in libraries. In *Proceedings of the 28th In- ternational Symposium on Microarchitecture*, 1995. Also available as WRL Research Report 95.6.
- [Cha94] K. M. Chandy. Concurrent program archetypes. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, 1994.

- [CI98] Alex Cereghini and Wes Ingalls. Personal communication, 1998.
- [Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT Press, 1989.
- [Cox84] M. D. Cox. A primitive, 3-dimensional model of the ocean. Technical Report Technical Report 1, GFDL Ocean Group, 1984.
- [CQ93] Mark J. Clement and Michael J. Quinn. Analytical performance prediction on multicomputers. In *Proceedings of SuperComputing '93*, 1993.
- [Cra68] J. M. Craddock. *Statistics in the Computer Age*. The English Universities Press Limited, 1968.
- [Cro94] Mark Edward Crovella. *Performance prediction and tuning of parallel programs*. PhD thesis, University of Rochester, 1994.
- [CS95] Robert L. Clay and Peter A. Steenkiste. Distributing a chemical process optimization application over a gigabit network. In *Proceedings of SuperComputing '95*, 1995.
- [Das97] Chita Das. Personal communication with rich wolski, 1997.
- [DDH⁺98] Ewa Deelman, Aditya Dube, Adolfy Hoisie, Yong Luo, Richard L. Oliver, David Sundaram-Strukel, Harvey Wassermann, Vikram S. Adve, Rajive Bagrodia, James C. Browne, Elias Houstis, Olaf Lubeck, John Rice, Patricia J. Teller, and Mary K. Vernon. Poems: End-to-end performance design of large parallel adaptive computational systems. In *Proceedings of the First International Workshop on Software and Performance*, October 1998.
- [DFH⁺93] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. C. While. Parallel programming using skeleton functions. In *Proceedings of Parallel Architectures and Languages Europe (PARLE) '93*, 1993.
- [Din98] P. Dinda. The statistical properties of host load. In *Proceedings of the 4th Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, 1998. Also available from <http://www.cs.cmu.edu/~pdinda/html/papers.html>.
- [DMP97] Ralf Diekmann, Burkahrad Monien, and Robert Pries. Load balancing strategies for distributed memory machines. In F. Karsh and H. Saltz, editors, *Multi-Scale Phenomena and their Simulation*. World Scientific, 1997.

- [DN95] J. Dongarra and Peter Newton. Overview of vpe: A visual environment for message-passing parallel programming. In *Proceedings of the 4th Heterogeneous Computing Workshop*, 1995.
- [DO98] P. Dinda and D. O'Hallaron. The statistical properties of host load. Technical Report Technical Report CMU-CS-98-143, School of Computer Science, Carnegie Mellon University, 1998. Also available from <http://www.cs.cmu.edu/~pdinda/html/papers.html>.
- [DP96a] Jay Devore and Roxy Peck. *Statistics: The Exploration and Analysis of Data*, page 272. Duxbury Press, 1996.
- [DP96b] Jay Devore and Roxy Peck. *Statistics: The Exploration and Analysis of Data*, page 88. Duxbury Press, 1996.
- [DP96c] Jay Devore and Roxy Peck. *Statistics: The Exploration and Analysis of Data*, page 567. Duxbury Press, 1996.
- [DS95] J. Demmel and S. L. Smith. Performance of a parallel global atmospheric chemical tracer model. In *Proceedings of SuperComputing '95*, 1995.
- [DTMH96] D. M. Deaven, N. Tit, J. R. Morris, and K. M. Ho. Structural optimization of lennard-jones clusters by a genetic algorithm. *Chemical Physical Letters*, 256:195, 1996.
- [FB96] Silvia Figueira and Francine Berman. Modeling the effects of contention on the performance of heterogeneous applications. In *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing*, 1996.
- [FJ98] A. Fersha and J. Johnson. Performance oriented development of spmd program based on task structure specifications. to appear, also available from <http://socrates.ani.univie.ac.at/~ferscha/publications.html>, 1998.
- [FK97] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, Summer 1997. Also available at <ftp://ftp.mcs.anl.gov/pub/nexus/reports/globus.ps.Z>.
- [Fos95] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
- [FOW84] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In *Proceedings of the International Symposium on Programming 6th Colloquium*, 1984.

