Long-running Multi-component Climate Applications on Grids

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

By
Sivagama Sundari M.

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

October 2010
Firstly, I would like to extend my sincere thanks to my research supervisors Dr. Sathish Vadhiyar and Prof. Ravi Nanjundiah for their constant guidance, involvement, encouragement and assistance that has made this work and thesis possible. I am especially grateful to Dr. Sathish Vadhiyar, for his highly systematic, organized approach and high involvement in the work that has ensured consistent progress of my work through the years resulting finally in the completion of my thesis. I also express my heartfelt gratitude to Prof. Ravi for his patient, cheerful and encouraging presence in all our weekly meetings, and for his supportive guidance with valuable suggestions.

I would also like to warmly thank the chairman of SERC, Prof. Govindarajan, for his timely and consistent support in all matters pertaining to my studentship at SERC and IISc, and for providing access to the SERC computing facilities. I also thank all the staff and students of SERC for the help they have rendered in various ways over the last few years.

I thank Dr. Subrata Chattopadhyay of C-DAC, Bangalore for granting access to the C-DAC systems through the GARUDA Grid network. I also thank Mr. Kiran, SERC, for his patient assistance with the HPC systems of SERC. My thanks to Sivaprasad, Jahfer Sharif and Suraj Harshan of Prof. Vinay’s lab in CAOS for helping me with NCL. I would also like to thank my labmate, Karthikeyan R., for his code contributions to my batch queue simulator.

I also take this opportunity to express my admiration for all those who have been involved directly or indirectly in the development of CCSM (and other such large-scale community climate models).

I am also, in general, extremely thankful to all those who have ever interacted with me over running CCSM, for being aware of the nature and the complexity of the application, and
hence providing me with the solace of their appreciation of the effort that has gone into my CCSM-related work.

Naturally, my thanks to ALL my labmates for all the interactions I have had with them that made working in the lab a pleasurable (and occasionally intolerable, yet fondly recollected) experience. And to all my other friends, I would like to express my happiness and relief they are still my friends.

Finally, I thank my family -parents, sister, husband and in-laws- for all their love, understanding, care and support.
Abstract

Climate science or climatology is the scientific study of the earth’s climate, where climate is the term representing weather conditions averaged over a period of time. Climate models are mathematical models used to quantitatively describe, simulate and study the interactions among the components of the climate system - atmosphere, ocean, land and sea-ice. CCSM (Community Climate System Model) is a state-of-the-art climate model, and a long-running coupled multi-component parallel application involving component models for simulating the components of the climate system. Each of the component models is a large-scale parallel application, and the parallel components exchange climate data through a specialized component called coupler. Typical multi-century climate simulations using CCSM take several weeks or months to execute on most parallel systems.

In this thesis, we study the applicability of a computational grid for effective execution of long-running coupled multi-component climate applications like CCSM. Initial studies of the application characteristics led us to develop a dynamic component extension strategy for temporal inter-component load-balancing. By means of experiments on different parallel platforms with different number of processors, we showed that using our strategy can lead to about 15% reduction and savings of several days in execution times of CCSM for 1000-year simulation runs. Our initial studies also indicated that unlike typical grid applications, CCSM has limits on scalability to very large number of processors and hence cannot directly benefit from the large number of processors on a computational grid. However, its long-running nature and the limits of execution imposed on jobs on most multi-user batch queueing systems, led us to investigate the benefits of its execution on a grid of batch systems. The idea is that multiple batch queues can improve the processor availability rate with respect to the application thereby possibly im-
proving its effective throughput. We explored this idea in detail with simulation studies involving various system and application characteristics, and execution models. By conducting large number of simulations with different workload characteristics and queuing policies of the systems, processor allocations to components of the application, distributions of the components to the batch systems and inter-cluster bandwidths, we showed that multiple batch executions lead to upto 55% average increase in throughput over single batch executions for long-running CCSM. Having convinced ourselves of possible advantages in performance, we then ventured to construct an application-level middleware framework.

Our framework supports long duration execution of multi-component applications spanning multiple submissions to queues on multiple batch systems. It coordinates the distribution, execution, rescheduling, migration and restart of the application components across resources on different sites. It also addresses challenges including execution time limits for jobs, and differences in job-startup times corresponding to different components. Further, within the framework, we developed robust rescheduling policies that decide when and where to reschedule the components to the available resources based on the application execution characteristics and queue dynamics. Our grid middleware framework resulted in multi-site executions that provided larger application throughput than single-site executions, typically performed by climate scientists, and also removed the bottlenecks associated with a single system execution.

We used this framework for long-running executions of CCSM to study the effect of increased black carbon aerosols and dust aerosols on the Indian monsoons. Black Carbon aerosols are essentially of anthropogenic origin and occur due to improper burning of fossil fuels, and dust is a naturally occurring aerosol. The concentrations of both these aerosols is high over the Indian region. We study the impact of these aerosols on precipitation and sea surface temperature (SST) through multi-decadal simulations conducted with our grid-enabled climate system model. Our observations indicated that increasing the concentrations of aerosols leads to an increase in precipitation in the central and easter parts of India, and a decrease in SST over most of Indian ocean.
Publications


# Contents

Abstract iii  

List of Figures xi  

List of Tables xvi  

1 Introduction 1  

1.1 Motivation .................................................. 2  

1.1.1 The Computer Systems Perspective ......................... 2  

1.1.2 The Climate Science Perspective .......................... 5  

1.2 CCSM on Grids: Challenges Involved .......................... 7  

1.3 Problem Statement ........................................... 8  

1.4 Load Balancing ................................................ 9  

1.5 Simulation Studies ............................................. 11  

1.6 Robust Middleware Framework ................................ 13  

1.7 Rescheduling .................................................. 15  

1.8 Climatology Application ...................................... 16  

1.9 Organization of the Thesis .................................... 17  

2 Related Work 19  

2.1 Climate Science and Grid Computing ........................... 19  

2.2 Load Balancing in Multi-Component Applications ............. 21  

2.2.1 Coupling Strategies in Multi-Component Applications .... 22
3 Load Balancing CCSM: Dynamic Component Extension

3.1 Introduction .................................................. 36
3.2 Temporal Load Imbalance and Dynamic Component Extension .......... 40
3.3 Load Balancing Challenges in CCSM ................................ 42
   3.3.1 Inter-component load imbalance due to temporal load-imbalance in atmosphere ........................................... 43
   3.3.2 Intra-component load-imbalance due to short-wave radiation calculations ................................................. 44
3.4 Dynamic Extension of Atmosphere Component ............................ 46
   3.4.1 Time-step Level ........................................... 49
   3.4.2 Chunk level .................................................. 50
   3.4.3 Column level .................................................. 51
   3.4.4 Summary of Dynamic Component Extension in CCSM ................. 56
3.5 Experiments and Results ......................................... 57
3.6 Summary .......................................................... 70

4 Simulation Studies

4.1 Introduction .................................................. 72
4.2 Batch Grids - Motivations and Contributions ............................. 77
4.3 Simulation Methodology for MCA and PSA Simulation Studies .......... 78
   4.3.1 Workload Generator ........................................... 78
   4.3.2 Batch System Simulator ......................................... 79
   4.3.3 Application Simulator .......................................... 80
   4.3.4 Statistics Calculator ............................................. 82
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4 Simulation Setup for MCA and PSA Simulation Studies</td>
<td>86</td>
</tr>
<tr>
<td>4.5 Results of MCA and PSA Simulations</td>
<td>87</td>
</tr>
<tr>
<td>4.5.1 Simulations of Multi Component Applications</td>
<td>88</td>
</tr>
<tr>
<td>4.5.2 Real Executions of Multi Component Applications</td>
<td>91</td>
</tr>
<tr>
<td>4.5.3 Simulations of Parameter Sweep Applications</td>
<td>94</td>
</tr>
<tr>
<td>4.5.4 Summary of Results</td>
<td>98</td>
</tr>
<tr>
<td>4.6 Multiple-Batch Execution Policies for Multi Component Applications</td>
<td>99</td>
</tr>
<tr>
<td>4.7 Evaluation of Various Execution Models</td>
<td>102</td>
</tr>
<tr>
<td>4.8 Best Execution Model and Evaluation Methodology for CCSM</td>
<td>106</td>
</tr>
<tr>
<td>4.9 Analytical Model for RAR</td>
<td>111</td>
</tr>
<tr>
<td>4.10 Methodology for CCSM Performance Studies</td>
<td>112</td>
</tr>
<tr>
<td>4.10.1 Application Simulator</td>
<td>115</td>
</tr>
<tr>
<td>4.10.2 CCSM Scheduler/Rescheduler</td>
<td>118</td>
</tr>
<tr>
<td>4.11 Experiments and Results for CCSM Performance Studies</td>
<td>121</td>
</tr>
<tr>
<td>4.11.1 Experiment setup</td>
<td>121</td>
</tr>
<tr>
<td>4.11.2 Queue Waiting Times of CCSM Jobs</td>
<td>122</td>
</tr>
<tr>
<td>4.11.3 Resource Availability for CCSM Execution</td>
<td>122</td>
</tr>
<tr>
<td>4.11.4 Application Throughput</td>
<td>124</td>
</tr>
<tr>
<td>4.12 Summary</td>
<td>134</td>
</tr>
<tr>
<td>5 Morco: Grid Middleware Framework</td>
<td>135</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>135</td>
</tr>
<tr>
<td>5.2 Morco - Grid Middleware Framework</td>
<td>139</td>
</tr>
<tr>
<td>5.2.1 Coordinator</td>
<td>139</td>
</tr>
<tr>
<td>5.2.2 Job Monitor</td>
<td>141</td>
</tr>
<tr>
<td>5.2.3 Job Submitter</td>
<td>142</td>
</tr>
<tr>
<td>5.2.4 MCA Job Script</td>
<td>142</td>
</tr>
<tr>
<td>5.2.5 Component Interactions</td>
<td>143</td>
</tr>
<tr>
<td>5.2.6 CCSM Restarts, Packing and Synchronizing Communications</td>
<td>145</td>
</tr>
<tr>
<td>5.2.7 Salient Features of Morco</td>
<td>147</td>
</tr>
</tbody>
</table>
5.3 Genetic Algorithm for Scheduling and Rescheduling ............................. 148
5.4 Fault Tolerance ............................................................................. 150
5.5 Experiments and Results ................................................................. 151
5.6 Discussion ................................................................................. 160
5.7 Conclusions .............................................................................. 161

6 Performance-Oriented Rescheduling ................................................ 162
   6.1 Introduction ............................................................................. 162
   6.2 Modifications in Execution Model and Framework ...................... 163
   6.3 Adaptive Rescheduling Algorithm ........................................... 164
   6.4 Rescheduling Decision Cases ................................................... 171
   6.5 Experiments and Results .......................................................... 182
   6.6 Conclusions and Future Work .................................................... 191

7 Climatology: Effect of Doubling Aerosols ....................................... 192
   7.1 CCSM: Model Details ............................................................... 192
   7.2 Black carbon and dust aerosols ............................................... 193
   7.3 Morco: Grid Middleware .......................................................... 197
   7.4 Validation of Precipitation ......................................................... 197
   7.5 Impact on Indian Monsoon ....................................................... 202
   7.6 Impact on Sea Surface Temperature ......................................... 208
   7.7 Conclusions ............................................................................ 209

8 Conclusions and Future Work ......................................................... 213
   8.1 Conclusions ............................................................................ 213
   8.2 Future Work .......................................................................... 216

References ....................................................................................... 218
List of Figures
1.1

Community Climate System Model (CCSM3): Multi-component Interactions .

6

3.1

Sequential, Concurrent and Hybrid Coupling . . . . . . . . . . . . . . . . . .

37

3.2

Component Extension . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

39

3.3

Effect of (a) Temporal Load-Imbalance and (b) Component Extension on Coupled Application Execution . . . . . . . . . . . . . . . . . . . . . . . . . . . .

3.4

41

Computation intensive Absorptivity Calculations in Atmosphere and the corresponding idling in Land . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

45

3.5

Load-imbalance due to radcswmx . . . . . . . . . . . . . . . . . . . . . . . .

46

3.6

Pseudo-code for radiation calculations . . . . . . . . . . . . . . . . . . . . . .

47

3.7

Calculation of time-step sharing index . . . . . . . . . . . . . . . . . . . . . .

50

3.8

Calculation of chunk sharing index . . . . . . . . . . . . . . . . . . . . . . . .

51

3.9

Column-Processor Mapping Algorithm . . . . . . . . . . . . . . . . . . . . .

54

3.10 Pseudo-code for Modified Atmosphere . . . . . . . . . . . . . . . . . . . . . .

55

3.11 Pseudo-code for Modified Non-Atmosphere Component . . . . . . . . . . . .

56

3.12 Reduction of Atmosphere Computation Times due to DCE . . . . . . . . . . .

57

3.13 Results on Fire Cluster for Low Resolution . . . . . . . . . . . . . . . . . . .

60

3.14 Results on CDAC Regatta for Low Resolution . . . . . . . . . . . . . . . . . .

62

3.15 Results on CAOS Regatta for Low Resolution . . . . . . . . . . . . . . . . . .

63

3.16 Results on CDAC Param for Low Resolution . . . . . . . . . . . . . . . . . .

64

3.17 Effect of radiation frequency on DCE gain. Fire-16 cluster. . . . . . . . . . . .

67

3.18 Overheads and Gain due to DCE. IISc Regatta cluster. . . . . . . . . . . . . . .

68

xi


<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.19</td>
<td>Fraction of gain harnessed using dynamic component extension on fire-16 cluster</td>
<td>69</td>
</tr>
<tr>
<td>3.20</td>
<td>Simulation results showing effect of scalability in dynamical core on DCE gain</td>
<td>70</td>
</tr>
<tr>
<td>4.1</td>
<td>Average queue wait times for jobs on two SDSC systems</td>
<td>74</td>
</tr>
<tr>
<td>4.2</td>
<td>Simulation Framework</td>
<td>79</td>
</tr>
<tr>
<td>4.3</td>
<td>Fraction of Configurations with Probabilities of Benefits for CCSM Simulations</td>
<td>88</td>
</tr>
<tr>
<td>4.4</td>
<td>Probability of Benefits with Multiple Batch Submissions for all Application and System Configurations for CCSM Simulations</td>
<td>90</td>
</tr>
<tr>
<td>4.5</td>
<td>Average Probability of Benefits with Multiple Batch Submissions for Different System Configurations for CCSM Simulations</td>
<td>91</td>
</tr>
<tr>
<td>4.6</td>
<td>Percentage of Configurations with Probabilities of Benefits for CCSM Real Executions</td>
<td>92</td>
</tr>
<tr>
<td>4.7</td>
<td>Probability of Benefits with Multiple Batch Submissions for all Application and System Configurations for CCSM Real Executions</td>
<td>93</td>
</tr>
<tr>
<td>4.8</td>
<td>Average Probability of Benefits with Multiple Batch Submissions for Different System Configurations for CCSM Real Executions</td>
<td>94</td>
</tr>
<tr>
<td>4.9</td>
<td>Percentage of Configurations with Probabilities of Benefits for PSA Simulations</td>
<td>95</td>
</tr>
<tr>
<td>4.10</td>
<td>Probability of Benefits with Multiple Batch Submissions for all Application and System Configurations for PSA Simulations</td>
<td>96</td>
</tr>
<tr>
<td>4.11</td>
<td>Average Probability of Benefits with Multiple Batch Submissions for Different System Configurations for PSA Simulations</td>
<td>97</td>
</tr>
<tr>
<td>4.12</td>
<td>Simulation Setup</td>
<td>102</td>
</tr>
<tr>
<td>4.13</td>
<td>Percentage of cases in which each mode outperforms the other</td>
<td>104</td>
</tr>
<tr>
<td>4.14</td>
<td>Durations of different number of active sites involved in MCA execution</td>
<td>105</td>
</tr>
<tr>
<td>4.15</td>
<td>Execution of a long-running two-component application on two sites</td>
<td>108</td>
</tr>
<tr>
<td>4.16</td>
<td>Formal Description of the Multi-site Execution Model</td>
<td>110</td>
</tr>
<tr>
<td>4.17</td>
<td>Simulator Architecture</td>
<td>113</td>
</tr>
<tr>
<td>4.18</td>
<td>CCSM Execution Flow</td>
<td>115</td>
</tr>
<tr>
<td>4.19</td>
<td>Queue Waiting Times for CCSM Jobs</td>
<td>123</td>
</tr>
<tr>
<td>4.20</td>
<td>RARs for CCSM Jobs in Single and Multiple Batch Executions</td>
<td>125</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.10</td>
<td>Rescheduling Decision Case5</td>
<td>174</td>
</tr>
<tr>
<td>6.11</td>
<td>Rescheduling Decision Case6</td>
<td>175</td>
</tr>
<tr>
<td>6.12</td>
<td>Rescheduling Decision Case7</td>
<td>175</td>
</tr>
<tr>
<td>6.13</td>
<td>Rescheduling Decision Case8</td>
<td>176</td>
</tr>
<tr>
<td>6.14</td>
<td>Rescheduling Decision Case9</td>
<td>176</td>
</tr>
<tr>
<td>6.15</td>
<td>Rescheduling Decision Case10</td>
<td>177</td>
</tr>
<tr>
<td>6.16</td>
<td>Rescheduling Decision Case11</td>
<td>179</td>
</tr>
<tr>
<td>6.17</td>
<td>Rescheduling Decision Case12</td>
<td>180</td>
</tr>
<tr>
<td>6.18</td>
<td>Rescheduling Decision Case13</td>
<td>180</td>
</tr>
<tr>
<td>6.19</td>
<td>Rescheduling Decision Case14</td>
<td>181</td>
</tr>
<tr>
<td>6.20</td>
<td>Rescheduling Decision Case15</td>
<td>181</td>
</tr>
<tr>
<td>6.21</td>
<td>Execution profile of the application components on multiple sites</td>
<td>184</td>
</tr>
<tr>
<td>6.22</td>
<td>Execution profile of the application components on singesite</td>
<td>185</td>
</tr>
<tr>
<td>6.23</td>
<td>Rescheduling and No-rescheduling decision points</td>
<td>185</td>
</tr>
<tr>
<td>6.24</td>
<td>Comparison of application progress with single-site execution</td>
<td>186</td>
</tr>
<tr>
<td>6.25</td>
<td>Percentage of Time Spent in Different Number of Active Queues in the Multi-</td>
<td>187</td>
</tr>
<tr>
<td></td>
<td>site Executions with Greedy Rescheduling and Adaptive Rescheduling</td>
<td></td>
</tr>
<tr>
<td>6.26</td>
<td>Best execution rates for CCSM when executed across certain sets of Active</td>
<td>188</td>
</tr>
<tr>
<td></td>
<td>Queues</td>
<td></td>
</tr>
<tr>
<td>6.27</td>
<td>Percentage of Time Spent in Different Queues in the Multi-site Execution</td>
<td>188</td>
</tr>
<tr>
<td></td>
<td>with Adaptive Rescheduling</td>
<td></td>
</tr>
<tr>
<td>6.28</td>
<td>Percentage of Time Spent in Idling, Computing and Overheads</td>
<td>190</td>
</tr>
<tr>
<td>7.1</td>
<td>Distribution of atmospheric black carbon aerosols (July, JJAS and Annual)</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td>over the tropics (in g/m2)</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Distribution of atmospheric dust aerosols (July, JJAS and Annual) over the</td>
<td>195</td>
</tr>
<tr>
<td></td>
<td>tropics (in g/m2)</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Validation: tropics precipitation (JJAS)</td>
<td>198</td>
</tr>
<tr>
<td>7.4</td>
<td>Validation: tropics precipitation (July)</td>
<td>199</td>
</tr>
<tr>
<td>7.5</td>
<td>Validation: Indian region precipitation (JJAS)</td>
<td>200</td>
</tr>
</tbody>
</table>
7.6 Validation: Indian region precipitation (July) ........................................ 201
7.7 Impact of doubled aerosols on precipitation (JJAS) ................................. 203
7.8 Impact of doubled aerosols on precipitation (July) .................................. 204
7.9 Statistically significant differences in precipitation .................................. 205
7.10 Standard deviation for monthly precipitation rate (July) ......................... 206
7.11 Impact of doubled aerosols on area-averaged rainfall ........................... 207
7.12 Impact of doubled aerosols on SST (JJAS) ........................................... 210
7.13 Impact of doubled aerosols on SST (July) ............................................ 211
7.14 Statistically significant differences in SST ............................................. 212
List of Tables

3.1 Platform Specifications ........................................... 58
3.2 Model Restrictions on Number of Processors .................... 59
3.3 Results for High Resolution Runs ................................. 65
3.4 Results on Various Platforms .................................... 65
3.5 Results for the 16-processor configuration (Atmosphere-8,Ocean-4,Ice-2,Land-1,Coupler-1) on Various Platforms ................ 66
Chapter 1

Introduction

Computational science, involving mathematical modeling and computer simulations of physical systems, is being considered as the third mode of science after experimentation and theory. This is specially true for applications in areas like weather and climate science, astronomy, geosciences, high energy physics, bioinformatics, etc. where state-of-the-art research is predominantly computational. Further, these large scientific applications have an ever increasing computational demand, that led to the evolution of high performance computing systems from massively parallel multi-core systems installed at single geographical locations to the geographically distributed and multi-institutional computational grids that connect these powerful individual parallel systems.

A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [62]. This important paradigm of computational grids that can provide the applications with greater computational power than even the most powerful parallel system, is now being increasingly used for executing various large scale scientific applications. Some of these applications are developed from scratch for execution on a grid environment, just as parallel programs (for example, using the Message Passing Interface, MPI) are written for execution on a parallel environment. However, rewriting large and important legacy scientific applications for executions on grids can be a time-consuming and error-prone process. Hence, such applications are “grid enabled” starting from their current state.
Our work explores the execution of multi-component and coupled climate simulation applications on grids. These simulations are extremely important because they help us determine the impact of human activities on the earth’s future climate and resolve issues like global warming. Hence, they are of interest not only to climate scientists, but also to environmentalists, public policy makers and the public in general. Grid computing, as a technology providing access to high-end computational capabilities, could hence facilitate these computation intensive, high fidelity climate simulations and thereby, contribute to resolving questions of global importance.

This thesis presents the range of challenges involved, explored and addressed, including porting the application, studying its characteristics and requirements, developing execution models, evaluating performance advantages, developing middleware framework and finally, executing the application on a computational grid.

1.1 Motivation

This section aims to help the reader understand the importance of this interdisciplinary research work from two different perspectives-the computer systems perspective and the climate science perspective. It also gives a broad overview of the problem and the work done.

1.1.1 The Computer Systems Perspective

Grid computing enables computing on multi-institutional, dynamic, heterogeneous resource pools, virtually expanding the bounds of resources available to scientific applications. While one of grid computing’s primary use is to enable remote use of high performance computing hardware and software resources for problem solving, applications can benefit from execution on a grid system in various ways depending upon the application and the system characteristics.

A grid application that involves a large number of independent computational tasks, classified as a “bag-of-tasks” application, can be executed on a grid comprising of a very large number of volunteered computing resources widely distributed across Internet. This type of grid is also known as a ”desktop grid”. Examples for desktop grid applications include projects like SETI@Home[134], cancer@home[23] and climateprediction.net[33]. The Berkeley Open
Infrastructure for Network Computing, BOINC[11], is an open-source software platform for such computing using volunteered resources. The contribution of grid to such problems is feasibility, i.e. execution on grid makes it possible for such huge problems to be solved.

For some workflow applications that involve the flow of data across multiple processing levels or stages that are restricted to different geographical locations, the multi-institutional grid is a natural requirement. A workflow has been defined in [63] as “the automation of the processes, which involves the orchestration of a set of Grid services, agents and actors that must be combined together to solve a problem or to define a new service.” Most workflow applications[41, 40] comprise of a series of processes, involve multiple grid services that must be combined together, or have multiple data-parallel concurrent actors. For example, a popular workflow application, Montage[72], that creates custom image-mosaics out of existing astronomical images, involves multiple stages of processing starting from the initial set of images. Some processing levels in workflow are data-parallel involving multiple component tasks each corresponding to one of the input images. Yet another workflow example is the mesoscale weather prediction in LEAD-VGrADS[131]. This involves grid-services based workflows with data-streaming and a number of pre-processing steps that feed data to a computation intensive weather model. Further an ensemble of short runs of weather models are invoked with various parameters.

For yet other applications, grids have been used to provide feasibility to solve much larger problem sizes than those that can be solved on a single site[32, 109, 44, 53]. Execution on grids also enables exploration of a larger search space of different parameters of a problem[3, 65]. The large number of processors on a grid is used to improve the quality of the solutions[140, 74] as well as to enhance application performance[32, 109, 44]. Thus, grid platforms are useful to different applications in different ways, with the effectiveness of the integration of the application in grids depending upon various application characteristics as well as grid characteristics.

While workflow and bag-of-task applications have been extensively studied for execution on various types of grids, not much work has been done on the effectiveness of grids for long-running multi-component applications. Multi-component coupled Multiple Program Multiple Data (MPMD) applications (also called “coupled multi-physics” applications) are large scien-
tific applications comprising of interacting component applications. Such applications have be-
come prevalent since multi-disciplinary multi-component models are used to accurately model
interacting physical processes or phenomena in the areas of climate, space weather, solid rock-
ets, fluid-structure interaction, heart disease and cancer studies[36, 86, 42]. In these applica-
tions, each component is typically a model of one of the physically interacting subsystems of
a large physical system that is being modeled. Different components are developed by sepa-
rate scientific teams with corresponding expertise. These components are then brought together
and “coupled”, thus forming a multi-component application with interactions. For example,
the different components of CCSM that model the different climate subsystems have been de-
developed independently, and some like the atmosphere component model, CAM, and the ocean
component model, POP, are even prevalently run as standalone applications. Many of these
applications, used to model evolution of physical systems with time, are also long-running. For
example, assessment of climate change needs to be done far into the future and requires long
simulations. Such simulations would require computational time running of the order of weeks
(if not months). Such long-running executions cannot be performed on a single parallel system
since a single system cannot be guaranteed for long durations. Hence it is essential to execute
such long running simulations on multiple parallel systems of a grid. Integrating such applica-
tions in grids is also important for large-scale deployment and use of scientific applications on
grid systems by the scientific community.

Many current large-scale grid frameworks[142, 145] consist of multiple distributed sites
with each site having one or more batch systems with corresponding batch scheduling and
queuing policies. Executing long-running coupled applications on such grids with multiple
powerful batch systems is therefore highly essential for providing high performance for the
applications (For brevity, we refer to such grids as batch grids). Parallel batch systems provide
space sharing of available processors among multiple parallel applications or jobs. These batch
systems employ queues in which the incoming parallel applications are queued before allocation
by a batch scheduler to a set of processors for execution.

Tightly-coupled or single-component parallel applications involve heavy communications
and synchronizations among the parallel tasks or processes. Decomposing such applications
into smaller parallel applications and submitting to parallel systems located at different sites will result in frequent and heavy communications, and hence performance degradation. Multi-component MPMD applications consist of components which are parallel applications themselves, and in these applications, the components are loosely coupled and communications between components are lighter and less periodic than within components. Such applications therefore have a potential for multiple-site executions, which we seek to study and explore in our research.

1.1.2 The Climate Science Perspective

In order to understand and predict the behavior of the earth’s climate system, climatologists model the physical processes involved as a set of mathematical equations. These equations are based on the most fundamental laws of physics such as the laws of conservation of energy, mass and momentum. Numerical computer simulations of climate using these models have been some of the earliest and most significant scientific computer applications, successfully demonstrated in as early as the 1950s[94]. With various related scientific and technological advancements, climate models have evolved through the decades, with the state-of-the-art being the coupled atmosphere-ocean general circulation models (AOGCMs), an example of which is CCSM (Community Climate System Model), the climate application we are considering in our work. Other examples include HadCM3 (Hadley Centre Coupled Model)[73] and GFDL CM2.X (Geophysical Fluid Dynamics Laboratory Coupled Model)[67].

Community Climate System Model (CCSM)[29] is a coupled global climate model developed at National Center for Atmospheric Research (NCAR), with contributions from external research groups, funded by the National Science Foundation, Department of Energy (DOE) and National Aeronautics and Space Administration (NASA). It is a multi-component application and consists of four climate sub-system models, namely atmosphere (CAM), ocean (POP), land (CLM) and ice (CSIM), and a coupler (CPL) component through which these models interact (Figure 1.1). We used the most recent version of CCSM that was available during the period
of our research, CCSM3 (beta version)\(^1\). It is implemented as a multiple program multiple data (MPMD) programming model where each component is a separate parallel program by itself.

The CCSM code base comprises of more than a thousand FORTRAN module files. CCSM uses several libraries including some that are part of the CCSM code: ESMF[52], MPH[20] and MCT[86]; and other external libraries like MPI[107] and NetCDF[117]. Porting CCSM to a new, untested system is a challenging and time-consuming process, involving various steps including consistent and stable builds of all software libraries involved and all model components, and performing various tests recommended by the developers. A validated installation of CCSM on a system available for use by several climate scientists is in itself a contribution to the community, as it enables these researchers to perform high-resolution global simulations. Grid enabling the application makes it accessible to a larger number of researchers, whose only access to high performance computing systems is through the grid.

CCSM is a highly computationally intensive application. A typical simulation run could span a few years to a few thousand years of the earth’s climate depending upon the scientific issue to be studied, while the simulation time-steps are typically less than an hour. Such simulations could take a few days to a few weeks to execute on most of today’s multi-processor systems.

\(^1\)A more recent version CCSM4 has been released since. Apart from the concurrent execution model followed in CCSM3, CCSM4 also supports sequential and hybrid models.
systems. Moreover, as there are continuous research efforts to increase the numerical grid resolutions and include more scientific processes, the computational demands of the application are expected to increase in the near future. Therefore, it is very important to explore the possibility of significantly reducing simulation times by executing the application on grid systems.

However, execution of coupled climate models like CCSM on grid environments involves a large number of research challenges. The next section describes these challenges.

1.2 CCSM on Grids: Challenges Involved

There are several challenges involved in execution of CCSM on a grid system. The first and the most basic among them is the installation and testing of CCSM (and all underlying libraries) on every system (site) involved. Porting of a legacy application like CCSM on a new platform is an extremely challenging effort requiring several modifications to suit the available hardware and software resources. Once the application is successfully running on each site of the grid, it has to be modified such that different components of the model can execute on different sets of parallel resources on the Grid.

Another important challenge is in the area of performance studies, modeling and optimization of the application (CCSM). In order to improve the performance of the application, it is important to first profile its execution and look at the times spent in various functions or modules of the code. Since execution of an application on a grid involves execution across various computing platforms with different processing speeds and communication across various network connections with different communication bandwidths, it is important to model the application performance as functions of these variables in order to leverage better performance on the grid.

Further, since CCSM is a multi-component application with malleable components that can be executed on a variable number of processors, another challenge that needs to be addressed is the dynamic load balancing of climate components based on load changes on the resources and progression rates of different components. This involves optimal assignment and migration/re-assignment of different components of the climate model to the different available resources of the Grid based on various resource and network dynamics.
As mentioned in the previous section, we are considering batch grids for execution of CCSM. However, since component jobs submitted to multiple independently administered batch systems would be scheduled for execution at different points of time, there are various possibilities of execution including. For example, two such possibilities are: (i) waiting for resources to be allocated on all systems before starting the execution of the application, and (ii) not waiting at all and using all allocated resources dynamically remapping the application. Other issues including execution time limits also have to be addressed. Hence, development of a suitable execution model for execution of a long-running multi-component application on multiple batch systems is of utmost importance.

Grid executions of climate models involve development of application-level grid middleware for dynamic rescheduling and migration of components as resource conditions in Grid change. This involves various challenges:

1. Discovery of new changes in availability of resources at various locations: Since in a grid the resources become available and unavailable dynamically at various points of time, these events have to be detected so that appropriate action can be taken with respect to the application execution.

2. Distributed checkpointing mechanisms: Application components executing at multiple locations have to be simultaneously checkpointed. Checkpointing is the process of storing the application’s current state so that it can be stopped and restarted to continue from that point of execution progress at a later point of time or on a different set of processes.

3. Formulating efficient rescheduling and migrating policies/decisions: Application may or may not have to be rescheduled upon a change in resource availability. Decisions including whether to reschedule and if yes, how to reschedule, have to be dynamically made.

1.3 Problem Statement

The objective of our research is to explore various facets of, and to address the challenges involved in the execution of long-running, multi-component, coupled climate modeling applications across multiple interconnected and independently administered batch systems. These
challenges include porting the application on multiple sites, studying its characteristics and requirements, developing execution models for execution across multiple independently administered batch systems, evaluating performance advantages of multi-site executions over single-site executions, developing middleware framework and finally, executing the application on a computational grid to solve real climate science problems.

We have addressed all the above challenges in our work and have presented them in this thesis in five parts. The first part covers the initial performance studies of the climate models, the various characteristic features of the climate components, the load imbalances that are present and our load balancing solution. In the second part, we propose and evaluate execution models for climate models on grids, and study the effectiveness of execution on grids through extensive simulations with various application and grid parameters. The next part describes our implementation of a robust middleware for execution of climate models across multiple sites. The fourth part describes the dynamic rescheduling policy for performance improvement of grid executions using the middleware. Finally, in the last part, we describe the climate problem we have solved using our middleware framework.

The following sections give an overview of these parts.

### 1.4 Load Balancing

During the execution of a parallel application, if the workload on one processor is much less than that on another, a significant part of the first processor is spent idling during the time the other processor is performing computations. An even distribution of workload, i.e. a load balanced application will ensure that all processors are meaningfully occupied throughout the execution, thereby ensuring an efficient execution resulting in the reduction of overall time for application execution.

Load imbalances in a multi-component application like coupled climate models can be of two types: intra-component (load imbalance across processors allocated to a component) and inter-component (load imbalance across sets of processors executing different components). Various efforts have dealt with load balancing in climate models\[99, 102, 46, 61, 59, 110, 111,\]
113, 114, 100]. Although all these efforts show improvements in execution times of a single component, they do not address problems related to a coupled climate system model involving load imbalances between multiple components, i.e. they all address only intra-component load imbalances.

There have been some efforts that involve the other components in load-balancing. These are based on determining the optimum static allocation of processors to components. These do not perform dynamic load-balancing during application execution considering dynamic application and resource characteristics. Hence these are not suitable for dynamic environments like long-running climate applications and grids. For instance, the CCSM Load Balancing Workshop document [24] outlines the general methodology to be followed for such load-balancing. The technique suggested is a trial and error method. The largest possible number of processors are allocated to atmosphere and the remaining processors are allocated to the other components such that the atmosphere is not delayed. Carr et. al. [25] suggest general strategies for processor mapping of the highest resolution models based on experiments with CCSM3 on the Cray X1. However, these solutions are static and do not handle the large dynamic imbalances that occur in certain time-steps of CCSM execution.

Our load balancing solution, called the Dynamic Component Extension (DCE), is aimed at resolving this dynamic temporal load-imbalance by online-profiling and dynamic offloading (or extension) of the computations from the processors executing the component with the larger workload to the idling processors assigned to the other components.

We apply our strategy to CCSM by dynamically extending the processor space of the atmosphere component (atmosphere processors) to include the idling processors executing the non-atmosphere components during the long-wave radiation calculations in the atmosphere. The extension characteristics including the amount and points of extension, and the processors of other components to include during the extension are dynamically determined based on the times when the non-atmosphere processors are ready to share work and the different times taken by the different atmosphere processors to start their long-wave radiation calculations. By means of experiments with different processor configurations for two different modeling resolutions on five multi-processor systems, we show that our dynamic component extension strategy can
lead to about 15% reduction on 16 processors and savings of up to 50 days in execution times of CCSM for 1000-year simulation runs.

