Performance Modeling Based Scheduling and Rescheduling of Parallel Applications on Computational Grids

A Thesis
Submitted for the Degree of
Doctor of Philosophy
in the Faculty of Engineering

By
H. A. Sanjay

Supercomputer Education and Research Centre
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

October 2008

Dedication

This thesis is dedicated

To the sweet memory of my father-in-law

H.L. Rajappa Gowda(1955-2009)

And

To my Family

Acknowledgments

I would like to thank all people who have helped and inspired me during my doctoral study.

First of all I would like to express my deep and sincere gratitude to my research supervisor, Dr. Sathish Vadhiyar for his outstanding level of knowledge , constant understanding, encouraging , personal guidance and for having led me into a fascinating area of research. He was always approachable(24X7) , supportive, caring and ready to help in any sort of problem. I thank him for his extreme patience and excellent technical guidance in writing and presenting research. I learnt many lessons from him other than research i.e. fairness in evaluating the students and style of working at the time of paper deadlines. Finally, he was and continues to be my role model for his hard work and passion for research.

I am thankful to Prof. R. Govindarajan, Prof.Mathew Jacob and Prof. V. Mani for their valuable feedback, which helped me to improve the dissertation in many ways. I am also thankful to all professors, technical Staff and non-technical staff of SERC for their care and attention.

I would like to thank Management of my institute, J.N.N. College of Engineering who encouraged and deputed me for Ph.D.

I would like to thank my lab buddies Rakhi, Sandip , Antoine, Yadnyesh, Sivagama sundari, Raghavendra, Sachin, Pinaki, Senthil, Karthikeyan and Roshan for making it a convivial place to work. Special thanks to Raghavendra and Sachin for their technical support during the final stage of work. Special thanks to Rakhi and Antoine for useful insights and many interesting conversations about our research.

I would like to thank my IISc friends Venkatesh, Surendra, Manohar, Deepak , Rupesh and Hari Prasad for their care and great time at Tea-Board. I would also like to thank my friends Prof. Shantharajappa A. N. (He is also my teacher), Prof Vijaykumar. B. P., Jayachandra. A. N

and Sanjeev Kunte for their technical and emotional support.

I am grateful to my friend, caretaker, teacher and ofcourse my guru C. Suresh for his moral support and inspiring words which led me into the research path.

I would like to thank my family for all their love and encouragement. For my parents who raised me and have been a constant source of support emotional, moral and of course financial. For my father-in-law and mother-in-law for their encouragement and support. For my sister, brother and in-laws for always being there for me. Also I am thankful to all my family members for their understanding and encouragement.

I would like to express my deep love to my sweet little daughter Saanvi, who cheered me whenever I was frustrated during the course of my entire graduate studies.

I would like to thank God the most. You have made my life more bountiful. May your name be exalted, honored, and glorified.

Finally, I want my wife Sanchi to know that none of this would have been possible without you. All I can say is it would take another thesis to express my deep love for you. Your patience, love and encouragement have upheld me, particularly in those many days in which I spent more time with my computer than with you.

Abstract

As computational grids have become popular and ubiquitous, users have access to large number and different types of geographically distributed grid resources. Many computational grid frameworks are composed of multiple distributed sites with each site consisting of one or more dedicated or non-dedicated clusters. Jobs submitted to a grid are handled by a matascheduler which interacts with the local schedulers of the clusters for scheduling jobs to the individual clusters. Computational grids have been found to be powerful research-beds for execution of various kinds of parallel applications. When a parallel application is submitted to a grid, the metascheduler has to choose a set of resources from a cluster for application execution. To select the best set of resources for application execution, it is important to determine the performance of the application. Accurate performance estimates of an application is essential in assisting a grid meta scheduler to efficiently schedule user jobs.

Thus models that predict execution times of parallel applications on a set of resources and a search procedure (scheduling strategy) which selects the best set of machines within a cluster for application execution are of importance for enabling the parallel applications on grids. For efficient execution of large scientific parallel applications consisting of multiple phases, performance models of the individual phases should be obtained. Efficient rescheduling strategies that can use the per-phase models to adapt the parallel applications to application and resource dynamics are necessary for maintaining high performance of the applications on grids. A practical and robust grid computing infrastructure that integrates components related to application and resource monitoring, performance modeling, scheduling and rescheduling techniques, is highly essential for large-scale deployment and high performance of scientific applications on grid systems and hence for fostering high performance computing.

This thesis focuses on developing performance models for predicting execution times of parallel problems/subproblems on dedicated and non-dedicated grid resources. The thesis also constructs robust scheduling and rescheduling strategies in a grid metascheduler that can use the performance models for efficient execution of large scientific parallel applications on dynamic grids. Finally, the thesis builds a practical and robust grid middleware infrastructure which integrates components related to performance modeling, scheduling and rescheduling, monitoring and migration frameworks for large-scale deployment and use of high performance applications on grids.

The thesis consists of four main components. In the first part of the thesis, we have developed a comprehensive set of performance modeling strategies to predict the execution times of tightly-coupled parallel applications on a set of resources in a dedicated or non-dedicated cluster. The main purpose of our prediction strategies is to aid grid metaschedulers in making scheduling decisions. Our performance modeling strategies, based on linear regression, can deal with non-dedicated systems where the loads can change during application executions. Our models do not require detailed knowledge and instrumentation of the applications and can be constructed without the involvement of application developers. The strategies are intended for rapid and large scale deployment of parallel applications on non-dedicated grid systems. We have evaluated our strategies on 8, 16, 24 and 32-node clusters with random loads and load traces from a grid system. Our performance modeling strategies gave less than 30% average percentage prediction errors in all cases, which is reasonable for non-dedicated systems. We also found that scheduling based on the predictions by our strategies will result in perfect scheduling in many cases. For modeling large-scale scientific applications, we use execution profiles and automatic program analysis, and manual analysis of significant portions of the application's code to identify the different phases of applications. We then adopt our performance modeling strategies to predict execution times for the different phases of the tightly-coupled parallel applications on a set of resources in a dedicated or non-dedicated cluster. Our experiments show that using combinations of performance models of the phases give 18% – 70% more accurate predictions than using single performance models for the applications.

In the second part of the thesis, we have devised, evaluated and compared algorithms for

scheduling tightly-coupled parallel applications on multi-cluster grids. Our algorithms use performance models that predict the execution times of parallel applications, for evaluations of candidate schedules. In this work, we propose a novel algorithm called Box Elimination (BE) that searches a space of performance model parameters to determine efficient schedules. By eliminating large search space regions containing poorer solutions at each step and searching high quality solutions, our algorithm is able to generate efficient schedules within few seconds for even clusters of 512 processors. By means of large number of real and simulation experiment, we compared our algorithm with popular optimization techniques. We show that our algorithm generates up to 80% more efficient schedules than other algorithms and the resulting execution times are more robust against performance modeling errors.

The third part of the thesis deals with policies for rescheduling long-running multi-phase parallel applications in response to application and resource dynamics. In this work, we use our performance modeling and scheduling strategies to derive rescheduling plans for executing multi-phase parallel applications on grids. A rescheduling plan consists of potential points in application execution for rescheduling and schedules of resources for application execution between two consecutive rescheduling points. We have developed three algorithms, namely an incremental algorithm, a divide-and-conquer algorithm and a genetic algorithm, for deriving a rescheduling plan for a parallel application execution. We have also developed an algorithm that uses rescheduling plans derived on different clusters to form a single coherent rescheduling plan for application execution on a grid consisting of multiple clusters. The rescheduling plans generated by our algorithms are highly efficient leading to application execution times that are higher than the execution times corresponding to brute force method by less than 10%. We also find that rescheduling in response to changing application and resource dynamics, using the rescheduling plans for multi-cluster grids generated by our algorithms, give much lesser execution times when compared to executions of the applications on a single schedule throughout application execution.

In the final part of the thesis, we have developed a practical grid middleware framework called **MerITA** (**M**iddleware for P**er**formance **I**mprovement of **T**ightly Coupled Parallel **A**pplications on Grids), a system for effective execution of tightly-coupled parallel appli-

cations on multi-cluster grids consisting of dedicated or non-dedicated, interactive or batch systems. The framework brings together performance modeling for automatically determining the characteristics of parallel applications, scheduling strategies that use the performance models for efficient mapping of applications to resources, rescheduling policies for determining the points in application execution when executing applications can be rescheduled to different sets of resources to obtain performance improvement and a check-pointing library for enabling rescheduling.

Publications

- "Performance Modeling based on Multidimensional Surface Learning for Performance Predictions of Parallel Applications in Non-Dedicated Environments"
  J. Yagnik, H.A. Sanjay, S. Vadhiyar
  In proceedings of 35th International Conference on Parallel Processing (ICPP), pp 513-520, August 2006, Columbus, Ohio, USA.

- "Performance Modeling Based Scheduling and Rescheduling of Parallel Applications on Computational Grids"
  H.A. Sanjay, S. Vadhiyar
  Poster presentation at International Conference on High Performance Computing (HiPC), Goa, India, 2007

- "Performance Modeling of Parallel Applications for Grid Scheduling"
  H.A. Sanjay, S. Vadhiyar
  Journal of Parallel and Distributed Computing , 68(8), 2008, 1135–1145.

- "Strategies for Scheduling Tightly-Coupled Parallel Applications on Clusters and Grids"
  H.A. Sanjay, S. Vadhiyar
  Submitted to Concurrency and Computation: Practice and Experience.

- "Strategies for Rescheduling Tightly-Coupled Parallel Applications in Multi-Cluster Grids"
  H.A. Sanjay, S. Vadhiyar
  To be Submitted.

- "A Framework for Adaptive Execution and Performance Enhancement of Tightly Coupled Parallel Applications on Grids"
  H.A. Sanjay, S. Vadhiyar
  Paper Under Preparation .

Contents

Abstract **4**

List of Figures **14**

List of Tables **20**

1 Introduction **22**

1.1 Challenges in Grid computing 23

1.2 Motivation 25

1.3 Problem Statement 27

1.4 Performance Modeling of Parallel Applications 27

1.5 Scheduling Strategies 30

1.6 Rescheduling strategies 32

1.7 A Grid Framework for Tightly Coupled Parallel Applications 33

1.8 Organization of Thesis 34

2 Related Work **37**

2.1 Performance Modeling of Parallel Applications 37

2.2 Scheduling 43

2.3 Rescheduling 46

2.4 Grid Computing Software Infrastructure 48

3 Performance Modeling of Tightly Coupled Parallel Applications **52**

3.1 Introduction 52

3.2 Overview of Modeling Strategy 54
3.3 Preliminary Work 56
3.3.1 Methodology 56
3.3.2 A Rational Polynomial Model 58
3.3.3 Generic Model Based on Rational Polynomials 61
3.3.4 Experiments and Observations 62
3.4 Comprehensive Performance Modeling Strategies 63
3.4.1 Modeling Equation and Terms 63
3.4.2 Model Correctness 65
3.4.3 General Methodology 66
3.4.4 Evaluation of Models 67
3.4.5 Modeling Computation 67
3.4.6 Modeling Communication 68
3.4.7 Modeling Scalability 69
3.4.8 Modeling Applications with Processor Constraints 70
3.4.9 Prediction of Execution Times 71
3.5 Cross-Platform Performance Modeling 72
3.6 Performance Models for Multi-Phase Applications 75
3.7 Usage Scenario: Putting It Altogether 75
3.8 Experiment Setup 77
3.9 Results 84
3.9.1 Comparison with a Modeling Strategy for Non-Dedicated Systems . . . 89
3.9.2 Assuming Uniform Loading Conditions 90
3.9.3 Modeling Overheads 92
3.9.4 Cross-Platform Performance Predictions 94
3.10 Experiments and Results for Modeling Multi-Phase Parallel Applications . . . 95
3.11 Summary 98

4 Scheduling Tightly-Coupled Parallel Applications on Clusters and Grids 101

4.1 Introduction 101

4.2 Algorithms . . . 102
4.2.1 Simulated Annealing . . . 103
4.2.2 Genetic Algorithm . . . 104
4.2.3 Ant Colony Optimization . . . 106
4.2.4 Branch and Bound . . . 109
4.2.5 Dynamic Programming . . . 111
4.2.6 Box Elimination . . . 111
4.2.7 Correctness . . . 118
4.2.8 Complexity . . . 119
4.2.9 Comparison with Other Algorithms . . . 120
4.3 Experiments and Results . . . 120
4.3.1 Real Experiments . . . 120
4.3.2 Simulation Setup . . . 124
4.4 Summary . . . 143

5 Rescheduling Strategies 147
5.1 Introduction . . . 147
5.2 Problem Statement . . . 150
5.3 Algorithms for Generating Rescheduling Plans in a Single Cluster . . . 153
5.3.1 Incremental Algorithm . . . 153
5.3.2 Division Heuristic . . . 156
5.3.3 Genetic Algorithm . . . 158
5.4 Rescheduling Plans for Multi-Cluster Grids . . . 159
5.5 Experiments . . . 162
5.5.1 Applications . . . 162
5.5.2 Results . . . 164
5.6 Discussion . . . 171
5.7 Summary . . . 172

6 A Grid Computing Framework **173**

6.1 Introduction . . . 173

6.2 MerITA Architecture . . . 175

6.3 Performance Modeler . . . 176

6.3.1 Cross-Platform Performance Modeling . . . 178

6.3.2 Prediction of Execution Times . . . 178

6.3.3 Performance Models for Multi-Phase Applications . . . 179

6.4 Scheduler . . . 179

6.5 Rescheduler . . . 180

6.5.1 Reschedule Planner . . . 181

6.5.2 Enabling Application Rescheduling . . . 182

6.6 MerITA Interactions - The Big Picture . . . 184

6.7 Discussion . . . 188

6.7.1 Interaction with local schedulers . . . 188

6.7.2 Standard Grid Mechanisms . . . 189

6.8 Summary . . . 190

7 Conclusions and Future Work **191**

7.1 Conclusions . . . 191

7.2 Future Work . . . 193

8 Appendix **194**

8.1 More Results on Performance Modeling . . . 194

8.1.1 More Results on Cumulative Performance Models . . . 211

8.2 More Results on Scheduling Strategies . . . 211

8.3 More Results on Rescheduling Strategies . . . 222

References **224**

List of Figures

1.1 Multi-cluster Environment 25

3.1 Predictions with Rational Polynomial based Model 60
3.2 Illustration of Modeling Procedure 73
3.3 Available CPU under Random Loading Conditions 80
3.4 Available Bandwidths under Random Loading Conditions 81
3.5 Available CPU under Random Loading Conditions 82
3.6 Available Bandwidths under Random Loading Conditions 82
3.7 Actual Execution Times for ScaLAPACK Eigen Value Problem on the Intel Cluster with Random Loading 84
3.8 Percentage Prediction Errors (PPE) for ScaLAPACK Eigen Value Problem on the Intel Cluster with Random Loading 85
3.9 Percentage Prediction Errors (PPE) at Different Times for ScaLAPACK Eigen Value Problem on the Intel Cluster with Random Loading 86
3.10 Comparison of Our Adaptive Method and Schopf' Method for 4 Processors with SSOR on the Intel Cluster with Random Loading; Schopf' Method - Average: 37.25%, Standard Deviation: 18.09%; Our Method - Average: 14.39%, Standard Deviation: 14.11% 90
3.11 Comparison of Our Adaptive Method and Schopf' Method for 8 Processors with SSOR on the Intel Cluster with Random Loading; Schopf' Method - Average: 45.17%, Standard Deviation: 14.18%; Our Method - Average: 17.39%, Standard Deviation: 13.49% 91

3.12 Percentage Prediction Errors with Single and Cumulative Models for Athena . 98
3.13 Percentage Prediction Errors with Single and Cumulative Models for ChaNGa . 99

4.1 Simulated Annealing (SA) 105
4.2 Genetic Algorithm (GA) 107
4.3 Ant Colony Optimization (ACO) 108
4.4 Branch and Bound (B&B) 110
4.5 Dynamic Programming (DP) 112
4.6 Box Elimination (BE) 115
4.7 3-D box of (cpu, bandwidth, processors) tuples for Box Elimination 116
4.8 FindScheduleWithConstraints() 117
4.9 Comparison of Algorithms in Terms of Actual and Predicted Execution Times for Different Times Available for Scheduling on 104 Cores of Intel Xeon Cluster. Application: MD with 2400 molecules. 122
4.10 Effect of Loads on Machines for 256 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds 126
4.11 Effect of Loads on Machines for 512 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds 127
4.12 Effect of Increasing Lightly Loaded Machines for 512 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds 129
4.13 Effect of Increasing Moderately Loaded Machines for 512 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds 130
4.14 Effect of Increasing Heavily Loaded Machines for 512 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds 131
4.15 Scalability With Processors. Application: MD with 2048 molecules, 30% Lightly , 40% Medium and 30% Highly Loaded. Time for Scheduling: 25 seconds 132
4.16 Scheduling Different Applications on 256 processors. Problem Sizes: 2048 (MD), 10000 (Eigen), 3328 (SSOR), 400000 (IS). 30% Lightly , 40% Medium and 30% Highly Loaded. Time for Scheduling: 25 seconds 134

4.17 Scheduling Different Applications on 512 processors. Problem Sizes: 2048 (MD), 10000 (Eigen), 3328 (SSOR), 400000 (IS). 30% Lightly , 40% Medium and 30% Highly Loaded. Time for Scheduling: 25 seconds 135
4.18 Performance of Algorithms With Different Times Available for Scheduling for 256 processors. Application: MD with 2048 molecules, 30% Lightly , 40% Medium and 30% Highly Loaded. 136
4.19 Performance of Algorithms With Different Times Available for Scheduling for 512 processors. Application: MD with 2048 molecules, 30% Lightly , 40% Medium and 30% Highly Loaded. 137
4.20 Effect of Prediction Errors on Execution Times for 256 processors. Application: MD with 2048 molecules, 30% Lightly , 40% Medium and 30% Highly Loaded, Time for Scheduling: 25 seconds. 138
4.21 Effect of Prediction Errors on Percentage Increase in Execution Times for 256 processors. Application: MD with 2048 molecules, 30% Lightly , 40% Medium and 30% Highly Loaded, Time for Scheduling: 25 seconds. 139
4.22 Effect of Prediction Errors on Execution Times for 512 processors. Application: MD with 2048 molecules, 30% Lightly , 40% Medium and 30% Highly Loaded, Time for Scheduling: 25 seconds. 140
4.23 Effect of Prediction Errors on Percentage Increase in Execution Times for 512 processors. Application: MD with 2048 molecules, 30% Lightly, 40% Medium and 30% Highly Loaded, Time for Scheduling: 25 seconds. 141
4.24 Effect of Cluster Heterogeneity in a Multi-Cluster Setup 144
4.25 Effect of Increasing Maximum Capacities in a Multi-Cluster Setup 145

5.1 Phases and Intervals 152
5.2 Incremental Algorithm (IA) 155
5.3 Illustration of Division Heuristic 157
5.4 Chromosome Representation of Rescheduling Plan 158
5.5 Genetic Algorithm (GA) 160
5.6 Algorithm for Generating Multi Cluster Rescheduling Plan 163

5.7 Comparison of the Algorithms with Brute Force Method (MD) 166
5.8 Comparison of the Algorithms with Brute Force Method for Generation of Rescheduling Plans (LAMMPS) 167
5.9 Effect of Cluster Heterogeneity in a Multi-Cluster Setup 170
5.10 Effect of increasing average number of machines per cluster in a multi-cluster grid setup 171

6.1 MerITA Framework 176
6.2 MerITA's Performance Modeling Interaction 177
6.3 MerITA Interactions 185

8.1 Percentage Prediction Errors for ScaLAPACK Eigen Value Problem on the Intel Cluster with Grads Loading 195
8.2 Percentage Prediction Errors at Different Times for ScaLAPACK Eigen Value Problem on the Intel Cluster with Grads Loading 195
8.3 Percentage Prediction Errors for ScaLAPACK Eigen Value Problem on the AMD Cluster with Random Loading 196
8.4 Percentage Prediction Errors at Different Times for ScaLAPACK Eigen Value Problem on the AMD Cluster with Random Loading 197
8.5 Percentage Prediction Errors for ScaLAPACK Eigen Value Problem on the AMD Cluster with GrADS Loading 197
8.6 Percentage Prediction Errors for FFT on the Intel Cluster with Random Loading 198
8.7 Percentage Prediction Errors at Different Times for FFT on the Intel Cluster with Random Loading 199
8.8 Percentage Prediction Errors for CG on the Intel Cluster with Random Loading 199
8.9 Percentage Prediction Errors at Different Times for CG on the Intel Cluster with Random Loading 200
8.10 Percentage Prediction Errors for MD on the Intel Cluster with Random Loading 200
8.11 Percentage Prediction Errors at Different Times for MD on the Intel Cluster with Random Loading 201

8.12 Percentage Prediction Errors for MD on the AMD Cluster with Random Loading 202
8.13 Percentage Prediction Errors at Different Times for MD on the AMD Cluster with Random Loading 202
8.14 Percentage Prediction Errors for MD on the Intel Cluster with GrADS Loading 203
8.15 Percentage Prediction Errors for MD on the AMD Cluster with GrADS Loading 204
8.16 Percentage Prediction Errors for Poisson Solver on the Intel Cluster with Random Loading 204
8.17 Percentage Prediction Errors at Different Times for Poisson Solver on the Intel Cluster with Random Loading 205
8.18 Percentage Prediction Errors for Poisson Solver on the Woodcrest Cluster with Random Loading 205
8.19 Percentage Prediction Errors at Different Times for Poisson Solver on the Woodcrest Cluster with Random Loading 206
8.20 Percentage Prediction Errors for Integer Sort on the Intel Cluster with Random Loading 207
8.21 Percentage Prediction Errors at Different Times for Integer Sort on the Intel Cluster with Random Loading 207
8.22 Percentage Prediction Errors for Integer Sort Application on the Intel Cluster with GrADS Loading 209
8.23 Percentage Prediction Errors for Integer Sort Application on the AMD Cluster with GrADS Loading 211
8.24 Percentage Prediction Errors for Integer Sort on the Woodcrest Cluster with Random Loading 212
8.25 Percentage Prediction Errors at Different Times for Integer Sort on the Woodcrest Cluster with Random Loading 212
8.26 Percentage Prediction Errors for SSOR on the Intel Cluster with Random Loading 213
8.27 Percentage Prediction Errors at Different Times for SSOR on the Intel Cluster with Random Loading 213
8.28 Percentage Prediction Errors for SSOR on the Intel Cluster with Grads Loading 214

8.29 Percentage Prediction Errors at Different Times for SSOR on the Intel Cluster with Grads Loading 214
8.30 Percentage Prediction Errors for SSOR Application on the AMD Cluster with GrADS Loading 215
8.31 Comparison of Our Adaptive Method and Schopf' Method for 2 Processors with SSOR on the Intel Cluster with Random Loading; Schopf' Method - Average: 27.26%, Standard Deviation: 18.30%; Our Method - Average: 19.36%, Standard Deviation: 27.26% 216
8.32 Comparison of Our Adaptive Method and Schopf' Method for 6 Processors with SSOR on the Intel Cluster with Random Loading; Schopf' Method - Average: 39.55%, Standard Deviation: 15.21%; Our Method - Average: 14.41%, Standard Deviation: 11.04% 217
8.33 Percentage Prediction Errors with Single and Cumulative Models for MD . . . 217
8.34 Percentage Prediction Errors with Single and Cumulative Models for MPB . . . 218
8.35 Percentage Prediction Errors with Single and Cumulative Models for LAMMPS 218
8.36 Effect of Loads on Machines for 64 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds 219
8.37 Effect of Loads on Machines for 128 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds 219
8.38 Effect of Loads on Machines for 1024 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds 220
8.39 Effect of Increasing Lightly Loaded Machines for 256 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds 220
8.40 Effect of Increasing Moderate Loaded Machines for 256 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds 221
8.41 Effect of Increasing Heavily Loaded Machines for 256 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds 222
8.42 Comparison of the Algorithms with Brute Force Method for Generation of Rescheduling Plans (ChaNGa) 223

List of Tables

3.1 Candidate functions for f_{bw} . . . 68
3.2 Candidate functions for f_{Pcomp} and f_{Pcomm} . . . 69
3.3 Experiment Infrastructure . . . 78
3.4 Usefulness of Predictions for Scheduling of Eigen Value Problem on Intel Cluster with Random Loading . . . 87
3.5 Predictions on the 8-processor Intel Cluster . . . 88
3.6 Predictions on 16-processor AMD and 24-processor Woodcrest (WC) . . . 88
3.7 Results with grads Loading - Assuming Uniform Loading Conditions during Application Executions . . . 92
3.8 Comparison of Modeling Overheads . . . 93
3.9 Cross-Platform Predictions on the 8-processor Intel, 16-processor AMD and 32-processor IBM Clusters . . . 96
3.10 Prediction Accuracy with Cumulative Models . . . 97

4.1 Multi Cluster Grid Setup . . . 143

5.1 Comparison of the Algorithms with Brute Force Method for Generation of Rescheduling Plans (MD) . . . 165
5.2 Comparison of the Algorithms with Brute Force Method for Generation of Rescheduling Plans (LAMMPS) . . . 166
5.3 Multi Cluster Grid Setup . . . 169

8.1 Usefulness of Predictions for Scheduling of MD on Intel Cluster with Random Loading 201
8.2 Usefulness of Predictions for Scheduling of Integer Sort on Intel Cluster with Random Loading 208
8.3 Usefulness of Predictions for Scheduling of SSOR on Intel Cluster with Random Loading 210
8.4 Comparison of the Algorithms with Brute Force Method for Generation of Rescheduling Plans (ChaNGa) 223

Chapter 1

Introduction

Computational approaches to problem solving have been widely used in various fields including high energy physics, earth system sciences, astronomy, geosciences, archeology, bioinformatics, biomedical science and financial modeling. Computers are used for modeling and simulating complex scientific and engineering problems, diagnosing medical conditions, controlling industrial equipment, forecasting the weather, managing stock portfolios, and many other purposes. In order to meet the ever increasing computational demands of many of these applications, powerful parallel systems with increasing number of processors and multi-core solutions are being built. The world's top parallel system as maintained by the Top500[135] project has 13124 cores. In spite of the huge amount of parallelism currently available in a single institution or site, few applications like parameter search problems are always in need of more resources. This has motivated the development of a computing paradigm called grid computing.

A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities[57]. Computational grids are geographically distributed platforms for computation, accessible to their users via a single interface. They provide computational power beyond the capacity of even the largest parallel computer system, and merge extremely heterogeneous physical resources into a single virtual resource. Hence grids have been found to be powerful research-beds for executing various kinds of applications[8, 16, 111, 20, 13, 40, 52, 51, 70, 75]. The applications include molecular modeling for drug design[49], brain activity analysis[105], particle physics[113],

geodise: aerospace design optimization[71], GECEM : Grid-Enabled Computational Electromagnetics[68], GeoFEM : multi-purpose/multi-physics parallel finite simulator[72], etc. The success of early grid prototypes in solving complex scientific problems have lead to the development of many grid infrastructures. Examples include GARUDA - Indian National Grid [67], Asia Pacific Grid [12], TeraGrid in US [134] and e-Science in UK [137].

Computational grids have been found to be powerful research-beds for execution of various kinds of parallel applications[111, 16, 8]. For these applications, grids have been used to provide feasibility to solve much larger problem sizes than cannot be solved on a single site[36, 102, 47, 53, 109, 130], explore large search space of different parameters of a problem[9, 65, 84], improve the quality of the solutions[128, 77] and enhance application performance[36, 102, 47].

In spite of the success of grids in solving scientifically significant problems, grid computing involves large number of challenges.

1.1 Challenges in Grid computing

Though grids have been successfully used for executing parallel applications, current grid systems and environments present many challenges to large-scale deployment, efficient execution and effective usage of large number of parallel scientific applications.

One of the foremost challenges is efficient scheduling of parallel applications on grid resources. Grids can consist of non-dedicated resources in which the external loads can vary over time. When these non-dedicated resources are used for the grid applications, the performance of the parallel applications may be impacted in dynamic and often unpredictable ways. Efficient scheduling of parallel applications on these non-dedicated grid resources involves understanding the computation and communication characteristics of the application in the presence of resource dynamics. This requires robust and accurate **performance modeling** of the parallel applications. For enabling large-scale integration of parallel applications in grids, the performance modeling strategy should involve small number of executions of the applications and should not require the intervention of the application developers.

Another challenge is the development of efficient **scheduling** strategies for mapping the

parallel applications to heterogeneous and non-dedicated grid resources[39]. The scheduling methods should be able to effectively use the application performance models to select the resources with the characteristics required by the applications. Many scheduling strategies for traditional parallel and distributed systems have been proposed for homogeneous and dedicated systems. Most of the existing scheduling strategies for grid systems use only the system characteristics for mapping the applications. Few methods that also use the application characteristics do not effectively use the application parameters.

Large scientific applications contain multiple phases where the computation and communication characteristics of the applications can vary between the different phases of application execution[33, 101]. Scheduling such applications on grid resources is challenging due to the dynamics exhibited by both the applications and the resources. Determining the major phases of such applications is important for effective scheduling. The applications will have to be rescheduled in response to the application and/or resource dynamics. **Rescheduling strategies** will have to be developed to reschedule such applications between the successive phases that require different resource characteristics. The rescheduling methods will have to taken into account the overhead of migration of applications between the resources during phase changes.

Finally, a practical **grid middleware** that incorporates various techniques for effective execution of large scientific parallel applications has to be built. The grid middleware should include the following:

- periodic monitoring of characteristics of both the non-dedicated time-shared resources and dedicated batch systems,
- problem solving environment (PSE) for easy integration of applications,
- automatic identification of application integration and subsequent execution of application benchmarks to obtain information for performance modeling,
- performance modeling with the obtained information,
- maintenance of the performance modeling parameters,
- identification and periodic updates of the major phases of the applications,

- determining effective schedules for the phases based on periodic resource information, phase characteristics and the overhead costs for migrating between the phases, and
- an effective migration infrastructure for migrating between the phases.

1.2 Motivation

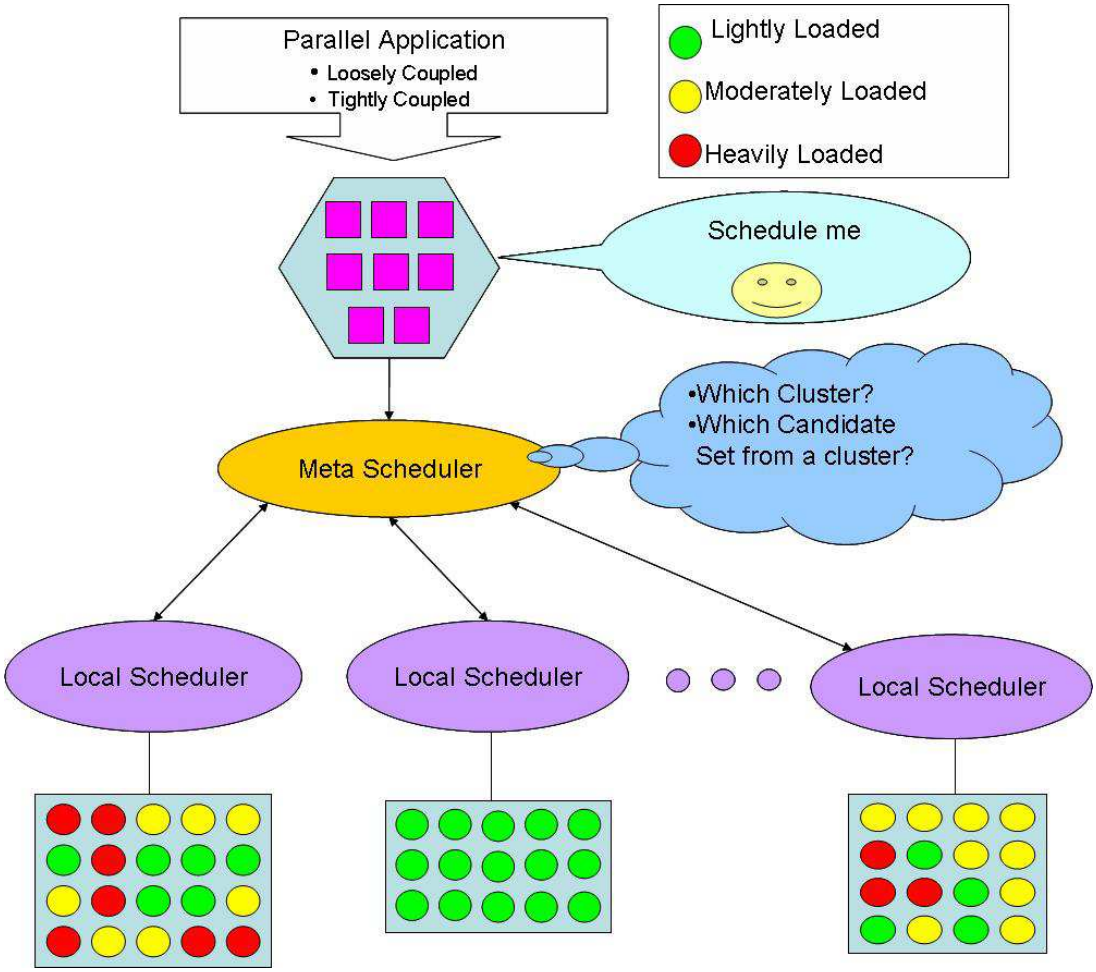


Figure 1.1: Multi-cluster Environment

As the computational grids have become popular and ubiquitous, users have access to large number and different types of geographically distributed grid resources. To select the best set of resources for their application, it is important to determine the performance of the application.

Accurate performance estimates of an application is essential in assisting a grid meta scheduler to efficiently schedule user jobs.

Many computational grid frameworks [67, 12, 134, 137] are composed of multiple distributed sites as shown in Figure 1.1. Each site has one or more clusters or Massively Parallel Processors (MPPs) with each cluster consisting of a set of homogeneous machines. Some of these clusters are dedicated batch systems while others are time-shared non-dedicated systems. Each cluster is controlled by a local scheduler for job and resource management. Jobs submitted to a grid are handled by a matascheduler which interacts with the local schedulers of the clusters for scheduling jobs to the individual clusters.

When a parallel application is submitted to a grid, the metascheduler has to choose a set of resources from a cluster for application execution. The metascheduler, with the help of local schedulers, has to evaluate different candidate resource sets, with each candidate set consisting of resources from a cluster, and select the most suitable resources for application execution. The candidate resource sets are mostly evaluated in terms of predicted execution times of the application on the candidate resources and the resource set with the minimum predicted execution time is chosen for application execution[111].

Thus models that predict execution times of parallel applications on a set of resources and a search procedure (scheduling strategy) which selects the best set of machines within a cluster for application execution are of importance for enabling the parallel applications on grids. For efficient execution of large scientific parallel applications consisting of multiple phases, performance models of the individual phases should be obtained. Efficient rescheduling strategies that can use the per-phase models to adapt the parallel applications to application and resource dynamics are necessary for maintaining high performance of the applications on grids. A practical and robust grid computing infrastructure that integrates components related to application and resource monitoring, performance modeling, scheduling and rescheduling techniques is highly essential for large-scale deployment and high performance of scientific applications on grid systems and hence for fostering high performance computing.

1.3 Problem Statement

This thesis focuses on developing performance models for predicting execution times of parallel problems/subproblems on dedicated and non-dedicated grid resources. The thesis also constructs robust scheduling and rescheduling strategies in a grid metascheduler that can use the performance models for efficient execution of large scientific parallel applications on dynamic grids. Finally, the thesis builds a practical and robust grid middleware infrastructure which integrates components related to performance modeling, scheduling and rescheduling, monitoring and migration frameworks for large-scale deployment and use of high performance applications on grids.

The following sections briefly introduce the individual components of the research.

1.4 Performance Modeling of Parallel Applications

Performance modeling and analysis has been and continues to be of great practical and theoretical importance in research labs in the design, development and optimization of computer and communication systems and applications. Performance predictions of parallel applications have been mainly used for scheduling decisions, identification of bottlenecks in applications and systems and fine tuning of algorithms to provide scalability for larger problem sizes and larger number of processors.

Over the years, many performance modeling efforts [118, 126] have been undertaken to predict the performance of parallel applications on various systems. The performance modeling strategies are of different types including trace-event simulations [15, 126], parameterized analytical models [118, 131] and curve-fitting models [131].

Trace-event simulation is one implemented as a set of procedures that when executed in a computer, mimics the behavior and the static structure of the real system. This type of model uses empirical methods as possibly the only way to achieve a solution. A simulation model can be considered an informal mathematical model.

Analytical models are mathematical models that are solved through analytical methods, i.e., by solving sets of equations and other mathematical relations. This type of solution is generally

only possible for relatively simple models. In practice, analytical models are also called the mathematical models that are solved by empirical methods, although not by simulation.

Curve-fitting technique is used to represent empirical data using mathematical models. With the correct model and calculus, one can determine important characteristics of the data, such as the rate of change anywhere on the curve.

Accurate performance predictions are necessary for selecting a set of target resources for application execution. Application scheduling uses predictions of application behavior. Inaccurate predictions can lead to inefficient schedules and thus can result in performance degradation of the applications when executed on the schedules. Performance models that do not reflect changing conditions are often insufficient for the resource selection and scheduling needs of parallel applications executing on shared grid resources.

Many performance modeling strategies have been proposed in the literature for predicting execution times of parallel applications. However, the existing strategies have different limitations that prevent them from being used for large non-dedicated grid systems.

- Most of the existing modeling strategies assume uniform loading conditions on the systems when the experiments for modeling are conducted and use the models to predict execution times for large problem sizes and/or larger number of processors for the same loading conditions[133, 131, 15, 106, 6, 2, 91, 22]. This assumption is unrealistic in non-dedicated environments.

- Some of the models require source code of the applications for the construction of the models and for instrumentation of the critical components[106, 6, 2, 22, 152, 11]. Analysis and instrumentation of source codes of applications are time-consuming for large complex applications and can prevent large-scale deployment of these applications on grids.

- Some modeling methods also require analytical models expressing the computation and communication characteristics of the applications[106, 6, 2, 133, 131, 119, 118, 120]. Building robust analytical models require detailed knowledge of the applications and such knowledge is available only with the application developers. Requiring this knowledge

can prevent application developers from integrating their applications into grids.

- Some of these modeling strategies also perform large number of benchmarks of the individual code segments to automatically determine the analytical models for the code segments[106, 6, 2, 133, 131, 152, 11].

- Some of the existing efforts for non-dedicated environments can deal with different loading conditions during training the models and predictions, but require the loads to be constant during an application execution[15, 152, 11]. Hence these models can deal with grids with only limited amount of non-dedicatedness.

In this work, we have developed a comprehensive set of performance modeling strategies[117] to predict the execution times of tightly-coupled parallel applications on a set of resources in a dedicated or non-dedicated cluster. The main purpose of our prediction strategies is to aid grid metaschedulers in making scheduling decisions. Following are the specific aspects of our performance modeling strategies, based on linear regression, that result in good prediction accuracies of our models on non-dedicated grid systems.

- Our prediction strategies can deal with non-dedicated systems where the loads can change during application executions. Our techniques periodically monitor and measure loads on the processors and network links during application execution. The aggregates of these measurements across all processors and links are used as parameters of our linear regression models.

- By continuously updating these aggregate values of loads, and using the predicted values in our performance models, our strategies are able to dynamically adapt to changing CPU and network loads on the grid resources.

- Our techniques also continuously evaluate the fitness of a performance model function for changing loads and can use different functions at different times for the same application based on grid load and application dynamics.

Besides, our models do not require detailed knowledge and instrumentation of the applications and can be constructed without the involvement of application developers. Moreover, our

strategies can derive an initial coarse-level performance model for an application by conducting only up to 30 experiments with the application. The model is subsequently refined with increasing executions of the application by the grid users. These features enable the use of our modeling strategies for rapid and large scale deployment of parallel applications on non-dedicated grid systems. Our performance modeling strategies gave less than 30% average percentage prediction errors in all cases, which is reasonable for non-dedicated systems. We also found that scheduling based on the predictions by our strategies will result in perfect scheduling in many cases and can result in maximum loss in efficiency of the scheduler by only 11%.

Modeling phases of large scientific applications where the computation and communication complexities can vary in different phases of application involves additional challenges. Using execution profiles and automatic program analysis[121] and manual analysis of significant portions of the application's code, we identify the different phases of applications. We then adopt our comprehensive set of performance modeling strategies to predict execution times for the different phases of the tightly-coupled parallel applications on a set of resources in a dedicated or non-dedicated cluster. Our experiments show that using combinations of performance models of the phases give 18% - 70% more accurate predictions than using single performance models for the applications.

1.5 Scheduling Strategies

Scheduling is difficult and challenging in grid computing due to the very dynamic and unpredictable nature of Grid resources. The scheduling problem can be viewed as a multivariate optimization problem, where the application is being assigned to a set of machines to optimize the overall execution time. Job scheduling problem is an NP-Complete problem [66]. The problem is extensively studied and various heuristics have been proposed in the literature. Scheduling algorithms adopting resource-centric scheduling objective functions aim to optimize metrics related to resource characteristics including utilization, throughput etc. Scheduling algorithms adopting application-centric scheduling objective functions aim to optimize the performance of the application. To achieve high performance, application-level schedulers in the grid are

tightly integrated with the applications[19]. The unique characteristics of grids including heterogeneity of candidate resources, the dynamism in resource performance, and the diversity of applications make the scheduling and job allocation more challenging.

Although computational grids have gained importance, less attention has been paid to parallel job scheduling on multi-cluster systems[28]. Most of the existing research on scheduling deals with single cluster systems [10, 136, 85, 144, 3, 24, 4]. Multi-cluster systems, compared to the classical parallel computers, pose several technical challenges that introduce additional degree of complexity to the scheduling problem while amplifying the existing ones. For example, resources in grid computing are: (1) distributed (2) heterogeneous and (3) highly shared in both time and space. Therefore, continuously arriving jobs and dynamically changing available CPU capacity make traditional scheduling algorithms difficult to apply in grids. Therefore, online scheduling algorithms promise better results by adapting schedules to changing resource behavior.

While many algorithms exist for scheduling loosely coupled parallel applications on grids[16, 143, 153], very few research efforts have focused on scheduling tightly-coupled parallel applications [1, 79, 93]. The existing algorithms for scheduling tightly-coupled parallel applications are based on evolutionary techniques including simulated annealing and genetic algorithm[79, 154].

In this work, we have devised, evaluated and compared algorithms for scheduling tightly-coupled parallel applications on non-dedicated clusters consisting of homogeneous machines. Our algorithms are also applicable for grid frameworks consisting of multiple clusters where machines within a cluster are homogeneous while machines from different clusters can be heterogeneous. When a tightly-coupled parallel application is submitted to such a grid, our algorithms will be invoked simultaneously on multiple clusters and the schedule of machines from one of the clusters that gives the overall minimum execution time will be chosen for application execution. Our algorithms use performance models that predict the execution times of parallel applications, for evaluations of candidate schedules. In this work, we propose a novel algorithm called Box Elimination (BE) that searches a space of performance model parameters to determine efficient schedules. By eliminating large search space regions containing poorer

solutions at each step and using Roulette wheel based mechanism, our algorithm is able to generate efficient schedules within few seconds for even clusters of 512 processors. By means of large number of real and simulation experiments, we compared our algorithm with popular optimization techniques, namely, simulated annealing, ant colony optimization, genetic algorithm, incremental dynamic programming algorithm and branch and bound algorithm. We show that our algorithm generates up to 80% more efficient schedules than other algorithms and the resulting execution times are more robust[54] against performance modeling errors.

The primary contributions in this work are development of a novel scheduling algorithm and evaluations and comparisons of algorithms for scheduling tightly-coupled parallel applications on non-dedicated clusters and multi-cluster grids.

1.6 Rescheduling strategies

Grids can be highly dynamic due to contention caused by external loads on the heterogeneous grid resources. Heterogeneity and contention can lead to performance degradation of long running scientific applications when executed on grids. So rescheduling decisions, in response to changes in resource performance, are necessary to achieve better performance or to prevent performance degradation. Rescheduling decisions are also necessary because of lower system mean time before failure (MTBF) of systems on which applications execute.

Rescheduling of parallel applications is important in non-dedicated dynamics computational grids. There are very few previous efforts for rescheduling of parallel applications in grid environments [155, 17, 116, 141, 81, 92, 88, 61]. Most of the efforts on rescheduling deal with work-flow applications[155, 17, 116]. Very few efforts[141, 17] have developed rescheduling strategies for tightly-coupled parallel applications.

In this work, we use our performance modeling and scheduling strategies to derive rescheduling plans for executing multi-phase parallel applications on grids. A rescheduling plan consists of potential points in application execution for rescheduling and schedules of resources for application execution between two consecutive rescheduling points. The rescheduling plan is built for a specific set of resource characteristics and considers change in application be-

havior between different phases. We have developed three algorithms, namely an *incremental* algorithm, a *divide-and-conquer* algorithm and a *genetic* algorithm, for deriving a rescheduling plan for a parallel application execution. We have also developed an algorithm that uses rescheduling plans derived on different clusters to form a single coherent rescheduling plan for application execution on a grid consisting of multiple clusters.

The rescheduling plans generated by our algorithms are highly efficient leading to application execution times that are higher than the execution times corresponding to brute force method by less than 10%. We also find that rescheduling in response to changing application and resource dynamics, using the rescheduling plans for multi-cluster grids generated by our algorithms, give much lesser execution times when compared to executions of the applications on a single schedule throughout application execution.

1.7 A Grid Framework for Tightly Coupled Parallel Applications

Computational grids enable the researchers and grid community users to share their computing resources. This offers variety of resources to match different kinds of applications and dramatic increase in the number of computing resources. In order to make grids as highly available environments and to achieve good application performance, grid middleware, that provides the single system image of the available resources to the executing applications, has to be built. The middleware software infrastructure has to integrate modeling, scheduling, and rescheduling strategies to support efficient execution of applications.

Our goal is to develop a framework that automatically performs all the submission steps and also provides automated performance modeling strategies for any parallel application and runtime mechanisms for adapting application execution to grid dynamics using both application and resource specific considerations.

Large number of robust grid frameworks exists for supporting deployment and execution of loosely-coupled parallel applications[153, 149, 16]. Recently, efficient mechanisms have been built for enabling workflow applications on grids[42, 99, 155]. However, not much work exists

for efficiently executing tightly-coupled parallel applications with potential multiple phases of computation and communication characteristics on multi-cluster grids. The primary reason for less work in the area is the possible degradation in performance when heavy communications in such applications occur on the slow-latency inter-cluster links when executed on grids. Hence the usefulness for multi-cluster grids for executing such applications has largely been related to expanding problem sizes[8] and not essentially for improving performance. While we adhere to the general principles of confining the executions of such applications in a cluster at a point of application execution, we claim that efficient scheduling and rescheduling mechanisms for execution of different phases of the applications in different clusters can bring performance benefits to the tightly-coupled applications when executed on multi-cluster grids.

There are very few efforts towards development of such grid computing software infrastructure for efficient and adaptive executions of large-scale parallel applications with complicated computation and communication behavior[73, 96, 30, 86, 81, 35, 103]. Most of the efforts[96, 30, 86, 35, 103] aim at maximizing resource utilization or providing high availability and adopt resource-centric scheduling.

Existing frameworks[98, 97, 43, 129, 55, 141, 34] for enabling MPI based tightly-coupled parallel applications on grids do not adequately deal with application dynamics for large-scale parallel applications, have large application execution overheads, and are generally not suitable for multi-cluster grids that can consist of batch systems. In this work, we present MerITA (**M**iddleware for P**er**formance **I**mprovement of **T**ightly Coupled Parallel **A**pplications on Grids), a system for effective execution of tightly-coupled parallel applications on multi-cluster grids consisting of dedicated or non-dedicated, interactive or batch systems. Our work brings together performance modeling for automatically determining the characteristics of parallel applications, scheduling strategies that use the performance models for efficient mapping of applications to resources, rescheduling policies for determining the points in application execution when executing applications can be rescheduled to different sets of resources to obtain performance improvement and a checkpointing library for enabling rescheduling.

1.8 Organization of Thesis

The remainder of the thesis is organized as follows.

Chapter 2 reviews existing work on performance modeling strategies, scheduling and rescheduling algorithms, and grid computing infrastructures for supporting high performance parallel scientific applications.

In Chapter 3, we present our comprehensive set of performance modeling strategies to predict the execution times of tightly-coupled parallel applications on a set of resources in a dedicated or non-dedicated cluster. We also present cross-platform modeling techniques for porting the results of performance modeling on one platform or cluster to other clusters in the grid. Our performance modeling strategies gave less than 30% average percentage prediction errors in all cases, which is reasonable for non-dedicated systems. This section also gives results comparing our techniques with a previous known technique for non-dedicated system. We also present performance modeling techniques for large scientific applications, where the computation and communication complexities can vary in different phases of application execution. Here, we describe our strategy to generate phase specific performance models. Our experiments show that using combinations of phase specific performance models give 18% - 70% more accurate predictions than using single performance models for the applications.

In Chapter 4, we describe the various performance model based scheduling algorithms we used in this work. We also describe in detail a novel algorithm, called Box Elimination, developed in this work. We show that our algorithm generates up to 80% more efficient schedule than other algorithms and the resulting execution times are more robust against performance modeling errors.

In Chapter 5, we describe our rescheduling algorithms for forming a set of intervals where each interval contains one or more phases of a parallel application. We present three algorithms to form efficient sets of intervals. The algorithms also generate a schedule for execution of each interval. Our algorithms result in efficient sets of intervals and are on the average only 4.5% less efficient than brute-force techniques.

In Chapter 6, we describe our MerITA (**M**iddleware for P**er**formance **I**mprovement of **T**ightly Coupled Parallel **A**pplications on Grids) architecture. Our framework integrates the

performance modeler, scheduler, rescheduler with checkpointing and resource monitor infrastructures. The framework allows the application to adapt to grid dynamics to achieve better performance by using efficient scheduling and rescheduling strategies.

Chapter 7 presents conclusion and some future directions for our research.

Chapter 2

Related Work

In this chapter, we first describe some of the efforts related to performance modeling of parallel applications. We then review efforts on scheduling of tightly-coupled parallel applications on single cluster and multi-cluster environments. We then describe the recent efforts in rescheduling applications for adapting to grid dynamics. Finally, we discuss grid middleware infrastructures that support adaptivity to dynamics and execution of high performance parallel applications.

2.1 Performance Modeling of Parallel Applications

Most of the existing efforts on performance modeling deal with dedicated environments and require detailed analytical models and source codes for application components.

The work by Xu et. al. [150] builds a two-level hierarchical performance model for predicting execution times of parallel programs. At the top level, a graphical model called *thread graph* is constructed for a parallel program. The thread graph, consisting of communication structures, events and segments, represents a high level abstraction of the program and is used by a graph traversal algorithm to estimate the parallel execution time. The execution times of the individual segments of the thread graph are estimated using a low-level model that combines analytical and experimental methods to capture system level effects on performance. However, their model needs analysis of the application for constructing the thread graph for the top level

model and analyzing the loop level constructs for the lower level model. The source code also needs instrumentation to obtain the measurements corresponding to small number of iterations in the individual segments to predict the execution times for larger number of iterations.

The PACE toolkit[106, 6] performs analysis of the application's source code to form a set of performance model objects for the application components using CHIPS performance model language. These objects are compiled and linked with the application workload and hardware models. The underlying hardware for the execution of the application is described by a hardware language. The resulting model executable is parameterized by application and system parameters and can generate performance predictions. Though PACE can perform predictions on heterogeneous platforms, it needs at least some parts of the source code for the applications and description of the hardware configuration. The resulting performance models also do not consider transient external loads that can occur during the application execution.

The POEMS project[2] has built a robust infrastructure including a specification language, component models, a database for storing task dependencies and performance results of individual components, to predict the execution times of the parallel applications. The different components used by the application are specified by a detailed specification language. The task dependency graph of the application is generated automatically by a compiler. The POEMS project needs source code of the application for component specification and task graph generation. In order to obtain execution times corresponding to the nodes of the task graph, the computation segments are either bench-marked or specified using analytical models. The number of benchmarks are the product of number of computational segments, processors and problem size. The specification of analytical models require the intervention of the application developer. POEMS also does not consider external load and hence is applicable for only dedicated environments.

Prophesy's[133, 131] curve-fitting model utilizes both experimental results and the complexities of the applications to predict execution times of applications for larger problem sizes. The curve-fitting models used in Prophesy do not separate system and application parameters and encapsulates all kinds of system dynamics in the model coefficients. Prophesy also uses parameterized analytical models to address the deficiencies of their curve fitting models. In

an analytical model for an application, the different model coefficients are assigned specific weights related to system parameters. The weights are determined by detailed analysis of the code and are calculated using system benchmarks. For example, if an application has a quadratic communication complexity and most of the communications are broadcasts, then the time taken for broadcasts is used for the quadratic component of the model equation. The broadcast time is calculated by executing broadcast benchmarks on the system. Thus in Prophesy, the model developer needs detailed knowledge about the components of the application. Secondly, benchmark experiments have to be conducted for various application specific subcomponents including broadcasts, floating-point additions, multiplications etc. The number of such benchmarks increase as more applications are added since different applications have different subcomponents. Our model strategies use only generic benchmarks relating to CPU loads, and latencies and bandwidths of network links for all applications. These benchmarks are integral to many parallel, distributed and grid systems.

The Modeling Kernel (MK) by Block et. al.[22] automatically extracts computation and communication characteristics of a parallel application using SAGE compiler. It then uses a instrumentation and monitoring system to gather execution profiles of the code segments and converts the numerical information from the execution profiles into model functions in terms of problem size and number of processors. The Modeling Toolkit is applicable to only dedicated platforms and needs source code for compiler analysis.

The work by Clement and Quinn[37] uses compiler analysis techniques to instrument the application code and obtain various parameters of an application on a system, including the number of operation counts, cache misses, transitions from serial to parallel parts etc. After performing a number of experiments on a given system with different applications and determining the parameters, they then use linear regression techniques to obtain the cost of the associated parameters. To predict the execution time of an application on a system, they use the parameters of the application along with the costs obtained during training the models. Their work requires source code of the applications to perform compiler analysis while our work uses executable binaries. Their results also show high prediction errors of close to 60%. Our work also extends their linear regression techniques to account for external CPU and network loads

on non-dedicated systems.

The work by Lee et. al.[91] uses two methods, namely, linear regression and neural networks, to predict execution times of parallel applications with multiple parameters. Their linear regression model uses cubic splines to fit the training data. Their models require specifications of ranges of problem sizes. Since grid systems expand due to addition of new resources and since users want to increase problem solving capacity with increasing grid sizes, it is not practical to specify problem size ranges in grids. The work also considers prediction of execution times for different problem sizes but for a fixed number of processors. The cubic splines used in the work is also applicable for only those applications with cubic complexities.

Feed forward neural networks have also been used in the performance predictions of parallel applications[83]. Although neural networks can be used for predictions in non-dedicated environments, the amount of training and hence the time for training the models is relatively higher than linear regression models as illustrated in their results.

Some recent efforts on performance modeling deal with limited kind and amount of non-dedicatedness on the systems. Dimemas[15] is used for analyzing the performance of parallel applications. The parallel application instrumented with Dimemas instrumentation libraries is executed either on a dedicated or non-dedicated sequential or parallel system. This leads to the generation of Dimemas trace files. The trace files contain the CPU times of computation and communication primitives. Dimemas also accept trace files from Metasim tracer of the PMac[31] project for single processor performance. These trace files along with the specification of a target parallel system are fed to a Dimemas simulator which outputs the performance behavior of the application on the target system. Though the application can be run on a non-dedicated system for collecting trace files and the simulator can predict the application's execution time for different loading conditions, Dimemas cannot predict for systems where the load varies during application execution. Also, dimemas simulator requires detailed specification of the target parallel system.

The work by Grove et. al.[76] requires the application developer to specify the complexity of serial portions and message passing constructs for his application using a performance modeling language. The message passing communication complexities take into account the amount

of network contention caused by the application at the time of communications. The model is then executed on a Performance Evaluation Virtual Parallel Machine (PEVPM) that predicts the execution time of the application. Thus the work takes into account only the contention caused on the network by the application itself and not due to external load. Our modeling strategies work for non-dedicated environments where the loads on the machines can vary during application execution.

The work by Dinda[45] analyzes the impact of system loads on predicted execution times. The work utilizes various load measurement and load prediction strategies and provides a concrete formula for utilizing available load predictions to predict execution times. They translate confidence intervals in the available loads to confidence intervals in prediction times. Their experimental results, where they analyze the confidence intervals of their predictions, are reported only for sequential application traces.

The effort by Yan et. al[152] predicts execution times of parallel applications on non-dedicated heterogeneous systems. They consider a system where a set of workstations can be used for the execution of parallel application and external load can appear in the form of workstation owners executing their own processes. They use Program Execution Graphs (PEGs) for modeling application characteristics. Angalano[11] extended this work by using Petri-Net models for characterizing application behavior. Both the efforts need detailed analysis of the source code to identify the program computation segments, communication primitives and precedence relationships. The efforts also need detailed benchmarking of the individual code segments for different problem sizes and number of processors. The efforts also use a parameter called O_i that represents the average execution time of the owner process or external load on machine i. Calculation of O_i is not feasible on grid systems where the amount of interference by external load can vary over time.

The work that is most closely related to our work is the one by Schopf and Berman[119, 118, 120]. Their structural prediction modeling approach builds a model for an application in terms of the models for the components of the applications. The motivation and the scope of their work is similar to ours in that they try to predict execution times in dynamic non-dedicated environments where the load can vary at different times. They measure load parameters from a

monitoring, measuring and prediction tool called Network Weather Service (NWS)[147, 146]. They use stochastic values of the measurements in their component models. They then use arithmetic operations based on stochastic values to obtain stochastic prediction execution times. They use two strategies, namely, normal distribution and interval method to obtain stochastic values of measurements. Unlike our work, their work requires profiling tools for determining the communication and computation requirements of the application. Their component models are also based on detailed parameterized analytical models requiring intervention of the application developer for conveying the complexities of the application in terms of system and application characteristics. For large load dynamics, stochastic techniques give large ranges of predicted execution times and hence the usefulness of such large ranges are unclear.

To summarize, it is increasingly difficult to obtain knowledge regarding both the applications and the execution platforms. Application developers often lack sufficiently detailed knowledge about platforms. Similarly, a job scheduler cannot obtain application-specific knowledge without user intervention. Our automated modeling strategies obtain knowledge of both the applications and platforms. Our modeling strategies work for non-dedicated environments where the loads on the machines can vary during the execution of the application. Our models work with the executable binaries of the applications. Except the minimum problem size, it does not need any other knowledge about the applications.

There are very few efforts [132] towards modeling scientific parallel applications [33, 101]. Taylor et. al. [132] address the issue by using a coupling parameter, which quantifies the interactions between software kernels, to develop performance predictions. Their component models are based on detailed parameterized analytical models requiring intervention of the application developer for conveying the different kernels and complexities. The modeling is applicable to only dedicated environment. They present results for small applications. There are several strategies for automatic phase detection and prediction in sequential applications[44, 46, 123, 110]. These techniques use various application parameters including high-level structure of the source code, working sets, conditional branches and basic blocks to identify phases in the application. Our performance modeling strategy predicts the execution times of the phases of the parallel applications on a set of resources in dedicated or non-dedicated cluster.