- [Geh95] Jörn Gehring. Dynamic program description as a basis for runtime optimization. Technical Report PC²/TR-002-97, Paderborn Center for Parallel Computing, 1995.
- [Ger98] Apostolos Gerasoulis. Personal communication, 1998.
- [GKM82] S. Graham, P. Kessler, and M. McKusick. gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, 1982. Also published in SIGPLAN Notices, volume 17, number 6, pages 120–126.
- [GLFK98] Andrew Grimshaw, Michael Lewis, Adam Ferrari, and John Karpovich. Architectural support for extensibility and autonomy in wide-area distributed object systems. Technical Report UVa CS Technical Report CS-98-12, University of Virginia, Computer Science Dept., June 1998.
- [GR96] Jörn Gehring and Alexander Reinefeld. Mars - a framework for minimizing the job execution time in a metacomputing environment. *Future Generation Computer Systems*, Spring 1996.
- [Gus90] John Gustafson. Fixed time, tiered memory, and superlinear speedup. In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, 1990.
- [GW] Lawrence L. Green and Robert Weston. Hpcpp cooperation on high speed civil transport. <http://hpcpp-www.larc.nasa.gov/~llgreen/hpcpp-hsct.html>.
- [GWtLt97] Andrew S. Grimshaw, William A. Wulf, and the Legion team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.
- [Han75] Eldon R. Hansen. A generalized interval arithmetic. *Interval Mathematics, proceedings of the International Symposium*, 1975.
- [Han92] Eldon R. Hansen. Global optimization using interval analysis. *Mono-graphs and textbooks in pure and applied mathematics*, 1992.
- [HK97] David F. Hegarty and M. Tahar Kechadii. Topology preserving dynamic load balancing for parallel molecular simulations. In *Proceedings of Supercomputing '97*, 1997.
- [HM96] Jeffrey K. Hollingsworth and Barton P. Miller. An adaptive cost model for parallel program instrumentation. In *Proceedings of EuroPar'96*, 1996. Also available at <http://www.cs.umd.edu/~hollings/papers/>.

- [HS91] Reinhard V. Hanxleden and L. Ridgeway Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13, 1991.
- [HT] John Helly and John Truong. San diego bay project perspective. www.sdsc.edu/SDBAY/sdbay.html.
- [HT97] M. Heath and V. Torczon, editors. *Eighth SIAM conference on parallel processing for scientific computing*, 1997.
- [IKP96] S. Irani, A. R. Karlin, and S. Phillips. Strongly competitive algorithms for paging with locality of reference. *SIAM Journal on Computing*, 25(3):477–97, June 1996.
- [Jam87] Leah Jamieson. *The Characteristics of Parallel Algorithms*, chapter 3, *Characterizing Parallel Algorithms*. MIT Press, 1987. eds. L. Jamieson, D. Gannon and R. Douglass.
- [Kar96] John F. Karpovich. Support for object placement in wide area heterogeneous distributed systems. Technical Report CS-96-03, University of Virginia, Department of Computer Science, January 1996.
- [Kas96] N. K. Kasabov. *Foundations of neural networks, fuzzy systems and knowledge engineering*. The MIT press, 1996.
- [KB97] C. Kesselman and J. Bannister. Qualis: Guaranteed quality of service for metacomputing, 1997. Private communication.
- [KME89] A. Kapelnikov, R. R. Muniz, and M. D. Ercegovac. A modeling methodology for the analysis of concurrent systems and computations. *Journal of Parallel and Distributed Computing*, 6:568–597, 1989.
- [KMMO94] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki. Competitive randomized algorithms for nonuniform problems. *Algorithmica*, 11(6):542–71, June 1994.
- [Koh97] James Arthur Kohl. Personal communication, 1997.
- [KRB83] Samuel Kotz, Campbell B. Read, and David L. Banks, editors. *Encyclopedia of Statistical Sciences*. John Wiley & Sons, 1983.
- [LG97] Richard N. Lagerstrom and Stephan K. Gipp. Psched: Political scheduling on the cray t3e. In *Proceedings of the Job Scheduling Strategies for Parallel Processing Workshop, IPSP '97*, 1997.
- [LH95] Johannes Lüthi and Günter Haring. Mean value analysis for queuing network models with intervals as input parameters. Technical Report TR-950101, Institute of applied science and information systems, university of Vienna, July 1995.