Our solution resulted in a performance optimized version of CCSM for execution on a single parallel system. However, several other research and implementation challenges had to be addressed before CCSM could be executed on a batch grid. The experience of our work with load balancing showed us that CCSM was a complex application with various non-linear and dynamic features that cannot be modeled and expressed in simple mathematical forms. Also, we found that CCSM had restrictions on scalability to very large number of processors. Hence, it was imperative that we build models to simulate its execution and perform extensive simulation studies to investigate the effectiveness of a batch grid for its execution.

1.5 Simulation Studies

The second part of our work deals with study of various aspects of execution of climate components on a grid involving multiple batch systems. We refer to a grid of multiple interconnected, independently administered batch systems as a “batch grid”. Execution on a batch grid as opposed to execution on a single batch system can be advantageous due to two main reasons: (i) greater duration of availability of resources and (ii) availability of greater number of resources for execution of the application.

Parallel batch systems provide space-sharing of available processors among multiple parallel applications or jobs. These batch systems employ queues in which the incoming parallel applications are queued before allocation by a batch scheduler to a set of processors for execution. Hence, the overall response time of an application is the sum of its queue waiting time and execution time. In order to discourage some jobs from engaging resources for extremely long durations, most batch systems impose a maximum execution time limit per submission. Therefore, very long running applications like CCSM require multiple submissions and incur queue waiting times for each submission. When multiple batch systems are used, queue waiting (or inactive) periods of some systems would overlap with execution (or active) periods of other systems. Hence, if the application can be made to execute dynamically on all active resources,
the total period available for execution would be longer than the period available for execution on any individual resource. If inter-site execution of a coupled multi-component application can be enabled, with different components executing on different batch systems, then the total number of processors given to the application would be much larger than the number of processors available on any single site. Thus, when all queues are active, then the execution rate of the application can be higher than the execution rate on a single system.

However, the above performance benefits are subject to several application and system factors like inter-site bandwidth, application communication characteristics, application component scalabilities, queue scheduling policies, external workload on the queues, etc. We study the effects of these different factors on the potential improvement in performance of a parallel application due to simultaneous submissions of the parallel sub applications to multiple batch systems of a grid over submission of the entire parallel application to a single batch system. At the application-level, we used two simulators, one that models the most prominent multi-component application, CCSM[29, 38], and the other that models parameter-sweep applications. At the network-level, we simulated different interconnection speeds between the batch systems. At the batch-level, we used different job traces produced from Feitelson’s job models[93] and containing different distributions of jobs with different processor, execution time requirements and execution time limits. We then generated the queue waiting times of the jobs by using three different queuing policies, namely, FCFS, conservative and easy back-filling. We performed large number of simulations with different distributions of processors to components and system configurations with 24 different queues. We show that there are a large number of configurations for which performance improvements are obtained for the applications on batch grids. We further performed real experiments with CCSM by executing the climate components across two AMD Opteron clusters and show similar benefits.

Hence, we have built simulation frameworks involving simulators/models of application performance, workload, queue scheduling, dynamic execution, etc. to study the effects of these parameters on the performance benefits of multi-site execution and have used them to perform a large number of experiments to understand the effect of the various parameters on the gains.

The results of our simulation studies indicated high probabilities of upto 55% average in-
crease in throughput with execution of CCSM on a batch grid over execution of CCSM on a single batch system. Our studies also led us to propose an efficient execution model for such multi-site multi-submission executions. However, this required special middleware for coordination.

1.6 Robust Middleware Framework

The third part of our work involves the construction of Morco (Middleware framework for long-running multi-component applications), a middleware framework we developed for execution of coupled climate models on batch grids.

Our framework effectively addresses the various related challenges discussed in Section 1.2. Apart from coordinating the executions of the component jobs submitted independently to different batch systems, Morco also automatically resubmits the batch jobs corresponding to the components multiple times to the queues to continue and sustain long running executions. As the number of active batch systems available for execution can vary during execution, it performs dynamic migration and rescheduling of components.

Some of the salient features of our Morco framework are listed below:

1) **Long-running multi-component applications:** Our framework is built for multi-component applications that involve periodic communications between the components. Since the applications considered are long-running, it is reasonable to assume that they can be stopped and restarted either using in-built restart facilities or using an external checkpointing library. Since components of most coupled multi-component applications cannot be split, we perform distribution of the communicating components across the sites rather than splitting a component across sites.

2) **Multiple queues across multiple sites:** The framework can handle variable number of queues and clusters at different sites. It can handle multiple queues within the same cluster as well as queues across clusters. Its primary function is to coordinate execution of a single large long-running multi-component application through jobs submitted multiple times to multiple queues.
3) **Dynamic reconfiguration of application:** Our framework supports multiple reconfigurations of application within a submission corresponding to other submissions becoming active or inactive. It automatically detects the job submitted to queue becoming active and reconfigures the multi-component application to include the newly available resources. Similarly, it also automatically detects when a job in a queue is close to timeout and reconfigures the multi-component application restricting it to other active queues. It uses a genetic algorithm based scheduler to dynamically compute the configuration, i.e. the mapping of components to processors. This requires a performance model for predicting performance of the multi-component application across multiple sites. The genetic algorithm is described in the next section.

4) **Portability to Different Batch Systems:** At any instant there is exactly one job corresponding to our target application in each queue, and hence it does not unfairly affect the queue wait times of other external jobs in the queue. Our framework performs and monitors our submission and acts in response to the actions of the schedulers in the batch system; thus, it can be used without any modifications to existing independently managed batch schedulers at various sites. Also, since its only interaction with the batch system is to submit a job, it can be used with a wide range of batch system schedulers.

5) **Fault-tolerance:** Since our framework supports execution of an application across an open network with potential network instabilities that can cause failures in MPI executions, we have included an adjunct fault-tolerance framework for automatically detecting such and other failures and re-running the application from the previous restart-dumps. We have also incorporated a large-scale fault-tolerance feature within the coordinator daemon of the main framework to handle major failures such as node failures resulting in failure of the framework daemons.

While Morco is unique for the kind of application class and grid system it caters to, similar frameworks have been developed for other grid applications. For example, Buisson et al. [19] in their work on scheduling malleable applications in multi-cluster systems, have developed a middleware framework called DYNACO for their application runner, MRunner, to execute malleable applications. Our framework, though similar, is designed for multi-submission executions on generic batch scheduling systems. Another example is a middleware framework developed by Markatchev et al. [96] for checkpointing, migration and reconfiguration for ex-
ecution of traditional long-running applications. While they consider batch systems and execution time limits of the systems, and perform migration of batch jobs before reaching the time limits, unlike us, they do not perform simultaneous execution of an application across jobs on multiple queues. More recently, Ko et al. [83] have presented a solution for coupled multiphysics applications across multiple queues. However, they do not consider long-running applications.

Hence, our grid solution Morco is the first grid middleware framework, to our knowledge, that supports robust and effective execution of long-running coupled multi-component applications using multiple submissions of batch jobs on multiple independently administered batch systems of a grid.

### 1.7 Rescheduling

While the Morco framework described in the previous section enabled the execution of long-running multi-component applications on multiple batch systems, it was developed based on a “greedy” execution model. The greedy strategy involves rescheduling at every event, related to a batch queue becoming active or inactive, resulting in a change in the set of active queues. However, a decision to not reconfigure could yield a better performance in certain cases. For example, if after an event, we know that another event is about to occur, it might be more efficient to wait for the second event and then reconfigure to the best configuration instead of reconfiguring at both the events and incurring higher reconfiguration overheads.

The fourth part of our work, therefore, is the development of a dynamic rescheduling policy which considers various such cases to decide whether to reschedule upon the occurrence of an event. The algorithm considers various dynamic parameters including the availabilities of the batch systems, and execution rates to decide whether and where to reschedule when the available set of active batch systems changes. The rescheduling decisions involve the use of a genetic algorithm for component mapping, performance models for performance estimation of different schedules, and dynamic predictions of batch queue dynamics.

The objective of our rescheduling algorithm is two-fold, (a) to minimize the number of
rescheduling events to minimize overheads, and (b) to use the best possible configuration across the set of available queues. Our rescheduling algorithm is based on a single-step look-ahead strategy, i.e., our current decision is based on selecting the option with best execution progress until the next reconfiguration event. We incorporated this rescheduling policy in our Morco framework to make it more adaptive and efficient.

In summary, we have developed Morco, a middleware framework for execution of multi-component applications on independently administered batch grids consisting of multiple batch systems. The framework, which is generic and non-intrusive, requires no special administrative privileges or co-allocated global scheduling. The framework with adaptive rescheduling, dynamic resource allocation and fault-tolerance, supports continuing execution across multiple time-distributed submissions on each queue as well as concurrent execution across submissions on multiple batch queues. With an experiment involving an 8 day-execution of CCSM on a batch grid with four batch queues on three systems, we have shown that our framework enables multi-site executions yielding good application.

1.8 Climatology Application

Having developed a robust middleware framework for execution of long-running multi-component coupled climate applications on batch grids, we used this framework for execution of CCSM to study a climate problem. The climate problem we have considered is to study the impact of black carbon aerosols and dust aerosols on the Indian monsoons.

This problem is of importance because of the increase in concentrations of anthropogenic black carbon aerosols in the Indian regions. Further, degradation of land is resulting in larger amounts of dust aerosols in the atmosphere. While a large number of studies with various models have been performed to study the climatic impact of aerosols, especially in the Indian region [98, 1, 146, 87, 155], they differ from our work either in terms of the climate model used, with most of them using regional models or atmospheric general circulation models, or in terms of the experimental details.

We simulated 25 years of climate with the default aerosol concentrations and another 25
years with doubled aerosol concentration. We looked at the impact of doubling on various climate parameters including sea surface temperature, total precipitation rates and, frequencies of droughts and excess rainfalls.

Our observations indicate that increased aerosols result in an increase in precipitation in the central and eastern parts of India, and a decrease in precipitation in parts of the Arabian Sea. We also noticed a slight decrease in sea surface temperature (SST) over most of Indian ocean and a larger decrease in SST in the Persian gulf regions.

1.9 Organization of the Thesis

The details of how we have addressed the challenges mentioned in the previous sections are given in the rest of the thesis, which is organized as follows:

Chapter 2 discusses related work in various domains that our research has spanned including multi-component load-balancing, simulation studies for grid executions of applications and, middleware for climate modeling and multi-component applications on grids.

Chapter 3 presents a detailed study of CCSM as a multi-component application, the problem of temporal load-imbalance in CCSM and our solution to the problem, the Dynamic Component Extension (DCE). We also present in the chapter our experimental results of about 15% reduction in execution times with DCE.

In Chapter 4, we propose and evaluate execution models for coupled climate models on grids, and study the effectiveness of execution on grids through extensive simulations with various application and grid parameters. This chapter contains details of our models, simulation framework, simulation experiments and results. The simulation studies in this chapter indicated that we could get upto 55% average increase in throughput with grid executions.

Chapter 5 is a description of Morco, the adaptive middleware framework we have developed and all its features. With a week-long CCSM execution on a batch grid with four batch queues on three systems, we have shown that our framework enables multi-site executions yielding good application throughput.

Chapter 6 describes our adaptive rescheduling policy that involves dynamically deciding
whether or not to reschedule whenever the set of available resources changes. This aims at minimizing the number of rescheduling events and thus minimizing the total rescheduling overheads. Further, the best mapping of application to the set of available resources is also dynamically determined.

In Chapter 7, we describe the climate problem we have studied using our middleware framework. The problem involves studying the impact of doubling of black carbon aerosols and dust aerosols in the atmosphere. We used Morco to perform two 25-year climate simulation runs across four queues in three sites for over a long period of around two months.

Chapter 8 summarizes our work and primary contributions. This chapter also lists possible future research directions in this area.
Chapter 2

Related Work

In this chapter, we first mention some of the existing work in grid and distributed systems in the application area of climate sciences. We then mention work related to the various parts of the work we had mentioned in Chapter 1: load balancing in a multi-component application, simulation studies for execution of multi-component applications on grids, middleware for execution of multi-component applications on grids and the climatology problem.

2.1 Climate Science and Grid Computing

There have been several grid projects related to climate sciences. The most popular among them is the Climateprediction.net [34]. It is a distributed computing project to produce predictions of the Earth’s climate up to 2100 and to test the accuracy of climate models. To achieve this, the project requests people around the world to donate idle processing cycles of their computers, when the computers are not used by their owners. Thus a master-worker paradigm is formed with the processing center of the project acting as the master and the computers distributed across the world/Internet forming the workers. The workers run low-resolution climate models with different climate parameters, and the master processes results from the workers to validate the models and parameters. The work is significantly different from ours because we consider a very long duration run of a more complex state-of-the-art climate model that requires high performance computing systems with high processor density and high inter-processor commu-
nication rates, while they use an ensemble of several lower resolution model runs. The climate models in the Climateprediction.net framework are run with various altered parameters and hence it can be classified as a bag-of-tasks application framework.

Another popular grid effort in the area of next general climate research is the Earth System Grid (ESG) [51]. It is part of the SciDAC (Scientific Discovery through Advanced Computing) program. ESG integrates supercomputers with large-scale data and analysis servers located at numerous national labs and research centers. Access to ESG is provided through a system of federated Data Gateways, that collectively allow access to massive data and services for global and regional climate models, IPCC (Intergovernmental Panel on Climate Change)[78] research, and analysis and visualization software. While this is a significant project and also the source of our model codes and inputs, it is a data grid that dispenses data, and does not involve user execution of models on the grid.

While the above are some significant efforts to use grids in the area of climate prediction that are widely different from our work, there are also some efforts that are closely related. These are efforts that involve porting climate models onto production grid systems.

Most significantly, the work of Ayon et. al. [8] involves grid-enabling CCSM, the same climate models that we have considered in our work. They have developed a TeraGrid based web portal that allows TeraGrid users to run CCSM simulations on TeraGrid resources without having to individually port and install CCSM and without encountering lower level details such as specifics of batch queues and library locations. However, while this work makes CCSM accessible to a large number of users through the grid installation, the executions are carried out on a supercomputing system at a single site chosen from the set of sites. It also doesn’t handle automatic migration or restart upon time-out of resources. Further, it does not look into aspects of performance. Our work, in contrast, supports continuous performance-efficient adaptive dynamic executions across multiple resources.

A similar work of grid enabling NASA’s Global Circulation Model, Goddard Earth Observing System Model (GEOS-5)[66] for execution on various high performance computing systems of NASA Centre for Computational Sciences (NCCS) has been reported [129]. The objective of this work is also to expose model functionality while abstracting away the techni-
cal details to the end users. This too does not consider the long-running nature of the climate simulations or the dynamic nature of the grids.

Cozzini et al. [132] have executed an atmospheric model RegCM3 (Regional Climate Model Version 3) and an Ocean Model ROMS (Regional Ocean Modeling System) on the EU-IndiaGrid Infrastructure involving C-DAC, India and ICTP, Italy. However, while they have ported the code on each site of the grid, they have not reported executing of the two components simultaneously across the grid.

Mechoso et al. [97] have demonstrated superlinear speed up with a global atmosphere and a world ocean model run on a heterogenous, distributed Metacomputer on the CASA Gigabit Network Testbed. Their work focuses on achieving this by choosing the appropriate computer architectures and masking communication with computation unlike our work which is based on queue waiting times. Their work involves optimizing application performance across two heterogenous systems and does not involve enabling and optimizing execution through batch queues. Execution of an application across multiple batch queues requires special middleware framework for coordination of component jobs which makes our work different and more complex.

Thus, our work is unique in supporting performance efficient long-running executions of a state-of-the-art climate model across multiple independently administered grid resources.

### 2.2 Load Balancing in Multi-Component Applications

The first part of our work deals with addressing temporal load-imbalances arising due to the component coupling strategy adopted in CCSM, an important multi-component application. In this section, we first mention some of the work related to software coupling strategies for multi-component applications. We then describe related efforts on load balancing and coupling in coupled climate models.
2.2.1 Coupling Strategies in Multi-Component Applications

Some early papers in the area of multi-component or multi-physics applications talk about the various component coupling strategies and the challenges involved. For example, Chow and Addison [31] present an overview of the software strategies and the tools and libraries available for multiphysics parallel applications. The work mentions the importance of dynamic load-balancing strategies for concurrent multi-component applications. Lawrence [88], in an overview of multi-disciplinary aeropropulsion simulations, describes three approaches based on the nature of multi-disciplinary coupling: loosely coupled, coupled process and multiphysics. Our work applies to the second strategy. Lermusiaux el. al. [91] consider sequential, concurrent and hybrid strategies to perform coupled physical and biogeochemical ocean simulations. This work is an example of various component coupling strategies being evaluated for a multi-component application.

Larson [85] presents a heuristic set of definitions and organizing principles for coupled models. The work also describes the complexities involved in coupling a multi-component application using the case study of CCSM. In our work, we also consider the execution time profile of the coupled applications. Larson et. al. have developed MCT [86], a toolkit that allows construction of coupled multi-physics systems using parallel constituent models and has been used for coupling in few multi-component applications[29, 35, 101]. MCT supports several static component frameworks including single and multiple executables, and sequential, parallel and hybrid component executions [86].

However, none of the existing work on multi-component applications deal with dynamic optimization strategies involving multiple components. Our work deals with dynamic load balancing across multiple components using component extensions. Although there is a huge amount of literature relating to dynamic load-balancing in single component parallel applications and dynamic intra-component load balancing (e.g. [45]) in multi-component applications, we are not aware of any work related to multi-component applications that addresses temporal load imbalances in components and the resultant inter-component load imbalances.
2.2.2 Load Balancing Coupled Climate Models

Michalakes[99] studied and quantified the load-imbances in NCAR’s Community Climate Model (CCM1). The load-imbances in the different components are classified as temporal load-imbances or spatial load-imbances. In particular, the radiation calculations have been identified as a source of high temporal load-imbalance. Although no load-balancing solutions are proposed, it has been suggested that dynamic remapping of work to processors could help if the cost for movement of work between processors is low.

Michalakes and Nanjundiah[102] analyzed the load imbalances in the physics computations of PCCM2[46], a parallel implementation of NCAR’s Community Climate Model(CCM2). They identified the diurnal load-imbalance in the short-wave radiation calculations due to additional computations required over grid-points in the day region, accounting for about 75% of the total imbalance during an average time step. They also identified three broad types of time-steps based on the amount of radiation calculations.

Foster and Toonen[61] proposed four dynamic mapping schemes to reduce the diurnal load-imbalance described in [102]. Their work is based on the observation that in atmosphere physics, the computations at each grid point are independent of those at other points. Thus, their load-balancing strategies involve migrating some grid-points and their associated computations between processors. Due to the rapid temporal changes in load-distribution and the different load distribution characteristics of the three types of time-steps, they recalculate the mapping at every time-step. Since their new mapping calculated for atmosphere physics cannot be used for the dynamics, different grids are used for atmospheric physics and atmospheric dynamics and each time step involves communication to move data between the two grids. In spite of these overheads, their experiments show that there is an overall speedup due to load-balancing. The load-balancing is done only between pairs of processors by swapping every second column, swapping half the excess day columns with night columns or moving day columns based on pre-computed costs.

For their load-balancing work, Ford and Burton [59] used a global circulation model developed by U.K. Meteorological Office. Their work involves balancing two kinds of load-imbances: predictable dynamic imbalances like those in short-wave radiation calculations,
and unpredictable dynamic imbalances like those in convection calculations. Their work on load-balancing the short-wave radiation calculations is an extension of that of Foster and Toonen[61]. Instead of load-balancing only between pairs of processors, the load-balancing is done across all processors. To address the unpredictable load-imbalances, they use dynamic chunk formation, work stealing and dynamic feedback mechanisms.

The work by Muszala et. al. [110] load balances the physics calculations by dynamically remapping the grid-points to processors. Here, a very fast simulated annealing technique is used to determine the mapping. However, no actual implementation in the CCM has been done and instead, the experiments were conducted using LBSF, a proxy model of the CCM as test-bed. Their results show that their method outperforms that of Foster and Toonen for higher number of iterations. Yet another work by Muszala et. al [111] for load balancing the parameterization of moist convection uses model data to predict computational intensity of grid-cells. Specifically, if the CMFDQR (moist convective rain-out) field of a grid-point is greater than a specified threshold, the grid-cell is marked to be computationally intensive. Load-balancing is done by balancing the number of computationally intensive grid-cells across pairs of processors. The major difference between this work and that of Foster and Toonen is that this work uses spatial and temporal locality to allow data decomposition from a load-balancing step to be retained for multiple time-steps, thereby making the overhead to be cost-effective. However, while high improvements in performance of the convection parameterization subroutine are indicated, this subroutine forms a very small percentage of the calculations in CCSM. Therefore, the overall performance improvement percentage, which has not been reported, may not be high. But even a slight percentage improvement might translate to the several hours of saving in simulation time that they have reported when the simulated time is a few centuries.

There have been several efforts in load-balancing for various other climate models. For example, the efforts by Nanjundiah [113, 114] deal with a global atmospheric general circulation model, the NCMRWF model[133], in which the load-imbalances due to increase in cumulus convection computations from the pole to equator was found to be significant. This was addressed by allocating a mix of latitude-circles from the extra tropics and the tropics to each processor instead of contiguous latitude-circles. In the work by Michalakes et. al. [100], the
atmospheric model used is a parallel implementation of the non-hydrostatic version of the Penn State/NCAR Mesoscale Mode, MM5[69] (a region model i.e. whose domain is not global). This model includes nesting capabilities to achieve high resolutions cost-effectively and allows dynamic migration of columns at run-time for load-balancing.

Although all these above efforts show improvements in execution times of a single component, they do not address problems related to a coupled climate system model involving load imbalances between multiple components. Efforts that involve the other components in load-balancing are based on determining the optimum static allocation of processors to components. These do not involve any algorithmic modifications for load-balancing. For instance, the CCSM Load Balancing Workshop document [24] outlines the general methodology to be followed for such load-balancing. The technique suggested is a trial and error method. The largest possible number of processors are allocated to atmosphere and the remaining processors are allocated to the other components such that the atmosphere is not delayed. Carr et. al. [25] suggest general strategies for processor mapping of the highest resolution models based on experiments with CCSM3 on the Cray X1. These static processor allocations only aim at preventing idling of atmosphere, while the non-atmosphere component processors will still have large idling times. Further, as the model evolves, the component characteristics can change significantly and hence the static configuration may not remain optimal.

Our work tries to address several of the drawbacks discussed above by dynamically offloading columns of radiation calculations to other components and thereby minimizing component idling and also reducing the total execution time. While we have applied dynamic component extension to reduce temporal load imbalance due to long wave radiation calculations in the atmosphere component of CCSM in this work, our strategy is generic and can be applied for any huge temporal load imbalances in a component. Mirin and Worley [104] have recently studied the scalability of a single component application, Climate Atmospheric Model (CAM)[22] which is the atmosphere component of CCSM. Due to the different parallelization limits of the two phases of CAM, physics and dynamics, the authors use auxiliary processes to support larger parallelism for the physics phase. Thus, in the context of the multi-component CCSM, using suitable processor distribution to the components, our strategy can be used to support the large
parallelism of the physics phase by extending the physics phase of CAM to the idling processors executing other components instead of or in addition to the auxiliary processors. While the extension scheme of Mirin and Worley for the single-component CAM application needs dynamic process management features of MPI-2[108], our strategy for the multi-component CCSM employs simple load balancing of the existing processors used for application execution.

2.3 Simulation Studies for Multi-site Execution on Grids

The second part of our work involves simulation studies for multi-site execution of CCSM on Grids. We performed these studies to evaluate the effects of various system and application specific characteristics on the benefits incurred in a multi-site execution. Specifically, we have considered application performance models of CCSM and execution of components on different batch queues.

Multiple batch queues have been used for improving the response times of the jobs submitted to a system in the work by Subramani et. al.[141]. In this effort, a job submitted to a system is redundantly submitted to other batch systems. When the job starts execution in one of the systems, the redundant jobs submitted to the other systems are aborted.

Casanova analyzed the impact of redundant submissions on the other jobs in the system[27]. In this work, a job submitted to a system is redundantly submitted to other batch systems. When the job starts execution in one of the systems, the redundant jobs submitted to the other systems are canceled. The author concluded that while redundant tasks decrease average turnaround times of jobs and helps load balancing across clusters, they can cause heavy load in the systems and unfairness to the users who do not use such redundant jobs. In our work, we do not replicate jobs on multiple batch queues. We decompose a single job into multiple sub-jobs and submit these sub-jobs to many batch queues.

There has been increasing interest in co-allocating parallel applications across multiple clusters[18, 7, 6, 79, 118]. Bucur and Epema have extensively studied the benefits of co-allocation of processors from different clusters in a grid for job executions [18, 17, 16, 15]. In their work, they analyze the impact of using different scheduling policies, component sizes and
number of components on co-allocation. Their work considers 3 different scheduling policies, namely, GS policy in which there is a single global scheduler with a single global queue for submitting both single and multi-component jobs from all clusters, LS policy which deals with only local queues of the clusters where both single and multi-component jobs from a cluster are submitted to the local queue of the cluster and LP policy which considers both a single global queue for multi-component jobs and local queues of the clusters for single-component jobs from the clusters. Using large number of simulations with various workload logs, application characteristics and inter-cluster speeds, they show that execution of multi-component jobs across multiple clusters can reduce mean response times of jobs and improve processor utilization and that scheduling policies with only local queues perform better than those that consider global queues. They also conclude that restrictions on the number of components and component sizes help improve the performance of co-allocation.

Our work is complementary to the efforts by Bucur and Epema and by Ernemann et al. since we analyze the benefits of coscheduling multiple components of a specific application on multiple clusters of a grid. Our work considers scheduling policies with only local queues and the number of components and component sizes are restricted based on the application scalability. Their efforts consider improving mean response times of short jobs where the different components of a job are submitted to the different batch queues only once and the components complete executions within the execution time limits associated with the batch queues. However, our work focuses on improving the simulation rate of a single long running multi-component application where the different components are submitted multiple times to different batch queues. The execution of a component is stopped within the execution time limit associated with its submission to a batch queue and the component is submitted again to the same or different batch queue and continued from its previous execution.

In some of their scheduling policies developed by Bucur and Epema, they assume the existence of a global queue for submission of multi-component jobs. They also assume the use of the same job execution policy (FCFS) on all the local queues of the clusters. In our model, we attempt to execute multi-component jobs using a given set of local queues of different clusters with existing and possibly different job execution policies. Also, in their model, they as-
sume that all clusters become simultaneously available for execution of components. In our model, while submissions are made to the queues of all the clusters, an execution of the multi-component application can proceed when at least one of the clusters become available for execution. Finally, unlike their efforts, our work considers components with unequal and larger sizes in our simulations.

Thus our work is unique in analyzing the benefits of co-allocation for long running applications spanning multiple batch submissions with the batch queues of potentially different existing job execution policies and where the number of batch queues available for execution can change during the execution of the application.

The work by Nurmi et. al.[121] deals with execution of workflow applications on different batch systems of a grid. In their work, they schedule different tasks of a workflow application to different batch systems of a grid based on predictions of execution times of the tasks on the systems and the queue waiting times in the systems[13]. They show that considering predictions of queue waiting times leads to efficient schedules. In our work, we consider multi-component applications that contain periodic communications between different batch systems unlike the workflow applications.

The work by Platt et al.[126] and Bal et al.[6] analyzed the impact of inter-cluster speeds on the performance of parallel applications when executed across wide-area clusters. They also suggested optimizations of some applications to improve their wide-area performance. Their work compares the execution times of the applications when executed across wide-area and on local clusters and analyzes the benefits of wide-area computing due to the increase in the number of processors made available to the applications. In our work, we analyze the benefits of wide-area computing over computing in a local cluster for the same number of processors by decomposing the application into multiple components and analyzing the benefits due to reduction in queue waiting times.

The work by Ernemann et al.[50] is similar to our work and analyzes the mean response times of synthetic applications when executed across multiple clusters for different ratios of execution times of applications when executed on a single local cluster and on multiple clusters. Their results show that multi-cluster computing can yield improved response times due to
decreased queue waiting times as long as the execution times due to multi-cluster computing does not increase more than 1.25 times the execution times on local clusters.

For distributed applications to make use of resources at different sites with heterogeneous characteristics and site autonomy, several middleware tools and services have been developed[92, 12, 151, 143, 81, 60]. While most of them are limited to specific projects, resource types and middleware environments, or do not support co-allocation, the MetaScheduling Service (MSS) developed Waldrich et al. [150] overcomes these limitations using the metacomputing-enabled MPI-implementation MetaMPICH[127]. In some recent works like the Grid Interoperability Project[14] and Gridbus Grid Service Broker[149], solutions have been developed for accessing resources from different grids. Kertesz et. al.[82] have developed a meta-brokering architecture that enables the interoperability of various grids through their own resource brokers.

The work by Grimme and Papaspyrou [70] has built a service-oriented grid infrastructure to support workflow-based scientific applications for climate science. The framework performs scheduling and data management of loosely coupled workflow tasks executed on distributed sites. The primary aim of the work is to provide coherent access to distributed data. Our work focuses on execution of multi-component applications on grids where the communications between the components are more frequent and intensive than the workflow applications. Elmroth and J. Tordsson[49] have developed a grid resource manager for performing resource brokering and job scheduling. The objective of job scheduling is to use the job and resource characteristics to minimize the response time that includes the times for file staging, batch queuing and job execution. The work uses advanced reservations for co-allocation of resources. In our execution model, the components are co-allocated dynamically without reservations. Moltó et al.[105] have developed generic multi-user resource brokering and metascheduling techniques for remote execution of scientific applications. The metascheduling in our work is targeted for the specific application domain of climate modeling.

To our knowledge, our work is the first effort in quantitatively showing the benefits of executing a parallel application across multiple batch systems over executing the application on a single batch system. It is also unique in analyzing the benefits of inter-site execution for long
running applications spanning multiple batch submissions with the batch queues of potentially different existing job execution policies and where the number of batch queues available for execution can change during the execution of the application.

2.3.1 Differences with Workflow Applications

A workflow has been defined by Fox and Gannon [63] as “the automation of the processes, which involves the orchestration of a set of Grid services, agents and actors that must be combined together to solve a problem or to define a new service.” Most workflow applications[41, 40] consist of a series of processes, either involving multiple grid services that must be combined together at various levels or stages of executions. Each level is data or control-dependent on the previous level. Components or services in a level are independent and execute concurrently. Thus a single level consists of data-parallel non-communicating concurrent actors.

For example, a popular workflow application, Montage[71], that creates custom image-mosaics out of existing astronomical images, involves multiple stages of processing starting from the initial set of images. Some processing levels in workflow are data-parallel involving multiple component tasks each corresponding to one of the input images. Yet another workflow example, more closely related to our application domain is the mesoscale weather prediction in LEAD-VGrADS[130]. This involves grid-services based workflows with data-streaming and a number of pre-processing steps that feed data to a computation intensive weather model. Further an ensemble of short runs of weather models are invoked with various parameters.

These are significantly different from our global coupled climate model in which all application components execute continuously and concurrently for very long durations. Our application is a time-marching or time-evolving MPI MPMD application consisting of multiple “components” executing concurrently throughout the execution duration. Our applications involve complex communication patterns and data exchanges between the components unlike the well-defined interfaces and data-flow patterns or dependencies in traditional workflow applications. The “components” of CCSM frequently exchange information through a fifth component called the coupler with calls involving communications throughout the period of the applica-
tion’s execution. Further, each of the CCSM components have very very distinct scientific functionality. The components are scientific models simulating the various interacting climate sub-systems, namely, atmosphere, land, ocean and sea-ice. In most of the practical workflow applications, the components perform similar tasks.

Hence, our application, in its current form, is different from a workflow application and cannot be supported by any of the existing workflow-based technologies[10, 152].

2.3.2 Rescheduling on Individual Machines and Batch Systems

There have been numerous efforts on rescheduling and migrating single-component applications on individual machines[76, 57, 153, 147]. For example, Varela et al. have developed a modular framework called Internet Operating System (IOS)[43] for supporting adaption to both changing application and resource characteristics. Their decisions for rescheduling consider process-level characteristics. In the work by Wrzesinska et. al.[153], 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 efficiency 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.

Rescheduling across multiple batch systems also involves considering overheads and evaluating benefits for rescheduling decisions. Also, overall similarity exists between the rescheduling frameworks that consider individual machines and different batch queues or systems. However, rescheduling for batch systems involves considering queue dynamics and job submissions on different batch systems. The definitions for availability and non-availability are different for our work. While individual machines become unavailable due to failures, in our execution model, batch systems become inactive or unavailable due to the jobs reaching the execution time limits. While processes are migrated from individual machines that become available, in our work, batch job submissions are made to the systems that become inactive. Also, scheduling decisions are much more complex for multi-component applications on batch systems. While
overall scheduling decisions considering all machines are made for individual systems, scheduling on a set of active batch systems involves coupled scheduling decisions for each system. The components and number of processors for the components in a batch system are dependent on the component and processor allocation in the other batch systems. Also, rescheduling for individual machines involves simple process or job startups. In our framework, rescheduling involves detection of jobs becoming active on different batch systems, complex coordinations between the active jobs through the coordinator and the job scripts, reading individual schedules sent by the coordinator, and re-execution of MPI applications with the new schedules within the job scripts.

2.4 Middleware Infrastructures

Cactus[21, 68] is a generic and modular middleware framework for enabling execution of scientific applications on high performance systems. It has a core set of components, called “flesh”, for providing services for computing. Application scientists can develop their own application specific components, called “thorns” and integrate with the other components of the Cactus through well defined interfaces. Cactus also provides common set of thorns for use in wide range of applications. One of the thorns relate to migration of tasks to different resources on performance degradation[2]. We can extend this work for our framework to incorporate our rescheduling and migration policies based on batch queue dynamics. Our work can be integrated with the Cactus framework by developing specific thorns for the different components in our framework, our rescheduling policies and component-level migrations, and using the common thorns and core components of Cactus for I/O of restart file, batch queue monitoring, resource discovery, prediction, resource selection, basic scheduling and checkpointing functionalities[139]. Many of our coordinator’s functionality can be performed by the Cactus flesh. Such integration will help efficient executions of our multi-component applications in a generic and portable manner.

In this work, we had assumed that sub-components executed on different batch systems can communicate with each other. This assumption is reasonable since some MPI (Message Passing
Interface) communication libraries including PACX-MPI[64] and MPICH-GX[123, 30] support communications between MPI applications executed on different batch systems by means of special communication processes or proxies executed on the front-end nodes of the batch systems. Another solution called Smartsockets[95] has been proposed to facilitate direct connections between multiple MPI processes executing on different grid sites. Smartsockets uses an integrated solution consisting of techniques including port forwarding, TCP slicing and SSH tunneling and optimizes performance related to connection setup. It also supports a simple programming interface for use by application developers. The Smartsockets solution can be used in our framework to enable communications between processes of different components executing on different batch systems.

Execution of any application on resources across multiple sites located in different administrative domains requires special middleware. Only few tools like Calana [39] or the GridWay meta-scheduler[77] are capable of such allocation. But most of these lack mechanisms for coallocating resources, i.e. reserving common time-slots for availability of resources by negotiating with agents of the resource owners, for execution of a distributed application.