2.2 Scheduling

One motivation of Grid computing is to aggregate the power of widely distributed resources, and provide non-trivial services to users. To achieve this goal, an efficient grid scheduling system is essential. A grid scheduler receives applications from grid users and selects feasible resources for these applications according to acquired information. Scheduling algorithms have been intensively studied as a basic problem in traditional parallel and distributed systems, such as symmetric multiple processor machines (SMP), massively parallel processors computers (MPP) and cluster of workstations (COW). It is well known that the complexity of a general scheduling problem is NP-Complete[66].

Most of the existing efforts in scheduling parallel applications deal with independent tasks[16, 143, 153] and parameter sweep applications[32, 62]. Some efforts with inter-dependent tasks assume knowledge of task precedence constraints[26]. The effort by Aggarwal et. al. [4] considers scheduling a set of DAG based jobs on computational grids. They present genetic algorithm based scheduler to minimize the makespan. Building task graphs expressing precedence constraints for complex parallel applications is non-trivial.

There have also been many efforts related to scheduling of workflow applications[78, 95]. We differentiate tightly-coupled applications from the workflow applications that have been successfully scheduled and executed on grids[42]. A workflow application is represented as a Directed Acyclic Graph (DAG) where each node of the graph represents a parallel or sequential application that forms a component of the overall application and an edge denotes the control and data dependency between two application components. While substantive data transfers take place along edges of a workflow graph, the communications on an edge are not as frequent as the inter-task communications in tightly-coupled applications and mostly happen once after the completion of an application component. Hence, while loosely coupled and workflow applications achieve good performance when executed across multiple clusters, tightly-coupled applications exhibit poor performance due to low-speed network links between the clusters. Thus, tightly-coupled applications are typically executed within a single cluster consisting of homogeneous machines. Thus, scheduling strategies that have been developed for allocating workflow application[78, 95] components on different clusters of a grid[25, 99, 4] cannot be

used for tightly-coupled applications due to higher frequency of inter-task communications in tightly-coupled applications.

There have been some efforts in scheduling a set of tightly-coupled parallel applications on a set of multiprocessor clusters[1, 79, 93]. Abwajy et. al.[1] consider on-line dynamic scheduling policy that manages multiple jobs across both single and multiple clusters with the objectives of improving mean response time and system utilization. Although their work deals with scheduling tightly-coupled parallel applications, they use system level scheduling and focus on maximizing cluster utilization. Our work primarily deals with maximizing application performance. Few scheduling strategies adopt application level scheduling. AppLeS[18] is a general framework designed to meet performance goals specified by the application. Each application has its own scheduling agent that monitors available resources and generates a schedule.

He et. al[79] developed techniques for scheduling a set of parallel jobs on processors of a multi-cluster Grid. They use a multi-tiered architecture comprising a metascheduler called MUSCLE for allocating parallel jobs to different clusters and a workload manager called TITAN at the single cluster level for scheduling in a cluster. TITAN employs genetic algorithm to choose schedules for the jobs to minimize response times of the jobs and idle times of processors and to meet job deadlines. Similar to our work, their algorithm evaluates candidate schedules using a performance prediction system called PACE[106]. Unlike our work, their scheduling architecture and performance models are intended for dedicated systems. Our work also evaluates various algorithms including genetic algorithm and shows that our Box Elimination (BE) algorithm yields better schedules than genetic algorithm.

The work by Li[93] also deals with scheduling a set of parallel jobs across multiple clusters. The work can schedule execution of a parallel application across multiple clusters. In our work, we confine a parallel application execution to a single cluster. Li's work uses a simplifying cost model that predicts execution time across multiple clusters in terms of computation-to-communication ratio of parallel applications and ratio of communication bandwidth within a parallel machine to the communication bandwidth of inter-cluster links. They propose a minimum effective execution time (MEET) algorithm that optimizes their cost model by minimizing the number of clusters used for parallel application execution. However, our work considers pro-

cessor and network loads in our performance models and hence our scheduling algorithms are suitable for non-dedicated systems. Moreover, the work by Li performs simulations for various fixed values of application computation-to-communication ratios. In our work, we determine characteristics of real parallel applications using linear regression techniques.

Sabin et. al.[115] consider scheduling parallel jobs in a heterogeneous multi-site environment where each site has homogeneous clusters. Thus their scheduling environment is similar to ours. When a tightly-coupled parallel application is submitted, the application is scheduled to an individual site using a scheduling algorithm. They initially used greedy algorithm and extended it to use multiple redundant requests for scheduling. They also used application efficiency at sites, obtaining completion times using conservative backfilling and reservation requests to enhance their scheduling algorithm. However, in their model, each job specifies processor requirements. In our work, we choose the set of processors for a parallel job. Moreover, in their work, the individual clusters are space shared and local scheduling at the clusters is using backfilling with FCFS policy. Our work deals with non-dedicated time-shared clusters.

The work by Yarkhan and Dongarra[154] uses simulated annealing based approach for scheduling tightly-coupled parallel applications on non-dedicated grids. Our work compares different algorithms including simulated annealing for scheduling parallel applications. We show by means of experiments that simulated annealing based techniques give inefficient schedules in many cases.

Our scheduling algorithms are intended for scheduling tightly-coupled parallel applications on a non-dedicated cluster consisting of homogeneous machines. The algorithms are also applicable for grid frameworks consisting of multiple clusters where machines within a cluster are homogeneous while machines from different clusters can be heterogeneous. The algorithms will be invoked simultaneously on multiple clusters and the schedule of machines from one of the clusters that gives the overall minimum execution time will be chosen for application execution. Our algorithms use performance models that predict execution times of parallel applications, for evaluations of candidate schedules.

2.3 Rescheduling

Compared to scheduling, only few efforts exist in the area of rescheduling parallel applications[155, 17, 116, 141, 81, 92, 88, 69, 94, 142, 157]. Most of the efforts[69, 94, 142, 157] migrate applications either to efficiently use underutilized resources, to provide fault resilience or to reduce the obtrusiveness to workstation owner. The work by Kurowski et. al.[88] adopts rescheduling to shorten job pending times and reduce machine overloads. The efforts by Yu et al.[155] and Sakellariou et. al.[116] developed rescheduling techniques for work-flow applications.

The work by Huedo et. al.[81] presents Globus based framework for rescheduling executing applications. The framework performs steps related to job submission and also monitors application execution. In order to obtain a reasonable degree of performance, job execution is adapted to dynamic resource conditions and application demands. Adaptation is achieved by supporting automatic application migration following performance degradation, better resource discovery, change in resource requirements, owner decisions or remote resource failure. But their scheduling and rescheduling strategies are more resource centric, since they use Globus RSL (Resource Specification Language) script to specify the application requirements. RSL script cannot adequately specify the computational and communication requirements of the application. Thus their framework allows rescheduling of task only if there is change in resource characteristics. In our strategy, we consider changes in both resource and application characteristics for rescheduling.

In multi-cluster systems and more generally in grids, various types of parallel applications can benefit from being able to change their processor configuration after they have started execution. Malleability of parallel applications may yield improved application performance and better system utilization since it allows more flexible scheduling policies. Recently, many frameworks have been developed to reschedule executing parallel applications on grids [80, 149, 97, 43, 129, 55, 148, 82, 141]. In this section, we compare our work with those efforts that deal with both application and resource dynamics.

The work by Hussein et al.[82] considers migration of components in coupled scientific models in response to resource and application dynamics based on predictions using exponen-

tial smoothing techniques. The ReSHAPE framework[129] can shrink and expand processor configurations for parallel applications on homogeneous cluster. The reconfiguration framework supports only homogeneous applications with uniform behavior throughout application execution and allows only shrinking to processor configurations on which the applications have previously run. The framework employs trial-end-error for rescheduling to different processor configurations with the objective of executing on a processor configuration that gives best performance for the application. Thus the method can involve rescheduling to inefficient processor configurations.

Varela et al. have developed a modular framework called Internet Operating System (IOS)[98, 43, 97] for supporting adaption to both changing application and resource characteristics. Their decisions for rescheduling consider only process-level characteristics and are not suitable for large scale multi-phase scientific applications involving application-level change in characteristics in different phases. In the work by Bal et. al.[148], an adaptation coordinator uses profile information from application processes and calculates weighted average efficiency for the application. This metric considers processor utilization by the processes and the speed of the processors. The framework adds and removes nodes whenever the metric values are beyond some predefined range. The framework does not consider large-scale reconfiguration or migration of executing applications and are more suitable for reconfiguring processes of divide-and-conquer applications.

All the above frameworks use profiling to obtain process-level characteristics during application execution. Online profiling can lead to large application overheads and these efforts report increase in execution times of up to 20% in some cases[97, 148]. The GrADSolve infrastructure by Vadhiyar and Dongarra[141, 139] uses performance models that give expected or predicted performance for each iteration or phase of a parallel application. The framework consists of a performance monitor component that contacts a rescheduler if the running average of ratios between actual and expected performance for different iterations or phases is beyond some predefined range. The rescheduler either reschedules the application to a different set of resources if performance gain can be achieved or makes the contract monitor lower its performance expectations. Thus the GrADSolve framework allows adaption to both resource and

application dynamics. However, the framework is more suitable for iterative applications with uniform behavior throughout application execution. The rescheduling decisions do not consider future change in application characteristics. In our strategies, we form potential points of rescheduling based on the global knowledge of all the application phases.

2.4 Grid Computing Software Infrastructure

Grid computing middleware allows users to submit their applications and perform various tasks for grid executions including resource discovery, resource selection, data and executable transfers for job execution, job submission, monitoring, possible job migration, and termination. A grid middleware tries to achieve various objectives including maximum performance for the applications, seamless use of remote services and secure communications. There are very few efforts to building grid computing software infrastructures to support tightly- coupled parallel applications.

MetaLoRaS [96] is a metascheduler consisting of a queuing system with two-level hierarchical architecture for non-dedicated multi-clusters. The most important contribution of the MetaLoRaS middleware is an effective cluster selection mechanism, based on estimation of job turnaround times. Parallel applications are assigned to clusters where the minimum estimated turnaround time is obtained. In their work scheduling and rescheduling decisions consider only resource performance. The rescheduling components of their middleware migrates waiting jobs from one queue to another. Their work does not consider the migration of executing jobs and is not suitable for parallel applications.

There have been many efforts on executing loosely coupled parallel applications on grids[153, 149, 16]. Loosely coupled applications are amenable to performance benefits with grid executions since the communications between individual tasks are negligible. Scheduling mechanisms for such applications do not have to consider the communication characteristics of the application. Adapting such applications to grid dynamics are also relatively simpler since individual tasks can be migrated to better resources with minimal coordination with other tasks for adaptation[148]. Recently, there has been interest on integration of workflow applications

on grids[42, 99, 155]. Scheduling and rescheduling strategies that have been developed for allocating the workflow application components on different clusters of a grid cannot be used for tightly-coupled applications due to higher frequency of inter-task communications in tightly-coupled applications.

There has also been recent interest in developing rescheduling or reconfiguration frameworks for MPI based tightly-coupled parallel applications[97, 129, 55, 141]. AMPI[80] and the work by Fernandez et al.[55] allow application reconfiguration by over-decomposing parallel application into large number of entities (processes or threads) and migrating the threads when the availability of nodes change. These techniques are not practical for grid systems where node availability can frequently vary. When the resource availability is small, many entities will be executed on a single node leading to application performance degradation and overloading of system resources. The work by Fernandez et al.[55] shows about 36-88% increase in execution times of some applications when many threads or mapped to a single processor.

As mentioned in previous sections, malleability helps applications perform better when resource availability varies. Several approaches have been used to build malleable parallel applications. Varela et al.[98, 43, 97] have developed Internet Operating System for adaptive distributed computing of MPI applications. Their framework, consisting of profiling, decision and protocol components, supports reconfiguration of parallel applications in response to resource and application dynamics. They use MPI-2 dynamic process spawning mechanisms and rearrange MPI communicators for enabling reconfiguration of an executing application to different number of nodes. They also provide interfaces to the user for splitting and merging of nodes.

MPI-2 dynamic process spawning is not implemented in all MPI implementations and is also disabled on some production batch systems to discourage over consumption of resources by the users. Thus their reconfiguration frameworks are not portable across different systems. Also, their process-level on-the-fly reconfiguration during application execution requires all the processors for application execution to directly communicate with each other. This is not applicable for rescheduling applications between different distributed clusters where most of the nodes in a cluster have private IPs and will not be able to communicate with nodes of other

clusters. Although our stop-and-restart mechanism using SRS[138] library can incur higher reconfiguration overheads then their on-the-fly reconfiguration as shown in their work[43], it can enable reconfigurations across different clusters.

The ReSHAPE[129] framework is intended for dynamic resizing of homogeneous applications. The framework uses a resizing library implemented for dynamic process management. They support efficient data redistribution for remapping data to different sets of processors. The framework also uses MPI-2 dynamic process spawning for on-the-fly reconfiguration and hence is not applicable for multi-cluster batch systems.

All the above frameworks use profiling to obtain process-level characteristics during application execution and hence result in large application overheads. The frameworks also consider only process-level characteristics for rescheduling and are not suitable for large scale multi-phase scientific applications. Our MerITA framework based on performance models form potential points of rescheduling based on the global knowledge of all the application phases.

GrADS [73] presents program development framework for tightly-coupled parallel applications. Two key concepts are central to this approach. First, applications are encapsulated as configurable object programs (COPs), which can be optimized rapidly for execution on a specific collection of Grid resources. A COP includes code for the application (e.g. an MPI program), a mapper that determines how to map an application's tasks to a set of resources, and an executable performance model that estimates the application's performance on a set of resources. Second, the system relies upon performance contracts that specify the expected performance of modules as a function of available resources. The GrADS infrastructure first determines which resources are available and, using the COP's mapper and performance model, schedules the application components onto an appropriate subset of these resources. Then the GrADS software invokes a binder to tailor the COP to the chosen resources and a launcher to start the tailored COP on the Grid.

Our MerITA framework is similar to the GrADS[73] framework. In GrADS, performance model for an application should be written by the application developer or user requiring user intervention. The rescheduling decision is made for every iteration of the application, which leads to extra checkpointing cost. In our work, we developed a grid framework for tightly-coupled

parallel applications. Our performance modeling tool automatically builds an application specific model to aid the grid metascheduler. We use an algorithm called Box Elimination algorithm to find the best schedule for the application. Our framework also performs steps involved in application submission. Adaption to grid dynamics is achieved by application rescheduling following performance degradation due to non-dedicatedness in the resources and change in requirements for the applications.

Chapter 3

Performance Modeling of Tightly Coupled Parallel Applications

3.1 Introduction

Performance modeling of tightly-coupled parallel applications that predict execution times of the applications on a set of dedicated or non-dedicated resources is important for efficiently scheduling the applications.

As discussed in Chapter 2. most of the existing modeling strategies assume uniform loading conditions on the systems when the experiments for modeling are conducted and use the models to predict execution times for large problem sizes and/or larger number of processors for the same loading conditions[133, 131, 15, 106, 6, 2, 91, 22]. This assumption is unrealistic in non-dedicated environments. Hence the strategies cannot be used for grids consisting of both dedicated and non-dedicated clusters. Some of the models require source code of the applications for the construction of the models and for instrumentation of the critical components[106, 6, 2, 22, 152, 11]. The source code of many legacy parallel applications may not be readily available for performance modeling. Some modeling methods also require analytical models expressing the computation and communication characteristics of the applications in order to predict the execution times of the applications[106, 6, 2, 133, 131, 119, 118, 120]. Building robust analytical models require detailed knowledge of the applications. Such detailed

knowledge is available only with the application developers. Requiring application developers to provide the performance models along with their applications will prevent many applications from being integrated into the grids. Some of these modeling strategies also perform large number of benchmarks of the individual code segments to automatically determine the analytical models for the code segments[106, 6, 2, 133, 131, 152, 11]. Some of the existing efforts for non-dedicated environments can deal with different loading conditions during training the models and predictions, but require the loads to be constant during the application execution[15, 152, 11]. The work by Schopf and Berman[119, 118, 120] predicts execution times of the applications when the external loads can change during the application executions, but require detailed analytical models.

In this chapter, we present comprehensive set of performance modeling strategies to predict the execution times of tightly-coupled parallel applications on a set of resources in a dedicated or non-dedicated cluster with the purpose of aiding grid meta-schedulers in making scheduling decisions. Our performance modeling strategies based on curve-fitting, predict execution times of applications in non-dedicated systems where the external CPU and network loads on the systems during predictions can be different from the loads when experiments for modeling were conducted. Our prediction strategy is adaptive to changing CPU and network loads on the grid resources and uses different performance model functions at different times for the same application. We have also developed cross-platform performance modeling techniques whereby the performance modeling results of an application on one cluster can be used for predicting execution times of the application on another cluster. Our models do not require detailed knowledge of the applications, and hence can be constructed without the involvement of application developers. Also, our models do not require detailed instrumentation and profiling of applications but only require observation of total execution times. Thus, our models can work with the application executables and do not require the source codes. Our models can also deal with changing loads during application executions. The modeling techniques are intended for simple parallel application kernels which are invoked in complex parallel applications. These parallel kernels have single phases of computation and communication and are integral to many scientific applications.

Section 3.2 presents overview of our modeling strategy. In Section 3.3, we discuss our preliminary and motivating work for performance predictions of tightly-coupled parallel applications. In Section 3.4, we identify the drawbacks of the preliminary work and describe in detail a comprehensive and robust strategy for performance modeling on non-dedicated systems including the various components of our performance models, and the use of candidate functions for the models. In Section 3.5, we present our cross-platform performance modeling technique for performance predictions on different clusters. In Section 3.6 , we present technique for modeling multi-phase applications. Section 3.7 describes the usage scenario of our performance models.Section 3.8 details our experimental setup for single phase application. Section 3.9 describes results for predicting execution times of different applications on different clusters and with different loading conditions and also compares our strategy with earlier work in terms of modeling overheads. Section 3.10 presents experiments and results for modeling multi-phase parallel applications and Section 3.11 gives a summary of the approach and results presented.

3.2 Overview of Modeling Strategy

Our application performance models are based on rational polynomial model functions. The functions and their coefficients are found using linear regression by solving a system of linear equations for which efficient algorithms and software tools are available[41, 38, 89]. Most parallel applications have polynomial computation and communication complexities. Hence, modeling applications using polynomial functions can give better insights into the application characteristics than modeling using black-box models including Artificial Neural Networks (ANNs)[91, 125] and Radial Basis Function (RBF) networks [107]. We use rational polynomials since certain parameters, namely, the CPU and network loads have inverse relationships with the application execution time.

Our model equations splits the execution time of a parallel application into at least two parts, corresponding to computation and communication characteristics of the application. This representation is useful for scheduling purposes since a scheduler can allocate the appropriate

CPU and network resources of a grid based on the computation and communication requirements, respectively, of the application. The model equations generalize the complexity equations of many parallel numerical drivers that deal with memory-resident data[21, 59]. Existing strategies[133, 131] that use linear regression for modeling parallel applications, predict execution times by using a single fixed model function where multiple system and application parameters are combined in the terms of the function. The model function will be applicable only for the system on which the function was derived. For predicting the execution time for the application on a different system, a different function has to be derived. Our modeling strategy, by expressing the execution time of the application using separate functions for computation and communication complexities, scalability with processors, and the effects of external loads on the application, can efficiently determine the impact of various parameters on the application. The same sets of functions can be used to predict the execution time on a different system by varying the coefficients for the functions. For example, the complexities and the scalability with increasing processors are application-specific parameters and can be used on more than one system.

In our strategy, the correctness of our models is verified by comparing the execution times predicted by our models with the actual execution times and calculating percentage prediction errors. For some applications for which the communication and computation complexities are known, we verified our models for the applications by comparing the complexities derived in our model with the known complexities of the applications.

We first begin by building a model for a well-known parallel application with uniform communication behavior between processes. We particularly consider a parallel eigen value solver with known communication and computation complexities to illustrate our modeling procedure. This model is explained in Section 3.3.2. We then generalize the model for generic parallel applications. This preliminary model described in Section 3.3.3. This preliminary model can predict application execution time for the same number of processors on which the model was built. We then build a comprehensive model for predicting execution times for different number of processors. This comprehensive model is described in Section 3.4.

3.3 Preliminary Work

Modeling the performance behavior of parallel applications to predict the execution times of the applications for larger problem sizes and number of processors has been an active area of research for several years. The existing curve fitting strategies for performance modeling utilize data from experiments that are conducted under uniform loading conditions. Hence the accuracies of these models degrade when the load conditions on the machines and network change.

In our preliminary efforts on performance modeling of parallel applications [151], we explored a curve-fitting strategy based on rational polynomials for predicting execution times of applications in non-dedicated systems. Our work considered predicting execution times of a parallel application for different problem sizes for a fixed number of processors, i.e., given execution times of the application for few different problem sizes for a certain number of processors, our methodologies predicted the execution times of the application for certain other problem sizes for the same number of processors in non-dedicated environments. The modeling techniques were intended for simple parallel application kernels which are invoked in complex parallel applications. These parallel kernels have single phases of computation and communication and are integral to many scientific applications. In most cases, the developers of the parallel kernels will be able to provide coarse computation and communication complexities. These complexities were used in our models to predict the execution times of the kernels. Although the coarse complexities were available, automatically refining the model by careful parameterization of the coarse complexity terms with adequate application and system parameters in order to provide decent predictions on noisy non-dedicated systems is still a non-trivial task.

3.3.1 Methodology

We evaluated the models with the help of ScaLAPACK[21] parallel eigen value problem. The ScaLAPACK kernel that was used for the experiments was PDSYEV, the kernel for parallel eigen value solver for a double-precision symmetric matrix. ScaLAPACK uses 2-D block-

cyclic distribution of matrix to processors. We chose the ScaLAPACK PDSYEV eigen value solver for validating our models since it is representative of simple parallel application kernels used in many large scale simulations. The coarse computational and communication complexities of the ScaLAPACK PDSYEV eigen value problem are well known, and are $O(n^2)$ and $O(n^2)$, respectively. As mentioned earlier, these coarse complexities are refined by our models for obtaining accurate predictions. The ScaLAPACK application also has uniform data distribution using block-cyclic mapping, and single phase of computation and communication within an iteration. The uniform data distribution allows us to build simplistic models for the entire application instead of individual models for the processes of the applications. Our models are also applicable for applications with other uniform data distributions including block and cyclic mappings.

The experiments for modeling were conducted on a Intel Pentium IV based Linux cluster consisting of 8 nodes with small load dynamics[1]. All the 8 nodes were connected to each other by 100 Mbps Ethernet link using a 8-port 100 Mbps Ethernet switch. Each of the nodes had a single 2.8 GHz CPU, a 512 MB RAM, a 80 GB hard disk and running Fedora Core 2.0, Linux 2.6.5-1.358 operating system. For the experiments reported in this section, only 4 processors were used. Before conducting an experiment corresponding to the application execution with a problem size, synthetic CPU and network loads were applied to the processors and links between processors respectively. The loads were maintained at constant amounts throughout the duration of the application execution but were varied by random amounts for different application executions.

In order to construct models and predict the execution times for larger problem sizes for a given number of processors, *P*, experiments were conducted by executing the application on *P* processors for smaller problem sizes. In our work, we restricted our predictions to only those problem size ranges where the entire data needed by the application will fit in the local memories of the processors and hence will not incur disk swapping costs.

At the start of conducting an experiment with a problem size, the transient CPU and network

[1]We considered a system as having small load dynamics when the load conditions change from one application execution to another but remain the same throughout the duration of an application execution. We considered a system as having large load dynamics when the load conditions can change even during an application execution.

characteristics were obtained. Network Weather Service (NWS) [147], a tool for measuring and forecasting system parameters, was used for obtaining available CPUs of the processors used for application execution and available bandwidths on the links between the processors. Available CPU is a fraction of the CPU that can be used for the application and inversely proportional to the amount of CPU load. Available bandwidth of a link to an application is usually lesser than the link capacity and is also inversely proportional to the network load on the link. *min_avail_cpu* and *min_avail_band* are then calculated as the minimum of available CPUs of all processors involved in the application execution and minimum of available bandwidths of all the links between the processors respectively. Then the experiment was conducted by executing the application with the problem size and the duration of application execution is observed.

The problem sizes, *min_avail_cpu* and *min_avail_band* values, and the execution times, corresponding to different experiments with different small problem sizes, were used as inputs for training our models. The trained models were then used for predicting execution times for a large problem size for given values of *min_avail_cpu* and *min_avail_band*. For the ScaLAPACK Eigen value problem, experiments were conducted for matrix sizes 100-8000 in step sizes of 100 and the measurements were used for training the models. The resultant models were used to predict execution times for matrix sizes 9000-12000 in step sizes of 100.

The models were evaluated and compared based on average percentage prediction errors. For a given model, the average percentage prediction error is given by:

$$\frac{1}{N}\sum_{i=1}^{N}\frac{|(actual_i - predicted_i)|}{|actual_i|}$$

where *N* is the number of experiments, $actual_i$ is the measured execution time and $predicted_i$ is the predicted execution time by the model for experiment *i*. Since in some cases, the average percentage prediction errors can be misleading, we also evaluated the models by comparing the trends shown by the prediction and actual values.

3.3.2 A Rational Polynomial Model

The model we investigated was a rational polynomial based model taking into account the computation and communication complexities of the problem. ScaLAPACK parallel Eigen

value problem has cubic computation complexity and quadratic communication complexity in terms of problem sizes. If *n* is the problem or matrix size, the rational polynomial based model can be formulated as:

$$P_n = a_1 n^3 + b_1 n^2 + c_1 n + d_1 \tag{3.1}$$

$$P_{cpu} = \frac{a_2 n^3 + b_2 n^2 + c_2 n + d_2}{min\_avail\_cpu} \tag{3.2}$$

$$P_{bw} = \frac{a_3 n^2 + b_3 n + c_3}{min\_avail\_band} \tag{3.3}$$

$$t_{predicted} = P_n + P_{cpu} + P_{bw} \tag{3.4}$$

Equation 3.1 calculates the time needed for problem initializations and synchronizations that happens once at the beginning of application execution. This time depends on the problem size and is not impacted significantly by the external load. Equation 3.2 calculates the total predicted time for computation and equation 3.3 calculates the total predicted time for communication. The total predicted time for the application is given by equation 3.4. The coefficients in the equations are determined by polynomial fit using data points in the training samples. The problem of finding the coefficients is a linear regression problem and is equivalent to solving a system of linear equations. The determined coefficients along with the system and application parameters are used in the equations to predict the execution time for a given problem size.

Figure 3.1 shows the predictions of execution times with the rational polynomial based model. The curve corresponding to predicted times follows very closely with the curve corresponding to actual execution times. The average percentage prediction error is 18% with variance of 0.01 and indicates the reasonableness of the model. Thus our rational polynomial based model gave good predictions for non-dedicated environments.

Figure 3.1 also shows a third curve corresponding to the parameterized analytical model for the ScaLAPACK Eigen value problem. The parameterized analytical model was formulated by the ScaLAPACK authors. The formula expresses total execution time in terms of matrix size, number of processors, latency, bandwidth, and speed of the processors. We scaled down the theoretical speed of the processors by the available CPU obtained from NWS to obtain effective speeds and also used the minimum available bandwidth and maximum available latency in the formula. As can be observed, our model performed much better than the analytical model for

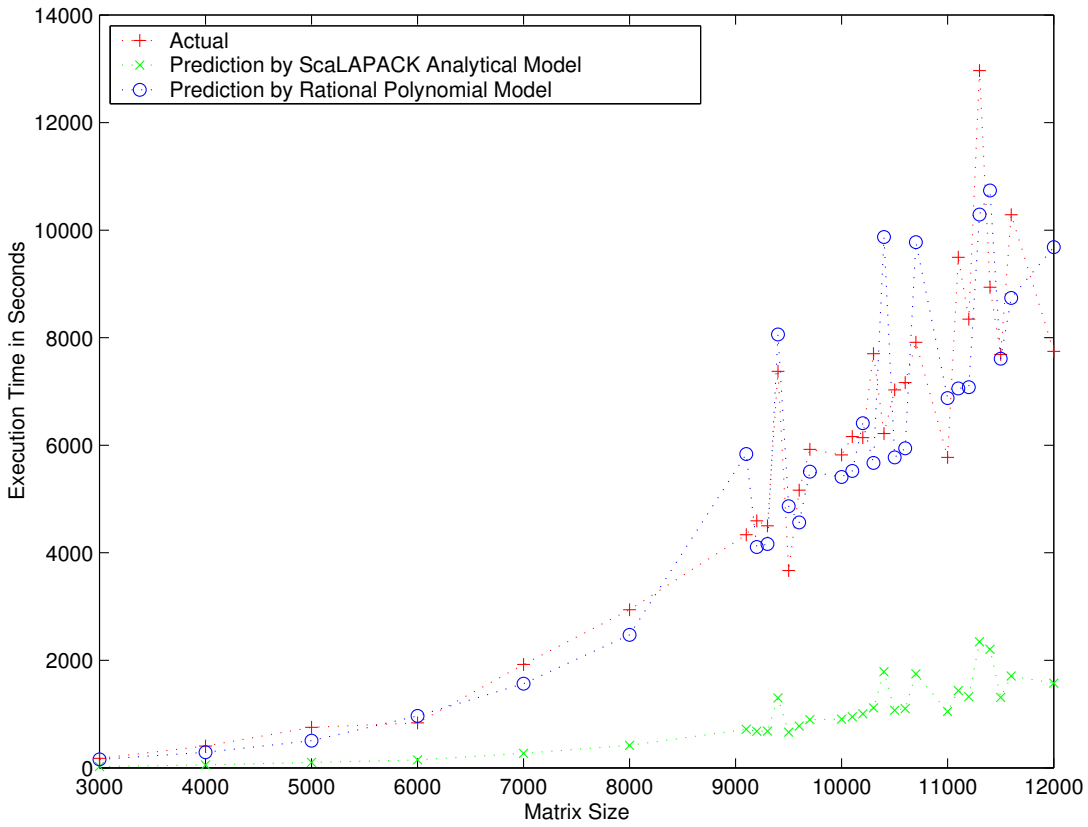


Figure 3.1: Predictions with Rational Polynomial based Model

the ScaLAPACK Eigen value problem. This is because the ScaLAPACK analytical model did not take into account contentions due to external loads while our curve-fitting model embedded the average contention behavior in the coefficients used in its equation.

3.3.3 Generic Model Based on Rational Polynomials

We then proposed a general model involving rational polynomials for predicting execution times of generic parallel applications executing in non-dedicated environments with potential large load dynamics. On systems with large load dynamics, the amount of external loads on the processors involved in application execution can change drastically during the course of application execution. During training of the models, we measured available CPUs and available bandwidths on all processors and links involved in application execution at periodic intervals of time from the beginning to the end of the application execution. We then calculated for each processor and link, *avg_avail_cpu* and *avg_avail_band* respectively. These are the averages of the periodic available CPUs and available bandwidths collected for the processor and the link respectively, during the application execution. Finally, we calculated *min_avg_avail_cpu* and *min_avg_avail_band* values by finding the minimum of *avg_avail_cpu* and *avg_avail_band* values respectively, on all processors and links. The *min_avg_avail_cpu* and *min_avg_avail_band* values were used along with the problem sizes and execution times for training the models.

The trained models were used to predict the execution time of the application when executed with a particular problem size and for particular values of *min_avg_avail_cpu* and *min_avg_avail_band* of the system. We forecast the *min_avg_avail_cpu* and *min_avg_avail_band* values based on the history of previous measured values and use the forecasted values in our models to predict the execution time of an application. The history was continuously updated with new measured values. Thus, we forecast the load dynamics of the system based on the observed load dynamics. We used the forecasting tools from NWS to forecast *min_avg_avail_cpu* and *min_avg_avail_band*.

The model based on rational polynomials for predicting execution time, $t_{predicted}$, is given

by:

$$t_{predicted} = O(n^{comp}) + \frac{O(n^{comp})}{min\_avg\_avail\_cpu} + \frac{O(n^{comm})}{min\_avg\_avail\_band} \tag{3.5}$$

where $O(n^{comp})$ is the computational complexity and $O(n^{comm})$ is the communication complexity of the application. The *min_avg_avail_cpu* and *min_avg_avail_band* used in Equation 3.5 were measured values during training the model and forecasted values for prediction.

The advantage of the proposed model was that the application developer only needed to specify the approximate computational and communication complexities of the parallel application. The exact parameters were learned by regression with few trial runs.

3.3.4 Experiments and Observations

We evaluated the rational polynomial based model on 8-processor and 32-processor systems with large load dynamics by predicting execution times for 3 parallel applications: Eigen value problem, Fast Fourier Transforms (FFT), and Conjugate Gradient (CG). Approximate complexities of the applications and approximations of the load dynamics were used to parameterize the model for the applications. In all cases, the model gave less than 20% average percentage prediction errors. Models with prediction errors of less than 30% are considered to be reasonable in the literature especially when used to predict execution times in non-dedicated environments with high load dynamics.

The above techniques were tested on random load dynamics which may not be realistic grid load. In this approach, we expect that approximate complexities of the application are provided to our models by the user. This will not be applicable in all cases, since a developer will not be able to provide the complexities of the application.

By keeping this work as motivation, we planned to further investigate performance modeling of parallel applications. We planned to develop techniques to predict execution times for different types of load dynamics in the environment and for varying number of processors. We also plan to develop systematic methods to determine the approximate complexities of any parallel application automatically.

3.4 Comprehensive Performance Modeling Strategies

Although our preliminary performance models based on rational polynomials gave accurate predictions for non-dedicated clusters, the models had few limitations. Our preliminary performance models can predict execution times only for a fixed set of processors for varying problem sizes. Techniques that can predict execution times for varying number of processors should be developed. In the approach, we expect that approximate complexities of the application will be provided to our models by the user. In many cases, the user or application developer does not have information about complexities of the application. Hence, techniques for automatically determining the complexities will have to be built. Our preliminary models were also tested on systems with random load dynamics. Models that can give reasonable predictions for loads corresponding to real grids should be developed.

In this section, we discuss our comprehensive and robust performance modeling strategies that address the limitations of our preliminary performance models based on rational polynomials.

3.4.1 Modeling Equation and Terms

In our modeling method, we calculate the time taken for the execution of parallel application as:

$$T(N, P, minAvgAvailCPU, minAvgAvailBW) = \frac{f_{comp}(N)}{f_{cpu}(minAvgAvailCPU) \cdot f_{Pcomp}(P)} + \frac{f_{comm}(N)}{f_{bw}(minAvgAvailBW) \cdot f_{Pcomm}(P)} \tag{3.6}$$

where

- N: problem size;
- P: number of processors;
- minAvgAvailCPU, minAvgAvailBW: represent the transient CPU and network characteristics respectively.
- f_{comp}: indicates the computational complexity of the application in terms of problem size;

- f_{cpu}: function to indicate the effect of processor loads on computations;
- f_{Pcomp}: used along with computational complexity to indicate the computational speedup or the amount of parallelism in computations;
- f_{comm}: indicates the communication complexity of the application in terms of problem size;
- f_{bw}: function to indicate the effect of network loads on communications;
- f_{Pcomm}: used along with communication complexity to indicate the communication speedup or the amount of parallelism in communications;

In order to calculate minAvgAvailCPU and minAvgAvailBW, we measure *AvailCPU* and *AvailBW*. AvailCPU is a fraction of the CPU that can be used for the application and is inversely proportional to the amount of CPU load. AvailBW of a link is the bandwidth on the link available to an application. AvailBW is usually lesser than the link capacity and is inversely proportional to the network load on the link. Network Weather Service (NWS)[147, 146], a tool for measuring and forecasting system parameters, was used for obtaining AvailCPUs of the processors used for application execution and AvailBWs on the links between the processors. During training the model functions for application, we measure AvailCPUs and AvailBWs on all processors and links involved in application execution at periodic intervals of time from the beginning to the end of the application execution. We then calculate for each processor and link, *AvgAvailCPU* and *AvgAvailBW* respectively. These are the averages of the periodic AvailCPUs and AvailBWs collected for the processor and the link respectively, during the application execution. Finally, we calculate *minAvgAvailCPU* and *minAvgAvailBW* values by finding the minimum of *AvgAvailCPU* and *AvgAvailBW* values respectively, on all processors and links. By considering minAvgAvailCPU and minAvgAvailBW in our modeling strategy, we assume that the slowest processor and link used by the application affect the overall execution time of the application. Our strategy does not consider latencies on the network links since our models are intended for tightly-coupled applications executing within a cluster where the nodes are connected by low-latency links.

3.4.2 Model Correctness

The formula shown in Equation 3.6 splits the execution time of a parallel application into two parts, f_{comp} and f_{comm}, for representing computation and communication aspects, respectively, of the parallel application. This representation is useful for scheduling purposes since a scheduler can allocate the appropriate CPU and network resources for the application based on the computation and communication requirements, respectively, of the application. The scalability of the computational and communication times with increasing number of processors, is represented by f_{Pcomp} and f_{Pcomm}, respectively. Since, in most parallel applications, the execution times decrease with the increase in number of processors, these functions are contained in the denominators. The increase in CPU and network loads on non-dedicated systems increase the computation and communication times, respectively. $minAvgAvailCPU$ and $minAvgAvailBW$ represent the inverse of the CPU and network loads, respectively. Hence, the corresponding functions, namely, f_{cpu} and f_{bw}, are contained in the denominators.

The formula shown in Equation 3.6 generalizes the parallel runtime equations of many parallel numerical drivers that deal with memory-resident data[21, 59]. For example, the parallel runtime equation for a FFT application using binary exchange algorithm on a dedicated homogeneous system is $T = t_c \frac{N}{P} \log N + t_m \frac{N}{P} \log P$ where t_c is the time for a floating point operation and t_m is the transfer time for a unit message. Thus, the various functions of Equation 3.6 for the parallel FFT application are: $f_{comp} = N \log N$, $f_{comm} = N$, $f_{Pcomp} = P$ and $f_{Pcomm} = \frac{P}{\log P}$. Since Equation 3.6 represents parallel runtime for non-dedicated systems, it includes the effects of CPU and network loads in f_{cpu} and f_{bw}, respectively. The values for t_c and t_m are determined as model coefficients using a multi-phase linear regression procedure.

The modeling techniques are intended for simple parallel application kernels which are invoked in complex parallel applications. These parallel kernels have single phase of uniform computations and communications and are integral to many scientific applications. Equation 3.6 represents a coarse-level model and does not include fine-level performance behavior of the applications including computation-communication overlap, memory access stride and range, cache misses and the corresponding system parameters including cache configurations and memory bandwidth. Our model is primarily intended for scheduling purposes and most of

the schedulers of parallel applications[18] do not consider these fine-level parameters. Our performance modeling only considers applications with memory-resident data. While many scientific applications involve extensive I/O, many significant parallel applications like the ChaNGa n-body solver[33], Athena astrophysical gas dynamics code[14], and MPB electromagnetics code[101] deal with data that are entirely resident in the memory.

3.4.3 General Methodology

During training, a number of experiments were conducted by executing the application with different number of processors, problem sizes and under different loading conditions. For each experiment, the minAvgAvailCPU, minAvgAvailBW, and execution time were observed. These parameters along with problem size and number of processors were used for training the model functions of Equation 3.6. To determine the different model functions in Equation 3.6, we consider a list of candidate functions including linear, quadratic, higher-order polynomial, logarithmic, and a combination of logarithmic and polynomial functions. For a particular combination of model functions in Equation 3.6, the coefficients for the functions are determined by polynomial fit using data points in the training samples. The problem of finding the coefficients is a linear regression problem and is equivalent to solving a system of linear equations, **Ax=b**, where the vector x contains the coefficients for the functions used in the equation, matrix A is a e$\times$t matrix where e corresponds to the number of training experiments and t corresponds to the terms of the functions used in the equation, and vector b contains the execution times for the e experiments.

3.4.4 Evaluation of Models

For each combination of model functions of Equation 3.6, we evaluate the combination in terms of standard error defined as:

$$\begin{aligned} SSE &= \sum_{i=1}^{N}(y_i - y_i')^2 \\ Error\_variance &= \frac{SSE}{(n-p)} \\ Standard_error &= \sqrt{Error\_variance} \end{aligned} \tag{3.7}$$

where N denotes the number of experiments, y_i denotes the actual execution time for experiment i, y'_i denotes the predicted execution time for experiment i using Equation 3.6, and p is the number of terms of the functions in the combination. The different combinations of model functions are then compared in terms of their standard error values. Our modeling strategy consists of number of stages for determining the model functions of Equation 3.6. Due to the uncertainties caused by the non-dedicatedness in the environment, instead of choosing a model function with the minimum standard error value, a set of good model functions with low standard error values are chosen at the end of each stage of modeling.

The following subsections detail the individual steps involved in determining the different functions used in Equation 3.6.

3.4.5 Modeling Computation

In order to find the computational complexity, f_{comp} in Equation 3.6 for the application, we use a candidate set of 77 model functions. These are polynomial, logarithmic, and mixture of polynomial and logarithmic functions and are commonly used in many curve fitting packages and tools[41, 38, 89]. These functions also encapsulate the behavior of many parallel applications. We execute the application on a single non-dedicated processor with different problem sizes and observe the minAvgAvailCPUs and execution times. Since most of the systems follow round-robin scheduling, we use $minAvgAvailCPU$ value for f_{cpu}. For each candidate function, $f_{candidateComp}$ for f_{comp}, we fit $\frac{f_{candidateComp}}{minAvgAvailCPU}$ with the experiment data. At the end of this stage, we sort the functions in terms of the ascending order of their standard error values and

Table 3.1: Candidate functions for f_{bw}

S.No.	f_{bw}(minAvgAvailBW)
1.	$\sqrt{minAvgAvailBW}$
2.	$minAvgAvailBW$
3.	$minAvgAvailBW^{1.5}$
4.	$minAvgAvailBW^{2}$
5.	$minAvgAvailBW^{2.5}$
6.	$minAvgAvailBW^{3}$
7.	$\log(minAvgAvailBW)$
8.	$minAvgAvailBW \cdot \log(minAvgAvailBW)$

choose at most 20 candidate functions for f_{comp} for subsequent stages of modeling.

3.4.6 Modeling Communication

To find the communication complexity, f_{comm}, and the effect of network loads on the application, represented by f_{bw} in Equation 3.6, we execute the parallel application on 2 processors with different problem sizes and under different CPU and network loads. For each of the experiments, we observe minAvgAvailCPU and minAvgAvailBW values. We then use the following equation for execution times on 2 processors:

$$T1(N, minAvgAvailCPU, minAvgAvailBW) = \frac{f_{comp}(N)}{2 \cdot minAvgAvailCPU} + \frac{f_{comm}(N)}{2 \cdot f_{bw}(minAvgAvailBW)} \tag{3.8}$$

For f_{comp} in Equation 3.8, we use the 20 functions with top accuracy values determined in the computation modeling phase. For f_{comm}, we use the 77 candidate functions used for computation modeling. For f_{bw}, we evaluate 8 candidate functions shown in Table 3.1.

Thus, we evaluate a total of 12320 ($20 \times 77 \times 8$) combinations of functions for Equation 3.8 and fit the execution times T1 with the actual execution times corresponding to the experiments. For each combinations of functions, we determine the coefficients of the functions using linear regression and determine standard error values. For each of the 20 functions for f_{comp}, we choose at most 10 combinations of functions for f_{comm} and f_{bw} with minimum standard error values to form *filtered_list*. Finally, 20 combinations of f_{comp}, f_{comm} and f_{bw} with minimum

Table 3.2: Candidate functions for f_{Pcomp} and f_{Pcomm}

S.No.	f_{Pcomp}(P), f_{Pcomm}(P)
1.	$\sqrt{P}$
2.	P
3.	$P^{1.5}$
4.	P^2
5.	$P^{2.5}$
6.	P^3
7.	$\log(P)$
8.	$P \cdot \log(P)$
9.	$1/\sqrt{P}$
10.	$1/P$
11.	$1/P^{1.5}$
12.	$1/P^2$
13.	$1/P^{2.5}$
14.	$1/P^3$
15.	$1/\log(P)$
16.	$1/(P \cdot \log(P))$

standard error values are chosen from the *filtered_list* for the next stage.

3.4.7 Modeling Scalability

In the final stage, the scalability functions, f_{Pcomp} and f_{Pcomm} of Equation 3.6 are determined. For this, the application is executed with 2, 4 and 8 processors under different loading conditions and the resulting execution times are observed. We then fit different combinations of functions in Equation 3.6 with the observed execution times. For each of the 20 f_{comp}, f_{comm} and f_{bw} combinations determined at the end of the communication modeling phase, we evaluate different combinations of functions for f_{Pcomp} and f_{Pcomm}. The 16 candidate functions for f_{Pcomp} and f_{Pcomm} are shown in Table 3.2.

Thus, we evaluate a total of 5120 ($20 \times 16 \times 16$) combinations of functions in this stage. For each of the 20 f_{comp}, f_{comm} and f_{bw} combinations determined in the previous stage, we choose at most 50 combinations of f_{Pcomp} and f_{Pcomm} functions with minimum standard error values. Thus at the end of the training phase, we obtain a list of at most 1000 combinations of functions for Equation 3.6. The combinations are sorted in the ascending order of their standard

error values to form *sorted_list*. One of these models is chosen for predicting the execution time of the application at a given point of time.

While choosing the top functions in each stage, we ensure that the chosen functions have standard errors that are at most 20% greater than the minimum standard error and that the number of chosen functions are below a threshold limit. The values for these limits were found by trial-and-error in order to provide flexibility for our prediction strategies to dynamically select, based on resource dynamics, one of the functions with minimum standard error and, yet limit the number of function evaluations for a given prediction.

3.4.8 Modeling Applications with Processor Constraints

Some parallel applications may have constraints regarding the minimum number of processors that they need to execute. In these cases, we cannot use single processor executions to determine the computation complexity functions. In these cases, we first execute the application with different problem sizes and under different loading conditions on the minimum number of processors. We then form different combinations of $\frac{f_{comp}}{minAvgAvailCPU}$, f_{comm} and f_{bw} functions for Equation 3.8 and fit the model shown in the equation with the actual execution times. Evaluating all combinations of model functions is time consuming since it requires evaluation of 47432 combinations (77 for computation $\times$ 77 for communication $\times$ 8 for bandwidth). To prune down the number of evaluations, we form a 77 $\times$ 77 matrix where an (i,j) entry denotes a combination with function f_i for computation and f_j for communication. A random entry in the matrix that is not evaluated is chosen. For this random entry, 8 combinations are evaluated corresponding to 8 bandwidth functions and this entry is marked as evaluated. If use of a computation function i leads to a standard error value greater than a threshold for continuous 5 times, then the function i is marked as not suitable for modeling computation and row i is marked as evaluated. Similarly, if use of a communication function j leads to a standard error value greater than a threshold value for continuous 5 times, then the function j is marked as not suitable for communication and column j is marked as evaluated. Different random combinations are considered for evaluations until all the entries in the matrix are marked as evaluated. We then proceed to the scalability modeling phase.

3.4.9 Prediction of Execution Times

A grid scheduler, before it chooses a set of resources for an application, may want to know the predicted execution time of the application on the set of resources for its decision making. To predict the execution time of an application with a given problem size and given number of processors, the top combination of functions in the sorted_list, obtained in the training phase, is used in Equation 3.6. The prediction of execution time needs values for problem size, number of processors, minAvgAvailCPU and minAvgAvailBW. Problem size and number of processors for execution are obtained from the user. During training the models, the minAvgAvailCPU and minAvgAvailBW values were obtained by observing the system loads during application execution. For prediction of execution time of an application, the minAvgAvailCPU and minAvgAvailBW values have to be predicted and cannot be measured before the application execution, since the values represent the system loads that will exist during the period of application execution. Hence we forecast the values based on the history of load dynamics measured on the system. We use the forecasting tools from NWS to forecast minAvgAvailCPU and minAvgAvailBW. To update the history of measured values, a persistent daemon continuously uses NWS to measure the AvailCPU and AvailBW values on all processors and links respectively, of the cluster. The daemon then calculates the averages of these values every 10 minutes and stores these 10-minute averages in a history database. During prediction of execution time of an application on a subset of processors, the measured values corresponding to the subset of processors and links are extracted from the history database and used in the NWS forecasting tools to obtain predicted values for MinAvgAvailCPU and MinAvgAvailBW.

We then use the problem size and number of processors supplied by the user and the predicted MinAvgAvailCPU and MinAvgAvailBW to predict the execution time of the application. After the application is executed on the resources chosen by a grid scheduler, the actual execution time of the application and the MinAvgAvailCPU and MinAvgAvailBW values are observed and added to the training data along with the problem parameters as a data point. The combinations of functions in the sorted_list are evaluated by means of standard error values with the updated training data. This can lead to reordering of the combinations in the sorted_list. Hence a different combination may be used for the next prediction. In order to avoid evaluating

all the combinations in the sorted_list at the end of every prediction, we eliminate those combinations of functions whose ranks in the sorted_list are within the lowest 10 percentile for 5 continuous predictions. Thus the sorted_list shrinks over time and converges to contain only the most promising combinations. In order to reduce the errors due to non-dedicatedness of the systems and to take into account new load dynamics, the original set of candidate combinations of functions are reconsidered by revisiting the communication modeling phase of training at the end of every 50 predictions. This can lead to addition of new combinations to the sorted_list.

Figure 3.2 illustrates the complete procedure for deriving the model functions for predicting execution times.

3.5 Cross-Platform Performance Modeling

We have also developed techniques whereby the sorted list of model functions and training data obtained for an application on a given parallel platform called *reference platform* can be used for predicting execution times of the application on another parallel platform called *target platform*. This *cross platform performance modeling* is helpful in avoiding the steps needed in forming the sorted list on the target platform, including conducting training experiments, performing computation and communication modeling, and evaluating different combinations of functions.

In order to use a combination of functions obtained on a *reference platform* for a *target platform*, the coefficients of the different functions of Equation 3.6 have to be scaled to take into account the performance difference between the reference and the target platforms. Since the minAvgAvailCPU is a percentage, the same percentage or same amount of CPU load on two different platforms can indicate two different transient CPU speeds. Hence, the computation complexity f_{comp} have to be scaled by a CPU scaling factor.

The CPU scaling factor, *Rcpu*, is determined by obtaining single processor execution times of the application with a moderate problem size on dedicated target and reference processors, and finding the ratio between the two times. In order to determine the scaling factor for the communication complexity, we conduct dedicated 2-processor experiments with different problem sizes on the reference and target platforms. The top model in the sorted_list is then fitted with

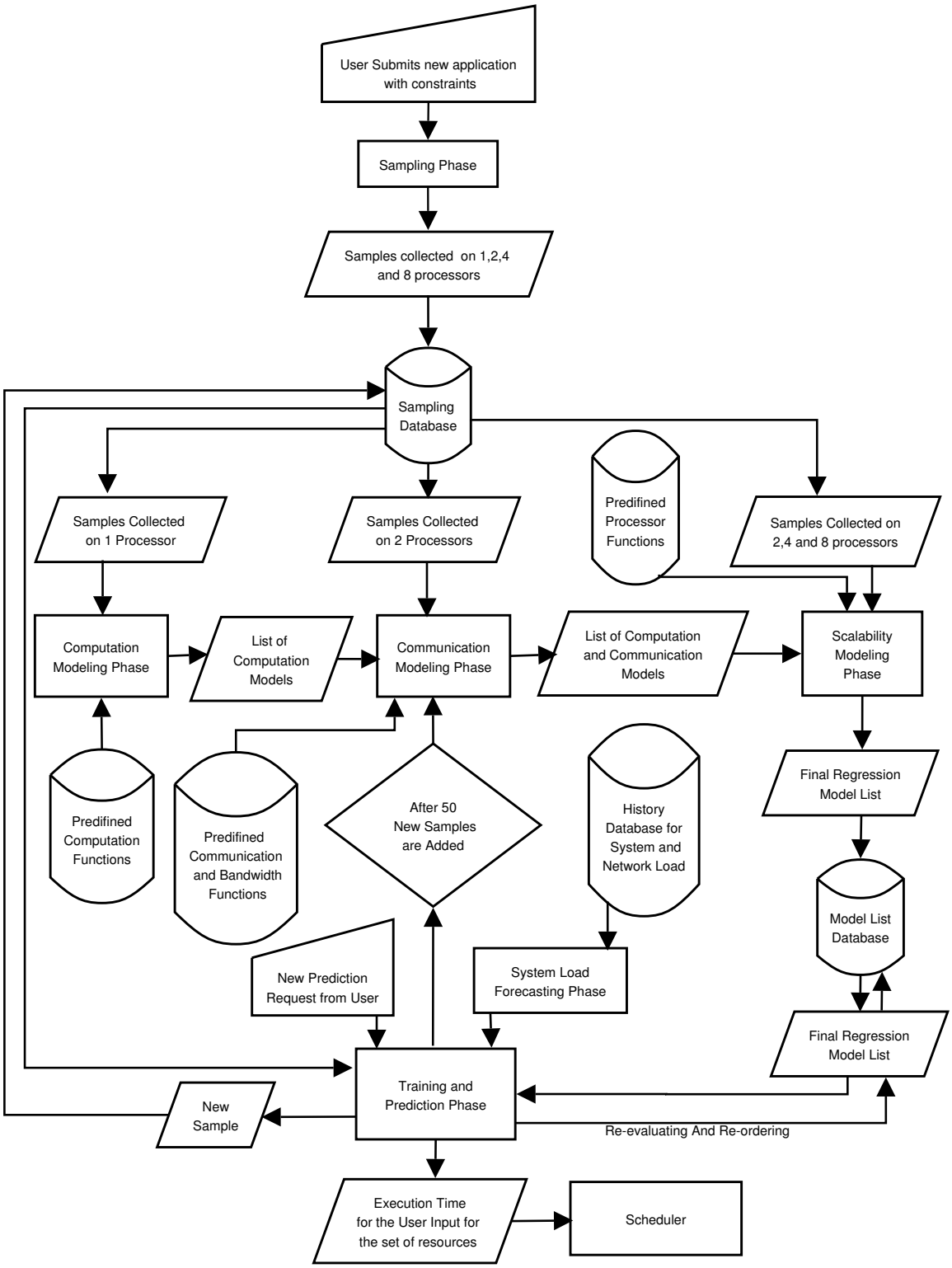


Figure 3.2: Illustration of Modeling Procedure

the 2-processor experiment data on the reference platform. The coefficients of the functions in the top model and the resulting goodness of fit are determined. The trained function is then fitted with the 2-processor experiment data on the target platform after scaling the computation complexity with *Rcpu*. The communication complexity of the resulting function is then scaled with different scaling factors until the goodness of fit for the 2-processor experiment data on the target platform matches with the goodness of fit obtained on the reference platform. The resulting scaling factor is noted as the bandwidth scaling factor, *Rbw*. *Rbw* may be a non-unit real number due to difference in communication and computation overlaps in the reference and target platforms.

For predicting the execution time of an application on a target platform, the top model in the sorted_list with the minimum standard error, obtained on the reference platform, is considered. The coefficients of the computation and communication functions in the model are obtained by one of the following two methods based on prediction accuracies of the methods for the previous experiments conducted on the target platform. When the number of executions on the target platform are insufficient for training the top model, the coefficients of the functions in the model are obtained by training the model with the data corresponding to non-dedicated executions for different problem sizes and processors available for the reference platform. If f_{comp}, f_{comm}, f_{Pcomp}, f_{Pcomm}, f_{cpu}, f_{bw} are the functions for the top model obtained on the reference platform, the predicted execution time of the application with problem size N and P processors of the target platform is then given by:

$$T_{predicted} = \frac{Rcpu \cdot f_{comp}(N)}{f_{cpu}(minAvgAvailCPU) \cdot f_{Pcomp}(P)} + \frac{Rbw \cdot f_{comm}(N)}{f_{bw}(minAvgAvailBW) \cdot f_{Pcomm}(P)} \tag{3.9}$$

The minAvgAvailCPU and minAvgAvailBW values shown in Equation 3.9 are obtained by forecast of CPU and network loads on the target platform. When the number of executions on the target platform are sufficient for training the top model, the coefficients of the functions in the top model are obtained by training the model with the training data available on the target platform. The parameters corresponding to the application execution on the target platform are then added to the training data and the coefficients of the functions in the sorted_list are

recalculated using the updated training data. The top model in the list with the recalculated coefficients are used for the next prediction.

3.6 Performance Models for Multi-Phase Applications

For predicting the execution times of large-scale parallel applications with multiple phases of computation and communication complexities, the major phases of execution in the applications will have to be determined. There are a number of strategies for automatic phase detection and prediction in parallel applications[44, 46, 110]. For our work, we mark the phases at compile time. We use the work by Shen et al.[122, 121] that uses a technique called *active profiling* for identifying execution phases. Active profiling uses controlled inputs and analysis of execution traces of basic blocks to identify candidate phase markers or phase boundaries, real inputs to eliminate false positives, and detailed analysis for identifying inner phase markers. For some parallel applications not amenable to active profiling, we use manual analysis of high-level structure of the source code and consider long-running subroutines and loop-nests as candidate phases[46]. For each of the detected phases, we can then use the CPU and network load measurements and execution times within the phase boundaries to derive per-phase performance models as shown in Equation 3.6. The predictions by the per-phase performance models can be used for analyzing the behavior of individual phases, giving accurate predictions for the entire applications, deriving efficient schedules for individual phases and determining potential points of rescheduling.

3.7 Usage Scenario: Putting It Altogether

When a tightly-coupled parallel application is integrated into the grid by the application developer, the minimum problem size and the minimum number of processors for the application are specified by the developer. A small set of resources in a cluster is then chosen for training an initial list of performance model functions for the application. In this training phase, the application is executed with different problem sizes, number of processors, CPU and network loads, and the resulting execution times are observed. These parameters are used to determine

the coefficients of various combinations of model functions for computation and communication complexities, scalability, and effect of external loads on the application. At the end of this phase, a filtered list of combinations of performance model functions with small standard error values are formed. The filtered list of functions are ordered in the ascending order of standard error values. These functions can predict execution times of the application for a given set of problem and resource characteristics on the cluster where the model functions were trained. We refer to this cluster as a *trained* cluster. At this stage, only one cluster in the grid is trained and the other clusters are untrained.

The results of the training phase, namely, the ordered list of performance model functions and the training data, are then ported from one of the trained clusters to the untrained clusters by scaling the coefficients of the functions. The scaling factors are determined by conducting single-CPU and 2-processor experiments with the application on dedicated processors of the trained and an untrained cluster and observing the performance difference. When a user submits a problem to the grid corresponding to the application with a given problem size, the grid scheduler evaluates different sets of resources from different clusters. For a given candidate set of resources in a cluster with a given set of resource characteristics, the grid scheduler uses the top performance model function in the ordered list, with minimum standard error value, for the cluster to predict the execution time of the problem. The grid scheduler then chooses a set of resources in one of the clusters for application execution.

