Advanced Middleware Support for Distributed Data-Intensive Applications

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Wei Du, M.S. B.S.

* * * * *

The Ohio State University

2005

Dissertation Committee:

Dr. Gagan Agrawal, Adviser
Dr. P.Sadayappan
Dr. Atanas Rountev

Approved by

____________________
Adviser
Graduate Program in
Computer Science and
Engineering
© Copyright by

Wei Du

2005
ABSTRACT

In recent years, people have developed numerous ways to collect and generate data. The size of datasets has been growing dramatically, and these datasets are often distributed across multiple sites. Processing such large-scale distributed datasets plays an increasingly important role in many domains like life sciences, high energy physics, environmental sciences, chemistry, among others. This class of applications often involves access to data repositories or data collection instruments like sensors, and requires fast or even interactive response time. The broad goal of our work is to provide advanced middleware support to ease the development of distributed data-intensive applications, and facilitate their efficient execution.

A coarse-grained pipelined execution model provides a natural vehicle for executing such applications on the distributed or grid environment. Here, the processing associated with an application is carried out at several stages, which are executed on a pipeline of computing units. Each stage handles the intermediate results obtained from the previous one, and packs the current output for the following stage. Typically, the first stage in this pipeline is the unit where the input data is available, and the last stage is where the final results are to be viewed.

In this thesis, I present language and compiler supports for developing such applications. These supports could allow programmers to focus on writing a sequential code to specify the application-specific tasks without worrying about the details not
directly related to applications. Our extended language constructs can help expose both pipelined and data parallelism to the compiler. Then our compilation system is responsible for selecting a set of candidate filter boundaries, determining the volume of communication required if a particular boundary is chosen, performing the decomposition, and generating code for execution on coarse-grained pipelined execution model.

The filter decomposition problem is separately investigated and formulated in a mathematical way. Three different algorithms are proposed and tested. Algorithm MIN_ONETRIP is a dynamic programming algorithm, optimizing the one trip cost for a packet passing through the pipeline. Another dynamic programming algorithm, MIN_BOTTLENECK, minimizes the time spent on the bottleneck stage. Finally, MIN_TOTAL is an approximate greedy algorithm that tries to minimize the total execution time. Results show that the heuristic algorithms work quite well in practice, with the possible exception of MIN_ONETRIP when the number of packets became larger.

Packet size is an important factor that impacts performance of applications executed in the coarse-grained pipelined fashion. It defines the unit of data transfer from one pipeline stage to another. We developed and validated an analytical model for choosing the right packet size for an application. Since the pipeline includes both communication and computation phases, the frequency and/or volume of communication between two different phases can be varied. Two communication models, fixed-frequency and fixed-size models, are studied, and a series of mathematical expressions are derived for both.
To support the adaptivity aspect of the applications, we adopted a hybrid methodology, which combines compile-time analysis with runtime feedback. A program analysis algorithm states the execution time of an application component as a function of the values of the adaptation parameters and other runtime constants. These constants are determined by initial runs of the application in the target environment. Based on these calculated constants, adaptation parameters can be modified and corresponding adjustment steps allow user to achieve a given change in execution time, or to retain the same performance under a particular variation in available resources.

Substantial amount of experiments have been carried out to assess this dissertation, which shows the proposed algorithms and models work quite effectively in practice. The dissertation concludes with a set of open research questions that frame the future work.
This is dedicated to my grandmother.
ACKNOWLEDGMENTS

I would like to thank everyone who helped me during my graduate study at the Ohio State University. It is hard to name them all here, and it is hard to thank them enough.

First and foremost, I want to thank my advisor professor Gagan Agrawal for his constant support and guidance. He has offered invaluable advice and instruction to me on identifying problems, conducting research, and fine polishing solutions. His diligence and commitment to science have been and will be a great influence on me for many years to come. I am grateful for having the opportunity to learn from him and to work with him. I also thank Professor Atanas Rountev, Professor P. Sadayappan for spending time to serve in my committee, advising me in my research and helping me with my dissertation.

I have also learned a lot from my fellow graduate students and colleagues in the High End Computing Systems Group. I thank Renato Ferreira for sharing insights and thoughts during the initial stage of my Ph.D study. I also want to thank my labmates Ruoming Jin, Xiaogang Li, Liang Chen, Xuan Zhang, Ge Yang, Leo Glimcher, and Li Weng for making the work space quite fun through lively and interesting discussions. Many of the research findings in this dissertation directly or indirectly originated from discussions and collaboration with many brilliant people, in addition
to those I just mentioned. Also, I want to express my thankfulness to the members of DataCutter project, including Tahsin Kurc, Umit Catalyurek, Mike Beynon, Alan Sussman, and Joel Saltz, for providing the DataCutter implementation, manual codes, datasets, and for helping with the experiments.

Finally I want to thank my parents for their unconditional support, understanding and patience in all my endeavors. Without their support this dissertation simply would not have been possible. Special thanks to my husband Dahai Gai. His love has made me capable of weathering all the ups and downs throughout the ordeal of my doctoral study.
VITA

1995 ................................. B.E. Computer Science & Technology
         Heilongjiang University, Harbin, China
1998 ................................. M.E. Computer Science & Technology
         Heilongjiang University, Harbin, China
2001 ................................. M.S. Computer & Information Science
         University of Delaware, Newark, DE.,
         USA
2005 ................................. Ph.D Computer Science & Engineering
         Ohio State University, Columbus, OH.,
         USA

PUBLICATIONS


Wei Du and G. Agrawal, ”Using General Grid Middleware and Compiler Technology for Distributed Data Mining: Preliminary Report”, in workshop on High Performance Data Mining, as part of SIAM conference on Data Mining (SDM), April 2002.

**FIELDS OF STUDY**

Major Field: Computer Science and Engineering

Studies in:

- High Performance Computing  Prof. Gagan Agrawal
- Computer Networks  Prof. Dong Xuan
- Computational Theory  Prof. Gagan Agrawal
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td></td>
<td>Dedication</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>Acknowledgments</td>
<td>vi</td>
</tr>
<tr>
<td></td>
<td>Vita</td>
<td>viii</td>
</tr>
<tr>
<td></td>
<td>List of Tables</td>
<td>xiv</td>
</tr>
<tr>
<td></td>
<td>List of Figures</td>
<td>xv</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.1 The Problem</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1.2 Issue 1: Support for Coarse-Grained Pipelined Execution Model</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1.3 Issue 2: Support for Application Adaptation</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>1.4 Contributions</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>1.5 Dissertation Roadmap</td>
<td>9</td>
</tr>
<tr>
<td>2.</td>
<td>Background</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2.1 Target Distributed Data-Intensive Applications</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2.2 Issues and Challenges</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>2.3 Coarse-Grained Pipelined Parallel Execution Model</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>2.4 DataCutter: A Middleware for Data Intensive Computations in a Grid Environment</td>
<td>18</td>
</tr>
</tbody>
</table>
3. Filter Decomposition for Supporting Coarse-Grained Pipelined Parallelism

3.1 Introduction ................................................................. 22
3.2 Filter Decomposition Problem ........................................ 24
  3.2.1 Cost Model ............................................................ 25
  3.2.2 Filter Decomposition Problem .................................... 26
3.3 Algorithms for Filter Decomposition ................................. 28
  3.3.1 MIN_ONETRIP Algorithm ....................................... 29
  3.3.2 MIN_BOTTLENECK Algorithm ................................. 34
  3.3.3 MIN_TOTAL Algorithm .......................................... 39
  3.3.4 Discussion ........................................................... 40
  3.3.5 A Brute Force Approach ......................................... 42
3.4 Experimental Results .................................................... 42
  3.4.1 Experimental Setting and Applications ......................... 43
  3.4.2 Experiments with Virtual Microscope ......................... 46
  3.4.3 Experiments with Iso-Surface .................................. 48
  3.4.4 Discussion ........................................................... 51
3.5 Conclusions ............................................................... 52

4. Packet Size Optimization for Supporting Coarse-Grained Pipelined Parallelism

4.1 Introduction ............................................................... 57
4.2 Packet Size Optimization Problem ................................... 59
  4.2.1 Overview of the Problem ......................................... 59
  4.2.2 Solution Approach ................................................ 60
4.3 Choosing Optimal Packet Sizes ........................................ 63
  4.3.1 Fixed-Frequency Communication Pattern ..................... 63
    4.3.1.1 Bottleneck at the First Stage ............................. 65
    4.3.1.2 Bottleneck at Second or Later Stages .................... 66
  4.3.2 Fixed-Size Communication Pattern ............................ 67
    4.3.2.1 Bottleneck at Second or Later Stages .................... 68
    4.3.2.2 Bottleneck at First Stage ................................ 70
  4.3.3 Applicability to the Grid Environment ......................... 72
4.4 Experimental Results .................................................... 72
  4.4.1 Experimental Setting and Applications ......................... 73
  4.4.2 Measuring Pipeline Parameters .................................. 75
  4.4.3 Evaluation of Fixed-Frequency Model ......................... 77
    4.4.3.1 K-Nearest Neighbor Search ............................... 77
    4.4.3.2 Zbuffer Based ISO-Surface Rendering .................... 78
    4.4.3.3 ActivePixel Based ISO-Surface Rendering ............... 79
4.4.4 Evaluation of Fixed-Size Model ........................................ 80
  4.4.4.1 K-Nearest Neighbor Search .................................... 80
  4.4.4.2 Zbuffer Based ISO-Surface Rendering ....................... 80
  4.4.4.3 ActivePixel Based ISO-Surface Rendering .................. 81
4.5 Summary ................................................................. 81

5. Language and Compiler Support for Exploiting Coarse-Grained Pipelined Parallelism ................................. 88
  5.1 Introduction .......................................................... 88
  5.2 Data Parallel Java ................................................... 90
  5.3 Compiler Analysis .................................................... 98
    5.3.1 Overview of the Compilation Problem ....................... 98
    5.3.2 Required Communication Analysis ............................ 100
  5.4 Code Generation Issues ............................................. 104
  5.5 Experimental Results ................................................ 108
    5.5.1 Applications .................................................... 109
    5.5.2 Experiment Design .............................................. 110
    5.5.3 Isosurface Algorithms .......................................... 114
    5.5.4 k-nearest Neighbor ............................................. 115
    5.5.5 Virtual Microscope ............................................ 118
  5.6 Summary ............................................................. 119

6. Language and Compiler Support for Adaptive Applications .......................................................... 121
  6.1 Introduction .......................................................... 121
  6.2 Target Applications .................................................. 123
  6.3 Language Extensions ................................................ 125
    6.3.1 Adaptation Constructs ......................................... 128
  6.4 Adaptation Approach and System Architecture .................. 128
    6.4.1 Program Analysis for Supporting Program Adaptation .... 132
    6.4.2 Adaptation Module ............................................. 136
  6.5 Experimental Results ................................................ 137
    6.5.1 Experimental Settings and Applications .................... 137
    6.5.2 Results from Virtual Microscope ............................ 139
    6.5.3 Results from Comp-Steer .................................... 140
    6.5.4 Results from K-Means Clustering ............................ 140
    6.5.5 Summary of Results ........................................... 141
  6.6 Summary ............................................................. 142
7. Related Work ..................................................... 151
   7.1 Support for Large-Scale Data-intensive Applications .......... 151
      7.1.1 Grid Middleware Systems .................................. 152
      7.1.2 Workflow Systems .......................................... 153
      7.1.3 Other Efforts ................................................ 155
   7.2 Support for Coarse-Grained Pipelined Parallelism ............... 157
   7.3 Scheduling Chained-tasks on Chained Processors ............... 158
   7.4 Communication Analysis and Optimization ...................... 158
   7.5 Support for Application Adaptation ........................... 160
      7.5.1 Middleware Support ....................................... 160
      7.5.2 Language and Compiler Support .......................... 161
      7.5.3 Other Efforts .............................................. 162