VIOLA MetaScheduling framework [48] describes a coallocating framework, in which an iterative querying process is used to determine a reservation based common timeframe across resources maintained by the local schedulers. The execution of an application MetaTrace is demonstrated with this framework. MetaTrace is a coupled application with two components: Trace (MPI with F90) and Partrace (MPI with C++). A time dependent vector field describing the water flow is transmitted after ever simulation time-step from Trace to Partrace, which computes the dynamics of particles solved in the water or deposited in the ground. The transfer uses MPI send and receive calls. The execution was done in the VIOLA test-bed using MetaMPICH. The application is executed only when all the resources are available in the same timeframe. This can lead to very high waiting times or even an error condition due to inability to coallocate in the preview period. Hence, solutions for execution of distributed applications across multiple resources are necessary.

Recently, some middleware solutions for such execution have been reported. For example, Markatchev et al. [96] have developed a middleware framework for checkpointing, migration
and reconfiguration for execution of traditional long-running applications across multiple sites. They have demonstrated their framework using a molecular dynamics simulation application, GROMACS. They consider heterogeneous platforms and perform dynamic resource discovery. They also consider independently administered batch systems and execution time limits of the systems, and perform migration of batch jobs before reaching the time limits. However, unlike our framework, their work does not support simultaneous execution of an application across jobs on multiple batch systems.

In yet another recent work, Ko et al. [83] have presented a solution for coupled multiphysics applications across multiple queues. Their solution also includes dynamic resource allocation, load balancing and handling different queue wait times. Their work however does not handle continuous execution of long running applications. Our framework also solves problems due to different execution time limits and queues becoming inactive during execution, and includes robust rescheduling policies.

One other work that is related to ours is that of Buisson et al. [19], who in their work on scheduling malleable applications in multi-cluster systems, have developed a middleware framework called DYNACO for their application runner, MRunner, to execute malleable applications. The framework components dynamically monitor and adapt the execution of components. Our framework, though similar, is designed for multi-submission executions on generic batch scheduling systems.

To our knowledge, ours is the first work on building a framework for execution of a long running multi-component application where the different components are submitted multiple times to different batch queues. The execution of a component is stopped within the execution time limit associated with its submission to a batch queue and the component is submitted again to the same or different batch queue and continued from its previous execution. In our model, we attempt to execute multi-component jobs using a given set of local queues of different clusters with existing and possibly different job execution policies. In our model, while submissions are made to the queues of all the clusters, an execution of the multi-component application can proceed when at least one of the clusters become available for execution.
2.5 Climate Application: Impact of Aerosols

We have used our grid middleware framework to study the impact of increased black carbon and dust aerosols on the climatic conditions, especially the Indian monsoons. This is of great interest to climate scientists and a large number of studies with various models have been performed to study the climatic impact of aerosols, especially in the Indian region. Different researchers use different models and experimental conditions to study different aspects of the problem.

Menon et al. (2002) [98] investigated the effects of black carbon aerosols in China and India using a global climate model. They noted that the absorbing black carbon aerosols heat the air, alter regional atmospheric stability and vertical motions, and affect the large-scale circulation and hydrological cycle with significant regional climatic effects. Chakraborty et al. (2004) [1] studied the impact of anthropogenic absorbing aerosols on the climate over the Indian region using the NCMRWF general circulation model. They show an overall increase in rainfall and a reduction in surface temperature in the Indian region. Ramanathan et al. (2005) [146] have performed coupled ocean-atmosphere simulations to study the impacts of atmospheric brown clouds in South Asia on regional climate and hydrological cycle. Their studies have also indicated that aerosols have significant climatic effects.

Lau et al. (2006) [87], with a global circulation model, have suggested that increased dust load coupled with black carbon emissions lead to Asian summer monsoon anomalies. More recently, Zhang (2009) [155] has studied the effect of the collective effects of anthropogenic aerosols using CCSM3. Their experiments involve increasing aerosols locally and decreasing aerosols globally, and they have incorporated MOZART [106] simulation based dynamic footprints of the increased Asian emissions.

Our work involved using CCSM3 with uniform doubling of dust and black carbon aerosols globally. Our observations indicate that increased aerosols result in an increase in precipitation in the central and eastern parts of India, and a decrease in precipitation in parts of the Arabian Sea. We also noticed a slight decrease in sea surface temperature (SST) over most of Indian ocean and a larger decrease in SST in the Persian gulf regions. Regions with largest decrease in SST appear to be associated with regions with highest dust concentrations. Over Arabian Sea where SST reduces, there is a significant reduction in rainfall.
Chapter 3

Load Balancing CCSM: Dynamic Component Extension

An important step for enabling grid executions of an application is identifying potential performance bottlenecks and load imbalances of the application. This can be done on individual clusters and the results can be applied to Grid. We carried out such analysis for CCSM and identified large-scale temporal load imbalances that resulted in poor performance. We propose a strategy for load balancing computations across components. In this chapter, we describe in detail our software coupling strategy, called Dynamic component extension (DCE), to address temporal load imbalances present in CCSM and other similar multi-component applications.

3.1 Introduction

With major advances in high performance computing, the scientific community is moving towards multi-disciplinary multi-component models to accurately model interacting physical processes or phenomena. Examples of such applications include models of climate, space weather, solid rockets, fluid-structure interaction, heart disease, cancer, ocean plankton population, and nuclear energy [36, 86, 42, 38, 144, 90]. Typically, these applications involve long-running simulations of constituent model components with periodic communication or coupling between the components.
While each of the constituent model components is generally parallel, the two basic software strategies used to couple these components are sequential and concurrent. The sequential strategy, illustrated in Figure 3.1(a), involves a driver that for each coupling cycle or time-step, sequentially invokes each of the constituent models on the same sets of processors. The concurrent strategy, illustrated in Figure 3.1(b), typically follows Multiple Program Multiple Data (MPMD) paradigm and involves concurrent execution of the models on different sets of processors with coupling often performed through a dedicated coupler. For example, the parallel climate model (PCM) [125] involving multiple coupled components is a sequential SPMD version while community climate system model (CCSM) [29] is a concurrent MPMD version of coupled climate models.

The strategy used depends on various factors including amount of exploitable concurrency, target platform, memory footprint, development plan, model scalability, etc. The sequential strategy is adopted when there is not much exploitable concurrency among components of the application. The concurrent strategy is favorable for complex multi-disciplinary coupled applications as the models can be built, developed and tested as stand-alone applications by independent teams. Hence, the concurrent strategy is widely used for various large-scale applications [36, 29]. However, when the model components have complex interacting patterns and dependencies, the concurrent strategy can lead to large processor idling. Idling of processors executing certain components can also be caused due to large temporal load-imbalances caused
by highly computationally intensive phases in a component within a time step or coupling period. Hybrid strategies can be used to alleviate the processor idling in concurrent strategy. In this strategy, a subset of components with dependencies is executed sequentially as illustrated in Figure 3.1(c). An example of such an implementation of the climate system model is the fast atmosphere-ocean model (FOAM)[58]. While the hybrid strategy can address the idling due to component dependencies, it cannot address the idling due to temporal load imbalances in a component.

In this chapter, we present a software strategy called dynamic component extension to minimize idling due to temporal load-imbalance and improve performance in concurrent and hybrid versions of multi-component applications. In this strategy, the processor space of the components containing temporal load-imbalance is extended to include the processors executing other components during the computationally intensive phases of a time step or coupling period. The strategy is illustrated in Figure 3.2. The extension is dynamic since it takes into account the ready times of the processors for sharing the computational loads. The strategy is intended for multi-component application execution in a cluster of homogeneous processors. We demonstrate the potential of the strategy to minimize idling and improve performance in Community Climate System Model (CCSM), a classic and foremost example of a multi-component application. Our work is directly applicable for only those multi-component applications where only one component has temporal load imbalance, is irregular and executes longer than other components in some coupling intervals. We have not considered applications where two or more components can have longer execution times than the remaining components.

Community Climate System Model (CCSM)[29] is a global climate system model from National Center for Atmospheric Research (NCAR)[115]. The CCSM climate model consists of five components, namely, atmosphere, ocean, land and ice and a coupler component which transforms data and coordinates the exchange of information across the other model components. The CCSM is implemented as a multiple program multiple data (MPMD) programming model where each component is a separate parallel application by itself. Of the five components, atmosphere is the most computationally intensive component and is allocated the highest number of processors[25, 24]. The atmosphere model consists of calculations for dynamics and
physics. The physics includes calculations corresponding to short-wave (visible and ultra-violet radiations received from the sun) and long-wave radiations. The long-wave radiation calculations deal with calculations of radiations emitted by the earth-atmosphere system. The most computationally intensive part of these long-wave radiation calculations is the estimation of emissivity and absorptivity of various constituents of the atmosphere. Since these co-efficients change very slowly with time, they are not computed at every time step but once every few simulated hours (typically between 3-12 hours). Thus the atmosphere component involves large periodic temporal load-imbalances due to the computationally intensive emissivity and absorptivity components of long-wave radiation calculations (for brevity, we will refer to calculations related to emissivity and absorptivity as long-wave radiation calculations even though long-wave radiation calculations also consist of other calculations). The temporal load-imbalances result in idling of the processors executing non-atmosphere components during the long wave radiation calculations in the atmosphere component. Studies [124] show that radiation computations should be done as frequently as possible preferably at the same frequency as the rest of the model.
We apply our strategy to CCSM by dynamically extending the processor space of the atmosphere component (atmosphere processors) to include the idling processors executing the non-atmosphere components during the long-wave radiation calculations in the atmosphere. The extension characteristics including the amount and points of extension, and the processors of other components to include during the extension are dynamically determined based on the times when the non-atmosphere processors are ready to share work and the different times taken by the different atmosphere processors to start their long-wave radiation calculations. By means of experiments with different processor configurations for two different modeling resolutions on five multi-processor systems, we show that our dynamic component extension strategy can lead to about 15% reduction on 16 processors and savings of up to 50 days in execution times of CCSM for 1000-year simulation runs.

Section 3.2 discusses the effect of temporal load imbalance on performance of a coupled application and explains the benefits of dynamic component extension strategy to improve the performance. Section 3.3 identifies the challenges in load balancing long-wave radiation calculations in CCSM. In Section 3.4, we describe our dynamic component extension (DCE) strategy applied to long-wave radiation calculations in atmosphere component of CCSM. Section 3.5 describes the experiments and presents results that illustrate the benefits of using our approach over the existing code for CCSM. Finally, section 3.6 is a summary of this chapter.

3.2 Temporal Load Imbalance and Dynamic Component Extension

Temporal load imbalance or variation in computing load of a component across time-steps can have a significant effect on the performance of a concurrent multi-component system. We illustrate temporal load imbalance and dynamic component extension with a simple multi-component application consisting of four components executing three time steps as shown in Figure 3.3(a). We assume that the components couple or synchronize at the end of each time step. The figure shows that the components are perfectly load balanced in the first and the third time steps. The figure also shows that component C2 performs more computations in the sec-
Figure 3.3: Effect of (a) Temporal Load-Imbalance and (b) Component Extension on Coupled Application Execution

ond time step than in the other time steps. Let $\Delta W$ represent these additional computations. This temporal load-imbalance due to $\Delta W$ in C2 causes large idling and low utilization of the processors executing the other components in the second time step. The time taken for the $\Delta W$ computations in C2 and the corresponding idling times in other component processors are denoted as $\Delta T$ in the figure. Using dynamic component extension scheme, C2 can be extended to also execute in other component processors during the $\Delta W$ computations in the second time step as shown in Figure 3.3(b). Assuming ideal speedup for the computations, the figure shows complete elimination of the processor idling and the resulting decrease in the overall completion time, denoted as gain. The time taken for the $\Delta W$ computations with the dynamic component extension is denoted as $\Delta T'$.

Generalizing the above example, we consider a multi-component application consisting of N components, C1,C2,...,CN with component Ci executing on n(Ci) processors. Component
Cj has periodic temporal load imbalance due to additional computations, $\Delta W$, in certain time steps. Assuming that the computations in Cj have ideal speedup and that the components are perfectly load-balanced in other time steps, the time taken for the $\Delta W$ computations, $\Delta T'$, and the overall reduction in execution time of the application, $gain$, due to dynamic component extension are given by

$$
\Delta T' = \Delta T \times \frac{n(C_j)}{\sum_i n(C_i)}
$$

(3.1)

$$
gain = \Delta T \times \sum_{i \neq j} \frac{n(C_i)}{\sum_i n(C_i)}
$$

(3.2)

In general, the gain due to dynamic component extension in a multicomponent application is application-dependent and depends on various factors including number of processors, distribution of processors to components, scalability of components, extensibility of additional computations in components, component coupling frequency and pattern, impact of the components with temporal load imbalances on the overall execution time etc. We demonstrate our strategy with CCSM, an important multi-component application and present our experiences in dealing with these factors and our solutions to the generic and application-specific challenges encountered.

### 3.3 Load Balancing Challenges in CCSM

There are two sources of load imbalance in CCSM: (i) Load imbalance across components (inter-component load imbalance), and (ii) Load imbalance across processes of each component (intra-component load imbalance).

Load balancing across components by allocating suitable number of processors to each component is complicated because the components are coupled at various frequencies and the computations and communications are interleaved. The periodic coupling of information between different components of CCSM occurs via coupler at different frequencies for different components. For example, the atmosphere exchanges information with other components every simulated hour, while the ocean exchanges information with other components only once in 24 simulated hours (the lower frequency of coupling with ocean is due to the fact that ocean has much higher inertia and the state changes more slowly than the other components). Further,
several calculations are performed periodically at different frequencies. For example, the radiation parameter calculations which are part of atmospheric physics are performed only once in 36 time-steps of atmospheric simulation, where each time-step corresponds to 20 simulated minutes.

However, the general guideline [25, 24] for load-balancing is to give a large number (close to two-thirds) of the available processors to atmosphere and then try to minimize idling of atmosphere processors by giving sufficiently many processors to the other components using trial-and-error. In spite of following the above guideline for load-balancing, we identified two major load-imbalances resulting in idling of processors.

3.3.1 Inter-component load imbalance due to temporal load-imbalance in atmosphere

A major percentage of atmosphere calculations is the long-wave radiation calculations. For instance, in an experiment with 8 processors for atmosphere, 4 for ocean, 2 for ice and one each for land and coupler, calculations of absorptivities in long-wave radiations consumed 35% of the time for atmosphere calculations. Figure 3.4(a) shows the times spent by an atmosphere processor performing calculations between receive and send communications with the coupler at different time-steps. While the coupler communications have a period of 1 simulated hour, the periodicity of the absorptivity calculations is set to 3 simulated hours for this experiment. The huge spikes in the graph occurring at every third simulated hour correspond to these computations. Thus, these long-wave radiation calculations cause temporal load imbalance in the atmosphere.

In these time-steps, there is a large load imbalance among the components and hence large idling of the processors executing non-atmospheric components. This is reflected as peaks in the times for communications with the coupler for the non-atmospheric model components since they are forced to wait until the coupler finishes its communications with the atmosphere. The coupler in turn is idle waiting to communicate with the atmosphere. Figure 3.4(b) shows the times spent by the land processor for receiving communications from the coupler. We find that corresponding to large computations in the atmosphere processor shown in Figure 3.4(a), there
are large communication times in the land processor as seen in Figure 3.4(b).

For atmospheric physics calculations, the atmospheric grid, consisting of latitudes on one axis and longitude on another, is divided into chunks, where each chunk is a collection of a fixed number of columns. A column represents all the vertical levels corresponding to a latitude-longitude pair. Each atmosphere processor performs the physics calculations corresponding to a set of chunks. The chunk formation and assignment to processors are based on a load-balancing option set at compile time. For each chunk, as part of physics calculations, a call is made to the computationally intensive absorptivity calculating function, \textit{radabs}, the source of temporal load imbalance. Inside \textit{radabs}, the long-wave radiation calculations are performed for the columns that constitute the chunk. There are no dependencies between the calculations corresponding to any two columns.

### 3.3.2 Intra-component load-imbalance due to short-wave radiation calculations

The physics calculations for each chunk has a call to a fairly computation intensive short-wave radiation calculation function \textit{radcswmx} before the call to \textit{radabs}. This function, performing calculations only for the grid-points in the day region, is a cause for very high intra-component load imbalance between the atmosphere processors. Although the standard 1-D decomposition of the grid along the longitudes results in the processors with latitudes close to the equator getting equal number of day and night grid-points, those at the poles may have all grid-points corresponding to day (or night) depending upon the season. This is illustrated by the graph in Figure 3.5 that shows the times spent by the atmosphere processors in \textit{radcswmx} at different time-steps. The graph also shows that the load imbalance between the atmosphere processors varies over time.

CCSM allows a “load-balanced” chunk formation scheme which addresses the load imbalance in the short-wave radiation calculations. However, the use of this scheme results in an increase in the time taken for remapping the dynamic grid to the physics grid at every time step. The pseudo code involving short and long-wave radiation calculations is given in Figure 3.6. By default, the \textit{radcswmx} time step is every 3 hours and the \textit{radabs} time-step is every 12 hours.
Figure 3.4: Computation intensive Absorptivity Calculations in Atmosphere and the corresponding idling in Land
However, the work by Pauluis and Emanuel[124] describes the numerical instabilities due to infrequent radiation calculations. In our calculations, we perform \textit{radabs} calculations every 3 simulated hours without loss of generality.

### 3.4 Dynamic Extension of Atmosphere Component

Our component extension method tries to address both the above imbalances without introducing delays in any of the components. We reduce the inter-component load imbalance by offloading the absorptivity calculations of atmosphere to idling processors of other components. The amount of offloading by each atmosphere processor is in turn determined from the amount of intra-component imbalance in short-wave radiation calculations. In our scheme, each atmosphere processor offloads or sends some columns of each chunk to processors of other components. Then, each processor of each component including atmosphere performs absorptivity calculations on the columns it possesses. After calculations, the atmosphere processors receive the results corresponding to the columns it offloaded to other components.
Algorithm: Pseudo-code for radiation calculations

for each time-step do
  for 1 to nchunks do
    if radcswmx time-step then
      call radcswmx /* Short wave radiation calculations.
      Source of Intra Component Load Imbalance */
    end
    if radabs time-step then
      for 1 to ncol do
        call radabs /* Long Wave radiation calculations.
        Source of temporal load imbalance in atmosphere and inter-component load imbalance */
      end
    end
  end
end

Figure 3.6: Pseudo-code for radiation calculations
The first step of communication that involves sending radabs input from atmosphere to other components is performed before short-wave radiation calculations. By placing the sends before radcswnx, the radabs calculations in non-atmosphere components can be overlapped with the short-wave radiation calculations in atmosphere and part of the intra-component load imbalance can also be reduced. The destination processors are those that have finished their respective model calculations and are waiting to communicate with the coupler, which in turn is waiting to communicate with the atmosphere. Thus, the idling components can proceed beyond this point of execution only when atmosphere completes radabs and communicates with the coupler. It is however possible that atmosphere completes some or all of the radabs calculations before a component reaches its point of idling. Hence, not all of the radabs calculations can be extended to all non-atmosphere processors. Moreover, ocean has a coupling period of 1-day and other components have a coupling period of 1-hour, while the radabs period is typically 3-12 hours. Hence, extension to ocean processors and to other non-atmosphere component processors should be at different levels. In order to determine the amount of extension or the amount of sharing of radabs calculations of atmosphere processor with each non-atmosphere processors, we periodically execute CCSM without modifications for a simulated day and collect the times when each processor of each component is ready to share radabs calculations. Each component, at the end of the timed simulated day, has a ready-times array of size equal to the number of calls to radabs in atmosphere per simulated day. The ready-times for the non-atmospheric components are called the recv-ready times and the ready-times for the atmosphere component are called the send-ready times. These ready times are then used for atmosphere extension or load balancing in other simulated days. Thus the extension is performed dynamically based on the ready times collected periodically. Three primary issues have to be addressed for extending long-wave radiation calculations:

1. At what time-steps should atmosphere offload radiation calculations to a non-atmosphere component?

2. In a given time-step, should the radabs calculations of all the chunks be offloaded?

3. How many columns of a chunk should an atmosphere processor offload to each of the
other non-atmosphere processors?

We perform a 3-level extension strategy to address the above issues.

### 3.4.1 Time-step Level

At this level, the time-steps (one time-step=20 simulated minutes) at which atmosphere radiation calculations can be extended to non-atmospheric processors have to be decided. Atmosphere radiation calculations cannot be offloaded from atmosphere to non-atmosphere processors at all time-steps since the non-atmosphere processors may be performing their own calculations at certain time-steps when the atmosphere processors are performing radiation calculations. This typically happens when the coupling frequency of non-atmosphere components is lower than the frequency of radiation calculations as in the case of ocean. Ocean communicates with coupler only once in 24 simulated hours while radiation calculations due to radabs occur every 3 simulated hours. Ocean processors may be busy with their own calculations for the first 12 simulated hours of the simulated day and may become idle only from the 13th simulated hour. In this case, the atmosphere processors will not be able to offload radabs calculations to ocean processors during simulated hours 3, 6, 9 and 12 and will be able to offload or share only from the 15th simulated hour.

While the coupling and radiation calculation frequencies are static, the time step at which a component is ready depends upon the number of processors allocated to the component and can vary dynamically. Hence, the ready time step has to be determined at run-time and updated periodically. As explained earlier, corresponding to each radabs call, there is a send-ready time in the atmosphere and a recv-ready time in each non-atmosphere component. These ready times are determined by periodically executing unmodified CCSM for a simulated day and observing the times for various phases of execution. Our objective at this level is to determine the first time-step of a simulated day for which some of the radabs calls can be offloaded to a non-atmosphere component without the atmosphere being delayed. This time-step is the one immediately before the time-step for which the send-ready time of the atmosphere, corresponding to the first radabs call in the time-step, is greater than the recv-ready time of the component, and is calculated by each non-atmosphere component as outlined in Figure 3.7.
The \textit{radabs} time step after which offloading can take place is noted as the component’s time-step sharing index and is broadcast to all the processors. At every \textit{radabs} time step of each simulated day of the simulation with offloading, the atmosphere includes in its destination processor set, all the processors of every component that has (a) a lower coupling frequency than atmospheric \textit{radabs} frequency, and (b) a time-step sharing index less than the current \textit{radabs} time step.

### 3.4.2 Chunk level

When the coupling frequency of a component is greater than or equal to the \textit{radabs} frequency, the idling of the component occurs at every time step of atmospheric \textit{radabs} calculation. Thus, the offloading of atmosphere to this component can occur at all \textit{radabs} time steps. However, as seen in Figure 3.6, for a given time-step, the \textit{radabs} function is invoked for every chunk. Thus, for a given time-step, the non-atmosphere component may become idle and may be able to share \textit{radabs} with the atmosphere processors calculations only after the atmosphere processors have finished processing some chunks in the time-step. Thus, the chunk at which \textit{radabs} calculations can be shared for a given time-step has to be precisely calculated.

For each time-step, all processors maintain another array of send-ready times and recv-ready times, where each array element corresponds to a chunk within the time-step. The array index where the send-ready time exceeds the recv-ready time determines the number of chunks for which radiation calculations can be offloaded at that time step. We calculate the maximum of these numbers, across the time-steps, as the component’s chunk sharing index. Thus the chunk sharing index is the starting chunk number from which the atmosphere can offload radiation
calculations to processors of a particular component. The calculation is shown in Figure 3.8.

```plaintext
1 maxj = 0;
2 for i = 1 to ntimesteps do
3     for j = i to i+nchunks do
4         if send-ready-time(j) > recv-ready-time(j) then
5             maxj = Max(maxj,j-i) break
6         end
7     end
8 end
9 if maxj > 0 then
10    chunk-sharing-index = maxj-1
11 end
12 else
13    chunk-sharing-index = nchunks
14 end
```

**Figure 3.8: Calculation of chunk sharing index**

After the component root processor calculates its chunk sharing index as outlined above, it broadcasts it to all processors. In the following simulated days, the atmosphere, when processing each chunk, includes in its destination processor set, all the processors of the component that has (a) a higher coupling frequency than the *radabs* frequency, and (b) a chunk sharing index less than the index of the current chunk in the current time-step. Thus the points of extension to coupler, land and ice components are dynamically determined.

### 3.4.3 Column level

This step involves the finest level of decision making. For a given time step and chunk, a set of non-atmosphere processors may be ready to share *radabs* calculations. However, the number of columns within the chunk that will be shared by each atmosphere processor with each non-atmosphere processor has to be calculated. As mentioned earlier, the times at which each atmosphere processor enters *radabs* calculation can vary between atmosphere processors due to differences in times taken by the processors for short-wave radiation calculations. Thus, the short-wave radiation calculation time of each processor should be taken into account to calculate the number of columns that each atmosphere processor can offload to each of its
destination non-atmosphere processor so that all atmosphere and non-atmosphere processors finish the radabs calculations at about the same time.

Since the input data for radabs is sent before the short-wave radiation calculations, the time taken for the short-wave radiation calculations by each atmospheric processor in the previous time step is used to determine the availability times of the processor for performing radabs calculations. In each radabs time-step consisting of multiple radabs calls as shown in Figure 3.6, these values are sent to a non-atmosphere component by an atmosphere processor just before the first radabs call for which the component processors are included in the atmosphere processor’s destination processor set. Based on these availability times, a component processor determines the number of columns in the current chunk that it will process and the source atmosphere processors containing these columns.

Each processor that is involved in the extension for the current radabs time-step, calls the Column-processor mapping algorithm in Figure 3.9 with the array of availability times(AvailabilityTimes), the number of atmosphere processors(NumberOfAtmProcs) and the number of other processors (NumberOfOtherProcs) as inputs, and obtains as results the details of the communications it has to perform during the extension. The time taken for radabs calculations for each column is the pre-determined constant, RadabsColTime. The AvailabilityTime for each processor is the time at which it will be available to perform the calculations for the next column. It is initialized to zero for the component processors, and to their respective average short-wave radiation calculation times for the atmosphere processors.

The mapping is determined in two steps. The first step (see for loop in the figure), determines the number of columns to be given to each processor. For each column of the total available columns, we determine the processor which will complete processing it in minimum time and assign the column to the processor (lines 5-10). The AvailabilityTime of this processor is now updated to its expected completion time and its column counter Ncol is incremented (lines 11-12). When all the columns are thus assigned to the processors, Ncol gives the number of columns assigned to each processor.

In the first step, we had assumed that all the columns are available as a single pool. However, some of the participating processors (atmosphere processors) already possess some columns
(NcolAtm). Therefore, in the second step, a mapping for senders to receivers has to be determined. The number of columns a processor has to send or receive is the difference between the number of columns it originally possesses (NcolAtm for atmosphere processors and zero for others) and the number of columns assigned to it in the previous step (lines 17-18). A simple greedy technique is followed (see the while loops in the figure) to determine the processor communications. We start with a sender and a receiver, indexed respectively by SndIndx andRecvIndx, from the list of senders and receivers. While there are senders or receivers (lines 15-16), we transfer the maximum number of columns from the current sender to the current receiver (line 19), update the sender’s and receiver’s remaining number of columns to be communicated(line 20) and if the sender and/or receiver meets its target number of columns, we move to a new sender and/or receiver (lines 27-32). Note that the maximum number of columns in a transfer is the minimum of the number of columns that a sender can send and the number of columns a receiver can receive (line 19). As the transfers are determined, the processor that calls the function records the details of the transfers for which it is the sender or receiver (lines 21-26). The communicating processor’s id (CommProcs) and the number of columns to be transfered (CommCols) are noted for each transfer, and the number of such transfers (NumOfComms) is also returned.

Using this strategy, each processor determines the number of columns (amount of extension) and the processors (targets of extension) with which it has to communicate. For each chunk, it then involves in input communications, radabs computations and output communications, using the same mapping until the number of sharing processors change.

The pseudo codes of the modified algorithm described in this section are given in Figures 3.10 and 3.11, and can be compared with the pseudo code of the original version given in Figure 3.6.

Figure 3.12 demonstrates the potential of the Dynamic Component Extension (DCE) strategy in reducing the times taken for the computation intensive absorptivity calculations in the atmosphere. The figure shows the times spent by an atmosphere processor performing calculations between the receive and send communications with the coupler at different time steps. The spikes occurring at every third simulated hour correspond to the absorptivity calculations in the
Algorithm: Column-processor Mapping

Input: AvailabilityTimes, NumberOfAtmProcs, NumberOfOtherProcs
Output: NumOfComms, CommProcs, CommCols

NumberOfProcs = NumberOfAtmProcs + NumberOfOtherProcs

for col in 1 to NumberOfAtmProcs*NcolAtm do
    MinTime = ∞;
    for proc in 1 to NumberOfProcs do
        Time = AvailabilityTimes(proc) + RadabsColTime
        if Time < MinTime then
            MinProc = proc; MinTime = Time
    end
    Ncol(MinProc) = Ncol(MinProc) + 1
    AvailabilityTimes(MinProc) = AvailabilityTimes(MinProc) + RadabsColTime
end

SndIndex = 1; RcvIndex = 1; NumOfComms = 0

while SndIndex <= NumberOfAtmProcs do
    while RcvIndex <= NumberOfOtherProcs do
        NumOfSndCols = NcolAtm - Ncol(SndIndex)
        NumOfRcvCols = Ncol(RcvIndex)
        MaxCols = Min(NumOfSndCols, NumOfRcvCols)
        NumOfSndCols = NumOfSndCols - MaxCols; NumOfRcvCols = NumOfRcvCols - MaxCols
        if MyProcId == SndIndex then
            NumOfComms = NumOfComms + 1; CommProcs(NumOfComms) = RcvIndex; CommCols(NumOfComms) = MaxCols
        end
        if MyProcId == RcvIndex then
            NumOfComms = NumOfComms + 1 CommProcs(NumOfComms) = SndIndex CommCols(NumOfComms) = MaxCols
        end
        if NumOfSndCols == 0 then
            SndIndex = SndIndex + 1
        end
        if NumOfRcvCols == 0 then
            RcvIndex = RcvIndex + 1
        end
    end
end
return(NumOfComms, CommProcs, CommCols)

Figure 3.9: Column-Processor Mapping Algorithm
Algorithm: Modified algorithm

1. for each time-step do
2.   atmosphere model execution
3.     for each chunk do
4.       if radabs time-step then
5.         determine/update destination processor set
6.         if change in destination processor set or in time-step then
7.           send availability times to new component(s); calculate column mapping
8.           using Figure 3.9.
9.       end
10.      send radabs input data
11.      availability timer start
12.     end
13.    if radcswmx time-step then
14.      call radcswmx
15.     end
16.    if radabs time-step then
17.      availability timer stop
18.      call radabs with own column range; receive radabs output data
19.     end
20. end
21. compute average availability time
22. atmosphere model execution
23. end

Figure 3.10: Pseudo-code for Modified Atmosphere
Algorithm: Modified algorithm for components

for each time-step do
  component model execution
  if radabs time-step then
    recv availability times from atmosphere
    determine number of chunks using chunk sharing index
    for each chunk do
      determine/update destination processor set
      if change in destination processor set or in time-step then
        calculate mapping using Figure 3.9
      end
    end
    recv radabs input data; call radabs with own column range; send radabs
    output data
  end
end

communication with coupler

component model execution

Figure 3.11: Pseudo-code for Modified Non-Atmosphere Component

atmosphere. As previously mentioned, we execute the unmodified version of CCSM during the first simulated day to determine the points and amounts of extension and apply dynamic component execution only from the second simulated day. Hence the spikes corresponding to the first simulated day are higher than those for the other simulated days when DCE was applied. We find that DCE was able to reduce the computation time of absorptivity calculations by more than 35%.

3.4.4 Summary of Dynamic Component Extension in CCSM

We dealt with several key aspects of dynamic component extension strategy while extending the atmosphere component of CCSM to reduce the temporal load imbalance in the atmosphere component due to computationally intensive long wave radiation calculations. We first identified atmosphere as the component containing temporal load imbalance by conducting profiling runs. We then modified the codes of atmosphere and other components to share parts of the heavy computations in atmosphere. By using a 3-level load balancing strategy, we dynamically
determined various parameters of extension at different granularities. By periodically executing the CCSM application without our load-balancing and obtaining the times spent by the different components in different phases, we automatically and dynamically identified the idling periods in processors of other components and the times when the component containing temporal load imbalance is ready to extend its heavy computations to other component processors. At periodic intervals, we turn-off (using a conditional-if statement) the load-balancing part of the code and turn-on the profiling part of the code. After executing in this phase for one simulated-day, we turn-off the profiling and turn-on the load-balancing parts of the code. The load balancing strategy will then be using the profiled information for all simulated days until the next profiling phase. Thus we dynamically determine the load imbalances across components and the source of the imbalances using profiling phases of unmodified CCSM at periodic intervals. The idling periods and ready times were then used to determine the points, amounts and target processors of extension. Similar steps can be followed when applying our dynamic component extension strategy to other multi-component applications to improve the application performance.

3.5 Experiments and Results

We used five different platforms to evaluate our strategy and compare its performance with that of the original version of CCSM. The specifications of these platforms are given in Table 3.1.
<table>
<thead>
<tr>
<th>Platform Name</th>
<th>Number of Procs.</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fire-16</td>
<td>16</td>
<td>8 dual-core AMD Opteron 1214 based 2.21 GHz Sun Fire servers, CentOS release 4.3, 2 GB RAM, 250 GB Hard Drive and connected by Gigabit Ethernet</td>
</tr>
<tr>
<td>Fire-48</td>
<td>48</td>
<td>12x2 dual-core AMD Opteron 2218 based 2.64 GHz Sun Fire servers, CentOS release 4.3, 4 GB RAM, 250 GB Hard Drive and connected by Gigabit Ethernet</td>
</tr>
<tr>
<td>Regatta</td>
<td>16</td>
<td>AIX, 16-way SMP IBM pSeries 690 node with 16 POWER4 CPUs running at 1.3GHz and 160 GB memory with IBM AIX</td>
</tr>
<tr>
<td>Regatta</td>
<td>32</td>
<td>32-way SMP IBM pSeries 690 node with 34 POWER4 CPUs running at 1.1GHz and 64 GB memory with IBM AIX</td>
</tr>
<tr>
<td>Param</td>
<td>216</td>
<td>AIX, 54 4-way SMP IBM pSeries 630 nodes, each having 4 POWER4 CPUs running at 1 GHz and 8 GB memory with IBM AIX</td>
</tr>
</tbody>
</table>

Table 3.1: Platform Specifications

The first 3 systems are located in Indian Institute of Science (IISc) while the last 2 systems are located in Centre for Development of Advanced Computing (CDAC), Bangalore, India. On CDAC Param, out of the 54 systems, we used 16 systems of 64 processors for our experiments.

All the experiments were conducted with NCAR’s CCSM, CCSM3.0.1beta14. We have ensured that our modifications do not affect the scientific validity of the results, and verified that the results of the modified version and those of the original version match bit-by-bit. CCSM has an option for the users to enable load sharing mechanism for the short-wave radiation calculations. On each platform, we executed four versions of CCSM on various number of processors. The four versions are: (i) the original version which by default has the short-wave load-balancing turned off, (ii) the original version with short-wave load-balancing turned on, (iii) our modified version with the short-wave load-balancing turned off and (iv) our modified version with short-wave load-balancing turned on. Each of our experiments corresponded to a 30-day climate simulation. For each experiment, execution times were noted for each of the four versions. We mostly show results for versions i and iii. Similar results were obtained for versions ii and iv.
The restrictions on the number of processors on which each component of CCSM can execute is tabulated in Table 3.2. Note that although ocean is the most computationally intensive component next to atmosphere, in some configurations ice has a greater number of processors than ocean because of the model restriction on ice. When executing CCSM on a certain number of processors, the available number of processors has to be divided among the 5 components using a division strategy. The division strategy or configuration for a certain number of processors affects the performance of CCSM. In most of our experiments, when executing CCSM on a certain number of processors, we chose a configuration that satisfies the model restrictions shown in Table 3.2 and for which the original version of CCSM gave the best performance.