After the execution of the application on the set of resources in a cluster, the various parameters corresponding to the application including the problem size, the number of processors, the resource characteristics and the execution time are added to the training data for the cluster. The functions in the ordered list are reevaluated with the updated training data resulting in change in the coefficients of the functions and potential reordering of the functions in the ordered list in terms of their standard error values. The new top performance model function in the reordered list is used for the next prediction in the cluster. Thus, different performance model functions can be used at different points of time based on the changing load conditions on the resources. After every 50 predictions, the set of filtered performance models is reformed by evaluating the initial list of performance models with the updated training data. Thus, by

reevaluating the filtered performance models list at the end of every execution and reforming the filtered performance models list at the end of 50 executions, our strategy adapts itself to the grid load dynamics.

When an untrained cluster is used for sufficient number of executions of the application, the parameters of the executions are used for training the initial list of models on the cluster to reform the filtered ordered list on the cluster. This changes the state of the cluster from untrained to trained.

3.8 Experiment Setup

Experiments were conducted validating the modeling strategies in terms of predictions of execution times. In order to avoid intrusion on production grid systems, we conducted experiments on systems that are under our administrative control.

The experiments were conducted on four different clusters shown in Table 3.3. As can be seen in the table, the four clusters have completely different processor and network characteristics and represent the typical heterogeneous nature of grid systems. The IBM cluster was used in a dedicated mode with the help of space-sharing using IBM Load-Leveler batch queuing system. The other clusters were used in non-dedicated modes. On the dual-core AMD and Woodcrest systems, *taskset* option was used to control the process assignment to the cores. On all the clusters, 1,2,4 and 8 processors were used for training the models and up to the maximum number of available processors were used for predictions. During training, about 30 problem sizes within a small problem size range were used and during predictions, random problem sizes within and outside the training problem size range were used.

We have tested our prediction strategies with 7 different parallel applications.

1. ScaLAPACK[21] Eigen value solver for a double precision symmetric matrix using PDSYEV kernel. 2-D block cyclic distribution was used.

2. 1-D FFT application from FFTW[59] package. The transform data are distributed over multiple processes. The application uses routines for parallel one-dimensional transforms of complex data.

Table 3.3: Experiment Infrastructure

Cluster	Maximum Number of Processors Used	Specifications
Intel Cluster	8	Intel Pentium IV, 2.8 GHz, running Fedora Core 2.0, Linux 2.6.5-1.358 operating system, with a 512 MB RAM, a 80 GB hard disk, and connected by a 100 Mbps switched Ethernet.
IBM Cluster	32	IBM P720 system arranged in 8 nodes, each node is a 4-way IBM Power 5 SMP with hard disk capacity of 146 GB and running Suse Linux 9.0 sp 1 operating system. Each processor in a node has 1.65 GHz CPU speed and 1 GB RAM. All the 8 nodes are connected to each other by Gigabit Ethernet links through Gigabit Nortel switch.
AMD Cluster	16	8 Dual-Core AMD Opteron 1214 based 2.21 GHz Sun Fire servers running CentOS release 4.3 with 2 GB RAM, 250 GB Hard Drive and connected by Gigabit Ethernet.
Woodcrest Cluster	24	12 Dual-Core Intel Xeon 5130 based 2.0 GHz HP xw6400 Workstations running Fedora Core 6 (Zen), with 4 GB RAM, 160 GB Hard Drive and connected by 100 Mbps switched Ethernet.

3. Conjugate Gradient (CG) application to solve a system of linear equations with a real symmetric positive definite matrix. Diagonal matrix is used for preconditioning. Row-wise block striped partitioning was used.

4. Molecular dynamics simulation (MD) of Lennard-Jones system systolic algorithm. N particles are divided evenly among the P processes running on the parallel machine. The calculation of forces is divided into P stages. The traveling particles are shifted to the right neighbor processor in a ring topology.

5. Poisson Solver using 2-D Jacobi. Uses 1-D domain decomposition with non-blocking communication. The domain decomposition is along the x-axis. The application is run until the maximum number of Jacobi iterations are reached.

6. Integer Sort (IS). Parallel Integer sort application. Local sorting by individual processes followed by a global sort across processes.

7. Symmetric Successive Over-Relaxation (SSOR) using red-black ordering. Cartesian topology is used for arrangement of processes. Grid points are updated using five-point finite-difference stencil.

During the course of an application execution, available CPUs of the nodes and available bandwidths of the inter-node links are collected every 2 minutes. We used synthetic loads on our system to simulate the load conditions on real grid systems. 2 different loading conditions were used for our experiments: *random* and *grads*. In random loading, synthetic CPU and network loading programs were continuously run on the processors in the background. For loading the CPUs of a system at a given point of time, a set of processors was randomly chosen out of the available processors in the system and synthetic loading programs were run on the processors in the set. The amount of loading on each processor was randomly varied such that the available CPU value of the processor is varied between 6.5%-72% of the total CPU. Small available CPU percentages imply large loading of the processor. The duration of the load is also randomly varied between 3-8 minutes. This process of random selection of processors, introducing random amounts of loads on the processors, and maintaining the loads for random

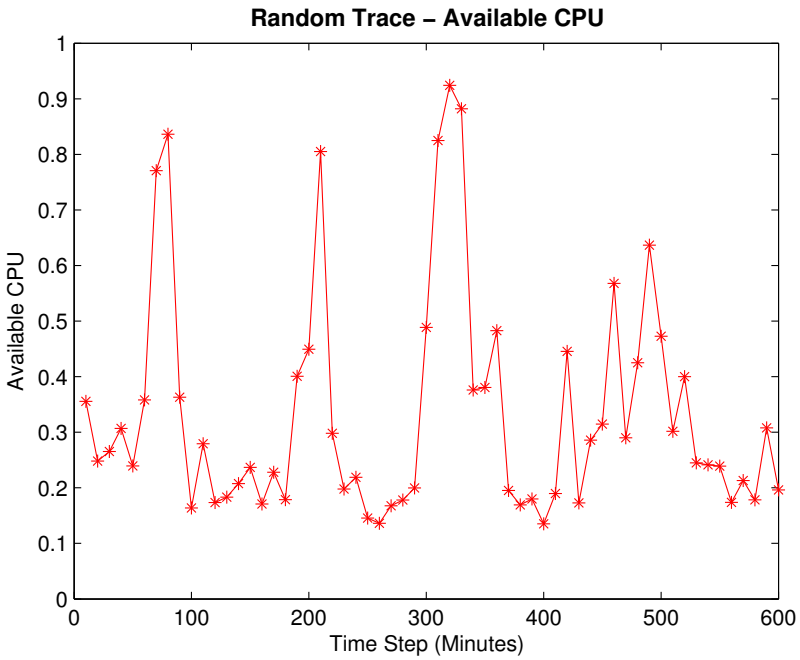


Figure 3.3: Available CPU under Random Loading Conditions

durations of time, is repeated continuously on the system. For network loading, we used a loading program to introduce synthetic network loads on the links of the system and to reduce the available bandwidths of the links. The amount and duration of the load can be specified to the loading program. The loading program takes as input a source and destination host. It then continuously sends packets of fixed sizes from the source to the destination thereby reducing the end-to-end bandwidth from the source to the destination host. At a given point of time, a random number of source-destination pairs is chosen out of all possible source-destination pairs in the system. Random amounts of network loads are introduced on the links between the source-destination pairs by running the synthetic network program so as to vary the available bandwidths of the links between the hosts from 2% to 80% of the total bandwidth capacities of the links. The network loads are maintained for random durations between 3-8 minutes. Similar to CPU loading, the network loading process is repeated continuously. Random loading was used for most of our experiments. Figures 3.3 and 3.4 show the available CPU and available bandwidth values respectively, for a 3-hour period on a processor and link respectively, in the Intel system for random loading conditions.

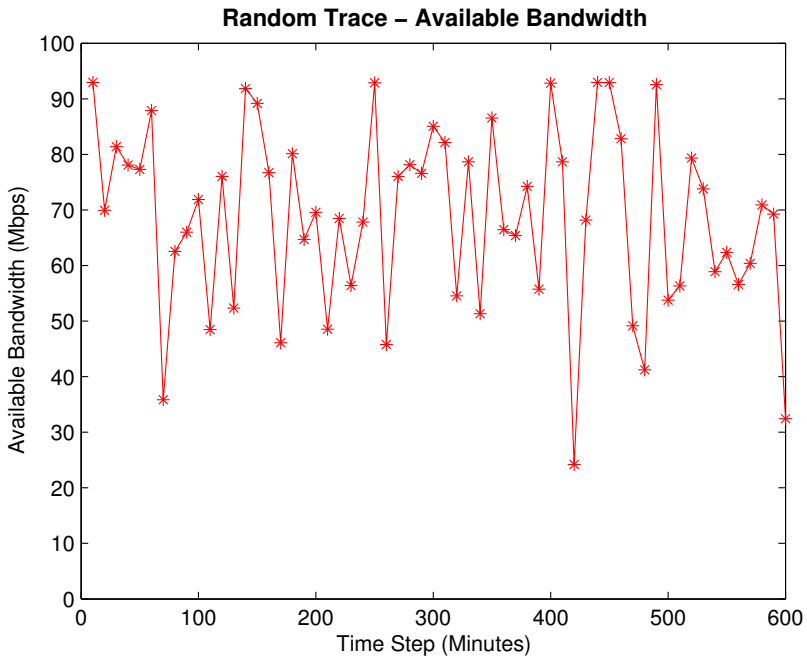


Figure 3.4: Available Bandwidths under Random Loading Conditions

In the grads loading, the traces[74] of CPU and network loads measured by NWS in the GrADS[73] grid research bed were used to guide the parameters to our synthetic CPU and network loading programs. These parameters include the amount and duration of each CPU and network loading, on each processor and link respectively. Grads loading was used for some of our experiments on the Intel cluster. The load traces on nodes of the GrADS research bed, that had similar CPU and network characteristics to our Intel cluster, were used for loading. Even though we use simulated load conditions of GrADS testbed, we verified that the load dynamics introduced by our simulated loads matched with those on the machines of the GrADS testbed[74]. By validating our modeling strategies for the loading conditions that exist on a real grid system, we demonstrate the applicability of our work to scheduling under existing grid conditions. Figures 3.5 and 3.6 show the available CPU and available bandwidth values respectively, for a 3-hour period on a processor and link respectively, in the Intel system for grads loading conditions.

From Figures 3.3-3.6, we find that the CPU loads on the GrADS real grid testbed are more stable than the random CPU loads. However, the large variations in the random CPU loads

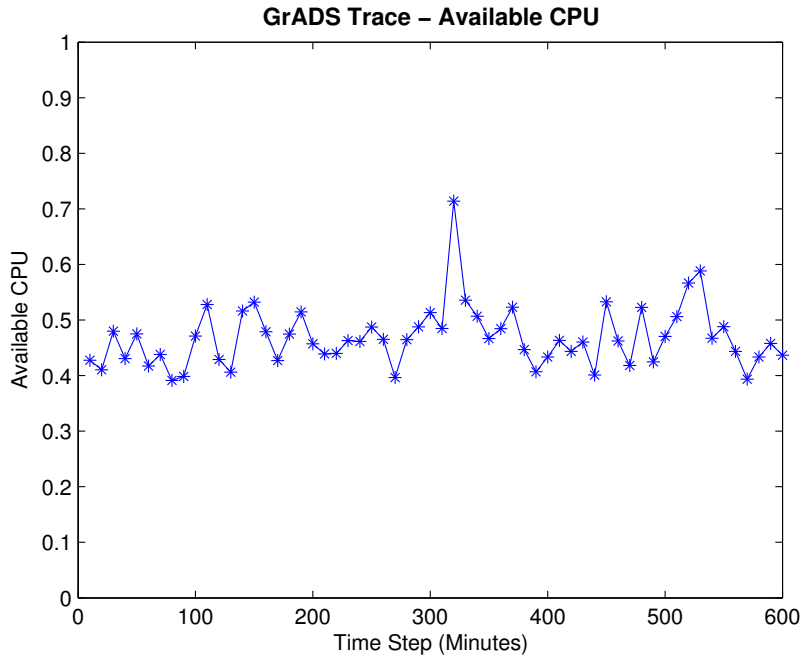


Figure 3.5: Available CPU under Random Loading Conditions

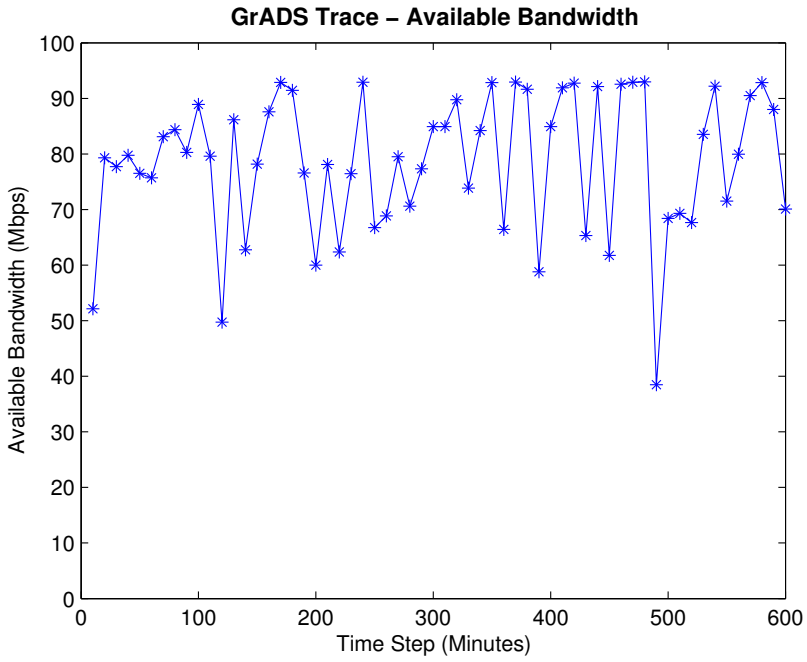


Figure 3.6: Available Bandwidths under Random Loading Conditions

help to represent systems with high load dynamics and hence help in better validation of the robustness of our performance modeling strategies than the grads CPU loads. We find that the random network loads have similar dynamics as the grads loads and hence are realistic. Thus, by conducting experiments both on extreme load dynamics and real grid conditions, we show the usefulness of our models under different conditions.

For evaluating the modeling strategies, we calculated the average percentage prediction errors, and also determined the usefulness of the strategies for scheduling. The average percentage prediction error is given by:

$$\frac{1}{N}\sum_{i=1}^{N}\frac{|(actual_i - predicted_i)|}{|actual_i|}$$

where *N* is the number of experiments, $actual_i$ is the measured execution time and $predicted_i$ is the predicted execution time by the model for experiment *i*. The average percentage prediction errors show only the average behavior and does not show the specific strengths and weaknesses of the modeling strategies. Our modeling strategies are primarily meant to improve the quality of scheduling in grid systems. Hence a more useful evaluation related to scheduling is to compare the minimum execution times predicted by our model and the minimum actual execution times for various problem sizes. We first split the problem sizes of an application into different groups such that the actual execution times of the application when executed with different problem sizes in a group on a dedicated system differ by a maximum of 50 seconds. For each of the problem size groups, we obtain actual and predicted execution times for different problem sizes in the group and different number of processors. We then obtain the problem size, *minPredictedProblemSize*, and number of processors, *minPredictedProcessors*, corresponding to minimum of the predicted execution times. We then obtain the actual execution time, *actualForminPredicted*, corresponding to *minPredictedProblemSize* and *minPredictedProcessors*. We compare *actualForminPredicted* with *minActual*, the minimum of actual execution times. The percentage difference between the two values indicate the loss in efficiency a grid scheduler would incur if it uses the predictions by our modeling strategies for evaluating the candidate schedules. For a perfect scheduler, the percentage difference should be 0.

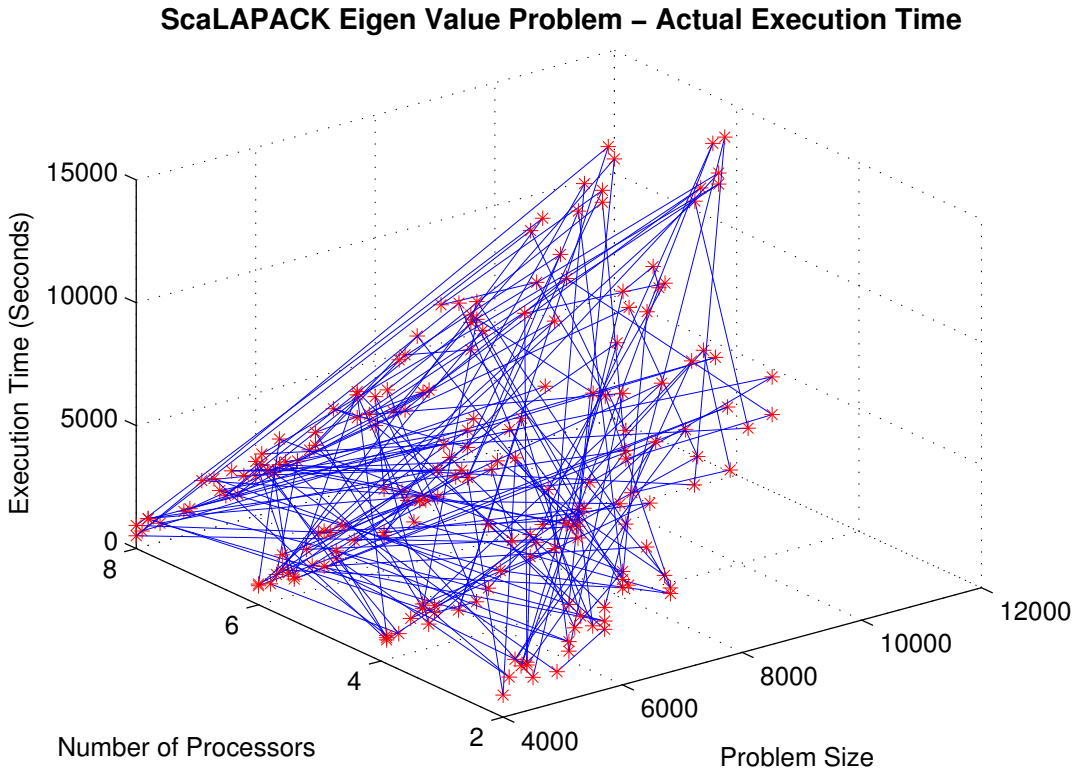


Figure 3.7: Actual Execution Times for ScaLAPACK Eigen Value Problem on the Intel Cluster with Random Loading

3.9 Results

Figure 3.7 shows the actual execution times for ScaLAPACK Eigen value problem for different problem sizes and number of processors on Intel system with random loading conditions. As can be seen in the figure, due to the presence of external loads, there is no strict correlation between execution times and number of processors and/or problem sizes.

Figure 3.8 plots the percentage prediction errors for different problem sizes and number of processors for the Eigen value problem. We find that the percentage prediction errors are less than 30% for 72% of the predictions and less than 40% for 85% of the predictions.

Figure 3.9 plots the percentage prediction error values for different experiments in the order the experiments were conducted for the Eigen value problem. We find that during the initial experiments, the error values are high with some predictions having about 40-85% percentage prediction errors. In the latter stages, the error values converge to 20-30%. The figure also

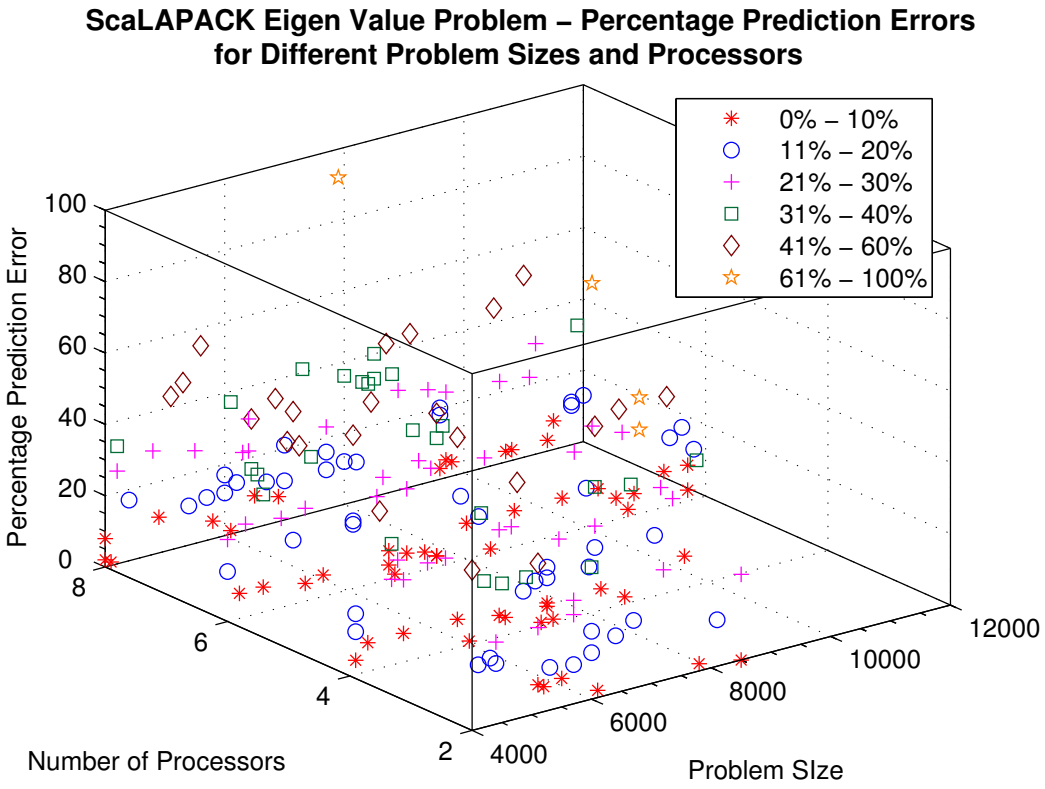


Figure 3.8: Percentage Prediction Errors (PPE) for ScaLAPACK Eigen Value Problem on the Intel Cluster with Random Loading

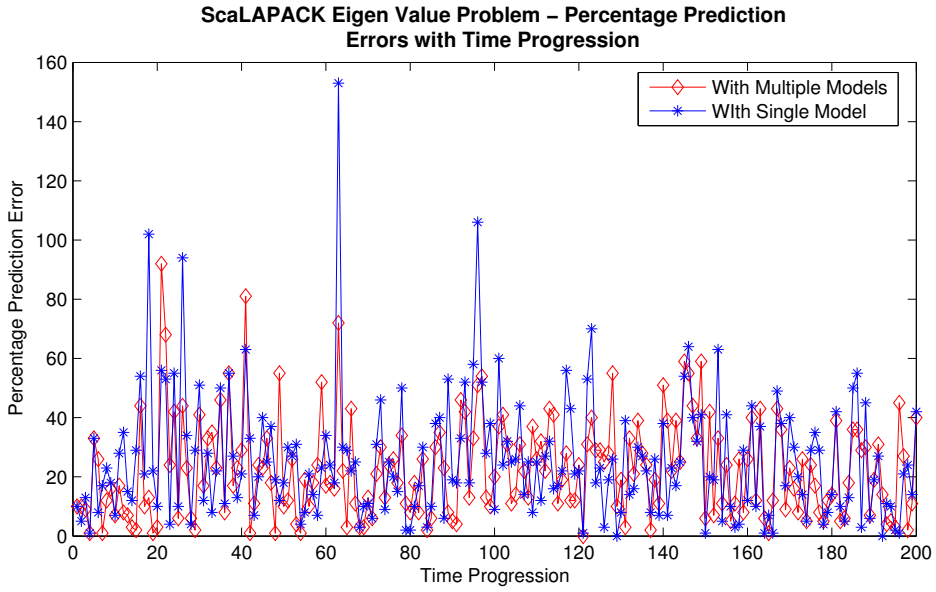


Figure 3.9: Percentage Prediction Errors (PPE) at Different Times for ScaLAPACK Eigen Value Problem on the Intel Cluster with Random Loading

shows predictions by a single model for all the experiments. We chose the model that was the best or had minimum standard error at the end of the training phase for these predictions. As can be seen, using a mixture of good models and choosing different models at different points of time give smaller prediction errors than using a single model for predictions on non-dedicated environments. The average percentage prediction errors for all experiments using multiple models is 22.47% with a standard deviation of 16.73% while the average using a single model is 25.41% with a standard deviation of 20.79%. Thus our strategy of choosing different models for predictions is adaptive to the load dynamics on the grid systems.

Table 3.4 indicates the usefulness of prediction methodology for scheduling a ScaLAPACK Eigen value problem on the non-dedicated Intel cluster. The last column denotes the percentage difference between *actualForminPredicted* and *minActual* and is indicative of the loss in efficiency of a scheduler when using the predicted execution times by our modeling strategies. As can be seen, the maximum percentage increase in execution time when a scheduler uses our predicted times is only 8%. In a grid system with high load dynamics, where it is difficult to achieve perfect scheduling, the loss in efficiency of the scheduler by only 8% is tolerable. In

Table 3.4: Usefulness of Predictions for Scheduling of Eigen Value Problem on Intel Cluster with Random Loading

Problem Size Group	Problem Size (N) and Processors (P) for Minimum Predicted Execution Time: (1)	Problem Size (N) and Processors (P) for Minimum Actual Execution Time: (2)	Actual Execution Time for (1): (3)	Actual Execution Time for (2): (4)	Percentage Increase in Execution Time due to Prediction ((3-4)/4)
4700-4900	4800,8	4800,6	987.27	951.02	3.00%
5100-5300	5100,6	5100,8	850.76	824.66	3.00%
5500-5700	5500,8	5700,4	1153.53	1060.72	8.00%
6200-6400	6300,8	6300,8	1572.39	1572.39	0.00%
6700-6900	6700,8	6700,8	1786.28	1786.28	0.00%
7000-7200	7100,8	7100,6	2502.56	2331.02	7.00%
8100-8300	8100,8	8100,8	3002.43	3002.43	0.00%
9500-9700	9600,8	9600,8	4394	4394	0.00%

some of the cases, the scheduling decisions will be exact as can be seen by the 0 values in the last column.

Similar results were obtained for ScaLAPACK Eigen value solver, 1-D FFT, Conjugate Gradient, Molecular dynamics simulation, Poisson Solver, Integer Sort and Symmetric Successive Over-Relaxation applications on Intel, AMD and Woodcrest clusters with both random and grads loading. These results are shown in Appendix 8.1.

Tables 3.5 and 3.6 summarize the results regarding percentage prediction errors obtained on the 8-processor Intel, 16-processor AMD and 24-processor Woodcrest clusters. The results demonstrate that our modeling strategies give good predictions for both random loading conditions and the loading conditions that exist on one of the current grid systems, and for higher number of processors than those used during the training.

In summary, in all our experiments, 48-98% of predictions were obtained with less than 30% prediction error and 73-100% of predictions were obtained with less than 40% prediction error. The average percentage prediction error is less than 30% in all cases. Although these

Table 3.5: Predictions on the 8-processor Intel Cluster

Problem	Load Type	PPE (Single Model)		PPE (Multiple Models)	
		Avg.	Std. Dev.	Avg.	Std. Dev.
Eigen	random	25.41	20.79	22.47	16.73
Eigen	grads	40.5	37.2	27.93	21.2
FFT	random	31.16	26.9	25.16	22.63
CG	random	22.2	16.22	20.82	15.79
MD	random	18.85	13.66	16.66	14.30
MD	grads	10.05	7.47	8.7	7.67
Poisson	random	31.62	24.96	27.86	22.36
IS	random	11.53	9.01	11.68	9.38
IS	grads	11.00	14.8	11.4	15.59
SSOR	random	22.98	19.65	16.22	13.78
SSOR	grads	10.5	11.8	9.69	8.07

Table 3.6: Predictions on 16-processor AMD and 24-processor Woodcrest (WC)

Prob. : Cluster	Load	PPE (Single Model)		PPE (Multiple Models)	
		Avg.	Std. Dev.	Avg.	Std. Dev.
Eigen: AMD	random	34.7	23.2	23.3	19.6
Eigen: AMD	grads	17.38	14.96	16.61	11.30
MD: AMD	random	23.7	19.7	25.05	20.2
MD: AMD	grads	12.06	11.08	13.09	13.34
Poisson: AMD	grads	14.47	12.96	15.95	13.88
SSOR: AMD	grads	35.25	83.93	13.49	11.07
IS: AMD	grads	32.02	57.89	13.88	16.46
Poisson: WC	random	37.41	19.53	28.78	14.37
IS: WC	random	38.24	23.73	27.7	20.93

percentages are high in the context of predictions on unloaded environments, these numbers are reasonable and to our knowledge, the best reported on non-dedicated environments. Using multiple models was found to be beneficial over using a single model for prediction in almost all cases. Finally, scheduling using our predictions will result in perfect scheduling in many cases and the maximum loss in efficiency of a scheduler when using our models is only 11%.

3.9.1 Comparison with a Modeling Strategy for Non-Dedicated Systems

Comparison of our earlier simpler model for non-dedicated systems with the models for dedicated systems were made in our previous work[151]. The results showed that our earlier simpler models gave better predictions than the models for dedicated systems on non-dedicated environments. Hence our current robust models can only perform better than the models for dedicated systems on non-dedicated environments. In this subsection, we compare the predictions by our modeling strategies with the modeling strategy by Schopf[119, 118, 120] for non-dedicated systems. In this strategy, the model for an application is built from component models corresponding to application components. The predictions for execution times are given as stochastic values, i.e. a range of values, instead of point values. The component models are constructed based on detailed analytical knowledge of the components. Thus the work by Schopf requires the intervention by the application developer. Nevertheless, we compare our work with Schopf's in terms of prediction accuracies. We consider the same 2D Symmetric Successive Over Relaxation (SSOR) problem that they had considered in their work. Stochastic values of available bandwidths and available CPUs based on the NWS traces were input to the Schopf's model for predicting execution time of an application with a given set of application parameters. Figures 3.10 and 3.11 show the comparison results for SSOR on Intel system with random loading conditions for 4 and 8 processors respectively. The results for 2 and 6 processors are shown in Appendix 8.1. The graph shows the actual execution time, the maximum, minimum and average prediction execution times by Schopf' and the predicted execution times by our modeling strategies. We find that the average percentage prediction errors due to our modeling are lesser than the average percentage prediction due to Schopf' modeling. Except for few points, our predicted execution times closely follow the actual execution times. We also find that for large

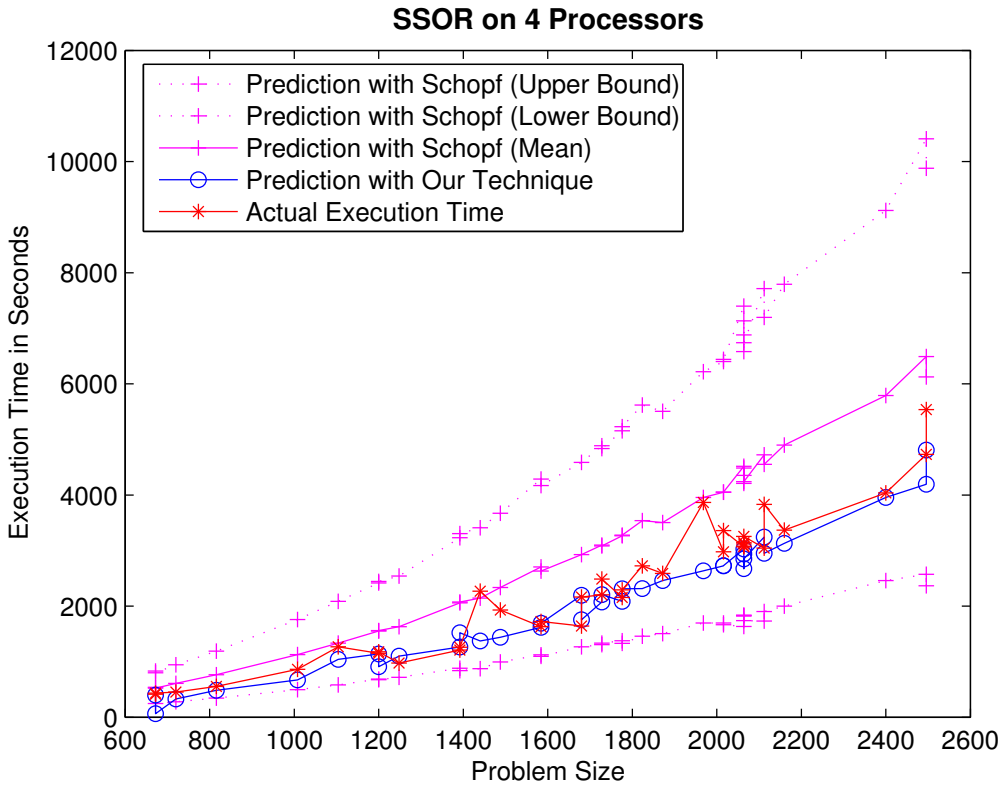


Figure 3.10: Comparison of Our Adaptive Method and Schopf' Method for 4 Processors with SSOR on the Intel Cluster with Random Loading; Schopf' Method - Average: 37.25%, Standard Deviation: 18.09%; Our Method - Average: 14.39%, Standard Deviation: 14.11%

problem sizes, the stochastic values by Schopf have large ranges. These large stochastic values will not be useful for grid schedulers to make scheduling decisions.

3.9.2 Assuming Uniform Loading Conditions

All the previous efforts in predicting parallel execution times assume uniform loading conditions during application execution, i.e. the CPU and network loads remain constant during an application execution. In this subsection, we investigate the impact of this assumption on the prediction accuracies. For this, the same modeling strategies described in Section 3.4 is used for predictions. However, for Equation 3.6, instead of using the average CPU and network loads during application executions for training the models and using predictions of these loads during predicting execution times, we use the average CPU and network loads that existed at the

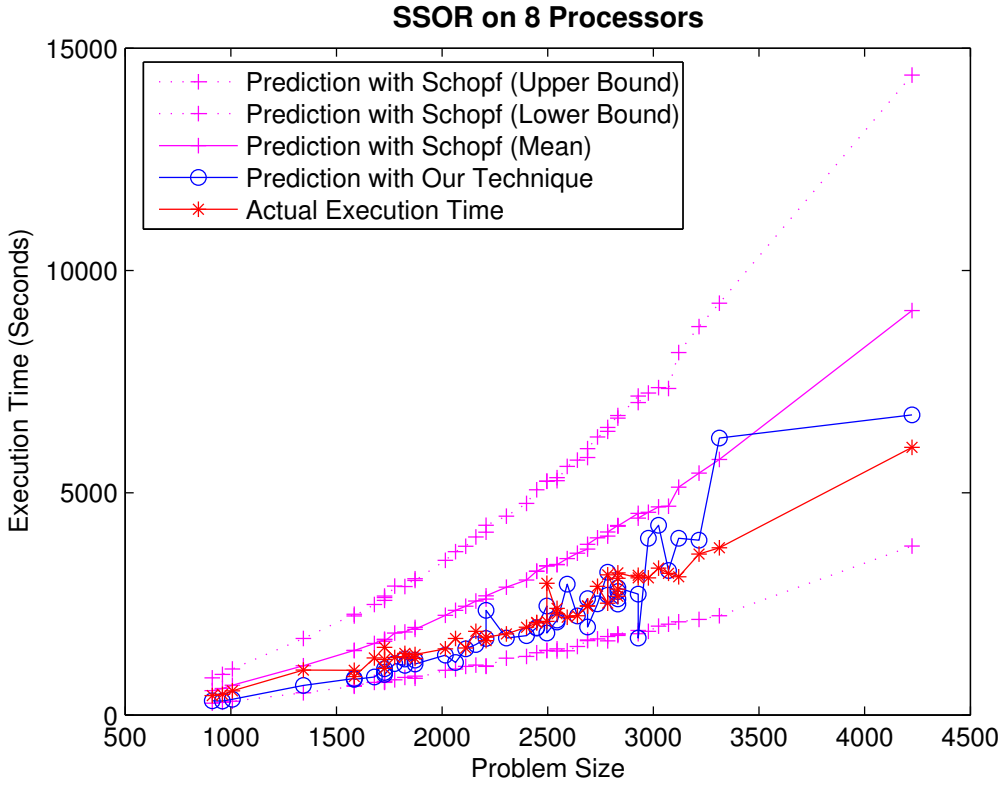


Figure 3.11: Comparison of Our Adaptive Method and Schopf' Method for 8 Processors with SSOR on the Intel Cluster with Random Loading; Schopf' Method - Average: 45.17%, Standard Deviation: 14.18%; Our Method - Average: 17.39%, Standard Deviation: 13.49%

Table 3.7: Results with grads Loading - Assuming Uniform Loading Conditions during Application Executions

Problem	Cluster	Percentage Prediction Errors (PPE)	
		Avg.	Std. Dev.
Eigen	16-proc. AMD	20.38	20.55
Poisson	16-proc. AMD	23.11	24.48
SSOR	16-proc. AMD	34.48	66.77
IS	8-proc. Intel	37.76	87.57
MD	8-proc. Intel	21.30	19.78

beginning of application executions, both for training the models and for predicting execution times. Some of the results obtained using this approach are shown in Table 3.7. Comparing these results with the corresponding results obtained using our strategies and shown in Tables 3.5 and 3.6, we find that assuming uniform loading conditions during application executions can result in high prediction errors.

3.9.3 Modeling Overheads

Table 3.8 details the various overheads in deriving a performance model using some of the techniques which are representative of the other performance modeling strategies that were described earlier. We find that some of the strategies require user intervention to manually analyze the source code of the application and the associated overhead cost is highly non-deterministic. The number of experiments needed to derive models in some strategies are functions of number of processor and problem size configurations. Our modeling strategy does not need source code analysis by the user and need the smallest number of experiments to derive the models. The time taken for a prediction with our model is 2-4 seconds. The primary reason for the small number of experiments in our strategy is that we do not require our initial models to be accurate. Rather, we continuously evaluate our models as applications are executed in grid systems. These evaluations are conducted simultaneously with the predictions, i.e. a scheduler can make use of a previous model for immediate predictions while the model is refined for use in later predictions.

In terms of implementation in production grid systems, all modeling strategies require the

Table 3.8: Comparison of Modeling Overheads

Strategy	**Overheads**
Adve and Vernon[2]	(Overhead for deriving a deterministic graph for the application) + ([number of problem size configurations] × [number of processor configurations] × [number of tasks in the application]) experiments for model derivation
Prophesy[133, 131]	([number of problem size configurations] × [number of processor configurations]) experiments for model derivation
Anglano [11]	(Overhead for source code analysis for identifying computation and communication segments) + ([Number of segments] × [number of problem size configurations] × [number of processor configurations]) experiments for model derivation
Lee et. al.[91]	(300-1000) experiments for model derivation
PACE[106, 6]	(Overhead for source code analysis to identify subtasks) + (Overhead for mapping their performance modeling language outputs to the subtasks for compilation)
Ipek et. al.[83]	(250-500) experiments for model derivation
Our strategy	(20 1-processor + 20 2-processor + 10 4-processor + 10 8-processor) experiments for model derivation + (overhead for deriving sorted list of functions [˜5 minutes])

application developer to register information about the locations of the executables to a centralized application database. Models requiring source codes [150, 106, 6, 2, 22, 152, 11, 120] incur additional registration overhead for obtaining the source files and other libraries for the application. These models also incur the overhead of extensive analysis and instrumentation of the source code. The existing models for dedicated systems[150, 133, 15, 106, 6, 2, 91, 22] incur synchronization overheads among the processors to ensure dedicatedness during experiments for deriving the model functions. Our modeling strategy incurs the following additional overheads:

1. During the experiments, our strategy incurs the overhead of monitoring the cpu and bandwidth loads. However, many of the grid systems consist of infrastructures for monitoring information about resources[147].

2. Our models also incur a small overhead in porting the functions derived on one cluster to other clusters for cross-platform modeling. However, while the existing strategies perform extensive experiments on all clusters to derive model functions for each cluster, our strategy, due to cross-platform modeling, requires extensive experiments on only one cluster.

3. After an application is executed on a cluster, our strategy incurs the overhead of adding parameters corresponding to the application run to the cluster database and reevaluation of the model functions. This step is needed for model adaptivity to changing load dynamics. Moreover, these evaluations are conducted simultaneously with the predictions, i.e. a scheduler can make use of a previous model for immediate predictions while the model is refined for use in later predictions.

3.9.4 Cross-Platform Performance Predictions

For validating the cross-platform performance prediction techniques described in Section 3.5, we used 3 platforms: the 8-processor Intel cluster, 16-processor Woodcrest cluster and 32-processor IBM cluster. For a given pair of reference and target clusters and for a given application, we use the top model function for the application that was obtained after conducting *exp-refer* experiments, corresponding to different problem sizes and processors, on the reference cluster. The computation and communication functions of the model are then scaled by the cor-

responding scaling factors, as explained in Section 3.5, and the resulting model is then used to predict execution times for the application on the target cluster. The percentage prediction errors for the model on the target cluster were obtained by conducting *exp-target* experiments with the application, corresponding to different problem sizes and processors, on the target cluster and obtaining the actual and predicted execution times for the experiments.

Since the IBM cluster is a space-shared batch system, the processors are not subject to external load and are dedicated for our experiments. Hence, we used a value of 1 for minAvgAvailCPU, i.e. unloaded processor, for our experiments on the IBM cluster. However, the network links can still be subjected to external load since the other user applications may cause network traffic on the shared switches and links used by our experiments. We measured the bandwidths on a link of the cluster periodically for 4 days and used the average of the bandwidths for minAvgAvailBW.

Table 3.9 summarizes the prediction results due to cross-platform modeling for different reference and target cluster combinations. About 60-100% of predictions were obtained with less than 30% prediction error. The average percentage prediction error is less than 30% in all cases. We also find that model functions and training data obtained on a non-dedicated reference platform under certain loading conditions can be used for good predictions on both non-dedicated environments under different loading conditions and dedicated batch systems.

3.10 Experiments and Results for Modeling Multi-Phase Parallel Applications

We used five large scale multi-phase parallel applications for evaluating our rescheduling strategies.

1. Molecular dynamics simulation (MD) of Lennard-Jones system systolic algorithm. N particles are divided evenly among the P processes running on the parallel machine. The calculation of forces is divided into P stages. The traveling particles are shifted to the right neighbor processor in a ring topology.

Table 3.9: Cross-Platform Predictions on the 8-processor Intel, 16-processor AMD and 32-processor IBM Clusters

Prob.	Reference Cluster	Loading on reference cluster	exp-refer	Target Cluster	Loading on target cluster	exp-target	Avg. PPE
Eigen	Intel	random	309	IBM	dedicated	50	18.53
Eigen	AMD	grads	203	IBM	dedicated	50	25.9
Eigen	Intel	random	309	AMD	grads	150	26.67
MD	Intel	grads	212	IBM	dedicated	50	23.04
MD	AMD	grads	206	IBM	dedicated	50	22.08
MD	Intel	grads	212	AMD	random	100	28.23
MD	AMD	grads	206	Intel	grads	150	18.68
SSOR	Intel	grads	235	AMD	grads	150	21.20
SSOR	AMD	grads	240	Intel	grads	150	22.55
IS	Intel	random	300	IBM	dedicated	100	25.70
IS	Intel	random	300	AMD	grads	150	26.06
IS	AMD	grads	226	Intel	grads	123	27.08

2. ChaNGa (Charm N-body GrAvity solver)[33], an application for performing collisionless N-body simulations. It performs cosmological simulations with periodic boundary conditions or simulations of isolated stellar systems. Time stepping is done with a leapfrog integrator with individual time-steps for each particle.

3. Athena[14], a grid-based code for astrophysical gas dynamics with nested and adaptive mesh capabilities. Parallelization is achieved using domain decomposition with MPI calls to swap data in ghost cells at grid boundaries.

4. LAMMPS[90] (Large-scale Atomic/Molecular Massively Parallel Simulator), a classical molecular dynamics simulation code for studying crack propagation in 2-D solid materials.

5. MIT Photonic-bands (MPB)[101], a program for computing the band structures (dispersion relations) and electromagnetic modes of periodic dielectric structures. This program computes definite-frequency Eigen states (harmonic modes) of Maxwell's equations in periodic dielectric structures for arbitrary wave vectors, using fully-vectorial and three-

Table 3.10: Prediction Accuracy with Cumulative Models

Application	PPE (Single Model)		PPE (Cumulative Model)	
	Avg.	Std. Dev.	Avg.	Std. Dev.
MD	46.36	35.05	24.62	16.41
Athena	22.18	11.97	18.26	8.84
ChaNGA	86.87	73.37	25.2	20.41
MPB	28.47	16.19	18.37	12.30
LAMMPS	35.34	27.99	21.7	13.21

dimensional methods.

By using active profiling[122, 121] and manual analysis of source codes, we determined the total number of application execution phases as 30 for MD, 50 for ChaNGa, 100 for Athena, 26 for LAMPPS and 32 for MPB. The performance model equations for the application phases were obtained for a 48-core AMD Opteron cluster consisting of 12 2-way dual-core AMD Opteron 2218 based 2.64 GHz Sun Fire servers with CentOS 4.3 operating system, 4 GB RAM, 250 GB Hard Drive and connected by Gigabit Ethernet.

Prediction Accuracy due to Using Cumulative Performance Models

We show the accuracy of multiple performance models for the different phases of a single application in predicting the execution time of the application. We executed different applications with different problem sizes and number of processors on non-dedicated AMD cluster, and observed the total execution times and the execution times of the individual phases. For each application, we developed *single performance model* for the complete application executions using the total executed times and used these performance models for predicting execution times for different application and processor configurations. We also developed performance models for the individual phase executions using the phase execution times. We then formed *cumulative performance model* for the complete application using the performance models for the individual phases of the applications and used these cumulative models to predict application execution times. Table 3.10 compares the percentage prediction errors (PPE) when using single and cumulative performance models for predicting execution times of the applications.

We find that developing and using single performance model function gives highly inac-

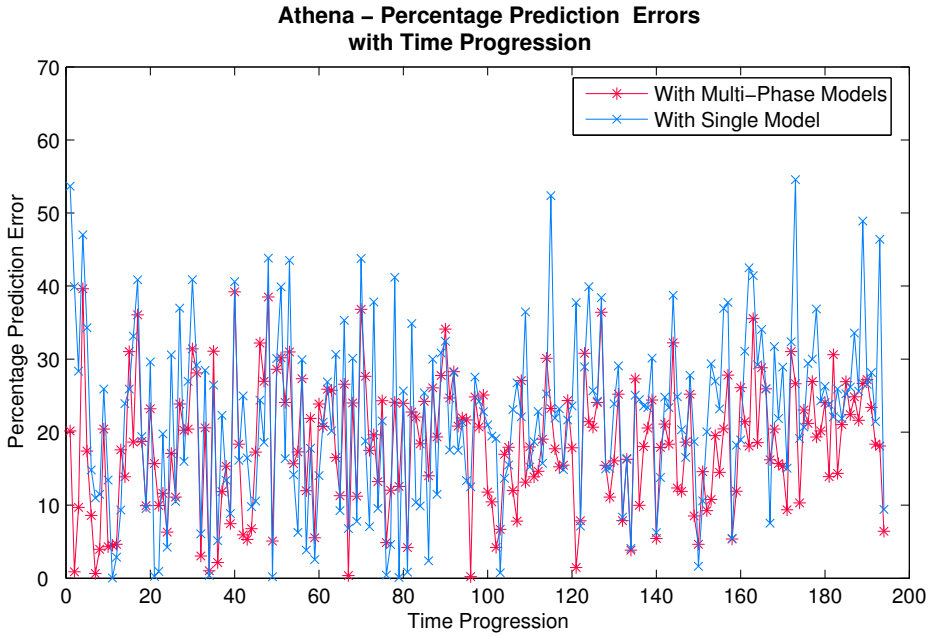


Figure 3.12: Percentage Prediction Errors with Single and Cumulative Models for Athena

curate predictions for ChaNGa, LAMMPS and MD applications. These applications exhibit highly non-uniform behavior during executions and the computation and communication characteristics widely vary between the phases. In comparison, Athena and MPB have fairly uniform computation and communication characteristics throughout application executions. Hence using single performance models for Athena and MPB gave reasonably accurate predictions. In all cases, using cumulative performance models using the models of individual phases gave less than 25% prediction errors, which is highly reasonable prediction accuracy for non-dedicated systems.

Figures 3.12 and 3.13 shows the percentage prediction errors with single and cumulative model for Athena and ChaNGa applications for different application and processor configurations. The x-axis represents the different configurations. Similar results were obtained for MD, MPB and LAMMPS applications which are shown in Appendix 8.1.

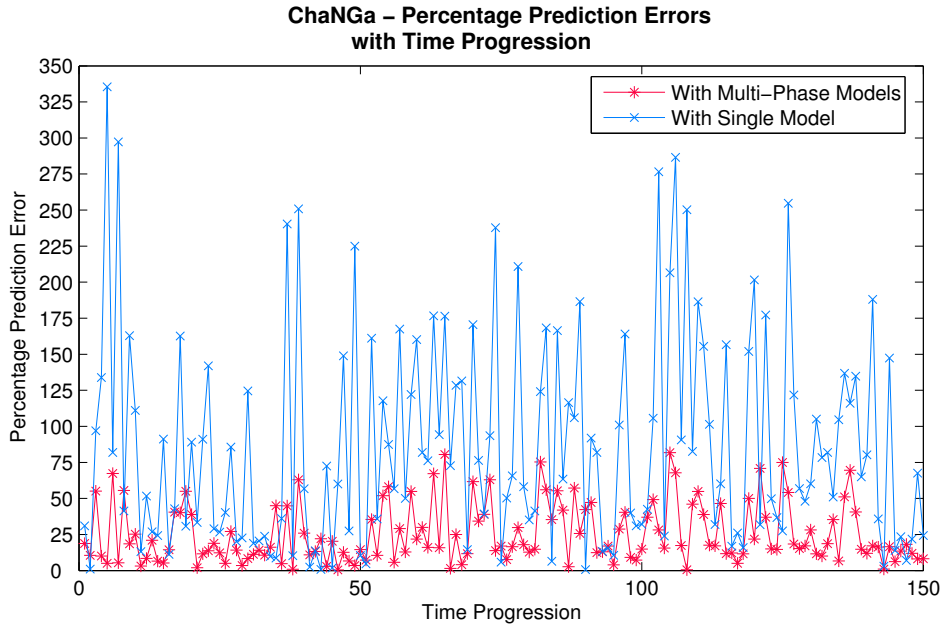


Figure 3.13: Percentage Prediction Errors with Single and Cumulative Models for ChaNGa

3.11 Summary

In this chapter we had devised performance modeling techniques for predicting execution times of tightly-coupled parallel applications for the purpose of scheduling the applications on the grid resources. Our performance modeling strategy consist of multiple phases for modeling different aspects of application characteristics and resource dynamics. Our strategy is also adaptive to grid dynamics since we use different model functions at different times, for predicting execution times, based on the changing resource loads. We have also developed cross-platform modeling techniques for porting the results of performance modeling on one platform or cluster to other clusters in the grid. Our modeling overheads are smaller and more deterministic when compared to other models. Our performance modeling strategies gave less than 30% average percentage prediction errors in all cases, which is reasonable for non-dedicated systems.Our cross-platform techniques were able to give less than 30% average percentage predictions when porting the models to either dedicated or non-dedicated platforms. We also found that scheduling based on the predictions by our strategies will result in perfect scheduling in many cases and can result in maximum loss in efficiency of the scheduler by only 11%.

We also find that developing and using single performance model function gives highly

inaccurate predictions for the applications which exhibit highly non-uniform behavior during executions and the computation and communication characteristics widely vary between the phases. By using cumulative performance models for multi-phase parallel application gave less than 25% prediction errors, which is highly reasonable prediction accuracy for non-dedicated systems.

Chapter 4

Scheduling Tightly-Coupled Parallel Applications on Clusters and Grids

In the previous chapter we discussed methodology for modeling and predicting execution times of tightly-coupled parallel applications on dedicated and non-dedicated clusters. In this chapter we present scheduling strategies that can efficiently make use of application specific performance models.

4.1 Introduction

Scheduling is difficult and challenging in grid computing due to the very dynamic and unpredictable nature of Grid resources. While various strategies have been developed for scheduling parallel applications with independent tasks, very little work exists for scheduling tightly-coupled parallel applications on clusters and grid environments.

As discussed in chapter 2, most of the existing efforts in scheduling parallel applications deal with independent tasks[16, 143, 153]. Scheduling strategies that have been developed for allocating workflow application components on different clusters of a grid[25, 99] cannot be used for tightly-coupled applications due to higher frequency of inter-task communications in tightly-coupled applications.

In this work, we have devised, evaluated and compared algorithms for scheduling tightly-

coupled parallel applications on a non-dedicated cluster consisting of homogeneous machines. Our algorithms are also applicable for grid frameworks consisting of multiple clusters where machines within a cluster are homogeneous while machines from different clusters can be heterogeneous. When a tightly-coupled parallel application is submitted to such a grid, our algorithms will be invoked simultaneously on multiple clusters and the schedule of machines from one of the clusters that gives the overall minimum execution time will be chosen for application execution. Our algorithms use performance models that predict the execution times of parallel applications, for evaluations of candidate schedules.

We also propose a novel efficient algorithm called Box Elimination (BE) that searches a space of performance model parameters to determine efficient schedules. The primary contributions of this work are development of a novel scheduling algorithm and evaluations and comparisons of algorithms for scheduling tightly-coupled parallel applications on non-dedicated clusters.

In Section 4.2, we describe the various algorithms we use in this study. We also describe in detail our Box Elimination algorithm. In Section 4.3, we present our real and simulation experimental setup and show comparison results. Section4.4 summarizes the chapter.

4.2 Algorithms

In chapter 3, we presented performance modeling strategies for predicting execution times of tightly-coupled parallel applications on non-dedicated homogeneous resources. We calculated the time taken for execution of a parallel application as:

$$T(N, P, minAvgAvailCPU, minAvgAvailBW) = \frac{f_{comp}(N)}{f_{cpu}(minAvgAvailCPU) \cdot f_{Pcomp}(P)} + \frac{f_{comm}(N)}{f_{bw}(minAvgAvailBW) \cdot f_{Pcomm}(P)} \tag{4.1}$$

where

- N: problem size or data size; P: number of processors;

- $minAvgAvailCPU$, $minAvgAvailBW$: represent the transient CPU and network characteristics respectively.
- f_{comp}, f_{comm}: indicate the computational and communication complexity respectively, of the application in terms of problem size;
- f_{cpu}: function to indicate the effect of processor loads on computations;
- f_{Pcomp}: used along with computational complexity to indicate the computational speedup or the amount of parallelism in computations;
- f_{bw}: function to indicate the effect of network loads on communications;
- f_{Pcomm}: used along with communication complexity to indicate the communication speedup or the amount of parallelism in communications;

  In this section, we describe six algorithms for finding schedules of machines in a cluster for execution of a parallel application with a given problem size. The first five algorithms are based on popular optimization techniques while the last algorithm, the Box Elimination algorithm, is our contribution. All of the algorithms use Equation 4.1 for evaluation of candidate schedules in terms of the predicted execution times of the schedules. Timers are set in all the algorithms by means of the $alarm()$ function to make the algorithms time-tunable. This lets the user of our scheduling codes to specify the duration of scheduling. The algorithms, at the point of expiry of the timer, exit from their current operations and return the best schedules at that point. In all our algorithms, max_procs denotes the total number of available machines in the cluster, out of which a set of machines is returned as the best schedule. In a multi-cluster grid setup, the algorithms are invoked for each of the clusters and the cluster that contains the overall best schedule with minimum execution time is chosen for executing the application.

4.2.1 Simulated Annealing

Simulated Annealing is a technique to find a good solution to an optimization problem by trying random variations of the current solution. A worse variation is accepted as the new

solution with a probability that decreases as the computation proceeds. The slower the cooling schedule, or rate of decrease, the more likely the algorithm is to find an optimal or near-optimal solution.

Yarkhan and Dongarra[154] developed a simulated annealing based algorithm for choosing a set of machines for execution of a tightly-coupled parallel application. This algorithm initially generates a random schedule and perturbs the schedule at various steps. Poorer schedules with higher execution times are accepted probabilistically based on a temperature value. We show the algorithm in Figure 4.1. The algorithm by Yarkhan et al. used the performance model equation only as an objective function for search space exploration. We have modified the algorithm to use our performance model of Equation 4.1 both as an objective function for comparison of schedules and for generating the initial schedule of machines (lines 2-7).

4.2.2 Genetic Algorithm

Genetic Algorithm is an evolutionary algorithm which generates each individual from some encoded form, known as a "chromosome" or "genome". Chromosomes are combined or mutated to breed new individuals. "Crossover" or recombination of chromosomes is often also used in genetic algorithms. Here, an offspring's chromosome is created by joining segments chosen alternatively from each of two parents' chromosomes which are of fixed length. Genetic algorithms are useful for multidimensional optimization problems in which the chromosome can encode the values for the different variables being optimized.

Genetic algorithm has been used in a number of scheduling problems [87, 64, 79]. In this algorithm, a population of chromosomes is initially generated and the chromosomes undergo cross-over and mutations across various generations. The chromosomes with high fitness values are retained over successive generations. In our problem, a chromosome corresponds to a schedule or a set of machines. We calculate the fitness of a chromosome as the reciprocal of the execution time of the corresponding schedule. The algorithm is

```
1  Algorithm:Simulated Annealing (SA)
2  for each machine i do
3      Evaluate execTime[i] of machine i using Eq. 4.1 ;
       /* For bandwidth in Eq. 4.1, use average bandwidth of
          links to i                                          */
4  end
5  sortedList = machines in ascending order of execTime ;
6  for procs = max_procs to 2 do
7      curSched = top procs machines in sortedList ;
8      curExecTime = execution time of curSched using Eq. 4.1;
9      for temperature=100 to 1 in steps of 0.8 do
10         num_accepts = 0 ;
11         for steps = 1 to 80 do
               /* Perturb schedule randomly                  */
12             newSched = Randomly replace a machine in curSched ;
13             newExecTime = execution time of of newSched using Eq. 4.1 ;
14             if newExecTime < curExecTime then
15                 prevExecTime = curExecTime ; curSched = newSched ;
                   curExecTime = newExecTime ;
16                 num_accepts = num_accepts + 1 ;
17             end
18             else
19                 r = random number ;
20                 if r < e^((prevExecTime - curExecTime)/temperature) then
21                     prevExecTime = curExecTime; curSched = newSched;
                       curExecTime = newExecTime;
22                     num_accepts = num_accepts + 1;
23                 end
24             end
25             if curExecTime > prevExecTime or num_accepts > 20 then
26                 break ;
27             end
28         end
29         if curExecTime > prevExecTime then
30             break;
31         end
32     end
33 end
```

Figure 4.1: Simulated Annealing (SA)

shown in Figure 4.2. As can be seen, we not only use the performance model equation for evaluating the fitness of the chromosomes, but also during mutations to generate better mutated chromosomes (lines 11-24).

4.2.3 Ant Colony Optimization

Ant Colony Optimization (ACO) [48] has been used in a number of optimization problems [124, 23, 29]. The basis of this algorithm is the behavior of real ants in search of food sources. Ants deposit a chemical called pheromone on their trails to food sources. The ants follow the trail with high value of pheromone deposited on the trail by other ants. This leads to repeated reinforcement of trails that lead to better food sources. This concept is used in optimization problems where artificial ants explore search space and better solutions are visited with higher probabilities due to a pheromone variable. The optimization problems also use pheromone evaporation to discard initial random solutions. The primary advantage of algorithms based on ACO is coordination of positive feedback on better solutions through indirect communications.