- [LH97] Johannes Lüthi and Günter Haring. Fuzzy queuing network models of computer systems. In *Proceedings of the 13th UK Performance Engineering Workshop*, 1997.
- [LLKS85] Lawler, Lenstra, Kan, and Shmoys. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.
- [LM86] Richard J. Larsen and Morris L. Marx. *An Introduction to Mathematical Statistics and Its Applications*. Prentice-Hall, 1986.
- [LMH96] Johannes Lüthi, Shikharesh Majumdar, and Günter Haring. Mean value analysis for computer systems with variabilities in workload. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium, IPDS '96*, 1996.
- [LMS⁺98] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications". In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, 1998.
- [LS93] S. Leuttenegger and X. Sun. Distributed computing feasibility in a non-dedicated homogeneous distributed system. Technical Report Technical Report Number 93-65, NASA - ICASE, 1993.
- [Lue98] Johannes Luethi. Histogram-based characterization of workload parameters and its consequences on model analysis. In *Proceedings of the Workshop on Workload Characterization in High-Performance Computing Environments, jointly with MASCOTS'98*, 1998.
- [MBC84] M. Ajmone Marson, G. Balbo, and G. Conte. A class of generalized stochastic petri nets for the performance analysis of multiprocessor systems. *ACM TOCS*, pages 93–122, May 1984.
- [MFS⁺95] C. R. Mechoso, J. D. Farrara, J. A. Spahr, K. Sklower, and M. Stonebraker. Development of an earth system model: A progress report. Technical report, Univ. of California, Los Angeles, April 1995.
- [ML90] V. W. Mak and S. F. Lundstrom. Predicting the performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*, pages 257–270, July 1990.
- [MLH95] Shikharesh Majumdar, Johannes Lüthi, and Günter Haring. Histogram-based performance analysis for computer systems with variabilities or uncertainties in workload. Technical Report Research Report SCE-95-22, Department of systems and computer engineering, Carleton University, Ottawa, Canada, November 1995.

- [MMFM93] C. R. Mechoso, C. Ma, J. D. Farrara, and R. W. Moore. Parallelization and distribution of a coupled ocean-atmosphere general circulation model. *Monthly Weather Review*, 121(7), July 1993.
- [Moh84] J. Mohan. *Performance of Parallel Programs: Model and Analyses*. PhD thesis, Carnegie Mellon University, July 1984.
- [Mol90] M.K. Molloy. Performance analysis using stochastic petri nets. *IEEE Transactions on Parallel and Distributed Systems*, C-31:913–7, September 1990.
- [mp] Solaris SunOS 5.5 man page. dis - object code disassembler.
- [MR95] Shikharash Majumdar and Ravathy Ramadoss. Interval-based performance analysis of computing systems. In *Proceedings of the Third International Workshop on Model Analysis and Simulation of Computer and Telecommunication Systems*, 1995.
- [MR98] Celso L. Mendes and Daniel A. Reed. Integrated compilation and scalability analysis for parallel systems. In *Proceedings of PACT '98*, 1998.
- [MRW92] A. Malony, D. Reed, and H. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4), July 1992.
- [MSW81] William Mendenhall, Richard L. Scheaffer, and Dennis D. Wackerly. *Mathematical Statistics with Applications*, pages 4–6. Duxbury Press, 1981.
- [NB92] P. Newton and J. C. Browne. The code 2.0 graphical parallel programming language. In *Proceedings of the ACM International Conference on Supercomputing*, July 1992.
- [Neu90] Arnold Neumaier. *Interval methods for systems of equations*, chapter Basic Properties of Interval Arithmetic. Cambridge University Press, 1990.
- [PM94] Jan Pedersen and John Moul. Determination of the structure of small protein fragments using torsion space monte carlo and genetic algorithm methods. In *Proceedings of Meeting on Critical Assessment of Techniques for Protein Structure Prediction, Asilomar Conference Center*, December 1994.
- [PRW94] A. T. Phillips, J. B. Rosen, and V. H. Walke. Molecular structure determination by convex global underestimation of local energy minima. Technical Report Tech Report UMSI 94/126, University of Minnesota Supercomputer Institute, 1994.