The results on the Fire cluster are shown in Figure 3.13. The numbers in the brackets represent the percentage improvement of our version over the original version of CCSM.

The results show that our version can give performance improvement up to 38% over the original CCSM version on small clusters. Our performance improvements with the short-wave load-balancing enabled, clearly show that load balancing short-wave radiation calculations alone, as was done in many previous research efforts, is not sufficient for improving the performance of CCSM. Much higher performance improvements can be obtained by load balancing long-wave radiation calculations.

As expected, the percentage improvement steadily decreases as the number of processors increase since most of the additional processors are added to atmosphere in order to obtain the best timing for the original version. This results in reduction in the ratio of the number of non-atmosphere processors to the number of atmosphere processors. This reduction not only

<table>
<thead>
<tr>
<th>Component</th>
<th>Restrictions on Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atmosphere</td>
<td>Depends on the dynamic core used. Must be even and have no prime-factors other than 2, 3 and 5</td>
</tr>
<tr>
<td>Coupler</td>
<td>No restrictions</td>
</tr>
<tr>
<td>Land</td>
<td>No restrictions</td>
</tr>
<tr>
<td>Ocean</td>
<td>Must be one of a resolution dependent set of values: 1, 2, 4, 8, 10, 12, 16, 20, 24, 32, 40, 48, 64, 80, 96</td>
</tr>
<tr>
<td>Ice</td>
<td>Cannot be 1; If less than or equal to 7, should be a factor of NLON, else should be even and the number divided by 2 should be a factor of NLON, where NLON is number of longitudes.</td>
</tr>
</tbody>
</table>

Table 3.2: Model Restrictions on Number of Processors
Figure 3.13: Results on Fire Cluster for Low Resolution
reduces the number of time-steps or chunks at which a component’s processor can be involved in long-wave radiation calculations, but also affects the percentage benefit that can be obtained due to offloading the calculations. Further, since the radabs calculations scale better than the rest of the model, as the number of atmosphere processors increases, the percentage of total time the processors spend in executing radabs decreases significantly. Also, other intra-component load-imbalances in the atmosphere become significant increasing the overheads due to synchronization between the atmosphere and other processors in our modified version. Hence, for very large number of processors, the overheads in our modified version due to synchronizations, communications, determination of mapping, etc. exceed the benefits due to load-balancing.

However, we contend that for many platforms, the percentage improvement of our version over the original CCSM version for smaller number of processors can be comparable to the percentage improvement obtained by executing the original CCSM version on higher number of processors. This is because the CCSM code does not scale well for higher number of processors and achieves saturation for certain platforms. This is illustrated by our results on CDAC Regatta with the short-wave load balancing enabled as shown in Figure 3.14(b). For this case, the percentage improvement of our version over the original version for 26 processors is 6.64% while the percentage improvement due to executing the original CCSM version on 32 processors over executing the original version over 26 processors is 8.8%. Thus, by using 6 extra processors, only an additional improvement of 2.2% is obtained. Also, in many batch systems like CDAC Regatta, usage of more processors lead to more queue waiting time that can offset the benefit obtained.

The results for CAOS Regatta and CDAC Param are shown in Figures 3.15 and 3.16, respectively. As can be seen from Figure 3.16, we obtained poor performance improvements on CDAC Param cluster. We later found out that the clocks on the processors of the cluster are out of synchronization by large magnitudes (up to 100s of seconds). This leads to calculation of incorrect availability times for processors resulting in sub-optimal load balancing of long-wave radiation calculations.

We also conducted few experiments with the highest resolution of CCSM, T85_gx1v3, and with short-wave load balancing disabled. The percentage improvements in execution times due
Figure 3.14: Results on CDAC Regatta for Low Resolution
Figure 3.15: Results on CAOS Regatta for Low Resolution
Figure 3.16: Results on CDAC Param for Low Resolution
Cluster | Processors | Improvements |
--- | --- | --- |
Fire | 8 | 19% |
Fire | 12 | 6% |
CAOS Regatta | 12 | 14.6% |
CAOS Regatta | 16 | 8% |

**Table 3.3: Results for High Resolution Runs**

<table>
<thead>
<tr>
<th>Platform</th>
<th>Procs</th>
<th>Original ( \text{radabs} ) Time (seconds) [version (i)]</th>
<th>DCE ( \text{radabs} ) Time (seconds) [version (iii)]</th>
<th>Gain in ( \text{radabs} ) due to DCE</th>
<th>Savings for a 1000-year Simulated Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fire-16</td>
<td>16</td>
<td>355.06</td>
<td>268.68</td>
<td>24%</td>
<td>13.63 days</td>
</tr>
<tr>
<td>Fire-48</td>
<td>48</td>
<td>144.598</td>
<td>102.676</td>
<td>29%</td>
<td>4.37 days</td>
</tr>
<tr>
<td>IISc Regatta</td>
<td>16</td>
<td>515.71</td>
<td>360.82</td>
<td>30%</td>
<td>19.3 days</td>
</tr>
<tr>
<td>CDAC Regatta</td>
<td>32</td>
<td>307.94</td>
<td>250.28</td>
<td>19%</td>
<td>3.02 days</td>
</tr>
<tr>
<td>CDAC Param</td>
<td>32</td>
<td>338.62</td>
<td>274.69</td>
<td>19%</td>
<td>4.36 days</td>
</tr>
</tbody>
</table>

**Table 3.4: Results on Various Platforms**

to offloading are shown in Table 3.3.

Table 3.4 shows the percentage reduction in execution time of \( \text{radabs} \) due to dynamic component extension (DCE) for different platforms and different number of processors \(^1\). The last column denotes the savings in execution time of CCSM for a 1000-year simulated run and is calculated by multiplying the savings obtained for a 30-day simulated run by 12000. The results in the last column are significant since CCSM is commonly executed for such multi-century runs. We find that using our strategy can lead up to 30% reduction in execution time of \( \text{radabs} \) and result in savings of 3-19 days in execution time of CCSM for a 1000-year simulations.

In order to compare the gain due to DCE gain across all platforms, we executed CCSM on 16 processors, the largest number of processors available on all platforms. The results on various platforms for executions on 16 processors are shown in Table 3.5\(^2\). The results show that although the percentage gains are almost constant across platforms, the number of days

---

\(^1\)CCSM does not scale well for larger number of processors since the maximum number of processors on which the atmosphere model can execute is limited by the number of latitudes (48 for low resolution).

\(^2\)Results for 16 processors for Fire-16 and IISc Regatta are different in Table 3.4 and 3.5 due to different CCSM configurations used. CCSM configuration of (Atmosphere-10,Ocean-2,Ice-2,Land-1,Coupler-1) was used for the corresponding results in Table 3.4.
Table 3.5: Results for the 16-processor configuration (Atmosphere-8,Ocean-4,Ice-2,Land-1,Coupler-1) on Various Platforms

<table>
<thead>
<tr>
<th>Platform</th>
<th>Original Execution Time [version (i)]</th>
<th>DCE Execution Time [version (iii)]</th>
<th>Percentage Gain</th>
<th>Savings for a 1000-year simulated run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fire-16</td>
<td>1315.426</td>
<td>1112.820</td>
<td>15%</td>
<td>28.14 days</td>
</tr>
<tr>
<td>Fire-48</td>
<td>1129.155</td>
<td>964.375</td>
<td>15%</td>
<td>22.89 days</td>
</tr>
<tr>
<td>IISc Regatta</td>
<td>1750.586</td>
<td>1459.620</td>
<td>17%</td>
<td>40.41 days</td>
</tr>
<tr>
<td>CDAC Regatta</td>
<td>2056.061</td>
<td>1704.762</td>
<td>17%</td>
<td>48.79 days</td>
</tr>
<tr>
<td>CDAC Param</td>
<td>2477.55</td>
<td>2095.74</td>
<td>15%</td>
<td>53 days</td>
</tr>
</tbody>
</table>

saved for a 1000-year simulated run increases with the decrease in processor speeds. We obtain maximum savings of 53 days for the CDAC Param platform with the slowest processors. Thus our strategy gives large benefits for low end clusters.

As mentioned earlier, the frequency of radiation computations is once every 3-12 hours of simulation. Latest studies [124] show that radiation computations should be done as frequently as possible preferably at the same frequency as the rest of the model. Figure 3.17 shows the effect of increasing radiation frequency on various execution times for a 30-day simulation obtained on the Fire-16 cluster with the same configuration as in Table 3.5. As can be seen in the figure, dynamic component extension results in decrease of radabs, physics and total execution times in all cases. Another interesting observation was that while the DCE execution times are better than the original execution times for all frequencies, the gain is significantly higher for the highest frequency. Our investigations showed that this is because more radiation calculations were extended to other component processors for large radiation frequencies.

Figure 3.18(a) illustrates the gain obtained with dynamic component extension in CCSM. The result was obtained with 30-day runs on the IISc Regatta cluster with the same configuration as in Table 3.5. The figure shows that the dynamic component extension during the radabs (absorptivity) calculations reduces the execution time of radabs calculations by 17%. The figure also shows that the overheads due to dynamic component extension are negligible. These overheads include communication of columns and availability times from the atmosphere to component processors, time-stamping, computation of availability times and mapping columns.
Figure 3.17: Effect of radiation frequency on DCE gain. Fire-16 cluster.

to processors.

Figure 3.18(b) compares the actual gain obtained due to dynamic component extension to the maximum gain for the same configuration and processors. This maximum gain is obtained using simplistic assumptions for extending \textit{radabs} calculations, namely, the entire \textit{radabs} calculations in the atmosphere processor can be extended to the other components and zero overheads for extension. The maximum gain is obtained using Equation 3.2 in Section 3.2. The figure shows that our implementation of dynamic component extension in CCSM is efficient and achieves about 74\% of the maximum achievable gain. We also find that the primary reason for not achieving the maximum gain is the non-availability of all components for extension at all \textit{radabs} points. Figure 3.19 shows that the ratio of actual gains to the theoretical gain limit is fairly stable with respect to the number of processors. The ratio of actual gain to theoretical gain is in general higher when the atmospheric load balancing is off because our technique addresses most of this load imbalance. However, there is a kink observed at 14 processors where the load balancing on curve has a better ratio. This is because the best configurations with load balancing on and load balancing off were different at this point unlike at other points. Note that
Figure 3.18: Overheads and Gain due to DCE. IISc Regatta cluster.
Figure 3.19: Fraction of gain harnessed using dynamic component extension on fire-16 cluster

Each point in the curve is obtained from two runs: original and DCE. The best configuration on the given number of processors for original run was obtained and the same used for the run with DCE. Since the atmosphere computations take different times with load balancing on and load balancing off, the best configuration for the same number of processors may be different for the two cases.

Based on experimental timings and scalability of various components, we have built an execution model for CCSM. Using this model, we performed simulations to find the execution times for the original and the DCE version and compute the percentage gain due to DCE for various application configurations. As mentioned earlier, the CCSM application consists of a dynamics and a physics phase in each time step. We applied the DCE strategy to the physics phase of CCSM. Current efforts by the CCSM community [103, 104] are to increase the scalability of dynamics phase. As dynamics becomes more scalable, the percentage execution time of the physics phase increases. Thus our DCE strategy that optimizes the physics execution time will lead to increasing reduction in overall execution time of CCSM. This is illustrated in Figure 3.20 where the overall gain for the application is shown for 64 and 128 processors for
Figure 3.20: Simulation results showing effect of scalability in dynamical core on DCE gain

increasing scalability of dynamics. Parallel fraction of dynamics on the x-axis is the fraction of the dynamics computations that is parallelizable. If the parallel fraction of dynamics is \( f \) and \( t \) is the time taken for dynamics computations on a single processor (sequential execution time), then the time taken for execution of dynamics on \( N \) processors would be \((1-f)t + ft/N\). Higher the fraction \( f \), lower the time taken for a large \( N \) and lower the fraction of atmospheric computation time spent in dynamics computations. Therefore, a higher parallel fraction for the dynamics computations implies a higher fraction of atmospheric computation time spent in physics computations. Since a larger fraction of atmospheric computation time is optimized with DCE, the resultant gain percentage is naturally larger.

3.6 Summary

In this chapter, we proposed a software strategy for multi-component applications called Dynamic Component Extension for minimizing processor idling and improving application performance. We demonstrated the potential of this strategy by applying it to improve the performance of the most prominent multi-component application, CCSM. By automatic identification
of idling points and dynamic determination of points and amounts of extension, we extended the computation intensive long wave radiation calculations of the atmosphere component to the processors executing other components. By evaluating our strategy on 5 different parallel platforms for different number of processors, we showed that our strategy can reduce CCSM execution times by about 15% on a 16-processor system (queue or cluster) and save several days of execution for 1000-year simulation runs. We also showed that the overheads due to our strategy are negligible and that our strategy can give increasing benefits with increasing frequency of radiation calculations and increasing scalability of dynamics phase of CCSM. Our strategy can enable climate researchers to conduct multi-century climate runs at a reasonable time even on moderately sized clusters common in academic institutions.

The CCSM community has been advocating and working towards increasing the frequency of radiation calculations, adopting highly scalable dynamic cores and using larger resolutions. These will lead to larger percentages of execution times for long wave radiation calculations and increased significance of our dynamic component extension strategy in optimizing the radiation calculations.

The DCE strategy, that has been applied and demonstrated for execution on single clusters, also has the potential to improve the throughput of multi-component climate simulations on grids. Executions of different components on different clusters and sites of a grid can lead to greater load balances across the components due to heterogeneity and load fluctuations in the grid resources. The DCE strategy and the associated load monitoring discussed in this chapter can be extended to include monitoring of load variability and the component performance on each site, and to analyze the trade-offs in shipping the workloads across slow networks from a heavily loaded to lightly loaded components executing on two different sites. Thus DCE can be applied to executions on grids by considering both application and resource dynamics of the grid resources.
Chapter 4

Simulation Studies

Grids can be used for providing high throughput for coupled climate modeling applications by executing different components at different sites. This will help in leveraging the larger number of resources from different sites. This also helps avoid the resource bottlenecks created by single cluster executions of climate models. However, such executions can impose overheads, thereby negating the benefits of grid executions of climate models. Also, there are various system parameters like the external load, number of sites and bandwidths that can affect the performance. Thus, it is not clear upfront if grid executions will give benefits in practice. Thus, prior to building grid middleware frameworks that typically involve significant implementation effort, it is essential to estimate the possibility and the amount of benefits. We have developed robust simulation frameworks to analyze the benefits through simulations of executions on grids. In this Chapter, we present the various simulation studies we performed to evaluate various aspects of multi-site executions.

4.1 Introduction

Parallel batch systems provide space sharing of available processors among multiple parallel applications or jobs. These batch systems employ queues in which the incoming parallel applications are queued before allocation by a batch scheduler to a set of processors for execution. Thus a batch system is associated with a set of queues and a scheduling policy that selects a
job from the queue and maps it to a set of processors. An application submitted to a batch system incurs additional time for waiting in a queue before actual execution. The overall response time of an application is the sum of its queue waiting time and execution time. Application with small processor requirements can be backfilled to the available processors and hence incur smaller queue waiting times than applications with large processor requirements. Thus the queue waiting times of the jobs submitted to a batch system are proportional to their processor requirements as illustrated in Fig. 4.1. The figure shows the average queue waiting times for jobs with different processor requirements on two IBM systems in SDSC (San Diego Supercomputer Center). The job traces were obtained from the logs maintained by Feitelson[56]. We find, that except for some outliers, the general trends in the graphs show that the queue waiting times increase with request sizes.

Consider a parallel application $J$ needing $P$ processors that consists of or can be decomposed into multiple sub parallel applications or components, $J_1, J_2, \cdots, J_n$ with processor requirements, $P_1, P_2, \cdots, P_n$, respectively, such that $P = P_1 + P_2 + \cdots + P_n$. In this case, simultaneous submission of the multiple sub parallel applications, $J_1, J_2, \cdots, J_n$, to multiple batch systems of a batch grid can result in improved response times of the application over submitting the entire parallel application, $J$, to a single batch system since the maximum of the queue waiting times in the former case can be lesser than the single queue waiting time in the latter case. While this advantage of simultaneous submissions of the sub applications to multiple batch systems is generally well understood[26], the actual improvements in response times of the applications depend on various factors.

Application Factors

1. **Decomposability of Parallel Applications:** Some parallel applications like the NAS Parallel Benchmarks (NPB)[5] have absolute processor requirements and hence cannot be decomposed into sub applications with arbitrary processor requirements. On the other hand, bag-of-task applications like Parameter Sweep Applications can be decomposed arbitrarily. Yet other applications like the multi-component CCSM by design have well-defined components.

2. **Communication Characteristics of the Applications:** Tightly-coupled parallel appli-
Figure 4.1: Average queue wait times for jobs on two SDSC systems
cations involve heavy communications and synchronizations among the parallel tasks or processes. Decomposing such applications into smaller parallel applications and submitting to different batch systems will result in frequent and heavy communications between the batch systems. Since these batch systems can be located in different sites of a grid, the heavy communications between the sub applications will lead to larger execution times of the applications than when executed on a single system. Thus the overall response time of the application can significantly increase in spite of the reduction in queue waiting times. Moreover, previous studies have shown that executions of applications with large communication requirements on multiple clusters of grids will lead to performance degradations[6, 18]. Bag-of-task applications like the Parameter Sweep Applications (PSA) have almost no synchronizations or communications and are therefore most suitable for multiple-site execution.

Multi-component MPMD applications[29, 38] consists of component applications which are parallel application themselves. In these applications, the components are loosely synchronized and communications between components are lighter and less periodic than within components. These multi-component applications and parameter-sweep applications, where the parallel tasks are independent, are amenable to decomposition into multiple parallel sub applications and can potentially benefit due to submissions of the sub applications to multiple batch systems.

System Factors

1. **Interconnections Between the Batch Systems:** Processors in a single batch system are connected by high speed links while processors of different batch systems are typically connected by WAN links and other high latency networks. The greater the speed of the interconnects, the more the probability of improved response times of the applications due to multiple batch submissions.

2. **Queue Characteristics:** As mentioned earlier, the benefits due to multiple batch submissions of parallel sub applications are because of the differences between queue waiting time of a single batch submission and the maximum of queue waiting times of multiple
batch submissions. These queue waiting times can be different in different batch systems depending on queuing and scheduling policies followed in the batch systems, the loads in the queues, the job distribution in terms of their processor requirements and maximum execution times associated with the queues.

In this chapter, we describe various types of simulations we have conducted to evaluate the performance of application executions on batch grids.

In the first set of experiments, we have studied the effects of these different factors on the potential improvement in performance of a parallel application due to simultaneous submissions of the parallel sub applications to multiple batch systems of a grid over submission of the entire parallel application to a single batch system. At the application-level, we used two simulators, one that models the most prominent multi-component application, CCSM[29, 38], and the other that models parameter-sweep applications. At the network-level, we simulated different interconnection speeds between the batch systems. At the batch-level, we used different job traces produced from Feitelson’s job models[93] and containing different distributions of jobs with different processor, execution time requirements and execution time limits. We then generated the queue waiting times of the jobs by using three different queuing policies, namely, FCFS, conservative and easy backfilling. We then performed large number of simulations with different distribution of processors to components and system configurations with 24 different queues. We show that there are a large number of configurations that lead to performance improvement of the applications. We further performed real experiments with CCSM by executing the components of CCSM across two AMD Opteron clusters and show similar benefits.

We assume that sub-components executed on different batch systems can communicate with each other. For simplicity, we evaluate the advantages of batch grids using two queues. Thus, our results show the comparison between executing the entire parallel application with $P$ processor requirements on a single batch system and executing two sets of sub-components of the parallel application with processor requirements, $P1$ and $P2$ ($P = P1 + P2$), respectively, on two batch systems of a batch grid.

In the next set of experiments, we propose and evaluate various execution models for execution of long-running multi-component applications. Finally, we discuss the design of a
multi-queue execution simulator with the best execution model and performed further simulation studies.

4.2 Batch Grids - Motivations and Contributions

Application jobs submitted to batch systems incur overheads associated with the times spent in the batch queues waiting for resources to become available for execution. These queue waiting times tend to be higher when greater number of processors are requested. The central idea of this work is to split the application into components and execute them on different batch systems. The smaller requests for the components on the different batch systems are expected to have lower wait times than the single complete request, leading to potential gains in the overall application execution rates. The focus of this work is to study the incidence of this gain and its variation with different application and system factors. We have developed a simulator framework consisting of multiple components for studying the potential gains due to multiple batch executions for different application and system configurations.

The primary contributions of our work are:
(1) investigation of the benefits of execution of applications on batch grids over execution on batch systems,
(2) development of a simulation framework, including development of an even-based batch-system simulator, inter-site application execution model and a trace-based probability calculator,
(3) definition of new probability metrics for comparison of batch-grid vs batch-system executions of long-running applications, and
(4) large-scale analysis of incidence of benefits due to execution on batch grids for various system and application configurations.

For our study, we have considered two very important categories of long-running applications: (i) coupled multi-component applications, and (ii) parameter sweep applications. Different coupled multi-component applications, would have different application communication
and computation characteristics. We have focused on a specific real multi-component application, CCSM, because of its significance. We have considered the category of parameter sweep applications to generalize our results to a wider range of applications. Parameter sweep applications do not involve inter-component communications, hence have a simple execution model and the corresponding results, therefore, are generic.

The following section describes in detail our simulation framework.

4.3 Simulation Methodology for MCA and PSA Simulation Studies

We have developed a simulator framework consisting of multiple components for studying the potential gains due to multiple batch executions for different application and system configurations. Our simulation framework, shown in Figure 4.2, consists of four components: (i) workload generator, (ii) application simulator, (iii) batch system simulator and (iv) probability calculator. The workload generator is used to generate job traces with job arrival times, job execution times and job processor request sizes. The application simulator estimates the application execution rates for various intra-site and inter-site distributions of components. This is used in the calculation of our comparison metric, SPW (simulated time per wall-clock time) described in the next section. The batch system simulator is our event-based simulator that processes the job traces produced by the workload generator and outputs the queue waiting time for each job. The new traces with the queue waiting times are used by the probability calculator along with the application simulator, to estimate the probabilities of multiple-site executions outperforming single-site executions. The details of the parameters and calculations involved in the different components are given in the next section.

The following subsections explain the various components of the simulation architecture.

4.3.1 Workload Generator

For our simulation studies, we used the workload model developed by Lublin and Feitelson[93]. Using a workload model enables us to vary workload characteristics of the batch queues and
study their effect on the performance of a parallel application when executed across multiple batch queues. This model was developed by applying rigorous statistical procedures to logs collected from real batch systems of three different locations and was shown to be the most representative model available in a general sense. The workload model generates a job trace consisting of arrival times, processor requirements and execution times of the jobs. Job processor requirements, runtimes and arrivals are modeled based on a two-stage uniform distribution, a hyper-Gamma distribution and a Gamma distribution, respectively. The model parameters are preset to values representative of real logs of supercomputers. The workload generator can be used to generate job traces with a range of processor requirements, execution times, a specific and mean inter-arrival times.

### 4.3.2 Batch System Simulator

The job traces of the workload generator are input to the batch system simulator component of our simulator architecture. The batch system simulator uses the job traces with arrival times, processor requirements and execution times of the jobs along with a batch queue scheduling policy to generate *batch queue traces* that contain information regarding queue waiting times of the jobs, namely, the times of submissions of the jobs to the queue and the times of beginning of executions. For batch queue scheduling to generate *batch queue traces* with queue waiting times, we simulated three queue scheduling policies, namely, FCFS, conservative (CONS) and EASY backfilling[55], for scheduling jobs in each of the job traces.

![Figure 4.2: Simulation Framework](image-url)
4.3.3 Application Simulator

In order to calculate statistics regarding the amount of work performed by an application corresponding to a job in a batch queue within a specified execution time, our simulator architecture uses simulations of the application to estimate the amount of work performed by the application in a given amount of time. We have developed two application simulators corresponding to two applications, a multi-component parallel application and a parameter sweep application.

Multi-Component Applications

Coupled multi-component applications [42, 38, 144, 90] are an important class of scientific applications and have gained importance in recent times due to rapid advancements in computational science and multi-disciplinary simulations. One classic and foremost example of this application category is coupled climate models. In particular, we consider a coupled multi-component climate simulation application, the Community Climate System Model (CCSM)[29], an MPMD application developed by NCAR (National Center for Atmospheric Research). CCSM involves multiple components, four components corresponding to the four climate sub-systems: atmosphere, land, ocean and sea-ice, and a fifth component called coupler for coordinating the periodic communications between the other components.

For developing a simulator for CCSM, we executed CCSM on a cluster where the links of the cluster have a certain bandwidth denoted as base intra-cluster bandwidth. For the distribution of processors to the different components of CCSM for an execution on a given number of processors, we followed the general guidelines[24, 25] and processor restrictions for the components. These restrictions include even number of processors for atmosphere and more than one processor for ice component. We then observed the component computation times between their coupling period (period between communications with the coupler) and the component communication times with the coupler for different executions of CCSM on different number of processors and different distributions of processors to components.

We then simulated the execution time of CCSM across two parallel systems or clusters where the intra-cluster communication networks have bandwidths of base intra-cluster bandwidth and the two clusters are connected by a single link of a specified bandwidth, denoted
as *inter-cluster bandwidth*. For modeling communications between two components located at different clusters, we scaled the component-coupler communication times, that were earlier observed from real executions on a cluster with *base intra-cluster bandwidth*, based on the specified bandwidth of the inter-cluster link. For example, for simulating CCSM execution on a 1 Mbps inter-cluster link based on observations of real execution on a cluster with Gigabit Ethernet network, we scaled down the component communication times with the coupler observed on the Gigabit network by a factor of 1000.

**Parameter Sweep Applications**

Parameter sweep applications (PSA) constitute yet another important class of scientific applications. Since the individual tasks of these applications are independent and do not involve communications, these applications have been widely demonstrated and deployed on grid systems.

The simulator for parameter sweep applications consider the execution of a parameter sweep application consisting of P tasks. For simulation of the application when executed on two queues, we divided P into different combinations of P1 and P2 tasks such that P1+P2=P. Unlike in CCSM, the execution times of the parameter sweep application are same when executed on a single batch system and on multiple batch systems since the tasks in the different batch systems do not communicate. Thus the bandwidth of the link connecting the two systems does not affect the execution times of the applications. Hence we do not model the execution times of the applications in our simulator. The difference between single batch and multiple batch executions of the parameter sweep applications are only in terms of the queue waiting times. For a given queue trace, we obtain the gain due to the multiple batch execution of parameter sweep applications as the difference in average of the queue waiting times of the jobs with P processors in the trace and the maximum of the averages of the queue waiting times of jobs with P1 and P2 processors. We choose the combination of P1 and P2 that yields the maximum gain for a given total number of tasks, P.
4.3.4  Statistics Calculator

The statistics calculator uses the batch queue traces along with the estimates from the application simulator to calculate statistics for the applications. The kind of statistics that are relevant for an application depends on the kind of the application. We obtain different kinds of statistics for our two applications.

Statistics for Multi-Component Applications

CCSM is typically executed for long periods to simulate climate systems for multiple centuries. The execution times for such runs can be several weeks. However, batch queue systems have execution time limits of few days for job executions. CCSM provides restart facilities where simulations for an execution can be continued from the previous executions. Hence, for multi-century simulations, CCSM is submitted to a batch system and resubmitted to the system after the previous batch execution. In order to compare the execution of such long-running CCSM when executing all components in a single batch system and when executing the components in multiple batch systems, response time of an application is not a suitable metric for evaluation since it is used for jobs that can complete execution within the maximum execution limit of a batch system. We use simulated time per wall clock time, $SPW$, as a metric for comparison. For CCSM, $SPW$ is the number of simulated days for a particular batch submission and is obtained by dividing the number of days, $simulatedDays$, that can be simulated by CCSM within the maximum execution time limit for a CCSM job in a queuing system, by the sum of queue waiting time for the job, $qwait$, and the maximum execution time limit for the queue, $execTimeLimit$.

\[ SPW = \frac{simulatedDays}{(qwait + execTimeLimit)} \]  
(4.1)

Thus, the $SPW$ metric accounts for the amount of effective work performed in CCSM by considering the queue waiting time and the communication costs for multiple batch executions.

We used different configurations of CCSM and different queue traces for comparing single and multiple batch executions of CCSM. Each CCSM configuration corresponds to a total number of processors, $P$, for CCSM execution and distribution of the $P$ processors to the components. For each queue trace, we used a single queue with the queue trace characteristics for
a single batch execution and two queues with the same queue trace characteristics for multiple batch executions of different components of CCSM. For each CCSM configuration with \( P \) total number of processors and for each queue trace, we determine the best decomposition of the application with \( P \) processor requirements into two sub-applications with processor requirements, \( P_{1_{\text{best}}} \) and \( P_{2_{\text{best}}} \), such that \( P_{1_{\text{best}}} + P_{2_{\text{best}}} = P \). We then calculate the probabilities that multiple batch executions for the best decomposition with \( P_{1_{\text{best}}} \) and \( P_{2_{\text{best}}} \) processor requirements, will provide benefits over single batch executions with \( P \) processor requirements. For determining the best decomposition for multiple batch execution with \( P \) total number of processors, we use \( SPW \) values as follows.

We first calculate \( SPW \) for the single-batch execution, \( SSPW \), using Equation 4.1. The maximum execution time limit for the queue, \( \text{execTimeLimit} \), used in the equation is the input to our simulator. The number of days, \( \text{simulatedDays} \), that can be simulated by CCSM within the maximum execution time limit in the equation is estimated for \( P \) processors using the CCSM application simulator described in Section 4.3.3. For obtaining the queue waiting time, \( q_{\text{wait}} \), in Equation 4.1 for a submission with \( P \) processor requirements, we used the average of queue waiting times in the queue trace for jobs using \( P \) processors as the queue waiting time for the single batch submission.

For multiple batch executions with a given CCSM configuration of \( P \) total number of processors, and a given queue trace, we evaluate different combinations of locations of the components in the two queues. A particular combination of locations of CCSM components in the two queues corresponds to a particular decomposition of total number of processors, \( P \), into processor requirements, \( P_1 \) and \( P_2 \), for the two queues, based on the number of processors allocated to each component in the CCSM configuration. For this particular decomposition with \( P_1 \) and \( P \) processor requirements, we calculate \( SPW \) for the multiple-batch execution, \( MSPW \), using Equation 4.1. Similar to the calculation of \( SSPW \), the \( \text{execTimeLimit} \), used in the equation is the input to our simulator and \( \text{simulatedDays} \) in the equation is estimated for multiple batch execution with \( P_1 \) and \( P_2 \) processor requirements on the two queues using the CCSM application simulator described in Section 4.3.3. For obtaining the queue waiting time, \( q_{\text{wait}} \), in Equation 4.1 for multiple-batch submissions on the two queues, with processor requirements,
P1 and P2 \( (P = P1 + P2) \), we obtained the average of queue waiting times for the jobs using P1 processors and the average of queue waiting times for the jobs using P2 processors in the queue trace and used the maximum of the two averages as the queue waiting time for the multibatch submission. The maximum of the two average queue waiting times is considered because the application can start execution only after both the sub applications are dequeued from their respective queues and allocated processors for execution by their respective batch schedulers.

We then obtained the gain, \( MGAIN \), due to multiple batch submissions for the particular decomposition, \( (P = P1 + P2) \), as the difference between \( MSPW \) and \( SSPW \) values. For a given CCSM configuration with \( P \) processors, we experimented with different decompositions, and chose the decomposition, \( P = P1_{\text{best}} + P2_{\text{best}} \), for which \( MGAIN \) is maximum.

For a given CCSM configuration requiring \( P \) total number of processors, a given queue trace and the best decomposition with \( P1_{\text{best}} \) and \( P2_{\text{best}} \) processor requirements on the two queues for multiple batch executions, we calculate the probability for \( MSPW \) to be greater than \( SSPW \) value. We obtain the probability instead of an average measure because queue waiting times do not follow a strict non-decreasing relationship with the number of processor requirements. In some cases, jobs with smaller processor requirements can incur higher queue waiting times than jobs with larger processor requirements. The probability is calculated as follows.

We divided the queue waiting times of the jobs with \( P \) processor requirements in the queue trace into 10 intervals, \([L1, U1], [L2, U2], \ldots, [L10, U10]\). For each interval, \( i \), we calculate two probabilities, \( p1(i) \) and \( p2(i) \), corresponding to single and multiple batch executions, respectively. For the interval, \( i \), we also calculate the upper bound for \( SSPW \) with \( P \) processor requirements, denoted as \( SSPW_u \), using \( L_i \) as the queue waiting time in Equation 4.1. \( p1(i) \) is the probability that the queue waiting time of a job with \( P \) processor requirements is in the interval \([L_i, U_i]\). For multiple batch executions, we find queue waiting time, \( L'_i \), satisfying the equation:

\[
\frac{\text{simulatedDays}}{(L'_i + \text{execTimeLimit})} = SSPW_u
\]  

(4.2)

In the equation, \( \text{simulatedDays} \) is the estimated number of simulated days within the maximum execution time limit for multiple batch execution with \( P1_{\text{best}} \) and \( P2_{\text{best}} \) processor requirements on the two queues. We calculate \( p2(i) \) as the probability that the queue waiting time
of the jobs with requirements of $P1$ or $P2$ processors is less than $L_i'$. For a given interval $i$, and for a particular multiple batch submission, if the queue waiting time for the submission is less than $L_i'$, shown in Equation 4.2, then the $MSPW$ for the multiple batch submission will be greater than the $SSPW$ of all single batch submissions whose queue waiting times are in the interval $[L_i, U_i]$. This is because the $SSPW_u$ in Equation 4.2, used for the calculation of $L_i'$, is the upper bound of all these queue waiting times.

We thus calculate the joint probability, $p(i) = p_1(i) \times p_2(i)$, for multiple batch submissions to provide benefits over single batch submissions if the queue waiting time of the single batch submission with $P$ processor requirements is in the interval $[L_i, U_i]$. We then obtain the total probability for a multiple batch submission to provide benefits over single single batch submission by adding the probabilities $p(i)$ for all intervals. In these calculations, to obtain the probability that a queue waiting time for a job with a given processor requirements, $N$, is in an interval or less than the lower bound of the interval, we used the queue trace and divided the number of jobs with $N$ processor requirements and whose queue waiting times satisfy the condition by the total number of jobs with $N$ processor requirements.

**Statistics for Parameter Sweep Applications (PSA)**

To obtain probabilities for the parameter sweep applications (PSA), we experimented with different configurations of the application corresponding to different number of tasks. For each configuration with $P$ number of tasks and a given queue trace, we decomposed the $P$ tasks into the best decomposition with $P_{1\text{best}}$ and $P_{2\text{best}}$ ($P_{1\text{best}} + P_{2\text{best}} = P$) tasks for multiple batch executions on two queues, such that the queue waiting time for the multiple batch submission is minimum for the best decomposition. We then calculate the probability that decomposing the $P$ tasks into $P_{1\text{best}}$ and $P_{2\text{best}}$ tasks using the best decomposition and submitting to multiple batch systems will give benefit over submitting the entire application with $P$ tasks to a single batch system using similar probability calculations used for CCSM application. We divided the queue waiting times with $P$ processor requirements into 10 intervals, and calculated probabilities $p_1(i)$ and $p_2(i)$, corresponding to single and multiple batch executions, respectively, for an interval $i$. For PSA, $p_2(i)$ is the probability that the queue waiting time of the jobs with requirements of
$P_1$ or $P_2$ processors is less than the lower bound, $L_i$, of the queue waiting time for the interval. We then calculate the joint probability, $p(i) = p_1(i) \times p_2(i)$, for interval, $i$, and obtain the total probability, across all intervals, for a multiple batch submission to provide benefits over single single batch submission.