The work by Ahmed Al-Ani[5] used ACO for feature subset selection problem. The problem was to select a set of features from a large set such that the selected features can be used for efficient classification in problems related to pattern recognition. We have adopted the algorithm for our machine selection problem. In the algorithm shown in Figure 4.3, a pheromone variable, T_i, associated with each machine, is updated in every iteration based on the execution time of the schedules containing the machine (lines 17-24). The set of machines used for the next step of the algorithm is based on the pheromone value calculated in the current step. An evaporation factor, ρ, for pheromone, is also used in the algorithm. Performance model equation is used in the algorithm (lines 26-38) for generation of better schedules in the next iteration.

```
1  Algorithm:Genetic Algorithm (GA)
2  for procs = max_procs to 2 do
3      Randomly generate initial population of 60 chromosomes of size procs;
4      Evaluate the fitness of chromosomes using Eq. 4.1 ;
5      for generations = 1 to 50 do
           /* CrossOver                                                    */
6          Divide chromosomes into pairs ;
7          for each pair of chromosome do
8              Pick a random chromosome position ;
9              Cross-over chromosome segments from that position ;
10         end
           /* After cross-over, there are 120 chromosomes in the
              population                                                   */
           /* Mutation                                                     */
11         for each of 60 newly generated chromosomes do
12             for mutation steps = 1 to 20% of chromosome length do
13                 Randomly choose a machine X in a chromosome to mutate ;
14                 topfitness = 0;
15                 for each machine Y not in chromosome do
16                     Replace machine X with Y to form mutated chromosome ;
17                     Evaluate fitness of mutated chromosome using Eq. 4.1 ;
18                     if fitness > topfitness then
19                         topfitness = fitness ; topmc = Y ;
20                     end
21                 end
22                 Replace machine X with topmc in the chromosome ;
23             end
24         end
25         Evaluate the fitness of 120 chromosomes using Eq. 4.1 ;
           /* Selection                                                    */
26         Select the best 60 ranking individuals for next generation;
27     end
28 end
```

Figure 4.2: Genetic Algorithm (GA)

```
1  Algorithm:Ant Colony Optimization (ACO)
2  nTopAnts = 30 ; ρ = 0.9 ; num_ants = 60 ;
3  for procs = max_procs to 2 do
4      for i = 1 to procs do
5          T_i = 1; △T_i = 0 ;
6      end
7      remaining_mc_count = 0.25 × procs ;
8      for iter = 1 to 50 do
9          if iter = 1
10           for i = 1 to num_ants
11            Assign random procs machines to Ants[i] ;
12         for i = 1 to num_ants
13           eTime[i] = execution time for schedule  assigned to Ant[i] using Eq. 4.1 ;
14         sortedAnts = ants arranged in ascending order of execution times ;
15         topAnts = top nTopAnts in sortedAnts ;
16         minEtime = execution time of topAnts[0] ;
17         for i = 1 to numtopAnts do
18             curEtime = exec. time of topAnts[i] ;
19             for γ machine j ∈ topAnts[i] do
20                 a = minETime; b = curETime;
21                 △T_j = (a−b) / Max_{k∈TopAnts}(a−eTime[k]);
22                 T_j = (ρ × T_j) + △T_j ;
23             end
24         end
25         pool = union of machines in topAnts ;
26         for i = 1 to num_ants do
27             Generate random (0.75 × procs) unique m/cs. from pool for Ants[i] using
               Roulette wheel using pheromone value of m/cs. ;
28         end
29         for i = 1 to num_ants do
30             for j = 1 to remaining_mc_count do
31                 totalWeight = 0;
32                 for machine y ∉ Ants[i] do
33                     eTime = Exec. time of Ants[i] if y is added; weight_y = T_y / eTime;
                       totalWeight+ = weight_i ;
34                 end
35                 k = m/c with max weight_y / toalWeight ;
36                 Add machine k to Ants[i] ;
37             end
38         end
39     end
40 end
```

Figure 4.3: Ant Colony Optimization (ACO)

4.2.4 Branch and Bound

Branch and Bound method has been used in a number of discrete combinatorial optimization problems [104, 50, 127]. The algorithm begins by considering the original problem for the complete search space. This is represented as a root node of the branch-and-bound tree. The search space is then divided (*branching*) into different sub problems. Each of these problems correspond to a child node of the root node. The method is recursively applied to each of the child nodes. For a minimization problem, an upper bound is associated with each node based on the current best solution. If the solution associated with a node exceeds the upper bound of the node, the node along with the descendants of the node are pruned from the tree (*bounding*).

The branch-and-bound algorithm for our problem is shown in Figure 4.4. In the algorithm, each node is associated with a set of machines. A node corresponds to a search space consisting of all schedules that can be formed from the machines. The root node of the branch-and-bound tree is associated with all available machines. The child nodes of a node in the tree are formed by removing a machine in the set associated with the parent node (line 10). The nodes are evaluated using the performance model equation for the application. The execution time corresponding to a node is calculated as the time corresponding to a schedule of all machines in the set associated with the node. In order to restrict the number of evaluations, we arrange the child nodes of a parent node in ascending order of minimum execution times and consider only the first half of the child nodes with minimum execution times and prune the other child nodes (lines 11-14). For the remaining nodes in a level of the tree, an upper bound is calculated as the minimum of execution times of schedules corresponding to nodes in the previous level of the tree (line 22). Thus, nodes in the current level whose execution times are greater than the minimum execution time of the nodes in the previous level are pruned from the tree. In our algorithm, we use the performance model equation for evaluation of the schedules and pruning of the tree.

```
1  Algorithm:Branch and Bound(B&B)
2  Root = Schedule with all machines ; Tree = Root ;
3  curExecTime = execution time for Root using Eq. 4.1 ;
4  bound = curExecTime ; curLevel = 1 ;
5  numMachinesInLevel = max_procs ;
6  nodesinCurLevel = Root ;
7  while numMachinesInLevel ≠ 0 do
8      nodesInNextLevel = φ ;
9      for curNode ∈ nodesInCurLevel do
10         newNodes = list of numMachinesInLevel nodes formed by removing one of
           the machines in curNode ;
11         Evaluate execution time of schedules in newNodes using Eq. 4.1 ;
12         sortedList = newNodes arranged in ascending order of execution times ;
13         halfList = the first half of sortedList ;
14         childNodes = subset of halfList with execution time < bound ;
15         Expand Tree by adding childNodes with currentNode as parent ;
16         nodesInNextLevel = nodesInNextLevel + childNodes ;
17     end
18     if nodesInNextLevel = φ then
19         break;
20     end
21     nodesInCurLevel = nodesInNextLevel ;
22     bound = Minimum execution time of nodesInCurLevel ;
23     numMachinesInLevel = numMachinesInLevel − 1 ;
24 end
```

Figure 4.4: Branch and Bound (B&B)

4.2.5 Dynamic Programming

Dynamic Programming finds a solution to a problem in terms of solutions to a set of subproblems. It solve an optimization problem by caching subproblem solutions (memorization) rather than recomputing them. The overall solution is expressed as a recursive equation containing solutions to the smaller subproblems. For our problem of scheduling tightly-coupled parallel application, we apply dynamic programming to find a best schedule of n machines, $sched_n$, in terms of the best schedule of $n-1$ machines, $sched_{n-1}$. The recursive formulation we use for dynamic programming is shown in Equation 4.2. t denotes the execution time for a schedule determined using the performance model equation and $minMc$ is the machine corresponding to minimum of execution times of all schedules constructed out of machines in $sched_{n-1}$ and another machine. The algorithm is shown in Figure 4.5. The schedule is constructed incrementally by starting with a schedule of one machine with the highest available CPU and adding machines, that give the best predicted execution time, to the schedule. At each stage of the algorithm, we also consider the second best schedule of machines for addition of machines in the next step (line 15). This is done to reduce the effects of local minima.

$$\begin{aligned} sched_n =& \{sched_{n-1}, minMc\} s.t. \\ t(\{sched_{n-1}, minMc\}) =& Min_{i \notin sched_{n-1}} t(\{sched_{n-1}, i\}) \end{aligned} \tag{4.2}$$

4.2.6 Box Elimination

The Box Elimination algorithm is based on the observation that the predicted execution time of an application with a given problem size on a set of machines is expressed in Equation 4.1 in terms of minimum available CPUs of the machines in the set, minimum available bandwidth of the links between the machines and number of machines in the set. Unlike other algorithms, which search through a space of schedules of machines, the Box Elimination algorithm searches through a space of hypothetical points, where each point corresponds to a (minimum available CPU, minimum available bandwidth, number of processors) tuple. Each hypothetical point is then mapped to a schedule of machines

```
1  Algorithm:Dynamic Programming (DP)
2  curSchedule = Machine with maximum CPU availability ;
3  curScheduleList = curSchedule ;
4  curMachineCount = 1 ;
5  while curMachineCount < max_procs do
6      newMachineCount = curMachineCount + 1 ;
7      newScheduleList = φ ;
8      for curSchedule ∈ curScheduleList do
9          Form schedules with newMachineCount from machines in curSchedule and a
           single machine not in curSchedule ;
10         Evaluate execution times of schedules using Eq. 4.1 ;
11         Pick schedules with the minimum and second minimum execution times to form
           bestNewSchedule and nextBestNewSchedule ;
12         newSchedule = bestNewSchedule, nextBestNewSchedule ;
13         newScheduleList = newScheduleList + newSchedule ;
14     end
15     curScheduleList = top two schedules from newScheduleList with minimum
       execution times ;
16     curMachineCount = newMachineCount + 1 ;
17 end
```

Figure 4.5: Dynamic Programming (DP)

whose minimum available CPU and bandwidth values are greater than and closest to the corresponding values in the hypothetical point. The strength of the algorithm is in its ability to eliminate search space regions of hypothetical points based on a hypothetical point that was searched. The elimination of the search space points is based on the characteristic of the performance model function used in Equation 4.1 and is not possible in other algorithms where the search space points are schedules or lists of machines.The results for the other algorithms presented in this work should be considered as evaluations of implementations of the algorithms that use the commonly used search space of schedules[154, 27] and not as evaluations of the inherent characteristics of the algorithms themselves.

The algorithm works on a 3-D box of grid points with each grid point corresponding to a ($minAvgAvailCPU$, $minAvgAvailBW$, number of processors) tuple. The box is bounded on the x-axis by (cpu_{min}, cpu_{max}), the minimum and maximum respectively, of $AvgAvailCPU$ of *all* machines in the cluster, on the y-axis by (bw_{min}, bw_{max}), the minimum and maximum respectively, of $AvgAvailBW$ of all the links, and on z-axis by ($P_{min} = 1$, $P_{max} = max\_procs$). The x-axis values are incremented in steps of 0.1. For y-axis values, we use different increment steps based on the maximum bandwidth of the links. We use increment steps of 10 Mbps for maximum bandwidths of 100 Mbps and 1 Gbps, 50 Mbps for maximum bandwidth of 5 Gbps and 100 Mbps for maximum bandwidth of 10 Gbps.

The algorithm, shown in Figure 4.6, begins by finding the center grid point, (cpu_c, bw_c, P_c) of the 3-D box. This point is then mapped to a schedule of machines whose $minAvgAvailCPU$ and $minAvgAvailBW$ values are greater than and closest to cpu_c and bw_c respectively (line 4). The schedule is then mapped back to a corresponding hypothetical point (cpu_g, bw_g, P_g) in the 3-D box (line 6). Eight 3-D sub-boxes, SB_1-SB_8, are formed in the 3-D box with reference to (cpu_g, bw_g, P_g) as shown in Figure 4.7. The algorithm then repeatedly generates a random point in the uncovered region of the 3-D box, finds closest schedule of machines and maps the schedule to a hypothetical point in the box (lines 15-18). For a hypothetical grid point, (cpu_h, bw_h, P_h), two sub-boxes

corresponding to two regions of search space are eliminated:

1. sub-box bounded by (cpu_{min}, bw_{min}, P_{min}) and (cpu_h, bw_h and P_h). This elimination is based on the observation that there exists no point ($cpu_p \leq cpu_h$, $bw_p \leq bw_h$, $P_p \leq P_h$) for which the predicted execution time by Equation 4.1 is less than the predicted execution time for (cpu_h, bw_h and P_h). Thus the sub-box corresponds to a search space of poorer solutions.
2. sub-box bounded by (cpu_h, bw_h, P_h) and (cpu_{max}, bw_{max} and P_{max}). The elimination of this sub-box is made possible by our mapping function, $FindScheduleWithConstraints$, shown in in Figure 4.8, that maps a hypothetical grid point, (cpu_{grid}, bw_{grid}, P_{grid}) to a schedule of machines defined by ($cpu_{real} \geq cpu_h$, $bw_{real} \geq bw_h$, P_{real}). This schedule is then mapped to the closest grid point, (cpu_h, bw_h, P_h). The mapping function ensures that there is no schedule of machines in the sub-box with values of $minAvgAvailCPU$ and $minAvgAvailBW$ and number of processors higher than (cpu_h, bw_h, P_h).

The objective of the mapping function is to determine a maximal set of machines whose $minAvgAvailCPU$ and $minAvgAvailBW$ values, denoted by cpu_{real} and bw_{real} respectively, are greater than cpu_{grid} and bw_{grid} respectively, such that the sub-box containing higher values of $minAvgAvailCPU$, $minAvgAvailBW$ and number of processors can be eliminated. The mapping procedure has two phases. In the first phase (lines 5-17), a set of machines whose $AvgAvailCPU$ values are $\geq cpu_{grid}$ are formed. However, the $minAvgAvailBW$ value of this set can be $< bw_{grid}$. Hence machines are successively removed from the set in a number of steps until the minimum of the $AvgAvailBW$ values of the links connecting the machines in the set is $\geq bw_{grid}$. In each step of removal, the machine with the greatest number of links whose $AvgAvailBW$ values are less than bw_{grid} is removed. The resulting set formed in the first phase consists of machines whose $minAvgAvailCPU$ and $minAvgAvailBW$ values are greater than cpu_{grid} and bw_{grid} respectively. However, this may not be the maximal set due to the heuristic followed in the removal of machines. Hence, in the second phase of mapping procedure (lines 18-23), we incrementally add machines that were removed in the first phase. In each step of

Algorithm: Box Elimination (BE)
$UnExploredGridProints = \{$all grid points$\}$;
Find the center grid point of the box, cpu_c, bw_c, P_c ;
($currentSchedule$, cpu_r, bw_r, P_r) = `FindScheduleWithConstraints` (cpu_c, bw_c, P_c) ;
$currentExecTime$ = Evaluate $currentSchedule$ using Equation 4.1 ;
(cpu_g, bw_g, P_g) = FindClosestGridPoint(cpu_r, bw_r, P_r) ;
$UnExploredGridProints = UnExploredGridProints-$ (cpu_g, bw_g, P_g) ;
FormEightSubBoxes(cpu_g, bw_g, P_g) ;
EliminateTwoSubBoxes(cpu_g, bw_g, P_g) ;
$UnExploredGridProints = UnExploredGridProints-$ eliminated sub boxes ;
Initialize counters for six sub boxes to 1 ;
while $UnExploredGridPoints \neq \phi$ **do**
    Form a Roulette wheel consisting of six sectors for six sub boxes based on counter values;
    Probabilistically choose a sub box, b_i, based on the Roulette wheel ;
    Choose a random point, cpu_{new}, bw_{new}, P_{new} in the sub box ;
    ($newSchedule$, cpu_{newr}, bw_{newr}, P_{newr}) = `FindScheduleWithConstraints` (cpu_{new}, bw_{new}, P_{new}) ;
    $newExecTime$ = Evaluate $newSchedule$ using Equation 4.1 ;
    (cpu_{newg}, bw_{newg}, P_{newg}) = FindClosestGridPoint(cpu_{newr}, bw_{newr}, P_{newr}) ;
    $UnExploredGridProints = UnExploredGridProints-$ (cpu_{newg}, bw_{newg}, P_{newg}) ;
    FormEightSubBoxes(cpu_{newg}, bw_{newg}, P_{newg}) ;
    EliminateTwoSubBoxes(cpu_{newg}, bw_{newg}, P_{newg}) ;
    $UnExploredGridProints = UnExploredGridProints-$ eliminated sub boxes ;
    if $newExecTime < currentExecTime$ **then**
        Increment counter value for sub box, b_i ;
        $currentExecTime = newExecTime$;
        $currentSchedule = newSchedule$;
    end
end

Figure 4.6: Box Elimination (BE)

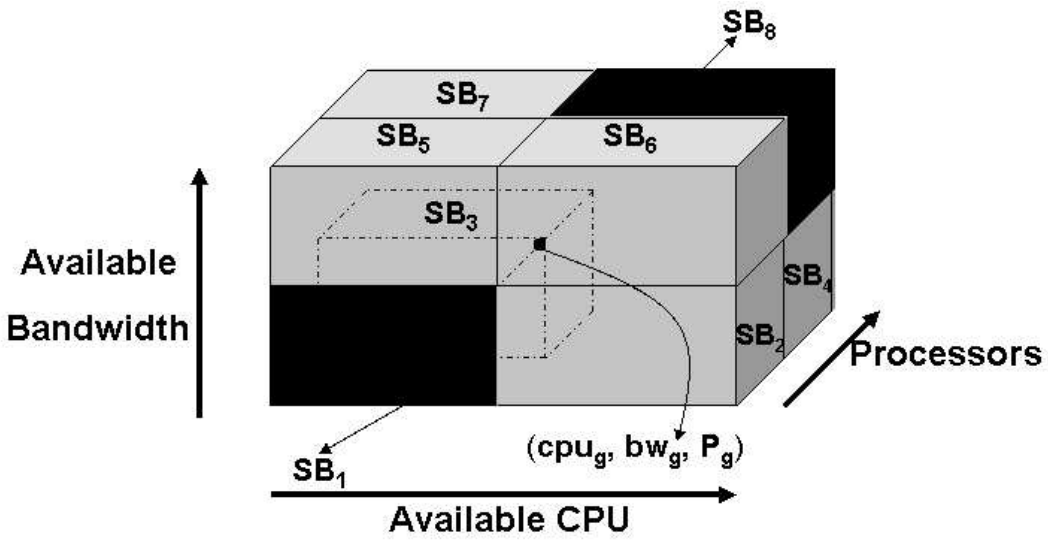


Figure 4.7: 3-D box of (cpu, bandwidth, processors) tuples for Box Elimination

addition, we choose a machine such that the minimum of the $AvgAvailBW$ of the links connecting the machine to the machines already in the set is $\geq bw_{grid}$. The final schedule of machines correspond to (cpu_{real},bw_{real}, P_{real}) and mapped to the closest grid point in the 3-D box, (cpu_h,bw_h, P_h).

It is easy to show that the region bounded by (cpu_h, bw_h, P_h) and (cpu_{max}, bw_{max} and P_{max}) does not have a schedule of machines. This is because the final schedule from the mapping procedure corresponds to maximal set of machines for which the $minAvgAvailCPU$ and $minAvgAvailBW$ values are cpu_{real} and bw_{real} respectively. Also, since these values are lower bounds for a schedule of machines, increasing any one of the lower bounds will only lead to schedules with smaller number of machines. Hence, the sub-box denoting values larger than (cpu_h, bw_h, P_h) can be eliminated. In Figure 4.7, the two eliminated sub-boxes, SB_1 and SB_8, corresponding to the grid-point (cpu_g, bw_g, P_g) are darkly shaded.

To generate random hypothetical points in the 3-D box, we use a Roulette-wheel-based mechanism where the Roulette-wheel is initially formed of six equal-sized sectors corresponding to the six sub-boxes, SB_2-SB_7, with reference to the grid point (cpu_g, bw_g, P_g) shown in Figure 4.7. When a generated random point in the 3-D box results in a schedule

```
1  Algorithm:FindScheduleWithConstraints()
   input  : cpu_grid, bw_grid, P_grid
   output: schedule, cpu_real, bw_real, P_real
2  initSchedule = list of machines whose available CPU ≥ cpu_grid ;
3  currentSchedule = initSchedule ;
4  min_bw = minimum of bandwidths of links connecting machines in currentSchedule ;
5  while min_bw < bw_grid do
6      for all machines m ∈ currentSchedule do
7          violation_count[m] = 0 ;
8          avg_bw[m] = 0 ;
9      end
10     for each machine m ∈ currentSchedule do
11         bw_violation_count[m] = number of links between m and other machines in
           currentSchedule whose bandwidths are < bw_grid ;
12         avg_bw[m] = average of bandwidths of links connecting m and other machines in
           currentSchedule ;
13     end
14     top_m = machine with maximum bw_violation_count ;
       /* If machines have equal bw_violation_count, machine with
          minimum avg_bw is chosen                                    */
15     currentSchedule = currentSchedule − top_m ;
16     min_bw = minimum of bandwidths of links connecting machines in
       currentSchedule ;
17 end
18 for each machine m ∈ initSchedule and ∉ currentSchedule do
19     min_bw = minimum of bandwidths on links connecting m and machines in
       currentSchedule ;
20     if min_bw > bw_grid then
21         currentSchedule = currentSchedule + m ;
22     end
23 end
24 schedule = currentSchedule ;
25 P_real = number of machines in schedule ;
26 cpu_real = minimum of available CPUs of machines in schedule ;
27 bw_real = minimum of bandwidths of links connecting machines in schedule ;
28 return (schedule, cpu_real, bw_real, P_real) ;
```

Figure 4.8: FindScheduleWithConstraints()

better than any of the previously determined schedules, the sub-box containing the random point is determined, and the corresponding sector in the Roulette wheel is enlarged. Thus the probability of generating random points in the sub-box or regions containing good schedules or solutions increases over time. When the search space in a sub-box is completely explored, its corresponding sector is eliminated in the Roulette-wheel and the remaining sector sizes are adjusted so that the sum of the probabilities of generating random points in the sectors equals 1.

Compared to the other algorithms, the Box Elimination (BE) algorithm performs several steps in processing a schedule due to generation of a random point the 3-D box, mapping the point to a schedule using the iterative mapping procedure, mapping the schedule back to a grid point in the box, elimination of sub-boxes and adjusting the Roulette wheel. However, the number of processed schedules is less than the other algorithms due to the elimination mechanism followed.

4.2.7 Correctness

The Box Elimination algorithm, at each step, eliminates two sub-boxes, SB_1 and SB_8, with reference to a grid point. The grid points in sub-box, SB_1, correspond to lower number of processors, lower available CPUs and bandwidths when compared to the reference grid point. Since the execution time of an application is inversely proportional to the number of processors, available CPUs and bandwidths, as shown in Equation 4.1, the grid points in SB_1 have higher execution times than the reference grid point. Hence SB_1 need not be explored for schedules with small execution times and can be eliminated.

The mapping function, $FindScheduleWithConstraints$ finds a maximal schedule with $minAvgAvailCPU$ and $minAvgAvailBW$ values that are at least equal to the corresponding values for the reference grid point. Schedules with number of processors, available CPUs and bandwidths greater than the maximal schedule do not exist. This is because the $minAvgAvailCPU$ and $minAvgAvailBW$ for the maximal schedule represent lower bounds and increasing one of the lower bounds will lead to schedules with

smaller number of processors than the maximal schedule. Thus, sub-box SB_8, that represent regions where schedules do not exist need not be explored and can be eliminated.

Thus, at each step or iteration, the Box Elimination (BE) algorithm eliminates regions containing schedules with large execution times or lower-quality schedules. The best schedule with the minimal execution time is contained only in the remaining regions. Given sufficient time, the BE algorithm will explore these remaining regions and eventually find the best schedule.

4.2.8 Complexity

At each step or iteration, the Box Elimination algorithm approximately eliminates $(1/4)^{th}$ of the remaining unexplored schedules. Thus the number of steps or iterations in the algorithm $\log_{4/3}(points)$ where $points$ is the number of grid-points in the 3-D box. $points$ is dependent on the minimum and the maximum values of the number of processors, available CPUs and bandwidths and the discretization of the box. In each step, the algorithm performs the steps of generating a grid point using Roulette Wheel mechanism, determining a mapping using $FindScheduleWithConstraints$ function, and elimination of boxes. Since we have six Roulette wheel sectors, the complexity in generating a grid point is $O((((3/4)rem\_points)/6))$, where rem_points is the number of remaining grid-points. For the mapping function, the complexity is $O(proc\_cnt^2)$ where $proc\_cnt$ is the number of processors that satisfies the CPU availability constraint. The complexity for elimination of sub-boxes SB_1 and SB_8 are $O(pr\_limit + cpu\_limit + bw\_limit)$ where the three parameters correspond to range of number of processors, available CPU and available bandwidth, respectively. Thus the total time complexity of the Box Elimination algorithm is approximately $O(\log_{4/3} points \times ((((3/4)rem\_points)/6) + proc\_cnt^2 + (pr\_limit + cpu\_limit + bw\_limit)))$.

4.2.9 Comparison with Other Algorithms

In the other algorithms, the worst-case number of steps or explorations to determine the best schedule vary linearly with the number of schedules. In simulated annealing algorithm, a new schedule varies from a previous schedule by one machine or processor. Thus, there can be $proc!$ number of explorations to determine the best schedule, where $proc$ is the number of machines or processors. Similar to simulated annealing algorithm, genetic algorithm and ant colony algorithm also perform $proc!$ number of explorations to determine the best schedule. Dynamic programming has $proc^2$ explorations, since it builds the schedule with the best single processor and then adds best processor to the current schedule by ranking the processors. Branch and bound algorithm can be logically considered as performing the steps of dynamic programming in the reverse order. It performs extra steps related to determining the bounds and pruning the tree. Thus the complexity of branch and bound is also $O(proc^2)$ and slightly higher than the complexity of dynamic programming.

4.3 Experiments and Results

We compared the various scheduling algorithms using both real experiments on 104 cores of an Intel Xeon cluster and simulation experiments for larger number of cores or processors.

4.3.1 Real Experiments

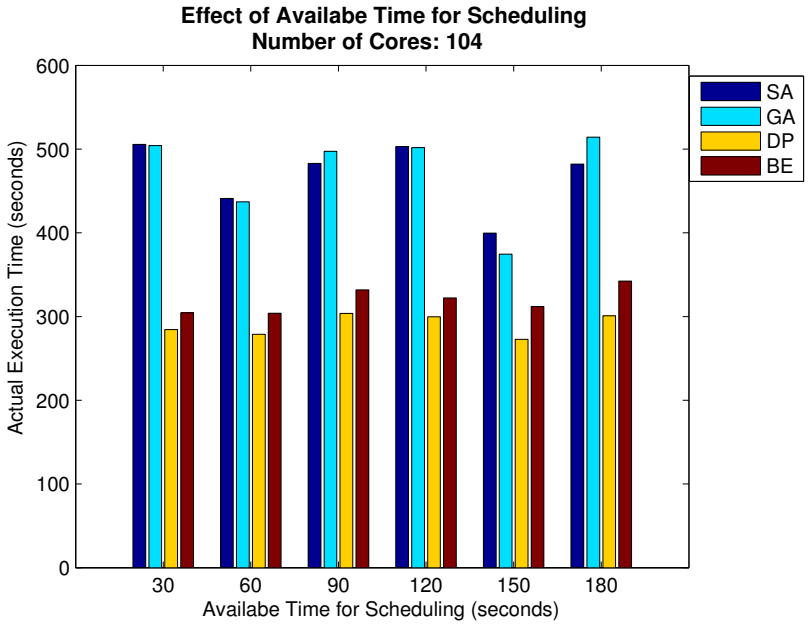
We evaluated our algorithms using the Molecular Dynamics (MD) application with 2400 molecules on a 128-core cluster, consisting of 16 nodes, with each node consisting of two Intel Quad core Xeon E5440 CPUs running at 2.83 GHz. Each node has 16GB RAM, 500 GB total hard disk capacity and runs Debian Etch (version 4.0) with Linux kernel 2.6.24. The nodes are connected by Gigabit Ethernet through a 24-port Gigabit Ethernet switch. We used 104 cores and 13 nodes for our experiments.

For each experiment corresponding to an execution of the MD application, we introduced synthetic CPU and network loads in the system by continuously executing synthetic CPU and network loading programs on the processors in the background and maintained the loads for the duration of the experiment. For loading the CPUs of the system for an experiment, a set of processors was randomly chosen out of the available processors in the system and synthetic loading programs were run on the processors in the set. The amount of loading on each processor was randomly varied such that the available CPU value of the processor is varied between 6.5%-72% of the total CPU. Small available CPU percentages imply large loading of the processor. For network loading, we used a loading program to introduce synthetic network loads on the links of the system and to reduce the available bandwidths of the links. For an experiment, a random number (between 1-8) of source-destination pairs was chosen out of all possible source-destination pairs in the system. Random amounts of network loads were introduced on the links between the source-destination pairs by running the synthetic network program so as to vary the available bandwidths of the links between the hosts from 2% to 80% of the total bandwidth capacities of the links.

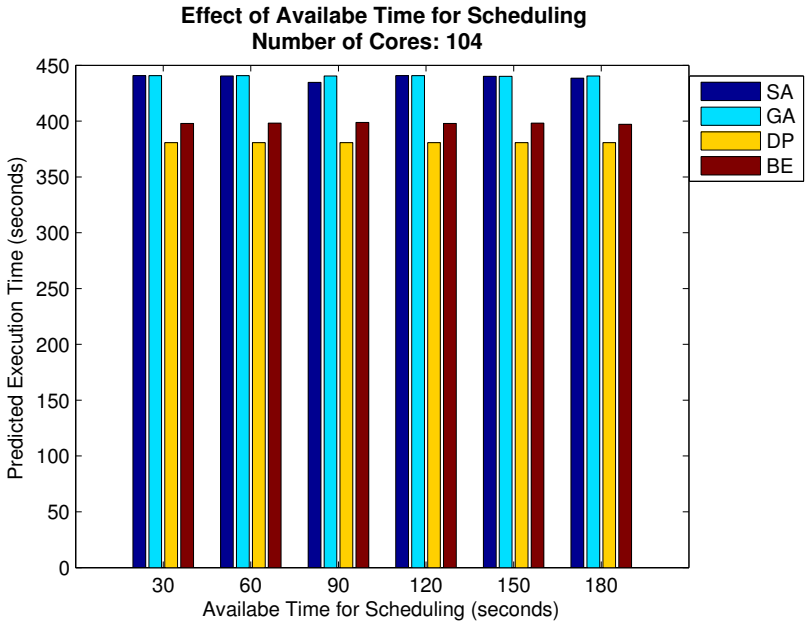
After loading the CPUs and network links for an experiment, we observed the available CPUs and bandwidths of the processors and links, respectively, using NWS[147]. Each of the scheduling algorithms was then invoked with these available CPUs and bandwidths along with the problem size and the resulting schedules of machines determined by the algorithms were obtained. The schedules determined by the algorithms were then compared by executing the MD application on each of the schedules and observing the actual execution times on the schedules.

Figure 4.9(a) shows the actual execution times of the MD application with 2400 molecules for different available times for scheduling. Figure 4.9(b) shows the corresponding predicted execution times obtained from the performance models of MD application for the same schedules, problem size and resource loads. Each bar in the figures corresponds to an average of 10 experiments.

Few observations can be made from the figures. We find that in all cases the dynamic



(a) Actual Execution Times



(b) Predicted Execution Times

Figure 4.9: Comparison of Algorithms in Terms of Actual and Predicted Execution Times for Different Times Available for Scheduling on 104 Cores of Intel Xeon Cluster. Application: MD with 2400 molecules.

algorithm (DP) gives the best schedules with minimum execution times. The execution times corresponding to the DP algorithm are about 8.5% less than the execution times corresponding to the box elimination (BE) algorithm in all cases. This is because the DP algorithm is able to incrementally evaluate different schedules and determine a good schedule within the allotted times for 104 cores or processors. Our BE algorithm involves some randomness in the generation and evaluation of the schedules. Hence, the schedules determined by the BE algorithm, though competent with the schedules by the DP algorithm, result in slightly larger execution times. The random strategies employed by the simulated annealing (SA) and genetic algorithm (GA) were not able to converge on good schedules in the time allotted to the algorithms and hence their execution times were significantly higher than the DB and BE algorithms. We also find that contrary to our expectations, the execution times do not decrease with the time available for scheduling. This is because all the algorithms are able to converge to good schedules within 30 seconds for 104 cores. The algorithms are not able to find better schedules with the availability of more time for scheduling.

The results also show that the predicted execution times shown in Figure 4.9(b) are highly accurate and are within 10-30% of the actual execution times shown in Figure 4.9(a). This confirms the conclusions of our previous chapter regarding the accuracy of the performance models. We also find that the relative differences between predicted execution times of the schedules for the different algorithms, as shown in Figure 4.9(b), match the relative differences between actual execution times shown in Figure 4.9(a). The predictions also show that the dynamic programming algorithm gives the best schedules, the schedules by the box elimination algorithm are competent with the schedules by the DP algorithm, the SA and GA algorithms give the worst performance, and the execution times do not decrease with the availability of more time for scheduling. Thus the predicted execution times by the performance models can be used to adequately compare the different algorithms for larger configurations.

The results in this section showed that the DP algorithm consistently gave the best performance. However, we claim that for large number of cores or algorithms, the DP algorithm

will give poorer schedules since the incremental evaluation of schedules followed in the DP algorithm will not adequately evaluate schedules containing larger number of processors within the allotted time for scheduling. The results in this section also showed that the availability of more time for scheduling does not improve the quality of the schedules determined by the algorithms. We claim that for larger number of processors, the algorithms will take more time to determine good schedules and hence the execution times for the schedules will decrease with increasing times available for scheduling.

We verify these claims using simulation experiments to evaluate and compare the various algorithms in terms of the predicted execution times of the schedules generated by the algorithms for larger number of processors in the following subsections. We also investigate the impact of errors in predicted execution times on the quality of the schedules generated by the algorithms. The simulation experiments are explained in the following subsections. Since our algorithms are invoked in each of the clusters of a multi-cluster Grid for scheduling on the Grid, we first compare the performance of the algorithms for a single cluster setup. We then present results for multi-cluster grids.

4.3.2 Simulation Setup

We used the performance models of four parallel applications, namely, Molecular Dynamics application (MD), Eigen value solver (eigen), Symmetric Successive Over-Relaxation (SSOR) and Integer Sort (IS), for comparison of the scheduling algorithms. The performance model equations for the applications were obtained for a 8-processor Intel Pentium IV cluster with each processor having 2.8 GHz CPU and the processors were connected by a 100 Mbps switched Ethernet. In our experiments, the available CPU values ranged from 0.1-1.0 where a value of 1.0 indicates a free processor. Thus the available CPU of 0.75 for a processor in our simulation setup represents an Intel processor that is one-fourth loaded.

We compared the performance of the algorithms for different clusters containing power-of-2 number of processors sizes ranging from 32 to 1024 processors. For a simulation

experiment with a given cluster, we randomly chose the maximum available bandwidth of links in the cluster to be one of 100 Mbps, 1 Gbps, 5 Gbps and 10 Gbps. We then randomly varied the available bandwidth of each link to be within 20-80% of the maximum available bandwidth. For each experiment, we also varied *load percentage*, the ratio of *lightly*, *medium* and *heavily* loaded machines. We randomly varied the available CPU to be within 0.701-1.0 for *lightly* loaded machines, 0.351-0.7 for *medium* loaded machines and 0.05-0.35 for *heavily loaded machines*. For each experiment, the percentage of machines in a particular load category is randomly varied between 10%-80% such that the sum of all percentage equals 100. Since our algorithms are time-tunable, we also experimented with different times needed for running the scheduling heuristics. For a given cluster size, scheduling time and load percentage, we executed 50 experiments corresponding to different CPU loads and report average performance of the scheduling algorithms for the experiments. Unless otherwise stated, each data point in our graphs correspond to an average of 50 experiments.

Different Resource Parameters

Figures 4.10 and 4.11 show the effects of different load setups on the performance of the algorithms for 256 and 512 processors respectively, when the allotted scheduling time was 25 seconds. For these experiments, 70% of the machines were applied loads corresponding to one of the load categories (light, medium and heavy) and the remaining machines were applied loads from the other two load categories, resulting in three load setups. For each load setup, the average performance for 150 experiments is reported. Similar results were obtained for 64, 128 and 1024 processors. These results are shown in Appendix 8.2.

We find that dynamic programming (DP), with its intelligent incremental addition of nodes, and Box elimination method give schedules with smaller execution times than the other algorithms. The random strategies employed by the simulated annealing (SA), genetic algorithm (GA) and ant colony optimization (ACO) algorithms were not able to converge on good schedules in the time allotted to the algorithms. The learning pro-

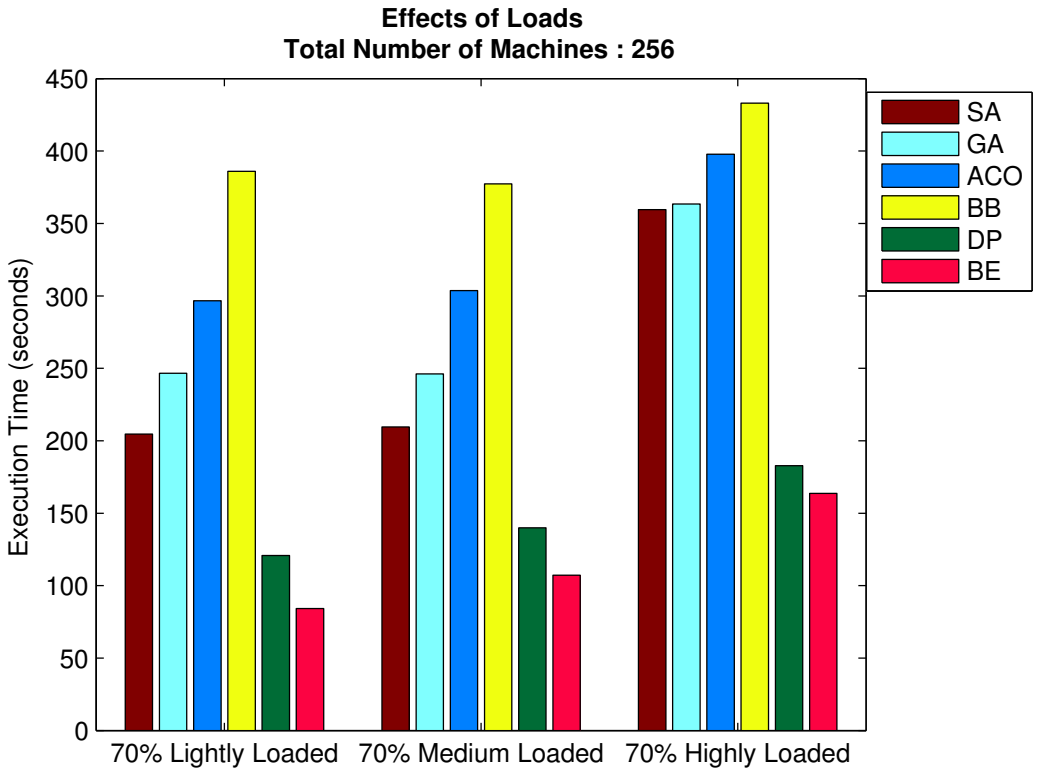


Figure 4.10: Effect of Loads on Machines for 256 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds

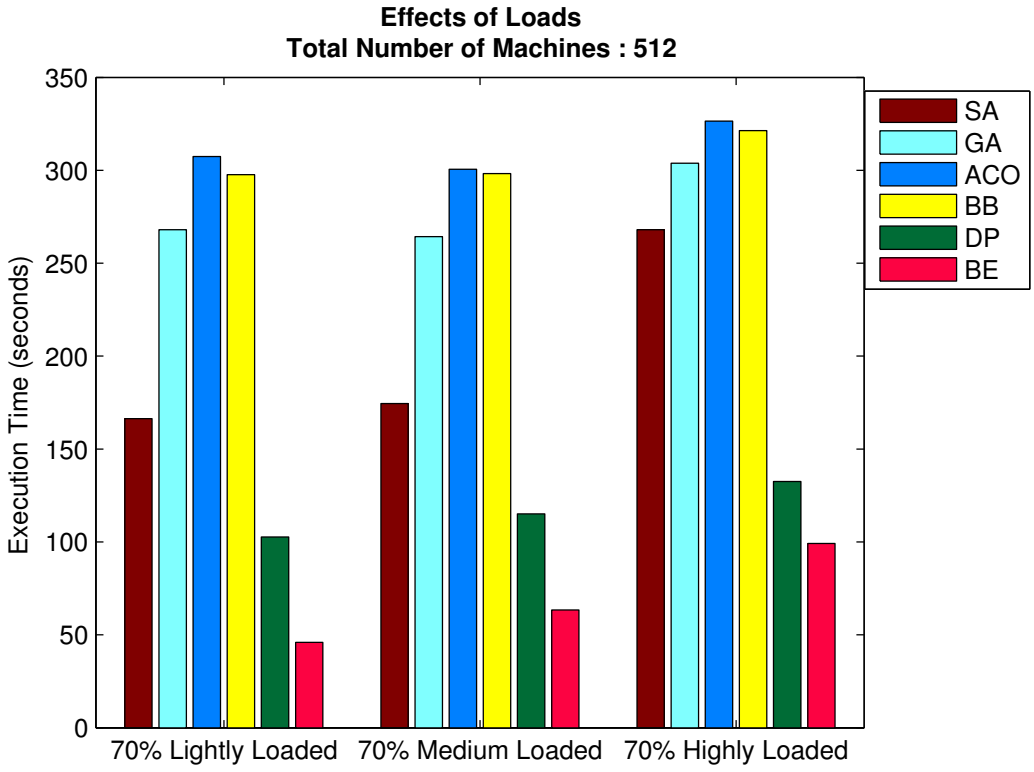


Figure 4.11: Effect of Loads on Machines for 512 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds

cess employed by the pheromone building process of ACO was not able to build large amounts of pheromones for high quality machines in the allotted time. Although, branch and bound (BB) algorithm works similar to dynamic programming by incrementally constructing schedules, the number of evaluations of schedules for a level of a tree in BB is much larger than in DP. We find that for larger number of processors, our box elimination algorithm (BE) clearly outperforms the DP algorithm by about 50%. This is because of the elimination of large number of search space regions employed by our algorithm.

We also find that the kind of loads (light, medium and heavy) does not have significant effect on the relative performance of the algorithms. However, we observe an interesting phenomenon for large number of processors. For larger number of processors, and for GA,ACO,BB and DP algorithms, the execution times are approximately the same for the 3 kinds of loads. This is counter-intuitive to the expectation that execution times corresponding to schedules generated by the algorithms will be larger in highly loaded setup than in lightly loaded setup. This is because in a tightly-coupled parallel application, the execution time is bounded by the slowest processor and hence a schedule consisting of even one highly loaded machine in the lightly loaded setup will result in performance equivalent to the highly loaded setup. The four algorithms perform large number of steps in a single iteration and hence cannot effectively eliminate all highly loaded machines in the lightly loaded setup. Although our Box Elimination algorithm performs large amount of processing for evaluation of a schedule, it is effectively able to eliminate all highly loaded machines in the lightly loaded setup since it directly deals with the CPU and network loads in its 3-D box of grid points. Thus we observe the true expected behavior for our algorithm.

Figures 4.12, 4.13 and 4.14 show the effects of using increasing percentage of lightly loaded and heavily loaded machines respectively, for 512 processors, on the schedules generated by the algorithms. We find that our BE algorithm gives significantly better performance than the other algorithms for large percentages of lightly loaded machines and small percentages of highly loaded machines. This is because, in both the cases, our algorithm is able to effectively construct schedules consisting predominantly of lightly

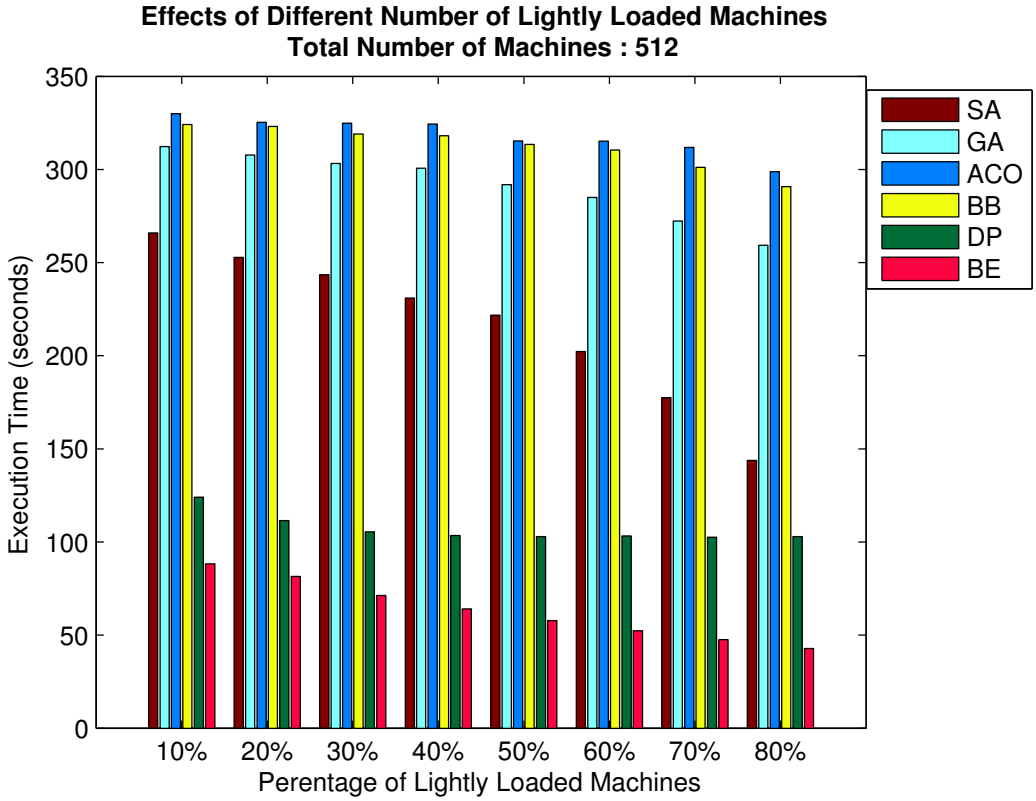


Figure 4.12: Effect of Increasing Lightly Loaded Machines for 512 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds

loaded machines. Similar results were obtained for 256 processors and are shown in Appendix 8.2.

Figure 4.15 shows the scalability of the scheduling algorithms with increasing number of processors. We find that for 32 processors, all algorithms were able to determine equivalent schedules in the 25 seconds allotted for scheduling. However as the number of processors increases, we find that dynamic programming (DP), with its intelligent incremental addition of nodes, and box elimination (BE) method gave schedules with smaller execution times than the other algorithms. We find that the performance of the BB algorithm degrades with increasing number of processors for clusters with more than 64 processors. This is because the number of schedule evaluations in BB algorithm increases exponentially with the number of nodes. We find that the DP algorithm scales well up to 256 processors where it reaches saturation. For greater than 256 processors,

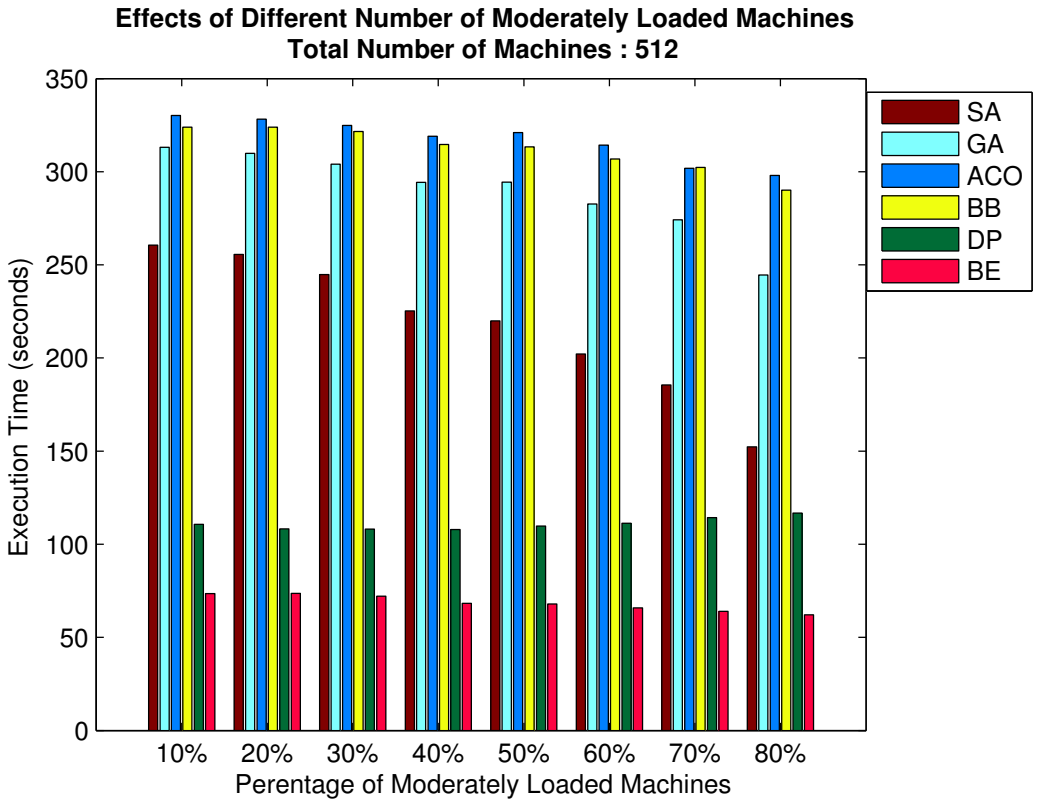


Figure 4.13: Effect of Increasing Moderately Loaded Machines for 512 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds

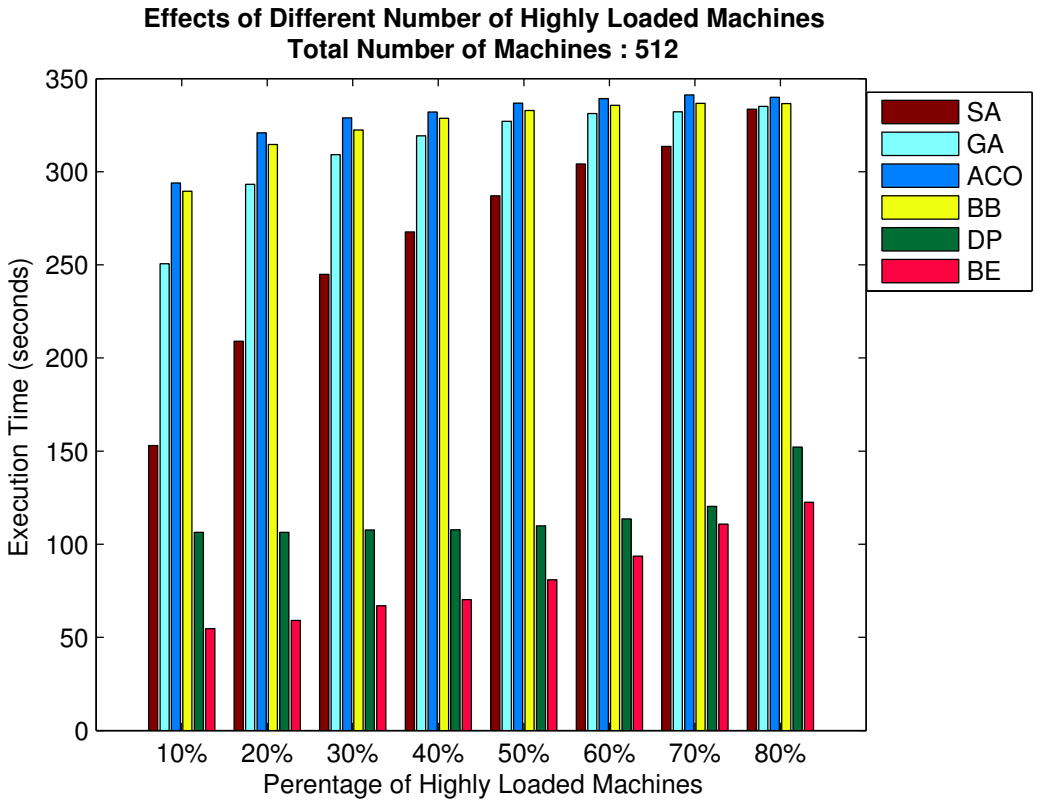


Figure 4.14: Effect of Increasing Heavily Loaded Machines for 512 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds

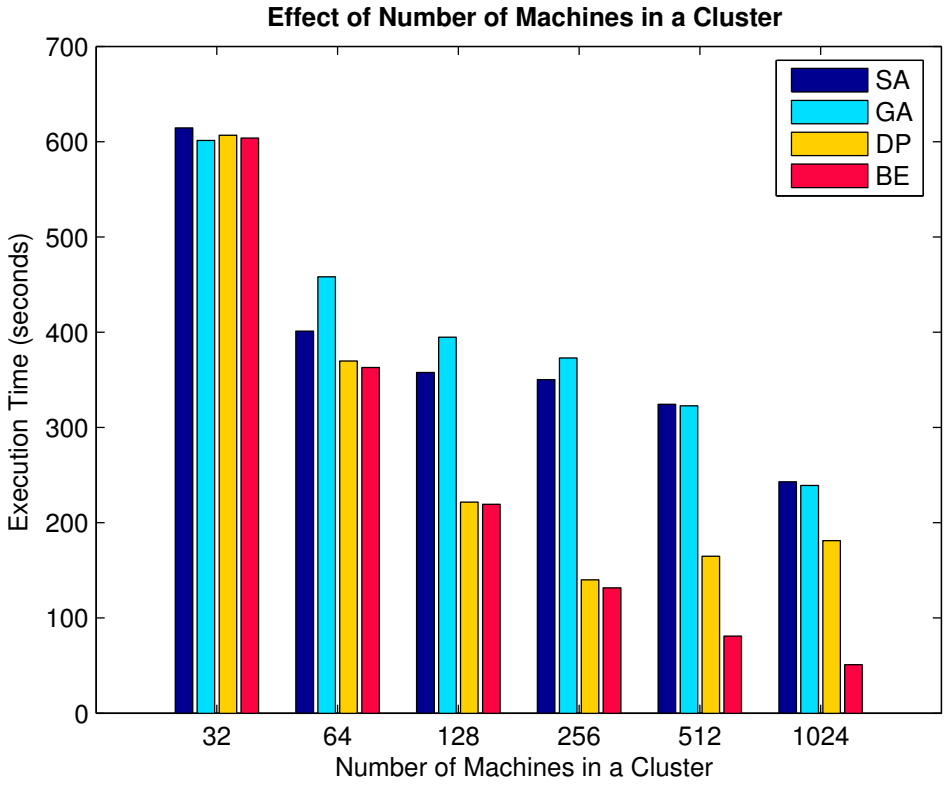


Figure 4.15: Scalability With Processors. Application: MD with 2048 molecules, 30% Lightly , 40% Medium and 30% Highly Loaded. Time for Scheduling: 25 seconds

its performance degrades with increasing number of processors. This is due to the incremental addition of nodes to schedules in the DP algorithm. For larger processors, the DP spends more time in schedules with lower number of processors. We find that our BE algorithm scales very well with increasing number of processors and clearly outperforms the DP algorithm by about 70%. This is because of the elimination of large number of search space regions employed by our algorithm. Thus, as claimed in Section 4.3.1, the DP algorithm can give efficient schedules only for small number of processors.

Different Applications

Figures 4.16 and 4.17 show the performance of the different scheduling algorithms for different applications with difference performance model equations. We observe a very interesting phenomenon in the graph. We find that the relative performance of the differ-

ent scheduling algorithms depends on the application that is scheduled. For example, we find that the DP algorithm gives worst schedules for eigen value problem even though it gives good schedules for the other applications. This is because the eigen value application is highly scalable with processors. The DP algorithm that incrementally adds nodes to the schedule spends most of its efforts in schedules with smaller number of processors. We also find that the SA algorithm gives significantly better schedules than the GA, ACO and BB algorithms for the SSOR application when compared with other applications. This graph shows that the existing optimization techniques, that have been evaluated for different number of processors and loads, will have to be reevaluated for different objective functions. We find that our BE algorithm performs better than the other algorithms for all applications except for the eigen value problem, where the schedule generated by it has a slightly higher execution time than the execution times corresponding to SA, GA, ACO and BB algorithms.