8. Conclusions ..................................................... 164
   8.1 Contributions .................................................. 164
   8.2 Future Work .................................................. 166

Bibliography ..................................................... 170
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Possible Placements and their Predicted Performance</td>
<td>41</td>
</tr>
<tr>
<td>3.2</td>
<td>Configurations Used for Our Experiments</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Placements Generated by Different Algorithms: Virtual Microscope, Config 1</td>
<td>47</td>
</tr>
<tr>
<td>3.4</td>
<td>Placements Generated by Different Algorithms: Virtual Microscope, Config 2</td>
<td>49</td>
</tr>
<tr>
<td>3.5</td>
<td>Placements Generated by Different Algorithms: Virtual Microscope, Config 3</td>
<td>51</td>
</tr>
<tr>
<td>3.6</td>
<td>Placements Generated by Different Algorithms: Virtual Microscope, Config 4</td>
<td>53</td>
</tr>
<tr>
<td>3.7</td>
<td>Placements Generated by Different Algorithms: Iso-Surface Rendering, Config 1</td>
<td>54</td>
</tr>
<tr>
<td>4.1</td>
<td>Summary of All Models</td>
<td>70</td>
</tr>
<tr>
<td>4.2</td>
<td>Parameters for Computational Phases for All Applications</td>
<td>76</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>K-nearest Neighbor Search Algorithm Implemented Using Coarse-Grained Pipelined Parallel Execution Model</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>The decomposition of the sequential k-nearest neighbors algorithm into three filters: \textit{range_query}, \textit{select} and \textit{combine}. Arrows show the connectivity among filters via streams.</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Specifying filters, layout and placement with DataCutter Interfaces.</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Execution Time-line of a Pipeline with Three Computation Stages and Two Communication Stages. \textit{Stage}_3 is the bottleneck stage.</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>MIN_ONETRIP Algorithm for Filter Decomposition</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>GetPlacement Procedure for Outputting the Placement from MIN_ONETRIP Algorithm</td>
<td>32</td>
</tr>
<tr>
<td>3.4</td>
<td>Recursive Definition of $N[i,j]$</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>MIN_BOTTLENECK Algorithm for Filter Decomposition</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>GetPlacementB Procedure for Outputting the Placement from the MIN_BOTTLENECK Algorithm</td>
<td>38</td>
</tr>
<tr>
<td>3.7</td>
<td>MIN_TOTAL Algorithm for Filter Decomposition</td>
<td>39</td>
</tr>
<tr>
<td>3.8</td>
<td>Exhaustive Search Algorithm for Filter Decomposition</td>
<td>43</td>
</tr>
<tr>
<td>3.9</td>
<td>Execution Time for Virtual Microscope: Config 1</td>
<td>46</td>
</tr>
</tbody>
</table>
### 3.10 Execution Time for Virtual Microscope: Config 2
- Page 48

### 3.11 Execution Time for Virtual Microscope: Config 3
- Page 50

### 3.12 Execution Time for Virtual Microscope: Config 4
- Page 52

### 3.13 Execution Time of ZBUF and ACTP: Small Dataset
- Page 54

### 3.14 Execution Time of ZBUF and ACTP: Large Dataset
- Page 55

### 3.15 Execution Time for ZBUF: Runs with the same packet
- Page 55

### 3.16 Execution Time for ACTP: Runs with the same packet
- Page 56

### 4.1 Execution Time-line of a Pipeline with Three Computation Stages and Two Communication Stages. $Stage_3$ is the bottleneck stage.
- Page 63

### 4.2 Execution Time-line of a 1st-Stage Bottleneck Pipeline under Fixed-Frequency Communication Pattern
- Page 64

### 4.3 Changing Size of the Packet Flowing Through the Pipeline Shown in Figure 4.2
- Page 64

### 4.4 Execution Time-line of a Pipeline With Bottleneck at Second or Later Stages: Fixed-Frequency Communication Pattern
- Page 66

### 4.5 Execution Time-line of a Pipeline With Bottleneck at Second or Later Stages: Fixed-Size Communication Pattern
- Page 68

### 4.6 Execution Time-line of a 1st Stage Bottleneck Pipeline under Fixed-Size Communication Pattern
- Page 69

### 4.7 T can be Represented as Summation of Two Simple Linear Functions
- Page 71

### 4.8 A Time-line Illustrating the Division of Computation and Communication Time
- Page 77

### 4.9 Runtime of KNN-1st under Fixed-Frequency Scheme
- Page 79

### 4.10 Runtime of Zbuffer Based ISO-Surface Rendering (ZBUF-2nd) under Fixed-Frequency Scheme
- Page 83

xvi
4.11 Runtime of ActivePixels Based ISO-Surface Rendering (ACTP) under Fixed-Frequency Scheme ........................................ 84

4.12 Runtime of KNN-2nd under Fixed-Size Scheme .................. 85

4.13 Runtime of Zbuffer Based ISO-Surface Rendering (ZBUF-1st) under Fixed-Size Scheme .................................................. 86

4.14 Runtime of ActivePixel Based ISO-Surface Rendering (ACTP) under Fixed-Size Scheme .................................................. 87

5.1 Parallel k-means clustering in Data Parallel Java (Part 1) .......... 91

5.2 Parallel k-means clustering in Data Parallel Java (Part 2) ........ 92

5.3 Outline of Z-buffer Based Iso-surface Extraction Code (Part 1) .... 94

5.4 Outline of Z-buffer Based Iso-surface Extraction Code (Part 2) .... 95

5.5 Computing Generated and Consumed Sets Using a Single Pass .... 102

5.6 Examples of Unpacking: Instance-wise (a) and Field-wise (b) .... 105

5.7 Examples of Unpacking: Complete Generated Code ............. 106

5.8 Results from Z-Buffer Based Isosurface Rendering, *small* dataset ... 112

5.9 Results from Z-Buffer Based Isosurface Rendering, *large* dataset ... 112

5.10 Results from Active Pixel Based Isosurface Rendering, *small* dataset . 113

5.11 Results from Active Pixel Based Isosurface Rendering, *large* dataset . 113

5.12 Results from k-nearest neighbors, *k = 3* .................................. 115

5.13 Results from k-nearest neighbors, *k = 200* .......................... 116

5.14 Results from Virtual Microscope, Small Query ....................... 117

5.15 Results from Virtual Microscope, Large Query ..................... 117

xvii
6.1 Illustration of the language construct extensions: Code segments from the Virtual Microscope application (Part 1) ........................................ 126
6.2 Illustration of the language construct extensions: Code segments from the Virtual Microscope application (Part 2) ................................. 127
6.3 Overall System Architecture .......................................................... 131
6.4 Algorithm to Determine the Relationship between Execution Time and Adaptation Variables ............................................................... 133
6.5 Example Code from K-means Clustering: Code Segment from run function and its related class ............................................................. 143
6.6 Example Code from Virtual Microscope: The is_sampled method of class VMPixel .............................................................. 144
6.7 Comparison of predicted and actual execution times, 1-1-1 pipeline configuration ................................................................. 144
6.8 Comparison of predicted and actual execution times, 2-2-1 pipeline configuration ................................................................. 145
6.9 Comparison of predicted and actual execution times, 4-4-1 pipeline configuration ................................................................. 145
6.10 Comparison of predicted and actual execution times, 1-1-1 pipeline configuration ................................................................. 146
6.11 Comparison of predicted and actual execution times, 2-2-1 pipeline configuration ................................................................. 146
6.12 Comparison of predicted and actual execution times, 4-4-1 pipeline configuration ................................................................. 147
6.13 Comparison of predicted and actual execution times, $k = 5$ ........ 147
6.14 Comparison of predicted and actual execution times, $k = 10$ .......... 148
6.15 Comparison of predicted and actual execution times, \( k = 10 \), minimum and median value of \( it \) for initial runs ............................. 148

6.16 Comparison of predicted and actual execution times, \( it = 10 \) ........ 149

6.17 Actual execution time for K-Means Clustering: Two Adaptation Parameters ................................................................. 149

6.18 Predicted execution time for K-Means Clustering: First Set of Runs 150

6.19 Predicted execution time for K-Means Clustering: Second Set of Runs 150
CHAPTER 1

INTRODUCTION

In recent years, scientists have developed numerous ways to collect and generate data. The size of datasets is growing dramatically. For example, the Large Hadron Collider (LHC) is generating 10 Gigabytes of colliding beam data per second. The long-duration satellite-based sensors can collect more than 200 MB data each day. The advanced DOE climate models continuously perform high-resolution simulations to produce tens of petabytes of output. These datasets are usually distributed across multiple sites. Several reasons may account for such storage pattern. First, the size of the datasets is too huge to be accommodated at a single location. Another reason is these datasets may be generated or collected by different groups of sources. For example, in a large-scale agent based simulation task Joint Semi-Automatic Force (JSAF), each agent simulates different civilian object, and records their status and behavior in a data file. In addition, with security or fault tolerance consideration, the datasets are sometime replicated. However, it is impractical to replicate these datasets at each user node, so they must be accessed remotely.

At the same time, processing such large-scale datasets plays an increasingly important role in many domains of scientific research as well as business decision making processes. This class of applications often involves access to multiple data repositories
or data collection instruments. In some cases, analysis of the data requires powerful computer systems that are available at locations other than the data storage sites, and the final results of the analysis needs to be presented at yet another location.

As one form of distributed architecture, the emerging Grid computing environment facilitates the proliferation and development of such applications, allowing the seamless connection of world-wide computers, databases and instruments. The geographically distributed, heterogeneous, dynamically changing resources can be readily integrated together to collaborate and provide dependable and consistent services to end users. Considerable effort has been devoted to study the large range of problems inspired by the nature of such environments. Ease of use, quality of services guarantee, security, resources reservation, performance monitoring, adaptation to changing environments, load balancing, coordinating and scheduling composing components are among the top issues for consideration.

Currently, extensive computer science research has been carried out to investigate all aspects of Grid computing, and numerous software support systems have been designed and implemented. Some of them aim to provide a comprehensive set of runtime services [67, 76, 59]. Others focus on more specific services, exemplified by NetSolve [41], AppLES [42] and Network Weather Service (NWS) [147].

1.1 The Problem

This large-scale distributed environment, in concert with all the available system support can, in principle, provide access to and utilization of diverse, high-performance resources for advances in both sciences and daily life of people. However, these supports and resources will be of little use if only trained experts or experienced
programmers can understand and make use of these systems. A more general and consistent development support is urgently needed to construct applications on top of such an environment. An ideal development model will allow programmers to focus on the already strenuous problem-solving issues, while hiding details unrelated to the applications. Unfortunately, such a model is currently unavailable. The broad goal of our work is to provide advanced middleware support to ease the developing task of data-intensive or data-driven applications, at the same time, to allow their efficient execution.

Applications processing large-scale datasets become increasingly important and popular in many domains. Examples arise from areas such as scientific data analysis, data mining, real-time visualization, image processing, etc. This class of applications often requires fast or even interactive response time. However, it is usually infeasible to perform all analysis at the site hosting the data. Similarly, networking and storage limitations make it impossible to download all data at a single site before processing. One viable way to implement these applications is to break the entire procedure into several stages, one of which executes on the site hosting or collecting data. There should also be one stage on the user’s local machine, responsible for gathering the final results, even presenting the results in a user friendly manner. Between these two ends, single or multiple stages might be necessary for application-specific processing.