For CCSM application, as described in the previous section, the probability of multiple batch executions providing benefits depends on both the difference in queue waiting times between multiple and single batch submissions, and the cost of communications between the two queues. However, for PSA, the probability depends only on the difference in queue waiting times between multiple and single batch submissions since PSA does not involve communications between the tasks. Thus a reduction in queue waiting times will lead to reduced response time in multiple batch executions for any parameter sweep application.

### 4.4 Simulation Setup for MCA and PSA Simulation Studies

In this section, we describe the specific parameters we used for our simulations using the simulator architecture.

In our workload generator, we specified the maximum processor requirement of the jobs as 128 processors and maximum execution time of 2 days. In order to generate job traces of different job characteristics, we categorize the jobs in terms of their execution times and processor requirements. We call jobs with small execution times (mean execution time of 3-4 minutes) as short jobs (S), and those with large execution times (mean execution time of 6 hours) as long jobs (L). Similarly, jobs with small processor requirements ($< 10$ processors) are called narrow jobs (N) and those with large processor requirements ($> 10$ processors) as wide jobs (W). When then tuned the input parameters of the workload model to generate job traces consisting of predominant number of jobs belonging to one of the 8 job categories, namely, S, L, N, W, SN, SW, LN, and LW. The mean inter-arrival times of the jobs in our queue traces were 3000 seconds. We thus generated 24 different queue traces corresponding to 8 job characteristics distributions and 3 scheduling policies. For all the queue traces, we fixed the maximum execution time limit of the jobs as 2 days. During our simulations of submitting a sub application with a processor
requirement to a batch system corresponding to one of the queue traces, we used the average of the queue waiting times of the jobs with the same processor requirement in the queue trace as the queue waiting time of the sub application.

For CCSM application simulator, we executed CCSM with 2X2.5-gx1v3 resolution and finite-volume (FV) dynamical core for atmosphere component on 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. For our simulations of CCSM using two batch submissions, we used bandwidths of 700 Kbps, 10 Mbps, 100 Mbps and 1 Gbps on the links connecting the two submissions. The first three bandwidths are commonly observed on the links connecting two clusters located at two different sites in many grid systems. The last bandwidth is seen on the links connecting two batch systems in a single site and on the links connecting two different submissions in a single batch system. Simulating this scenario is important since we claim that submitting multiple sub applications of a single application to the same batch queue can also lead to benefits.

For obtaining statistics for CCSM, we used 50 configurations of CCSM corresponding to different total number of processors and different distribution of processors to components. For obtaining statistics for parameter sweep applications, we experimented with different configurations of the application corresponding to different number of tasks (2 to 128).

### 4.5 Results of MCA and PSA Simulations

In this section, we show results corresponding to simulations of CCSM and parameter sweep applications for different queue traces and bandwidths between two batch systems. We also show results for a real inter-cluster configuration where we obtained actual execution times of CCSM on a real configuration and used these times with our queue traces. All the results show probabilities for benefits with executing on two batch systems over executing on a single system. The probabilities were obtained using the calculations described in Section 4.3.
4.5.1 Simulations of Multi Component Applications

Fig. 4.3 shows the fraction of configurations corresponding to different probabilities of benefits with multiple batch executions for the four bandwidths. The total number of configurations were 1200 corresponding to 50 CCSM configurations and 24 queue traces for the four different bandwidths. These bandwidths are 1 Gbps, 100 Mbps, 10 Mbps and 700 Kbps, bandwidths typical of links connecting two submissions to a single batch queue, clusters in a single site, two different sites in a country, and two different sites located in two different continents, respectively.

We find that there are large percentages of configurations with significant probabilities of benefits with multiple batch executions. For example, for 100 Mbps bandwidth, 50% of the total configurations have about 58% probability of obtaining benefits when executed across two batch queues. About 48% of the total configurations have about 50% probability for inter-cluster connection bandwidth of 10 Mbps. These two bandwidths correspond to queues located
in different locations of a country. We also find that even for batch grids with the two batch queues located in different continents, 20% of the configurations have up to 38% probability of obtaining benefits with multiple batch executions. For 10 Mbps, 100 Mbps, and 1 Gbps interconnection bandwidths, significant percentage of configurations (about 15%) have close to 100% probability and hence have definite chance of obtaining benefits with executions on batch grids.

In Fig. 4.3, we also note that the characteristics for the 700 Kbps curve is very different from the other three. As can be observed in the figure, with the other three bandwidths, the probability of inter-site benefits being at least 50% is observed in around half or more of the cases. However, with the 700 Kbps bandwidth, the same is observed in only one-tenth of the cases. This indicates that the results are biased towards single-site executions for the 700 Kbps curve, while they are biased towards multi-site executions in the other cases involving higher bandwidths and therefore lower inter-site communication overheads.

In order to understand the effects of the system parameters including queue scheduling policies and job types, we show the probabilities for all 1200 application and system configurations for 100 Mbps inter-cluster bandwidth in Fig. 4.4. This figure gives an indication of the most frequent probabilities for each of the two system parameters. Fig. 4.5 shows the average probabilities obtained for the 50 application configurations for each of the 24 system configurations.

In general, we find that the probability of benefits increases when the queues have predominantly short and narrow (SN) jobs. This is because in such queues, at any given time, most of the available processor space will be occupied by the narrow jobs. The remaining processor space will not be sufficient for accommodating the larger processor requirements for single batch execution leading to large queue waiting times. However multiple batch executions with smaller processor requirements incur small queue waiting times in these cases since jobs with smaller processor requirements have better chances of finding the required processors on completion of other short and narrow jobs than the jobs with larger processor requirements. However, the probabilities of benefits with multiple batch executions on queues with predominantly long wide jobs (LW) are small since these wide jobs delay the executions of CCSM for long durations equally on both single and multiple batch queues in spite of decompositions in multiple
Figure 4.4: Probability of Benefits with Multiple Batch Submissions for all Application and System Configurations for CCSM Simulations
Figure 4.5: Average Probability of Benefits with Multiple Batch Submissions for Different System Configurations for CCSM Simulations

batch executions. The extra cost of inter-cluster communications in multiple batch executions result in less benefits for the application. We also find that for queues with backfilling policies, the probabilities of benefits with multiple batch executions are higher. This is because the sub applications in multiple batch executions with small processor requirements have better chances of backfilling in these queues than the single application with total processor requirements.

4.5.2 Real Executions of Multi Component Applications

We also executed CCSM on real experiment testbed with 28 configurations and obtained real execution times. We then used these 28 real application execution times with the queue waiting times of the 24 queue traces to calculate probabilities. Our real experiment testbed consists of a 16-core cluster called fire-16 and a 48-core cluster called fire-48. The fire-16 cluster consists of 8 dual-core AMD Opteron 1214 based 2.21 GHz Sun Fire servers, CentOS release 4.3, 2 GB RAM, 250 GB Hard Drive and connected by Gigabit Ethernet. The fire-48 cluster consists of 12x2 dual-core AMD Opteron 2218 based 2.64 GHz Sun Fire servers, CentOS release 4.3,
4 GB RAM, 250 GB Hard Drive and connected by Gigabit Ethernet. The Gigabit clusters are connected to each other by a 100 Mbps Ethernet connection through a 100 Mbps switch. For the single-batch execution, we executed the 28 configurations on the fire-48 cluster. For multiple batch executions, we randomly distributed the different components of CCSM to the two clusters.

Figs. 4.6, 4.7 and 4.8 show the results obtained for the real executions of CCSM. Similar to the simulation results, the results obtained with real executions show large number of application and system configurations with significant probabilities of benefits with multiple batch submissions. We also find that the probabilities improve when backfilling scheduling policies are used.
Effect of Queues on Probabilities of Benefits due to Multiple Batch Execution for CCSM Real – Job–wise

Figure 4.7: Probability of Benefits with Multiple Batch Submissions for all Application and System Configurations for CCSM Real Executions
Figure 4.8: Average Probability of Benefits with Multiple Batch Submissions for Different System Configurations for CCSM Real Experiments

4.5.3 Simulations of Parameter Sweep Applications

Figs. 4.9, 4.10 and 4.11 show the results obtained for the simulations of different configurations of parameter sweep applications corresponding to different total number of tasks of the application. The total number of tasks were varied from 2 to 128.

Similar to earlier results, we find large number of configurations where significant benefits were obtained with multiple batch submissions. We also find that unlike in CCSM, multiple batch executions gave higher probability of benefits for queues containing predominantly wide jobs for PSA. In most of the PSA configurations, the total number of tasks were divided equally across the two batch systems. Hence the processor requirements in a single batch system for multiple batch submissions are larger in PSA than in CCSM. The larger processor requirements in PSA can be frequently met by the completion of wide jobs than by the completion of narrow jobs. Hence higher benefits with multiple batch submissions were obtained in queues with larger number of wide jobs than in the queues with large number of narrow jobs where the processor space is mostly occupied by the narrow jobs. We also find that the probabilities obtained with PSA with independent tasks are lesser than the probabilities obtained with CCSM that
Figure 4.9: Percentage of Configurations with Probabilities of Benefits for PSA Simulations
Figure 4.10: Probability of Benefits with Multiple Batch Submissions for all Application and System Configurations for PSA Simulations
involves communications between two batch systems. This is due to the completely different configurations in terms of total number of tasks used for CCSM and PSA.

Our primary focus in our experiments is on the incidence of benefits due to multiple batch executions rather than on the measure of benefits. This is because the benefits are spread over a wide range of both positive and negative values due to the randomness involved in the queue waiting times. Thus, metrics like mean or median of the reductions in total response times are non-representative and over-approximate the overall benefits due to multiple batch executions. The magnitude of gains due to multiple batch executions is of the order of a few hours. For example, for a PSA configuration, the average gain was around 20 hours while the maximum gain was around 40 hours. These gains are significant when compared with the maximum execution time limit of 48 hours for a job submission. These results show the huge benefits that can be obtained due to reduced queue waiting times without increase in the number of processors used for execution in multiple batch executions.
4.5.4 Summary of Results

The use of large number of processors available in grids with distributed sites to provide performance benefits, especially for loosely-coupled applications, is well studied and understood. We have shown in our results that grids can also provide benefits when the number of processors used for application execution is not increased from non-grid, single-site systems and for both loosely-coupled and coupled multi-component applications with periodic communications between the sites. The benefits are due to decompositions of the applications into multiple sub jobs with smaller processor requirements for executions on multiple batch systems of a grid and the corresponding reduction in queue waiting times incurred by the sub jobs. Our results show that there are large number of application and system configurations for which grid executions can provide benefits even when the distributed sites are located in different continents. While the probability of benefits due to multiple batch executions are about 50% for about 50% of the configurations in general, there is a significant fraction of configurations for which the probabilities are close to 100%. The presence of large number of short narrow jobs (SN) for CCSM leads to higher probabilities of benefits due to multiple batch executions. This is because the occupation of the processor space by SN jobs will lead to inadequate number of processors for execution of a large job in single batch execution while the remaining processor space will be sufficient for accommodating sub jobs with small processor requirements in multiple batch executions. In general, our simulation results for CCSM correspond with the real executions. We found that backfilling scheduling policies, that are commonly used in many batch systems, will lead to large probabilities of benefits for both CCSM and PSA applications. This is because the sub jobs with small processor requirements are able to back fill more in multiple batch executions than the jobs in single batch executions.

We have shown that even without an increase in the number of processors, applications can gain from execution on multiple batch grids due to lower queue waiting times corresponding to the lower processor requirements on individual sites of the batch grid. Specifically, there are large percentages of configurations with significant probabilities of benefits with multiple batch executions for both multi-component CCSM simulations and parameter sweep applications. While the CCSM simulations showed higher benefits on queues with predominantly narrow
jobs, the PSA simulations showed higher benefits on queues with predominantly wide jobs. We also found that for queues with backfilling policies, the probabilities of benefits with multiple batch executions are higher. Finally, for the CCSM multiple batch executions, the chances of benefits are higher for larger inter-site bandwidths.

## 4.6 Multiple-Batch Execution Policies for Multi Component Applications

Due to its long-running nature, the Multi-Component Application (MCA) execution completes over several submissions in a batch system. Since the results of the last submission would be needed to continue execution in the next submission, the submissions are sequentialized. Hence, at any point of time there is only a single submission of MCA and it is either being executed or is queued. Note that the execution times of each of the previous submissions would be equal to the maximum execution time allowed in the batch system.

One of the most significant challenges in multi-component executions on multiple batch systems is that queues become available at different times and components are generally required to run simultaneously. While some multiple-site scheduling systems support co-allocating such jobs, most sites do not. Hence, we consider different possibilities or policies of execution. For simplicity, we restrict ourselves to a single job request (of MCA) in each system at any point of time. In the description that follows, we use the term “active” to indicate the state of a batch system in which our job has has been assigned the requested number of processors and can immediately start executing. We use the term “inactive” to indicate the state when our job is waiting in the queue for resources.

Following are the different execution policies we consider.

1. **WaitForAll (WFA)** This is the simplest and the most natural of all strategies. We assign one component to each batch system and execute the MCA only when all the systems are active. The active systems idle at other times waiting for the inactive systems to become active. While this strategy is simple, it could keep the system resources engaged for the entire duration of a submission without performing any useful work.
2. **WaitTillThresholdandAbort (WTTA)** This is a slight modification of the previous strategy to prevent long continuous idling. Again we execute the MCA only when all queues are active. However, any partial set of active queues wait for all queues to be active only upto a threshold time-limit. The threshold time-limit is counted from the time the first of the set of active queues became active. If the time-limit is crossed and some systems are still inactive, the active submissions are canceled, the resources returned to the scheduler and new submissions are made. Like in the previous strategy, once all queues are active, the execution of MCA starts and continues upto the time when one of the queues has exceeded its maximum execution time limit in the active state. New submissions are then made to all the queues.

3. **WaitTillThresholdandExecute (WTTE)** In the previous strategy, a large threshold would mean resources uselessly engaged for long durations. However, a small threshold may not give sufficient time for all queues to become active and the MCA submissions could get canceled and resubmitted very often without being executed. In order to guarantee the MCA execution, the active systems wait like in the previous strategy for all systems to become active until the threshold. When the threshold is crossed, however, instead of canceling the submissions, we begin execution of MCA on the active systems. The execution would continue until the maximum execution time limit of one of the active queues is crossed. If in the meantime, i.e. during the MCA execution, some of the inactive queues become active they are canceled, the resources returned and new submissions are made on those systems.

4. **NoWait** Finally, we consider the strategy of execution without any idling or surrender of active resources. In this strategy, we use all available resources at all times by reconfiguring the MCA whenever there is a change in the set of active systems. This, unlike the previous strategies, requires stopping and restarting the MCA within the active duration of a submission, and its implementation could be challenging even for a malleable application like CCSM.

Execution rate of a MCA is the number of iterations or model-steps it has completed in
a wall-clock day. Under assumptions of perfect scalability, this is directly proportional to the resource availability per wall-clock-day. Since in a multi-site execution, the number of processors available for the MCA execution could vary with time, the resource availability per day should account for both the number of processors available as well as number of hours for which it was available. If in a period of D days, N1, N2, N3, ... Nn, are number of processors available for durations T1, T2, T3, ... Tn hours respectively, resource availability rate is defined as (N1*T1 + N2*T2 + N3*T3 + ... + Nn*Tn)/D processor-hours per day. In the first three MCA-execution strategies, all available resources (active durations of systems) are not completely utilized for MCA execution. Hence, we define another term, resource utilization rate, as the rate at which resources are utilized. Here, the durations correspond to the parts of the active durations where the systems were used for execution of the MCA. For the NoIdling strategy, since all available resources are fully utilized, the resource utilization rate equals the resource availability rate.
4.7 Evaluation of Various Execution Models

We have performed several simulation experiments to quantify and understand the effects of most of the factors mentioned in the previous section. Our simulation setup comprises of three major components: (i) a workload simulator, (ii) a multiple batch system simulator and (iii) the execution rate calculator as shown in Figure 4.12.

The workload simulator produces a list of jobs with submission time, processor request size and expected execution time for each job. Although such lists could be obtained from real logs of various supercomputing sites, using a workload model to generate the job sequence enables us to vary workload characteristics. We can then include their effect on the performance of a parallel application when executed across multiple batch queues. We used the workload model developed by Lublin and Feitelson[93]. This model was developed by applying rigorous statistical procedures to logs collected from real batch systems of three different locations and was shown to be the most representative model available in a general sense. Job processor requirements, runtimes and arrivals are modeled based on a two-stage uniform distribution, a hyper-Gamma distribution and a Gamma distribution, respectively. The model parameters are preset to values representative of real logs of supercomputers. For all our job traces, we specified the maximum processor requirement of the jobs as 128 processors and maximum
execution time of 2 days. We categorize a job as long (L) or short (S) based on its execution time and as narrow (N) or wide (W) based on its processor requirements. By tuning parameters that decide the ratios of number of L to S jobs and the number of W to narrow N jobs, we generated 8 different categories of job traces: L, S, W, N, SN, SW, LN and LW.

Our batch system simulator is an MPI program with number of processes equal to number of batch systems. Each process simulates the queueing policy of the system with a workload trace obtained from the workload simulator. The MCA job request is included along with the workload using one of the four MCA-execution policies described in the previous section. Three different queueing policies, namely, FCFS, CONS (conservative backfilling) and EASY (EASY backfilling) were considered: The execution rate calculator uses the execution trace output of the simulator to calculate the resource availability rates and resource utilization rates as discussed in the previous section.

An MCA with 5 components each of size 25 was considered for our experiments. For a given experiment with a set of batch systems, the components of MCA were distributed in all possible ways to all the batch systems. Different experiments were performed where the number of batch systems were varied from 2 to 5. Corresponding to each multiple batch experiment, a single batch system experiment was performed. The queue characteristics including the scheduling policy for each system were randomly chosen with uniform distribution. A total of over 6000 experiments were performed.

The multiple queues had better resource utilization values than the single queue only with the IdleUntilThreshold and NoIdling policies. And as expected the NoIdling policy performed the best.

Figure 4.13 shows the large percentage of cases in which the multiple site outperformed the single site execution for different number of queues with the NoIdling policies.

Figure 4.14 shows the average fraction of the total simulation time spent in executing the MCA on various number of active queues. Note that the fraction of time in which all queues are active is lower than the fraction of time in which some queues are executing. Hence, IdleUntilAll and IdleUntilAllWithThreshold policies, that require all systems to be active for execution, gave poor performance.
Thus, we adopted the best execution model, the NoIdle policy and performed further studies of CCSM performance on batch grids with this model.
Figure 4.14: Durations of different number of active sites involved in MCA execution
4.8 Best Execution Model and Evaluation Methodology for CCSM

In this section, we describe the application of the NoIdling execution model to CCSM. The initial experiments described in the first few sections of this chapter, dealt with application executions that completed within the execution time limits of the batch systems. In contrast, the current work deals with applications whose execution times are significantly longer than the execution time limits of the batch systems, and hence involves checkpointing, submitting, and migration of CCSM jobs across multiple batch systems. Also, the use of an execution model for multi-site simulation of CCSM in this section, enables the estimation of the actual performance benefits of multi-site executions over single-site execution, while only the probabilities of such benefits were earlier estimated in Section 4.5. Hence, the simulation architecture described in this section is significantly different from the one described in Section 4.3.

As mentioned earlier, CCSM is typically executed for long periods to simulate climate systems for multiple centuries. To support such long running simulations, CCSM contains restart facilities where the application can be made to store its execution state as restart dumps at periodic intervals or after some simulated days and simulations for an execution can be continued from the previous executions using the restart dumps of the previous executions. Batch queue systems are associated with limits for execution time for a job. If a job submitted to batch queue exceeds the execution limit of the queue, it is aborted by the system. The execution time limits are typically few days. Hence a long running application like CCSM involving multi-century simulations will not be able to complete execution within the execution time limit of a batch system. The CCSM application executing on a batch system will have to be made to create a restart dump before the execution time limit, stopped, resubmitted to the batch system and continued from the previous execution using the restart dump.

In our execution model, when CCSM is executed on $B$ batch systems or queues, $B$ job submissions are made to the queues with different processor requirements or request sizes. We refer to a job submission to a queue and the corresponding batch system as becoming active when the job completes waiting in the queue and is ready for execution. When some subset of
submissions become active, the components of CCSM are executed on the corresponding subset of active batch systems, \( B_A \subseteq B \). When a submission reaches the execution time limit of the corresponding queue, the batch system becomes unavailable for components that were executing on the system, and is hence removed from \( B_A \). In this case, the CCSM is made to create a restart dump and all the components are remapped or rescheduled to the new/updated active batch systems, \( B_A \), and continued execution from the restart dump. Also, a job submission is made to the batch system that became unavailable for execution. At some point, when this batch system becomes active, the CCSM executing on the already active systems is made to create a restart dump, the set \( B_A \) is updated to include the system that became newly active, components are remapped to new/updated \( B_A \) and continued execution from the restart dump. Specifically, CCSM is rescheduled whenever the set \( B_A \) changes, either due to some batch systems becoming active or due to one of the active batch systems becoming unavailable upon the corresponding submission reaching its execution time limit on the system. These points of execution at which the application is rescheduled are denoted as rescheduling points. This process continues until CCSM completes simulation of a certain number of simulated years. The process is illustrated in Figure 4.15 with an example of a two-component application executing on two batch sites.

Thus, in our execution model, we attempt to effectively utilize the given set of batch systems by repeated submissions to the batch systems. Unlike the existing work[18], we do not require all the batch systems to be coallocated for execution. Instead, the batch systems and processors for CCSM execution are shrunk and expanded as they become unavailable and available, respectively. While the existing efforts on rescheduling primarily reschedule applications when individual processors become available or unavailable, ours is the first work that considers rescheduling applications when an entire batch system become active or inactive due to submitted jobs completing the queue waiting times and submissions reaching the execution time limits on the batch systems, respectively. This involves considering new challenges related to job submission and queue dynamics on the different batch systems. In our execution model, a component of CCSM is always executed within a batch system and not executed across batch systems, because the intra-component communications are about ten times higher and three times more frequent than inter-component communications. Hence, executing a component
Figure 4.15: Execution of a long-running two-component application on two sites
across batch systems will lead to poor execution times due to large amount of communications across slow links connecting the batch systems. Thus the maximum number of batch systems that can be utilized for CCSM execution is equal to the number of components. In our work, we use only one processor for executing coupler since parallelization of coupler does not significantly improve performance. Hence, we consider a maximum of four batch systems for executing the non-coupler components.

We evaluate and compare the single-site and multi-site executions of CCSM in terms of the queue waiting times of the submissions, the resource availability rate (RAR) and application throughput. RAR gives a measure of the amount of processor-hours the resources were available for CCSM execution. Processor-hours for a submission is defined as the product of the number of processors used for the execution and the execution time corresponding to the submission. For a long-running execution consisting of multiple submissions, RAR is calculated for a system as the ratio of the sum of processor-hours for the submissions to the total execution time considered for evaluation. For multi-site executions, RAR is calculated as the ratio of the sum of the processor-hours between the rescheduling points and the total execution time considered for evaluation. We also compare the single and multi-site executions using application throughput. For CCSM, throughput is calculated as the number of climate days simulated per wall-clock day. For comparison of single and multiple system executions, we generated a set of four batch systems with different number of processors and considered the largest of the systems for single system executions. This is to enable fair comparison since single-system users would typically want to use the largest batch system available to them for better performance, larger problem sizes and smaller queue waiting times. Using any other batch system for single system executions will give an unfair advantage to multi system executions since users will typically incur higher queue waiting times for the same processor requirements.

The formal description of the execution model and the calculation of RAR and queue waiting times are shown in Figure 4.16. The getconfig function in line 22 obtains a new configuration or distribution of CCSM components based on the previous CCSM configuration and the number of active sites, $B_A$. The scheduling and rescheduling algorithm used in the getconfig function is described in a later section, Section 4.10.2.
Algorithm: Execution Model

1. $B = \{1, 2, ..., N_{sites}\}$: set of all participant sites; $B_A \subseteq B$: set of active sites
2. $R(i), i \in B$: request size used for site $i$; $S(i), i \in B$: number of submissions on site $i$
3. $N = \sum_{i \in B_A} R(i)$: total number of available processors
4. $(E, i) \in \{Active, Inactive\} \times B$: scheduling event of site $i$ becoming active/inactive
5. $C, C_{prev}$: current and previous CCSM configurations
6. $QT(i), QT_{avg}(i), i \in B$: total and average queue wait time for CCSM submissions in site $i$;
   $RAR$: Resource Availability Rate
7. $t_{prev}, t$: times at current and previous scheduling events
8. $t_q(i)$: time of CCSM job submission on batch system $i$
9. $T$: total wall-clock time of an experiment
10. $C_{prev}, C, B_A \leftarrow \emptyset$; $N = 0$; $S \leftarrow 0$; $t_{prev} = \text{current time}$
11. while not end of simulation do
12.     Wait until next event, $(E, j)$
13.     Stop CCSM run on $B_A$
14.     $C_{prev} \leftarrow C$; $t = \text{current time}$; $RAR = RAR + N \times (t - t_{prev})$; $t_{prev} = t$
15.     if $E$ is Active then
16.         $B_A \leftarrow B_A \cup \{j\}$; $N = N + R(j)$; $QT(j) = QT(j) + (t - t_q(j))$
17.     else
18.         /* $E$ is Inactive */
19.         $B_A \leftarrow B_A - \{j\}$; $N = N - R(j)$; $S(j) = S(j) + 1$
20.         Resubmit job on Site $j$; $t_q(j) = t$
21.     end
22.     $C \leftarrow \text{getconfig}(C_{prev}, B_A); \text{transfer\_restart\_files}(C_{prev}, C)$
23.     Restart CCSM run on $B_A$ with the configuration $C$
24. end
25. $T = \text{current time}$; $RAR = RAR/T$
26. foreach $i \in B$ do
27.     $QT_{avg}(i) = QT(i)/S(i)$
28. end

Figure 4.16: Formal Description of the Multi-site Execution Model
4.9 Analytical Model for RAR

In this section, we derive an analytical model for RAR and motivate the need for detailed simulation studies. The terms, $T$, $R(j)$, $S(j)$, and $B$, used in this section, are defined in Figure 4.16. RAR can be defined as:

$$RAR = \frac{\sum_{i=1}^{N_{Events}} (\sum_{j \in B_A(i)} t_i \times R(j))}{T}$$  \hspace{1cm} (4.3)

In Equation 4.3, $N_{Events}$ is the total number of scheduling events in the simulation period, $T$, where a scheduling event corresponds to a change in the set of active batch systems available for application execution. $t_i$ is the duration between two scheduling events. $B_A(i)$ is the set of active batch system corresponding to event $i$. $R(j)$ is the request size on a batch system $j$.

Equation 4.3 represents RAR in terms of number of scheduling events. RAR can be equivalently expressed in terms of number of submissions in each batch system as:

$$RAR = \frac{\sum_{j \in B} S(j) \times R(j) \times t_{MAX}}{T}$$  \hspace{1cm} (4.4)

where $t_{MAX}$ is the maximum execution time for a job, $S(j)$ is the number of CCSM job submissions in time $T$ on batch system $j$ and $B$ is the set of batch systems.

Since in our execution model, the CCSM job is either queued or active on each system, the total time, $T$, is the sum of the total queue waiting time and the total execution time of CCSM jobs on each site. Hence, the total simulation time, $T = S(j)(QT_{avg}(j) + t_{MAX})$, for each site $j$. Substituting the expression for $S(j)$ in terms of $T$ in Equation 4.4:

$$RAR = \frac{\sum_{j \in B} \frac{R(j)t_{MAX}T}{t_{MAX} + QT_{avg}(j)}}{T} = \sum_{j \in B} \frac{R(j)t_{MAX}}{t_{MAX} + QT_{avg}(j)}$$  \hspace{1cm} (4.5)

To compare single and multi-site executions, let $R_{\text{single}}$ and $QT_{\text{avg-single}}$ be the request size and total queue waiting time, respectively, for single-site executions, and $R_{\text{multi}}(i)$ and $QT_{\text{avg-multi}}(i)$ be the corresponding values on site $i$ for multi-site executions. We make two assumptions for analyzing the improvements due to multi-site executions. The first assumption, as represented in Equation 4.6, is that the total request sizes for single and multi-site executions are equal.
\[ R_{\text{single}} = \sum_{i \in B} R_{\text{multi}}(i) \quad (4.6) \]

The second assumption is that the total queue waiting time on a system is directly proportional to the request size on the system and hence the total queue waiting time for single site executions is greater than the total queue waiting time for every site \( i \) for multi-site executions. This is represented in Equation 4.7.

\[ QT_{\text{avg\_single}} \geq QT_{\text{avg\_multi}}(i), \forall i \in B \quad (4.7) \]

Then:

\[
RAR_{\text{multi}} = \sum_{j \in B} \frac{R_{\text{multi}}(j) t_{\text{MAX}}}{t_{\text{MAX}} + QT_{\text{avg\_multi}}(j)} \quad (\text{using Equation 4.5})
\]

\[
\geq \sum_{j \in B} \frac{R_{\text{multi}}(j) t_{\text{MAX}}}{t_{\text{MAX}} + QT_{\text{avg\_single}}} \quad (\text{by Equation 4.7})
\]

\[
\geq \frac{R_{\text{single}} t_{\text{MAX}}}{t_{\text{MAX}} + QT_{\text{avg\_single}}} \quad (\text{by Equation 4.6})
\]

\[ \geq RAR_{\text{single}} \]

Thus RAR for multi-site executions is at least equal to RAR for single-site executions if the assumptions given by Equations 4.6 and 4.7 are true. However these assumptions may not be applicable in many cases for real multi-component applications and batch systems. Moreover, application throughput cannot be directly inferred from RAR due to the variable scalabilities of the application components. Hence, analytical models will not suffice for complete analysis and comparison of single-site and multi-site executions, and detailed simulation studies are warranted.

### 4.10 Methodology for CCSM Performance Studies

Our simulation setup for this set of experiments comprises of five major components: (i) a workload simulator, (ii) a multiple batch system simulator, (iii) statistic calculator, (iv) CCSM (re)scheduler and (v) CCSM simulator as shown in Figure 4.17.
Figure 4.17: Simulator Architecture
The workload simulator produces a list of jobs with submission time, processor request size and expected execution time for each job. We used the same workload model developed by Lublin and Feitelson[93] described in Section 4.7 and the workload types are as described in Section 4.4.

The batch system simulator uses the job traces produced by the workload simulator, and simulates the job executions on a given set of batch systems with given queue scheduling policies. Three queue scheduling policies, namely, FCFS, conservative (CONS) and EASY backfilling[55] were used for scheduling or selecting jobs in the queues for allocation to processors. The simulator also simulates submissions corresponding to CCSM application by adding the submissions to the job traces of each batch queue. A CCSM submission is added to the job trace immediately after the completion of the execution time limit for the previous CCSM execution. The simulator using the specified queuing policy for each batch queue or system, produces MCA (Multi-Component Application) execution traces containing the queue waiting times in addition to the arrival and execution times, and request sizes of the CCSM jobs. The batch simulator is written in MPI to parallelize the simulations in different batch systems.

The statistic calculator uses the MCA execution traces to calculate the application throughput and resource availability rate (RAR). As mentioned earlier, application throughput is the number of climate days simulated per wall-clock day. To calculate application throughput, the statistic calculator needs to determine the allocation of processors in the different batch systems to the different components of CCSM whenever the number of active batch systems for CCSM execution changes in the execution traces. The calculator invokes the CCSM scheduler component to determine the schedule for CCSM execution for a given number of active batch systems, namely, the request sizes for CCSM components in the different batch systems. The CCSM scheduler evaluates different candidate schedules for CCSM execution in terms of predicted execution times and chooses the schedule with the minimum predicted execution time. To determine the predicted execution time for a particular candidate schedule, the CCSM scheduler invokes the CCSM simulator which simulates the workflow and the computational and communication phases of the different components of CCSM on the processors of the different batch systems to predict the execution time. The CCSM simulator uses performance models of the
computational and communication phases of CCSM to determine the predicted execution time.

### 4.10.1 Application Simulator

We developed a discrete-event simulator for CCSM for modeling the execution flow of the components. The execution flow of the components is illustrated by Figure 4.18.

As shown in the figure, each component communicates the data processed by it to the coupler at periodic intervals. This interval of communication, *coupling period (CP)*, can be different for different components. Within each CP, a component performs some computations of its local data, receives data from the other components through the coupler, performs some computations of this received data and sends its processed data to the other components through the coupler. These phases are denoted as *send-to-receive computations (S-R), receive communication*.

Figure 4.18: CCSM Execution Flow
tions (Re), recv-to-send computations (R-S) and send communications (Se), respectively. The figure also denotes the times spent by the components, C2 and C3, waiting for communications from/to the coupler due to the waiting by the coupler for operations to complete in some other components. Some components have certain anomalous (An) phases due to certain computations performed only in some CPs. For example, the atmosphere component has an anomalous computation phase corresponding to long-wave radiation calculations at periodic intervals. The CPs and intervals between An phases are expressed in simulated hours. For our experiments, the CPs were 24 simulated hours for ocean and 1 simulated hour for other components. The interval between two anomalous phases in atmosphere component was 8 simulated hours.

The CCSM simulator takes as input, the total wall-clock time for CCSM execution, the CPs for the four components, the periodicity of anomalous phases, the number of clusters, the inter-cluster bandwidth, and the allocation of processors in the clusters for the components. The simulator then models the execution flow of CCSM until the execution pattern per simulated-day converges. The number of simulated days and the execution time for the simulation are obtained and the ratio between the two values is scaled to obtain the total number of simulated days or application throughput for the total wall-clock time available for CCSM execution.

In order to model the execution flow and predict the number of simulated days for a given execution time (simulation rate), the simulator uses models for the different phases, namely, S-R, Re, R-S, Se and An, for each component. These phase models predict the execution times for the phases for a given number of processors. To construct these phase models, we conducted many experiments by executing CCSM with T42_gx1v3 resolution, a medium resolution climate model with Eulerian dynamical core for atmosphere component with different application and system configurations. For this resolution, CCSM scales well up to about 180 processors. For the distribution of processors to the different components of CCSM, we followed the processor restrictions for the components. These restrictions include even number of processors and number of processors without prime factors except 2, 3, and 5, for the atmosphere component. The experiments were conducted across two AMD Opteron clusters, fire-16 and fire-48, with different application and system configurations. fire-16 is a 16-core cluster and consists of 8 dual-core AMD Opteron 1214 based 2.21 GHz Sun Fire servers. fire-48 is a 48-core clus-
ter and consists of 12x2 dual-core AMD Opteron 2218 based 2.64 GHz Sun Fire servers. The servers in each cluster have CentOS 4.3 and are connected by Gigabit Ethernet. For each experiment, the head nodes of the two clusters were connected to each other by one of 10 Mbps, 100 Mbps and 1 Gbps switched Ethernet. CCSM was then executed with a given distribution of the components to the processors and with a given allocation of processors to each component, and the times for the different phases were observed.

We used a simple equation, \(\text{computeTime} = a + b/\text{componentSize}\), for modeling each of the computation phases of a component. \text{componentSize} is the number of processors allocated for the component and \text{computeTime} is the execution time corresponding to the computation phase. \(a\) and \(b\) denote the model coefficients and were obtained by linear regression using the observed execution times corresponding to the actual experiments across the two clusters. The equation is based on Amdahl’s law where the first term represents the serial portions and the second term represents the parallel parts of the parallel application. For a given bandwidth of the connection between the two clusters, we did not observe large variations in the communication times. For modeling a communication phase for a given inter-cluster bandwidth, we used the average of the observed communication times for the phase corresponding to the actual experiments across the two clusters. Thus the \textit{phase models} for a given inter-cluster bandwidth can be used for predicting the simulation rate of CCSM for any number of processors allocated to the components for an inter-cluster bandwidth of 10 Mbps, 100 Mbps or 1 Gbps. Since the modeling is at the level of the phases, the resulting application simulator can also be used for predictions across more than two clusters. We verified that our simulator achieves high accuracy for a number of application and system configurations.