Times Needed for Scheduling

Figures 4.18 and 4.19 compare the algorithms when different times are allotted for scheduling. We find that unlike the results for real experiments shown in Section 4.3.1, the execution times of the schedules decrease with increasing times available for execution. Thus, as claimed in Section 4.3.1, time available for scheduling impacts the quality of the schedules for higher number of processors. The SA, GA and ACO algorithms, due to randomness in schedule generation, and BB algorithm, due to large times spent in evaluation of schedules at various tree levels, were not able to generate better schedules even with increase in times allotted for scheduling. We find that our BE algorithm is able to generate highly efficient schedule even with 2 seconds. This is due to its procedure of finding a center point in its 3-D grid and the associated eliminations of large search space regions. We find that the execution times of the schedules by DP algorithm decreases with increased times allotted for scheduling and approaches the performance of our BE algorithm especially on 256 processors. However, we find that on 256 processors, the DP algorithm reaches saturation at 90 seconds and does not perform better than our

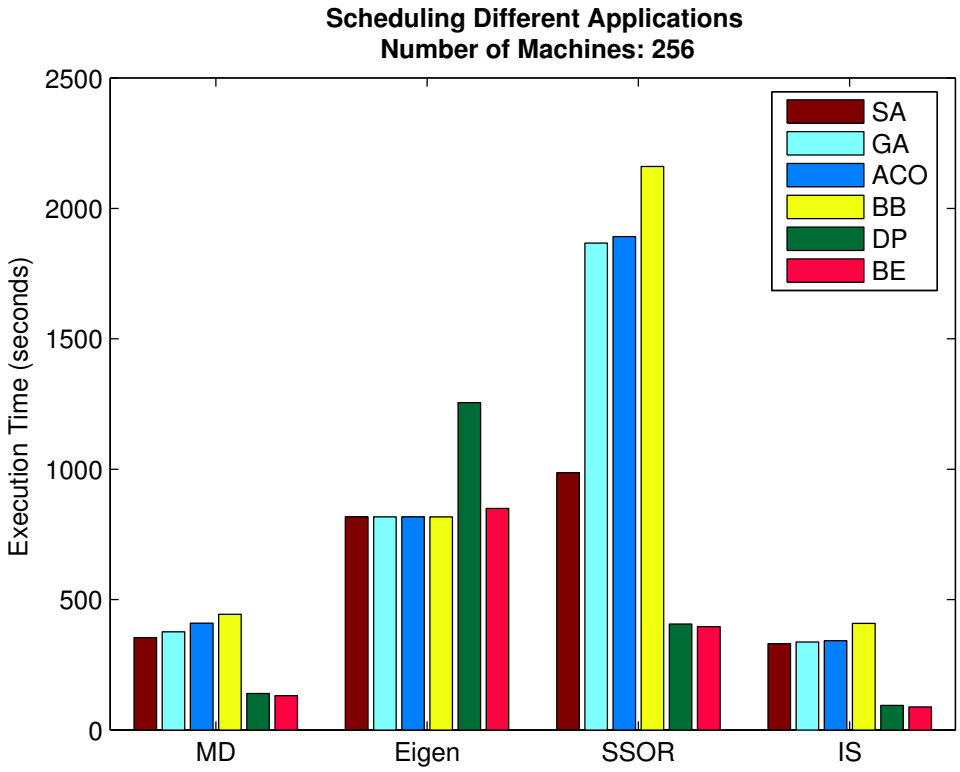


Figure 4.16: Scheduling Different Applications on 256 processors. Problem Sizes: 2048 (MD), 10000 (Eigen), 3328 (SSOR), 400000 (IS). 30% Lightly , 40% Medium and 30% Highly Loaded. Time for Scheduling: 25 seconds

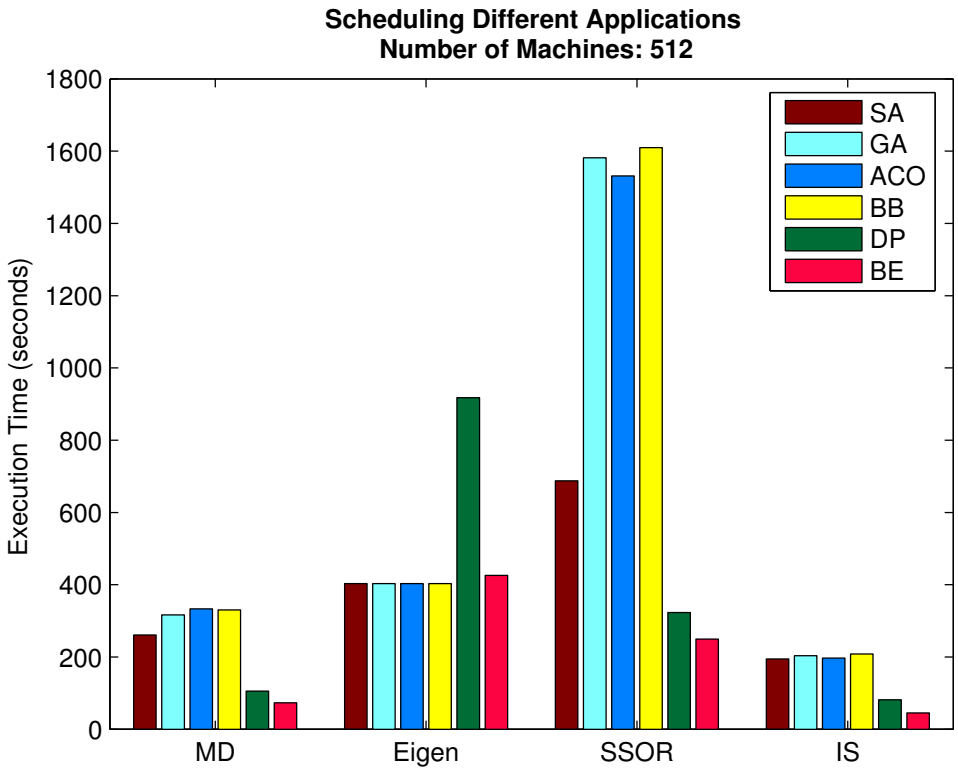


Figure 4.17: Scheduling Different Applications on 512 processors. Problem Sizes: 2048 (MD), 10000 (Eigen), 3328 (SSOR), 400000 (IS). 30% Lightly , 40% Medium and 30% Highly Loaded. Time for Scheduling: 25 seconds

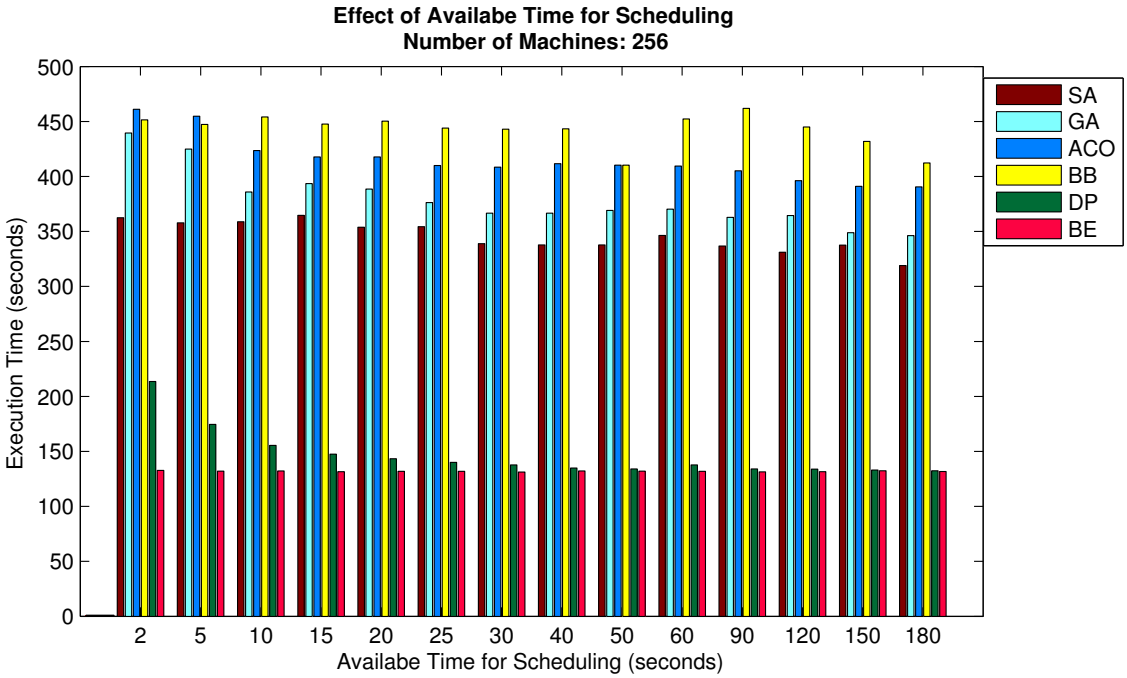


Figure 4.18: Performance of Algorithms With Different Times Available for Scheduling for 256 processors. Application: MD with 2048 molecules, 30% Lightly , 40% Medium and 30% Highly Loaded.

BE algorithm even for higher scheduling times. We find from the graphs that the schedules generated by our BE algorithm in 2 seconds are more efficient than the schedules generated by the DP algorithm in 3 minutes.

Robustness Against Performance Modeling Errors

We also compared the algorithms in terms of the robustness[54] or sensitivities of the execution times for their schedules against the errors due to performance models. In our previous work [117], we reported that the average percentage prediction errors of our performance models were 30% [1]. Hence, we modified our algorithms such that while evaluating the schedules by using performance model equations, we perturbed the predicted execution times by a random percentage in the interval [-p,+p], where p is the maximum percentage error. We then compared the schedules generated by the modified

[1] This included the modeling errors and errors in bandwidth and available CPU measurements.

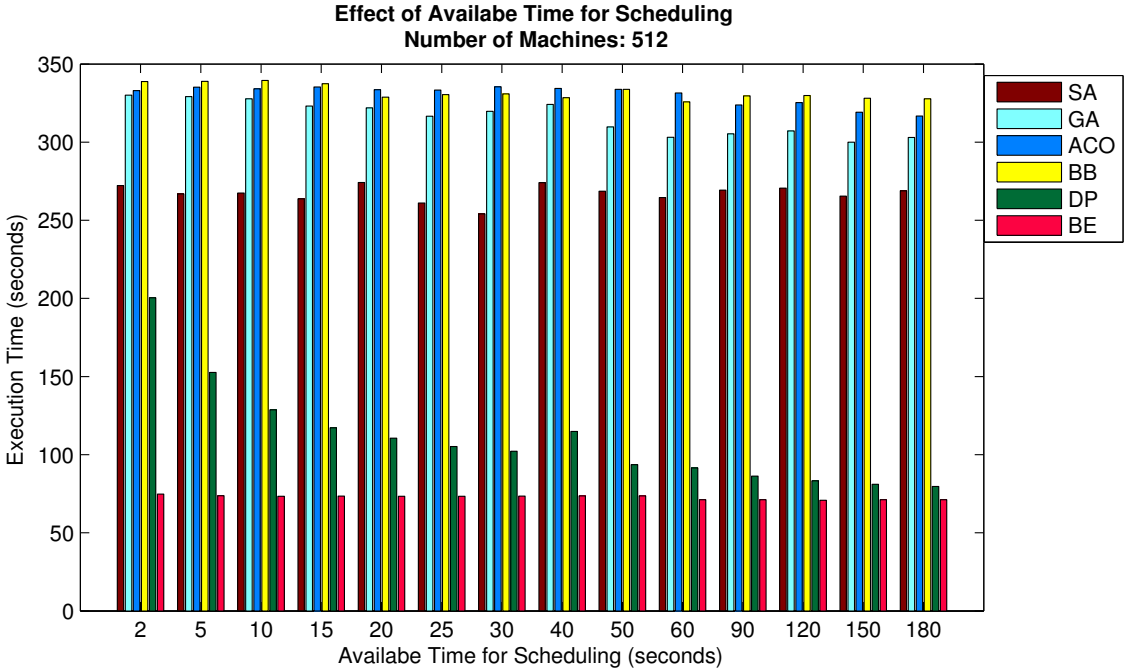


Figure 4.19: Performance of Algorithms With Different Times Available for Scheduling for 512 processors. Application: MD with 2048 molecules, 30% Lightly , 40% Medium and 30% Highly Loaded.

algorithms with the schedules generated by the unmodified algorithms in terms of the unperturbed predicted execution times using our performance models.

Figures 4.20 and 4.22 show the execution times for various values of prediction error intervals. Figures 4.21 and 4.23 show percentage increase in execution times for various values of prediction error intervals. We observe that the execution times of the schedules for the SA, GA and DP algorithms are highly sensitive to prediction errors since their percentages in increase of execution times increase with the amount of prediction errors. Although the execution times of BB are robust against performance modeling errors, the schedules generated by the algorithm are of poor quality as seen in Figure 4.20 and 4.22. Our BE algorithm shows marginal increase in execution times while generating high quality schedules. We also make an interesting observation for the ACO algorithm in all cases on 256 machine cluster and for SA algorithm in certain cases on both 256 and 512 machine clusters. The schedules of the ACO and SA algorithms are found to be better with errors in performance modeling based predictions. This happens when an

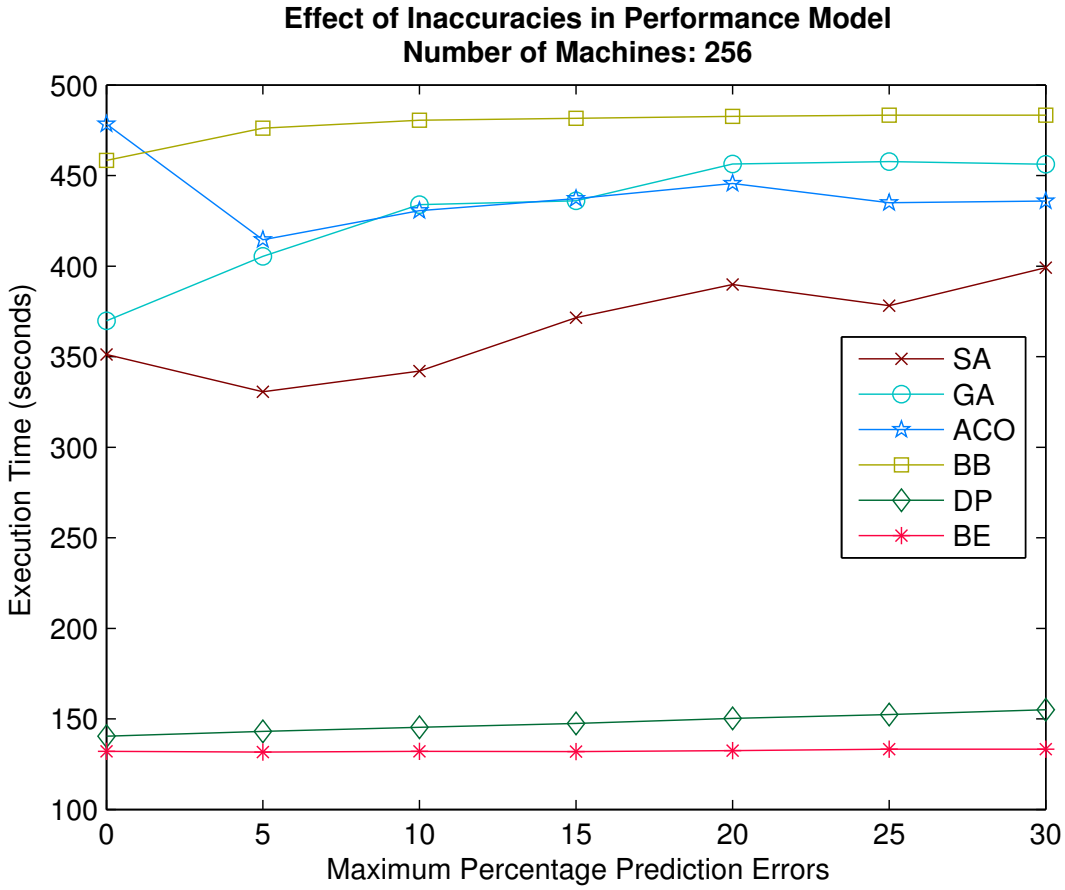


Figure 4.20: Effect of Prediction Errors on Execution Times for 256 processors. Application: MD with 2048 molecules, 30% Lightly , 40% Medium and 30% Highly Loaded, Time for Scheduling: 25 seconds.

algorithm using accurate predictions has missed evaluating several good quality solutions in the search space. These points in the search space are evaluated by chance when the algorithm uses perturbed predicted values.

Multi-Cluster Grid Experiment

In this experiment, we randomly generated 1100 multi-cluster grid setups and compared the performance of the algorithms for each setup. For each multi-cluster setup, we invoked each algorithm on each cluster. For a given algorithm, we choose the schedule generated by the algorithm for a cluster that gave the overall minimum execution time of all schedules generated by the algorithm in all clusters of the setup. We then compared the

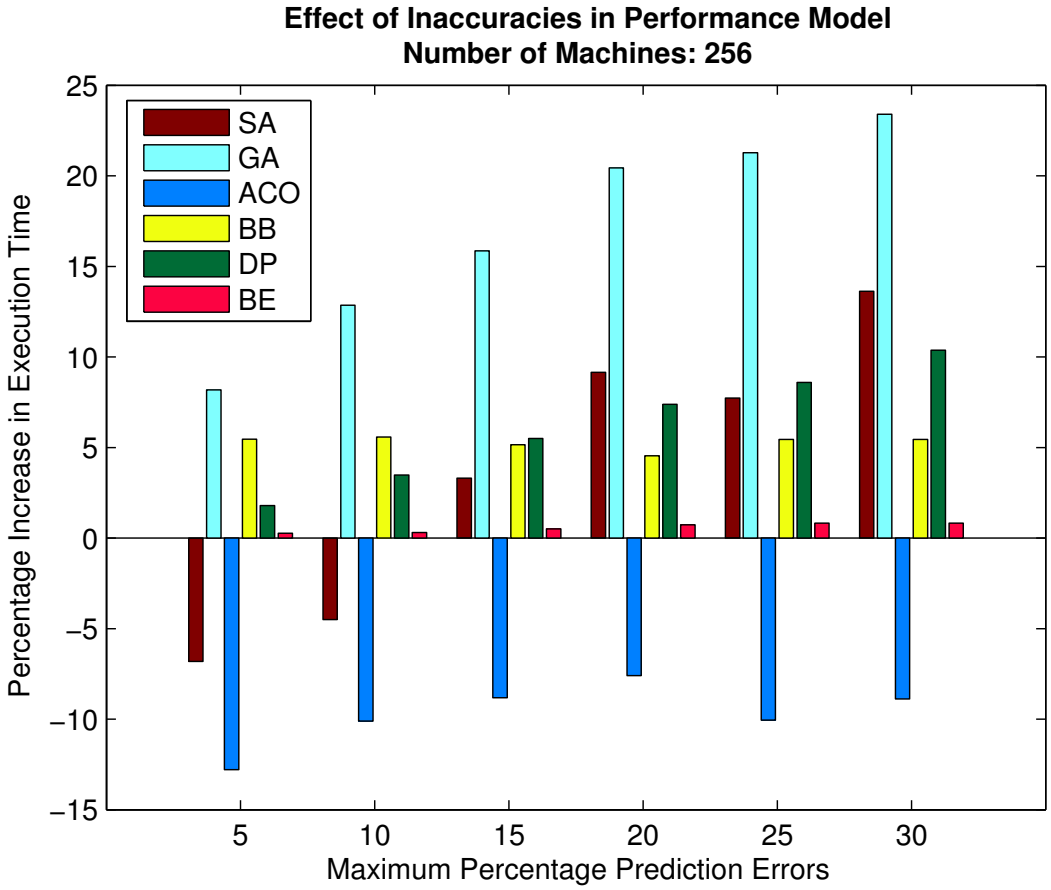


Figure 4.21: Effect of Prediction Errors on Percentage Increase in Execution Times for 256 processors. Application: MD with 2048 molecules, 30% Lightly , 40% Medium and 30% Highly Loaded, Time for Scheduling: 25 seconds.

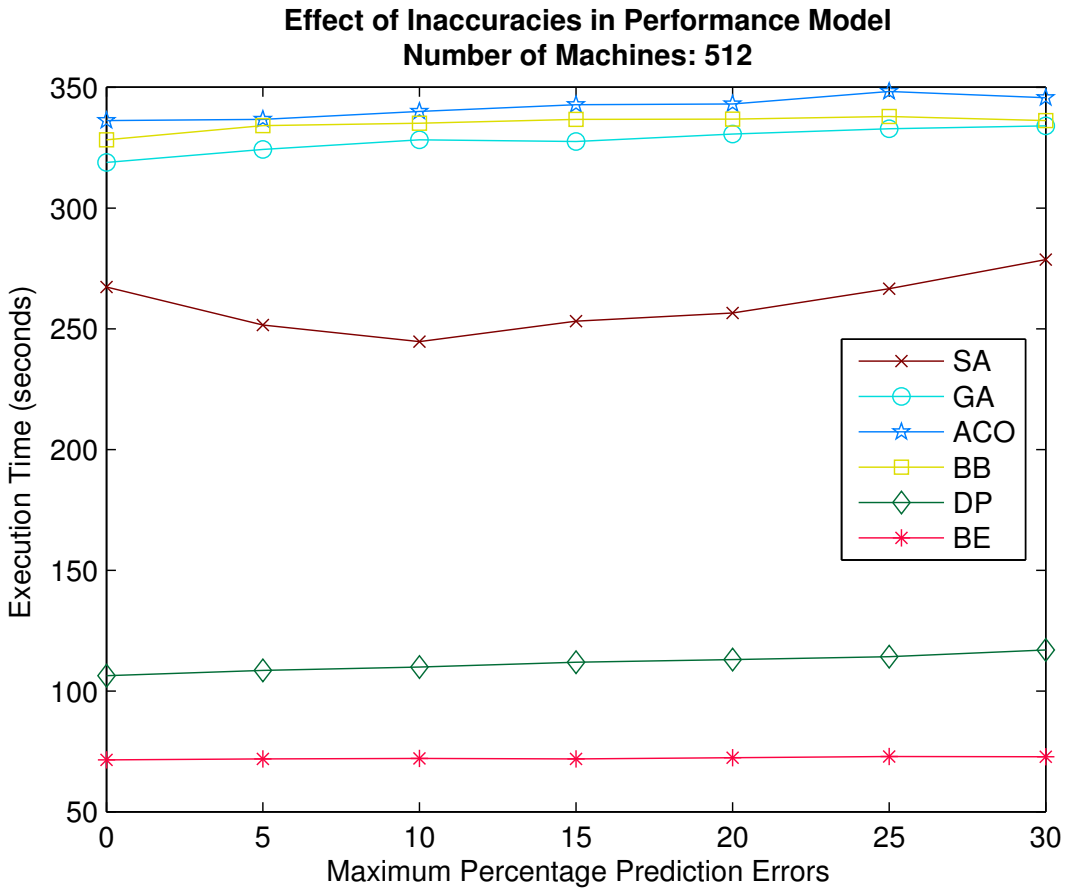


Figure 4.22: Effect of Prediction Errors on Execution Times for 512 processors. Application: MD with 2048 molecules, 30% Lightly , 40% Medium and 30% Highly Loaded, Time for Scheduling: 25 seconds.

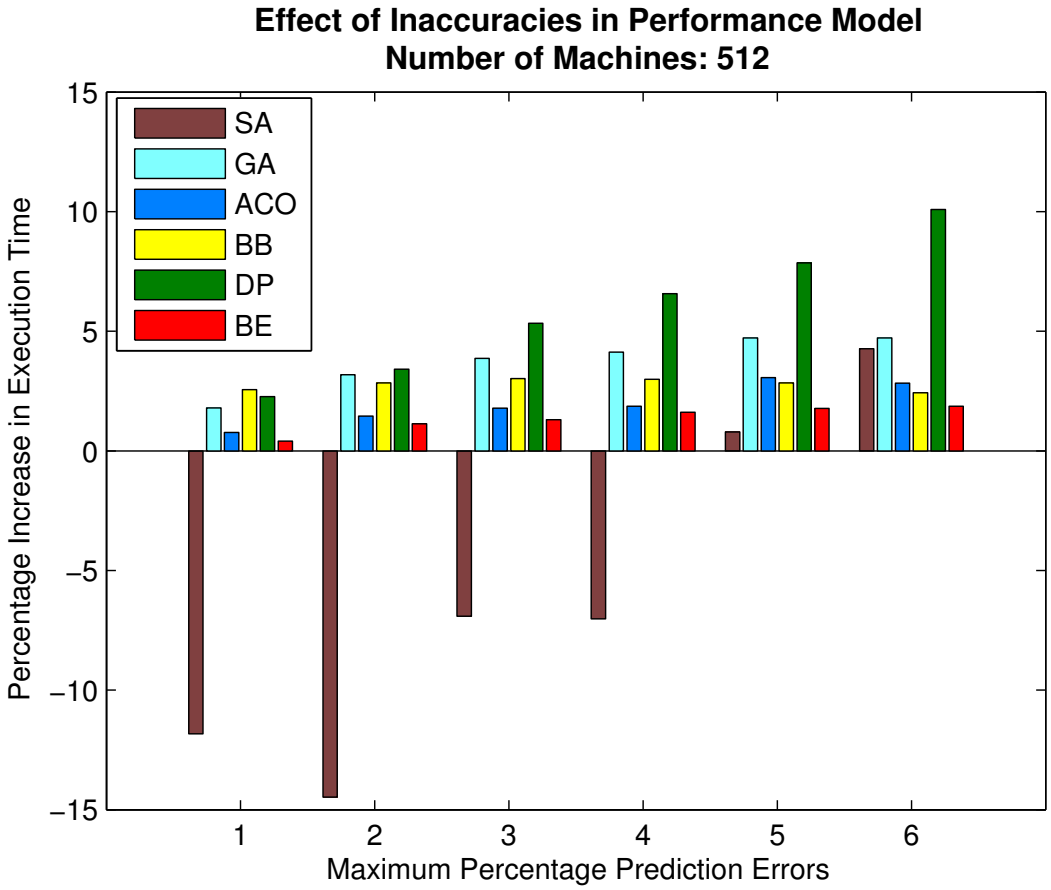


Figure 4.23: Effect of Prediction Errors on Percentage Increase in Execution Times for 512 processors. Application: MD with 2048 molecules, 30% Lightly, 40% Medium and 30% Highly Loaded, Time for Scheduling: 25 seconds.

algorithms on the basis of the overall minimum execution times. For these experiments, we used the performance modeling equation of Molecular Dynamics (MD) application. We use $U[x, y]$ to denote uniform probability distribution in the interval (x, y). For randomly generating each multi-cluster setup grid setup, we generated U[4,15] number of clusters and U[32,1024] number of processors in each cluster. In order to simulate heterogeneity of processors in different clusters, we used a random *CPU scaling factor* from the set (0.6,0.8,1.0,1.2,1.4)[2] for each cluster, and multiplied the coefficients of computational complexity of the performance model equation with the scaling factor. We also chose the maximum bandwidth of links in a cluster to be one of 100Mbps,1Gbps,5Gbps and 10Gbps. Finally, for each cluster, we also randomly fixed the percentages of lightly, medium and heavily loaded machines in the cluster. We used scheduling time of 25 seconds for these experiments.

Table 4.1 summarizes the performance of difference algorithms for the multi-cluster experiments. We find that the average execution time of schedules generated by our BE algorithm is much smaller than the execution times of the other algorithms in terms of both arithmetic and geometric means. For each multi-cluster setup, we also compared the best of the other algorithms and our BE algorithm in terms of execution times of the schedules. The average difference in execution times between our BE algorithm and the best of the other algorithms was 43% and the minimum and maximum differences were -4% and 82% respectively. Out of the 1100 experiments, our BE algorithm did not give the minimum execution times only in 12 cases in which the DP algorithm was giving the minimum times. In these 12 cases, the maximum difference in execution times of DP and BE algorithms was only 4%.

In order to determine the effect of heterogeneity of different clusters in a multi-cluster grid setup on the performance of the algorithm, we calculated the *weight* of a cluster as a product of number of machines in the cluster, the CPU scaling factor for the cluster and the percentage of lightly and medium loaded machines in the cluster. Clusters with

[2]The scaling factors are chosen from a small range since the CPU speeds of modern processors vary by small amounts.

Table 4.1: Multi Cluster Grid Setup

Algorithm	*Arithmetic Mean (seconds)*	*Std. Dev. (seconds)*	*Geometric Mean(seconds)*
SA	188.42	44.908	183.34
GA	217.41	49.46	212.19
ACO	249.89	52.21	244.84
BB	250.52	51.46	245.74
DP	124.53	21.45	122.9
BE	71.29	27.22	66.96

larger weights are expected to yield better schedules. We then calculated the ratio of the maximum and minimum weight for a multi-cluster setup. Higher values of this ratio indicate greater heterogeneity between the clusters. We divided the 1100 multi-cluster setups into different groups such that setups in the same group have the same ratio. Figure 4.24 shows the average performance of the algorithms for different groups of ratio values. We find that with increasing ratio of maximum to minimum cluster weight or increasing heterogeneity, our BE algorithm gives schedules of decreasing execution times. This behavior is expected because with increasing differences in capacities of the clusters, a scheduling algorithm chose schedules from the clusters of higher capacities or large weight values. However, this behavior is not observed for other algorithms since these algorithms chose equivalent sub-optimal schedules in all clusters irrespective of cluster heterogeneity. These characteristics of the algorithms are further illustrated in Figure 4.25, where we show the performance of algorithms for different values of maximum cluster weight in a multi-cluster grid setup.

4.4 Summary

In this chapter, we have devised a novel scheduling algorithm called Box Elimination that uses performance model of a tightly-coupled parallel application to schedule the application on a single cluster or a multi-cluster grid. By treating the search space as a set

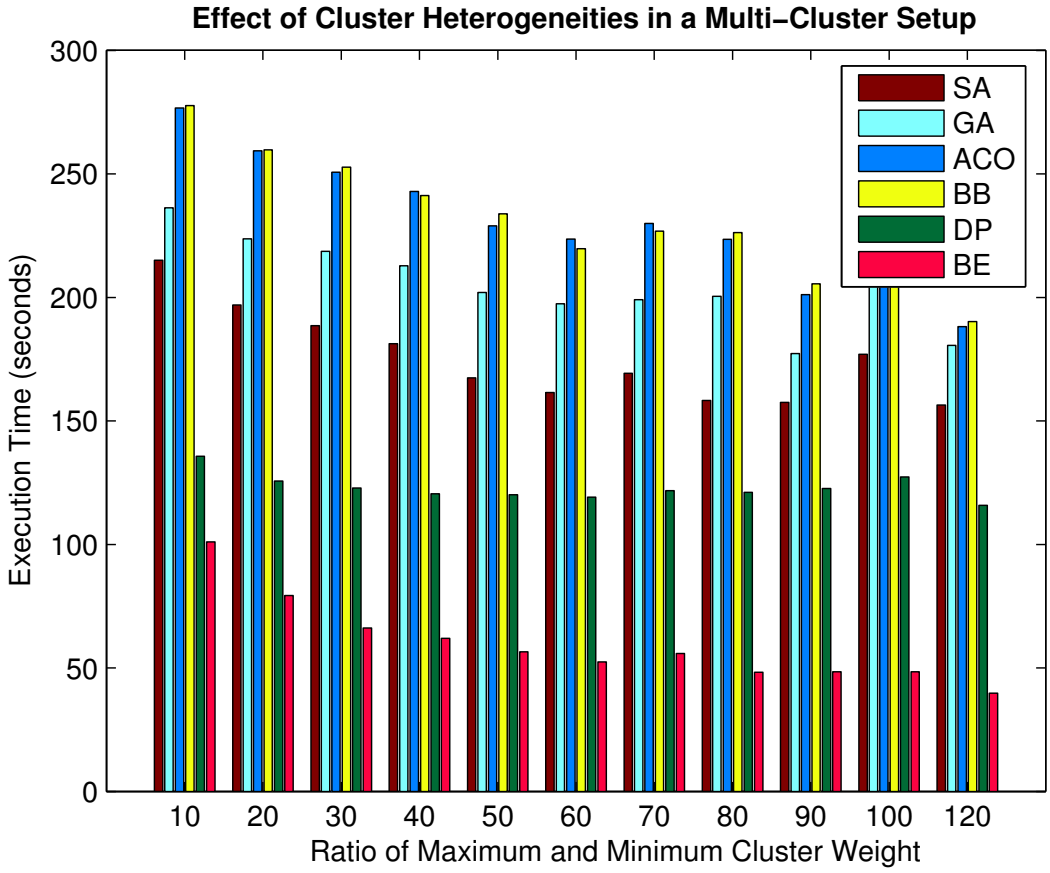


Figure 4.24: Effect of Cluster Heterogeneity in a Multi-Cluster Setup

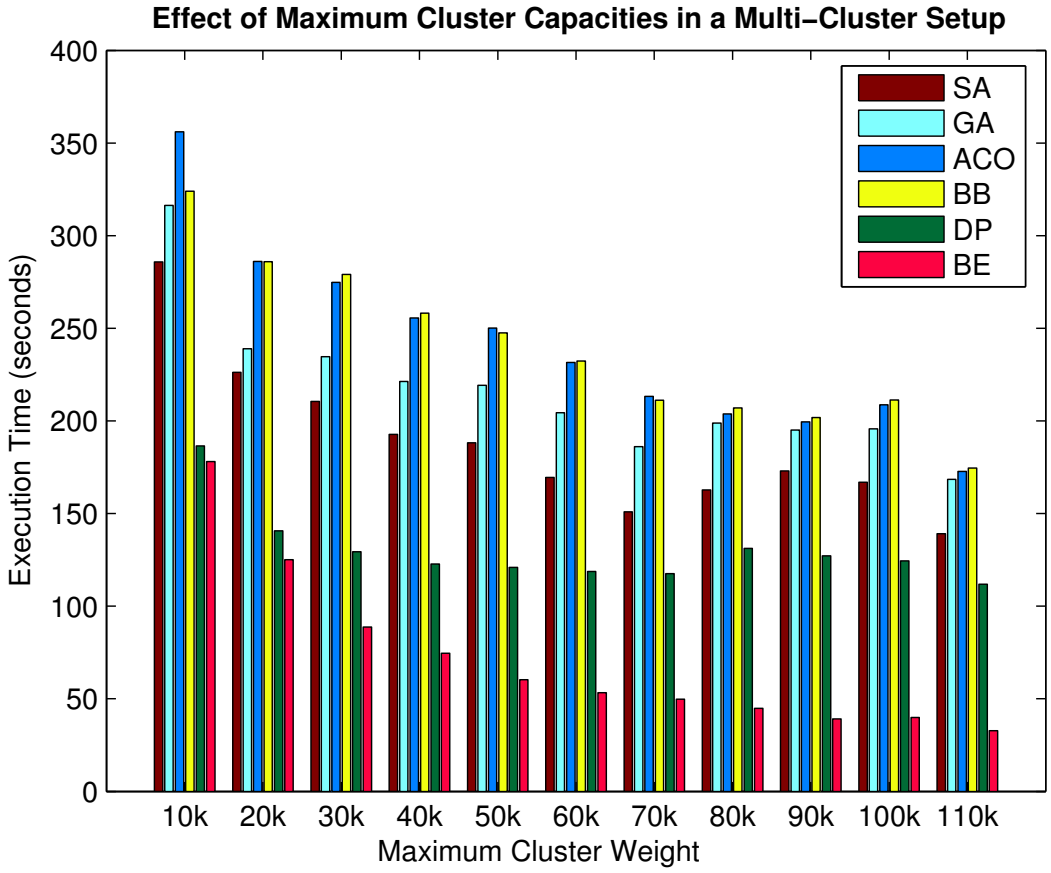


Figure 4.25: Effect of Increasing Maximum Capacities in a Multi-Cluster Setup

of performance model parameters and eliminating regions containing poor solutions, our algorithm is able to generate efficient schedules with minimum execution times for the application. By large number of simulation experiments, we showed that our algorithm generates schedules with up to 80% less execution times than other popular algorithms. We also showed that performance predictions errors have the least effect on the quality of the schedules generated by our algorithm.

Chapter 5

Rescheduling Strategies

In the previous chapters we discussed methodology for modeling and scheduling strategies for tightly-coupled parallel applications on dedicated and non-dedicated clusters. In this chapter, we present rescheduling strategies for executing multi-phase parallel application on grids. We use our performance modeling and scheduling strategies to derive rescheduling plans.

5.1 Introduction

Grids have been found to be powerful research-beds for executing various kinds of parallel applications[111, 16, 8]. Due to the dynamic nature of grid environments in terms of varying availability and performance of resources, there has been increasing interest in recent years to develop efficient rescheduling mechanisms that adapt application execution in response to change in resource characteristics[60, 129, 149, 100]. In addition to resource dynamics, long running multi-phase applications exhibit application dynamics due to different computation and communication characteristics in the different phases. Few middle-ware strategies have been built for rescheduling such applications in response to both application and resource dynamics[141, 80, 97, 148, 82]. Rescheduling involves changing the resource set on which an application is executing. As computational grids

are increasingly used for executing long running multi-phase parallel applications, it is important to develop efficient rescheduling frameworks that adapt application execution in response to resource and application dynamics. Rescheduling of parallel applications is important in non-dedicated dynamic computational grids.

There are two primary challenges in building a middle-ware framework that deals with rescheduling executing parallel applications. The first involves developing techniques or strategies for enabling an application to reschedule to different sets of resources. Typical strategies for enabling rescheduling are based on stop/restart, process spawning and process splitting/merging mechanisms. We refer to these mechanisms as *rescheduling enablers*. The second challenge involves developing policies for deciding when and where to reschedule the applications. While various efforts have concentrated on developing rescheduling enablers [138, 129, 97, 98, 43], the existing efforts on rescheduling decisions are inadequate for complex scientific parallel applications with multiple phases of different computation and communication characteristics. In this research, we focus on the second challenge of developing rescheduling decision algorithms. The techniques developed in this research can be used in conjunction with any of the rescheduling enablers.

Developing rescheduling decision algorithms involves collecting periodic information about application and resource characteristics, and using the collected information for making rescheduling decisions. Resource characteristics including CPU and network speeds and loads can be obtained using standard grid mechanisms, namely, Network Weather Service (NWS)[147, 146] and Monitoring and Directory Service (MDS)[156]. Two methods have been applied for obtaining application characteristics – application profiling and performance modeling. In application profiling, the computation and communication pattern and times are periodically obtained from the application processes during application execution. Based on the dynamic profiles, a rescheduler component remaps the processes to suitable resources in response to change in resource availability or to improve application performance or processor utilization. Most of the existing rescheduling strategies use dynamic application profiling[97, 148, 82].

While profiling is easy to implement by instrumentation of the applications or using MPI

profiling libraries, online profiling of application characteristics during application execution can incur large execution overheads. Moreover, inferring the overall application characteristics based on the process-level computation and communication patterns and times can be time consuming and cannot be done at run time. Hence most of the techniques perform rescheduling based on only small-scale changes in application behavior in terms of both space and time. They obtain a "narrow view" of the application by considering only process-level characteristics. Rescheduling based on process-level characteristics can lead to frequent rescheduling or *migration trashing*. The techniques also do not consider the potential future changes in application behavior.

Some strategies use performance modeling techniques to obtain application characteristics[141, 140, 17]. A performance model function takes as input, application characteristics including problem size, and resource characteristics including network and CPU characteristics for a set of processors, and produces as output, predicted time for execution of an application on the set of processors. The function is then invoked by the rescheduler with different candidate resource sets and the resource set for which minimum predicted execution time is obtained is used for application execution. Whenever the differences between predicted and actual execution times are beyond some predefined thresholds, the rescheduler considers rescheduling the application.

Performance models are comprehensive since they capture the behavior of the complete application or phases of the application. Model functions can be built offline and hence applications do not incur extra overheads during execution. Although earlier rescheduling frameworks relied on performance models that required application knowledge for construction[139], our work[117], on performance modeling, discussed in Chapter 3, presents performance modeling strategies to predict execution times of parallel applications for dedicated or non-dedicated resources. Our modeling strategies do not require detailed knowledge and instrumentation of the applications and can be constructed without the involvement of application developers.

In this work, we use our performance modeling and scheduling strategies to derive rescheduling plans for executing multi-phase parallel applications on grids. A reschedul-

ing plan consists of potential points in application execution for rescheduling and schedules of resources for application execution between two consecutive rescheduling points. The rescheduling plan is built for a specific set of resource characteristics and considers change in application behavior between different phases. The plan can be updated periodically during application execution thus ensuring adaptation to both resource and application dynamics. We have developed three algorithms, namely an incremental algorithm, a divide-and-conquer algorithm and a genetic algorithm, for deriving a rescheduling plan for a parallel application execution. Our algorithms use performance model functions built for a single homogeneous dedicated or non-dedicated cluster to derive rescheduling plan for the cluster. We have also developed an algorithm that uses rescheduling plans derived on different clusters to form a single coherent rescheduling plan for application execution on a grid consisting of multiple clusters. Using large number of simulations with five complex scientific multi-phase parallel applications, we show that the rescheduling plans generated by our algorithms result in significantly reduced execution times of the applications on dynamic multi-cluster grids. The rescheduling plans generated by our algorithms are also highly efficient when compared to the plans generated by brute-force methods.

Section 5.2 defines our problem of finding rescheduling plans and motivates the need for algorithms for generating plans. Section 5.3 describes our algorithms for forming rescheduling plans for an application on a cluster. Section 5.4 describes our method for forming rescheduling plans on multi-clusters using the plans generated for single clusters. Section 5.5 describes our experiment setup and presents results showing the efficiency of plans generated by our models and Section 5.7 summarizes the chapter.

5.2 Problem Statement

In our strategies for rescheduling multi-phase tightly-coupled parallel applications in multi-cluster grids, we consider executing a phase of application execution in a single cluster while the application can be rescheduled to different clusters between different

phases. We also assume that the parallel applications are *malleable* where the number of processors for application execution can be changed during the execution. Various frameworks support development of such malleable applications[138, 129].

Let us consider a multi-phase parallel application of a given problem size, S, with P contiguous phases of execution, $p_1 < p_2 < ... < p_P$, ordered by the times of execution. $p_i < p_j$ denotes that phase i is executed before phase j in the application. Let $f_1, f_2, \cdots, f_P$ represent the performance model functions, determined by Equation 3.6, for the P phases. Let us also consider T processors with their available CPU values given by a vector of size T and the available bandwidths of the links connecting the processors given by a matrix of size $T \times T$. Let $rcost$ be the worst-case overhead for rescheduling the executing application. The model functions, $f_1, f_2, \cdots f_P$, number of processors, T, the available CPUs , the available bandwidths and $rcost$ constitute the inputs to the problem of determining a rescheduling plan.

Let us consider a disjoint partitioning of the phases into L intervals, $I_1, I_2, \cdots I_L$, with the following properties.

1. Interval I_j consists of m_j contiguous phases, $p_{start_j} < p_{start_j+1}, \cdots p_{start_j+m_j-1}$ where $start_j$ is the index of the starting phase in I_j. $f_{start_j}, f_{start_j+1}, \cdots f_{start_j+m_j-1}$ are the corresponding performance model functions of the phases in interval I_j.
2. The intervals are ordered: $I_1 < I_2 < \cdots I_L$ where $I_l < I_m$ denotes that phases in interval I_l is executed before the phases in interval I_m.
3. The cumulative performance model function, g_j, for interval I_j is given as:

$$g_j = \sum_{k=1}^{m_j} f_{start_j+k-1} \tag{5.1}$$

4. t_j is the execution time corresponding to the schedule determined by box elimination heuristic for interval I_j, using the performance model function g_j, problem size S and the resource characteristics.

The concept of phases and intervals are illustrated in Figure 5.1. The disjoint set of intervals along with the schedules for executing the intervals represent a rescheduling

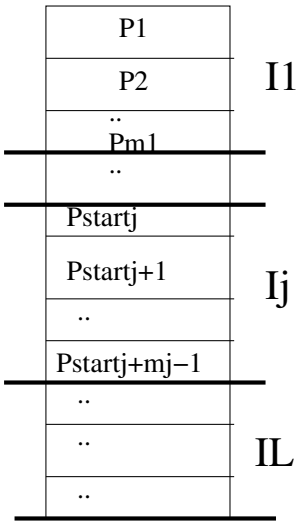


Figure 5.1: Phases and Intervals

plan. The application can be potentially rescheduled at the end of an interval.

The problem is to find an optimal rescheduling plan $\{I_1 < I_2 < \cdots I_{Lopt}\}$ such that

$$\sum_{i=1}^{Lopt} t_i + (Lopt - 1)rcost \tag{5.2}$$

is minimized subject to the condition

$$t_i > t_{thres}, \forall i \in \{1, 2, \cdots Lopt\} \tag{5.3}$$

In Equation 5.2, the first term represents the total time spent in execution of the application and the second term represents the total rescheduling cost of the rescheduling plan. The condition given in Equation 5.3 is used to avoid frequent rescheduling or rescheduling trashing by stipulating a minimum threshold of t_{thres} for execution of an interval. For our work, we chose 20 minutes for t_{thres} since the rescheduling cost can be 5 – 10 minutes in some real reconfiguration frameworks[141]. We define a valid interval as the interval for which the predicted execution time is less than t_{thres} and a valid rescheduling plan as the plan in which all intervals are valid.

The number of ways to partition a contiguous set of N items into L intervals is given by

$$part(N, L) = part(N - 1, L - 1) + part(N - 1, L) \tag{5.4}$$

Thus the total number of ways to partition a contiguous set of N items into different number of intervals is equal to the sum of the elements in the $(N-1)^{th}$ row of Pascal's triangle[108] and is equal to $2^{(N-1)}$. Thus for an application consisting of 50 phases of execution, the number of candidate partitions or rescheduling plans is 5×10^{14}. Since evaluating such high number of candidate rescheduling plans is time-consuming and since the rescheduling plan will have to be updated periodically during runtime to adapt to resource dynamics, we adopt various heuristics for generating efficient rescheduling plans.

5.3 Algorithms for Generating Rescheduling Plans in a Single Cluster

All our algorithms accept as input the application problem size, $problemSize$, the total number of execution phases for the application, $phaseCount$, the performance model functions, $f_1, f_2, \cdots f_{phaseCount}$, for the phases, the rescheduling cost, $rcost$, the total number of processors, P, and the available CPU and bandwidth values of all the processors and links, respectively. The algorithms produce as output a rescheduling plan, $plan$, which includes a set of identifiers of the phases at the end of which the application can be potentially rescheduled and the schedules for executing the application between the phases. We refer to these phases as *interval markers*.

5.3.1 Incremental Algorithm

This algorithm incrementally tries to construct a rescheduling plan by adding intervals to the plan in increasing order. The pseudo code of the algorithm is given in Figure 5.2.

For forming the first interval, it considers adding phases to the interval in increasing order starting from the first phase, p_{start}. After each addition of a phase, it checks if the interval is valid using the condition in Equation 5.3 (line 11). Once the algorithm finds a valid interval, with the last phase of the interval denoted by p_{end}, it uses Box Elimination algorithm to find the schedule, $prevSched$, for the interval using the cumulative

function of all performance model functions for the phases in the interval, the problem size and resource characteristics (lines 7 and 15). It then tries to form a lager interval by adding the next phase, p_{end+1}, to the interval (line 16) and again invoking Box Elimination algorithm to determine a new schedule, $currentSched$ for this larger interval (line 18). It predicts the execution time of the larger interval when executed on the schedules, $prevSched$ and $currentSched$ as $t_{prevSched}$ and $t_{currentSched}$, respectively (lines 19 – 22). The algorithm compares the cost of continuing on $prevSched$, $t_{prevSched}$, and the cost of rescheduling to $currentSched$, $(t_{currentSched} + t_{rcost})$. Rescheduling to $currentSched$ involves a rescheduling cost, $rcost$. If $t_{prevSched} < (t_{currentSched} + rcost)$ (line 23), then the algorithm decides that the application behavior has not changed significantly in the newly added phase. This is because the earlier schedule, $prevSched$, determined for executing the phases, $p_{start} - p_{end}$, is found to be equivalent to $currentSched$ in executing the phases, $p_{start} - p_{end+1}$, indicating that the application behavior does not significantly change in the phase, p_{end+1}, to necessitate changing the machines considered for execution. In this case, the algorithm considers adding more phases to the interval (line 24). Thus this algorithm uses the quality in schedules to detect significant change in application behavior.

If $(t_{currentSched} + rcost) < t_{prevSched}$ (line 26), the algorithm determines that the application behavior has changed significantly in the phase, p_{end+1}. In this case, the algorithm forms the interval, $p_{start} - p_{end}$, and adds p_{end} as an interval marker to the rescheduling plan and begins the new interval from the phase, p_{end+1} (line 27). This process of forming subsequent intervals by incrementally adding phases and using the change in quality of schedules to decide the interval markers continues till the algorithm adds the last phase of application execution to the rescheduling plan.

This algorithm tries to find a balance between reducing the rescheduling costs by forming larger intervals and reducing the execution times of the intervals by forming new intervals when it encounters a phase with different application behavior.

1 **Algorithm:**Incremental Algorithm
2 $plan = \emptyset$; $start = 1$; $end = 1$;
3 **while** $end \leq phasecount$ **do**
    /* Form the next interval */
4 $I = \emptyset$;
5 **while** $end \leq phasecount$ **do**
        /* For meeting condition given by Equation 5.3 */
6 $g = f_{start} + f_{start+1} + f_{start+2} + \cdots + f_{end}$;
        /* Cumulative function of performance models of phases p_{start} to p_{end} */
7 $currentSched = BE(problemSize, P, availableCPU, availableBW, g)$;
8 $currentP =$ number of processors corresponding to $currentSched$;
9 $currentAvailCPU =$ average available CPU of the processors in $currentsched$;
10 $currentAvailBW =$ average available bandwidth of the processors in $currentSched$;
11 $t_{currentSched} = g(problemSize, currentP, currentAvailCPU, currentAvailBW)$;
12 **if** $t_{currentSched} < t_{thres}$ **then** $end = end + 1$;
13 **else** break ;
14 **end**
    /* Found an interval that meets the condition by Equation 5.3. The following attempts to expand the interval. */
15 $prevSched = currentSched$;
16 $end = end + 1$;
17 $g = f_{start} + f_{start+1} + f_{start+2} + \cdots + f_{end}$;
18 $currentSched = BE(problemSize, P, availableCPU, availableBW, g)$;
19 $(currentP, currentAvailCPU, currentAvailBW) =$ parameters corresponding to $currentSched$;
20 $t_{currentSched} = g(problemSize, currentP, currentAvailCPU, currentAvailBW)$;
21 $(prevP, prevAvailCPU, prevAvailBW) =$ parameters corresponding to $prevSched$;
22 $t_{prevSched} = g(problemSize, prevP, prevAvailCPU, prevAvailBW)$;
23 **if** $t_{prevSched} < (t_{currentSched} + rcost)$ **then**
        /* previous schedule is better for the current sets of phases */
24 $I = I \cup p_{end}$; $end = end + 1$; go to line 17 ;
25 **end**
26 **else**
27 $plan = plan \cup (I, currentSched)$; $start = end$; /* For the next interval */
28 **end**
29 **end**

Figure 5.2: Incremental Algorithm (IA)

Correctness

The incremental algorithm builds a rescheduling plan based on the assumption that a significant change in the complexity of a new phase from the previous phases marks the beginning of a new interval with different computational and communication complexity from the previous interval. At each stage, the algorithm incrementally considers the next phase, decides whether the application has to be rescheduled and either adds the phase to the current interval or to a new interval. During this decision, the algorithm considers rescheduling costs and the minimum time threshold for an interval.

Since the algorithm incrementally adds the phases to either existing intervals or to a new interval, each of the application phases will be contained in an interval and will be contained in exactly one interval.

Complexity

The total number of steps in incremental algorithm will be equal to the the number of phases in the application. For each step, box elimination algorithm is involved. So the complexity of incremental algorithm will be approximatey equal to $O(phases \times complexity\_BE)$ where $complexity\_BE$ is the complexity of the box elimination algorithm.

5.3.2 Division Heuristic

In this scheme, we progressively divide the application execution into increasing number of intervals. We form the best rescheduling plan for a given number of intervals using the best rescheduling plans for the lower number of intervals. The best rescheduling plan for some number of intervals, $inter$, denoted as $bestplan_{inter}$, is the plan for which the total predicted application execution time is minimum among all the candidate rescheduling plans containing $inter$ intervals. The total predicted execution time for a rescheduling plan is calculated by adding the sum of the predicted times for the intervals in the plan and

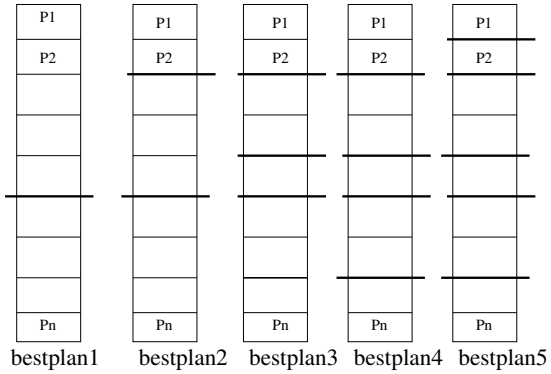


Figure 5.3: Illustration of Division Heuristic

the sum of the rescheduling costs. The predicted time for an interval is calculated as in the incremental algorithm by forming the cumulative function of all the performance model functions for the phases in the interval and using the Box Elimination algorithm with the cumulative function to find the best schedule of machines for executing the interval and the corresponding execution time.

We first form $bestplan_2$ for two intervals by considering all candidate rescheduling plans with two intervals. Each candidate rescheduling plan corresponds to assigning one of the phases of application execution as an interval marker. We then try to form $bestplan_3$ for three intervals using $bestplan_2$ for two intervals. For forming the best rescheduling plan, $bestplan_{inter}$, for $inter$ intervals, using $bestplan_{inter-1}$ for $inter - 1$ intervals, we use the interval markers in $bestplan_{inter-1}$ and consider the phases other than these interval markers as candidates for another interval marker. Thus, the interval markers in $bestplan_{inter-1}$ are contained in the rescheduling plans evaluated for $inter$ intervals. This is illustrated in Figure5.3.

We continue this procedure for higher number of intervals until no valid rescheduling plan can be formed for a given number of intervals. In this case, we consider the best rescheduling plans generated for all the lower number of intervals and choose the overall best rescheduling plan.

The division heuristic is suitable for applications that exhibit hierarchical behavior of application execution. These applications contain a small set of large phases of execution

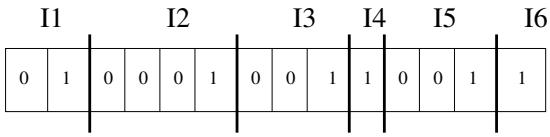


Figure 5.4: Chromosome Representation of Rescheduling Plan

and each of these large phases can be further divided hierarchically into smaller phases.

5.3.3 Genetic Algorithm

Genetic algorithm has been used in a number of optimization problems involving large search spaces [87, 64, 79]. In this algorithm, a population of chromosomes is initially generated and the chromosomes undergo cross-over and mutations across various generations. The chromosomes with high fitness values are retained over successive generations.

In our problem of determining an efficient rescheduling plan, a chromosome represents a candidate rescheduling plan. We represent the chromosome or rescheduling plan as an array where the number of elements in the array is equal to the number of phases of application execution. The array consists of 0's and 1's with the 1's representing the interval markers. This is illustrated in Figure 5.4.

We calculate the fitness of a chromosome using Equation 5.6.

$$reverseFitness = execTime \times (1 + \frac{violationCount}{totalIntervals}) \times (1 + \frac{avgViolationExtent}{execTime}) \tag{5.5}$$

$$fitness = \frac{1}{reverseFitness} \tag{5.6}$$

where $execTime$ is the predicted execution time of the rescheduling plan or chromosome calculated as shown in the incremental and genetic algorithms by using Box Elimination schedules for each interval and adding the sum of the predicted execution times of the intervals and the rescheduling costs. We define *violating interval* as an interval that does not satisfy the condition specified in Equation 5.3. $violationCount$ is the number of violating intervals and $avgViolationExtent$ is the average of the extent of the violations in the

violating intervals. The violation extent of a violating interval is the difference between the threshold limit, t_{thres} and the execution time of the interval. Thus the fitness metric attempts to minimize the rescheduling time and improve the validity of a rescheduling plan. After executing our genetic algorithm for 50 generations, we choose the best valid rescheduling plan with the highest fitness value.

The algorithm is shown in Figure 5.5. The mutation step (lines 9-16) tries to generate a chromosome with high fitness value from base chromosomes by splitting a large interval into two equal intervals and merging a small interval with the neighboring interval. Splitting a large interval into two small intervals will reduce the value of the first term, $execTime$, of Equation 5.5 since the sum of the execution times of the two small intervals will be less than the execution time of the large interval. This is because efficient schedules can be formed for small intervals with small number of phases while schedule of machines for large interval cannot efficiently meet the requirements of the characteristics for the large number of phases in the interval. Merging of two intervals helps to minimize the second two terms of Equation 5.5 and hence improves the validity of the plan.

5.4 Rescheduling Plans for Multi-Cluster Grids

The algorithms in the previous section are intended for single-cluster rescheduling plans (SCRPs) for a cluster based on the performance models obtained for the cluster. Invoking an algorithm on different clusters with the corresponding performance models on the clusters will lead to formation of different SCRPs for the different clusters. This section discusses an algorithm for the formation of a coherent rescheduling plan involving multiple clusters of a grid using the SCRPs generated for the individual clusters by the algorithms described in the previous section. The algorithm for multi-cluster rescheduling plan (MCRP) takes as input the application problem size, $problemSize$, the total number of execution phases for the application, $phaseCount$, the number of clusters, $clusterCount$, a $clusterCount \times phaseCount$ matrix of performance model functions, f, where $f_{i,j}$ denotes the performance model obtained on the i^{th} cluster for the j^{th} phase,

Algorithm: Genetic Algorithm

Randomly generate initial population of 200 chromosomes ;
Evaluate the fitness of chromosomes using Equation 5.6 ;
for *generations = 1 to 50* **do**
    /* CrossOver */
    Divide chromosomes into pairs ;
    for *each pair of chromosome* **do**
        Pick a random chromosome position ; Cross-over chromosome segments from that position ;
    end
    /* After cross-over, there are 400 chromosomes in the population */
    /* Mutation */
    for *each of 200 newly generated chromosomes* **do**
        Find the largest interval, containing maximum number of phases ;
        Divide the interval into two equal-sized intervals containing the same number of phases by converting the 0 of the middle phase into 1 ;
        Find the smallest interval, containing minimum number of phases ;
        if *predicted exec time of the interval* $< t_{thres}$ **then**
            Merge the interval with the neighboring interval by converting the 1 of the last phase into 0 ;
        end
    end
    Evaluate the fitness of 400 chromosomes using Equation 5.6 ;
    /* Selection */
    Select the best 200 chromosomes with high fitness values for next generation;
end

Figure 5.5: Genetic Algorithm (GA)

a vector of total number of processors, P, where P_i denoted the number of processors in the i^{th} cluster, and the available CPU and available BW values of the processors and links of all the clusters, the cost for rescheduling in the same cluster, $intraClusterCost$ and the cost for rescheduling to a different cluster, $interClusterCost$. The algorithm is shown in Figure 5.6.

The MCRP generation algorithm begins by invoking one of the SCRP generation algorithms, $scrpAlgo$, described in the previous section for generating rescheduling plans for each of the clusters (line 5). It then decides a schedule for executing the first interval by first sorting the first intervals in the rescheduling plans of all the clusters in terms of the number of phases in the intervals (line 7). It then chooses the smallest first interval, $multiClusterInterval_1$, that is valid in all the clusters (line 14). Note that an interval that is valid in one cluster may not be valid in another cluster due to the different performance models and resource capacities on the different clusters. The algorithm then finds schedules in each of the clusters for executing $multiClusterInterval_1$ using Box Elimination (lines 18-23) and chooses the cluster and schedule of machines in the cluster with the minimum predicted execution time for $multiClusterInterval_1$ execution (line 24). The algorithm adds the $multiClusterInterval_1$ and the corresponding schedule to MCRP (line 26).

For determining the j^{th} interval of MCRP and schedule for the interval, the algorithm regenerates SCRPs for the remaining phases not contained in MCRP for the individual clusters and forms $multiClusterInterval_j$ similar to the formation of $multiClusterInterval_1$ for the first interval. The algorithm then determines three predicted execution times:

1. $timeInPrevSchedule$ which is the predicted execution time for $multiClusterInterval_j$ when executed on the schedule determined for the previous interval, $multiClusterInterval_{j-1}$ (lines 29 and 30). This schedule is denoted as $sameSchedule$. In this case, the application will be continued to execute on the same schedule and will not be rescheduled.
2. $timeinPrevCluster$ which is the predicted execution time for

$multiClusterInterval_j$ when executed on the schedule obtained by invoking the Box Elimination algorithm for $multiClusterInterval_j$ for the cluster containing the schedule determined for the previous interval (lines 31 and 32). This schedule is denoted as $scheduleonSameCluster$. In this case, the cost for rescheduling to a different schedule on the same cluster, $intraClusterCost$, is added to $timeinPrevCluster$.

3. $bestTime$ which is the minimum of predicted execution times for $multiClusterInterval_j$ obtained on all the clusters by invoking Box Elimination algorithm for the interval in each of the clusters (line 24). This schedule is denoted as $bestSchedule$. In this case, the cost for rescheduling to a different schedule on a different cluster, $interClusterCost$, is added to $timeinPrevCluster$.