A coarse-grained pipelined execution model provides a good solution for implementing such applications on the grid environment. Here, the processing associated with an application is carried out in several stages, which are executed on a pipeline of computing units. Each stage handles the intermediate results obtained from the previous one, and packs the current output for the following one. Typically, the first
stage in this pipeline is the unit where the input data is available, and the last stage is where the final results are to be viewed. To realize support for data-intensive applications with coarse-grained pipelined execution model, we have to deal with the following questions.

- How to program an application for execution in a staged fashion?

- How to deploy an application on the designated grid resources?

- How to achieve the best possible performance of an application?

- Taking into account the ever-changing resource availability in a Grid environment, how to adapt the applications so as to meet the user expectation, such as total runtime, memory usage, battery lifetime, etc.?

To answer the first two questions, application developers need to have some insights about the underlying execution environment, beyond the already complicated design and implementation issues associated with a particular application. Such information may include, but not limited to, where are the computing powers located, what are the available CPU cycles, how are these hosts connected, what are the possibly usable bandwidths on each connecting links. To tackle the last problem, programmers also want to worry about how much the variation will affect the application performance or resource consumption. Obviously, each of these tasks is challenging.
1.2 Issue 1: Support for Coarse-Grained Pipelined Execution Model

In our research agenda, the essential goal is to insulate the application designers and implementors from the underlying complexity of the distributed and heterogeneous execution environment without sacrificing efficiency. To this end, we developed different algorithms and models to perform program decomposition and performance tuning, and integrated these algorithms into the language and compiler support for exploiting pipelined parallelism.

Filter Decomposition for Supporting Coarse-Grained Pipelined Parallelism In Chapter 3, the filter decomposition problem is investigated and formalized. Three different algorithms are proposed. Algorithm MIN_ONETRIP is a dynamic programming algorithm, optimizing the one trip cost for a packet passing through the pipeline. Another dynamic programming algorithm, MIN_BOTTLENECK, minimizes the time spent on the bottleneck stage. Finally, MIN_TOTAL is an approximate greedy algorithm that tries to minimize the total execution time. Results show that the heuristic algorithms work quite well in practice, with the possible exception of MIN_ONETRIP when the number of packets became larger.

Packet Size Optimization for Supporting Coarse-Grained Pipelined Parallelism An important factor that impacts execution of applications with coarse-grained pipelined execution model is packet size, which defines the unit of data transfer from one pipeline stage to another. A very large packet size will prevent the pipelined parallelism from being exploited. On the other hand, a very small packet size could lead to high overheads because of communication latencies. Since the pipeline includes both communication and computation phases, the frequency
and/or volume of communication between two different phases can be varied. We have developed and validated an analytical model for choosing the right packet size for an application. Two communication models, fixed-frequency and fixed-size models, are studied, and a series of closed-form expressions are derived for both of them. Our experiments show that the choice of packet size makes a significant difference in the execution time, and the packet size suggested by the model results in the lowest or very close to lowest possible execution time. Further details will be introduced in Chapter 4.

Language and Compiler Support for Exploiting Coarse-Grained Pipelined Parallelism Our goal is to allow programmers to write a sequential code to specify the application-specific tasks, with a little language extensions, which can help expose both pipelined and data parallelism to the compiler. We have developed a compilation system by integrating the algorithms and models presented in Chapter 3 and 4. The system is responsible for selecting a set of candidate filter boundaries, determining the volume of communication required if a particular boundary is chosen, performing the decomposition, and generating code for execution on coarse-grained pipelined execution model. For this purpose, a one-pass algorithm is designed for determining the required communication between consecutive filters, a cost model is established to estimate the execution time for a given decomposition, and a dynamic programming algorithm is used for making the decomposition decisions. Detailed explanation and evaluation of the system will be presented in Chapter 5.
1.3 Issue 2: Support for Application Adaptation

Our previous work assumes a static and stable execution environment for applications, that is, once the resources are reserved, they will be exclusively used for a particular application. So same performance will be ensured during the entire execution of the application. In practice, with the dynamic nature of a grid environment, data analysis applications executing on it compete for communication bandwidth and computing power. Though resources such as CPU cycles and network bandwidth can be reserved, it is very hard, if not impossible, to guarantee the actual usage of these resources. Even worse, resource reservation mechanisms may not be supported by the underlying system. Hence, the ability of self-adaption is very desirable for a grid application. In particular, we have focused on applications for which the users have significant flexibility in the desired output. Meanwhile, there might be some constraints associated with the applications, for example, response time or resource consumption limitations. We integrated a small set of language extensions to allow the programmers specify adaptation parameters, whose values can be varied within a certain range. Our approach for adaptation adopts a hybrid methodology, which combines compile-time analysis with runtime feedback. A program analysis algorithm states the execution time of an application component as a function of the values of the adaptation parameters and other runtime constants. These constants are determined by initial runs of the application in the target environment. Once these constants are known, we know which adaptation parameters can be modified and by how much to achieve a given change in execution time, or to maintain the same execution time for a given change in available resources. Substantial amount of experiments have been carried out to assess our system, which shows the combined compiler-time/runtime
model can predict the execution times quite well, and therefore, support adaptation. Chapter 6 details the design and implementation issues.

Our work has been implemented and evaluated using DataCutter, which is an existing runtime system for supporting pipelined parallelism in a grid environment [29, 27]. Specifically, DataCutter supports a filter-stream model of execution. Typically, one filter executes one stage of the pipeline, using one or more input streams. The results of the processing are packed and sent as one or more output streams.

1.4 Contributions

Together, the language constructs and the compiler techniques we proposed provide a number of potential benefits. Above all, application developers can write sequential codes, using a small set of language extensions, to implement parallel programs. Moreover, with our system support, the application can run on the large-scale distributed environment under an ideal deployment pattern, while the decomposition decisions are made at compile time instead of development phase, taking into consideration the resources availability and capacity. Further, our system can help meet the programmer-specified performance requirement by adapting applications to underlying ever-changing execution environment. To summarize, our contributions are as follows.

1. A pipelined execution model for grid computation. The coarse-grained pipelined execution model provides a natural vehicle for executing large-scale data-intensive applications on the grid environment. We believe this model is suitable for closely-coupled applications as well as the grid workflow systems.
2. A new perspect at the language and compiler level to exploit coarse-grained pipelined parallelism. We integrate language design and compiler optimization techniques to support deployment of distributed data-intensive applications.

3. A set of algorithms which perform application decomposition to achieve high performance. We propose three algorithms to find the optimal partition for an application, with the goal to optimize its execution time over a pipeline of computing units.

4. An analytical model which formulates and tackles the granularity selection problem for a pipelined execution. The model accommodates both computation and communication phases of the pipeline, and rigorously describes the entire processing procedure. A series of equations are derived to express the optimal communicating packet size under various pipeline patterns.

5. A novel hybrid approach to support program adaptation. With very little language support, information gleaned at compilation stage combined with runtime feedback are used to adjust the application to meet the demanding performance constraints under the variation in resource availability.

1.5 Dissertation Roadmap

The rest of this dissertation is organized as follows. In Chapter 2, we briefly introduce the target class of applications, the background knowledge about the coarse-grained pipelined execution model we use for deploying applications, and the DataCutter runtime system on which our experiments are conducted. In Chapter 3, the
filter decomposition problem is investigated. After formalizing the problem, we propose three algorithms to optimize different portion of the entire execution cost of an application. An analytical model is developed for choosing the right packet size for an application in Chapter 4. Chapter 5 details the language and compiler support for exploiting coarse-grained pipelined parallelism. A hybrid methodology is proposed in Chapter 6 to support for adaptive applications, followed by the review of related efforts on supporting large-scale data-intensive applications in Chapter 7. Finally, Chapter 8 concludes the report by summarizing contributions of this dissertation and discussing the future research directions.
CHAPTER 2

BACKGROUND

In this chapter, we lay the groundwork for the rest of the dissertation by providing the description of our target data-driven applications, a more detailed introduction of the coarse-grained pipelined execution model with which we deploy applications, and the DataCutter runtime system on which we conducted a series of experiments.

2.1 Target Distributed Data-Intensive Applications

We are in an era identified by information explosion. Scientists have developed numerous ways to generate and collect information. Datasets with sizes measured in dozens or hundreds of petabytes are likely to be common within a decade. With the huge amount of data, the core challenges to be overcome are how to store and manage these data effectively, how to process them to extract useful information efficiently, and how to present the final results to the end users intuitively. Our focus is on a variety of data-driven applications, arising in domains like scientific data analysis, data mining, visualization, and image analysis, and spanning both scientific and commercial interests.

High Energy Physics: Tens to hundreds of terabytes of colliding beam data is generated per year by experiments at accelerator labs across the world. Updates
to these experiments plus the new experiments at the Large Hadron Collider (LHC), Laser Interferometer Gravitational-wave Observatory (LIGO), and Sloan Digital Sky Survey (SDSS) will grow these datasets easily from 100 Terabyte to 100 Petabyte scale over the next decade. The globally dispersed communities of hundreds of scientists need to access and analyze these datasets to explore the fundamental forces of nature and structure of universe: Compact Muon Solenoid (CMS) detector and a Toroidal LHC Apparatus (ATLAS) at CERN are to search for the finest origins of mass and probe matter; LIGO is to detect the gravitational waves of astronomical or cosmological sources; and SDSS will survey different characteristics of celestial objects [120, 18, 117, 8, 39].

**Space Science :** The Digital Sky project integrates the Digital Palomar Observatory Sky Survey (DPOSS), the Two-Micron All Sky Survey (2MASS), the NARO VLA Sky Survey (NVSS) and the VLA FIRST radio survey together to provide multi-wavelength digital library describing the real sky. The data from these surveys are expected to cover one billion objects about their locations and brightness, while the image data will contain several tens of terabytes. This facility makes it feasible for astronomers to undertake detailed studies about both discrete sources and extended structures [120, 1].

**Earth Science :** The Pathfinder Advanced Very High Resolution Radiometer (AVHRR) Land datasets consist of remotely-sensed data acquired from the long-duration satellite-based sensors, and are mainly designed for multispectral analysis of meteorologic, oceanographic, and hydrologic parameters. Usually, a daily dataset is more than 200 MB, and a typical analysis involves data for ten days to a year. These data allow scientists to achieve a comprehensive understanding of the global
environments by viewing the Earth’s condition as an integrated system [6]. The Enhanced Thematic Mapper Plus (ETM+) carried on board the Landsat 7 satellite of the U.S. Geological Survey (USGS) Earth Resources Observation System (EROS) provides image data from eight spectral bands, and the approximate image size is 300M bytes for a single scene of size 170 × 183 kilometers (106 × 115 miles). Since the first satellite launched in 1972, the instruments have acquired millions of images, which provide invaluable information to various applications in agriculture, geology, forestry, regional planning, education, mapping, and global change research [4].

Scientific Simulation: The advanced DOE SciDAC climate models continuously perform high-resolution simulations to produce tens of petabytes of output [2]. The hydrodynamics and chemical transport simulators are used to study and predict the reaction of chemicals in bay areas. Each simulator generates a grid of data points representing the current status of the simulated region at each time stamp. A normal output data file for a large grid at a single time step is several megabytes, and it will require thousands of time steps to simulate a particular scenario [?, 43, 107]. The space weather simulations can model how energy output of the Sun affects the Earth’s magnetosphere and ionosphere. In the process, a set of magnetohydrodynamics equations are to be solved in an interested region of space. A possible region is denoted by 50 × 48 × 64 points, and the storage required for eight plasma parameters as double precision floats for a single timestamp will exceed 10MB. One simulation could last for several days, and the generating results are in terabyte scale [60, 108, 112].