We validated the accuracy of the application simulator by conducting different experiments with CCSM using \textit{fire-16} and \textit{fire-48} AMD clusters. We performed three sets of 48 experiments, for 10 Mbps, 100 Mbps and 1 Gbps inter-cluster bandwidths, with each set corresponding to a specific inter-cluster bandwidth obtained by connecting the two clusters by an appropriate Ethernet switch. Each experiment in a set corresponded to execution of CCSM with a specific distribution of the CCSM components to the two clusters and a specific number of processors for each component. For a given experiment, the actual execution times of the entire application
and the individual phases were obtained. The phase execution times were used to build the phase models in the application simulator. The resulting simulator was then used to obtain the predicted execution times and percentage prediction errors (PPEs) for the experiments in the set. We obtained average PPEs of 26%, 23% and 21% for the experiments with 10 Mbps, 100 Mbps and 1 Gbps, respectively.

While earlier efforts have reported PPEs of around 5-10% [137, 89, 156], these efforts model simple parallel applications on a single cluster and have not modeled multi-component applications with the kind of complexities involved in CCSM. CCSM involves multiple complexities including different execution characteristics of different components with different computation and communication phases, and complex interaction patterns between the components including overlap of communications with computations, and idling of processors executing some components. Hence, considering the complexity of the application, and the execution platform involving multiple clusters, we consider a mean PPE of around 25% to be reasonable.

We use our performance models primarily to rank the candidate schedules for determining the best CCSM configuration and best schedule for execution of the application. We found that in most cases, our performance models resulted in good correlations between the predicted and the actual execution times for different configurations involving different component sizes and distributions, with correlation coefficients of 0.98. Hence, our performance models are highly useful for determining good ordering of different schedules in terms of execution times and obtaining an efficient schedule for application execution.

### 4.10.2 CCSM Scheduler/Rescheduler

In our execution model, the CCSM components are scheduled and rescheduled to the available batch systems at various points of execution. Scheduling involves determining the set of processors for each component to minimize the application execution time. Scheduling of CCSM even in a single batch system is non-trivial due to various factors including the heterogeneity of the components in terms of different scalabilities and also different communication requirements and patterns of the components. Scheduling CCSM on multiple batch systems should also consider the different number of processors available in the different batch systems. As mentioned
earlier, CCSM takes restart dumps of the components between rescheduling for long-running executions. Hence, when rescheduling a component of CCSM during a long-running execution, the proximity of the corresponding restart dump has to be considered to minimize the cost due to transfer of restart dumps. Thus rescheduling has to take into account the previous schedule or configuration while determining the current schedule for execution.

Scheduling for single batch executions involves determination of processor allocation or the number of processors for each component. Scheduling for multiple batch executions involves processor allocation and component mapping. Component mapping determines the mapping of the different components to the different active batch systems. For multiple batch executions, we follow the policy of using the maximum number of batch systems for CCSM executions. Initially all batch systems are considered to be active for CCSM execution. A processor allocation is valid if it adheres to the processor restrictions for the CCSM components. A component mapping is valid if the number of processors determined for each component is less than the number of processors available in the corresponding batch system on which it is allocated.

Since our primary focus is on showing the benefits with multiple batch executions without increasing the number of processors used for single batch executions, we determine for multiple batch executions, a processor allocation that is similar to the best processor allocation or schedule determined for single batch execution. Hence, we first determine the single-system best schedule for the largest batch system by evaluating all valid processor allocations and choosing the processor allocation or schedule for which the application simulator predicts the minimum execution time. To restrict the number of valid processor allocations, the scheduler chooses only power-of-two number of processors for some components. The single-system best schedule is used for execution of CCSM on a single batch system and as a base processor allocation for determining the schedule for multiple batch executions. The number of valid processor allocations evaluated for determining the single-system best schedule in this stage is approximately order of $10^5$.

Since the base processor allocation may not give the best schedule for multiple batch executions due to communications on the slow inter-cluster links and may not yield a valid component mapping, it cannot be directly used for multiple batch executions. Hence we consider valid
processor allocations equivalent to the base processor allocation by considering for each component, few valid processor allocations immediately lower and higher than the number of processors for the component in the base processor allocation. For our current work, we consider two lower and two higher valid processor allocations for each component. Hence the maximum number of processor allocations considered in this stage is $5^4$, corresponding to the four components. For each of the valid processor allocations, the scheduler evaluates all valid mappings of the four components to the four batch systems in terms of the execution times predicted by the application simulator for the mappings and chooses the schedule with the minimum predicted execution time. During rescheduling, for each of the valid mappings, the predicted cost for the transfer of the restart dumps of the components corresponding to the previous mapping for the previous execution is added to the predicted execution time for evaluation of the mapping. The maximum number of valid mappings for evaluation during initial scheduling or rescheduling is thus $4! \times 5^4 = 15000$. While the worst-case number of evaluations of valid schedules is $10^5 + 4! \times 5^4$, in practice, we found that the scheduler rejects large number of valid schedules and evaluates approximately order of $10^3$ schedules. Moreover, for rescheduling during application execution in multiple batch systems, the evaluations for determining the single-site best schedule need not be repeated. We found that our scheduler typically spends less than 5 seconds to determine the best schedule for multiple batch executions. In real settings, the scheduler will be invoked only once in several hours and hence scheduling overheads of even few minutes are tolerable.

In some cases, using the processor allocation of the single-system best schedule as a base processor allocation for multiple batch executions may not result in any valid mapping of components to the batch systems even after considering the lower and higher configurations. In these cases, the scheduler follows a worst-fit policy where the components are arranged in the descending order of the processor allocations corresponding to the single-system best schedule and the batch systems are arranged in the descending order of the number of available processors. The scheduler then tries to assign the $i^{th}$ component to the $i^{th}$ batch system. For a component that cannot be accommodated in the corresponding batch system, the scheduler decreases the number of processors allocated to the component by using the highest valid processor al-
location for the component that can be accommodated in the corresponding batch system. The resulting processor allocation of the components is then used as the base processor allocation for scheduling in multiple batch systems.

4.11 Experiments and Results for CCSM Performance Studies

4.11.1 Experiment setup

We conducted 1000 simulation experiments for single and multiple batch executions for each inter-cluster or inter-site bandwidth to analyze the gains in executing long-running CCSM applications on batch grids. For our experiments, we used three different inter-cluster bandwidth values, namely, 10 Mbps, 100 Mbps and 1 Gbps. The first two bandwidths are commonly observed on the links connecting two clusters located at two different sites in many grid systems. The last bandwidth is seen on the links connecting two batch systems in a single site and on the links connecting two different submissions in a single batch system. Simulating this scenario is important since we claim that using multiple submissions of the components of an application to the same batch queue can also lead to benefits.

For each simulation experiment for a given inter-cluster bandwidth, we randomly chose four queues corresponding to execution of four components of CCSM. For each queue, the total number of processors in the queue is randomly chosen from \{32, 64, 128, 256, 512\}. The workload characteristics, namely, the number of long (L), short (S), wide (W), and narrow (N) jobs submitted to a queue, were randomly generated for each queue. The queue scheduling policy for each queue is also randomly chosen from one of FCFS, CONS and EASY policies. We used the four queues to simulate long running CCSM execution on multiple batch systems and used the largest of the four queues to simulate execution on a single batch system. We then compared the multiple and single batch CCSM executions in terms of queue waiting times, resource availability rates (RARs) and application throughput.
4.11.2 Queue Waiting Times of CCSM Jobs

The primary reason for the potential improvement in throughput of CCSM when executed on multiple batch systems is the reduction in queue waiting times incurred by the CCSM component jobs on the individual queues. For each simulation experiment, we compared the queue waiting times of the CCSM jobs in the largest queue when used for single and multiple batch executions. The largest queue, besides being common to single and multiple batch executions, also contributes most of the processor space for CCSM application in multiple batch executions. Figure 4.19(a) shows the average queue waiting times for single and multiple batch executions for different number of processors in the largest queue. We find that the average reduction in queue waiting times due to multiple system executions is 60%. We also find that the percentage reduction in queue waiting times increases for increasing size of the largest queue. This is because with increase in size of the queue, the difference between the number of processors available in the queue and the number of processors requested for CCSM increases in multiple batch executions leading to greater decrease in queue waiting times. This is illustrated in Figure 4.19(b) that compares the percentage difference in queue waiting times between single and multiple batch systems for different ratios of request size for CCSM and total size of the largest queue in multiple batch executions. We find that for small ratios or for large gaps between request sizes and total sizes, the percentage reductions in queue waiting times due to multiple batch executions are large.

4.11.3 Resource Availability for CCSM Execution

Figure 4.20(a) shows the average resource availability rates (RARs) for single and multiple batch executions for different sizes of the largest queue. The figure shows that the average increase in RAR due to multiple batch executions is 12%. Figure 4.20(b) shows the average RARs for different request sizes of CCSM on the largest queue for multiple batch executions. The figure show that when the request sizes are less than 55 processors on the largest queue, the RAR increases with the request sizes due to more processors available for CCSM execution. When the request sizes are greater than 55, the RAR saturates since the benefits due to larger number of processors for execution are negated by the larger queue waiting times incurred for
Figure 4.19: Queue Waiting Times for CCSM Jobs
higher number of processors. This is further illustrated in Figure 4.21 which shows the RARs for different ratios of request sizes and total number of processors in the largest queue for multiple batch executions. The figure shows that RARs decrease with the increasing ratios since the queue waiting times increase when the difference between the total number of processors and request size decreases. We also find that the average RARs for single batch executions are better than those for multiple batch executions in some cases. This is because in these cases, the gains due to reduced queue waiting times in multiple batch executions are offset by the loss due to reduced total number of processors available on multiple batch systems for CCSM executions.

### 4.11.4 Application Throughput

Figure 4.22 shows the comparison of average CCSM simulation rates with single and multiple batch executions for different queue sizes or number of processors of the largest batch system and for three different inter-cluster bandwidths.

The figure shows that irrespective of the number of processors available in a single system, adding more processors from other system for execution of CCSM will yield performance benefits to CCSM. We find that the average number of simulated days with multiple batch executions for a given size of the largest queue is at least 8% greater than the average with single batch executions. These results show that CCSM, in spite of involving periodic communications between different components through a coupler, is highly suitable for execution across different batch systems on batch grids. These results also illustrate that batch grids involving batch systems with complicated queuing and workload dynamics can be used for efficient executions of multi-component applications.

Comparing the results across the three inter-cluster bandwidths, we find that the number of simulated days increases when the inter-cluster bandwidth increases from 10 Mbps to 100 Mbps. However, the application throughput saturates at 100 Mbps and does not increase when the inter-cluster bandwidth is increased to 1 Gbps since the communications of small messages exchanged between the CCSM components reach the maximum speed at 100 Mbps. In general, we find that the bandwidth of the links connecting the different batch systems do not impact the percentage increase in application throughput due to multiple batch executions. This is due to
Figure 4.20: RARs for CCSM Jobs in Single and Multiple Batch Executions
Figure 4.21: Resource Availability Rate (RAR) for CCSM Jobs in Single and Multiple Batch Executions for Different (Request Size/Total Size) Ratios of Largest Queue
Figure 4.22: CCSM Simulation Rates in Single and Multiple Batch Executions for Different Queue Sizes of Largest Queue
high ratios of computations to inter-component communications in the individual components of CCSM.

Figure 4.23 shows the percentage increase in the number of CCSM simulated days for all the experiments with single and multiple batch executions for different averages of queue sizes of multiple batch systems. We find that multiple batch executions provide 55% average improvement in throughput over single batch executions, and provide huge gains for some small queue sizes. This is because when small queues are used for single system executions, the request sizes for CCSM are almost equal to the queue sizes. This leads to large queue waiting times for CCSM on single batch executions due to the presence of other jobs in the system. However, the request sizes of CCSM are split into small request sizes when executed on multiple batch executions leading to significant decrease in queue waiting times in the largest queue. This results in high gains in throughput in multiple batch executions.

Figure 4.24 shows the percentage increase in the number of CCSM simulated days for all the experiments with single and multiple batch executions for different averages of CCSM request sizes on the multiple batch systems. The figure also shows that for very large average request sizes, the gains due to multiple batch executions are low. This is because very large request sizes lead to similar large queue waiting times on both single and multiple batch systems. This result regarding limiting request sizes to obtain high benefits due to coallocation agrees with the conclusions in the work by Bucur and Epema[18].

Figure 4.25(a) shows the impact of different processor requirements of non-CCSM workloads in the queues on the CCSM simulation rates for single and multiple batch executions with inter-cluster bandwidth of 100 Mbps. The top bars correspond to workloads with small (narrow) processor requirements on the queue used for single batch executions. This queue is also the largest queue used for multiple batch executions. The bottom bars correspond to workloads with large (wide) processor requirements on the single or largest queue. The left bars correspond to narrow workloads and the right bars correspond to wide workloads on the other three batch systems used for multiple batch executions.

The figure clearly shows that the number of simulated days for single batch executions are significantly larger for narrow workloads than for wide workloads. This is because narrow
Figure 4.23: Percentage Increase in CCSM Simulation Rates due to Multiple Batch Executions for Different Averages of Queue Sizes of Multiple Batch Systems
Figure 4.24: Percentage Increase in CCSM Simulation Rates due to Multiple Batch Executions for Different Averages of CCSM Request Sizes on Multiple Batch Systems
Figure 4.25: CCSM Simulation Rates for different widths and lengths of external workloads for 100 Mbps inter-cluster bandwidth
workloads leave adequate space for execution of CCSM jobs leading to small queue waiting times for the CCSM jobs. When executing in the presence of wide workloads, CCSM jobs will have to wait for the completion of the wide jobs before processors become available for execution. This leads to larger queue waiting times for CCSM jobs. In all cases, we find that multiple batch executions give significant improvement in throughput over single batch executions. The best throughput is obtained for multiple batch executions when all the queues execute narrow workloads. The simulation rates are small when three of the queues contain wide workloads.

Figure 4.25(b) shows the impact of different execution times of non-CCSM workloads in the queues on the CCSM simulation rates for single and multiple batch executions with inter-cluster bandwidth of 100 Mbps. The top bars correspond to workloads with small (short) execution times on the queue used for single batch executions, which is also the largest queue used for multiple batch executions. The bottom bars correspond to workloads with large (long) execution times on the single or largest queue. The left bars correspond to short workloads and the right bars correspond to long workloads on the other three batch systems used for multiple batch executions. The figure shows that in general, the presence of short jobs improve the gains in throughput obtained on multiple batch executions. This is because short jobs vacate the systems quicker than long jobs and lead to small queue waiting times for CCSM jobs.

Figure 4.26 shows the impact of backfilling in the queues on the CCSM simulation rates for single and multiple batch executions with inter-cluster bandwidth of 100 Mbps. We find that in general, the type of backfilling does not impact the amount of gains in throughput due to multiple batch executions. The CCSM throughput on multiple batch executions is largely dependent on the execution rate of CCSM jobs on the largest queue. Our investigations showed that CCSM jobs had large request sizes on the largest queue and hence could not be backfilled to available processors after termination of non-CCSM jobs. On the other queues, the reduction in queue waiting times of CCSM jobs due to backfilling is negligible since backfilling also reduced the queue waiting times of non-CCSM jobs.
Figure 4.26: CCSM Simulation rates for different backfilling policies (100 Mbps)
4.12 Summary

In this chapter, we first did some simulation studies to calculate the probabilities of multi-site execution outperforming single-site execution due to reduction in queue wait times. These were conducted for parameter sweep applications and multi-component applications with various application and system characteristics. These studies indicated that there are good chances of benefits with multi-site executions. For example, for 100 Mbps bandwidth, 50% of the total configurations were found to have about 58% probability of obtaining benefits when executed across two batch queues.

We then evaluated various execution policies for execution of a long-running multi-component application across multiple independently administered queues. Of the policies considered, we found that the NoWait strategy, which used all available resources at any instant of time, performed the best. Hence, with a multiple batch system simulator that modeled this strategy, we performed more detailed simulation studies to estimate the performance benefit percentages for multi-site execution of CCSM. We again performed an extensive set of simulations with various parameters. These studies indicated at least 8% improvement in average throughput in all cases, with as high as 55% improvement in some cases.

Thus, we have studied the various aspects of execution of CCSM on multiple batch systems, and our simulation studies indicate good performance improvement with multi-site executions over single-site execution in several cases. Hence, in the next chapter, we consider and address with our middleware, the challenges involved in real implementation of such multi-site executions.
Chapter 5

Morco: Grid Middleware Framework

In this chapter, we describe a generic middleware framework we have constructed for executing long-running multi-component applications spanning multiple submissions of the components to the batch queues of multiple batch systems. In our work, the number of batch queues available for execution can change during the execution of the application. Our framework coordinates the distribution, execution, migration and restart of the components of the application on the multiple queues, where the component jobs of the different queues can have different queue waiting and startup times. To our knowledge, ours is the first work that supports execution of a long-running multi-component application across multiple batch systems with coordination and migration of the components between the systems based on batch queue dynamics.

We have also developed a genetic algorithm for mapping the components of the application to the processors of the different batch systems and fault-tolerance features in our framework to sustain executions in the presence of failures. Our framework is highly applicable for applications like climate system models that typically execute for long durations.

5.1 Introduction

Computational grids have been used over the years for efficient execution of different kinds of parallel applications, including loosely-coupled, workflow-based and in some cases tightly-coupled applications[80, 75, 9, 138]. Various robust grid middleware frameworks have been de-
developed for supporting grid executions of these kinds of applications[122, 128, 84, 119]. However, grids have not been sufficiently employed for executions of long-running multi-component applications (MCAs).

Multi-component MPMD applications[29, 38, 36] constitute an important class of parallel applications and have gained prominence in many significant scientific applications. These applications consist of components which are parallel applications themselves. The components are loosely synchronized and communications between components are lighter and less periodic than within components. These applications have become prevalent since multi-disciplinary multi-component models are used to accurately model interacting physical processes or phenomena in the areas of climate, space weather, solid rockets, fluid-structure interaction, heart disease and cancer studies[36, 86, 42]. Integrating such applications in grids is highly essential for large-scale deployment and use of scientific applications on grid systems by the scientific community.

For example, anthropogenic climate change is a matter of growing concern. Assessment of climate change requires long simulations with multi-component climate system models running into hundreds of years. Such long simulations are required as the ocean component of the climate has a long response time. Additionally assessment needs to be done far into the future. Such simulations would require computational time running of the order of weeks (if not months). Hence it is essential that such long running simulations be supported with an appropriate middleware framework. Integrating such applications in grids is also important for large-scale deployment and use of scientific applications on grid systems by the scientific community. Currently we are not aware of the availability of a grid middleware framework that supports such long-running multi-component MPMD applications.

Many current large-scale grid frameworks[142, 145] consist of multiple distributed sites with each site having one or more batch systems with corresponding batch scheduling and queuing policies. Executing long-running applications on such grids with multiple powerful batch systems is highly essential for providing high performance for the applications. We consider execution of long running multi-component applications (MCAs) on grids with multiple batch systems (batch grids, for brevity) to provide good application throughput. We define ap-
application throughput as the amount of work performed by the application in a certain amount of wall clock time.

Execution of the components of a long-running MCA on different queues of a batch grid is challenging due to different factors. One practical challenge is to coordinate the executions of the different components in different batch queues. Parallel batch systems employ queues in which the incoming parallel applications are queued before allocation by a batch scheduler to a set of available processors for execution. Thus the component jobs of an MCA incur queue waiting times in the different queues before execution on a set of processors. Different components can incur different queue waiting times on the different systems depending on the characteristics of the batch queues, namely, the processor requirements of the jobs, the total number of processors in the systems, the loads on the queues, and the queuing and scheduling policies. Thus, the different components can start executions at different times on different queues and will have to be synchronized. Another challenge is related to the execution time durations for the jobs in the batch queues. Batch systems or queues are associated with maximum execution time limits for the jobs. Jobs exceeding the execution time limit in a batch queue are terminated by the system. Components jobs of a long running MCA (multi-component application) executing on multiple batch queues will have to be checkpointed before reaching the execution time limits on the respective batch queues, resubmitted and continued from the previous executions on possibly a different set of batch systems. The set of active batch systems available for simultaneous execution of the components of a long-running MCA can vary at different points of execution due to the different startup times of the components and due to execution times of some component jobs reaching the execution time limits of the systems on which they are executing.

We have developed Morco (Middleware framework for long-running multi-component applications), a framework for execution of long-running multi-component applications (MCAs) on multiple batch systems of a batch grid. The applications span execution time limits of the batch queues and hence span multiple submissions of the individual components to the queues of the systems. Our framework automatically coordinates the execution of the components on the different queues, determines the allocation of processors in the batch systems for the compo-
nents, schedules and reschedules the components on the queues as the number of active systems change, checkpoints, transfers checkpoint data, migrates the application to a different set of active systems and continues execution. We have employed a novel execution model in which the set of active batch systems available for execution is dynamically shrunk and expanded, and the components are rescheduled on possibly different batch systems. We have also developed a genetic algorithm for determining the schedule for execution of the components on a set of active batch systems.

We have used our framework for execution of a foremost long-running multi-component parallel application, CCSM (Community Climate System Model)[29, 38]. We performed execution of CCSM for 6.5 days on a grid consisting of three parallel systems and four batch queues and show that multi-site executions can provide good application throughput with climate simulations.

To our knowledge, this work is the first effort in building a framework for executing long running multi-component applications on multiple batch systems of a grid in which the execution times of the components are significantly greater than the execution time limits associated with the batch systems. Our framework also considers batch systems with existing and possibly different job execution policies. Ours is also the first work that supports execution of a long-running multi-component application across multiple batch systems with coordination and migration of the components between the systems based on batch queue dynamics. Thus, the set of batch systems used for execution change during execution. While earlier efforts have considered co-allocation of multiple batch systems for parallel application execution, they have either used reservations ensuring simultaneous availability of all batch systems for execution[112, 28] or have assumed fixed job execution policies (e.g., FCFS) on all systems[18, 16].

Section 5.2 describes in detail the various components of our grid framework and the component interactions. In Section 5.3, we explain the genetic algorithm used in our framework for scheduling the components to the batch systems. In Section 5.4, we discuss the fault tolerant mechanisms built in our framework. Section 5.5 presents our experiment setup and various results related to long executions of CCSM on multiple batch systems and Section 5.6 discusses applicability of our framework to generic multi-component applications. Section 5.7 summa-
rizes our work and lists our future plans.

5.2 Morco - Grid Middleware Framework

Our middleware framework uses the execution model described in Section 4.8 of Chapter 4, for execution of long running MCAs on multiple batch grids using multiple submissions.

Our Morco middleware framework consists of three primary components to synchronize the executions of the components of a multi-component application (MCA) on multiple batch systems: a coordinator that determines mapping of components, and schedules and reschedules the component executions on the systems, a job monitor on each front end node of the batch systems that interfaces with the coordinator, and a job submitter on the front end node that repeatedly submits a MCA job upon completion of the previous MCA job. Our framework also consists of a MCA job script which executes and re-executes the MPI multi-component application on a system corresponding to specified mappings of components to processors at various points of time within a MCA job submitted by the job submitter. In addition to these components, our framework also includes some components to ensure fault-tolerance. The architecture is illustrated in Figure 5.2. The following subsections describe the various components, their primary functions, the interactions between the components and the salient features of Morco.

5.2.1 Coordinator

The coordinator daemon is the most significant daemon and is executed on a location that is accessible from the front-end nodes of all the systems. Typically, the coordinator is executed on one of the front-end nodes. The coordinator contains a record of all the information pertaining to the queues and the MCA jobs. This includes the number, and sizes of the queues, the job sizes for the MCA’s components, the IP addresses of the front-end node on each site, the locations of executables and restart files on each site, the set of active queues, the times at which the queues became active, the state of the MCA job on each queue, and the previous and current MCA configurations (component-queue mapping). While some of the information like the number of queues and components are static and given as input when the coordinator is started, other
information like the state and configuration of the MCA jobs are dynamically updated. For example, whenever the set of active sites changes, the coordinator calculates a new configuration based on the location of the components in the previous configuration and updates its data-structures containing the configuration information.

Since the coordinator has knowledge of the state of the entire system, it can take actions and/or instruct other daemons to take actions. Some of the actions taken by the coordinator include determining the mapping of components to batch systems, scheduling and rescheduling component executions and transferring restart files.

The communications between the coordinator and the job monitors are via socket calls. The coordinator, which acts as the server, initializes its data-structures with various queue and application related information from an input file supplied by the user. This information includes number and size of queues, restart backup and front-end locations (IP address, directory), processor speeds, network bandwidth, paths and names of component executables and restart files. It then listens for messages from the job monitors in an infinite loop. Upon receiving a mes-
sage, the message is processed, interpreted and suitable action is taken. The possible messages are \textit{START}, \textit{STOP} and \textit{STOPPED} corresponding to a queue becoming active, a queue about to become inactive and clean stop of current configuration, respectively. Upon receiving one of the first two messages, the coordinator sends a stop directive to the active application execution on all sites. Until the execution is stopped, restart files are dumped and a \textit{STOPPED} message is received from every active site, the coordinator continues to listen for messages. Any additional \textit{START} and \textit{STOP} messages received in this period are recorded without any action. After all \textit{STOPPED} messages are received, the coordinator updates its data structures with the new states of the queues and calculates the schedule (mapping of components to batch systems) as detailed in Section 5.3. The coordinator also performs the transfer of the restart files (using scp) to the new site locations and a backup location for fault tolerance. The schedule information is then transferred back through the already established socket connections to the clients on the job monitors. As the other components of the framework take over the continuation of the execution from this point, the coordinator continues listening for messages. A detailed description of the actions taken by the coordinator are described in Section 5.2.5.

5.2.2 Job Monitor

The job monitor daemons track the local behavior of the MCA jobs on the batch systems and interfaces with the coordinator. A job monitor daemon is started for each queue used in the framework on the front-end node of the respective system. The monitor notifies the coordinator of various events including:

1) sending a \textit{START} message when the batch system becomes active due to the start of a MCA job in the system,

2) sending a \textit{STOP} message when a submitted MCA job nears the execution time limit of the queue, and

3) sending a \textit{STOPPED} message when the current execution has been stopped and the MCA job is ready for reconfiguration.

The job monitor also processes the configuration data, related to mapping of components to the processors in the queue, supplied by the coordinator at every reconfiguration event and
writes the configurations to local files for reading by the MCA jobs.

5.2.3 JobSubmitter

Job submitter is another daemon that is started for each queue and runs on the front-end node of the respective batch system. Its main functionality is to iteratively submit MCA (multi-component application) jobs to the queue through a MCA job script. The job submitter monitors the status of a submitted job, waiting for it to complete before submitting the next job through the MCA job script such that only one MCA job exists in the system, in the queued or execution state, at a given time. The job submitter submits the MCA job script to the batch queue using the local queue submission mechanisms. For example, the script is submitted using llsubmit loadlevel command on a system with IBM loadleveler execution mechanism and using qsub on a system with PBS execution mechanism.

5.2.4 MCA Job Script

The MPI application with a set of MCA’s components is executed on a set of processors in a queue by the execution of the MCA job script on the queue. Within a MCA job submission, corresponding to the execution of the MCA job script by the job submitter, the job script executes multiple MPI applications corresponding to multiple rescheduling events with the component mappings specified by the coordinator.

At a given rescheduling event, the job scripts on the active batch queues synchronize using coordinator and simultaneously launches multiple MPI applications, one for each queue, that coordinate to form a single MPI world. Existing middleware mechanisms including PACX-MPI[64] and MPICH-GX[123] can be used to coordinate multiple MPI applications executed on different batch systems to form a single MPI world. An MPI application on a single queue will execute a set of components of the multi-component application.

After submitting a MPI application with a set of MCA’s components on a set of processors, the job script waits for the MPI application to complete due to rescheduling or reconfiguration of the multi-component application by the coordinator. If the MPI application completes due to reconfiguration, the job script waits for a new configuration from the coordinator and submits
a new MPI application corresponding to the configuration. If the MPI application completes due to nearing the execution time limit, the MCA job script completes execution. The general outline of the MCA-job-script is shown in Figure 5.2.

```
1 Algorithm: MCA Job-script
2    /* Job Monitor notifies Coordinator that this job is active; MyQueue is added to set of Active queues */
3 repeat
4    /* wait for configuration information and restart files from coordinator */
5        while component-config file and transfer-complete not present wait ;
6        MyComponents = get components on MyQueue from component-config file ;
7 repeat
8            build MyComponents ;
9            unpack restart files for MyComponents ;
10           execute MyComponents on MyQueue using MPI ;
11            /* Execution stopped due to reconfiguration */
12           pack restart files for MyComponents ;
13            RunStatus = status of the last run ;
14            /* for fault tolerance */
15 until RunStatus is PASS ;
16 until exec-time-limit reached ;
```

```
Figure 5.2: Outline of submit-script.sh
```

### 5.2.5 Component Interactions

A multi-component application (MCA) job is submitted to each of the batch systems with a request for a specific number of processors by the job submitter. We refer to a job submission to a queue and the corresponding batch system as becoming active when the job completes waiting in the queue and is ready for execution. When a job on a batch system is active, it coordinates with our middleware framework and executes some components of the application depending upon the number of active batch systems at that instant. The components executed by the job can change when the number of active batch systems changes. When the job is close to its maximum execution time-limit on the batch system, it coordinates with the rest of our framework, creates the necessary restart data and exits the queue. The job submitter then
submits a new MCA job to the queue. These interactions among the framework components are described in detail below.

When a job corresponding to the MCA job script submitted by a job submitter to a system becomes active or has entered the execution state after waiting in the batch queue, the job monitor on the system informs the coordinator of the START status of the job. The batch system on which the job has entered the START state is considered to be active. The coordinator based on the number of active batch systems, uses a genetic algorithm to determine the schedule of execution of the multi-component application on the set of active batch systems. The schedule contains the set of components and the number of processors for the components allocated to each active system. The schedule is sent to the job monitors of the active systems which write the schedule to files called component-config files. The batch job corresponding to the MCA job script of an active system reads its component-config and executes a MPI application consisting of a set of components on a set of processors specified in its component-config file.

At some point, some more batch systems can become active. The coordinator, after receiving the START messages from the job monitors of these newly active systems, performs rescheduling of the multi-component application (MCA) to the current set of active systems, total-active, that includes the active systems on which the MCA has already begun execution, already-active, and the new batch systems that have become active, newly-active. The coordinator sends stop signals through files that are read periodically by the MPI applications executing on the already-active systems. The MPI applications, after receiving the stop signals, create the restart files and stop executions. The job monitors at the already-active sites then send a STOPPED message to the coordinator. The coordinator, after receiving the STOPPED messages from all the already-active sites, uses the genetic algorithm to determine the schedule of execution on the total-active batch systems. The schedule is sent to the job monitors of the active systems which write the schedule to component-config files. The coordinator also transfers the restart dump files generated by the applications in the previous set of active systems to the new schedule, and takes a backup of the restart files for use in case of a complete system failure, thereby providing fault-tolerance. It then informs the batch jobs of the active systems to resume execution by sending a transfer-complete file. The batch job of each active system, cor-
responding to MCA job script, reads its component-config and executes its set of components using MPI application on the set of its processors as specified in the component-config file. The components read the restart dump files and continue execution.

At some point, a MPI application executing on one of the active systems can near the execution time limit of the system or the system can become inactive. The job monitor of the system detects this event by using an alarm initialized to a value less than the execution time limit. The alarm is initialized at the beginning of the MPI application execution. When the application execution nears the execution time limit, the job monitor receives a signal from the alarm, and sends a STOP message to the coordinator. The coordinator then performs rescheduling of the application using the same steps as when a new system becomes active: the coordinator stops the components executing on the other active batch systems, the components create the restart dump files, the coordinator determines a new schedule on the available set of active batch systems, and transfers the restart dump files generated by the applications in the previous set of active systems to the new schedule. The batch jobs of the currently active systems executes the components specified in the new schedule using MPI applications on the systems, and the MPI applications read the restart dump files and continue execution. The job submitter in the system that has become inactive waits for the job to complete and submits a new job through a MCA job script.

5.2.6 CCSM Restarts, Packing and Synchronizing Communications

CCSM has inbuilt facility for creation of restart file dumps and continuation from the restart dumps. However, the point of stopping of execution has to be specified a priori in terms of the number of days (or years) of climate simulations. Since our framework performs dynamic stopping and restarts based on the batch queues becoming active and since this cannot be predicted, the CCSM application was modified slightly to create a restart dump dynamically upon change of a value present in a file. For generic applications, the checkpoint dumps can be created dynamically on the occurrence of an event by using a checkpoint library.

We also use packing to reduce the times taken for transfer of the restart files. CCSM creates multiple restart dump files for each component including restart pointer file, restart header file
and restart data file. Our framework packs these files into a single directory file for each component. The restart startup scripts of CCSM were modified to unpack these and transfer them to suitable directories.

CCSM by default is a single-site multi-component MPMD application. Execution of CCSM with components (source and executables) distributed across multiple sites, involves modifications to the existing script file. Also, unlike most parallel programs, CCSM executables have to be rebuilt upon change of number of processors they execute on. Hence, the necessary scripts have to be modified to build and execute only certain components on certain sites based on the configuration specified. Instead of the default single run script on a single site, we use one modified run script for each of the sites. And these scripts synchronize at various points for smooth execution.

Our framework involves multiple interactions with daemons coordinating execution across multiple batch systems that can become active independently at any instant, in any order, and with or without overlaps. This creates a potential for race conditions and deadlocks. We have systematically ensured that these do not occur. Firstly, the direction of communication is maintained among the interacting components as shown in Figure 5.1. This ensures that each component receives from a single fixed source and upon receiving, sends to a single fixed receiver. The coordinator, however, receives from multiple job monitors and hence is a potential component for deadlocks and contentions. We resolve this by considering all possible ways in which messages can be in transit simultaneously. For example, after a STOP or START from a particular site, the coordinator expects a STOPPED message from all active sites. While these STOPPED messages are in transit, there is a possibility of receiving new STOP or START messages from other sites. Thus, the set of active queues can change during the time the coordinator tries to reconfigure the application for the previous set of batch queues, resulting in invalid reconfigurations. We handle such scenarios by processing all messages until all expected STOPPED messages arrive, and then updating the set of active queues in the coordinator to compute the next configuration. The coordinator also has a record of the queue active durations and if a queue is soon expected to stop, the coordinator does not include this in the restart set. We have conducted vigorous tests with dummy multi-component applications with one-minute timeouts
resulting in very large number of events per hour to ensure the robustness of the framework communications.

5.2.7 Salient Features of Morco

Some of the salient features of our Morco framework are listed below:

1) **Long-running multi-component applications:** Our framework is built for multi-component applications that involve periodic communications between the components. Since the applications considered are long-running, it is reasonable to assume that they can be stopped and restarted either using in-built restart facilities or using an external checkpointing library. Since components of most coupled multi-component applications cannot be split, we perform distribution of the communicating components across the sites rather than splitting a component across sites.

2) **Multiple queues across multiple sites:** The framework can handle variable number of queues and clusters at different sites. It can handle multiple queues within the same cluster as well as queues across clusters. Its primary function is to coordinate execution of a single large long-running multi-component application through jobs submitted multiple times to multiple queues.