The algorithm then performs a three-way comparison between $timeInPrevSchedule$, $(timeinPrevCluster + intraClusterCost)$ and $(bestTime + interClusterCost)$ to decide if $multiClusterInterval_j$ has to be executed on $sameSchedule$, $scheduleonSameCluster$ or $bestSchedule$ (lines 33-41). The algorithm continues performing the above steps until it includes all the phases in MCRP. The algorithm for generating MCRP is illustrated in Figure 5.6.

5.5 Experiments

We performed large number of simulations to evaluate our rescheduling strategies in terms of the predicted execution times of the applications. The predicted times include the rescheduling costs.

5.5.1 Applications

We used the five large scale multi-phase parallel applications described in Section 3.10 for evaluating our rescheduling strategies. These applications are MD, ChaNGa, Athena, LAMMPS, and MPB. By using active profiling[122, 121] and manual analysis of source

Algorithm:MCRPAlgo

$MCRP = \emptyset$; $currentInterval = 0$;
while *number of phases in* $MCRP \neq phaseCount$ **do**
  for $j = 1$ *to* $clusterCount$ **do**
    $SCRP_j = scrpAlgo(problemSize, phaseCount, f_j, P_j, availCPU_j, availBW_j)$;
  end
  Sort first intervals of SCRPs to form a vector of $sortedFirstIntervals$;
  for $j = 1$ *to* $clusterCount$ **do**
    $g_j =$ cumulative function of performance models in $sortedFirstIntervals_j$;
    $sched_j = BE(problemSize, P_j, availableCPU_j, availableBW, g_j)$;
    $(currentP, currentAvailCPU, currentAvailBW) = sched_j$ parameters;
    $t_j = g_j(problemSize, currentP, currentAvailCPU, currentAvailBW)$;
    if $t_j > t_{thres}$ **then**
      $multiClusterInterval_{currentInterval} = sortedFirstIntervals_j$;
      break;
    end
  end
  for $j = 1$ *to* $clusterCount$ **do**
    $g_i =$ cumulative function of models in $multiClusterInterval_{currentInterval}$;
    $newSched_j = BE(problemSize, P_j, availableCPU_j, availableBW, g_j)$;
    $(currentP, currentAvailCPU, currentAvailBW) = newSched_j$ parameters;
    $newT_j = g_j(problemSize, currentP, currentAvailCPU, currentAvailBW)$;
  end
  $bestTime$ = minimum of $newT$; $bestSchedule$ = corresponding $newSched$;
  if $MCRP = \emptyset$ **then**
    $MCRP = MCRP \cup (multiClusterInterval_{currentInterval}, bestSched)$;
  end
  else
    $sameShedule$ = schedule for $multiClusterInterval_{currentInterval-1}$;
    $timeInPrevSchedule$ = predicted execution time for $multiClusterInterval_{currentInterval}$ on $sameSchedle$;
    $scheduleonSameCluster$ = scheduled obtained by invoking BE on the cluster containing $sameSchedule$;
    $timeinPrevCluster$ = predicted execution time for $multiClusterInterval_{currentInterval}$ on $scheduleonSameCluster$
    Compare $(timeInPrevSchedule)$, $(timeinPrevCluster + intraClusterCost)$, $(bestTime + interClusterCost)$;
    if $timeInPrevSchedule$ *is minimum* **then**
      $MCRP = MCRP \cup (multiClusterInterval_{currentInterval}, sameSchedule)$;
    end
    else if $(timeinPrevCluster + intraClusterCost)$ *is minimum* **then**
      $MCRP = MCRP \cup (multiClusterInterval_{currentInterval}, scheduleonSameCluster)$;
    end
    else if $(bestTime + interClusterCost)$ *is minimum* **then**
      $MCRP = MCRP \cup (multiClusterInterval_{currentInterval}, bestSched)$;
    end
  end
  $currrentInterval = currentInterval + 1$;
end

Figure 5.6: Algorithm for Generating Multi Cluster Rescheduling Plan

codes, we determined the total number of application execution phases as 30 for MD, 50 for ChaNGa, 100 for Athena, 26 for LAMMPS and 32 for MPB. The performance model equations for the application phases were obtained for a 48-core AMD Opteron cluster consisting of 12 2-way dual-core AMD Opteron 2218 based 2.64 GHz Sun Fire servers with CentOS 4.3 operating system, 4 GB RAM, 250 GB Hard Drive and connected by Gigabit Ethernet. In our experiments, the available CPU values ranged from 0.1-1.0 where a value of 1.0 indicates an unloaded processor. Thus the available CPU of 0.75 for a processor in our simulation setup represents an AMD processor that is one-fourth loaded.

5.5.2 Results

In this section, we first evaluate the efficiency of the rescheduling plans generated by our rescheduling strategies by comparing the total predicted execution times corresponding to the rescheduling plans and the times for the rescheduling plans generated by a brute force method. We then evaluate our multi cluster rescheduling plan generation algorithm using simulations of dynamic multi-cluster grids.

Evaluation of Single-Cluster Rescheduling Plans

We evaluate the SCRPs generated by our algorithms by conducting simulations of molecular dynamics (MD) application on a cluster of 512 processors. We used the performance model functions of the different phases of MD application obtained on the AMD cluster and conducted 50 simulation experiments. We chose the maximum intra-cluster bandwidth of links in a cluster to be one of 100 Mbps, 1 Gbps, 5 Gbps and 10 Gbps. We randomly varied the available bandwidth of each link to be within 20-80% of the maximum available bandwidth. We also randomly varied the available CPU value between 0.1 and 1.0 for each processor. Available CPU represents the load of the processor.

Table 5.1 shows the average percentage difference between the execution times of the MD application corresponding to rescheduling plans generated by our algorithms and the plans generated by a brute force method for different rescheduling overheads. For

Table 5.1: Comparison of the Algorithms with Brute Force Method for Generation of Rescheduling Plans (MD)

Rescheduling Overhead (secs.)	Incremental	Division	Genetic
0	1.46	2.11	0.03
60	2.12	1.51	0.88
120	3.8	3.39	2.55
180	5.63	5.73	4.26
240	7.36	7.54	6.35
300	8.47	9.7	7.68

this comparison, we considered only 15 phases of the molecular dynamics application execution obtained by considering the execution with 30 phases and merging every two consecutive phases into one single phase. The brute force method evaluates all 2^{14} candidate rescheduling plans and chooses the best plan corresponding to minimum predicted execution time. The table shows that the rescheduling plans generated by our algorithms are highly efficient leading to application execution times that are higher than the execution times corresponding to brute force method by less than 10%. The maximum rescheduling cost of 5 minutes considered in our evaluation corresponds to the overheads in real check-pointing systems[138].

Figure 5.7 compares the average execution times of the rescheduling plans generated by the algorithms with the execution times of the plans generated by the brute force method for Molecular Dynamics simulation application on 512 processors.

Table 5.2 shows the average percentage difference between the execution times of the LAMMPS application corresponding to rescheduling plans generated by our algorithms and the plans generated by a brute force method for different rescheduling overheads. The table shows that the rescheduling plans generated by our algorithms are highly efficient leading to application execution times that are higher than the execution times corresponding to brute force method by less than 12%.

Figure 5.8 compares the average execution times of the rescheduling plans generated by the algorithms with the execution times of the plans generated by the brute force method

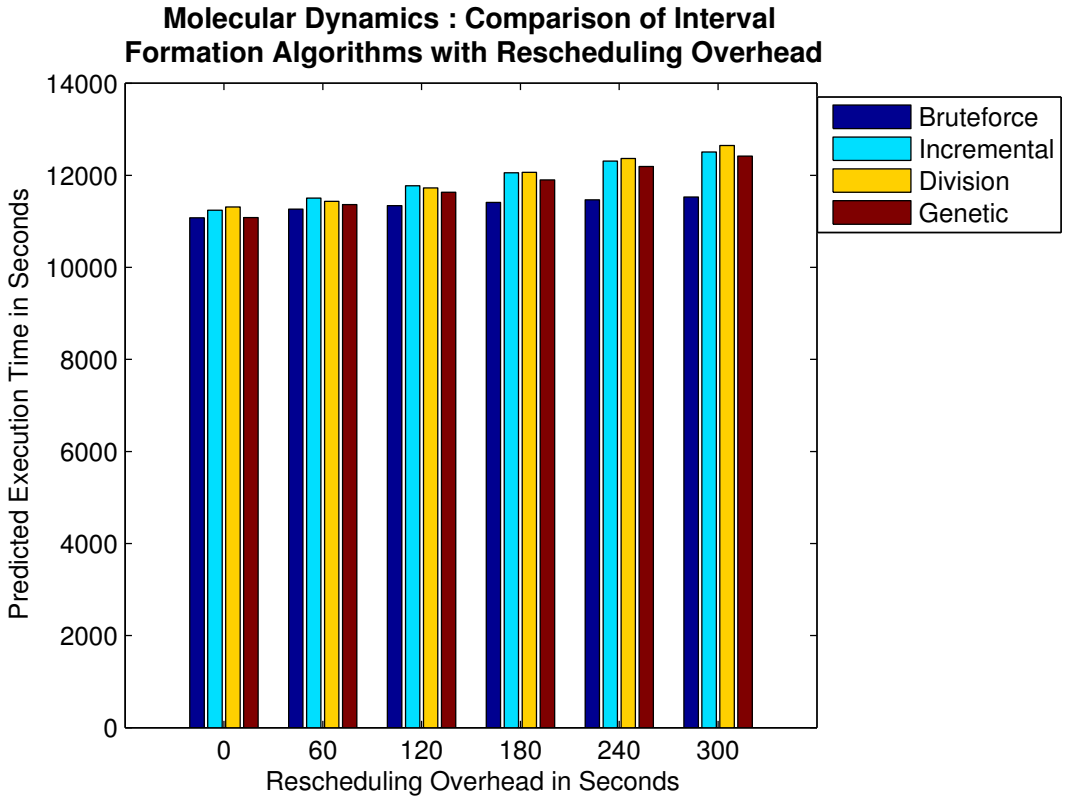


Figure 5.7: Comparison of the Algorithms with Brute Force Method (MD)

Table 5.2: Comparison of the Algorithms with Brute Force Method for Generation of Rescheduling Plans (LAMMPS)

Rescheduling Overhead (secs.)	Incremental	Division	Genetic
0	0.75	1.05	0.37
60	1.93	2.2	1.41
120	4.32	4.49	3.94
180	6.35	6.74	6.03
240	7.87	8.66	7.99
300	10.24	10.75	11.5

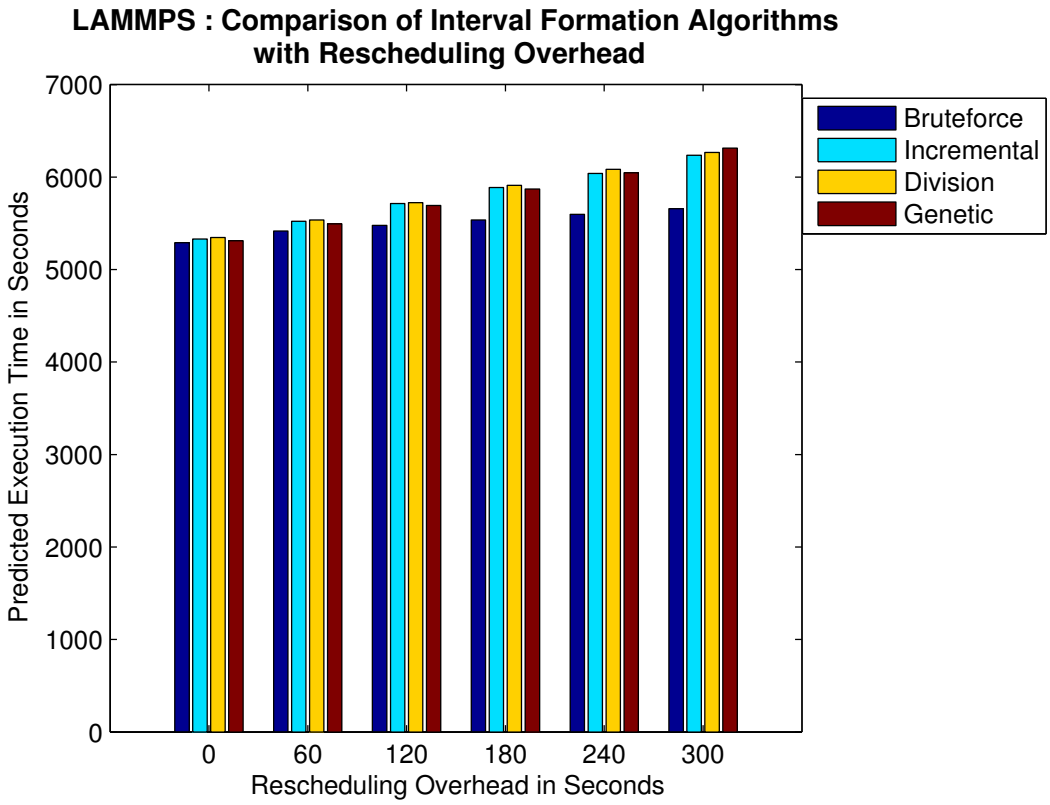


Figure 5.8: Comparison of the Algorithms with Brute Force Method for Generation of Rescheduling Plans (LAMMPS)

for LAMPPS application simulation on 512 processors.

Similar results were obtained for ChaNGa application on 512 processors and are shown in Appendix 8.3.

Rescheduling Plans for Multi-Cluster Grids

In this experiment, we randomly generated 250 multi-cluster grid setups and compared the multi-cluster rescheduling plans (MCRPs) developed using single cluster rescheduling plans (SCRPs) generated by different algorithms for each setup.

For these experiments, we used the performance model functions of the different phases of MD application. We use $U[x, y]$ to denote uniform probability distribution in the interval (x, y). For randomly generating each multi-cluster setup grid setup, we generated

U[5,12] number of clusters and U[32,512] number of processors in each cluster. In order to simulate heterogeneity of processors in different clusters, we used a random *CPU scaling factor* from the set (0.6,0.8,1.0,1.2,1.4) [1] for each cluster, and multiplied the coefficients of computational complexity of the performance model equations with the scaling factor. We chose the maximum intra-cluster bandwidth of links in a cluster to be one of 100 Mbps, 1 Gbps, 5 Gbps and 10 Gbps. We also chose the maximum inter-cluster bandwidth of links connecting two clusters to be one of 0.6 , 0.8, 1, 5, and 10 Mbps. These bandwidths are commonly observed on the links connecting two clusters located at two different sites in many grid systems. We randomly varied the available bandwidth of each link to be within 20-80% of the maximum available bandwidth. We also randomly varied available CPU value between 0.1 and 1.0 for each processor.

To simulate load dynamics in grids during application execution, we varied the available CPU and bandwidth values of the processors and links respectively, after addition of every interval to the multi-cluster rescheduling plan (MCRP) in our MCRP generation algorithm shown in Figure 5.6. Hence the rescheduling plans for subsequent intervals are formed based on the updated resource characteristics.

Table 5.3 summarizes the execution times of the MCRPs generating using SCRPs formed by different algorithms. The table also shows the execution times when the application is executed on a single schedule determined at the beginning of the application for the entire duration of the application. We find that rescheduling in response to changing application and resource dynamics, using the rescheduling plans generated by our algorithms, give much lesser execution times when compared to execution of the applications on a single schedule throughout application execution. Application execution on a single schedule is not able to adapt to high resource and application dynamics. We also find that MCRPs based on genetic algorithm give better average execution times than MCRPs based on incremental and division heuristics. This is because genetic algorithm explores different rescheduling plans at all stages using cross-over and mutation and attempts to achieve

[1] The scaling factors are chosen from a small range since the CPU speeds of modern processors vary by small amounts.

Table 5.3: Multi Cluster Grid Setup

Scheduling / Rescheduling Method	*Arithmetic Mean (hours)*	*Std. Dev. (hours)*
Incremental	6.8	4.72
Division	6.58	5.30
GA	5.97	4.05
Single Schedule	68.77	75.38

globally efficient rescheduling plans while in the incremental algorithm, the intervals are added to the rescheduling plans without the knowledge about the formation of the subsequent intervals. The assumption in the division heuristic about the hierarchical execution behavior of the applications is not applicable to molecular dynamics application. This results in large standard deviation values in the division heuristic although the average execution times in the division heuristic is lesser than in the incremental algorithm.

In order to determine the effect of heterogeneity of different clusters in a multi-cluster grid setup on the performance of the algorithms, we calculated the *weight* of a cluster as a product of number of machines in the cluster and the CPU scaling factor for the cluster. Clusters with larger weights are expected to yield better schedules. We then calculated the ratio of the maximum and minimum weight for a multi-cluster setup. Higher values of this ratio indicates greater heterogeneity between the clusters. We divided the 200 multi-cluster setups into different groups with different ranges of ratios. We eliminated the groups that had less than 5 setups. Figure 5.9 shows the average performance of the algorithms for different groups of ratio values. We find that with increasing ratio of maximum to minimum cluster weight or increasing heterogeneity, the application execution times decrease. This is because with increasing differences in capacities of the clusters, the schedules for most of the phases of application execution will be chosen from the cluster with largest weight or highest capacity. Hence, with increasing heterogeneity, applications incur more intra-cluster rescheduling costs and less inter-cluster rescheduling costs. Since the intra-cluster rescheduling costs are lesser than the inter-cluster rescheduling costs, the execution times of the applications decrease with increasing heterogeneity. We also find that genetic algorithm give better average execution times than incremental

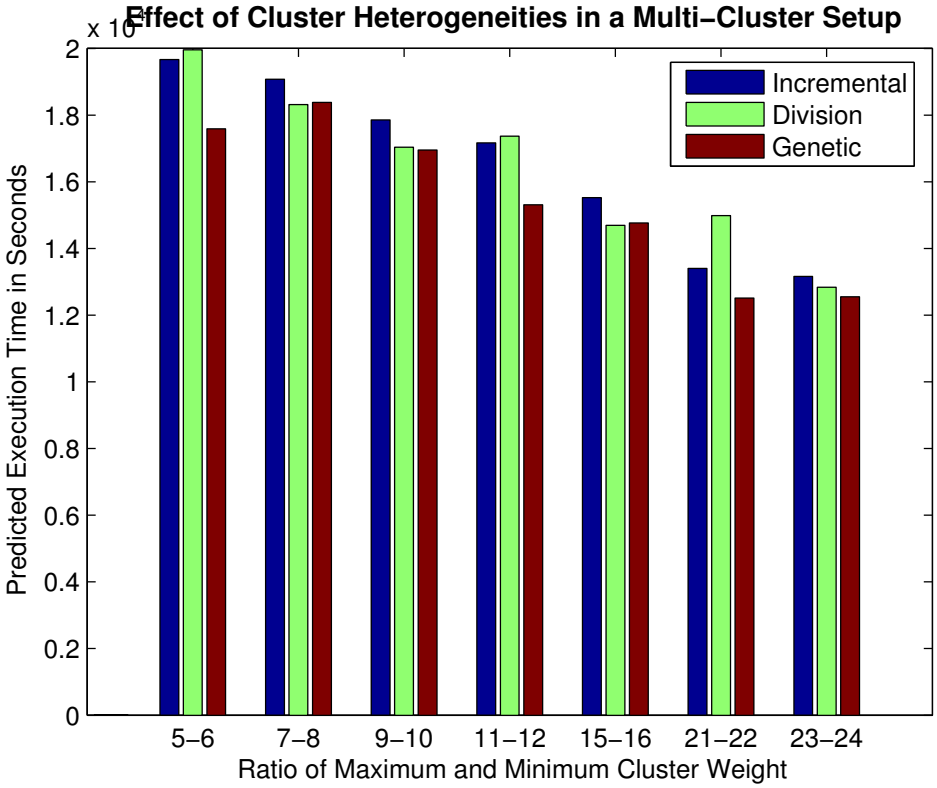


Figure 5.9: Effect of Cluster Heterogeneity in a Multi-Cluster Setup

and division heuristics for all cases.

Figure 5.10 shows the average performance of the algorithms for different average number of machines per cluster in a multi-cluster grid setup. In order to determine the effect of average number of machines per cluster in a multi-cluster grid setup on the performance of the algorithm, we divided the 200 multi-cluster setups into different groups with different ranges of average number of machines per cluster. We neglected the groups which have less than 5 setups. We find that with increasing average number of machines, our algorithms result in decreasing execution times for the applications. This behavior is expected because with increasing average number of machines, our algorithms chose schedules with larger number of processors resulting in reduced execution times. However, when the average number of machines in a cluster is greater than 240, the execution times do not show considerable decrease due to the scalability limits of the application.

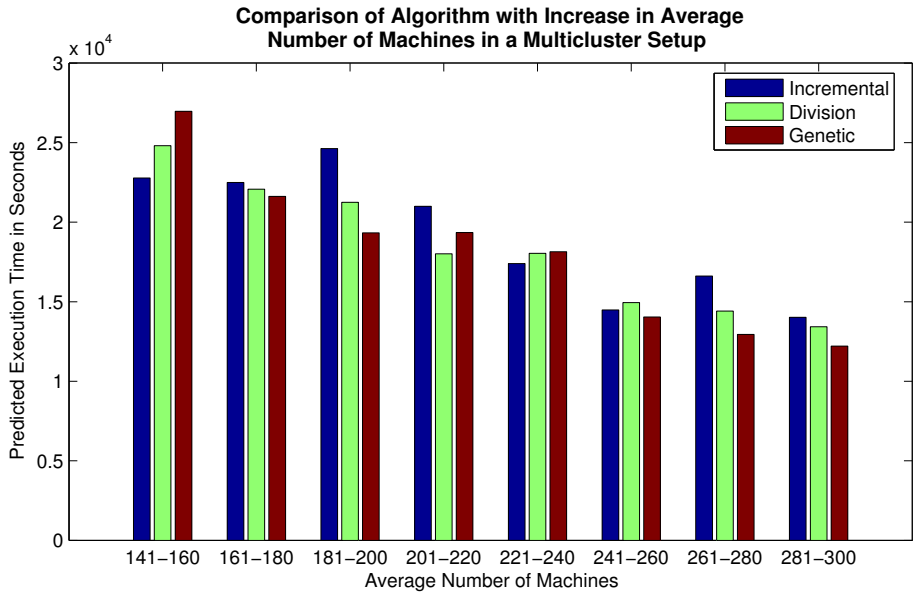


Figure 5.10: Effect of increasing average number of machines per cluster in a multi-cluster grid setup

5.6 Discussion

Our algorithms use practical considerations, namely, intra-cluster and inter-cluster rescheduling overheads, to determine if rescheduling is useful. For our experiments, we use values for these overheads corresponding to a real checkpointing and application migration/rescheduling system, namely, SRS[138]. All our algorithms compare the cost of continuing the application on the same resources and the cost of rescheduling and continuing on different sets of resources to decide if the application needs to be rescheduled between the phases. Using the algorithms that include practical aspects of rescheduling costs and the CPU and bandwidth load values obtained for practical loads, simulations are performed to compare executions with rescheduling and single-schedule executions. These results show that in all cases, rescheduling is useful and leads to huge performance benefits over single-schedule executions.

Although we have shown the results with non-dedicated systems, our rescheduling algorithms are also highly applicable to batch systems. In these batch scheduling systems, even though the processors are dedicated for application execution, the network links are

non-dedicated due to execution of other jobs on other processors that use the common network links and switches. Moreover, for applications with multiple computation and communication phases, the number of processors in a batch system for minimum execution time may differ for different phases. Hence, applications will have to be rescheduled in response to application dynamics.

5.7 Summary

In this chapter, we described strategies for deciding when and where to reschedule executing tightly-coupled multi-phase parallel applications on multi-cluster grids. The rescheduling plans generated by our algorithms were shown to be highly efficient in terms of reducing the total execution times of the applications using large number of simulations of large-scale applications on multi-cluster grids.

Chapter 6

A Grid Computing Framework

6.1 Introduction

In order to make grids highly available environments and to achieve good application performance, practical grid middle-ware, that provides the single system image of the available resources to the executing applications, is essential. In this chapter, we focus on the integration of the components into a grid middle-ware framework with emphasis on practical aspects of the framework related to executions of tightly-coupled applications on multi-cluster grid systems.

As discussed in Chapter 2, a large number of robust grid frameworks exists for supporting deployment and execution of loosely-coupled parallel applications[153, 149, 16]. Loosely coupled applications are amenable to performance benefits with grid executions since the communications between individual tasks are negligible. Scheduling mechanisms for such applications do not have to consider the communication characteristics of the application. Recently, efficient mechanisms have been built for enabling workflow applications on grids[42, 99, 155]. Scheduling and rescheduling strategies that have been developed for allocating the workflow application components on different clusters of a grid cannot be used for tightly-coupled applications due to higher frequency of inter-task communications in tightly-coupled applications. However, not much work

exists for efficiently executing tightly-coupled parallel applications with potential multiple phases of computation and communication characteristics on multi-cluster grids. The primary reason for less work in the area is the possible degradation in performance when heavy communications in such applications occur on the slow-latency inter-cluster links when executed on grids. Hence the usefulness for multi-cluster grids for executing such applications has largely been related to expanding problem sizes[8] and not essentially for improving performance. While we adhere to the general principles of confining the executions of such applications in a cluster at a point of application execution, we claim and show that efficient scheduling and rescheduling mechanisms for execution of different phases of the applications in different clusters can bring performance benefits to the tightly-coupled applications when executed on multi-cluster grids. Existing frameworks[98, 97, 43, 129, 55, 141, 34, 114] for enabling MPI based tightly-coupled parallel applications on grids do not adequately deal with application dynamics for large-scale parallel applications, have large application execution overheads, and are generally not suitable for multi-cluster grids that can consist of batch systems. These techniques are not practical for grid systems where node availability can frequently vary. When the resource availability is small, many entities will be executed on a single node leading to application performance degradation and overloading of system resources. The work by Fernandez et al.[55] shows about 36-88% increase in execution times of some applications when many threads or mapped to a single processor. The ReSHAPE[129] framework is intended for dynamic resizing of homogeneous applications. The framework uses a resizing library implemented for dynamic process management. They support efficient data redistribution for remapping data to different sets of processors. The framework also uses MPI-2 dynamic process spawning for on-the-fly reconfiguration and hence is not applicable for multi-cluster batch systems.

In this work, we present MerITA (**M**iddleware for P**er**formance **I**mprovement of **T**ightly Coupled Parallel **A**pplications on Grids), a system for effective execution of tightly-coupled parallel applications on multi-cluster grids consisting of dedicated or non-dedicated, interactive or batch systems Our work brings together performance model-

ing for automatically determining the characteristics of parallel applications, scheduling strategies that use the performance models for efficient mapping of applications to resources, rescheduling policies for determining the points in application execution when executing applications can be rescheduled to different sets of resources to obtain performance improvement and a check-pointing library for enabling rescheduling. While the individual components of our framework have been explained in our previous chapters, in this chapter we focus on the integration of the components into a grid middle-ware framework with emphasis on practical aspects of the framework related to executions of tightly-coupled applications on multi-cluster grid systems.

Section 6.2 describes overall MerITA architecture. Section 6.3 describes performance modeler component of MerITA. The scheduler component is described in Section 6.4 while rescheduler is described in Section 6.5. Section 6.6 explains integration of components and interaction among the components. Section 6.8 summarizes the chapter.

6.2 MerITA Architecture

Figure 6.1 shows the overall MerITA architecture.

The **metascheduler** component acts as an interface between the user problem submission and the other components of MerITA. The metascheduler orchestrates the application execution and re-execution. The **performance modeler** determines the characteristics of application executions using sample application executions and forms application performance models. The **reschedule planner** forms the overall plan of application execution including the schedules of different phases of application execution. The schedules are obtained by means of interactions with a **scheduler** component that uses a scheduling algorithm and resource characteristics to determine the resources for mapping the application phases. The **application launcher** executes the application on the determined schedule. The **application monitor** component monitors the application execution and forms the profiles of the application execution samples. The application stores its state periodically using a checkpoint storage infrastructure. In case of batch queue systems,

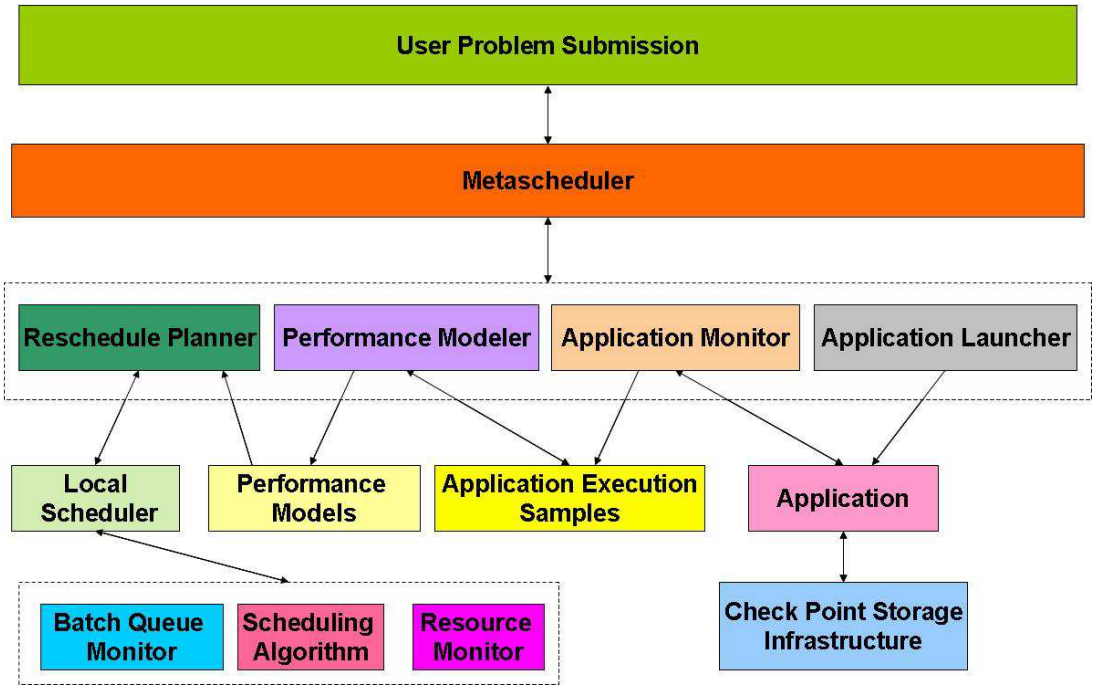


Figure 6.1: MerITA Framework

a batch queue monitor and predictor help predict the queue waiting time of the different schedules.

The following sections describe the individual components in detail.

6.3 Performance Modeler

For effective scheduling and rescheduling of tightly-coupled parallel applications, it is highly essential to determine the computation and communication characteristics and to predict the execution time of the applications on dedicated or non-dedicated, interactive or batch systems. Most of the existing grid frameworks use profiling to obtain application characteristics during execution[97, 148, 82]. Online profiling of application characteristics during application execution can incur large execution overheads. MerITA uses application automatic and comprehensive performance modeling strategies, described in Chapter 3, to predict execution times of tightly-coupled parallel applications.

The **performance modeler** component uses samples of execution for an application

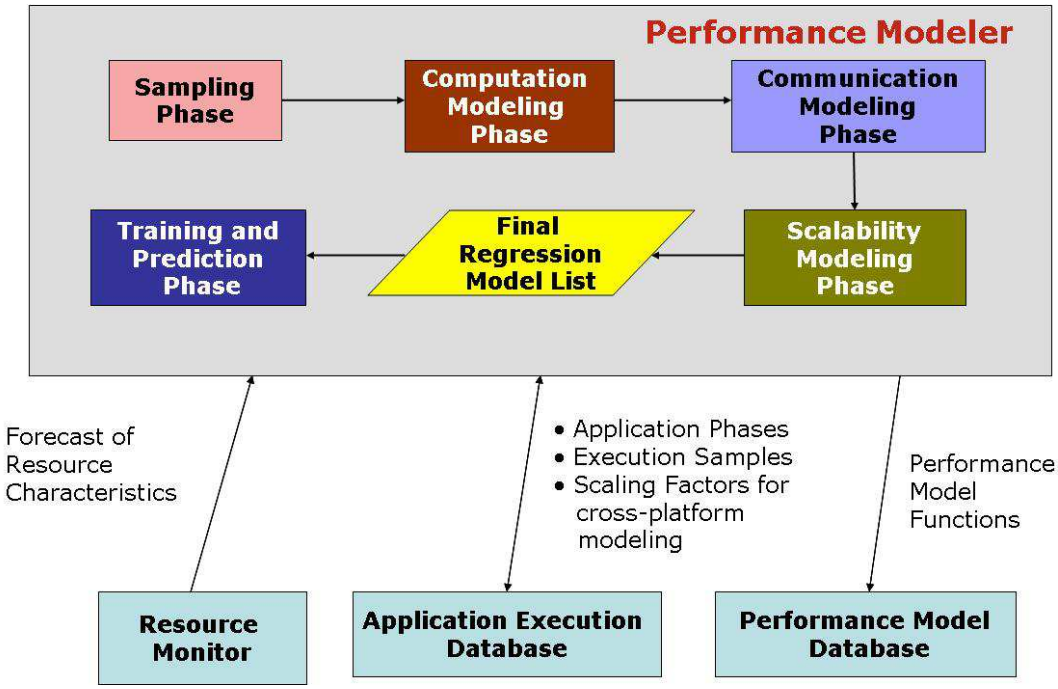


Figure 6.2: MerITA's Performance Modeling Interaction

present in an **application execution database** for a cluster to form a performance model function for the application corresponding to the cluster. An execution sample consists of the problem size, the number of processors used for execution and the resource characteristics that existed during the execution. The performance modeler helps in calculating the time taken for execution of the parallel application using Equation 3.6 discussed in Chapter 3.

By using the modeling procedure discussed in chapter 3, the performance modeler evaluates various candidate functions for f_{comp}, f_{comm}, f_{cpu}, f_{bw}, f_{Pcomp} and f_{Pcomm} using linear regression and choose a combination of functions that give the minimum average percentage prediction error. The performance modeler stores the resulting performance model function for the application in a **performance model database** for the corresponding cluster.

The architecture and interactions of the performance modeler are illustrated in Figure 6.2.

6.3.1 Cross-Platform Performance Modeling

MerITA also uses the cross-platform modeling techniques, described in Chapter 3, whereby the performance model functions and training data obtained for an application on a given parallel cluster called *reference cluster* can be used for predicting execution times of the application on another parallel cluster called *target cluster*. The coefficients of the different functions of Equation 3.6 are scaled to take into account the performance difference between the reference and the target clusters. The computation complexity f_{comp} is scaled by a CPU scaling factor which is determined by obtaining single processor execution times of the application with a moderate problem size on dedicated target and reference processors, and finding the ratio between the two times. The scaling factor for the communication complexity is obtained by conducting dedicated 2-processor experiments with different problem sizes on the reference and target clusters. The scaling factors are obtained by the *performance modeler* from the *application execution database*.

6.3.2 Prediction of Execution Times

To predict the execution time of an application with a given problem size and given number of processors in a cluster, the *performance modeler* uses the performance model function for the application stored in the *performance model database* for the cluster. The prediction of execution time needs values for problem size, number of processors, minAvgAvailCPU and minAvgAvailBW. Problem size and number of processors for execution are obtained from the user. During training the models, the minAvgAvailCPU and minAvgAvailBW values were obtained by observing the system loads during application execution. For prediction of execution time of an application, these values will have to be predicted, since the values represent the system loads that will exist during the period of application execution. Hence, the performance modeler forecasts the values based on the history of load dynamics measured on the system using the forecasting tools from NWS[147, 146]. The performance modeler then uses the problem size and number of processors supplied by the user and the predicted MinAvgAvailCPU and MinAvgAvailBW

to predict the execution time of the application.

6.3.3 Performance Models for Multi-Phase Applications

We discussed modeling multi-phase applications in chapter 3. For predicting the execution times of large-scale parallel applications with multiple phases of computation and communication complexities, the *performance modeler* obtains the major phases of application execution from the *application execution database*. For each of the phases, the performance modeler uses the CPU and network load measurements and execution times within the phase boundaries to derive per-phase performance models as shown in Equation 3.6. The predictions by the per-phase performance models can be used by the performance modeler for analyzing the behavior of individual phases and giving accurate predictions for the entire application.

6.4 Scheduler

When a user submits an application to MerITA, the *metascheduler* component of MerITA interacts with the *local schedulers* on the individual clusters and passes the application problem size to the local clusters. The local scheduler obtains the performance model for the application from the *performance model database* in the cluster and also obtains the predicted CPU and network loads of the cluster resources from NWS. The scheduler then invokes a scheduling algorithm which uses the performance model for the application in the cluster, the problem size and resource characteristics to determine the best schedule in the cluster for application execution. For the scheduling algorithm, the local scheduler of MerITA uses the efficient *Box Elimination (BE)* algorithm, described in Chapter 4, for determining the best schedule. The local scheduler returns the best schedule and the corresponding predicted execution time for the cluster to the metascheduler which in turn compares the predicted execution times from all the clusters to determine the best cluster and the corresponding best schedule for executing the application.

The scheduling algorithm invoked by the local scheduler uses the application performance model for comparing various candidate schedules and choosing the best schedule for application execution. In case of interactive systems, the schedules are compared in terms of execution times predicted by the performance models. In case of batch systems, the submitted applications are placed in a queue before allocated resources for execution. In this case, the schedules are compared in terms of the sums of predicted execution times and the times spent by the applications waiting in the queues. For predicting queue waiting times, MerITA uses **batch queue predictor** which interacts with a **batch queue monitor**. The batch queue monitor maintains a history of queue waiting times for the jobs submitted to the system. The batch queue predictor periodically obtains the history from the batch queue monitor and predicts the queue waiting time for a job with certain number of processors as the average of queue waiting times of the jobs in the history executed with the same number of processors in the batch system. When a scheduler considers a schedule with certain number of processors, it interacts with the batch queue predictor and obtains the prediction corresponds to the closest number of processors maintained by the predictor.

6.5 Rescheduler

Grids involve resource dynamics due to varying availability and performance of resources. When executing long-running multi-phase parallel applications on grids, it is necessary to change the resource set used for execution by the application and adapt the application execution in response to change in resource characteristics[60, 129, 149, 100]. In addition to resource dynamics, long running multi-phase applications exhibit application dynamics due to different computation and communication characteristics in the different phases[141, 80, 97, 148, 82].

There are two primary challenges in building a framework that deals with rescheduling executing parallel applications. The first involves developing policies for deciding when and where to reschedule the applications. These decisions are made in MerITA by a com-

ponent called **reschedule planner**. The second challenge involves developing techniques or strategies for enabling an application to reschedule to different sets of resources.

6.5.1 Reschedule Planner

The **reschedule planner** component of MerITA interacts with scheduler and performance modeler to derive rescheduling plans for executing multi-phase parallel applications on grids. A rescheduling plan consists of potential points in application execution for rescheduling and schedules of resources for application execution between two consecutive rescheduling points. The rescheduling plan is built for a specific set of resource characteristics and considers change in application behavior between different phases. The plan can be updated periodically during application execution thus ensuring adaptation to both resource and application dynamics. The reschedule planner is present in every cluster of multi-cluster grids and derives rescheduling plans for each cluster. The metascheduler component uses rescheduling plans derived on different clusters to form a single coherent rescheduling plan for application execution on a grid consisting of multiple clusters.

The *reschedule planner* discussed in chapter 5, first obtains the performance model functions of the different phases of the application from the *performance model database*. It then uses the application problem size, resource characteristics and a rescheduling cost for the cluster to derive an efficient rescheduling plan for the application execution on the cluster such that the total predicted times for execution and rescheduling is minimum. The rescheduling plan consists of disjoint set of intervals consisting of phases and the schedules for executing the intervals.The incremental algorithm, discussed in Chapter 5, that constructs a rescheduling plan by adding intervals to the plan in increasing order of execution time, is used by the reschedule planner for forming single cluster rescheduling plans (SCRPs).

The metascheduler uses the single cluster rescheduling plans (SCRPs) generated for the individual clusters by the reschedule planners for the clusters to form a coherent

rescheduling plan involving multiple clusters of a grid. The metascheduler uses the algorithm described in Chapter 5 for generating a multi-cluster rescheduling plan (MCRP) and uses the MCRP to determine the schedule for executing the next phase of application execution. The metascheduler interacts with the reschedule planners of the different clusters to obtain rescheduling plans for the clusters. It then finds the intervals from the plans that contain the phase and the corresponding schedules. The metascheduler then determines the best cluster and the best schedule in the cluster and the corresponding predicted time for execution of the interval containing the phase. The metascheduler also determines the predicted execution times for the interval on the schedule determined for executing the previous phases, and on the best schedule for the interval on the cluster used for executing the previous phases. It then uses these predicted execution times, the cost of rescheduling to the same cluster and the cost of rescheduling to a different cluster and performs a three-way comparison between the three predicted times to determine if the next phase has to be executed on the same schedule used for execution of previous phases, on a different schedule but on the same cluster used for execution of previous phases or on the best schedule across all the clusters.

6.5.2 Enabling Application Rescheduling

MerITA uses a user-level semi-transparent check-pointing library called SRS[138] (**S**top **R**estart **S**oftware) for enabling executing applications to reschedule to different number of processors and different clusters. Applications when instrumented with calls to SRS, attain the ability to be stopped at a particular point in execution and to be restarted and continued later on a different configuration of processors. The SRS library is implemented on top of MPI at the application layer and migration is achieved by stopping the entire application and restarting the application over a new configuration of machines. Although the method of rescheduling in SRS, by stopping and restarting executing applications, can incur more overhead than process migration techniques [43, 97, 129], SRS is highly applicable for batch systems and for reconfigurations across different clusters where most of the nodes in a cluster have private IPs.

SRS library uses Internet Backplane Protocol(IBP)[112] for storage of the checkpoint data. The IBP storage servers are started on some machines in each cluster of the grid. The specific storage servers to be used for check-pointing can be specified through a check-pointing configuration file before application execution. Typically storage servers in the cluster where the application is executed is specified for checkpoint storage.The SRS library consists of three main functions, namely, SRS_Check_Stop(), SRS_Register() and SRS_Read(). The application calls SRS_Check_Stop() at the end of different application phases to check if it has to stop execution. It calls SRS_Register() to register the variables along with their data redistributions for check-pointing. When the application stops execution, the SRS_Check_Stop() function checkpoints only those variables that were registered through SRS_Register(). When the application resumes execution on a different configuration of processors, it calls SRS_Read() for reading the check-pointed variables. By knowing the number of processors and the data distribution used for the variables, SRS_Read() automatically performs the appropriate data redistribution.

An external daemon called Runtime Support System (RSS) acts as an interface between the application instrumented with SRS and the **application launcher** component of MerITA. RSS exists for the entire duration of the application and spans across multiple migrations of the application. Before the actual parallel application is started on a cluster, the RSS is launched on one of the machines in the cluster by the application launcher. The actual application through the SRS library interacts with RSS to perform some initialization, to check if the application needs to be stopped during SRS_Check_Stop() and to store and retrieve pointers to the check-pointed data.

When the metascheduler generates a multi cluster rescheduling plan (MCRP) for the application execution in the grid, it stores the plan in a file and stages the file to the machine where the RSS is executing through the application launcher. The RSS reads the file and records the next phase when the application has to be stopped. This phase is called *stop phase*. The RSS also maintains a running counter of the number of phases executed in the application. When the application contacts the RSS through SRS_Check_Stop() at the end of every phase of execution, the counter is incremented. When the counter equals

the phase number associated with the stop phase, the RSS signals the application to stop through SRS_Check_Stop().

When the metascheduler wants to migrate the application from the current cluster to a different cluster, it first stops the current executing application through the plan file and RSS mechanisms mentioned above. It then contacts the RSS through the application launcher to move the data check-pointed in the storage servers of the current cluster to the storage servers of the other cluster. It then signals the RSS to store its current data to a file called *rss-file*. The metascheduler terminates the RSS in the current cluster, moves the rss-file to the other cluster, and starts the RSS in the other cluster. The RSS restores its previous state by reading its data from the previous cluster and continues execution. The metascheduler then starts the application on the new cluster and continues execution. Stopping and continuing the application based on rescheduling plans for multi-phase applications, check-pointing application data to storage servers different from the computational nodes, and the ability to checkpoint and continue RSS on a different resource were the extensions added to the SRS library for the MerITA framework.

6.6 MerITA Interactions - The Big Picture

Figure 6.3 shows the interactions between different MerITA components.

A user before submitting a parallel application to MerITA, instruments his application with calls to SRS check-pointing library. He identifies major phases of execution in his application and instruments SRS_Check_Stop() in the application at the end of the phases. In MerITA, the phases are marked at compile time. We use the work by Shen et al.[122, 121] that uses a technique called *active profiling* for identifying and marking the execution phases at compile time. Active profiling uses controlled inputs and analyze basic block traces executed to identify candidate phase markers or phase boundaries, real inputs to eliminate false positives, and detailed analysis for identifying inner phase markers. For some parallel applications not amenable to active profiling, we use manual analysis of high-level structure of the source code and consider long-running subroutines

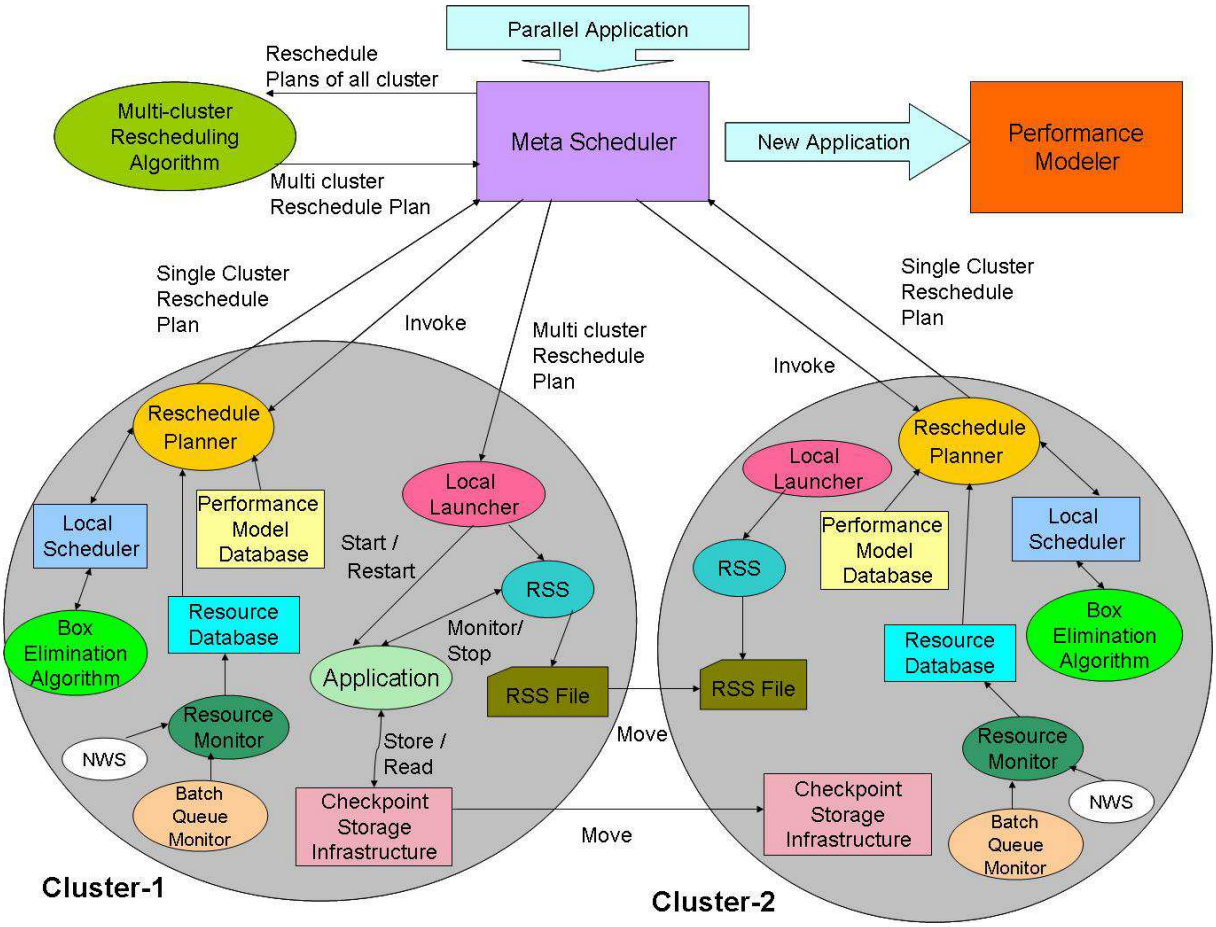


Figure 6.3: MerITA Interactions

and loop-nests as candidate phases[46]. We are planning to develop techniques for automatic identification of phases and instrumentation of phase marker and SRS calls in the application's executables.

The user then submits the application with a given problem size to the grid through our MerITA framework user interface. This user interface contains simple function calls similar to GrADSolve GridRPC system[139]. The interface sends the problem parameters to a job specific MerITA's metascheduler or job manager. The metascheduler interacts with the performance modeler database of each cluster in the grid and tries to obtain the performance model for the function.

When the application is submitted for the first time to the grid, the performance model databases of the clusters will not have the performance model for the application. In this case, the metascheduler chooses an arbitrary set of machines from a cluster for application execution. It also chooses a cluster called *base cluster* and executes the same application with the same problem size on 1, 2, 4 and 8 processors of the base cluster. The metascheduler also performs dedicated 1-processor and 2-processor executions of the application on all the clusters. Some parallel applications may have constraints regarding the minimum number of processors that they need to execute. In these cases, the metascheduler, instead of executing the application on 1, 2, 4 and 8 processors, executes the application on the minimum number of processors and few processor configurations greater than the minimum number of processors.

Corresponding to an execution, the metascheduler obtains the resource characteristics that existed during application execution by interacting with NWS. These resource characteristics include average CPU and network loads of the processors and links used by the application. The metascheduler also observes the total application execution time. The metascheduler records the application name, problem size, number of processors used for application execution, resource characteristics and execution time in the application execution database of the base cluster. Every cluster has its own application execution database. When the database contains sufficient number of executions[1] corresponding to

[1] we use 30 as base number of executions

different problem sizes and number of processors, the metascheduler invokes the performance modeler component of the cluster with the database.

The performance modeler, using our performance modeling techniques, builds performance model functions for the different application phases for the base cluster and stores the functions in the performance model database. The performance model functions can predict execution times of the application phases for a given problem size and resource characteristics on the cluster. The performance model functions of the base cluster are then ported and stored in the performance model databases of the other clusters by scaling the coefficients of the functions based on the performance difference corresponding to the dedicated 1-processor and 2-processor applications on the base cluster and the other clusters.

When a user submits an application to MerITA and the metascheduler finds performance model functions of the application phases in the performance model databases of the clusters, it invokes the schedule/reschedule planners of the individual clusters with the problem size. A schedule/reschedule planner of a cluster obtains the performance models of the phases from the performance model database, and resource characteristics from NWS, and constructs a rescheduling plan for application execution consisting of rescheduling points in execution at which the application has to be rescheduled and the schedules of machines for execution between the phases. The planner interacts with the local scheduler to obtain best schedules of machines in the cluster for execution of various phases.

The metascheduler obtains the rescheduling plans from the schedule/reschedule planners of all clusters and constructs a single coherent rescheduling plan (MCRP) for rescheduling the application across different clusters. The metascheduler then obtains the cluster containing the schedule of machines for execution of the first phase of the application from MCRP. It invokes the application launcher of this cluster passing the rescheduling plan, MCRP, to the launcher. The launcher stores the MCRP to a file and starts the RSS service of the SRS migration framework in one of the machine in the cluster. The RSS periodically reads the MCRP from the file to determine the point at which the application has to be stopped. The launcher then executes the application specifying the storage

servers in the cluster for storing application checkpoints. The application at the end of every phase of execution interacts with RSS through the SRS library to determine if it has to stop execution. The RSS maintains a running counter of the number of executed phases.

The metascheduler periodically interacts with the reschedule planners, obtains the single-cluster plans, forms the multi-cluster reschedule plan (MCRP) and sends the MCRP to the launcher of the cluster on which the application is executing. The updated MCRP is periodically read by the RSS service. When the number of phases in the application execution reaches the rescheduling point, RSS signals the application to stop through the SRS library. After the application stops, the launcher of the cluster interacts with the RSS to move the check-pointed data to the storage servers of a different cluster if the phase after the rescheduling point is to be executed in a different cluster. It then stops the RSS in the current cluster, sends the RSS checkpoint file, *rss-file*, to the launcher of the new cluster. The metascheduler forms a MCRP for the remaining phases of execution and interacts with the launcher of the new cluster with the new MCRP to launch and continue the application. The launcher starts and continues the RSS service from the RSS check-point file and *rss-file*. Then it starts and continues the application. The metascheduler interactions with the components of the clusters continue till the application completes its execution.

6.7 Discussion

6.7.1 Interaction with local schedulers

At the time of the writing, the initial prototype of MerITA was deployed on two clusters in Indian Institute of Science and robustness tests were conducted.

At the time of this writing, the MerITA infrastructure has been tested thoroughly and successfully on 3 clusters, a 32-processor non-dedicated AMD Opteron cluster in Indian Institute of Science, a 16-processor dedicated IBM Loadleveler batch system in In-

dian Institute of Science, and a 48-processor dedicated IBM Loadleveler batch system in Centre for Development of Advanced Computing, Bangalore, India. We are currently conducting experiments to compare the execution times when applications are executed with rescheduling in our MerITA framework, when the applications are executed without rescheduling and on the initial schedule chosen by the MerITA framework and when the applications are executed without the MerITA framework on all sets of processors. These experiments will also be useful to perform detailed performance studies of the individual components.

The integration of MerITA with the local schedulers of the batch systems in IISc and CDAC involved different interesting challenges and practical considerations. One of the challenges is to predict the queue waiting time for the job. The batch queue monitor of MerITA interacts with loadleveler to obtain the queue waiting time for each job, submitted to the cluster. Since the chosen clusters are using IBM Loadleveler, batch queue monitor uses *llq* utility to monitor the jobs for calculating the waiting time. Another challenge is submission of the job to the cluster. The local launcher of MerITA generates the submission script on the fly and submits the job through *llsubmit* utility of loadleveler. Another difficulty in batch system is to know the status of the submitted job. As the local launcher has to pass control to the metascheduler after job completes, the local launcher invokes or spawns a process after job submission. The spawned process monitors the submitted job using *llq* utility.

6.7.2 Standard Grid Mechanisms

As mentioned earlier, there is no existing Grid framework that can be used for effective execution of tightly-coupled parallel applications with robust performance modeling, scheduling and rescheduling mechanisms. However, our MerITA framework uses existing solutions for information management and checkpointing. MerITA uses the standard NWS[147, 146] system for obtaining information about loads on the resources. It also uses the SRS checkpointing system[138] for rescheduling and migration.

MerITA can also be integrated with other standard Grid mechanisms for low-level functions, namely, communications, file transfers, monitoring and security. It is easy to upgrade the infrastructure to use the grid standard mechanisms of Globus toolkit[56] namely Nexus[58] for communications, Ganglia Cluster Toolkit[63] for monitoring, webMDS for information management, GridFTP[7] for data access and movement, and GSI[145] for security. For communications between metascheduler and reschedule planers, Nexus-based communication mechanisms can be used. Grid FTP can be used for transfer of checkpoint data files and schedule related files from one cluster to another when the application is rescheduled by the rescheduler component of MerITA. Ganglia Cluster Toolkit, a toolkit that specializes in collecting monitoring data from clusters can be used to monitor the clusters. For monitoring information regarding performance models, training data, load information and waiting time, we can use webMDS. For authentication between remote machines, GSI-based security mechanisms can be used. Currently, we are using Unix TCP/IP socket functions to communicate between metascheduler and reschedule planers, UNIX *scp* utility for transferring the data files, NWS for cluster monitoring, flat files for information management and UNIX password-based security mechanism for security.

6.8 Summary

In this chapter, we described our MerITA (**M**iddleware for P**er**formance **I**mprovement of **T**ightly Coupled Parallel **A**pplications on Grids) architecture. Our framework integrates the performance modeler, scheduler, rescheduler with check-pointing and resource monitor infrastructures. The framework allows the application to adapt to grid dynamics to achieve better performance by using efficient scheduling and rescheduling strategies.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we had developed performance modeling techniques for predicting execution times of tightly-coupled parallel applications for the purpose of scheduling the applications on the grid resources. The challenges involved were building automatic procedures for generating computation and communication complexities for tightly-coupled parallel applications on non-dedicated clusters, developing the models with small overheads and porting the models developed on a cluster to other clusters. Our performance modeling strategies are adaptive to grid dynamics since we use different model functions at different times for predicting execution times based on changing resource loads. We have also developed cross-platform modeling techniques for porting the results of performance modeling developed on one or cluster to other clusters in the grid. Our modeling overheads are smaller and more deterministic when compared to other models. Our performance modeling strategies gave less than 30% average percentage prediction errors in all cases, which is reasonable for non-dedicated systems. We also found that scheduling based on the predictions by our strategies will result in perfect scheduling in many cases and can result in maximum loss in efficiency of the scheduler by only 11%. We found that developing and using single performance model function gave highly inaccu-

rate predictions for applications that exhibit highly non-uniform behavior during executions and where the computation and communication characteristics widely vary between the phases. Using cumulative performance models for multi-phase parallel applications resulted in less than 25% prediction errors.

We have developed a scheduling algorithm called Box Elimination that uses performance model of a tightly-coupled parallel application to schedule the application on a single cluster or a multi-cluster grid. By treating the search space as a set of performance model parameters and eliminating regions containing poor solutions, our algorithm is able to generate efficient schedules with minimum execution times for the application. By large number of simulation experiments, we showed that our algorithm generates schedules with up to 80% less execution times than the other popular algorithms. We also showed that performance predictions errors have the least effect on the quality of the schedules generated by our algorithm.

We also devised strategies for deciding when and where to reschedule executing tightly-coupled multi-phase parallel applications on multi-cluster grids. The rescheduling plans generated by our algorithms were shown to be highly efficient in terms of reducing the total execution times of the applications using large number of simulations of large-scale applications on multi-cluster grids. The execution times due to the rescheduling plans generated by our algorithms are higher than the execution times for the plans generated by brute-force methods by less than 10%.

Finally, we built MerITA (**M**iddleware for P**er**formance **I**mprovement of **T**ightly Coupled Parallel **A**pplications on Grids), a system for effective execution of tightly-coupled parallel applications on multi-cluster grids consisting of dedicated or non-dedicated, interactive or batch systems. The framework allows the application to adapt to grid dynamics to achieve better performance by using efficient scheduling and rescheduling strategies.

7.2 Future Work

The thesis primarily considers computation and communication as dominant characteristics of parallel applications. Future work can explore modeling other important characteristics including I/O behavior, memory access pattern, cache effects, etc. and building corresponding scheduling strategies that utilize these parameters to form efficient schedules. Future work can also model power consumption since this parameter has gained high importance in recent years due to its relevance in data centers.

We also plan to devise job scheduling algorithms to schedule a set of parallel applications on multiple clusters for simultaneous executions with the objective of minimizing the average response times of the applications. In the rescheduling strategies developed in the thesis, the check-pointing library functions are manually instrumented in the applications to make the applications migratable. We plan to develop techniques for automatic identification of phases and instrumentation of phase markers and SRS calls in the parallel application's executables.