Life Science: The completion and publication of the working draft of human genome has announced terabytes of data to be available for analysis [3]. This may facilitate the comparison of a unidentified sequence to sequences already known, and
the prediction about the sequence based on knowledge of similar sequences. The genome data also help biologists to identify and characterize the arrangements as well as the regulation of various genes. Studies of protein structure and function via predication and identification are trying to describe the three-dimensional information contained in the protein sequence as well as the relationship between its function and its 3D structure, therefore generating enormous amount of data as the database grow larger daily [7]. For the protein-protein interaction, a docking procedure is usually performed, which accesses thousands of petabytes of data [5]. In terms of medical applications, one human embryo image is about several gigabytes, a three-dimensional brain scan can easily exceed half of a terabyte, while a more comprehensive radiology scan can produce dozens of petabytes of data [120, 10].

**Business Decision Making**: Risk analysis is a very critical task performed in every financial institute to predict how various market scenarios would affect the company’s investments. Thousands of calculations are necessary to analyze each scenario. The situation will become even worse when these analysis need to be done for a cross-nation corporation, where a wealth of factors originating from different resources are to be integrated and analyzed together. The so called virtual organizations comprise collections of institutions or entities sharing and analyzing information [68]. These virtual organizations can be expected to use a variety of business decision-support functionalities that have been commonly used in centralized organizations to predict how various market scenarios would affect the company’s investments.
2.2 Issues and Challenges

Processing and analyzing large volumes of data plays an increasingly important role in many domains of scientific research as well as business decision making processes. In many applications, because of computational and storage requirements, and to ensure fault tolerance and high availability, datasets are distributed on storage systems across a wide area network, while the final results are desired at the user’s desktop. There are two obvious solutions for executing such applications: 1) downloading all data at the user’s local machine, or 2) performing all computations at sites hosting data repositories. However, neither of these two options is feasible in most situations. Sufficient storage space and/or network bandwidth is not likely to be available to download all data. At the same time, a node hosting a data repository may not be able to offer sufficient computing power for executing the entire application for every user. Therefore, a natural option is to use a pipeline of computing resources, where the site hosting the data repository is the first stage and the user’s local machine where the results are required is the final stage. Typically, one or more clusters and/or SMP machines serve as the intermediate stage(s).

Mapping of our target class of applications to the pipelined model is facilitated by an important observation. Our study of a variety of scientific and commercial data intensive applications shows that generalized reduction operations are very common in the processing structure. This observation applies across a large number of scientific data intensive applications [48, 107, 121, 110, 116, 140, 113], data mining algorithms [94] including association mining, clustering, and decision tree construction, OLAP applications involving algebraic and distributed aggregations [75], and
key visualization algorithms such as the ones for isosurface rendering [104]. Processing for generalized reductions consist of three main steps: (1) Retrieving data items of interest, (2) Applying application-specific transformation operations on the retrieved input items, and, (3) Mapping the input items to output items and aggregating, in some application specific way, all the input items that map to the same output data item. Most importantly, aggregation operations involve commutative and associative operations, i.e., the correctness of the output data values does not depend on the order input data items are aggregated. For simplicity, we can refer to the first two steps as local processing, and the third step as global combination.

An important implication for the common structure described above is that different steps involved in local processing can be performed independently on different sections of data. This enables exploitation of pipelined parallelism, as well as shared memory or distributed memory parallelism at an intermediate stage of the pipeline.

2.3 Coarse-Grained Pipelined Parallel Execution Model

By definition, pipelining is a technique used in advanced microprocessors where the microprocessor begins executing a second instruction before the first has been completed. While the coarse-grained pipelined execution, in contrast to fine-grained or instruction level parallelism model, is associated with the program or procedure level parallelism. In this model, the processing associated with an application is partitioned into several stages. These stages are executed on a pipeline of computing units, with all successive units working simultaneously, and each stage handling the intermediate results obtained from the previous stage.
This model provides a natural vehicle for executing data-driven applications in scenarios where the data is available on a repository or a data collection site on the internet, and the final results are required on a user’s desktop. It is usually not possible to perform all analysis at the site hosting such a shared data repository or a data collection instrument. Similarly, networking and storage limitations make it impossible to download all data at a single site before processing. Thus, the application needs to be broken into a process or stage that executes on the site hosting or collecting the data, one or more stages that executes on clusters or SMP machines, and a final stage that executes on the user’s local machine.

Figure 2.1: K-nearest Neighbor Search Algorithm Implemented Using Coarse-Grained Pipelined Parallel Execution Model.

Take the K-nearest neighbor search algorithm as an example. The problem is quite simple: given a 3-dimension range $R = (x_1, y_1, z_1), (x_2, y_2, z_2)$, and a point $w = (a, b, c)$, where $x_1$, $y_1$, $z_1$ are the coordinates of the upper left corner of $R$ and $x_2$, $y_2$, $z_2$ are the coordinates of the lower right corner of it, we want to find the nearest $K$ neighbors of $w$ within range $R$, where $K$ is a parameter given to the algorithm. The basic idea for finding k-nearest neighbors in a sequential environment is as follows. For each point $X = (x, y, z)$ in the file, if $X$ is in the range $R$, we compute the distance from $X$ to $w$ by $\text{dist} = \sqrt{(x - a)^2 + (y - b)^2 + (z - c)^2}$, and find the points with the $k$-lowest values of distance. To deploy this application on a coarse-grained pipelined
execution model, the problem is decomposed into two local processing stages: one for getting all points in the range $R$, referred to as \texttt{range\_query}, and the other for selecting the k-nearest neighbors.

### 2.4 DataCutter: A Middleware for Data Intensive Computations in a Grid Environment

DataCutter [26, 29] has been developed at the University of Maryland and the Ohio State University. It is a middleware framework for developing data intensive applications in a distributed environment. It targets distributed, heterogeneous environments by allowing decomposition of application-specific data processing operations into a set of interacting processes.

In DataCutter, data intensive applications are represented as a set of filters. A \textit{filter} is a user-defined object with methods to carry out application-specific processing on data. The interface for filters consists of an initialization function (\texttt{init}), a processing function (\texttt{process}), and a finalization function (\texttt{finalize}). Data exchange between any two filters is described via \textit{streams}. A \textit{stream} is a communication abstraction that allows fixed size untyped data buffer to be transported from one filter to another. All transfers to and from streams are through a provided buffer abstraction. A buffer represents a contiguous memory region containing useful data. Streams transfer data in fixed size buffers.

For the k-nearest search algorithm given in Section 2.3, in the filter-stream approach, the problem is decomposed into two filters for local reduction: one for getting all points in the range $R$, referred to as \texttt{range\_query}, and the other for selecting the k-nearest neighbors, called \texttt{select}. Since data files are distributed at different sites, one more filter is needed for performing global reduction, referred to as \texttt{combine}.
Communication between filters are implemented via streams. Filter \textit{range\_query} outputs all the points of the local data file which lies in \textit{R} to filter \textit{select} via stream \textit{R-S}, while filter \textit{select} communicate with \textit{combine} through \textit{S-C} stream by sending the local nearest \(k\) neighbors to the global reduction filter \textit{combine}. Finally, \textit{combine} will decide the globally nearest \(k\) neighbors. Figure 2.2 gives the decomposition.

Filter operations progress as a sequence of cycles, with each cycle handling a single application-defined \textit{unit-of-work}. A work cycle starts when the filtering service calls the filter \textit{init} function, which is where any required resources such as memory or disk scratch space are pre-allocated. Next the \textit{process} function is called to continually read data arriving on the input streams in buffers from the sending filters. The \textit{finalize} function is called after all processing is finished for the current unit-of-work, to allow release of allocated resources such as scratch space. The process of manually restructuring an application using the filter-stream programming model is referred to as \textit{decomposing} the application. When the application is decomposed into a set of filters, the runtime mapping of filters onto various hosts in a grid environment is called \textit{placement}. DataCutter provides means for specifying \textit{filters, streams} and \textit{placement}, as illustrated in Figure 2.3. In DataCutter, filters are location-independent, because stream \textit{names} are used to specify filter to filter connectivity rather than endpoint.
location on a specific host. This allows the placement of filters on hosts controlled by the runtime system so as to minimize processing, network and data copying overheads.

```cpp
class ApplicationFilter : public DC_Filter_Base_t {
    public:
        int init(int argc, char *argv[]) { ... };
        int process(stream_t st[]) { ... };
        int finalize(void) { ... };
    }

    DC_FilterLayout_t layout;
    layout.setName("k-nearest-neighbor");
    layout.Add("range_query", NULL, "R-S");
    layout.Add("select", "R-S", "S-C");
    layout.Add("combine", "S-C", NULL);

    DC_Placement_t placement;
    placement.Add("range_query", "maia");
    placement.Add("select", "oin");
    placement.Add("combine", "taygeta");
```

Figure 2.3: Specifying filters, layout and placement with DataCutter Interfaces.

DataCutter also provides support for transparent copies. Transparent copies allow a finer level of parallelism via multiple copies of a single filter. The filter runtime system maintains the illusion of a single logical point-to-point stream for communication between a logical producer filter and a logical consumer filter. When the logical producer or logical consumer is transparently copied, the system decides for each producer which copy to send a stream buffer to. Schemes like round-robin allocation are used to achieve load balancing.
We believe that DataCutter is a well fit for the coarse-grained pipelined execution model. In a grid environment, data is available on geographically distributed sites. Further, the processing needs to be performed in a heterogeneous environment. The ability to decompose the computation into interacting but location independent filters is important for taking full advantage of grid resources. Thus, we conclude that DataCutter’s filter-stream based programming model is especially suitable for distributed data mining algorithms.
CHAPTER 3

FILTER DECOMPOSITION FOR SUPPORTING COARSE-GRAINED PIPELINED PARALLELISM

3.1 Introduction

In this chapter, we focus on an important issue arising in any runtime or compilation system that supports coarse-grained pipelined parallel execution, which is filter decomposition. Simply put, it is the problem of partitioning a given application onto a set of stages, so as to achieve the best performance. More technically, we have an application divided into a sequence of atomic filters. We need to map these atomic filters into filters, such that the number of filters equals the number of stages in the pipeline. The total execution time for pipelined execution is roughly the sum of the time taken by one packet to traverse the pipeline, and the time taken by all packets at the bottleneck stage. This complicates the problem of performing filter decomposition to achieve the minimum execution time.

We consider this problem in the context of a runtime system or a profile-driven compilation system. An example of such a system is presented in Chapter 5. In such an environment, the amount of computation in each atomic filter and the volume of communication between consecutive atomic filters are known. Similarly, the
computing power for each stage and the available communication bandwidth between consecutive stages are known. Given such information, we can predict the execution time for a given decomposition. However, there are an exponential number of possible decomposition choices. Therefore, enumerating all choices and finding an optimal one is extremely expensive.

We propose three polynomial time algorithms for this problem. Algorithm MIN_ONETRIP is a dynamic programming algorithm that optimizes the one trip cost for a packet passing through the pipeline. Algorithm MIN_BOTTLENECK is also a dynamic programming algorithm, which minimizes the time spent on the bottleneck stage. Finally, MIN_TOTAL is an approximate greedy algorithm which tries to minimize the total execution time.