- [Ric95] John A. Rice. *Mathematical Statistics and Data Analysis, Second Edition*, page 134. Duxbury Press, 1995.
- [RSR97] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The autopilot performance-directed adaptive control system. *Future Generation Computer Systems, special issue (Performance Data Mining)*, 1997.
- [RSSH98] Emilia Rosti, Giuseppe Seazzi, Evgenia Smirni, and Mark S. Squillante. the impact of i/o on program behavior and parallel scheduling. In *Proceedings of SIGMETRICS*, 1998.
- [SB97a] Jennifer Schopf and Francine Berman. Performance prediction using intervals. Technical Report #CS97-541, UCSD, CSE Dept., 1997.
- [SB97b] Jennifer M. Schopf and Francine Berman. Performance prediction in production environments. Technical Report #CS97-558, UCSD, CSE Dept., 1997.
- [SBSM89] R. Saavedra-Barrera, A. J. Smith, and E. Miya. Machine characterization based on an abstract high-level language machine. *IEEE Transactions on Computers*, 38:1659–1679, December 1989.
- [Sch97] Jennifer M. Schopf. Structural prediction models for high-performance distributed applications. In *CCC '97*, 1997. Also available as www.cs.ucsd.edu/users/jenny/CCC97/index.html.
- [Sco79] D.W. Scott. On optimal and data-based histograms. *Biometrika*, 66, 1979.
- [Sev89] Kenneth C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. *Performance Evaluation and Review*, 17(1):171–180, May 1989.
- [Sil] Silicon Graphics. *Pixie Man Page*. also <http://web.cnam.fr/Docs/man/Ultrix-4.3/pixie.1.html>.
- [SK95] Steffen Schulze-Kremer. *Molecular Bioinformatics - Algorithms and Applications*. Walter de Gruyter, Berlin, 1995. Also available at <http://www.techfak.uni-bielefeld.de/bcd/Curric/ProtEn/proten.html>.
- [Ske74] Stig Skelboe. Computation of rational integer functions. *BIT*, 14, 1974.
- [SL90a] Gilbert C. Sih and Edward A. Lee. Dynamic-level scheduling for heterogeneous processor networks. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Systems*, 1990.