Chapter 8

Appendix

8.1 More Results on Performance Modeling

Figure 8.1 plots the percentage prediction errors for different problem sizes and number of processors for the ScaLAPACK Eigen Value Problem on the Intel Cluster with Grads Loading. We find that the percentage prediction errors are less than 30% for 62% of the predictions and less than 40% for 80% of the predictions. The results demonstrate that our modeling strategies give good predictions even for loading conditions that exist on one of the current grid systems.

Figures 8.2 plots the percentage prediction error values for different experiments in the order the experiments were conducted for the ScaLAPACK Eigen Value Problem on the Intel Cluster with Grads Loading. As can be seen, using a mixture of good models and choosing different models at different points of time give smaller prediction errors than using a single model for predictions on non-dedicated environments.

Figures 8.3-8.4 show the percentage prediction errors corresponding to experiments on the 16-processor AMD cluster with random loading conditions for the Eigen value problem. We find that the percentage prediction errors are less than 30% for 73% of the predictions and less than 40% for 90% of the predictions. The low percentage prediction errors show the potential of our models in predicting execution times for higher number

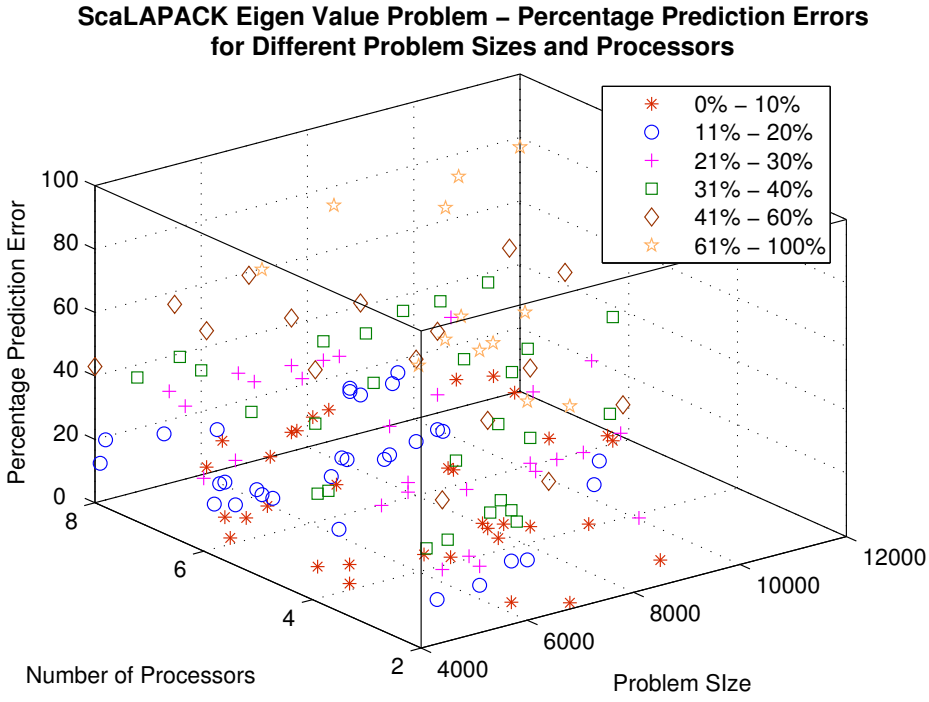


Figure 8.1: Percentage Prediction Errors for ScaLAPACK Eigen Value Problem on the Intel Cluster with Grads Loading

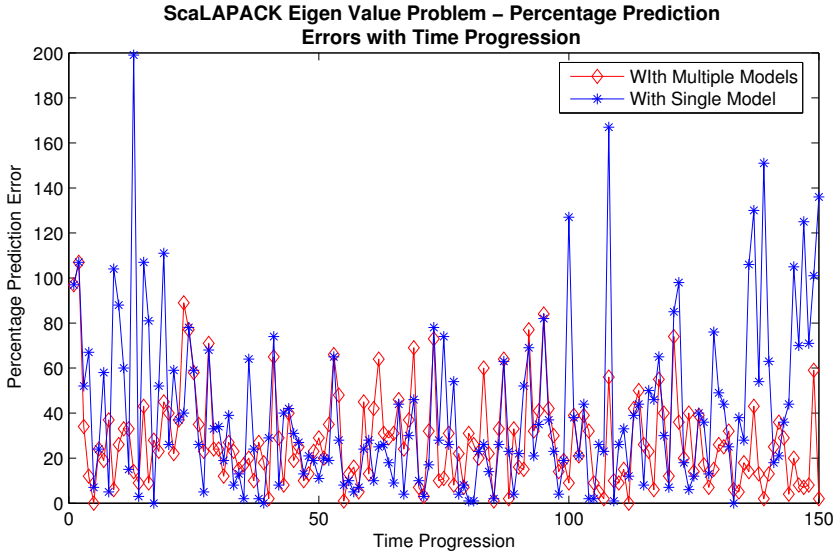


Figure 8.2: Percentage Prediction Errors at Different Times for ScaLAPACK Eigen Value Problem on the Intel Cluster with Grads Loading

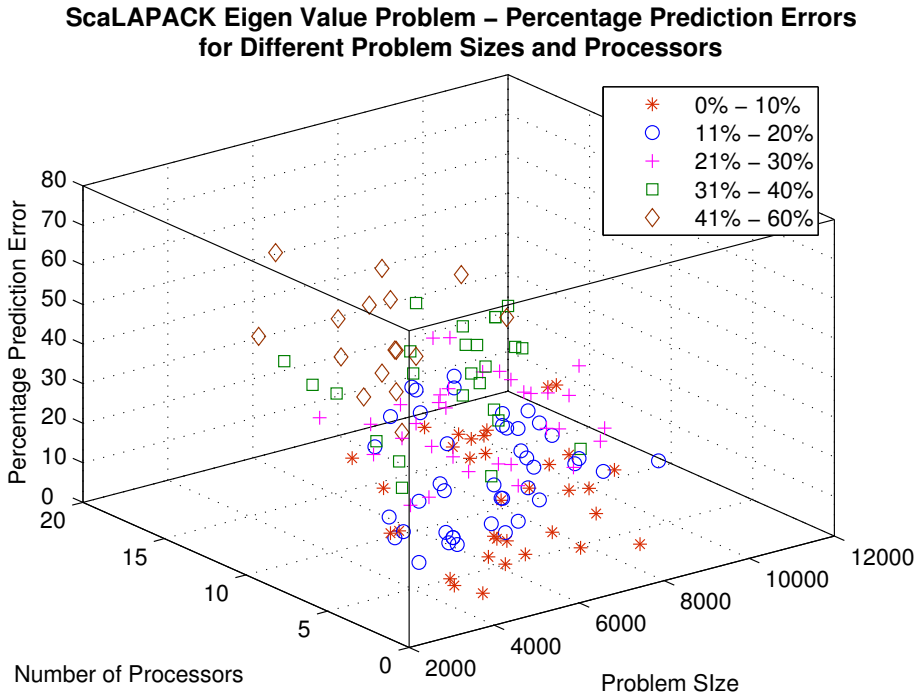


Figure 8.3: Percentage Prediction Errors for ScaLAPACK Eigen Value Problem on the AMD Cluster with Random Loading

of processors than those used during the training.

Figure 8.5 plots the percentage prediction errors for different problem sizes and number of processors for the ScaLAPACK Eigen Value Problem on the AMD Cluster with Grads Loading. We find that the percentage prediction errors are less than 30% for 87% of the predictions and less than 40% for 97% of the predictions.

Figure 8.6 plots the percentage prediction errors for different problem sizes and number of processors for FFT application on Intel cluster with random loading. We find that the percentage prediction errors are less than 30% for 68% of the predictions and less than 40% for 81% of the predictions.

Figures 8.7 plots the percentage prediction error values for different experiments in the order the experiments were conducted for FFT application on Intel cluster with random loading. For the FFT applications, we found that the problem size had to be increased with the increasing number of processors to avoid executing problems with with very

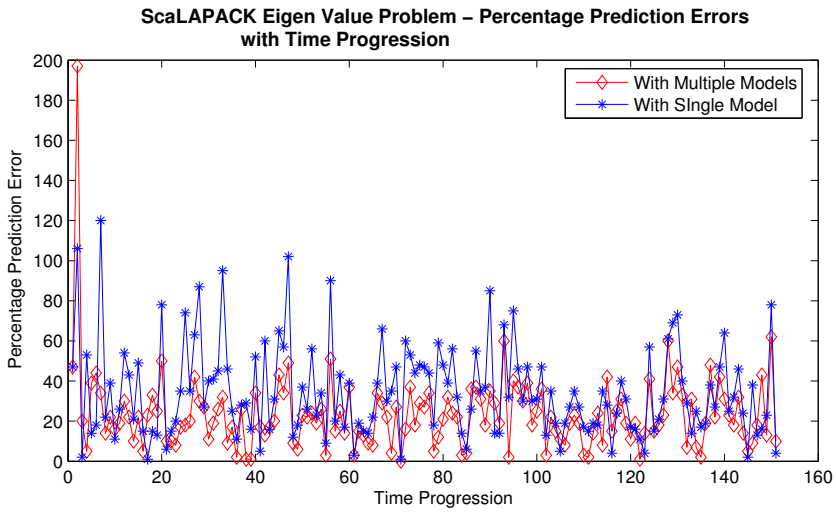


Figure 8.4: Percentage Prediction Errors at Different Times for ScaLAPACK Eigen Value Problem on the AMD Cluster with Random Loading

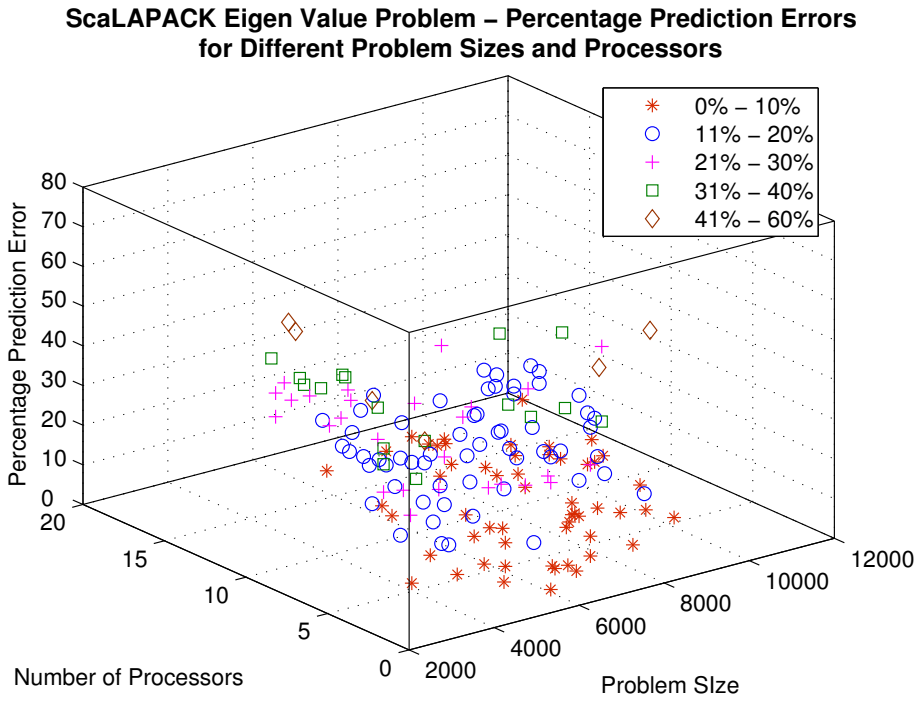


Figure 8.5: Percentage Prediction Errors for ScaLAPACK Eigen Value Problem on the AMD Cluster with GrADS Loading

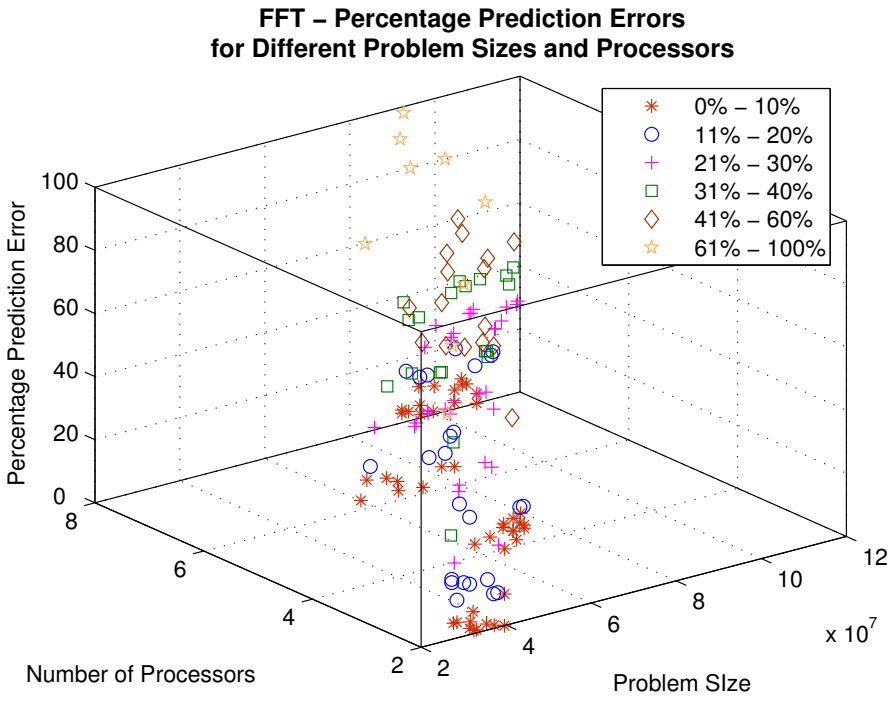


Figure 8.6: Percentage Prediction Errors for FFT on the Intel Cluster with Random Loading

small (less than 20 seconds) execution times.

Figures 8.8 and 8.9 show the results for CG application on Intel cluster with random loading. We find that the percentage prediction errors are less than 30% for 75% of the predictions and less than 40% for 86% of the predictions.

Figures 8.10 and 8.11 show the percentage prediction error and Table 8.1 shows the usefulness of prediction methodology for scheduling a MD application on Intel cluster with random loading. We find that for molecular dynamics applications, our performance models were able to given less than 30% percentage prediction errors in 85% of the cases and less than 40% for 94% of the predictions. We also find from Table 8.1 that except for 2 problem size groups, using our performance modeling strategies to schedule molecular dynamics application on the resources will give rise to perfect scheduling.

Figures 8.12 and 8.13 show the results for MD application on the 16-processor AMD cluster with random loading. We find that the percentage prediction errors are less than 30% for 67% of the predictions and less than 40% for 81% of the predictions.

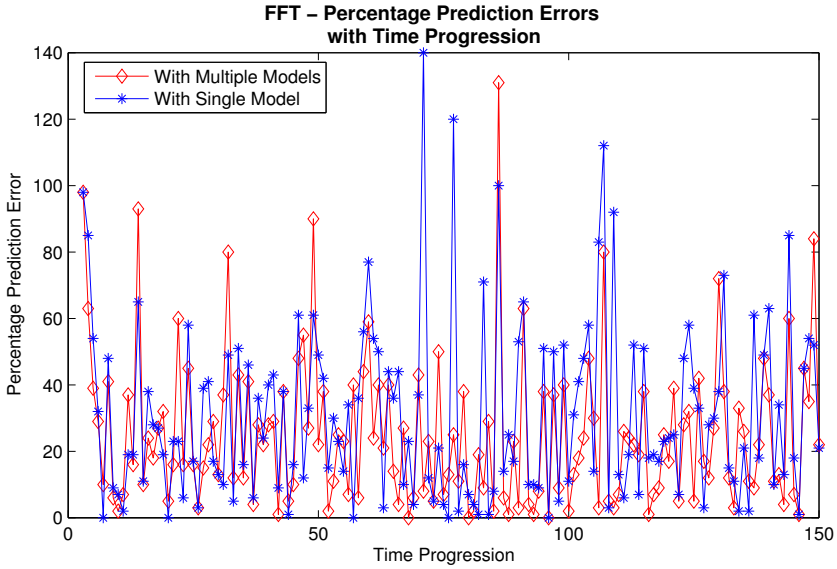


Figure 8.7: Percentage Prediction Errors at Different Times for FFT on the Intel Cluster with Random Loading

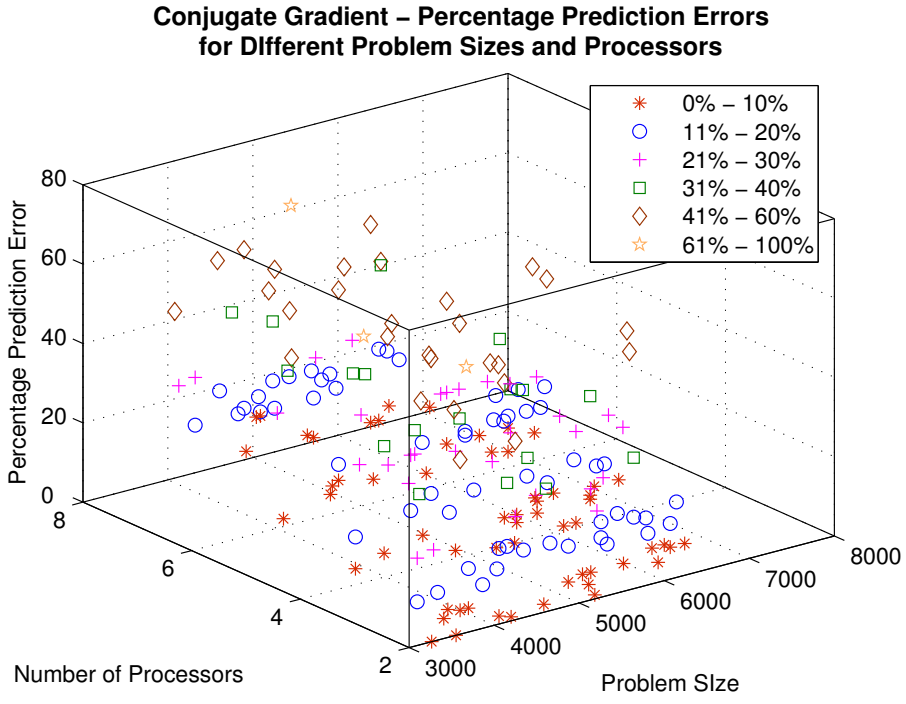


Figure 8.8: Percentage Prediction Errors for CG on the Intel Cluster with Random Loading

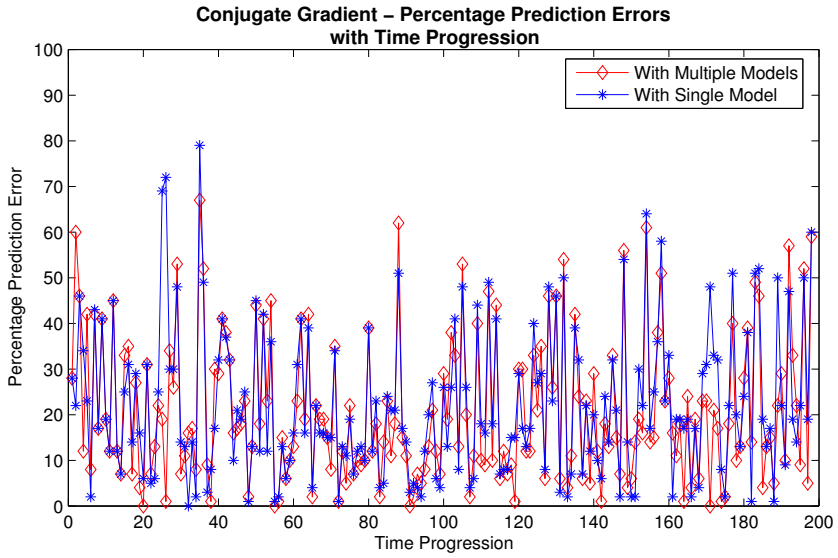


Figure 8.9: Percentage Prediction Errors at Different Times for CG on the Intel Cluster with Random Loading

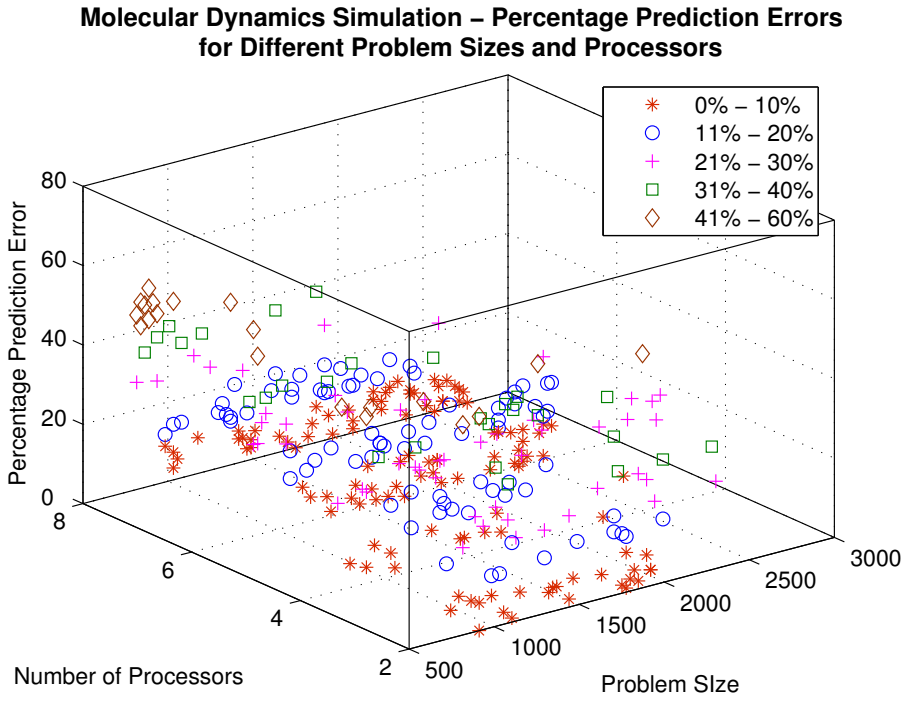


Figure 8.10: Percentage Prediction Errors for MD on the Intel Cluster with Random Loading

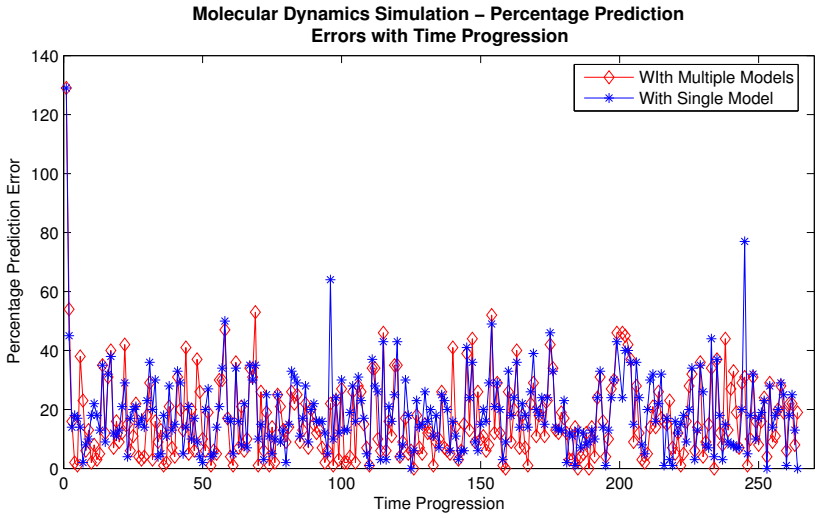


Figure 8.11: Percentage Prediction Errors at Different Times for MD on the Intel Cluster with Random Loading

Table 8.1: Usefulness of Predictions for Scheduling of MD on Intel Cluster with Random Loading

Problem Size Group	Problem Size (N) and Processors (P) for Minimum Predicted Execution Time: (1)	Problem Size (N) and Processors (P) for Minimum Actual Execution Time: (2)	Actual Execution Time for (1): (3)	Actual Execution Time for (2): (4)	Percentage Increase in Execution Time due to Prediction ((3-4)/4)
864-936	864,8	864,8	428.92	428.92	0.00%
1032-1080	1032,8	1032,8	589.06	589.06	0.00%
1296-1344	1296,8	1296,8	825.17	825.17	0.00%
1440-1488	1440,8	1440,8	927.55	927.55	0.00%
1608-1656	1608,8	1608,8	1166.04	1166.04	0.00%
1704-1776	1704,8	1704,8	1292.47	1292.47	0.00%
1896-1944	1944,8	1944,8	1650.34	1650.34	0.00%
2256-2304	2280,8	2304,8	2637.62	2439.54	8.00%
2400-2448	2400,8	2448,8	2896.47	2757.99	2.00%
2568-1616	2592,8	2592,8	3068.87	3068.87	0.00%

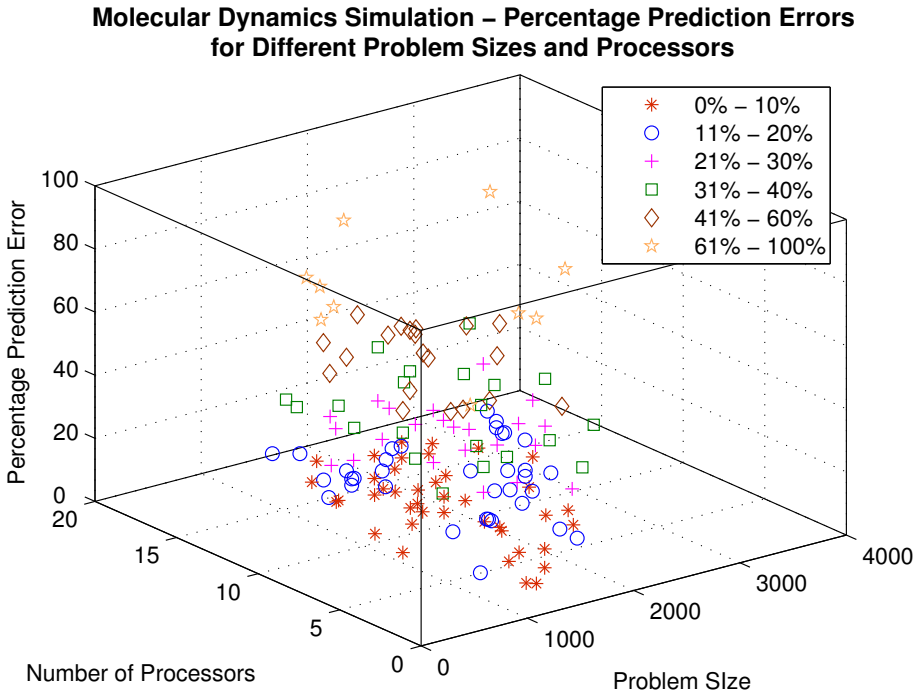


Figure 8.12: Percentage Prediction Errors for MD on the AMD Cluster with Random Loading

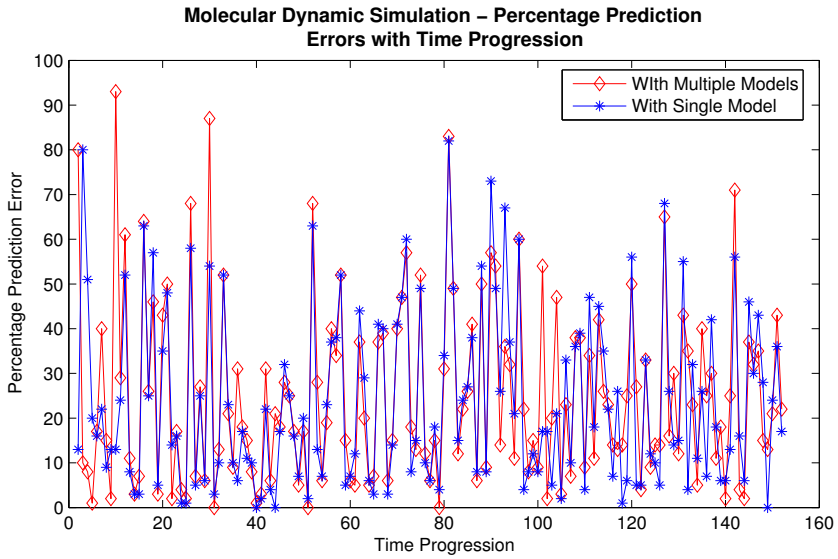


Figure 8.13: Percentage Prediction Errors at Different Times for MD on the AMD Cluster with Random Loading

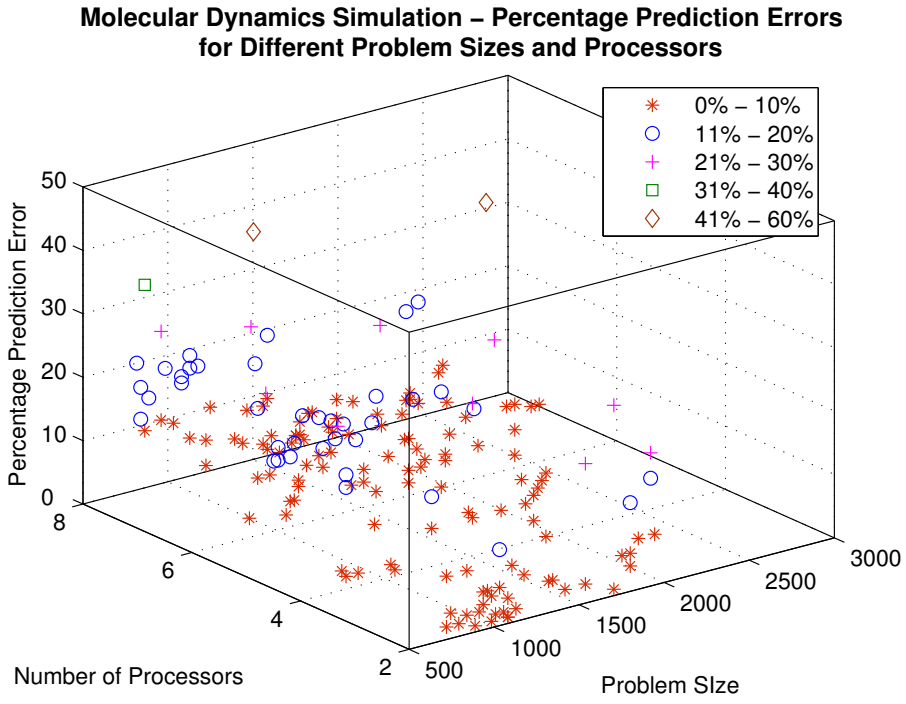


Figure 8.14: Percentage Prediction Errors for MD on the Intel Cluster with GrADS Loading

Figure 8.14 shows the results for MD application on the Intel cluster with GrADS loading. We find that the percentage prediction errors are less than 30% for 98% of the predictions and less than 40% for 99% of the predictions.

Figure 8.15 shows the results for MD application on the AMD cluster with GrADS loading. We find that the percentage prediction errors are less than 30% for 90.34% of the predictions and less than 40% for 94.31% of the predictions.

Figures 8.16 and 8.17 show the results for Poisson solver application on Intel cluster with random loading. We find that the percentage prediction errors are less than 30% for 62% of the predictions and less than 40% for 73% of the predictions.

Figures 8.18 and 8.19 show the results for Poisson solver application on the 24-processor Woodcrest cluster with random loading. We find that the percentage prediction errors are less than 30% for 49% of the predictions and less than 40% for 77% of the predictions.

Figures 8.20 and 8.21 show the percentage prediction error and Table 8.2 shows the usefulness of prediction methodology for scheduling an integer sort application on In-

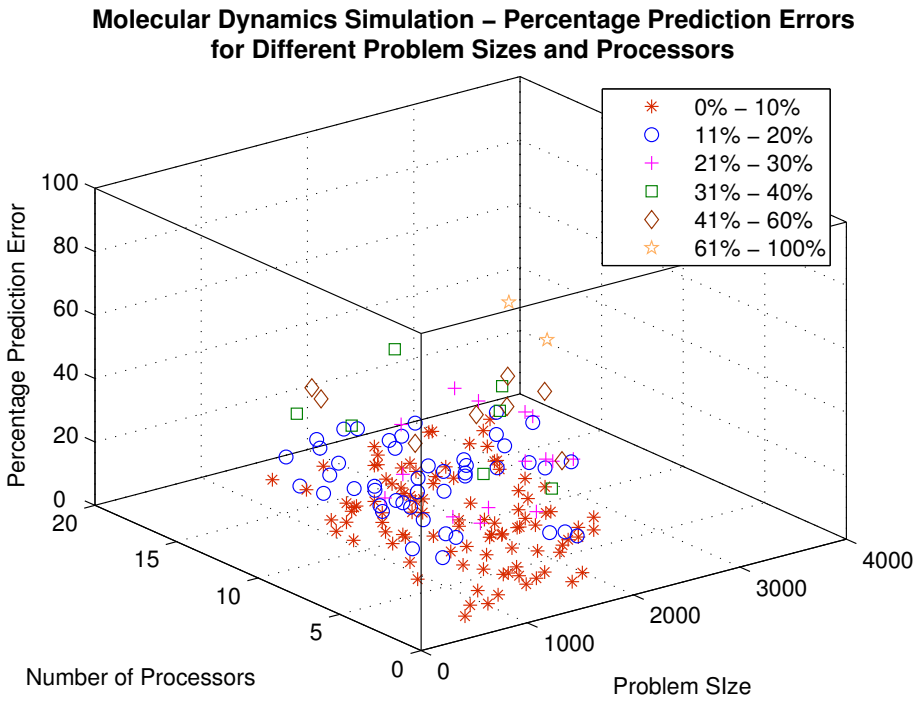


Figure 8.15: Percentage Prediction Errors for MD on the AMD Cluster with GrADS Loading

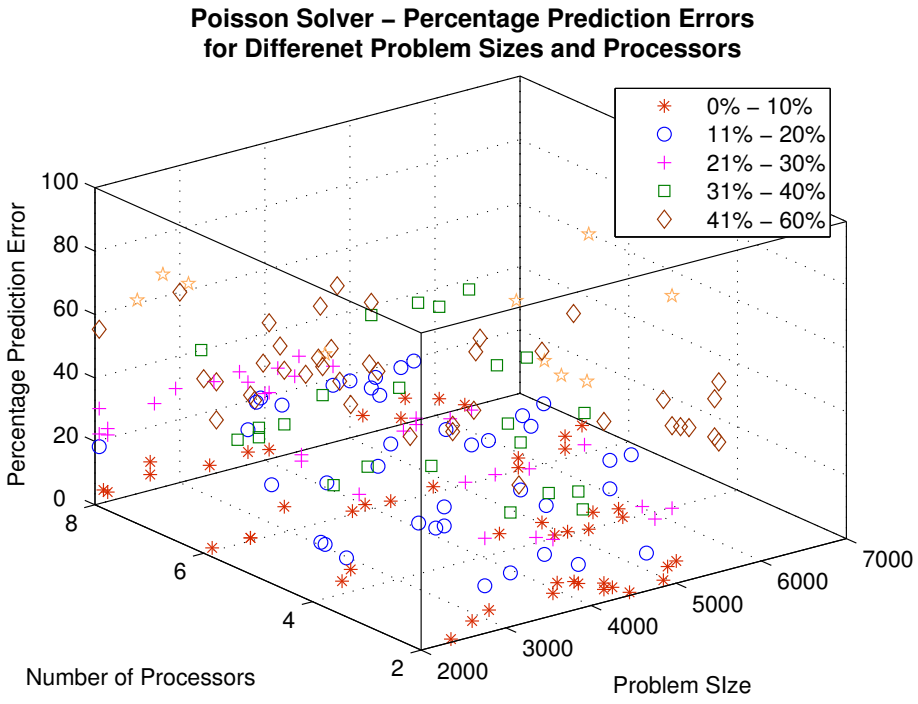


Figure 8.16: Percentage Prediction Errors for Poisson Solver on the Intel Cluster with Random Loading

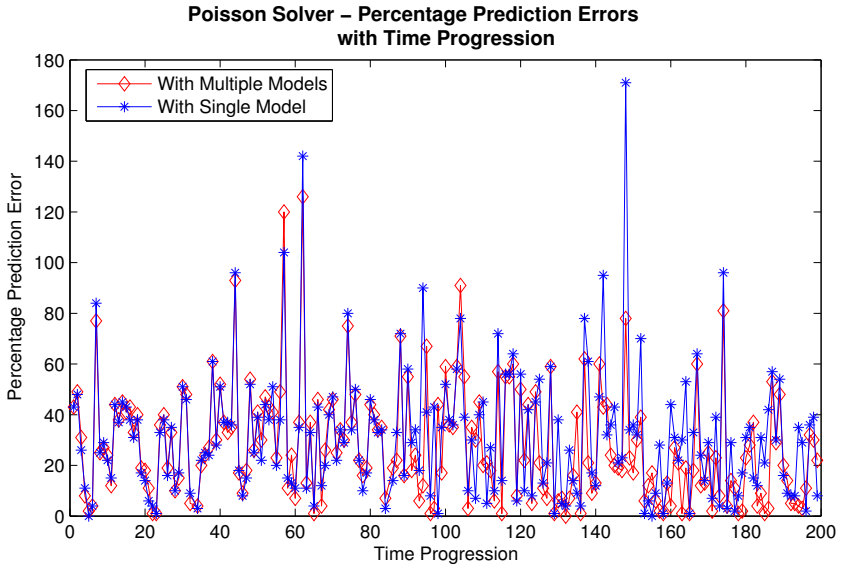


Figure 8.17: Percentage Prediction Errors at Different Times for Poisson Solver on the Intel Cluster with Random Loading

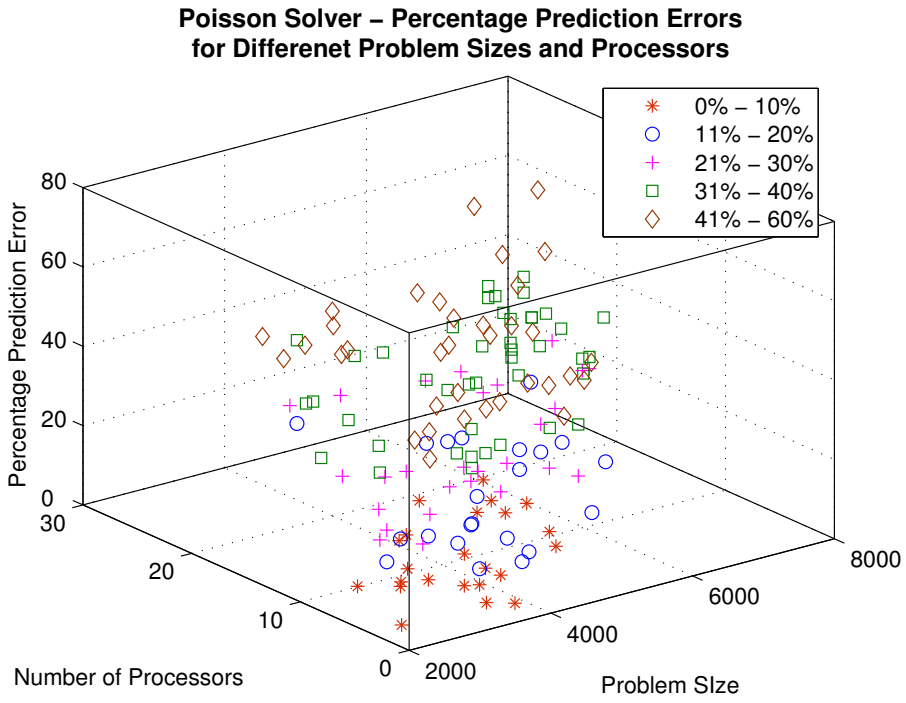


Figure 8.18: Percentage Prediction Errors for Poisson Solver on the Woodcrest Cluster with Random Loading

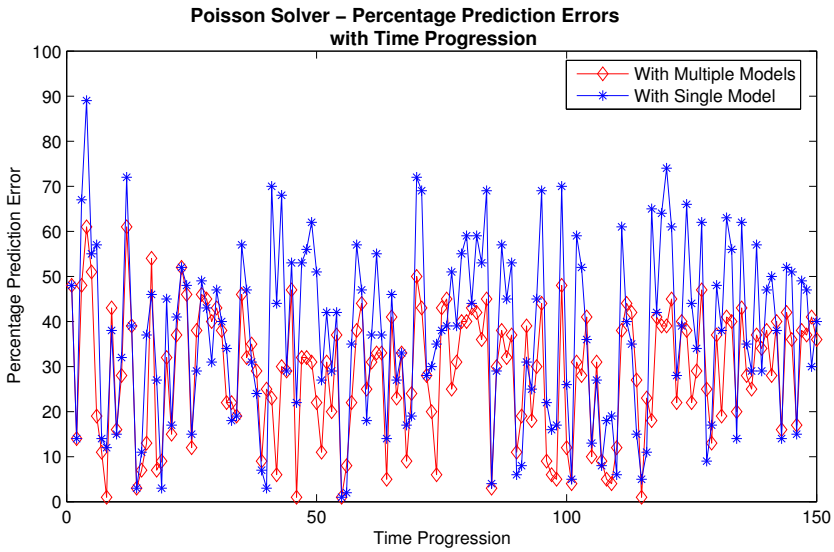


Figure 8.19: Percentage Prediction Errors at Different Times for Poisson Solver on the Woodcrest Cluster with Random Loading

tel cluster with random loading. We find that the percentage prediction errors are less than 30% for 95% of the predictions and less than 40% for 99% of the predictions. We also find from Table 8.2, the maximum percentage increase in execution time when a scheduler uses our predictions is only 11%. In most of the case, our performance modeling strategies to schedule integer sort application on the resources will give rise to perfect scheduling.

Figure 8.22 shows the results for integer sort application on the Intel cluster with GrADS loading. We find that the percentage prediction errors are less than 30% for 93% of the predictions and less than 40% for 97% of the predictions.

Figure 8.23 shows the results for integer sort application on the AMD cluster with GrADS loading. We find that the percentage prediction errors are less than 30% for 93% of the predictions and less than 40% for 96% of the predictions.

Figures 8.24 and 8.25 show the results for integer sort application on the 24-processor Woodcrest cluster with random loading. We find that the percentage prediction errors are less than 30% for 59% of the predictions and less than 40% for 75% of the predictions.

Figures 8.26 and 8.27 show the percentage prediction error and Table 8.3 shows the use-

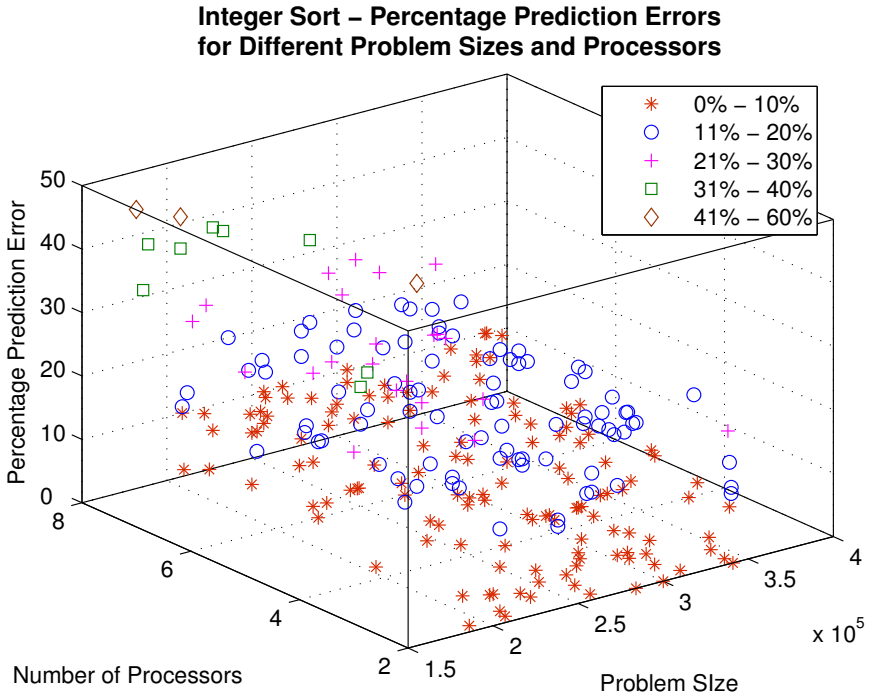


Figure 8.20: Percentage Prediction Errors for Integer Sort on the Intel Cluster with Random Loading

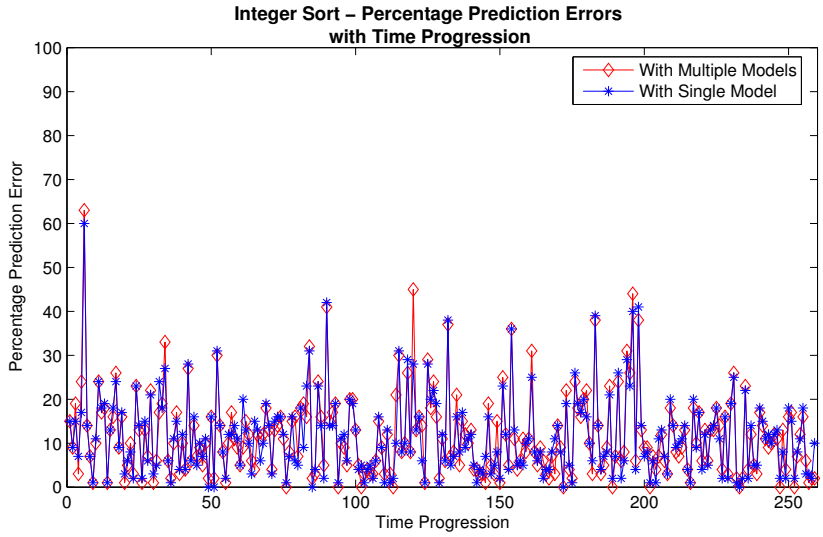


Figure 8.21: Percentage Prediction Errors at Different Times for Integer Sort on the Intel Cluster with Random Loading

Table 8.2: Usefulness of Predictions for Scheduling of Integer Sort on Intel Cluster with Random Loading

Problem Size Group	Problem Size (N) and Processors (P) for Minimum Predicted Execution Time: (1)	Problem Size (N) and Processors (P) for Minimum Actual Execution Time: (2)	Actual Execution Time for (1): (3)	Actual Execution Time for (2): (4)	Percentage Increase in Execution Time due to Prediction ((3-4)/4)
182000-189000	186000,8	186000,6	743.98	664.98	11.00%
222000-227000	222000,8	222000,8	729.89	729.89	0.00%
248000-252000	248000,8	250000,8	1157.69	1047.47	10.00%
257000-259000	257000,8	258000,8	1037.63	977.02	6.00%
279000-284000	284000,8	284000,8	1641.34	1584.8	3.00%
295000-300000	295000,8	295000,8	1459.17	1459.17	0.00%
327000-331000	327000,8	327000,8	1585.28	1585.28	0.00%
340000-343000	341000,8	341000,8	2003.63	2003.63	0.00%
379000-384000	382000,8	379000,8	2618.89	2495.66	4.00%

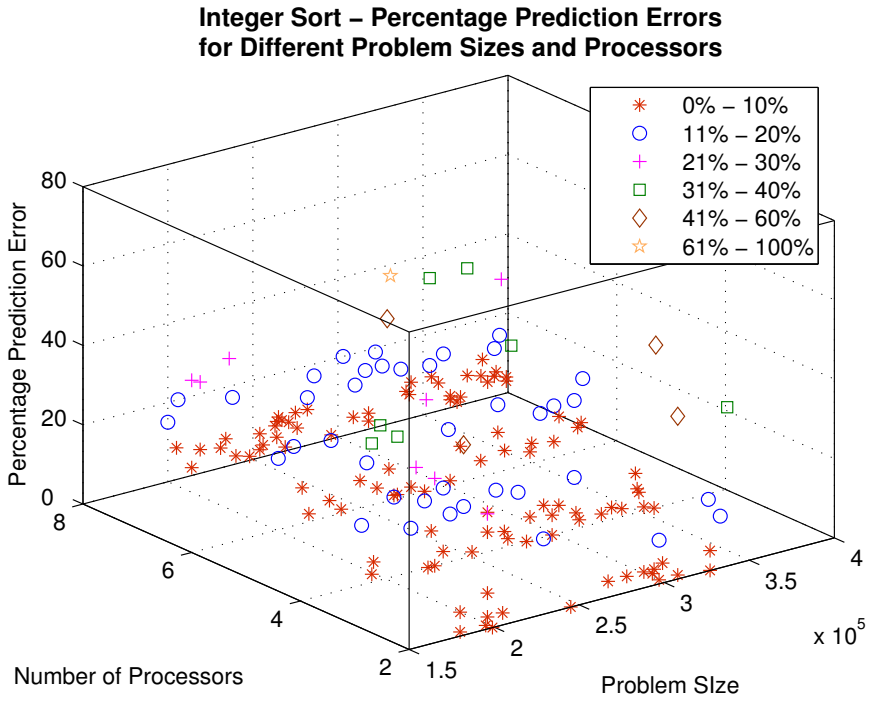


Figure 8.22: Percentage Prediction Errors for Integer Sort Application on the Intel Cluster with GrADS Loading

fulness of prediction methodology for scheduling a SSOR application on an Intel cluster with random loading. We find that the percentage prediction errors are less than 30% for 85% of the predictions and less than 40% for 94% of the predictions. We also find from Table 8.3 that except for 3 problem size groups, using our performance modeling strategies to schedule SSOR application on the resources will give rise to perfect scheduling.

Figures 8.28 and 8.29 show the results for SSOR application on Intel cluster with grads loading. We find that the percentage prediction errors are less than 30% for 98.5% of the predictions and less than 40% for 100% of the predictions.

Figure 8.30 shows the results for SSOR application on the AMD Cluster with GrADS Loading. We find that the percentage prediction errors are less than 30% for 95.3% of the predictions and less than 40% for 97% of the predictions.

Figures 8.31 and 8.32 show the comparison results for SSOR on Intel system with random loading conditions for 2 and 6 processors, respectively.

Table 8.3: Usefulness of Predictions for Scheduling of SSOR on Intel Cluster with Random Loading

Problem Size Group	Problem Size (N) and Processors (P) for Minimum Predicted Execution Time: (1)	Problem Size (N) and Processors (P) for Minimum Actual Execution Time: (2)	Actual Execution Time for (1): (3)	Actual Execution Time for (2): (4)	Percentage Increase in Execution Time due to Prediction ((3-4)/4)
720-816	768,8	768,8	268.16	268.16	0.00%
912-1008	960,8	912,8	466.91	435.76	7.00%
1104-1200	1104,8	1152,8	567.59	564.84	0.00%
1296-1392	1296,8	1296,8	692.11	692.11	0.00%
1584-1680	1584,8	1584,8	864.21	864.21	0.00%
1776-1872	1776,8	1872,8	1294.9	1271.5	1.00%
2112-2208	2112,8	2112,8	1508.61	1508.61	0.00%
2352-2448	2400,8	2400,8	1973.01	1973.01	0.00%
2544-2688	2544,8	2640,8	2277.15	2233.02	1.00%
2832-2976	2832,8	2832,8	2671.92	2671.92	0.00%

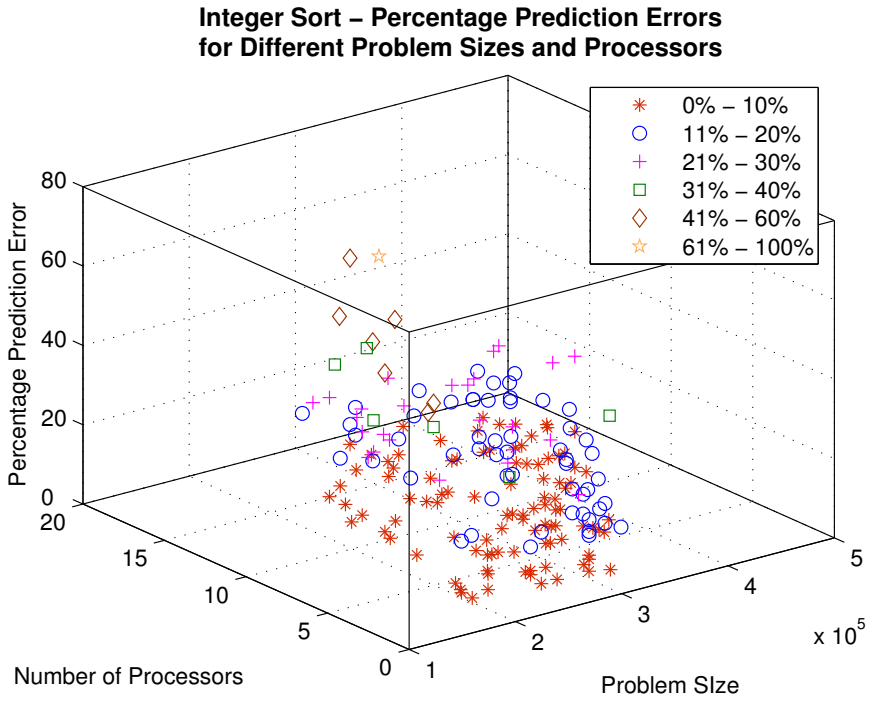


Figure 8.23: Percentage Prediction Errors for Integer Sort Application on the AMD Cluster with GrADS Loading

8.1.1 More Results on Cumulative Performance Models

Figures 8.33, 8.34 and 8.35 show the percentage prediction errors with single and cumulative model for MD, MPB and LAMMPS applications, respectively, for different application and processor configurations. The x-axis represents the different configurations.

8.2 More Results on Scheduling Strategies

Figures 8.36, 8.37 and 8.38 show the effects of different load setups on the performance of the algorithms for different number of processors when the allotted scheduling time was 25 seconds.

Figures 8.39, 8.40 and 8.41 show the effects of using increasing percentage of lightly loaded and heavily loaded machines, respectively, for 256 processors, on the schedules generated by the algorithms.