We have evaluated these three algorithms using three applications and many different configurations. Our results show that our heuristic algorithms work quite well in practice, with the possible exception of MIN_ONETRIP when the number of packets is large. However, the relative performance of the algorithms is not always what we would expect, because of the certain limitations in how we model the problem.

Moreover, one of the three proposed algorithms, the MIN_ONETRIP algorithm, is integrated into a compilation system and tested with real applications. The details of the system as well as the evaluation are presented in Chapter 5.

The rest of the chapter is organized as follows. The filter decomposition problem is formally defined in Section 3.2, followed by our proposed three algorithms in Section 3.3. Section 3.4 focuses on the experimental evaluation of our algorithms. We conclude in Section 3.5.
3.2 Filter Decomposition Problem

To execute an application in the coarse-grained pipelined fashion, an important initial step is to decompose the application into filters. The number of these filters needs to be equal to the number of stages in the environment available for execution. In performing the decomposition, our goal is to minimize the overall execution time. This execution time depends on the amount of computation in each filter and the volume of communication between consecutive filters.

The input to our filter decomposition algorithms is a program that is divided into a sequence of atomic filters. Typically, the number of such atomic filters is much larger than the number of stages in the execution environment, or the number of filters we need to generate. In Chapter 5, we will describe how our compiler could obtain the sequence of atomic filters from a data parallel program. Alternatively, an application developer may also specify such atomic filters. We are assuming that filter decomposition decisions are made at runtime or in a profile-driven compiler. Therefore, in such a scenario, the amount of computation in each atomic filter and the volume of communication between consecutive atomic filters are known. Similarly, the computing power for each stage and the available communication bandwidth between consecutive stages are known.

In this section, we define the filter decomposition problem formally. Initially, we state the cost model we use for comparing different filter decompositions. The algorithms for filter decomposition are presented in the next section.
3.2.1 Cost Model

Consider a pipeline of $m$ computing units, denoted by $C_1, \ldots, C_m$, and let the connection between units $C_i$ and $C_{i+1}$ be denoted by $L_i$. In total, there are $2 \times m - 1$ stages in the pipeline. Suppose $N$ packets are to be processed on the pipeline. We want to calculate the total execution time.

For simplicity, we assume that each packet is of the same size, the computing power at a given computation stage remains unchanged during the execution, and similarly, bandwidth offered by a given communication stage remains unchanged. Further, we assume that processing time is independent of the data. Thus, each packet takes the same amount of time on a given computing or communication stage.

The expression for execution time is based upon the notion of bottleneck stage. This is the stage whose processing rate is the slowest, and therefore, is always busy since the first packet is received, and before the last packet exits. We denote the bottleneck stage as $b^{th}$ stage in the pipeline.

The total execution time comprises two components, the time taken by one packet to reach from the start of the pipeline to the end, and the time spent by all but one packet at the bottleneck. Let the time spent by one packet at $stage_i$ be denoted by $T_i$. Then, the total execution time is

$$T = \sum_{i=1}^{2m-1} T_i + (N - 1) \times T_b$$

(3.1)

Figure 3.1 shows the execution time-line of a pipeline consisting of three computing stages and two communication stages.
Figure 3.1: Execution Time-line of a Pipeline with Three Computation Stages and Two Communication Stages. Stage\(_3\) is the bottleneck stage.

### 3.2.2 Filter Decomposition Problem

Let the application comprise \(n\) atomic filters, \(f_1, \ldots, f_n\). Then, we can denote the application as a two-tuple

\[
Appl = (\text{Comp}, \text{Comm}),
\]

where,

- \(\text{Comp}\) is a vector characterizing the computation workload of each atomic filter.
  \[
  \text{Comp} = (\text{Comp}_f_1, \ldots, \text{Comp}_f_n).
  \]

  The term \(\text{Comp}_f_i\) represents the computation workload of the \(i^{th}\) atomic filter \(f_i\).

- Similarly, \(\text{Comm}\) is a vector describing the communication amount between two consecutive atomic filters.
  \[
  \text{Comm} = (\text{Comm}_f_1, \ldots, \text{Comm}_f_n).
  \]
The term $Comp_{f_i}$ denotes the communication volume from the atomic filter $f_i$ to $f_{i+1}$. Note that the term $Comm_{f_n}$ represents the amount of data to be transferred by the atomic filter $f_n$, if $f_n$ is not placed on the last stage of the pipeline.

As we stated previously, we have $m$ computing units in a pipeline $C_1, \ldots, C_m$, which are connected by $m - 1$ communication links $L_1, \ldots, L_{m-1}$. For the computing unit $C_i$, the computing power is denoted by $P(C_i)$, and for the communication link $L_i$, the available network bandwidth is denoted by $B(L_i)$. Note that some of the computing units in our pipeline can be parallel machines, like a cluster or an SMP machine. In our description, such a possibility is captured by associating a higher computing power with the unit.

Now, we want to choose a placement for these $n$ atomic filters over the $m$ computing units, such that the total execution time for the pipeline is minimal. Let $\varphi(f_1, \ldots, f_n)$ be a placement, then

$$
\varphi(f_1, \ldots, f_n) = (F_1, \ldots, F_m)
$$

where

$$
F_i = f_{i_1, i_{i_1+1}, \ldots, i_k}
$$

i.e., atomic filters $f_{i_1}, f_{i_1+1}, \ldots, f_{i_k}$ are merged together, and placed on computing unit $C_i$, for $1 \leq i_1, i_k \leq n$. Thus, by Equation 3.1, the total execution time for placement $\varphi(f_1, \ldots, f_n)$ can be expressed as

$$
T_{\varphi(f_1, \ldots, f_n)} = \sum_{i=1}^{2m-1} T_i + (N - 1) \times T_b
$$

$$
= \sum_{i=1}^{m} Cost_{comp}(P(C_i), Comp_{F_i}) +
$$

27
\[ 
\sum_{i=1}^{m-1} \text{Cost}_{\text{comm}}(B(L_i), \text{Comm}_{\mathcal{F}_i}) + 
(N - 1) \times T_b 
\] 

\[ (3.2) \]

where,

\[ T_b = \max \{ \max_{i=1,\ldots,m} \text{Cost}_{\text{comp}}(P(C_i), \text{Comp}_{\mathcal{F}_i}), \]

\[ \max_{i=1,\ldots,m-1} \text{Cost}_{\text{comm}}(B(L_i), \text{Comm}_{\mathcal{F}_i}) \}. \]

\[ (3.3) \]

Here, the function \( \text{Cost}_{\text{comp}} \) takes the power of a computing unit and the computation task of a filter as inputs, and returns the time of execution. The function \( \text{Cost}_{\text{comm}} \) takes bandwidth of a communication link and the communication volume as parameters, and outputs the time of communication. The term \( T_b \) represents the execution time of the bottleneck stage, which is the most time-consuming stage among all the computation and communication stages.

Our goal is to find a placement \( \overline{\varphi}(f_1, \ldots, f_n) \), such that \( T_{\overline{\varphi}(f_1, \ldots, f_n)} \) is, or is very close to, the minimal execution time.

### 3.3 Algorithms for Filter Decomposition

This section presents several algorithms for the filter decomposition problem, as defined in the previous section.

Recall that we need to group \( n \) atomic filters into \( m \) stages. Alternatively, the problem can be viewed as of inserting \( m - 1 \) boundaries between \( n - 1 \) candidate boundaries. An exhaustive search will involve considering \( C_{m-1}^{n+m-1} \) placements. This term is exponential in the value of \( m \).

To avoid such an exhaustive search, we present three algorithms. These three algorithms are:
• MIN_ONETRIP is a dynamic programming algorithm that minimizes the time taken by one packet to traverse the pipeline. Therefore, it assumes that the time spent on the bottleneck stage is not dominant. The time complexity for this algorithm is $O(nm)$.

• MIN_BOTTLENECK is also a dynamic programming algorithm that minimizes the time spent by all packets at the bottleneck stage. Therefore, it assumes that the time spent by one packet to traverse the pipeline is not dominant. The time complexity for this algorithm is $O(n^2m)$.

• MIN_TOTAL is a heuristic greedy algorithm that minimizes the overall execution time, with time complexity is $O(nm)$.

The next three subsections present the above three algorithms. Finally, in Subsection 3.3.4, we compare these algorithms using an example.

3.3.1 MIN_ONETRIP Algorithm

The focus of this algorithm is on optimizing the single trip cost, that is, the cost for one packet passing through the entire pipeline. In the context of Equation 3.2, we optimize the sum of the first two terms. We can formulate the problem as following:

**Problem:** We are given an application

$$Appl = (\text{Comp}, \text{Comm}),$$

and a pipeline of $m$ computing resources, which are connected by $m-1$ communication links. We want to find a placement $\pi(f_1, \ldots, f_n) = (\mathcal{F}_1, \ldots, \mathcal{F}_m)$, such that $\sum_{i=1}^{2m-1} T_i$ is optimal, where

$$T_i = Cost_{comp}(P(C_i), \text{Comp}_{\mathcal{F}_i})$$
MIN_ONETRIP {
Inputs:
An application Appl, Appl = (Comp, Comm)
A sequence of $m$ computing units $C_1, \ldots, C_m$ with computing powers $P(C_1), \ldots, P(C_m)$
A sequence of $m - 1$ network links $L_1, \ldots, L_{m-1}$ with bandwidths $B(L_1), \ldots, B(L_{m-1})$
Goal:
$E[n, m]$
Array $E[0 \cdots n, 0 \cdots m]$;

{* init *}
for $j \leftarrow 0$ to $m$ { $E[0, j] = 0$; }
for $i \leftarrow 0$ to $n$ { $E[i, 0] = \infty$; }
$E[1, 1] = \text{Cost}_\text{comp}(P(C_1), \text{Comp}_{f_1})$;
$Sol[1, 1] = 2$;

{* compute $E[1,j]$, $j = 2, \ldots, m$ *}
for $j \leftarrow 2$ to $m$
    $E[1, j] = \min \left\{ E[1, j-1] + \text{Cost}_\text{comm}(B(L_{j-1}), \text{Comm}_{f_1}),
                    E[0, j] + \text{Cost}_\text{comp}(P(C_j), \text{Comp}_{f_1}) \right\}$
    $Sol[1, j] = 1$;

{* compute $E[i,j]$, $i = 2, \ldots, n$, $j = 1, \ldots, m$ *}
for $i \leftarrow 2$ to $n$
    for $j \leftarrow 1$ to $m$
        $E[i, j] = \min \left\{ E[i, j-1] + \text{Cost}_\text{comm}(B(L_{j-1}), \text{Comm}_{f_1}),
                      E[i-1, j] + \text{Cost}_\text{comp}(P(C_j), \text{Comp}_{f_1}) \right\}$
        $Sol[i, j] = \text{which}_{\min} \left\{ E[i, j-1] + \text{Cost}_\text{comm}(B(L_{j-1}), \text{Comm}_{f_1}),
                                   E[i-1, j] + \text{Cost}_\text{comp}(P(C_j), \text{Comp}_{f_1}) \right\}$
    }

Figure 3.2: MIN_ONETRIP Algorithm for Filter Decomposition

for a computation stage, or

$$T_i = \text{Cost}_\text{comm}(B(L_i), \text{Comm}_\pi)$$

for a communication stage.
We present a dynamic programming algorithm which can minimize this cost. The algorithm is based on the following observation. To get the final results on the last computing unit $C_m$, we can either put the last atomic filter $f_n$ on it, or we can finish all the computations in the first $m - 1$ computing units and then transmit the final results to $C_m$. Similarly, after determining the placement of the filter $f_n$, say on $C_m$, we can either put filter $f_{n-1}$ on $C_m$, or we can finish all the work from $f_1$ to $f_{n-1}$ before the computation reaches $C_m$, and then forward the results of $f_{n-1}$ to $C_m$.