3) **Dynamic reconfiguration of application:** Our framework supports multiple reconfigurations of application within a submission corresponding to other submissions becoming active or inactive. It automatically detects the job submitted to queue becoming active and reconfigures the multi-component application to include the newly available resources. Similarly, it also automatically detects when a job in a queue is close to timeout and reconfigures the multi-component application restricting it to other active queues. It uses a genetic algorithm based scheduler to dynamically compute the configuration, i.e. the mapping of components to processors. This requires a performance model for predicting performance of the multi-component application across multiple sites. The genetic algorithm is described in the next section.

4) **Portability to Different Batch Systems:** At any instant there is exactly one job corresponding to our target application in each queue, and hence it does not unfairly affect the queue wait times of other external jobs in the queue. Our framework performs and monitors our submission
and acts in response to the actions of the schedulers in the batch system; thus, it can be used without any modifications to existing independently managed batch schedulers at various sites. Also, since its only interaction with the batch system is to submit a job, it can be used with a wide range of batch system schedulers.

5) **Fault-tolerance:** Since our framework supports execution of an application across an open network with potential network instabilities that can cause failures in MPI executions, we have included an adjunct fault-tolerance framework for automatically detecting such and other failures and re-running the application from the previous restart-dumps. We have also incorporated a large-scale fault-tolerance feature within the coordinator daemon of the main framework to handle major failures such as node failures resulting in failure of the framework daemons. The fault-tolerance provided in Morco is described in detail in Section 5.4.

### 5.3 Genetic Algorithm for Scheduling and Rescheduling

Scheduling of a multi-component application (MCA) on multiple batch systems involves determining the allocation of processors in the batch systems for the MCA’s components, to obtain best performance or minimum execution time for the MCA execution. Scheduling should consider the different number of processors available in the different batch systems and the characteristics of the MCA’s components. Also, when rescheduling a component of the MCA during a long-running execution, the proximity of the corresponding restart dumps has to be considered to minimize the cost due to transfer of restart dumps. Thus rescheduling has to take into account the previous schedule or configuration while determining the current schedule for execution. Determination of a schedule for multiple batch executions involves both *processor allocation* and *component mapping*. Processor allocation is determining the number of processors for each MCA component. Component mapping is determining the mapping of the different components to the different active batch systems. A component mapping is valid if the number of processors determined for each component is less than the number of processors available in the corresponding batch system on which the component is allocated.

We have developed a genetic algorithm for scheduling multiple batch executions of a multi-
component application (MCA). The sizes and locations of the active queues, and the configuration of the last execution, are inputs to the algorithm. We model each chromosome as a string of length equal to the number of components, with each value indicating the processor sizes for each component. Thus, a chromosome specifies the component sizes or processor allocation. We then evaluate a chromosome for scheduling on a given set of active systems. For evaluating a chromosome that specifies the component sizes, we explore all possible distributions of the components, with the specific sizes, on the set of active systems. Thus, we try all possible component mappings to the active queues for a given processor allocation specified by the chromosome. For each mapping, we use an application-specific fitness function to calculate a fitness value for the mapping. The maximum fitness value for all component mappings for a given chromosome is used as the fitness value of the chromosome. For example, for CCSM, we calculated the fitness value for a component mapping as the the expected number of climate days that can be simulated within the maximum execution time limit of the batch systems.

The application-specific fitness function used by the genetic algorithm can be a multi-site execution performance model function that simulates the execution of the multi-component application on a set of active batch systems. For example, for CCSM, to estimate the number of climate days, we use a multi-site execution performance model of CCSM that considers intra and inter-site bandwidths and processor speeds to model the times for initialization, component computation and communication and restart transfers. These times along with the workflow pattern of the component execution are used in an event-based simulator to compute the number of climate days that can be simulated within the maximum execution time limit. The performance model was built using several real single and multi-site experiments for various component configurations and distributions. The performance modeling of CCSM to estimate the execution rate was described in detail in Section 4.10. Various application characteristics including amount of computations in components, pattern and amount of communications between the components etc. are embedded completely in the performance model and are not known to the components of our Morco framework, namely the scheduler and coordinator. Using this design, the Morco middleware is made generic to integration of any multi-component application.

The four standard steps of any genetic algorithm are initialization, selection, mutation and
crossover. We use a population size of 200 chromosomes and initialize each chromosome with random valid component sizes. The chromosomes are evaluated based on the fitness functions and are arranged in the descending order of fitness values. We use elitism where the first half of the chromosomes in this order with high fitness values are retained for the next generation. Of these selected chromosomes, we use a normalized fitness function as the probability distribution function to select candidate pairs for single-point crossover. Each child chromosome is mutated at a random point of mutation with a probability of 0.2. The algorithm is continued until the fitness value of the fittest chromosome does not change for 20 generations, or until a maximum of 1000 generations.

5.4 Fault Tolerance

In addition to the primary components described above, our Morco framework also has a set of components for providing fault-tolerance. Similar to the primary framework, the fault-tolerance framework also consists of a fault-tolerance coordinator that coordinates with the fault-tolerance monitors running on each site.

A fault-tolerance monitor of a site monitors the MCA run on its site and in case of a failure in one of the components being executed on its site, informs the local FAILED status to the fault-tolerance coordinator. The fault-tolerance coordinator then reports the global FAILED status to the fault-tolerance monitors running on all the other active sites. The current run on all the sites is then cleaned and a new run is launched with the same MCA configuration. The fault-tolerance monitors on each site kill the failed MPI execution and clean up the temporary files and processes that were created. The state of the application at the start of the current configuration is thus restored and the job-script is notified. The job-script relaunches the MPI execution after re-unpacking the previous restart files. Upon a clean stop of the application on all sites, the fault-tolerance monitors send their local SUCCESS status to the fault-tolerance coordinator. The fault-tolerance coordinator then sends the global SUCCESS status message to the fault-tolerance monitors on all the active sites. The fault-tolerance monitors in turn notify the respective job-scripts and the job-script proceeds to replace the old local restart files with
Chapter 5. MORCO: Grid Middleware Framework

the new restart files and notifies the primary job monitor. The primary job monitor then sends the \textit{STOPPED} message to the primary coordinator as described in Section 5.2.5.

The failure-detection module within the fault-tolerance monitor can be modified to suit the application and the expected failure types. In our setup, we have included checks for non-deterministic yet frequent errors like MPI run failures due to network instabilities, NFS read/write failures due to large restart dumps and a potential initial failure of the MCA during the complex multi-component handshake phase etc.

In order to handle more major failures, including failures of an entire site, front-end nodes of the sites, our framework daemons, etc., we have also incorporated a large-scale fault-tolerance feature within the coordinator daemon of the main framework. The coordinator daemon takes a backup of all the component restart files while transferring them at each MCA reconfiguration. Once the cause of failure is determined and rectified, a fresh launch of all the daemons in the restart mode can ensure continuation of the MCA execution from the point of the last restart dump.

5.5 Experiments and Results

We have used our framework for execution of a long-running CCSM. The efficiency of the genetic algorithm, used for scheduling the multi-component application to a set of active queues, determines the performance of the multi-component application when executed on multiple batch systems. We evaluated the genetic algorithm by comparing the schedules generated by the genetic algorithm with the schedules by an exhaustive search approach that evaluates all possible schedules to determine the best schedule. Figure 5.3 shows the times taken for determining the schedules and the predicted execution times for simulating a climate day in CCSM using the schedules generated by the genetic algorithm and the exhaustive search approach. The graphs show the results for 20 different CCSM and batch system configurations arranged in the order of the sum of the queue sizes. As the figures show, with the increase in sizes of the queues, the time taken by the exhaustive search method increases by large amounts while the genetic algorithm almost takes constant time to generate the best schedule. The second graph in the fig-
Figure 5.3: Evaluation of Genetic Algorithm for Scheduling

The figure shows that the schedules generated by the genetic algorithm are competitive when compared to the optimal schedules by the exhaustive search method. The mean percentage difference in predicted execution times of the schedules generated by the exhaustive search method and the genetic algorithm is only 0.065%, with the standard deviation of 0.1237.

We then tested our Morco middleware framework by executing CCSM across four batch queues in three clusters, namely, fire-16, an AMD Opteron cluster with 8 dual-core 2.21 GHz processors, fire-48, another AMD Opteron cluster with 12x2 dual-core 2.64 GHz processors, and varun, an Intel Xeon cluster with 13 8-core 2.66 GHz processors. Four queues were configured on these systems with OpenPBS: one queue, queue-14, of size 14 on fire-16, one queue, queue-48, of size 48 on fire-48, two queues, queue-32 and queue-64, of sizes 32 and 64, respectively, on varun. The AMD clusters are located at the Supercomputer Education and Research Centre and the Intel Xeon cluster is located at the Centre for Atmospheric and Oceanic Sciences, and are connected through a campus network with a bandwidth of around 500 Kbps. The AMD clusters are connected to each other with Gigabit ethernet switches. The connections within the three clusters are using switched Gigabit Ethernet.
External loads were simulated by submitting synthetic MPI jobs to the queuing systems based on the workload model developed by Lublin and Feitelson[93]. The maximum execution time limit for all jobs on all queues was set to 12 hours. The execution time limits on the Load Sharing Facility queues of our department range from 8 hours to 256 hours in multiples of 2. Our choice of 12 hours is close to the least limit. Larger limits will reduce the number of rescheduling events occurring in a simulation period, reduce the overheads, and thus increase the benefits due to multi-site executions with our framework. The CCSM MPMD application was submitted with MPICH2 using the \texttt{--configfile} option. The coordinator was started on the front-end node on \texttt{fire-16}. A job monitor and a job submitter corresponding to each queue were started on the front-end of its cluster.

We performed a long-running experiment in which our Morco framework executed CCSM for a period of 6.5 days across the 4 queues on 3 systems during which climate of 5 years, 4 months and 26 days was simulated. As the jobs on each of the four queues became active and inactive, the CCSM runs were automatically reconfigured and restarted by our framework. The execution profile of CCSM on the various queues during this multi-site execution is shown in Figure 5.4.

The figure shows the location of execution of various CCSM components along the execution time-line as the configurations change. The figure comprises of four subplots corresponding to the four queues in our experiment, as indicated by the labels at their top right corners. The x-axis shows the experiment timeline in hours, while the y-axis has the total number of processors available in each queue. The colored regions correspond to the execution of CCSM, while the white regions correspond to processor-periods that are either unused or used by other jobs in the queue. Each color in the figure corresponds to a single component. For any given x-axis value corresponding to a given time instant, the components executing in each queue and the number of processes used by each component are indicated by the component-colors and the height of each color, respectively. For example, during the \textit{2
\textsuperscript{nd}-12
\textsuperscript{th}} hour of execution, the land component, represented by the green bar, executed on \texttt{queue-14} and the atmosphere component, represented by the dark blue bar, executed on \texttt{queue-48}. The experiment involved a total number of 35 reconfigurations of CCSM components performed automatically by our
Figure 5.4: Multi-site Execution Profile
framework.

As the variations in the heights of colored regions in the figure indicate, the component sizes and hence the total number of processes used for CCSM execution varies with the set of active queues. Also, there are noticeable periods of inactive systems in each queue indicated by the white regions. Our framework automatically handles large changes in the number of active queues and the number of processors. For example, in the 1st hour, only queue-14 was active and all the five CCSM components were executed in this queue. This is indicated at hour 1 on the x-axis by the presence of all four colors in the topmost subplot and the blank regions on other subplots. The figure also shows two phases during the experiment near the 100th and 130th hours, during which there were zero active systems, i.e., when the CCSM jobs on all the batch systems were waiting in the queue. The black vertical lines indicate points during our experiment when there were system failures due to power fluctuations. These power failures resulted in the shutdown of the entire system on one of the sites. As mentioned in Section 5.4, our fault-tolerant framework takes backups of all the component restart files at each CCSM reconfiguration. We used these restart backups to continue our experiment from these points.

The figure also shows that during reconfigurations, some components are migrated dynamically from one site to another, while some retain their locations. During the 1st hour, queue-14 is active and all the components are executed on a very small number of processors in this queue. Then, the other three queues become active and the components are executed on all the four queues in the next configuration. Atmosphere and ocean components are automatically migrated by the framework from queue-14 to queue-48 and queue-32, respectively. Ice and coupler are both migrated from queue-14 to queue-64. Land continues to execute on queue-14, but on a larger number of processors as indicated by the height of the green bar. After around 11 hours of execution on this configuration, all queues near the execution time limits on the respective batch systems. Our Morco framework automatically detects the events and resubmits the CCSM jobs in all the queues. All batch systems at this stage are inactive as indicated by the empty (white) regions in all the queues. At the 13th hour, queue-14 becomes active, a new CCSM configuration is determined by the coordinator of our framework, and all the components are migrated from the previous configuration for execution on queue-14. This
configuration is similar to the configuration at the 1\textsuperscript{st} hour. In the 24th hour, when queue-32 also becomes active, a new configuration is determined by our framework, and all the components except atmosphere migrate to queue-32. Then, as queue-14 nears the execution time limit, and queue-64 becomes active, a new configuration is determined again, the atmosphere component is migrated from queue-14 to queue-32 and all the other components executing on queue-32 are migrated to queue-64.

In this way, whenever new batch systems become active or active systems reach execution time limit, our Morco framework automatically stops the execution on the current configuration, calculates a new configuration with different component sizes and different locations, reconfigures the CCSM components to the new configuration and continues the execution. It also handles smoothly the cases of no active queues and all queues becoming active. Thus, the experiment has demonstrated that our middleware framework can be effectively used for robust long-running simulations.

Figure 5.5 represents the percentages of total execution time on different number of queues. As the figure shows, our Morco framework is able to coordinate executions between components of the multi-component application on multiple batch systems for significant percentages of time.

A similar execution profile for a single-site run is shown in Figure 5.6. As shown in the figure, there are larger gaps between two CCSM executions when CCSM is executed only on a single site than when it is executed across multiple sites using our Morco framework as shown in Figure 5.4. The total length of the gaps is 88 hours for single-site runs and only 25 hours for multi-site runs. Thus, multi-site executions of CCSM using our Morco framework ensure continuous progress and regular updates of long-running climate simulations.

Figure 5.7 compares the execution progress of CCSM on multi-site runs with that on single-site runs. Each point in the figure corresponds to a restart point in the experiments. The almost flat-regions of the multi-site execution curve between 120-140 hours is due to all batch jobs becoming inactive during this time as seen in Figure 5.4. The other flat regions (e.g. 40-60 hours) correspond to execution on small number of processors on queue-14. We find that the progress of execution with multiple sites is comparable with single-site executions, in spite of
Figure 5.5: Use of Multiple Queues for Coordinated Executions by Morco

Figure 5.6: Single-site Execution Profile
the various overheads related to multi-site executions including system failures due to power fluctuations, restart overheads, multiple reconfiguration and rebuilding overheads.

Figure 5.8 gives the times consumed by various phases of the multi-site executions including the overheads. Idle time refers to the time when the CCSM jobs were not executing and were waiting in the queues. The remaining times correspond to executions of CCSM jobs when some of the systems become active. Some fraction of this active time is consumed by the multi-site execution overheads, while the remaining time is spent for useful CCSM computations. The most significant overhead is in the startup which includes MPI initializations across CCSM components, restart file reads, and initialization of various CCSM components. There are also noticeable overheads involved in compilation and preprocessing of components, as well as in writing and packing of restart files. A small overhead is also incurred by the transfer of restart files.

The multi-site executions also use a conservative estimate of 1 hour as the time for recon-
configuration of CCSM components before the components reach the execution time limits on the systems on which they are executing, and thus not utilize the remaining 1 hour for CCSM execution. The current structure of CCSM requires building or compiling CCSM executables for every reconfiguration during multi-site executions, resulting in building overhead of about 2-8 minutes every reconfiguration and a few (2-3) hours for the entire multi-site run.

Building CCSM and creating executables for all possible processor configurations *a priori* will result in increase in useful CCSM computations, and hence significant increase in application throughput. Multi-site executions also provide the added benefits of using the available systems and not relying on a single system, as is the case with single-site executions.

This work follows a greedy strategy, in which we reschedule at every queue event and use all available queues at every point of time. However, too many rescheduling events, especially when the gain due to rescheduling to a new configuration is not high, can lead to overall performance losses due to overheads. In the next chapter, we describe rescheduling policies for selective reconfigurations, where the middleware dynamically decides at each event whether or not to reconfigure so as to minimize these performance losses.
5.6 Discussion

While we have demonstrated the Morco framework with CCSM, the framework is generic and can support any long-running multi-component MPMD application with very few modifications. The various daemons of our framework, including the coordinator, job monitor and submitter, and the job script respond to generic events including components starts, stops and restarts, and can be used without modifications for other applications. The primary application specific components are the performance model used by the genetic algorithm to evaluate the fitness of the schedules, and the restart facilities in the application to dump the restart files during application reconfigurations by the coordinator. Performance modeling is a common approach for predicting the execution times of parallel applications and many performance modeling strategies exist for executions of parallel applications on heterogeneous clusters and networks[120, 89, 156]. In the absence of the performance models, sample profiling runs of the MCAs can be used to approximate the execution times on multiple batch systems.

Most of the multi-component applications are long-running and have in-built checkpoint and restart facilities developed by the model developers in anticipation of system failures and for execution on supercomputing sites with limits on the wall-clock time per job submission. Many climate and weather-forecasting models[22, 125, 135], and long-running applications in CFD and molecular dynamics[90, 4, 47] perform application-specific checkpointing and restarting for fault tolerant simulations. These applications can be made to create checkpoints during the reconfigurations by the coordinator. For other applications, the checkpoint dumps can be created dynamically on the occurrence of an event by using a checkpoint library[148, 57].

In our current framework, we use CCSM specific inputs, namely, the names and locations of the components and restart files, the scripts for building CCSM, and macros for executions, to our framework components. However, we can trivially generalize these using the standard naming schemes as in WSRF[154] and XML.

Our framework requires execution of daemons including coordinator, job submitter and job monitor on the front-end nodes of the batch system. The coordinator daemon should be executed on one of the front-end nodes or an external node that is accessible by all the front-end nodes. Executing daemons on front-end nodes to coordinate executions across batch system
is a common approach in many grid middleware frameworks\cite{64, 123, 37, 96}. Typically, the front-end nodes of batch systems have public IPs and are accessible by front-end nodes of other batch systems.

5.7 Conclusions

In this work, we have developed Morco, a middleware framework for execution of multi-component applications on batch grids consisting of multiple batch systems. The framework was demonstrated with a foremost multi-component application, CCSM. The framework performs complicated tasks including coordination of the different startup times of the component jobs on the different batch queues and rescheduling of the component jobs based on the number of active systems for execution. By performing an experiment involving 6.5 days of CCSM execution on a batch grid with four batch queues on three systems, we have shown that our framework enables multi-site executions yielding good application throughput.

In the next chapter we describe a dynamic rescheduling algorithm to further improve this throughput. The algorithm improves upon the greedy strategy for rescheduling described in this chapter by selective performance-oriented rescheduling.
Chapter 6

Performance-Oriented Rescheduling

6.1 Introduction

We have in the previous chapter described in detail the various components of our grid framework and the component interactions. While the framework enabled the execution of long-running multi-component applications on multiple batch systems, the execution model followed was a greedy strategy. In this chapter, we describe a more optimal adaptive rescheduling policy. The greedy strategy involves rescheduling at every event, related to a batch queue becoming active or inactive, resulting in a change in the set of active queues. However, a decision to not reconfigure could yield a better performance in certain cases. For example, if after an event, we know that another event is about to occur, it might be more efficient to wait for the second event and then reconfigure to the best configuration instead of reconfiguring at both the events and incurring higher reconfiguration overheads. In this chapter, we describe a one-step lookahead rescheduling policy which considers various such cases to decide whether to reschedule upon the occurrence of an event. The algorithm considers various dynamic parameters including the availabilities of the batch systems, and execution rates to decide whether and where to reschedule when the available set of active batch systems changes. The rescheduling decisions involve the use of a genetic algorithm for component mapping, performance models for performance estimation of different schedules, and dynamic predictions of batch queue dynamics.

In section 6.2, we differentiate the execution model and framework used for execution with
6.2 Modifications in Execution Model and Framework

The execution model is similar to the model described in Section 4.8. However, there are certain differences involved due to the adaptive rescheduling we are using in this chapter. Specifically, in the earlier execution model all active queues were used in execution at any instant of time. However, in this case, only a subset of active queues may be used for execution. This is illustrated in Figure 6.1.

The framework components and their interactions are also similar to the descriptions in Chapter 5 with some modifications pertaining to the adaptive rescheduling and the changes in the execution model. Like earlier, the job script is submitted to each of the batch systems by the job submitter, and when a job is active, its job monitor sends \textit{START} to coordinator. Here, the coordinator decides whether to reschedule; if the decision is to reschedule, it stops existing run (sends \textit{TRIGGER} to ‘active, but not running’ job monitors), waits for \textit{STOPPED} from all
active job monitors, determines new schedule, performs the restart file transfers and sends the new configuration information to all active job monitors. The job monitors, upon receiving this information, communicate with the job script to launch the next MPI execution. When a job on the batch system is close to time out, its job monitor sends STOP to the coordinator. The subsequent steps are the same as above, except that if this leaving system is an ‘active, but not running’ system, a reconfiguration is not needed. If it is an ‘active and running’ system, a reconfiguration is mandatory with the new configuration not involving this system. These are explained in greater detail in subsequent sections of this chapter.

6.3 Adaptive Rescheduling Algorithm

As described in the previous chapter, the number of active batch systems available for execution of MCA jobs can change during execution. At a given point of execution when the available set of active systems changes, the coordinator has to decide whether to continue the MCA with the current set of active systems used for execution or to reschedule to a new set. If the coordinator decides to reschedule, it has to determine the best schedule or set of active systems for execution.

In order to make these rescheduling decisions, namely, to decide whether or not to reschedule and to determine the best schedule or set of active systems for execution, the coordinator has to compare different candidate schedules. The most important parameter for comparison between two sets of active systems is the execution rates of the application on the two sets. The execution rate denotes the rate of simulations of a long running MCA. For example, for climate simulations, execution rate denotes the number of days that can be simulated in a certain amount of wall clock time. For a total of M available active systems or queues, we construct a lookup table with \( 2^M - 1 \) entries corresponding to all possible subsets of queues (except the null set). For each subset, the table contains the best configuration and the corresponding execution rate. A configuration for a subset of queues specifies the mapping of the MCA components to the queues in the subsets, and the processor allocation for each component. Thus, at a given point in execution, when the set of available active systems changes, the entries in the lookup
table corresponding to subsets of the current set are compared in terms of execution rates. We use a combination of the genetic algorithm described in Section 5.3 and real profiling runs to construct the lookup table. We first use the genetic algorithm that uses application performance models to determine the best estimated configuration and resource allocation for each subset of active systems. For each of these configurations, we perform real application profiling runs and obtain the actual execution rates. The actual execution rates of the configurations are stored in the lookup table and used for comparisons between different configurations. Thus, while the estimated execution rates by the performance models are used in the genetic algorithm to obtain relative rankings between the configurations, the actual execution rates obtained by the profiling runs are used in our rescheduling decisions leading to overall accuracy of the rescheduling policies. The profiling experiments were MCA runs executed for short durations (for CCSM, 4 days of climate simulations) on each configuration with the same experimental setup as for the longer runs. The 4-day CCSM simulation was timestamped and execution times were measured to estimate the execution rate, in terms of number of climate days per wall-clock day, as well as the initialization and restart overheads. Note that we had constructed and used a lookup table as above to improve upon the results of the application performance model by performing a reasonable number of real experiments. For small number of queues and processor configurations, the size of the lookup table and the resulting time to construct the lookup table are manageable. For larger number of queues, it becomes imperative to use the results of the application performance model directly.

In addition to the execution rates, the coordinator has to also consider other factors or parameters for comparison of configurations. A configuration with the best execution rate may contain a queue that can become inactive soon, i.e. the MCA job on the queue can near its execution time limit. Rescheduling to this configuration will lead to a situation where the application will have to be rescheduled again after very little progress in execution. Rescheduling an executing application involves overheads related to transfer of restart files and startup costs. Frequent rescheduling can result in high rescheduling overheads and cause overall loss in application performance. The coordinator also has to consider the times to next events on the the configurations for its rescheduling decisions. The next event on a configuration of active queues
can either be one of the queues in the configuration becoming inactive or an inactive queue outside the configuration becoming inactive. If the next event on the current configuration of active queues used for execution is one of the queues becoming inactive, the coordinator has to reschedule the application to a new set of active queues. If the next event is a queue becoming active, the coordinator can decide either to continue execution on the current configuration or migrate to a new configuration. Similarly, while comparing configurations to choose the best configuration, if two configurations have the same execution rate, the one with the longer time to next event is chosen. This is because by maximizing the time to next event, the number of rescheduling events are minimized.

We have developed a rescheduling algorithm that considers all these different factors including execution rates, rescheduling overheads, and times to next events for making a rescheduling decision. The objective of the algorithm is two-fold, (a) to minimize number of rescheduling events to minimize overheads, and (b) to use the best possible configuration across the set of available queues. Our rescheduling algorithm is based on a single-step look-ahead strategy, i.e., our current decision is based on selecting the option with best execution progress until the next reconfiguration event.

The algorithm is invoked by the coordinator when the set of active batch systems changes. The input to the algorithm is the set of active systems, \( S \). The algorithm first obtains a list of best configurations on all subsets of \( S \) and sorts these configurations in the decreasing order of execution rates. The algorithm then initializes a base configuration, \( b \). The base configuration is initialized to the current configuration of active systems used for execution if rescheduling is not mandatory. Rescheduling is mandatory when one of the active systems in the current configuration becomes inactive due to the MCA job reaching the execution time limit on the queue. In such cases, the base configuration is initialized to the configuration on top of the sorted list. Next, the algorithm compares each configuration, \( c \), in the list with the current base configuration, \( b \), and update the base configuration, \( b \), to \( c \) if \( c \) is evaluated to be better. After all configurations in the list have been thus evaluated, the base configuration gives the best configuration. If this configuration is the same as the current configuration used for execution, the coordinator decides to not reschedule. Else, the coordinator decides to reschedule the executing
application to the best configuration. The base configuration in our algorithm is compared with not only the top configuration in the lookup table with the best execution rate, but with the set of good configurations in the table, since a configuration with a lower execution rate can be a better choice if its time-to-next-event is higher, as illustrated later with sample cases (Figures 6.10 and 6.17). The pseudo code of the algorithm is given in Figure 6.2.

Algorithm: GetBestSchedule

1. S = Get_Subset_Configurations(); /* configuration with highest execution rate for each subset */
2. S = Sort_decreasing(S); /* Decreasing sort of execution rate */
3. if rescheduling mandatory then
   4. best_config = b = S(0);
   5. Remove S(0) from S;
   6. Base_rescheduling = TRUE;
4. else
5. best_config = b = existing configuration;
6. Base_no_rescheduling = TRUE;
7. end
8. foreach schedule c in S do
9.    tb = next event on base configuration (b);
10.   tc = next event on schedule (c);
11.   tcb = time to make same progress in c as by b in tb;
12.   tbc = time to make same progress in b as by c in tc;
13.   if Base_no_rescheduling then
14.      T = set_crossover_point;
15.      best_conf = compBaseNoRes(b, c, tb, tc, tcb, tbc, T, OH);
16.      if best_conf == c then
17.         b=c; Base_rescheduling= TRUE;
18.      end
19.   end
20.   best_conf = compBaseRes(b, c, tb, tc, tcb, tbc, OH);
21.   if best_conf == c then
22.      b=c;
23.   end
24. end
25. return best_conf;

Figure 6.2: Rescheduling decision algorithm

An important step in the algorithm is the comparison of a base configuration, b, with a new
**Algorithm:** compBaseNoRes()

```
input : b,c,tb,tc,tcb,tbc,T,OH
output: best_config
1 case T < tb < tc: best_config = c;
2 case tb < T < tc: best_config = c;
3 case tb =< tc =< T
4     if tcb < tb + OH and tbc < tc then
5         best_config = c
6     else
7         best_config = b
8     end
9 end
10 case T < tc =< tb
11     if tbc < tc + OH and tbc < tb then
12         best_config = b
13     else
14         best_config = c
15     end
16 end
17 case tc < T < tb: best_config = b;
18 case tc < tb < T: best_config = b;
19 if best_config == c then
20     b=c
21 end
22 return best_config;
```

Figure 6.3: compBaseNoRes()
configuration, $c$. This comparison is one of two kinds: (1) when the base configuration is the current configuration used for execution, i.e. when the base configuration does not involve rescheduling from the configuration, and (2) when the base configuration is not the current configuration, i.e. when the base configuration involves rescheduling from the current configuration. These two kinds of comparisons are made by invoking the functions, $\text{compBaseNoRes}$ and $\text{compBaseRes}$, in Figure 6.2.

If the base configuration, $b$, is the current configuration, and its execution rate is higher than the another configuration $c$, the coordinator continues executing the application with the current configuration. This is because even if the base configuration were soon to timeout, a switch to configuration $c$ at the point of timeout would yield a better throughput than a switch earlier, since $b$ with a better execution rate is allowed to execute for a longer duration. In other cases, following are the different parameters used for the comparison of $b$ and $c$ configurations:

1. $T$ is the time to crossover point beyond which execution on the new configuration $c$ will result in higher execution or simulation progress than execution on the base configuration. It can be observed that this crossover point exists only when the base configuration is the current configuration of execution, i.e. does not involve rescheduling.

2. $tb$ and $tc$ are the times to next events on $b$ and $c$ configurations respectively. The next event can be one the active queues in the configurations becoming inactive or an inactive queue out-

```plaintext
Algorithm: compBaseRes()

input : b, c, tb, tc, tcb, tcbOH
output: best_config

/* rate of b > rate of c */
if $tb < tc$ and $tcb < tb + OH$ and $tcb < tc$ then
  best_config = c
else
  best_config = b
end
if best_config == c then
  b = c
end
return best_config;
```

Figure 6.4: compBaseRes()
side the configurations becoming active.

3. \( tcb \) is the time that will be taken by the application when executed on configuration \( c \) to achieve the same simulation or execution progress that can be achieved if executed on configuration \( b \) in time \( tb \), i.e. before the next event happens on configuration \( b \). This is important for cases when the next event on \( b \) configuration happens earlier than the next event on \( c \) configuration. In such cases, the coordinator has to decide if executing application initially on \( b \) configuration and then rescheduling to \( c \) configuration after the next event happens with \( b \) at time \( tb \) will give an overall advantage to the application. Similarly, \( tbc \) is the time that will be taken by the application when executed on configuration \( b \) to achieve the same simulation or execution progress that can be achieved if executed on configuration \( c \) in time \( tc \), i.e. before the next event happens on configuration \( c \).

4. \( OH \) is the rescheduling overhead.

The parameter \( T \) is determined using the execution rates of configurations \( b \) and \( c \). \( tbc \) and \( tcb \) are determined using the execution rates and the times to next events, \( tc \) and \( tb \). Determination of \( tb \) and \( tc \) depends on the next event on the corresponding configurations. The next event on a configuration is either one of the active queues becoming inactive or one of the inactive queues becoming active. The time to the former event can be estimated by the coordinator at any point since it maintains a record of the time at which each queue last became active, and since the execution time limit for a batch queue is known a priori. An inactive queue becomes active when the MCA job waiting in the queue starts execution. Thus the time to this event is determined by estimating the queue waiting time of the MCA job on the queue. We predict queue waiting time of a job as the mean wait time of all jobs requesting greater than 80\% of the number of processors on the queue. The coordinator dynamically calculates the most recent value of queue waiting time when it invokes the rescheduling algorithm.

All possible comparison cases and the resulting decisions are shown in the function pseudocodes in Figures 6.3 and 6.4. The corresponding rescheduling decision tree is shown in Figure 6.5. There are a total of fifteen cases of comparisons corresponding to the fifteen leaf nodes of the decision tree. The color of each leaf node indicates the decision applicable in that particular case. We describe each of these fifteen cases in the next section.
6.4 Rescheduling Decision Cases

As mentioned in the previous section, our rescheduling algorithm begins with the Genetic Algorithm being invoked to get a set of best schedules. We consider schedules other than the best schedule because the best schedule might involve some queues with very low execution time limit making some other schedule with a similar execution rate but with queues involving higher execution time limits favorable. When two schedules involve the same queues, the time limits are the same and we therefore consider only the one with the better execution rate. Hence, we get a set $S$ of $N$ best schedules with unique queue sets.

The schedules are sorted in descending order of execution rates. $P$ is a point in execution where rescheduling decision is made. Now, consider the event at $P$. The event could be a queue leaving or a queue joining. If it is the former, then a rescheduling is inevitable. However, if the event is a new queue being active, the old schedule can be continued as such or a new schedule can be chosen. Further, once a decision to reschedule has been made, all the configurations in the set $S$ have to be evaluated and the best configuration has to be chosen.
Figure 6.6: Rescheduling Decision Case 1

We use a one-step look-ahead methodology to determine which is the best decision. For this, we compare each schedule c from the set S with the current best configuration, b, and update b to c if c is found to be better. The comparison involves various cases determined by various time estimates. The estimates needed are for tb and tc, the times of next event on b and c respectively. Based on the execution rates of b and c, and rescheduling overhead, we calculate T, tcb and tbc (described in the previous section).

When tb = tc and the base case is the existing configuration, by definition of T, the progress with c is better only beyond T, and hence, c is chosen only if tb = tc > T. The various other cases are shown in the figures. Cases 1 to 10 indicate comparison of a new schedule with the base case being the existing configuration, i.e. no reconfiguration vs reconfiguration.

Case 1: T < tb < tc  This is a case, shown in Figure 6.6, where continuation of the application execution without rescheduling on the current configuration, b, is compared with rescheduling to a new configuration, c. The rescheduling decision for this case primarily depends on the duration for which each configuration can last. The condition for Case 1 is that T < tb < tc, i.e, time-to-next-event with configuration b is less than the time-to-next-event on configuration c, and both are less than the crossover point T, beyond which c gives higher application progress than b. Since both configurations last beyond T, the application will
Figure 6.7: Rescheduling Decision Case 2

Figure 6.8: Rescheduling Decision Case 3
Figure 6.9: Rescheduling Decision Case 4

Figure 6.10: Rescheduling Decision Case 5
Figure 6.11: Rescheduling Decision Case6

Figure 6.12: Rescheduling Decision Case7
Figure 6.13: Rescheduling Decision Case 8

Figure 6.14: Rescheduling Decision Case 9
**Figure 6.15: Rescheduling Decision Case 10**

make greater process when rescheduled to configuration $c$ than when continued on the current configuration $b$ by time $tb$. Hence, the choice is $c$, i.e. rescheduling the application to execute on configuration $c$.

**Case 2: $tb < T < tc$** In this case, shown in Figure 6.7, we find that at $tb$ the progress made with $b$ is greater than the progress made with $c$. However, as can be noted, this is a case where the time to next event with $b$ is very low. At $tb$, the application has to be rescheduled to a different configuration. Considering this configuration to be $c$ (since $tc > tb$), we note that the progress made at $tc$, indicated by the red line, is lesser with $b$ chosen at $P$ than with $c$ chosen at $P$. Hence, the choice at $P$ is $c$.

**Case 3: $tb < tc < T; tc < tcb$** In this case, shown in Figure 6.8, though $tb$ is lesser than $tc$, the progress made by $b$ in $tb$ will not be achievable before $tc$ by schedule $c$ because $tc < tcb$. Hence, the choice at $P$ is $b$.