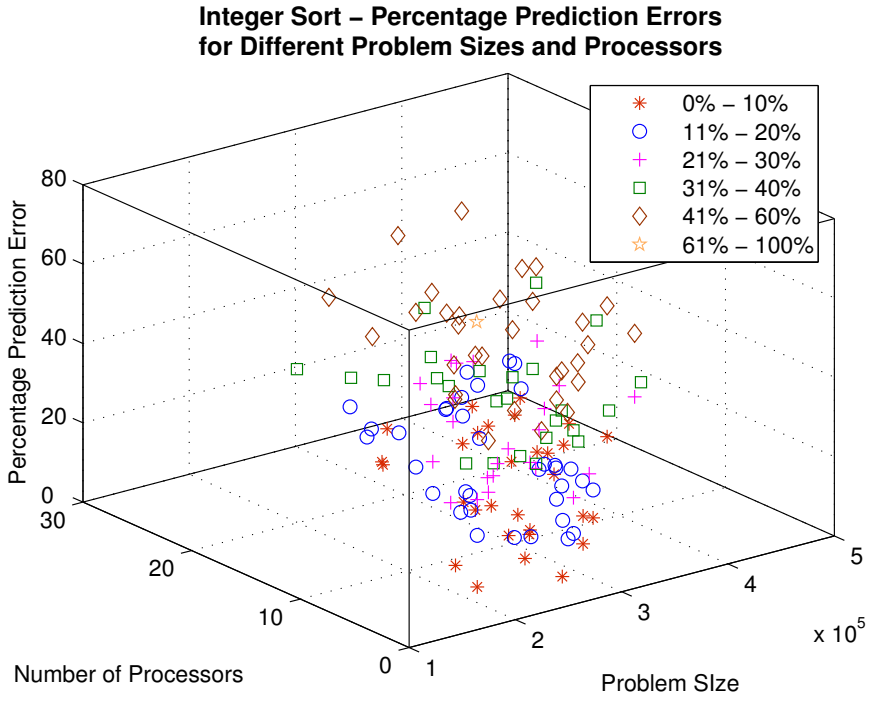


Figure 8.24: Percentage Prediction Errors for Integer Sort on the Woodcrest Cluster with Random Loading

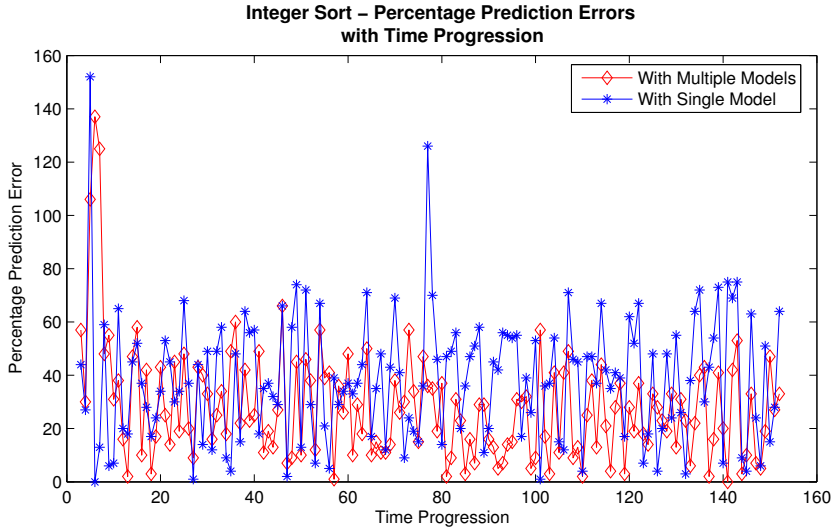


Figure 8.25: Percentage Prediction Errors at Different Times for Integer Sort on the Woodcrest Cluster with Random Loading

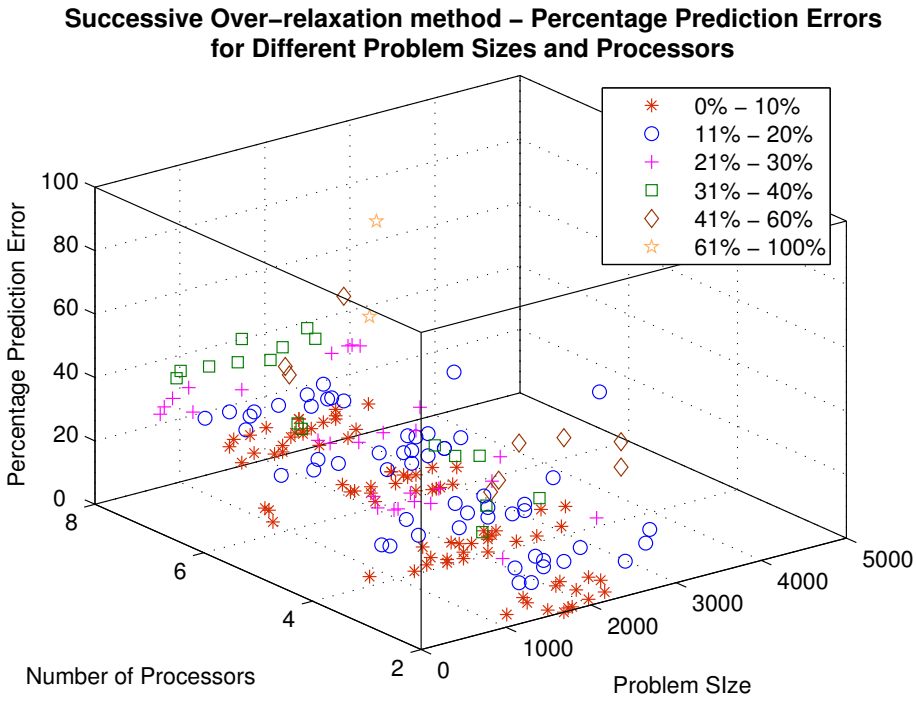


Figure 8.26: Percentage Prediction Errors for SSOR on the Intel Cluster with Random Loading

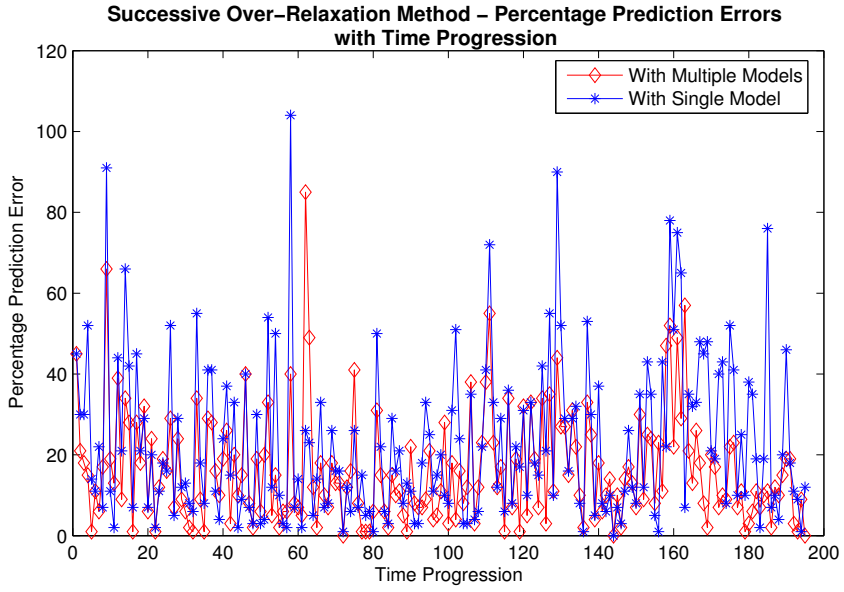


Figure 8.27: Percentage Prediction Errors at Different Times for SSOR on the Intel Cluster with Random Loading

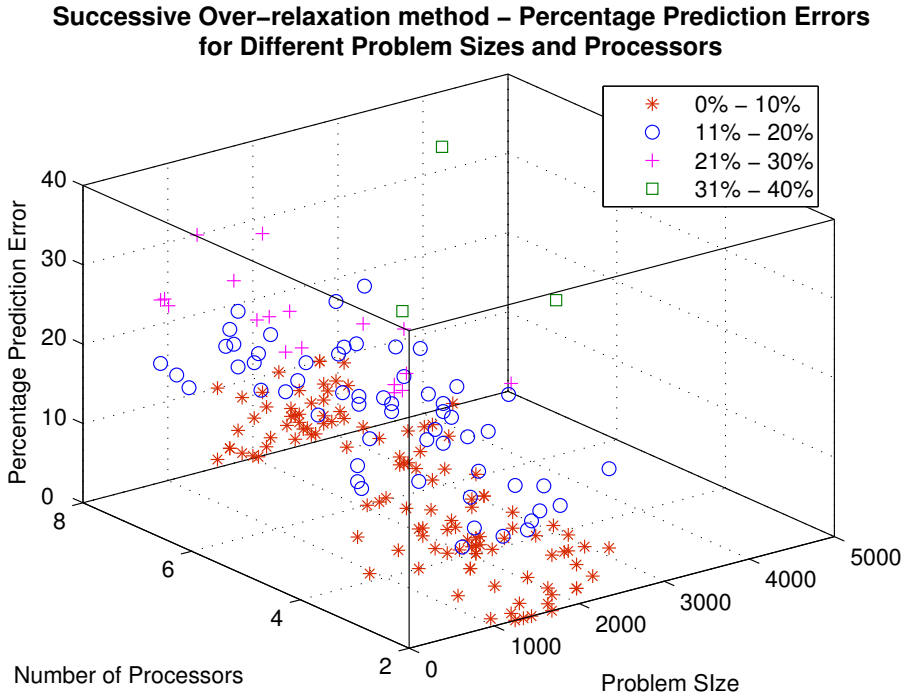


Figure 8.28: Percentage Prediction Errors for SSOR on the Intel Cluster with Grads Loading

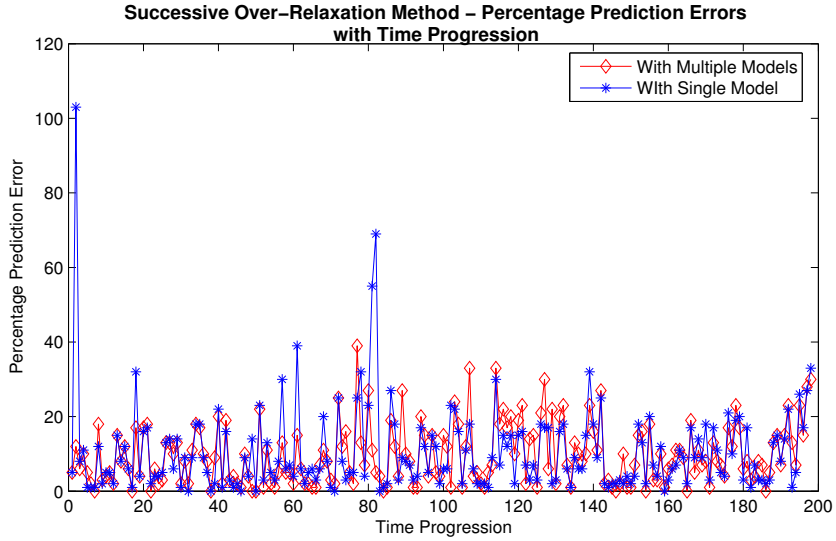


Figure 8.29: Percentage Prediction Errors at Different Times for SSOR on the Intel Cluster with Grads Loading

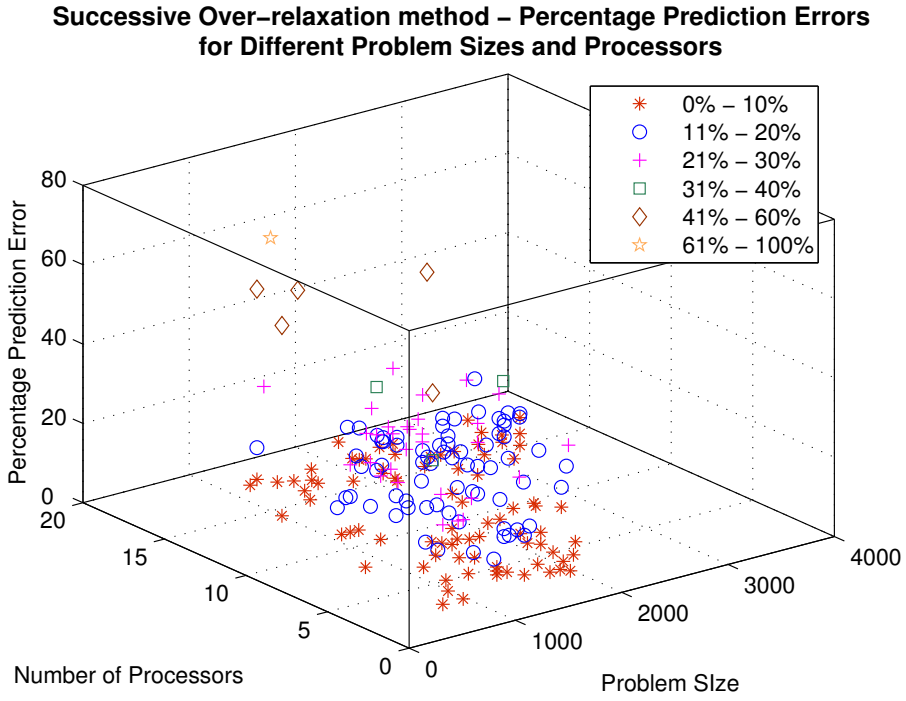


Figure 8.30: Percentage Prediction Errors for SSOR Application on the AMD Cluster with GrADS Loading

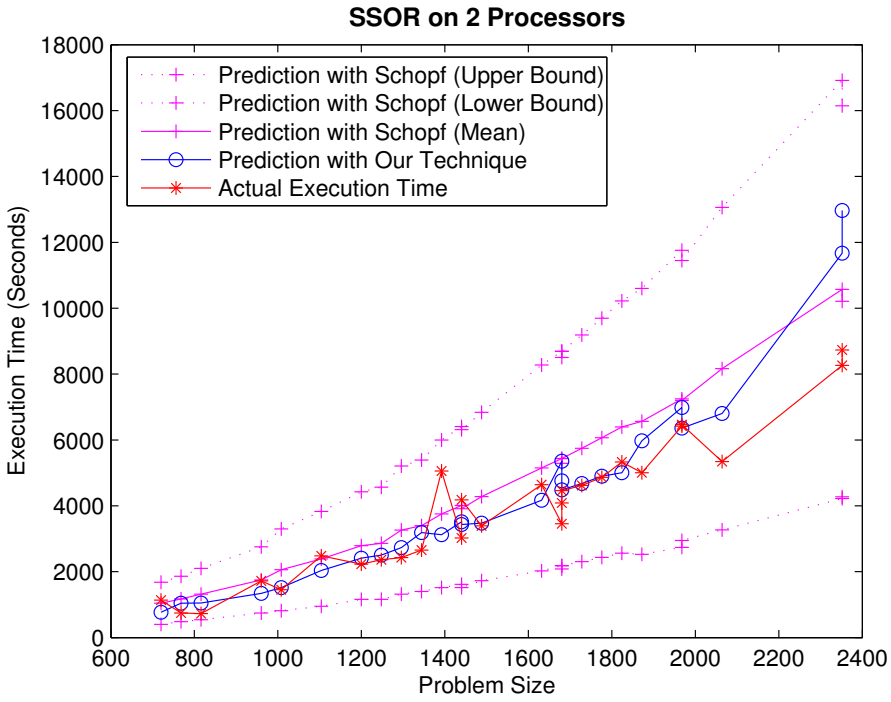


Figure 8.31: Comparison of Our Adaptive Method and Schopf' Method for 2 Processors with SSOR on the Intel Cluster with Random Loading; Schopf' Method - Average: 27.26%, Standard Deviation: 18.30%; Our Method - Average: 19.36%, Standard Deviation: 27.26%

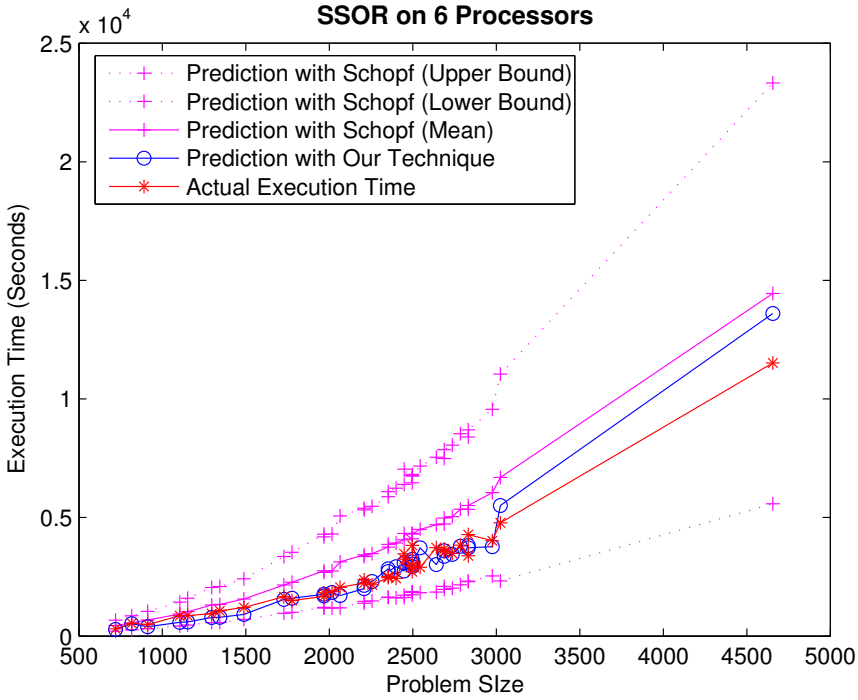


Figure 8.32: Comparison of Our Adaptive Method and Schopf' Method for 6 Processors with SSOR on the Intel Cluster with Random Loading; Schopf' Method - Average: 39.55%, Standard Deviation: 15.21%; Our Method - Average: 14.41%, Standard Deviation: 11.04%

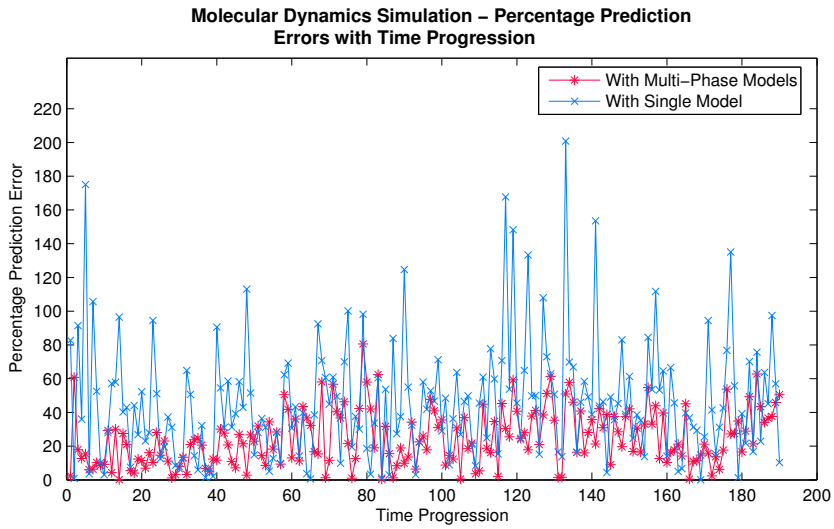


Figure 8.33: Percentage Prediction Errors with Single and Cumulative Models for MD

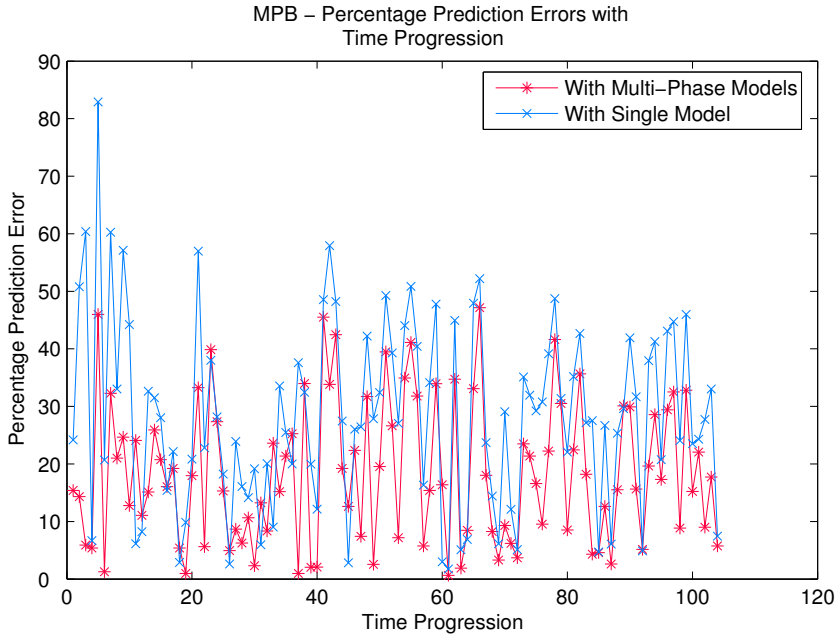


Figure 8.34: Percentage Prediction Errors with Single and Cumulative Models for MPB

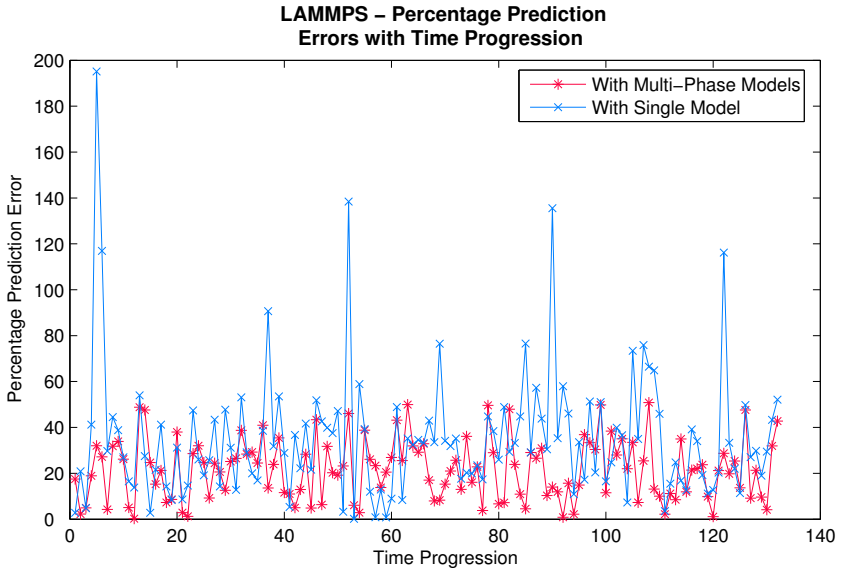


Figure 8.35: Percentage Prediction Errors with Single and Cumulative Models for LAMMPS

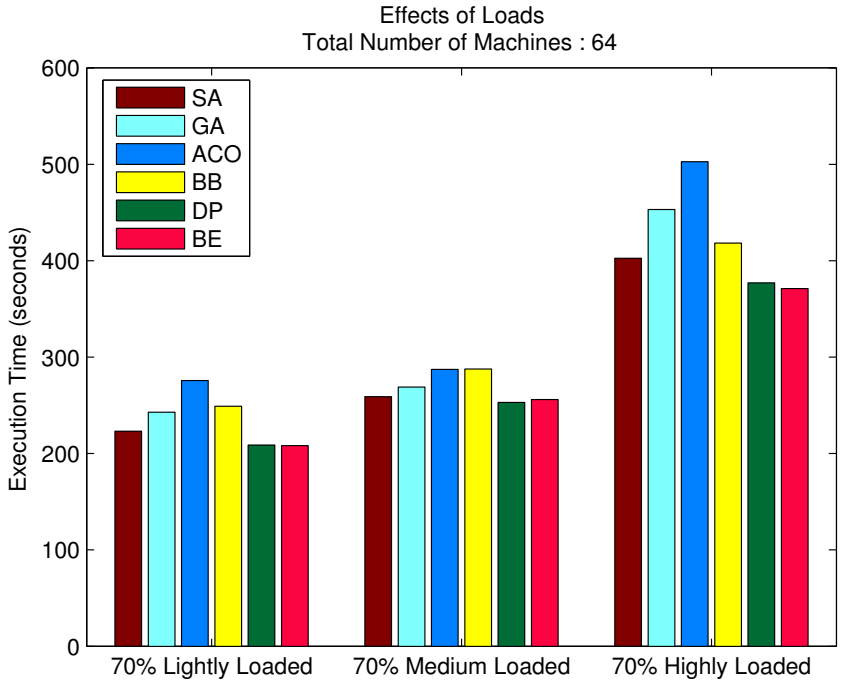


Figure 8.36: Effect of Loads on Machines for 64 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds

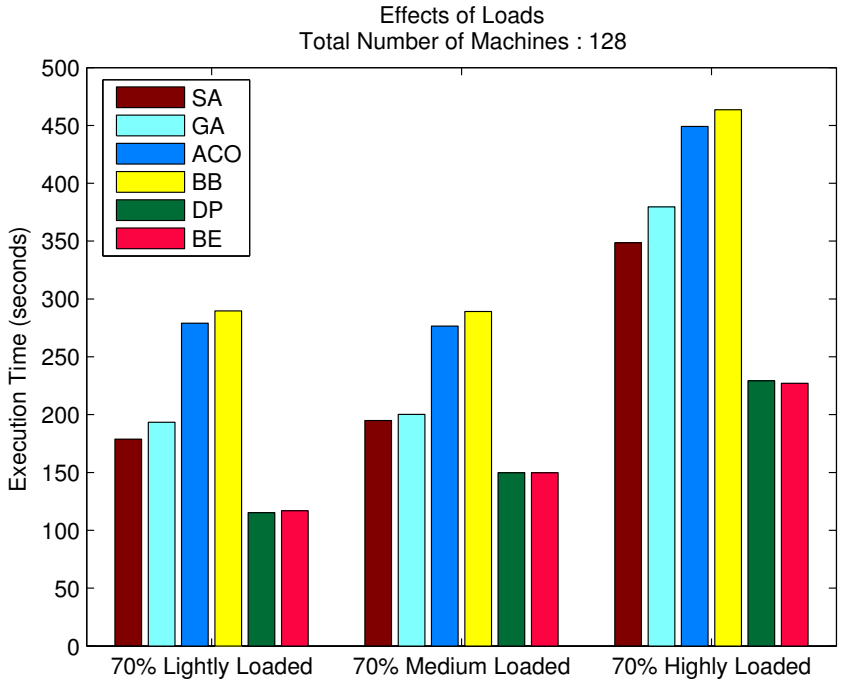


Figure 8.37: Effect of Loads on Machines for 128 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds

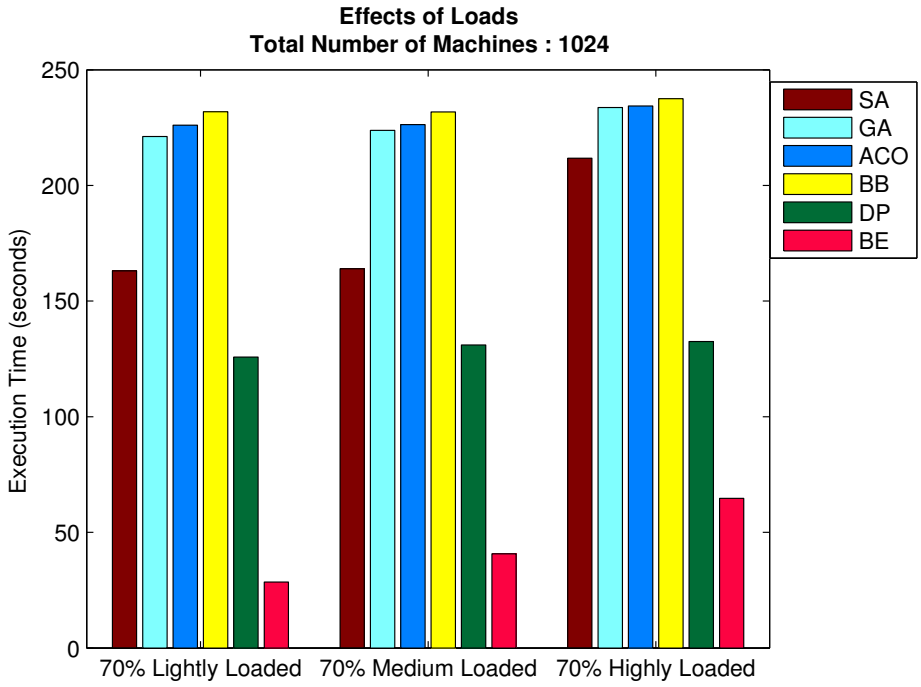


Figure 8.38: Effect of Loads on Machines for 1024 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds

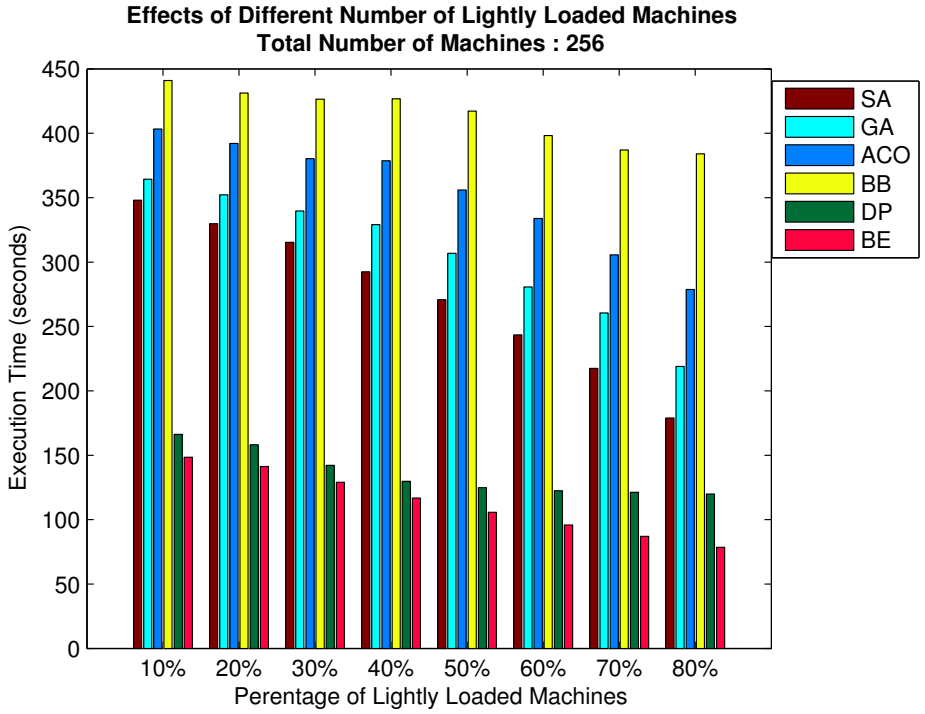


Figure 8.39: Effect of Increasing Lightly Loaded Machines for 256 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds

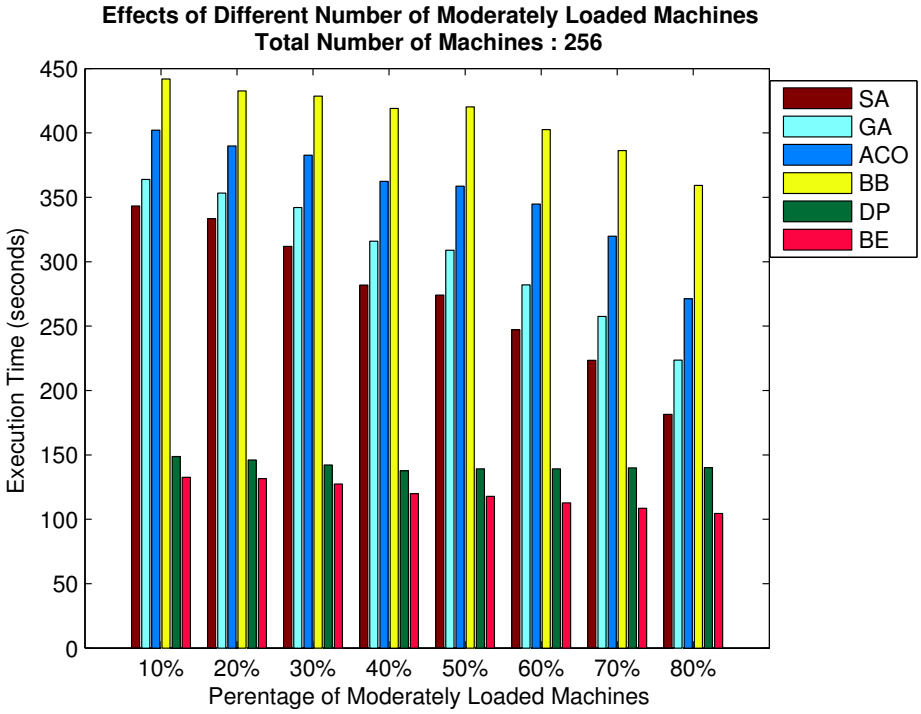


Figure 8.40: Effect of Increasing Moderate Loaded Machines for 256 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds

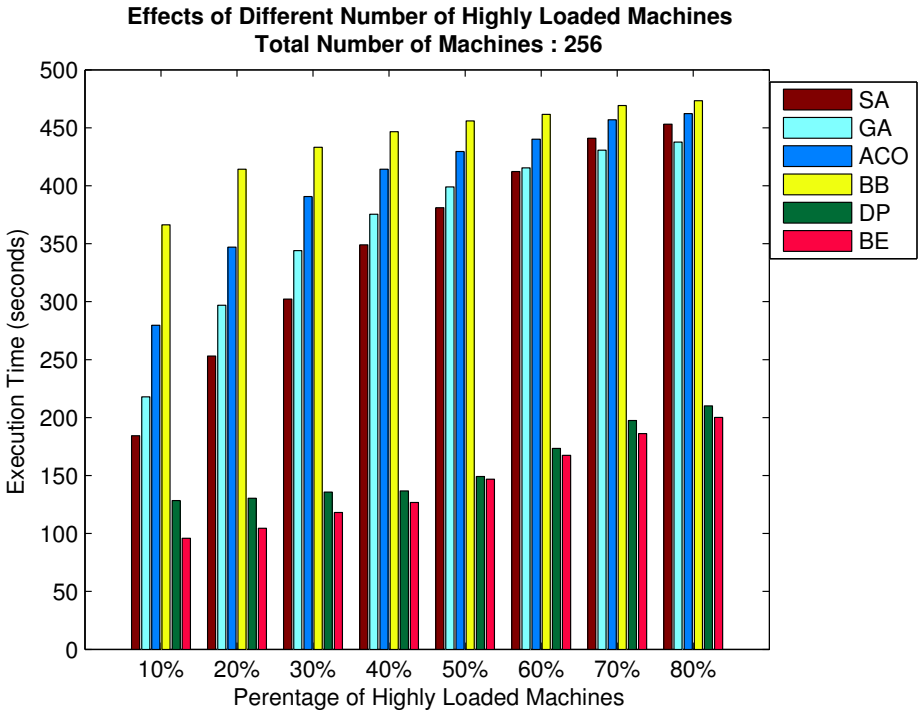


Figure 8.41: Effect of Increasing Heavily Loaded Machines for 256 processors. Application: MD with 2048 molecules, Time for Scheduling: 25 seconds

8.3 More Results on Rescheduling Strategies

Table 8.4 shows the average percentage difference between the execution times of the ChaNGa application corresponding to rescheduling plans generated by our algorithms and the plans generated by a brute force method for different rescheduling overheads. The table shows that the rescheduling plans generated by our algorithms lead to application execution times that are higher than the execution times corresponding to brute force method by less than 10%.

Figure 8.42 shows the comparison of the algorithms with brute force method for generation of rescheduling plans for ChaNGa application on 512 processors.

Table 8.4: Comparison of the Algorithms with Brute Force Method for Generation of Rescheduling Plans (ChaNGa)

Rescheduling Overhead (secs.)	Incremental	Division	Genetic
0	1.23	0.06	0.01
60	1.15	1.38	0.63
120	3.54	2.86	1.53
180	2.42	4.96	3.24
240	3.28	7.18	4.33
300	4.78	9.32	6.17

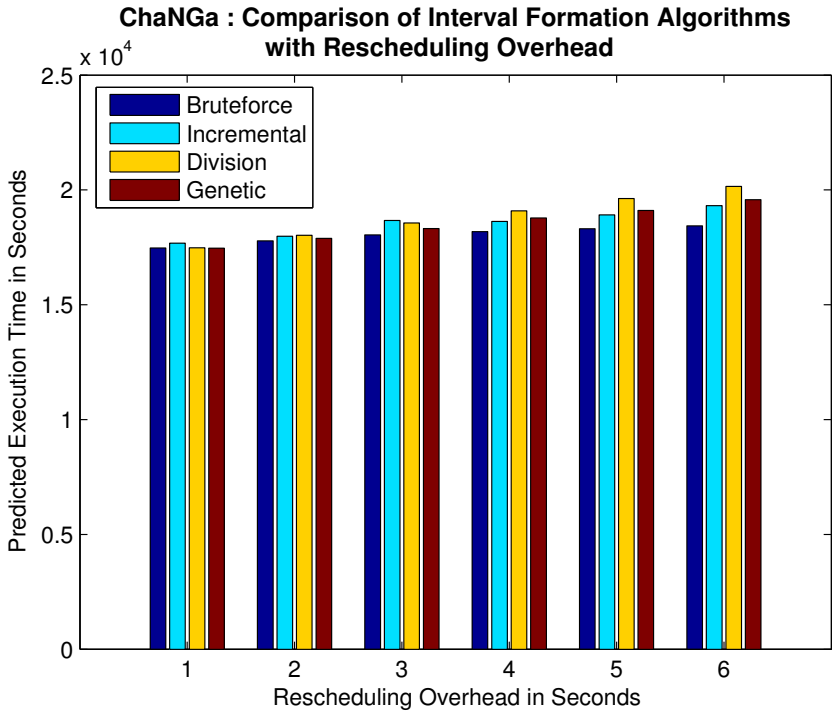


Figure 8.42: Comparison of the Algorithms with Brute Force Method for Generation of Rescheduling Plans (ChaNGa)

References

[1] J. Abawajy and S. Dandamudi. Parallel Job Scheduling on Multicluster Computing Systems. In *Proceedings of IEEE International Conference on Cluster Computing*, pages 11–18, 2003.

[2] V. Adve and M. Vernon. Parallel Program Performance Prediction using Deterministic Task Graph Analysis. *ACM Transactions on Computer Systems*, 22(1):94–136, 2004.

[3] Akshai K. Aggarwal and Robert D. Kent. An adaptive generalized scheduler for grid applications. In *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 188–194, Washington, DC, USA, 2005.

[4] Mona Aggarwal, Robert D. Kent, and Alioune Ngom. Genetic algorithm based scheduler for computational grids. In *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 209–215, Washington, DC, USA, 2005.

[5] A Al-Ani. Feature Subset Selection Using Ant Colony Optimization. *International Journal of Computational Intelligence*, 2(1), 2005.

[6] A. M. Alkindi, D. J. Kerbyson, and G. R. Nudd. Dynamic Instrumentation and Performance Prediction of Application Execution. *In High Performance Computing and Networking (HPCN2001)*, 2110:313–323, 2001.

[7] William Allcock, John Bresnahan, Rajkumar Kettimuthu, and Michael Link. The globus striped gridftp framework and server. In *SC '05: Proceedings of the 2005*

ACM/IEEE conference on Supercomputing, page 54, Washington, DC, USA, 2005. IEEE Computer Society.

[8] G. Allen, T. Dramlitsch, I. Foster, N.T. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 52–, 2001.

[9] C. An, M. Taufer, A. Kerstens, and C. Brooks III. Predictor@Home: A P̈rotein Structure Prediction Supercomputer' Based on Global Computing. *IEEE Transactions on Parallel and Distributed Systems*, 17(8):786–796, 2006.

[10] Stergios V. Anastasiadis and Kenneth C. Sevcik. Parallel application scheduling on networks of workstations. *Journal of Parallel and Distributed Computing*, 43:109–124, 1997.

[11] C. Anglano. Predicting Parallel Applications Performance on Non-Dedicated Cluster Platforms. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 172–179, 1998.

[12] ApGrid - asia pacific grid. *http://apgrid.org*.

[13] Astrogrid. *http://www.astrogrid.org*.

[14] Athena Code Home Page. *http://www.astro.princeton.edu/ jstone/athena.html*.

[15] R. Badia, J. Labarta, J. Gimenez, and F. Escale. DIMEMAS: Predicting MPI Applications Behavior in Grid Enviornaments. In *In Workshop on Grid Applications and Programming Tools (GGF8)*, Seattle York , U.S.A, June 2003.

[16] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, and Y. Robert. Centralized Versus Distributed Schedulers for Multiple Bag-of-Task Applications. In *20th International Parallel and Distributed Processing Symposium*, pages 10–, 2006.

[17] F. Berman and et. al. New Grid Scheduling and Rescheduling Methods in the GrADS Project. volume 33, pages 209–229, 2005.

[18] F. Berman and R. Wolski. The AppLeS Project: A Status Report. *Proceedings of the 8th NEC Research Symposium*, May 1997.

[19] Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, Gary Shao, Shava Smallen, Neil Spring, Alan Su, and Dmitrii Zagorodnov. Adaptive computing on the grid using apples. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, 2003.

[20] Biogrid. *http://www.biogrid.jp*.

[21] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[22] R. Block, S. Sarukkai, and P. Mehra. Automated Performance Prediction of Message-Passing Parallel Programs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 31, 1995.

[23] C. Blum and M. Sampels. An Ant Colony Optimization Algorithm for Shop Scheduling Problems. *Journal of Mathematical Modelling and Algorithms*, 3(3):285–308, 2004.

[24] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 759–767, Washington, DC, USA, 2005.

[25] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task Scheduling Strategies for Workflow-based Applications in Grids. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 759–767, 2005.

[26] W. Boyer and G. Hura. Non-evolutionary Algorithm for Scheduling Dependent

Tasks in Distributed Heterogeneous Computing Environments. *Journal of Parallel and Distributed Computing*, 65(9):1035–1046, 2005.

[27] T. Braun, H. Siegel, N. Beck, L. Bölóni, A. Reuther, M. Theys, B. Yao, R. Freund, M. Maheswaran, J. Robertson, and D. Hensgen. A Comparison Study of Static Mapping Heuristics for a Class of Meta-Tasks on Heterogeneous Computing Systems. In *HCW '99: Proceedings of the Eighth Heterogeneous Computing Workshop*, page 15, 1999.

[28] A. I. Bucur and H. J. Epema. The Influence of the Structure and Sizes of Jobs on the Performance of Co-allocation. In *JSSPP*, pages 154–173, 2000.

[29] T. Bui, T. Nguyen, C. Patel, and K.-A. Phan. An Ant-based Algorithm for Coloring Graphs. *Discrete Applied Mathematics*, 156(2), 2008.

[30] Buisson, Jrmy, Sonmez, Ozan, Mohamed Hasim, Lammers Wouter, Epema, and Dick. Scheduling malleable applications in multicluster systems. Technical Report TR-0092, Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, May 2007.

[31] L. Carrington, M. Laurenzano, A. Snavely, R. Campbell, and L. Davis. How Well Can Simple Metrics Represent the Performance of HPC Applications? In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 48, 2005.

[32] Henri Casanova, Dmitrii Zagorodnov, Francine Berman, and Arnaud Legrand. Heuristics for scheduling parameter sweep applications in grid environments. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 349, Washington, DC, USA, 2000.

[33] ChaNGa (Charm N-body GrAvity Solver)
. *http://librarian.phys.washington.edu/astro/index.php/Research:ChaNGa*.

[34] L. Chen, Q. Zhu, and G. Agrawal. Supporting Dynamic Migration in Tightly Coupled Grid Applications. In *Supercomputing '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 117, 2006.

[35] Po-Cheng Chen, Jyh-Biau Chang, Tyng-Yeu Liang, Ce-Kuen Shieh, and Yi-Chang Zhuang. A multi-layer resource reconfiguration framework for grid computing. In *MCG '06: Proceedings of the 4th international workshop on Middleware for grid computing*, page 13, 2006.

[36] W. Chrabakh and R. Wolski. GridSAT: A Chaff-based Distributed SAT Solver for the Grid. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 37, 2003.

[37] M. Clement and M. Quinn. Automated Performance Prediction for Scalable Parallel Computing. *Parallel Computing*, 23(10):1405–1420, 1997.

[38] CurveExpert. *http://curveexpert.webhop.biz*.

[39] H. Dail, H. Casanova, and F. Berman. A Decoupled Scheduling Approach for the GrADS Enviornament. In *SC 2002*, November 2002.

[40] Data Mining Grid. *http://www.datamininggrid.org*.

[41] DataFit. *http://www.curvefitting.com*.

[42] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming*, 13(3):219–237, 2005.

[43] T. Desell, K. Maghraoui, and C. Varela. Malleable Applications for Scalable High Performance Computing. *Cluster Computing*, 10(3):323–337, 2007.

[44] A. Dhodapkar and J. Smith. Comparing Program Phase Detection Techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 217–227, December 2003.

[45] P. Dinda. Online Prediction of the Running Time of Tasks. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, pages 383–394, San Francisco, U.S.A., August 2001.

[46] C. Ding, S. Dwarkadas, M. Huang, K. Shen, and J. Carter. Program Phase Detec-

tion and Exploitation. In *20th International Parallel and Distributed Processing Symposium*, April 2006.

[47] S. Dong, N. Karonis, and G. Karniadakis. Grid Solutions for Biological and Physical Cross-Site Simulations on the Teragrid. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.

[48] M. Dorigo, E. Bonabeau, and G. Theraulaz. Ant Algorithms and Stigmergy. *Future Generation Computer Systems*, 16(9):851–871, 2000.

[49] Molecular Modelling for Drug Design. *http://www.gridbus.org/vlab/*.

[50] L. Drummond, E. Uchoa, A. Gonçalves, J. Silva, M. Santos, and M. de Castro. A Grid-Enabled Distributed Branch-and-Bound Algorithm with Application on the Steiner Problem in Graphs. *Parallel Computing*, 32(9):629–642, 2006.

[51] EMBRACE. *http://www.embracegrid.info/page.php?page=home*.

[52] eMinerals. *http://www.eminerals.org*.

[53] X. Espinal, D. Barberis, K. Bos, S. Campana, L. Goossens, J. Kennedy, G. Negri, S. Padhi, L. Perini, G. Poulard, D. Rebatto, S. Resconi, A. de Salvo, and R. Walker. Large-Scale ATLAS Simulated Production on EGEE. In *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, pages 3–10, 2007.

[54] P. Sugavanam et. al. Robust Static Allocation of Resources for Independent Tasks under Makespan and Dollar Cost Constraints. *Journal of Parallel and Distributed Computing*, 67(4):400–416, 2007.

[55] R. Fernandes, K. Pingali, and P. Stodghill. Mobile MPI Programs in Computational Grids. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 22–31, 2006.

[56] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, pages 2–13, 2005.

[57] Ian Foster and Carl Kesselman. The Grid: Blueprint for a New Computing Infrastructure. 1999.

[58] Ian Foster, Carl Kesselman, and Steven Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[59] M. Frigo and S. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on Program Generation, Optimization, and Platform Adaptation.

[60] S. Fu and C-Z. Xu. Service Migration in Distributed Virtual Machines for Adaptive Grid Computing. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 358–365, 2005.

[61] Song Fu and Cheng-Zhong Xu. Stochastic modeling and analysis of hybrid mobility in reconfigurable distributed virtual machines. *J. Parallel Distrib. Comput.*, 66(11):1442–1454, 2006.

[62] Noriyuki Fujimoto and Kenichi Hagihara. Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid. In *Proc. ICPP 2003*, pages 391–398, 2003.

[63] Ganglia Cluster Toolkit. *http://ganglia.sourceforge.net/*.

[64] Y. Gao, H. Rong, and J Huang. Adaptive Grid Job Scheduling with Genetic Algorithms. *Future Generation Computer Systems*, 21(1):151–161, 2005.

[65] M. Gardner, W. chun Feng, J. Archuleta, H. Lin, and X. Mal. Parallel Genomic Sequence-Searching on an Ad-hoc Grid: Experiences, Lessons Learned, and Implications. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 104, 2006.

[66] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[67] Garuda - india's national grid computing initiative. *http://www.garudaindia.in*.

[68] GECEM : Grid-Enabled Computational Electromagnetics. *http://www.wesc.ac.uk/projectsite/gecem*.

[69] Jörn Gehring and Alexander Reinefeld. Mars-a framework for minimizing the job execution time in a metacomputing environment. *Future Gener. Comput. Syst.*, 12(1):87–99, 1996.

[70] GeneGrid. *http://www.qub.ac.uk/escience/projects/genegrid*.

[71] Geodise: Aerospace Design Optimisation. *http://www.geodise.org/*.

[72] GeoFEM : Multi-Purpose/Multi-Physics Parallel Finite Element Simulator. *http://geofem.tokyo.rist.or.jp*.

[73] The GrADS Project. *http://www.hipersoft.rice.edu/grads*.

[74] GrADS Traces. *http://pompone.cs.ucsb.edu/ rich/data*.

[75] GriPhyN Virtual Data System
. *http://www.ci.uchicago.edu/wiki/bin/view/VDS/VDSWeb/WebMain*.

[76] D. A. Grove and P. D. Coddington. Modeling message-passing programs with a performance evaluating virtual parallel machine. *Performance Evaluation*, 60(1-4):165–187, 2005.

[77] L. Han, A. Asenov, D. Berry, C. Millar, G. Roy, S. Roy, R. Sinnott, and G. Stewart. Towards a Grid-Enabled Simulation Framework for Nano-CMOS Electronics. In *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, pages 305–311, 2007.

[78] S. Hastings, T. Kurc, S. Langella, U. Catalyurek, T. Pan, and J. Saltz. Image Processing on the Grid: A Toolkit or Building Grid-enabled Image Processing Applications. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 36, 2003.

[79] L. He, S. Jarvis, D. Spooner, X. Chen, and G. Nudd. Dynamic Scheduling of Parallel Jobs with QoS Demands in Multiclusters and Grids. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 402–409, 2004.

[80] C. Huang, G. Zheng, L. Kalé, and S. Kumar. Performance Evaluation of Adaptive MPI. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 12–21, 2006.

[81] Eduardo Huedo, Ruben S. Montero, and Ignacio M. Llorente. A framework for adaptive execution in grids. *Softw. Pract. Exper.*, 34(7):631–651, 2004.

[82] M. Hussein, K. Mayes, M. Luján, and J. Gurd. Adaptive Performance Control for Distributed Scientific Coupled Models. In *ICS '07: Proceedings of the 21st annual International Conference on Supercomputing*, pages 274–283, 2007.

[83] E. Ipek, B. de Supinski, M. Schulz, and S. McKee1. An Approach to Performance Prediction for Parallel Applications. In *Euro-Par, Springer LNCS*, volume 3648, pages 196–205, 2005.

[84] M. Jayawardena and S. Holmgren. Grid-Enabling an Efficient Algorithm for Demanding Global Optimization Problems in Genetic Analysis. In *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, pages 205–212, 2007.

[85] D. Kebbal, E.-G. Talbi, and J.-M. Geib. Scheduling parallel adaptive applications in networks of workstations and clusters of processors. *cluster*, 00, 2001.

[86] Sakamoto Kentaro and Sato Hiroyuki. A resource-oriented grid meta-scheduler based on agents. In *PDCN'07: Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference*, pages 109–114, Anaheim, CA, USA, 2007. ACTA Press.

[87] S. Kim and J. Weissman. A Genetic Algorithm Based Approach for Scheduling Decomposable Data Grid Applications. In *International Conference on Parallel Processing, 2004*, pages 406–413.

[88] K. Kurowski, B. Ludwiczak, J. Nabrzyski, A. Oleksiak, and J. Pukacki. Dynamic grid scheduling with job migration and rescheduling in the gridlab resource management system. *Sci. Program.*, 12(4):263–273, 2004.

[89] LabFit. *http://www.angelfire.com/rnb/labfit*.

[90] LAMMPS Molecular Dynamics Simulator. *http://lammps.sandia.gov*.

[91] B. Lee, D. Brooks, B. de Supinski, M. Schulz, K. Singh, and S. McKee. Methods of Inference and Learning for Performance Modeling of Parallel Applications. In *Proceedings of Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, San Jose , California, U.S.A, March 2007.

[92] HwaMin Lee, KwangSik Chung, SungHo Chin, JongHyuk Lee, DaeWon Lee, Seongbin Park, and HeonChang Yu. A resource management and fault tolerance services in grid computing. *J. Parallel Distrib. Comput.*, 65(11):1305–1317, 2005.

[93] K. Li. Job Scheduling and Processor Allocation for Grid Computing on Metacomputers. *Journal of Parallel and Distributed Computing*, 65(11):1406–1418, 2005.

[94] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[95] S. Ludtke, P. Baldwin, and W. Chiu. EMAN: Semiautomated Software for High-resolution Single-particle Reconstructions. *Journal of Structural Biology*, 128:82–97, 1999.

[96] Josep L. Lrida, Francesc Solsona, Francesc Gin, Mauricio Hanzich, J. R. Garca, and Porfidio Hernndez. Metaloras: A re-scheduling and prediction metascheduler for non-dedicated multiclusters. In *Lecture Notes in Computer Science,Springer*, pages 195–203, 2007.

[97] K. Maghraoui, T. Desell, B. Szymanski, and C. Varela. The Internet Operating System: Middleware for Adaptive Distributed Computing. *International Journal of High Performance Computing Applcations*, 20(4):467–480, 2006.

[98] K. Maghraoui, T. Desell, B. Szymanski, and C. Varela. Dynamic Malleability in Iterative MPI Applications. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 591–598, 2007.

[99] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu, and L. Johnsson. Scheduling Strategies for Mapping Application Workflows onto the

Grid. In *HPDC '05: Proceedings of the High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium*, pages 125–134, 2005.

[100] R. Montero, E. Huedo, and I. Llorente. Grid Resource Selection for Opportunistic Job Migration. In *Euro-Par*, pages 366–373, 2003.

[101] MIT Photonic-Bands (MPB). *http://ab-initio.mit.edu/wiki/index.php/MIT*$_{P}hotonic_{B}ands$.

[102] C. Mueller, M. Dalkilic, and A. Lumsdaine. High-Performance Direct Pairwise Comparison of Large Genomic Sequences. *IEEE Transactions on Parallel and Distributed Systems*, 17(8):764–772, 2006.

[103] K. C. Nainwal, J. Lakshmi, S. K. Nandy, Ranjani Narayan, and K. Varadarajan. A framework for qos adaptive grid meta scheduling. *dexa*, 00:292–296, 2005.

[104] R. Nessah, F. Yalaoui, and C. Chu. A Branch-and-Bound Algorithm to Minimize Total Weighted Completion Time on Identical Parallel Machines with Job Release Dates. *Computers and Operations Research*, 35(4):1176–1190, 2008.

[105] The Neurogrid Project. *http://www.gridbus.org/neurogrid/*.

[106] G. Nudd, D. Kerbysin, E. Papaefstathiou, S. Perry, J. Harper, and D. Wilcox. PACE - A Toolset for the Performance Prediction of Parallel and Distributed Systems. *The International Journal of High Performance Computing Applications*, 14(3):228–251, Fall 2000.

[107] M. Orr, J. Hallam, K. Takezawa, A. Murra, S. Ninomiya, M. Oide, and T. Leonard. Combining Regression Trees and Radial Basis Function Networks. *International Journal of Neural Systems*, 10(6):453–465, December 200.

[108] Pascal's Triangle. *http://mathforum.org/dr.math/faq/faq.pascal.triangle.html*.

[109] L. Pearlman, C. Kesselman, S. Gullapalli, B. Spencer Jr., J. Futrelle, K. Ricker, I. Foster, P. Hubbard, and C. Severance. Distributed Hybrid Earthquake Engineering Experiments: Experiences with a Ground-Shaking Grid Application. In

13th IEEE International Symposium on High performance Distributed Computing, 2004.

[110] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting Phases in Parallel Applications on Shared Memory Architectures. In *20th International Parallel and Distributed Processing Symposium*, 2006.

[111] A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, and S. Vadhiyar. Numerical Libraries and the Grid: The GrADS Experiments with ScaLAPACK. *Journal of High Performance Applications and Supercomputing*, 15(4):359–374, Winter 2001.

[112] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the Network. *NetStore99: The Network Storage Symposium*, 1999.

[113] Particle Physics Data Grid. *http://www.ppdg.net/*.

[114] J. Ruscio, M. Heffner, and S. Varadarajan. DejaVu: Transparent User-Level Checkpointing, Migration, and Recovery for Distributed Systems. In *IPDPS '07: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

[115] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan. Scheduling of Parallel Jobs in a Heterogeneous Multi-Site Environment. In *Proceedings of 9th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2003)*, pages 87–104, 2003.

[116] R. Sakellariou and H. Zhaou. A Low-cost Rescheduling Policy for Efficient Mapping of Workflows on Grid Systems. volume 12, pages 253–262, 2004.

[117] H. A. Sanjay and S. Vadhiyar. Performance Modeling of Parallel Applications for Grid Scheduling. *Journal of Parallel and Distributed Computing*, 68(8):1135–1145, 2008.

[118] J. Schopf. Structural Prediction Models for High Performance Distriuted Applica-

tions. In *Proceedings of Cluster Computing Conference (CCC'97)*, Atlanta , USA, March 1997.

[119] J. Schopf and F. Berman. Performance Prediction in Production Environments. In *Proceedings of 12th International Parallel Processing Symposium*, Orlando , USA, March 1998.

[120] J. Schopf and F. erman. Using Stochastic Information to Predict Application Behavior on Contended Resources. *International Journal on Foundation in Computer Science*, 12(3):341–364, June 2001.

[121] X. Shen, C. Zhang, C. Ding, M. Scott, S. Dwarkadas, and M. Ogihara. Analysis of Input-Dependent Program Behavior Using Active Profiling. In *Workshop on Experimental Computer Science*, June 2007.

[122] X. Shen, Y. Zhong, and C. Ding. Predicting Locality Phases for Dynamic Memory Optimization. *Journal of Parallel and Distributed Computing*, 67(7):783–796, 2007.

[123] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *ISCA '03: Proceedings of the 30th Annual International Symposiumon Computer architecture*, 2003.

[124] K. Sim and W. Sun. Ant Colony Optimization for Routing and Load-Balancing: Survey and New Directions. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 33(5):560–572, 2003.

[125] K. Singh, E. İpek, S. McKee, B. de Supinski, M. Schulz, and R. Caruana. Predicting Parallel Application Performance via Machine Learning Approaches. *Concurrency and Computation: Practice and Experience.*, 19(17):2219–2235, 2007.

[126] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A Framework for Performance Modeling and Prediction. In *The 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, November 2002.

[127] P. Somol, P. Pudil, and J. Kittler. Fast Branch & Bound Algorithms for Optimal

Feature Selection. *IEEE Transactions on Pattern Analysis and Machine Intelligenc*, 26(7):900–912, 2004.

[128] C. Stewart, R. Keller, R. Repasky, M. Hess, D. Hart, M. Muller, R. Sheppard, U. Wossner, M. Aumuller, H. Li, D. Berry, and J. Colbourne. A Global Grid for Analysis of Arthropod Evolution. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 328–337, 2004.

[129] R. Sudarsan and C. Ribbens. ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 44, 2007.

[130] H. Takemiya, Y. Tanaka, S. Sekiguchi, S. Ogata, R. Kalia, A. Nakano, and P. Vashishta. Sustainable Adaptive Grid Supercomputing: Multiscale Simulation of Semiconductor Processing across the Pacific. In *Proceedings of the ACM/IEEE Supercomputing Conference (SC 2006)*, 2006.

[131] V. Taylor, X. Wu, J. Geisler, X. Li, Z. Lan, M. Hereld, I. Judson, and R. Stevens. Prophesy: Automating the Modeling Process. In *Proceedings of the Third Annual International Workshop on Active Middleware Services*, pages 3–11, Tokyo, Japan, August 2001.

[132] V. Taylor, X. Wu, Jonathan Geisler, and Rick Stevens. Using Kernel Couplings to Predict Parallel Application Performance. In *Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11*, 2002.

[133] V. Taylor, X. Wu, and Rick Stevens. Prophesy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):13–18, March 2003.

[134] TeraGrid. *http://www.teragrid.org*.

[135] TOP500 Supercomputing Sites. *http://www.top500.org*.

[136] Denis Trystram. Scheduling parallel applications using malleable tasks on clusters. *IPDPS*, 03, 2001.

[137] UK e-Science. *http://www.rcuk.ac.uk/escience/default.htm*.

[138] S. Vadhiyar and J. Dongarra. SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. *Parallel Processing Letters*, 13(2):291–312, 2003.

[139] S. Vadhiyar and J. Dongarra. GrADSolve: a Grid-based RPC System for Parallel Computing with Application-level Scheduling. *Journal of Parallel and Distributed Computing*, 64(6):774–783, 2004.

[140] S. Vadhiyar and J. Dongarra. Self Adaptivity in Grid Computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, 2005.

[141] Sathish S. Vadhiyar and Jack J. Dongarra. A performance oriented migration framework for the grid. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, Washington, DC, USA, 2003.

[142] G. Dick van Albada, J. Clinckmaillie, A. H. L. Emmen, Jörn Gehring, O. Heinz, Frank van der Linden, Benno J. Overeinder, Alexander Reinefeld, and Peter M. A. Sloot. Dynamite - blasting obstacles to parallel cluster computing. In *HPCN Europe '99: Proceedings of the 7th International Conference on High-Performance Computing and Networking*, pages 300–310, London, UK, 1999.

[143] K. van der Raadt, Y. Yang, and H. Casanova. Practical Divisible Load Scheduling on Grid Platforms with APST-DV. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, page 29.2, 2005.

[144] Jon B. Weissman and Xin Zhao. Scheduling parallel applications in distributed networks. *Cluster Computing*, 1(1):109–118, 1998.

[145] Von Welch, Frank Siebenlist, Ian Foster, John Bresnahan, Karl Czajkowski, Jarek Gawor, Carl Kesselman, Sam Metier, Laura Pearlman, and Steven Tuecke. Security for grid services. In *In Twelth International Symposium on High Performance Distributed Computing (HPDC-12*, pages 48–57, 2003.

[146] R. Wolski. Dynamically Forecasting Network Performance using the Network Weather Service. *Journal of Cluster Computing*, 1(1):119–132, 1998.

[147] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, October 1999.

[148] G. Wrzesinska, J. Maassen, and H. Bal. Self-adaptive Applications on the Grid. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 121–129, 2007.

[149] G. Wrzesinska, R. van Nieuwpoort, J. Maassen, and H. Bal. Fault-Tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, page 13.1, 2005.

[150] Z. Xu, X. Zhang, and L. Sun. Semi-empirical Multiprocessor Performance Predictions. *Journal of Parallel and Distributed Computing*, 39(1):14–28, 1996.

[151] J. Yagnik, H. A. Sanjay, and S. Vadhiyar. Performance Modeling based on Multidimensional Surface Learning for Performance Predictions of Parallel Applications in Non-Dedicated Environments. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 513–522, 2006.

[152] Y. Yan, X. Zhang, and Y. Song. An Effective and Practical Performance Prediction Model for Parallel Computing on Nondedicated Heterogeneous NOW. *Journal of Parallel and Distributed Computing*, 38(1):63–80, 1996.

[153] Y. Yang, K. van der Raadt, and H. Casanova. Multiround Algorithms for Scheduling Divisible Loads. *IEEE Transactions on Parallel and Distributed Systems*, 16(11):1092–1102, 2005.

[154] A. Yarkhan and J. Dongarra. Experiments with Scheduling Using Simulated Annealing in a Grid Environment. In *Lecture notes in computer science 2536 Grid Computing - GRID 2002*, volume Third International Workshop, pages 232–242, November 2002.

[155] Zhifeng Yu and Weisong Shid. An Adaptive Rescheduling Strategy for Grid Workflow Applications. In *Proceedings of the Parallel and Distributed Processing Symposium, 2007*, pages 1–8, 2007.

[156] X. Zhang, J. Freschl, and J. Schopf. A Performance Study of Monitoring and Information Services for Distributed Systems. In *HPDC '03: Proceedings of the 12th IEEE International Symposiumon High Performance Distributed Computing*, page 270, 2003.

[157] Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Softw. Pract. Exper.*, 23(12):1305–1336, 1993.