Let $T[i, j]$ denote the minimum cost of doing computations up to $f_i$ on computing units $C_1, \ldots, C_j$, while the results of $f_i$ are on $C_j$. Thus, the lowest cost of completing all $n$ filters on $m$ computing units would be $T[n, m]$.

We can define $T[i, j]$ recursively as follows. The atomic filter $f_i$ can either be placed on $C_j$ while all computations up to $f_{i-1}$ are completed before it, or it can be finished on previous $j - 1$ computing units and the results are forwarded to $C_j$. So, we have

$$T[i, j] = \min \left\{ \begin{array}{l} T[i - 1, j] + \text{Cost}_{\text{comp}}(P(C_j), \text{Comp}_i) \\ T[i, j - 1] + \text{Cost}_{\text{comm}}(B(L_{j-1}), \text{Comm}_{f_i}) \end{array} \right.$$ 

Our algorithm is outlined in Figure 3.2. In order to calculate $T[n, m]$, the algorithm needs to fill in all cells in the $(n + 1) \times (m + 1)$ matrix $E$. Since it takes $O(1)$ time to compute each cell, the total execution time of the algorithm is $O(mn)$.

Even though we use a $(n + 1) \times (m + 1)$ matrix to record $E$ in the algorithm, we only need to consider the values in $E[i - 1, j]$ and $E[i, j - 1]$ for computing $E[i, j]$. Also, we can write back $E[i, j]$ in the cell used by $E[i - 1, j]$, after $E[i, j]$ is computed. Thus, the algorithm only has a space complexity of $O(m)$.

To get the placement leading to the lowest cost, we use an array $\text{Sol}$, as shown in Figure 3.2. The element $\text{Sol}[i, j]$ notes which one of the two terms, $E[i - 1, j]$ or
GetPlacement(Sol)
local Array map;
{start from Sol[n,m], fill the array maps}
while (! (the first row or first column of Sol))
    if Sol[i,j] == 2
        set current element of map to i;
        i --;
    else
        set current element of map to ":;
        j --;
output (the inverse of array map);

Figure 3.3: GetPlacement Procedure for Outputting the Placement from MIN_ONETRIP Algorithm

$E[i, j - 1]$, is chosen when computing the value of $E[i, j]$. If $E[i, j - 1]$ is chosen, $Sol[i, j]$ is set to 2, which means atomic filter $f_i$ and $f_{i+1}$ will not be placed on the same computing unit. On the other hand, if $E[i - 1, j]$ is chosen, $Sol[i, j]$ is set to 1. Based on the information stored in array Sol, we can extract the optimal placement. This is described by the psuedo-code in Figure 3.3. The procedure utilizes an extra array map. It starts from Sol[n,m], and sets values for elements in map one by one. Here, ":" denotes a separation between atomic filters $f_i$ and $f_{i+1}$.

We use the following example to illustrate the algorithm.

**Example 1**: We are given an application AppE with six atomic filters,

$$AppE = (<200, 500, 800, 400, 200, 300>,$$
$$<100, 500, 1500, 200, 200, 500>)$$

32
and a pipeline of three computing resources, which are connected by two communication links. It is required that $f_1$ must be placed on $C_1$, since we always assume the pipeline starts from the site hosting datasets. Here the $\text{Comp}$ and $\text{Comm}$ are the execution time of atomic filters on a particular machine $C$, or the data transmission time on a particular link $L$, in $\mu s$. Then, the power of computing units and the bandwidth of communication links on the pipeline are specified in terms of machine $C$ and link $L$. Suppose the power of the computing units, relative to $C$, are 2, 3, and 1, respectively. And the bandwidth of the connection links are 1 and 0.6, with respect to $L$.

Following the MIN_ONETRIP algorithm in Figure 3.2, suppose we already have the matrix $E$, such that,

$$E = \begin{pmatrix}
\infty & 0 & 0 & 0 \\
\infty & 100 & 200 & 366.667 \\
\infty & 350 & \ddots \\
\vdots & \vdots & & \ddots \\
\end{pmatrix}$$

We want to calculate $E[2,2]$, that is the minimum cost for executing atomic filters $f_1$ and $f_2$ over computing units $C_1$ and $C_2$. For this purpose, we need to consider two conditions: 1) both $f_1$ and $f_2$ are on $C_1$; and 2) $f_1$ on $C_1$, $f_2$ on $C_2$. The latter case is covered by $E[1,2]$, and the first is by $E[2,1]$. So,

$$E[2,2] = \min\{350 + 500/1, 200 + 500/3\} = 366.667\mu s$$

After executing the algorithm, we also have the $\text{Sol}$ array as follows.

$$\text{Sol} = \begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & 2 & 1 & 1 \\
0 & 2 & 2 & 2 \\
0 & 2 & 2 & 2 \\
0 & 2 & 2 & 1 \\
0 & 2 & 2 & 1 \\
0 & 2 & 2 & 2 \\
\end{pmatrix}$$
From this array, we get the placement

$$\overline{\mathcal{P}}(f_1, \ldots, f_n) = (f_1, f_{2-5}, f_6).$$

### 3.3.2 MIN_BOTTLENECK Algorithm

This algorithm minimizes the time spent by all the packets at the bottleneck stage. Formally, we can describe the problem as follows.

**Problem:** We are given an application

$$\text{App} = (\text{Comp}, \text{Comm}),$$

and a pipeline of $m$ computing resources, which are connected by $m - 1$ communication links. We want to find a placement $\overline{\mathcal{P}}(f_1, \ldots, f_n) = (\mathcal{F}_1, \ldots, \mathcal{F}_m)$, such that $T_b$ is optimal, where $T_b$ is defined in Equation 3.3.

MIN_BOTTLENECK is also a dynamic programming algorithm which minimizes the time spent at the bottleneck stage. To see the approach, let us consider the placement of the final atomic filter, $f_n$. We have the following two options:

**Option 1:** The filters $f_1, \ldots, f_n$ are all placed on computing units $C_1$ through $C_{m-1}$, then the final results need to be transferred to $C_m$.

**Option 2:** The filter $f_n$ is placed on the last computing unit of the pipeline, $C_m$.

To find a placement which gives the minimum cost of the $i^{th}$ stage, with Option 1, we only need to compare the communication cost on the last link with the bottleneck stage cost of placing $f_1, \ldots, f_n$ over $C_1, \ldots, C_{m-1}$. However, more attention should be paid when dealing with the second option. When $f_n$ is assigned to $C_m$, it is possible
that no other filter is on \( C_m \). It is also possible that before assigning \( f_n \), atomic filter \( f_{n-1} \) is already on \( C_m \). Still, it is possible that atomic filters \( f_{n-2} \) and \( f_{n-1} \) are on \( C_m \) already. Following this pattern, we need to consider \( n - 1 \) possibilities.

Let \( N[i, j] \) denote the minimum cost of the \( b^{th} \) stage for computing atomic filters \( f_1, \ldots, f_i \) over computation pipeline \( C_1, \ldots, C_j \), with the results of \( f_i \) on \( C_j \). Hence, the lowest cost of the bottleneck stage for completing the application \( Appl \) will be \( N[n, m] \).
MIN_BOTTLENECK {
Inputs:
An application Appl, Appl = (Comp, Comm)
A sequence of m computing units C_1, ..., C_m with computing powers P(C_1), ..., P(C_m)
A sequence of m − 1 network links L_1, ..., L_{m−1} with bandwidths B(L_1), ..., B(L_{m−1})
Goal:
H[n, m − 1]

Array H[0 ··· n, 0 ··· m − 1];

{* init *}
for j ← 0 to m − 1 {  H[0, j] = 0;  }

{* compute H[i,0], i = 1, ..., n *}
for i ← 1 to n {  H[i,0] = Cost\_comp(P(C_1), ∑_{k=1}^{i} Comp_{f_k});  }

{* compute H[i,j], i = 1, ..., n, j = 1, ..., m *}
for j ← 1 to m − 1
  for i ← 1 to n
    H[i, j] = \min\left\{\max\{H[i, j−1], Cost\_comm(B(L_{j−1}), Comm_{f_i})\},\right.
      \max_{k=1,\ldots,i−1}\{H[i−k, j−1], Cost\_comm(B(L_{j−1}), Comm_{f_{i−k}}),\}
      \sum_{i=k+1}^{i} Cost\_comm(P(C_{j+1}), Comp_{f_i})\left.\right\}

Sol[i, j] = which\_min \left\{\max\{H[i, j−1], Cost\_comm(B(L_{j−1}), Comm_{f_i})\},\right.
      \max_{k=1,\ldots,i−1}\{H[i−k, j−1], Cost\_comm(B(L_{j−1}), Comm_{f_{i−k}}),\}
      \sum_{i=k+1}^{i} Cost\_comm(P(C_{j+1}), Comp_{f_i})\left.\right\}

Figure 3.5: MIN_BOTTLENECK Algorithm for Filter Decomposition

Now, we can recursively define N[i,j] as follows. The atomic filter f_i could be computed before the computation reaches C_j, with the resulting data being forwarded to C_j. The other choices include f_i being placed on C_j alone, f_i being placed together
with $f_{i-1}$ on $C_j$, $f_i$ being placed together with $f_{i-1}$ and $f_{i-2}$ on $C_j$, and so on. Formally, we can state this as a mathematical expression, as shown in Figure 3.4.

The MIN_BOTTLENECK algorithm is presented in Figure 3.5. In order to compute $N[n, m]$, an array $H$ of size $(n + 1) \times m$ is used. As can be seen, to calculate $H[i, j]$, we always need to do $i$ comparisons. Since each comparison can be done in $O(1)$ time, it takes $O(n)$ time to compute one element in the array $H$. Therefore, the total execution time of this algorithm is $O(n^2m)$.

Although a $(n + 1) \times m$ array was used for presenting the algorithm, in the actual implementation, we only need a $2 \times (n + 1)$ array to store $H$. When computing $H[i, j]$, only $H[1, j - 1], \ldots, H[i, j - 1]$ are required, so after filling up the $j^{th}$ column, we can reuse the space taken by the $(j - 1)^{th}$ column to store the $(j + 1)^{th}$ column. Thus, the space complexity of the algorithm is $O(n)$.

For extracting the optimal placement, we also use an auxiliary array $Sol$. The element $Sol[i, j]$ records the largest numbered filter on unit $C_j$. That is, if we let $k = Sol[i, j]$, then filters $f_{Sol[k,j-1]+1}, \ldots, f_k$ are placed together on computing node $C_j$. The pseudo-code is shown in Figure 3.6.