- [SL90b] Gilbert C. Sih and Edward A. Lee. Scheduling to account for inter-processor communication within interconnection-constrained processor networks. In *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.
- [SL93] Gilbert C. Sih and Edward A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2), 1993.
- [SRar] Evgenia Smirni and Daniel A. Reed. workload characterization of input/output intensive parallel applications. *Performance Evaluation*, to appear.
- [SSLP93] Jonathan Schaeffer, Duane Szafron, Greg Lobe, and Ian Parsons. The enterprise model of developing distributed applications. *IEEE Parallel and Distributed Technology*, 1(3), August 1993.
- [Stu26] H. A. Sturges. *Journal of American Statistical Ass.*, 21, 1926.
- [SW96a] Jens Simon and Jens-Michael Wierum. Accurate performance prediction for massively parallel systems and its applications. In *Proceedings of Euro-Par '96 (Second International European Conference on Parallel Processing)*, volume 2, pages 675–88, 1996.
- [SW96b] Jens Simon and Jens-Michael Wierum. Performance prediction of benchmark programs for massively parallel architectures. In *Proceedings of the 10th Annual Intl. Conf. on High-Performance Computers, (HPCS 96)*, 1996.
- [SW98a] J. Simon and J. M. Wierum. The latency-of-data-access model for analyzing parallel computation. *information processing letters*, 66(5), june 1998.
- [SW98b] Neil Spring and Rich Wolski. Application level scheduling of gene sequence comparison on metacomputers. In *Proceedings of the 12th ACM International Conference on Supercomputing*, 1998.
- [TB86] A. Thomasian and P. F. Bay. Analysis queuing network models for parallel processing of task systems. *IEEE Transactions on Computers c-35*, 12:1045–1054, December 1986.
- [TK93] Howard M. Taylor and Samuel Karlin. *An Introduction to Stochastic Modeling*. Academic Press, 1993.
- [Ver] A. Verstraete. Business information systems. www.smeal.psu.edu/misweb/infosys/ibismain.html.

- [Wal90] David W. Wall. Predicting program behavior using real or estimated profiles. Technical Report WRL TN-18, Digital Western Research Laboratory, 1990.
- [WH94] H. Wabnig and G. Haring. Paps - the parallel program performance prediction toolset. In *Proceedings of the 7th international conference on computer performance evaluation: modeling techniques and tools*, 1994.
- [Wil95] Gregory V. Wilson. *Practical Parallel Programming*. The MIT Press, 1995.
- [WK93] Yi-Shuen Mark Wu and Aron Kupermann. Prediction of the effect of the geometric phase on product rotational state distributions and integral cross sections. *Chemical Physics Letters*, 201:178–86, January 1993.
- [Wol96] Rich Wolski. Dynamically forecasting network performance using the network weather service(to appear in the journal of cluster computing). Technical Report TR-CS96-494, UCSD, CSE Dept., 1996.
- [Wol97] R. Wolski. Dynamically forecasting network performance to support dynamic scheduling using the network weather service. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, August 1997.
- [Wol98] Rich Wolski. Personal communication, 1998.
- [WSF89] D. Whitley, T. Starkweather, and D'Ann Fuquay. Scheduling problems and traveling salesman: The genetic edge recombination operator. In *Proceedings of International Conference on Genetic Algorithms*, 1989.
- [WSH98] Rich Wolski, Neil Spring, and Jim Hayes. Predicting the cpu availability of time-shared unix systems. In *submitted to SIGMETRICS '99 (also available as UCSD Technical Report Number CS98-602)*, 1998.
- [WSP97] R. Wolski, N. Spring, and C. Peterson. Implementing a performance forecasting system for metacomputing: The network weather service. In *SC97*, November 1997.
- [WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating System Design and Implementation*, November 1994.
- [WZ98] Jon B. Weissman and Xin Zhao. Scheduling parallel applications in distributed networks. *Journal of Cluster Computing*, 1998.

- [YB95] W. Young and C. L. Brooks. Dynamic load balancing algorithms for replicated data molecular dynamics. *Journal of Computational Chemistry*, 16:715–722, 1995.
- [YZS96] Yong Yan, Xiaodong Zhang, and Yongsheng Song. An effective and practical performance prediction model for parallel computing on non-dedicated heterogeneous now. *Journal of Parallel and Distributed Computing*, October 1996. Also University of Texas, San Antonio, Technical Report # TR-96-0401.
- [ZLP96] Mohammed J. Zaki, Wei Li, and Srinivasan Parthasarathy. Customized dynamic load balancing for network of workstations. In *Proceedings of HPDC '96*, 1996.
- [ZY95] X. Zhang and Y. Yan. A framework of performance prediction of parallel computing on non-dedicated heterogeneous networks of workstations. In *Proceedings of 1995 International Conference of Parallel Processing*, 1995.