**Case 4: $tb < tc < T; tc > tcb; tcb < tb + OH$** In this case, shown in Figure 6.9, since $tc > tcb$, the condition described in Case 3 is absent. Also, if $b$ were chosen and a rescheduling to $c$ is made at that point, the rescheduling overhead would leave us with the same progress at $tb + OH$ as at $tcb$ if we had chosen $c$. Since $tcb$ is less than $tb + OH$, we get the same progress faster.
with c. Hence, choice c.

**Case 5: T < tc < tb; tbc < tb; tbc < tc + OH** This is also a case where the base configuration is the existing configuration. Similar to Case 1, in this case, shown in Figure 6.10, \( tc \) and \( tb \) are both greater than \( T \). However, in this case, \( tb \) is higher than \( tc \) (\( T < tc < tb \)). Also \( tb \) is greater than \( tbc \) (\( tbc < tb \)), the time when the application when executed on configuration \( b \) can achieve the same progress that can be achieved if executed on configuration \( c \) in time \( tc \). Thus, at some point \( tbc \) before \( tb \) choice \( b \) makes as much progress as choice \( c \) makes by \( tc \). Also, in this case, \( tbc < tc + OH \). This means that though at \( tc \), choice \( c \) performs better than choice \( b \), rescheduling at \( tc \) would result in a progress made at \( tc + OH \) to be equal to the progress made by \( b \) at an earlier point, \( tbc ( < tc + OH) \). Thus the application will make better progress if continued on the current configuration, \( b \), than to reschedule on the configuration \( c \), execute till \( tc \), and then reschedule back to \( b \), incurring a rescheduling overhead of \( OH \), and continue execution after \( tc + OH \), as shown by the dotted line in the figure. Hence, the choice for this case is \( b \), i.e. continue execution on the current configuration.

**Case 6: T < tc < tb with tbc > tc + OH and tbc < tb** This case, shown in Figure 6.11, is similar to Case 5, and the same argument applies except that since \( tbc > tc + OH \), choice is \( c \).

**Case 7: T < tc < tb with tbc < tc + OH and tbc > tb** This case, shown in Figure 6.12 is yet another case that is similar to case 5, but since \( tbc \) will not be reached on \( b \) due to the event at \( tb \), choice at \( P \) is \( c \). The red lines in figure show progress with both choices.

**Case 8: T < tc < tb with tbc > tc + OH and tbc > tb** In this case, shown in Figure 6.13, arguments of both Case 6 and Case 7 are applicable. Following either one of them, we get the choice at \( P \) to be \( c \). The red lines in figure show progress with both choices.

**Case 9: tc < T < tb** This is a simple case as indicated in Figure 6.14. Choice is \( b \), since even at \( tc \), \( b \) is better.

**Case 10: tc < tb < T** Similar to Case 9, this case shown in Figure 6.15 is a simple one. Choice is \( b \), since even at \( tc \), \( b \) is better.

The following cases are between two rescheduled configurations. Hence, the rescheduling overheads are common to both. \( b \) is the base configuration with higher execution rate since the
configurations are arranged in decreasing order of execution rates.

**Case 11: \( tb \geq tc \)** This is a case, shown in Figure 6.16, where both configurations \( b \) and \( c \) involve rescheduling and hence rescheduling overheads. Since in this case, configuration \( b \) gives higher execution rate than configuration \( c \) and has higher time to next event (\( tb > tc \)), the choice for this case is \( b \), i.e. reschedule to configuration \( b \).

**Case 12: \( tcb < tb + OH \) and \( tcb < tc \)** This is another case where both configurations involve the rescheduling overheads. In this case, though configuration \( c \) gives lower execution rate than \( b \), it has a higher time-to-next-event, i.e., \( tc < tb \). \( tc \) is also greater than \( tcb \), the time when the application if executed on \( c \) will achieve the same progress that can be achieved if executed on \( b \) in time \( tb \). Moreover, in this case, \( tcb < tb + OH \). Beyond \( tcb \), choice \( c \) outperforms choice \( b \) as indicated by the dotted line showing progress beyond \( tb \) with choice \( b \). Thus, as shown in Figure 6.17, the application will make better progress if rescheduled to configuration \( c \) than rescheduling to configuration \( b \), execute till \( tb \), then reschedule to \( c \) and continue execution after \( tb + OH \). Hence, the choice for this case is \( c \).

**Case 13: \( tcb < tb + OH \) and \( tcb > tc \)** This case, shown in Figure 6.18, is similar to Case 12. However, since \( tc < tcb \), as indicated by the red lines showing progress with each decision,
Figure 6.17: Rescheduling Decision Case12

Figure 6.18: Rescheduling Decision Case13
Figure 6.19: Rescheduling Decision Case 14

Figure 6.20: Rescheduling Decision Case 15
the more optimal choice is b.

**Case 14: tcb > tb + OH and tcb < tc** This case, shown in Figure 6.19, is yet another case similar to Case 12. However, unlike in Cases 12 and 13, tcb > tb + OH. Choice is b, because as indicated in the figure, rescheduling at tb from b to c gives a better progress than a choice c made at P.

**Case 15: tcb > tb + OH and tcb > tc** This case, shown in Figure 6.20, is yet another case similar to Case 12. In this case, both the conditional clauses of Case 12 are violated. As discussed in Cases 13 and 14, the better choice is therefore, b. The red lines in the figure show progress with each decision, and as expected, indicate better progress with b.

### 6.5 Experiments and Results

Our experiments with the rescheduling policies described in Section 6.3 were conducted with CCSM. The experiment setting with four batch queues on three clusters is the same as described in Section 5.5. Since our framework is intended for long-running multi-component applications, we focused on two main experiments corresponding to single-site and multi-site runs, each with execution duration of about 8 days, thus totaling more than 2 weeks of experiments.

External loads were simulated by submitting synthetic MPI jobs to the queuing systems based on the workload model developed by Lublin and Feitelson[93]. The maximum execution time limit for all jobs on all queues was set to 12 hours. The CCSM MPMD application was submitted with MPICH2 using the \texttt{-configfile} option. The coordinator was started on the front-end node on \textit{fire-16}. A job monitor and a job submitter corresponding to each queue were started on the front-end of its cluster.

While the 8-day single-site run on \textit{queue-64} performed climate simulations of 6 years, 10 months and 21 days, the 8-day multi-site run performed climate simulations of 7 years, 1 month and 24 days. This involved 187848 computational time-steps with multiple inter-process communications involved in each time-step. As the jobs on each of the four queues became active and inactive, the CCSM runs were automatically reconfigured and restarted by our framework. The execution profile of CCSM on the various queues during this multi-site execution is shown
in Figure 6.21.

The figure shows the location of execution of various CCSM components along the execution time-line as the configurations change. The figure comprises of four subplots corresponding to the four queues in our experiment, as indicated by the labels at their top right corners. The x-axis shows the experiment timeline in hours, while the y-axis has the total number of processors available in each queue. The colored regions correspond to the execution of CCSM, while the white regions correspond to processor-periods that are either unused or used by other jobs in the queue. Each color in the figure corresponds to a single component. For any given x-axis value corresponding to a given time instant, the components executing in each queue and the number of processes used by each component are indicated by the component-colors and the height of each color, respectively.

For example, during the 12\textsuperscript{th}-18\textsuperscript{th} hour of execution, the ocean component, represented by the light blue bar, is executed on queue-32 and the atmosphere component, represented by the dark blue bar, executed on queue-64. The execution begins on the first queue that becomes active, queue-14. However, within an hour, when queue-48 becomes active, all components migrate to the larger queue-48 and continue execution. At the 7\textsuperscript{th} hour, queue-32 becomes active and all components migrate from queue-48 to queue-32. Note that the migration happens from a larger (and hence faster) configuration to a smaller configuration. At the 12\textsuperscript{th} hour, when queue-64 becomes active, the rescheduler decides to use both active queues, with ocean continuing on queue-32 with a larger number of processors and other components migrating to queue-64. Thus, a variety of decisions, migrations and executions were observed during the 8-day run.

Note that the queue-14 was not involved in the executions because of the very small number of processors it contributes which do not sufficiently offset the inter-site communication overheads or the rescheduling overheads. Most of the rescheduling decisions corresponding to events in queue-14 resulted in no-rescheduling decisions, indicated by the blue bars in Figure 6.23. The experiment involved a total number of 34 rescheduling decisions and 20 reconfigurations of CCSM components performed automatically by our framework, involving non-trivial complex coordinations.
Figure 6.21: Execution profile of the application components on multiple sites
Whenever new batch systems become active or active systems reach execution time limit, our Morco framework automatically decides whether to reschedule, (if yes) stops the execution on the current configuration, calculates a new configuration with different component sizes and different locations, reconfigures the CCSM components to the new configuration and continues the execution. Figure 6.23 shows the points during the multi-site execution where events resulted in reconfigurations (red bars) and no reconfigurations (blue bars).

A similar execution profile for the single-site run is shown in Figure 6.22. As shown in the figure, there are larger gaps between two CCSM executions when CCSM is executed only on a single site than when it is executed across multiple sites using our Morco framework as shown in Figure 6.21. The total length of the gaps is 118 hours for single-site runs and only 53 hours for multi-site runs. Thus, multi-site executions of CCSM using our Morco framework ensure continuous progress and regular updates of long-running climate simulations.

Both the multi-site and single-site runs were performed for 8 days each, during which the multi-site run simulated 7 years, 1 month and 24 days of climate, while the single-site run simulated 6 years, 10 months and 21 days of climate. Figure 6.24 compares the execution progress of CCSM on multi-site runs with that on single-site runs. Each point in the figure
The multi-site run with adaptive rescheduling discussed in this chapter outperformed the multi-site run with greedy strategy adopted in the previous chapter. The average throughput improvement using the adaptive policy over the greedy policy is 18%. Figure 6.25 shows the percentage of time spent with different number of active queues during the multi-site runs. While the multi-site run with the greedy policy had a large percentage of time spent in configurations comprising of processors from all the four queues, the multi-site run with adaptive reschedul-
Figure 6.25: Percentage of Time Spent in Different Number of Active Queues in the Multi-site Executions with Greedy Rescheduling and Adaptive Rescheduling
Figure 6.26: Best execution rates for CCSM when executed across certain sets of Active Queues

Figure 6.27: Percentage of Time Spent in Different Queues in the Multi-site Execution with Adaptive Rescheduling
ing had only spanned a maximum of two queues, namely, *queue-64* and *queue-32*, located in a single department. Application execution on more than two queues will involve the use of slow and shared campus network connecting the machines of two departments. Hence the application execution rate does not increase substantially when involving processors of more than two queues as shown in Figure 6.26. Our rescheduling algorithm does not involve execution on more than two queues since the gains in execution rates will be offset by the rescheduling overheads.

Although the adaptive rescheduling run mostly did not involve simultaneous use of multiple queues as described above, it did span across the different queues at different points of time ensuring a smooth and continuous progress. Figure 6.27 shows the percentage of time spent in various queues during the multi-site execution with adaptive rescheduling. We note that almost 40% of the execution time is spent on queues with fewer processors than that used for the single-site execution. These results indicate that for CCSM, the benefit of multi-site execution is mostly due to the increased duration of availability (as shown by the smaller flat periods for adaptive rescheduling curve in Figure 5.7) than due to the use of larger number of processors. Thus our results demonstrate the use of grids for performance improvement of a significant scientific application like CCSM mostly without increasing the number processors used for single cluster executions, while most of the existing work on grid employs larger number of processors for performance improvement. Hence grids can be powerful paradigms even for applications with low to moderate scalability.

Multi-site executions will provide greater benefits for CCSM on batch grids where the average queue wait times are much higher than the job execution time limits, i.e. when the queues are heavily loaded with other external jobs. In such cases, the effect of continuous progress will be more substantial. Applications with significant better scalability than CCSM will obtain larger benefits with multi-site executions.

Within the coordinator, the overhead due to reconfiguration was around 7-8 minutes when a decision to reconfigure was made, of which 5-6 minutes were spent in restart file transfers and backups for fault tolerance. A no-reconfiguration decision was generally arrived at in less than a minute with no loss of execution progress since the current application run is not stopped.
Figure 6.28 gives the times consumed by various phases of the single and multi-site executions including the overheads. Idle time refers to the time when the CCSM jobs were not executing and were waiting in the queues. The remaining times correspond to executions of CCSM jobs when some of the systems become active. Some fraction of this active time is consumed by the multi-site execution overheads, while the remaining time is spent for useful CCSM computations. The most significant overhead is in the startup which includes MPI initializations across CCSM components, restart file reads, and initialization of various CCSM components. There are also noticeable overheads involved in compilation and preprocessing of components, as well as in writing and packing of restart files. A small overhead is also incurred by the transfer of restart files.

As the figure indicates, the multi-site executions have lower percentages of idle time than the single-site execution. The single-site executions as expected have very low overheads. The percentage of time spent in overheads decreases from 6% in multi-site executions with greedy rescheduling to 4% in multi-site executions with adaptive rescheduling. This is due to the fewer number of reconfigurations performed with the adaptive rescheduling. There is also an increase
in percentage of idle time and decrease in percentage of useful computation time with multi-site executions with adaptive rescheduling. This is expected because the greedy rescheduling policy makes use of all available resources while adaptive rescheduling policy uses available resources only when it is expected to improve the performance.

### 6.6 Conclusions and Future Work

In conclusion, we have developed Morco, a middleware framework for execution of multi-component applications on independently administered batch grids consisting of multiple batch systems. The framework, which is generic and non-intrusive, requires no special administrative privileges or co-allocated global scheduling. The framework with adaptive rescheduling, dynamic resource allocation and fault-tolerance, supports continuing execution across multiple time-distributed submissions on each queue as well as concurrent execution across submissions on multiple batch queues. With an experiment involving an 8 day-execution of a complex multi-component application, CCSM, on a batch grid with four batch queues on three systems, we have shown that our framework enables multi-site executions yielding good application throughput.

In the next chapter, we use our framework for performing very long duration runs for multi-century climate simulations with CCSM and investigate long-term climate phenomena including Indian monsoon.
Chapter 7

Climatology: Effect of Doubling Aerosols

Rapid industrial growth in India would lead to increase of black carbon aerosols which are primarily anthropogenic. Further, changes in land-use/land-cover patterns would lead to degradation of land and thus increase the amount of dust aerosols in the atmosphere. Since these could occur concurrently and both are known to have a significant impact on the Indian monsoon, we have used our grid framework to study their impact on climate. In this chapter, we describe the user of our middleware framework for long-duration runs to study the impact of black carbon and dust aerosols on climate. In section 7.1, we discuss briefly the details of the climate model used. Section 7.2 discusses the importance and relevance of the climate problem studied. Section 7.4 presents the validation results of model runs with observations. Sections 7.5 and 7.6 present the results we have obtained.

7.1 CCSM: Model Details

We have studied the effects of increased black carbon and dust aerosols on regional and global climate using the National Center for Atmospheric Research (NCAR) Community Climate System Model version 3 (CCSM3).

CCSM has been shown to produce realistic simulations over a wide range of spatial resolutions, enabling inexpensive simulations lasting several millennia or detailed studies of continental-scale dynamics, variability, and climate change. Applications of CCSM include
studies of inter-annual and interdecadal variability, simulations of paleoclimate regimes, and projections of future anthropogenic climate change [38].

In this work, we have used the T42, gx1v3 resolution specification of CCSM. The atmosphere/land components (T42) use a 128 longitudes X 64 latitudes grid with 26 vertical levels for atmosphere. The horizontal grid resolution is approximately 2.8°. The ocean/sea ice components (gx1v3) use a finer grid of 320 longitudes X 384 latitudes. In the ocean model, based on the Parallel Ocean Program (POP) [136], there are 40 vertical levels associated with the gx1v3 resolution, with level thickness monotonically increasing from approximately 10 to 250 meters. In the horizontal grid of POP, the longitudinal resolution is approximately one degree and the latitudinal resolution is variable, with finer resolution near the equator (approximately 0.3°). The north pole has been displaced into Greenland.

7.2 Black carbon and dust aerosols

In CCSM3 five species of aerosols are parameterized: sea salt, soil dust, black and organic carbonaceous aerosols, sulphate and volcanic sulfuric acid. [38]. Large solid dust and black carbonaceous aerosols, the aerosols whose impact on the climate is the focus of this work, are strongly absorbing in the visible wavelengths. We specifically study the impact of doubling the concentrations of these aerosols on the Indian Monsoon.

The black carbon aerosols are important anthropogenic aerosols with their major sources being increasing industrial and vehicular pollution. Figure 7.1 shows the distribution of black carbon aerosols over the tropics. Note the high concentration of black carbon over Indian and Chinese regions in the annual mean. It is lower during JJAS and July due to raining out of aerosols. However as shown by Chakraborty et al (2004), high concentration of black carbon aerosol during the pre-monsoon season, can have a significant impact on the succeeding monsoon. Soil dust aerosols are also important for the Indian region due to the large amounts of dust aerosols being generated in the deserts stretching from Rajasthan to Saudi Arabia. Figure 7.2 shows the distribution of atmospheric dust aerosols over the tropics.

A large number of studies with various models have been performed to study the climatic
Figure 7.1: Distribution of atmospheric black carbon aerosols (July, JJAS and Annual) over the tropics (in g/m^2)
Figure 7.2: Distribution of atmospheric dust aerosols (July, JJAS and Annual) over the tropics (in g/m²)
impact of aerosols, especially in the Indian region. Menon et al. (2002) [98] investigated the effects of black carbon aerosols in China and India using a global climate model. They noted that the absorbing black carbon aerosols heat the air, alter regional atmospheric stability and vertical motions, and affect the large-scale circulation and hydrological cycle with significant regional climatic effects. Chakraborty et al. (2004) [1] studied the impact of anthropogenic absorbing aerosols on the climate over the Indian region using the NCMRWF general circulation model. They show an overall increase in rainfall and a reduction in surface temperature in the Indian region. Ramanathan et al. (2005) [146] have performed coupled ocean-atmosphere simulations to study the impacts of atmospheric brown clouds in South Asia on regional climate and hydrological cycle. Their studies have also indicated that aerosols have significant climatic effects. Lau et al. (2006) [87], with a global circulation model, have suggested that increased dust load coupled with black carbon emissions lead to Asian summer monsoon anomalies. More recently, Zhang (2009) [155] has studied the effect of the collective effects of anthropogenic aerosols using CCSM3. Their experiments involve increasing aerosols locally and decreasing aerosols globally, and they have incorporated MOZART [106] simulation based dynamic footprints of the increased Asian emissions.

In our experimental design, we have doubled the black carbon and dust aerosols parameters of CCSM3 globally and note its impact on the Indian monsoons. We have performed two runs: 25 year control run (CTL) and a 25 year perturbed run with the doubled aerosol concentrations (2*A). We used a spun-up initial condition of 654 years for both the runs.

The aerosol doubling was made by setting CARSCL and DUSTSCL to 2 in the CAM namelist input file, specifically the atm.stdin generating part of the cam.buildnml_prestage.csh file. CARSCL and DUSTSCL are respectively the carbon aerosol and the dust aerosol scaling factors for radiative transfer calculations.

We used NCAR Command Language (NCL) to generate the plots shown in this chapter. We have shown two kinds of plots with respect to the period concerned. JJAS plots show the monthly averages over the monsoon months of June, July, August and September. The other set of plots for the average values for July are shown because July is the month with the peak of the Indian monsoon.
The tropical region plots show the region between latitudes -30 to 30 for all longitudes from 0 to 360. The Indian region plots show the region enclosed by latitudes -10 to 30 and longitudes 40 to 120.

### 7.3 Morco: Grid Middleware

The experiments involved execution of CCSM across four queues on three clusters located in two different sites: CAOS, IISc and SERC, IISc. The four queues of sizes 14, 32, 48 and 64 were configured using OpenPBS. The two long climate runs involved multiple submissions, starts, stops, reconfigurations, migrations, restarts, etc. that were performed automatically by our grid middleware framework, Morco. The runs lasted a period of around 2 months with hundreds of reconfigurations, with the maximum execution time limit on each queue being around 12 hours. On an average, there were approximately 5 queue events per day- 4 per day involved a decision to reconfigure and 1 per day involved a decision to not reconfigure. During certain phases of unavailability of one (or some) of the clusters, the execution could progress on the other clusters due to the fault-tolerance provided within Morco. Similarly, the runs could continue despite disruptions due to limited disk space, power failure, etc. Hence, our middleware was useful in enabling long duration climate simulations across multiple queues.

### 7.4 Validation of Precipitation

We have used 346 month precipitation data from the GPCP Version 2x79 Experimental Combined Precipitation Dataset [54] for validation of our 25-year control run. Figures 7.3, 7.4, 7.5 and 7.6 respectively compare the model output with observations for precipitation in tropics:JJAS, tropics:July, Indian region:JJAS and Indian region:July. As can be seen in these figures, the model output shows similar spatial variations of global and regional precipitation as the observations.

In particular, large parts of north and south pacific ocean, north and south Atlantic ocean, central parts of south America, northern and southern parts of Africa and Australia have low amounts of total precipitation of less than 2 mm/day. Maximum rainfall rate in these months is
Figure 7.3: Validation: tropics precipitation (JJAS)

observed as around 12-16 mm/day in the equatorial regions across all longitudes. The validation plots for the month of July shown in Figure 7.4 has similar spatial variations as the JJAS months in Figure 7.3.

The validation plots in Figures 7.5 and 7.6 show a comparison of the precipitation in the Indian region. In all these, the western parts have a low precipitation rate of less than 2 mm/day and the Indian rainfall ranges from 4 mm/day to 12 mm/day in most parts of the country.
Figure 7.4: Validation: tropics precipitation (July)
Figure 7.5: Validation: Indian region precipitation (JJAS)
Figure 7.6: Validation: Indian region precipitation (July)
7.5 Impact on Indian Monsoon

The impact of doubling black carbon and dust aerosols on the precipitation (JJAS and July; tropics and Indian region) are shown in the Figures 7.7 and 7.8. The figures are the difference plots of precipitation output of the doubled-aerosol run and the control run. As can be observed, the increased aerosols result in a decrease in the rainfall in the Indian Ocean and the Arabian Sea, and an increase in rainfall in the Bay of Bengal region and in parts of central India. The plots showing the global tropics show that the precipitation rate decreases by more than a mm/day in parts of various oceans close to the equator and increases by more than a mm/day in the south-east Asian regions. While the spatial variations are similar in both the JJAS and the July plots, the magnitude of impact is slightly higher in the precipitation rates for the month of July.

Figure 7.9 shows the statistical significance of the precipitation data [116]. The probability values corresponding to a standard t-test with a 0.1 (or 90%) level of significance and different variances for the null hypothesis of both values coming from statistically identical populations. A probability of less than 0.1 indicates a rejection of hypothesis thereby establishing the significance of the values at those points, as outcomes of the perturbations introduced in the aerosol parameters.

Therefore, the most significant parts of the observation are in the Arabian Sea region where the impact of increased aerosols was noted to be a decrease in precipitation, and in the central parts of India where the impact of increased aerosols was noted to be an increase in precipitation.

Figure 7.10 shows the interannual variability of precipitation rates for the month of July. The figure shows that both the control as well as the perturbed runs had similar standard deviation patterns in the Indian region. Higher variation is observed in the regions in and around Gujarat in the western part of India and in the Bay of Bengal in the East.

Figure 7.11 shows the area-averaged rainfall for JJAS over the Indian region for each of the 25 years of the control and doubled aerosol runs. The rainfall indicated is for the land region enclosed withing the coordinates 5N-27.5N, 65E-90E. A mean of 6.4 mm/day and a standard deviation of 0.6 mm/day were observed for both the datasets. The area averaged mean rainfall and standard deviation do not show any significant changes. The impact of aerosols appears to be to redistribute the rainfall over the Indian region. We note that the frequency of
Figure 7.7: Impact of doubled aerosols on precipitation (JJAS)
CHAPTER 7. CLIMATOLOGY: EFFECT OF DOUBLING AEROSOLS

Figure 7.8: Impact of doubled aerosols on precipitation (July)

(a) Total mean precipitation (July) over the Indian regions-difference between doubled aerosol and control

(b) Total mean precipitation (July) over the tropics-difference between doubled aerosol and control
(a) Statistical Significance-precipitation data (July); t-test

(b) Difference plot with the most significant (90%) region contoured

Figure 7.9: Statistically significant differences in precipitation
Figure 7.10: Standard deviation for monthly precipitation rate (July)
Figure 7.11: Impact of doubled aerosols on area-averaged rainfall
droughts/excess rainfall (generally defined as 1 std deviation above or below the mean) changes with changes in aerosol loading. While the control run had 4 years of droughts (years 1, 6, 10 and 18) and 2 years of excess rainfall (years 15 and 17), the doubled aerosol run had 2 years of droughts (years 8 and 25) and 4 years of excess rainfall (2, 12, 14 and 16).

### 7.6 Impact on Sea Surface Temperature

Figures 7.12 and 7.13 show the difference plots in the Sea Surface Temperature between the doubled aerosol and the control runs, for the JJAS and July months, over the tropics and the Indian region. The global tropics plots indicate a slight decrease in the sea surface temperature in almost all parts of the tropics with a higher decrease in parts of Arabian Sea, the Persian gulf, the Red Sea and in small parts of the Atlantic ocean close to North Western Africa. Note that the regions over Arabian Sea and over Atlantic also have high dust aerosol concentrations. As dust partially absorbs and partially reflects incoming solar radiations and hence reduces the incident radiation at the surface, this could be causing the SST to decrease. The reduction in Arabian Sea SST also is quite close to the region where rainfall decreases significantly. The reduction in SST could make this region less favorable for the occurrence of rainfall (through enhanced vertical stability). We note a similar pattern of reduced SST and rainfall off the coast of Africa where again dust aerosols may be playing a significant role. Over land the interactions could be more complex. Both dust and black carbon could absorb the radiation. This could result in reduced moist convective instability. On the contrary, reduction of radiation to the surface could result in lower surface temperature which could increase the stability. The decrease in rainfall over oceanic regions could be related to increased stability and lower moisture availability. Over continental regions the reduced moist static stability could be the cause of increased precipitation as suggested by Chakraborty et al (2004) [1].

From the plots showing the Indian region, effect of doubling the aerosols on the sea surface temperature is observed to be a slight decrease over most of Indian ocean with a larger decrease in temperature in the Persian gulf regions. The observations in both the tropics and the Indian regions do not vary much between the JJAS and the July plots. However, it can be noted that
higher magnitudes of decrease in the sea surface temperature is observed over larger regions in the month of July. Also, the decrease in SST is largest to the west of Sahara Desert (i.e. west of Africa) and to the west and south of India.

Figure 7.14 shows the statistical significance of the precipitation data for July. The values are the probabilities obtained from an independent two-sample t-test with unequal variances and level of significance equal to 0.1 over the control and perturbed datasets. The null hypothesis of the equivalence of the two means is rejected when the p-value is less than 0.1, which is true for most parts of the Indian ocean shown in the plot (Note that the larger regions contoured in Figure 7.14(b) are the significant regions). Hence, the result of decrease in sea surface temperature due to increased aerosols is statistically significant.

7.7 Conclusions

We have used our grid middleware framework to study the impact of increased black carbon and dust aerosols on the climatic conditions, especially the Indian monsoons. Our observations indicate that increased aerosols result in an increase in precipitation in the central and eastern parts of India, and a decrease in precipitation in parts of the Arabian Sea. We also noticed a slight decrease in sea surface temperature (SST) over most of Indian ocean and a larger decrease in SST in the Persian gulf regions. Regions with largest decrease in SST appear to be associated with regions with highest dust concentrations. Over Arabian Sea where SST reduces, there is a significant reduction in rainfall.
Figure 7.12: Impact of doubled aerosols on SST (JJAS)
Figure 7.13: Impact of doubled aerosols on SST (July)
Figure 7.14: Statistically significant differences in SST
Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis, we have explored the execution of long-running coupled multi-component climate applications on batch grids. We have used Community Climate System Model (CCSM), an important long-running coupled multi-component application, as a significant case study for demonstration of our strategies for grid executions.

We have firstly conducted profiling studies of the scientific application, CCSM. We had identified significant load imbalances: intra-component as well as inter-component. To address these temporal load imbalances, we proposed the Dynamic Component Extension technique for minimizing processor idling and improving application performance. By automatic identification of idling points and dynamic determination of points and amounts of extension, we extended the computation intensive long wave radiation calculations of the atmosphere component to the processors executing other components. By evaluating our strategy on 5 different parallel platforms for different number of processors, we showed that our strategy can reduce CCSM execution times by about 15% and save several days of execution for 1000-year simulation runs. We also showed that the overheads due to our strategy are negligible and that our strategy can give increasing benefits with increasing frequency of radiation calculations and increasing scalability of dynamics phase of CCSM. Our strategy can enable climate researchers to conduct multi-century climate runs at a reasonable time even on moderately sized clusters.
common in academic institutions.

We then performed extensive simulation studies to evaluate the benefits of batch grids for long-running applications. With initial statistical studies, we have shown that even without an increase in the number of processors, applications can gain from execution on multiple batch grids due to lower queue waiting times corresponding to the lower processor requirements on individual sites of the batch grid. Specifically, there are large percentages of configurations with significant probabilities of benefits with multiple batch executions for both multi-component CCSM simulations and parameter sweep applications. We also found that for queues with backfilling policies, the probabilities of benefits with multiple batch executions are higher. Finally, for the CCSM multiple batch executions, the chances of benefits are higher for larger inter-site bandwidths.

We then performed detailed simulation studies with four different execution models for execution of long-running multi-component applications across multiple independent batch systems: WaitForAll, WaitTillThresholdandAbort, WaitTillThresholdandExecute and NoWait. We determined that the NoWait strategy, in which all available resources are used at all times, performed the best and hence, conducted further simulation studies with the NoWait strategy. We constructed an application-level simulator for modeling a foremost multi-component application, CCSM, and a comprehensive system simulator for modeling the characteristics of multiple batch systems. By means of large number of simulations with different application and system configurations, we showed that multiple batch executions can reduce the queue waiting times of CCSM jobs by an average of 60%, and increase the throughput of CCSM application by an average of 55% over single batch executions. We also analyzed in detail, the impact of request sizes of CCSM jobs, inter-cluster bandwidths, characteristics of external workloads on the batch systems, and queue scheduling policies on the benefits due to multiple batch executions. We found that small request sizes of CCSM jobs, and presence of large number of narrow and short jobs in the external workloads lead to larger benefits while inter-cluster bandwidths and queue scheduling policies do not impact the benefits due to multiple batch executions.

Having established the advantages of execution across multiple sites with simulation studies, we developed a grid middleware, Morco, that handles the challenges of real execution of
CCSM on multi-site grids. Morco is a middleware framework for execution of multi-component applications on batch grids consisting of multiple batch systems. The framework, which is generic and non-intrusive, requires no special administrative privileges or co-allocated global scheduling. The framework with adaptive rescheduling, dynamic resource allocation and fault-tolerance, supports continuing execution across multiple time-distributed submissions on each queue as well as concurrent execution across submissions on multiple batch queues. With an experiment involving an 8 day-execution of a complex multi-component application, CCSM, on a batch grid with four batch queues on three systems, we have shown that the framework performs complicated tasks including coordination of the different startup times of the component jobs on the different batch queues and rescheduling of the component jobs based on the number of active systems for execution. The experiments also showed that multi-site executions can give better application throughput than single-site executions in real environments. Moreover, the multi-site executions avoids the bottlenecks associated with single-site executions that are currently practiced by the climate scientists.

Finally, we used the middleware framework for two 25-year long executions of CCSM to study the effect of increased dust aerosols and black carbon aerosols on Indian monsoons. Our studies showed that there is a significant variation in the sea surface temperature and rainfall patterns due to the increase in aerosols. For example, a statistically significant decrease in sea surface temperature was noticed in the Persian gulf regions.

Summarizing the significance of our work, ours is the first work on malleability of long-running multi-component applications on multiple batch queue environments. This is a very important contribution of our work, since our analysis and demonstration with real experiments will motivate and allow leveraging the power of multiple batch systems for solving large-scale scientific problems, both (batch systems and large applications) of which are becoming widely prevalent. Ours is also the first work in temporal and inter-component load-balancing, enabling multi-site CCSM execution, quantifying performance benefits of multi-site batch executions, and in robust execution of long-running multi-component applications on multiple batch queues. Thus, our work has made important contributions in load-balancing and multi-site execution of multi-component applications, and will help enable more efficient execution of large
8.2 Future Work

This thesis has performed explorations of multi-site executions of long-running multi-component applications, modeled on the basis of CCSM. The explorations have opened avenues for future research and building framework. Some possibilities of future work are:

1. **Exploration of applications with larger number of components:** In this work, we have considered a maximum of five components since we have focused on CCSM. It could be interesting to explore applications with larger number of components. Applications with larger number of components can be executed on grids with larger number of sites and this would require all parts of the framework solution to be scalable. For example, the performance model used in the rescheduling algorithm in its current form involves construction of look-up table with the best configurations for all possible queue availability scenarios which is feasible only for a small number of queues.

2. **Restricted component malleability:** An assumption involved in this work is that the application components are all malleable. While there were restrictions on the number of processors used to execute each component of CCSM, all components were malleable. Future research could involve exploring the impact of having some non-malleable components on the multi-site execution of CCSM. Malleable components make it possible to shrink or expand to fit and continue executions on smaller or larger number of available processors. The presence of non-malleable components would imply a certain minimum number of processors necessary for execution and this can render many available queue scenarios unfit for execution. Non-malleability of components would also limit performance gains being obtained from execution on larger number of processors. Effective multi-site execution solutions for such cases therefore need to be developed.

3. **Execution of a single component across multiple sites:** In CCSM, due to the larger intra-component communication costs, we chose to execute each component within a
site. A highly scalable, computation intensive component with low intra-component communications can be an ideal candidate for execution across multiple sites. Applications with such components may show significant benefits due to multi-site execution since the larger number of processors available on a multi-site execution may be more effectively utilized.

4. **Building generic enabling frameworks for use by different multi-component applications:** As mentioned above, some of the design decisions and characteristics of our middleware are CCSM specific. A completely generic enabling framework for use by different multi-component applications would involve carefully resolving these CCSM specific issues. It should be scalable to large number of processors, components and queues, and also effectively handle restricted component malleability, components with evolving characteristics, component execution across multiple sites, etc.

5. **Exploration of different practical-to-use execution models:** While we have explored some execution models in our research, there are many more execution models, for example, with variable request sizes or for multiple applications, that could be explored, studied, evaluated and implemented. A variable request size execution model would involve taking into account the queue statistics, external load characteristics, application requirements and current global application execution state to dynamically change the number of processors requested in any given queue. Development of execution models involving simultaneous performance improvement of multiple applications or multiple instances of an application can be an even more challenging extension of our research.

Yet another problem to be considered is the determination of an optimal request size such that the resource availability is maximized while the resource idling is minimized. The rescheduling policy in particular can be further enhanced to dynamically release idling processor sets to queueing system for use by other jobs.

Other future work could include execution of CCSM or another multi-component application on large and highly dynamic and heterogeneous grid of resources, in contrast to our testbed of campus-wide batch grids.
References


[34] climateprediction.net. *http://www.climateprediction.net/*.


[51] ESG. *http://www.earthsystemgrid.org/*.


[78] Intergovernmental Panel on Climate Change.  *http://www.ipcc.ch/*.


[92] Load Sharing Facility. [http://www.platform.com/Products/Platform.LSF.Family/](http://www.platform.com/Products/Platform.LSF.Family/).


[106] MOZART (Model for OZone And Related chemical Tracers). http://www.acd.ucar.edu/gctm/.


[155] Zhang Y, Sun S, Olsen S C, Dubey M K, and He J. CCSM3 Simulated Regional Effects