We again use Example 1 to illustrate this algorithm. Suppose, the current status of matrix $H$ is as follows.

\[
H = \begin{pmatrix}
0 & 0 & 0 \\
100 & 100 & \cdots \\
350 & \cdots & \cdots \\
750 & \cdots & \cdots \\
950 & \cdots & \cdots \\
1050 & \cdots & \cdots \\
1200 & \cdots & \cdots \\
\end{pmatrix}
\]

We want to compute the element $H[2, 2]$, which will be the minimum bottleneck stage cost of running atomic filters up to $f_2$ on computing resources $C_1$ and $C_2$. The
**GetPlacementB(Sol)**

```plaintext
local Array map;
 (*start from Sol[n, m - 1], fill the map array*)
l ← n;
for (j ← 1 to m)
    for (i ← l downto Sol[l, m - j] + 1)
        set current element of map to i;
        set current element of map to "|";
l ← Sol[l, m - j];
output (the inverse of array map);
```

Figure 3.6: GetPlacementB Procedure for Outputting the Placement from the MIN_BOTTLENECK Algorithm

available options for doing this include: 1) $f_1$ and $f_2$ are both on $C_1$, and 2) $f_1$ is on $C_1$ and $f_2$ is on $C_2$. This gives us

$$H[2, 2] = \min \left\{ \max\left\{350, 500/1\right\}, \max\{100, 100/1, 500/3\} \right\} = 166.667\mu s$$

After computing the entire matrix $H$, we also get the elements in the array $Sol$.

$$Sol = \begin{pmatrix}
0 & 0 & 0 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 2 & 0 \\
0 & 2 & 0 \\
0 & 2 & 4
\end{pmatrix}$$

Using GetPlacementB procedure, we reach the placement

$$\overline{v}(f_1, \ldots, f_n) = (f_{1-2}, f_{3-4}, f_{5-6}).$$
MIN_TOTAL 

Inputs:
An application Appl, Appl = (Comp, Comm)
A sequence of m computing units C₁, ..., Cₘ with computing powers P(C₁), ..., P(Cₘ)
A sequence of m – 1 network links L₁, ..., Lₘ₋₁ with bandwidths B(L₁), ..., B(Lₘ₋₁)

Current_pos = 1
For i = 1 to m – 1 do {
  /* Place the filter boundary i */
  min_cost(i) = Infinite
  Aggregate_Power(i) = \sum_{k=i+1}^{m} P(C_k)
  For j = Current_pos to n do {
    /* Evaluate the cost of placing boundary i at candidate j */
    Exec_time(i, j) = Evaluate(j, P(Cᵢ), B(Lᵢ), Aggregate_Power(i))
    If (Exec_time(i, j) < min_cost(i)) {
      min_cost(i) = Exec_time(i, j)
      min_pos(i) = j
    }
  }
  Place the boundary i at the candidate min_pos(i)
  Current_pos = min_pos(i)
}

Figure 3.7: MIN_TOTAL Algorithm for Filter Decomposition

3.3.3 MIN_TOTAL Algorithm

The previous two algorithms target different portion of the cost exposed in Equation 3.2. This algorithm uses a greedy heuristic to minimize the overall cost.

Our approach is as follows. Starting from the beginning of the pipeline, we successively decide the filter assignment for each of the computing unit. Our algorithm is presented in Figure 3.7.

The algorithm iterates to choose the filters placed on computing units C₁ through Cₘ₋₁. The variable Current_pos denotes the first atomic filter which has not been
assigned to any computing hosts. Initially, its value is 1. For choosing filters placed on \( C_i \), we consider all options ranging from \( f_{\text{Current,\pos}} \) only to \( f_{\text{Current,\pos}}, \ldots, f_n \). We estimate the execution time associated with each choice, and keep the one with the minimum value.

However, it is not possible to accurately estimate the execution time without knowing the placement of the following atomic filters. For this purpose, we take an approximation. The term \( \text{Aggregate-Power}(i) \) denotes the aggregate power of all computing units from \( i + 1 \) to \( m \). For estimating the cost associated with placing atomic filters \( f_{\text{Current,\pos}}, \ldots, f_j \) \( (j = \text{Current,\pos}, \ldots, n) \) on \( C_i \), we assume that the \( i^{th} \) computing unit is connected with a single computing unit which has the power \( \text{Aggregate-Power}(i) \). The function \( \text{Evaluate} \) uses this approximation, along with the actual computing power \( P(C_i) \) and the network bandwidth \( B(L_i) \).

Clearly, it is easy to get the final placement from the array \( \text{min,\pos} \), since \( \text{min,\pos}[i] \) actually identifies the number of the last atomic filter on \( C_i \).

Applying the MIN-TOTAL algorithm on Example 1 in Section 3.3.1, assuming there are 4500 packets, we get the array \( \text{min,\pos}[2] = \{2, 5\} \). Converting it to the placement, we have

\[
\mathcal{V}(f_1, \ldots, f_n) = (f_{1-2}, f_{3-5}, f_6).
\]

### 3.3.4 Discussion

We now compare the three algorithms using Example 1. Consider an application with six atomic filters. If we want to deploy it on a pipeline with three computing units, there are 28 possible placement options in total. With the constraints set in Example 1, there are still 21 choices left. Suppose there are 4500 packets to
be processed in the pipeline. The other parameters we use were stated earlier in Section 3.3.1. The 21 possible placements and their corresponding predicted execution times are presented in Table 3.1.

<table>
<thead>
<tr>
<th>Placement</th>
<th>Predicted Run Time (ms)</th>
<th>Chosen by Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>$f_{2-6}$</td>
<td>9900.37</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_2$</td>
<td>7651.20</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_2$</td>
<td>11251.50</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_2$</td>
<td>2551.03</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_2$</td>
<td>2850.83</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_2$</td>
<td>3750.93</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_3$</td>
<td>7651.68</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_3$</td>
<td>11252.00</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_4$</td>
<td>2251.58</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_4$</td>
<td>2251.45</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_5$</td>
<td>3751.42</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_6$</td>
<td>11253.20</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_6$</td>
<td>6751.72</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_6$</td>
<td>6751.58</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_6$</td>
<td>6751.88</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_6$</td>
<td>4276.03</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_6$</td>
<td>4275.90</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_6$</td>
<td>4276.20</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_6$</td>
<td>4725.83</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_6$</td>
<td>4726.13</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_6$</td>
<td>5401.33</td>
</tr>
</tbody>
</table>

Table 3.1: Possible Placements and their Predicted Performance

Here the theoretical run time is calculated as follows. For a placement

$$\overline{r}(f_1, \ldots, f_n) = (f_{1-2}, f_{3-5}, f_6),$$

the computation workload on each computing unit is the summation of the individual workload of each atomic filter assigned to this unit. Thus, we have $Comp_{f_{1-2}} =$

41
\( Comp_{f_1} + Comp_{f_2} = 700 \mu s \). Similarly, we can compute \( Comp_{f_{a-\delta}} = 1400 \mu s \). Then the bottleneck stage in this pipeline will be the 1\textsuperscript{st} communication stage, with cost 500\( \mu s \). With the Equation 3.2, we can conclude \( T_{\overline{F}(f_1, \ldots, f_n)} = 225140 \mu s \).

As we can see, the placements selected by our three algorithms are among the top four most efficient ones. In fact, the MIN\_ONETRIP algorithm is more suitable when we have a small number of packets. Therefore, it is not surprising that it does not produce the best results when we have 4500 packets. In contrast, the algorithm MIN\_BOTTLENECK is more appropriate when the number of packets is large. The placement chosen by this algorithm is within 0.01\% of the optimal. MIN\_TOTAL, which is only an approximate greedy algorithm, finds the best results in this example.

### 3.3.5 A Brute Force Approach

We provide this section for the sake of completeness. To solve an optimization problem, a brute-force approach is always an option, though it might not be practical. To tackle the filter decomposition problem by traversing the entire search space exhaustively, we must consider a total of \( C_{m-1}^{n+m-1} \) possible placements, evaluate each of them, and recognize the most promising one. Obviously, this invokes exponential cost in the value of \( m \). This approach is briefed in Figure 3.8.

### 3.4 Experimental Results

In this section, we report results from a series of experiments conducted with the following two goals: 1) demonstrating the effectiveness of our algorithms on real applications, and 2) understanding the limitations of our current work to direct our
future efforts. Before discussing the results, we first describe the experimental setting and the applications we have used.

### 3.4.1 Experimental Setting and Applications

In the long run, we expect that pipelined parallelism can be exploited in wide-area networks. However, this is going to require high bandwidth networks and certain level of quality of service support. Recent trends are clearly pointing in this direction, as seen by the National Lambda Rail (NLR) effort and the Optiputer project. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single cluster. The cluster we used had 1 GHz Pentium machines connected through Gigabit Ethernet.
We used three applications that followed a very similar processing pattern: reading original data from a remote data repository, performing application-specific processing, and forwarding the final results to the user’s desktop. In our experiments, they were deployed on a pipeline of 3 computing units, which were connected through 2 communication links.

We simulated configurations with various relative communication bandwidths at different links and varying computing power at the stages. These are shown in Table 3.2. The numbers on the top line of each configuration represent the relative power of the computing unit, and those on the bottom denote the relative communication bandwidth. The value 1 denotes the computing power of the 1 GHz machines or the bandwidth available in a cluster with a Gigabit Ethernet. Lower computing power was simulated by putting more work on the computing processors, and lower communication bandwidth was simulated by delaying packets on the links.

We briefly explain the rationale behind the choice of these configurations. Config 1 is the default configuration. Config 2 has lower and uneven communication bandwidths, and corresponds to the environment where the remote data repository is severely bandwidth limited. Config 3 emulates the situation where both data repository and the intermediate processing unit have extremely fast processing speed and high bandwidth interconnect, relative to the user’s desktop. Config 4 shows the scenario where intermediate processing unit has the highest processing power, followed by the remote repository, and then the user’s desktop.

The first application we use is Virtual Microscope, which is an emulation of a microscope which allows users to view part of the original image with numerous resolutions [13]. A query specifies a rectangular region and a value for the subsampling
<table>
<thead>
<tr>
<th>Config 1:</th>
<th>1—1—1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1    1</td>
</tr>
<tr>
<td>Config 2:</td>
<td>1—1—1</td>
</tr>
<tr>
<td></td>
<td>0.1  0.5</td>
</tr>
<tr>
<td>Config 3:</td>
<td>1—1—0.01</td>
</tr>
<tr>
<td></td>
<td>1     0.001</td>
</tr>
<tr>
<td>Config 4:</td>
<td>0.1—1—0.01</td>
</tr>
<tr>
<td></td>
<td>1     0.001</td>
</tr>
</tbody>
</table>

Table 3.2: Configurations Used for Our Experiments

*factor* for the desired output. This application have 5 atomic filters, which are for reading original JPEG image from the disk-resident dataset, decompressing the image to RGB format, clipping the image with the user specified range, zooming the image according to the user desired resolution, and viewing the image.

Two other applications both implement isosurface rendering. They are, z-buffer based isosurface rendering and active pixels based rendering, referred to as ZBUF and ACTP, respectively. Isosurface rendering is a key visualization problem. The inputs to the problem include a three-dimensional grid, a scalar isosurface value, and a two-dimensional viewing screen associated with an angle. The goal is to view a surface, as seen from the given viewing angle, which captures the points in the grid where the scalar value matches the given isosurface value. For this application, there are 7 atomic filters in total, to fulfill the tasks ranging from reading the original images from disk, organizing them as cubes, extracting triangles out of the cubes, rastering the triangles, and finally rendering.

Note that in our implementation of different versions, we have one major difference from the cost model we have used. If two atomic filters are put together at one site, we
Figure 3.9: Execution Time for Virtual Microscope: Config 1

actually merge them into one filter, saving the cost of copying between buffers. As we
will observe later, this will have some impact on the effectiveness of our algorithms.

3.4.2 Experiments with Virtual Microscope

We use a 800 MB image for this application, which corresponds to a $29328 \times 28800$
region. We considered 3 queries on the image, which are denoted as Q1, Q2, and Q3,
respectively. The corresponding number of packets processed in these three queries
is 1, 4, and 4500. Here, one packet always contains a region of size $256 \times 256$.

In Tables 3.3, 3.4, 3.5, and 3.6, we summarize the placements generated by dif-
f erent algorithms under 4 configurations for the above three queries. Besides the
three algorithms we presented in the previous section, we have also shown the results
obtained from the exponential exhaustive search algorithm, denoted as Exha_Search.
<table>
<thead>
<tr>
<th>Query</th>
<th>Algorithm</th>
<th>Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$C_1$</td>
</tr>
<tr>
<td>Q1</td>
<td>MIN_ONETRIP</td>
<td>$f_{1-4}$</td>
</tr>
<tr>
<td></td>
<td>MIN_BOTTLENECK</td>
<td>$f_{1-4}$</td>
</tr>
<tr>
<td></td>
<td>MIN_TOTAL</td>
<td>$f_{1-4}$</td>
</tr>
<tr>
<td></td>
<td>Exha_Search</td>
<td>$f_{1-4}$</td>
</tr>
<tr>
<td>Q2</td>
<td>MIN_ONETRIP</td>
<td>$f_{1-3}$</td>
</tr>
<tr>
<td></td>
<td>MIN_BOTTLENECK</td>
<td>$f_{1-3}$</td>
</tr>
<tr>
<td></td>
<td>MIN_TOTAL</td>
<td>$f_{1-3}$</td>
</tr>
<tr>
<td></td>
<td>Exha_Search</td>
<td>$f_{1-3}$</td>
</tr>
<tr>
<td>Q3</td>
<td>MIN_ONETRIP</td>
<td>$f_{1-4}$</td>
</tr>
<tr>
<td></td>
<td>MIN_BOTTLENECK</td>
<td>$f_{1-4}$</td>
</tr>
<tr>
<td></td>
<td>MIN_TOTAL</td>
<td>$f_{1-4}$</td>
</tr>
<tr>
<td></td>
<td>Exha_Search</td>
<td>$f_{1-4}$</td>
</tr>
</tbody>
</table>

Table 3.3: Placements Generated by Different Algorithms: Virtual Microscope, Config 1

Figures 3.9, 3.10, 3.11, and 3.12 show the actual execution times. The execution times from Q3 are always normalized, so that they could be shown on the same chart with other values.

Two sets of observations can be made from these results. First, the performance variance between different algorithms is very small. This helps to show the effectiveness of our heuristic polynomial-time algorithms. Second, these experiments show some limitations of our modeling of the problem, that is, the relative performance of the algorithms is not what one would expect. The Exha.Search algorithm does not always perform best as we expect.

There are two major limitations of how we have modelled the problem that come into play here. We only use the execution time from one packet for each atomic filter as the input to our algorithm, assuming that it remains the same for different packets.
In practice, the amount of computation in a particular atomic filter may vary from packet to packet. Also, in our cost model, when we put two atomic filters $f_1$ and $f_2$ on the same computing unit $C_1$, the new execution time on $C_1$ is calculated as the sum of execution time of each individual filter. However, in our actual implementation, we generate a new filter that avoids the copying between the buffers. Therefore, the time taken by the new filter is less than the time required by $f_1$ plus the time required by $f_2$.

### 3.4.3 Experiments with Iso-Surface

There are two datasets we used for this set of experiments, which were generated by an environmental simulator ParSSim [17] and were previously used in earlier studies also [28, 55, 54]. These datasets comprised grid data for 10 time-steps, and were 1.5 GB and 6 GB, respectively. The two datasets are referred to as small and large datasets. We report experiments on processing a single time-step, the data
Table 3.4: Placements Generated by Different Algorithms: Virtual Microscope, Config 2

corresponding to which is 150 MB and 600 MB for these two datasets. The number of packets involved in the processing of small dataset is 3, and that for the large dataset is 47. For Iso-surface, we only report results from Config 1, as the trends from all configurations were very similar.

In Table 3.7, we list the placements generated by different algorithms for the two versions of Iso-surface, ZBUF and ACTP. Figures 3.13 and 3.14 plot the actual execution time of the application on two datasets, with the placement selected by our algorithms.

As we can see, the difference in the performance of the algorithms is limited. The MIN_TOTAL algorithm always finds a placement which performs best for the small dataset, while the algorithm MIN_ONETRIP provides the best performance for the large dataset. In both the cases, the exhaustive search algorithm does not
generate the most efficient placement. The main reason is that the Iso-surface rendering application is very data dependent, which introduces more variance for different packets.

To further understand the impact of data dependence, we performed additional experiments. Figures 3.15 and 3.16 show the execution time when the same packet is processed multiple times, based on the placements generated by our algorithms. As we can see, the results are much closer to our expectations. The performance of the exhaustive search algorithm is either the best, or very close to the best. Because MIN_ONETRIP models only a small part of the cost when the number of packets is large, its performance is the worst with 100 runs with the same packet. The performance of both MIN_TOTAL and MIN_BOTTLENECK is very similar to the exhaustive search one.
<table>
<thead>
<tr>
<th>Query</th>
<th>Algorithm</th>
<th>Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MIN_ONETRIP</td>
<td>$f_{1-1}$</td>
</tr>
<tr>
<td>Q1</td>
<td>MIN_BOTTLENECK</td>
<td>$f_{1-4}$</td>
</tr>
<tr>
<td></td>
<td>MIN_TOTAL</td>
<td>$f_{1-1}$</td>
</tr>
<tr>
<td></td>
<td>Exha_Search</td>
<td>$f_{1-1}$</td>
</tr>
<tr>
<td>Q2</td>
<td>MIN_ONETRIP</td>
<td>$f_{1-3}$</td>
</tr>
<tr>
<td></td>
<td>MIN_BOTTLENECK</td>
<td>$f_{1-3}$</td>
</tr>
<tr>
<td></td>
<td>MIN_TOTAL</td>
<td>$f_{1-3}$</td>
</tr>
<tr>
<td></td>
<td>Exha_Search</td>
<td>$f_{1-3}$</td>
</tr>
<tr>
<td>Q3</td>
<td>MIN_ONETRIP</td>
<td>$f_{1-1}$</td>
</tr>
<tr>
<td></td>
<td>MIN_BOTTLENECK</td>
<td>$f_{1-1}$</td>
</tr>
<tr>
<td></td>
<td>MIN_TOTAL</td>
<td>$f_{1-1}$</td>
</tr>
<tr>
<td></td>
<td>Exha_Search</td>
<td>$f_{1-1}$</td>
</tr>
</tbody>
</table>

Table 3.5: Placements Generated by Different Algorithms: Virtual Microscope, Config 3

3.4.4 Discussion

On the positive side, our results have shown that our heuristic algorithms can perform quite well in practice. However, our results have also shown limitations of our model. So, in our future work, we will try to model the following two aspects of the applications:

- Estimate of the performance change resulting from combining several atomic filters and avoiding the cost of copying between the buffers.

- Estimate of the impact of data dependence on the performance for each atomic filter.

Clearly, these characteristics can be very application dependent and hard to model. For example, we could try to limit the data impact by using the average execution
time of all packets on a certain atomic filter. Still, our model may not be accurate, especially if there is a large variance in the processing time for each packet. Further, we need to experiment with applications having a larger number of atomic filters and/or configurations with more stages, to gain additional insights about these algorithms.

### 3.5 Conclusions

The work presented here has been in the context of coarse-grained pipelined execution model. This model is very suitable for the execution of data-driven applications in an environment where the data is available on remote data repositories and the results are desirable on a user’s desktop. In this context, we have focused on the problem of filter decomposition.

We propose three polynomial time algorithms for this problem. Algorithm MIN_ONETRIP is a dynamic programming algorithm that optimizes the one trip cost for a packet passing through the pipeline. Algorithm MIN_BOTTLENECK is also a
<table>
<thead>
<tr>
<th>Query</th>
<th>Algorithm</th>
<th>Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$C_1$</td>
</tr>
<tr>
<td>Q1</td>
<td>MIN_ONETRIP</td>
<td>$f_1$</td>
</tr>
<tr>
<td></td>
<td>MIN_BOTTLENECK</td>
<td>$f_1$</td>
</tr>
<tr>
<td></td>
<td>MIN_TOTAL</td>
<td>$f_1$</td>
</tr>
<tr>
<td></td>
<td>Exha_Search</td>
<td>$f_1$</td>
</tr>
<tr>
<td>Q2</td>
<td>MIN_ONETRIP</td>
<td>$f_{1-3}$</td>
</tr>
<tr>
<td></td>
<td>MIN_BOTTLENECK</td>
<td>$f_1$</td>
</tr>
<tr>
<td></td>
<td>MIN_TOTAL</td>
<td>$f_1$</td>
</tr>
<tr>
<td></td>
<td>Exha_Search</td>
<td>$f_{1-3}$</td>
</tr>
<tr>
<td>Q3</td>
<td>MIN_ONETRIP</td>
<td>$f_{1-4}$</td>
</tr>
<tr>
<td></td>
<td>MIN_BOTTLENECK</td>
<td>$f_1$</td>
</tr>
<tr>
<td></td>
<td>MIN_TOTAL</td>
<td>$f_1$</td>
</tr>
<tr>
<td></td>
<td>Exha_Search</td>
<td>$f_1$</td>
</tr>
</tbody>
</table>

Table 3.6: Placements Generated by Different Algorithms: Virtual Microscope, Config 4

dynamic programming algorithm, which minimizes the time spent on the bottleneck stage. Finally, MIN_TOTAL is an approximate greedy algorithm which tries to minimize the total execution time.

We have evaluated these three algorithms using three applications and many different configurations. Our results show that our heuristic algorithms work quite well in practice, with the possible exception of MIN_ONETRIP when the number of packets is large. However, the relative performance of the algorithms is not always what we would expect, because of the certain limitations in how we model the problem.
<table>
<thead>
<tr>
<th>Query</th>
<th>Algorithm</th>
<th>Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$C_1$</td>
</tr>
<tr>
<td>ZBUF</td>
<td>MIN_ONETRIP</td>
<td>$f_1$</td>
</tr>
<tr>
<td></td>
<td>MIN_BOTTLENECK</td>
<td>$f_{1-2}$</td>
</tr>
<tr>
<td></td>
<td>MIN_TOTAL</td>
<td>$f_{1-3}$</td>
</tr>
<tr>
<td></td>
<td>Exha_Search</td>
<td>$f_1$</td>
</tr>
<tr>
<td>ACTP</td>
<td>MIN_ONETRIP</td>
<td>$f_1$</td>
</tr>
<tr>
<td></td>
<td>MIN_BOTTLENECK</td>
<td>$f_{1-4}$</td>
</tr>
<tr>
<td></td>
<td>MIN_TOTAL</td>
<td>$f_{1-3}$</td>
</tr>
<tr>
<td></td>
<td>Exha_Search</td>
<td>$f_{1-4}$</td>
</tr>
</tbody>
</table>

Table 3.7: Placements Generated by Different Algorithms: Iso-Surface Rendering, Config 1

![Execution Time Graph](image_url)

Figure 3.13: Execution Time of ZBUF and ACTP: Small Dataset
Figure 3.14: Execution Time of ZBUF and ACTP: Large Dataset

Figure 3.15: Execution Time for ZBUF: Runs with the same packet
Figure 3.16: Execution Time for ACTP: Runs with the same packet
CHAPTER 4

PACKET SIZE OPTIMIZATION FOR SUPPORTING COARSE-GRAINED PIPELINED PARALLELISM

4.1 Introduction

With the algorithms presented in Chapter 3, we can decompose an application into several stages. However, another important factor that impacts the execution of applications with the coarse-grained pipelined model is packet size, i.e., the unit of data transfer from one pipeline stage to another. Clearly, a very large packet size may prevent the pipelined parallelism from being exploited. At the same time, a very small packet size could lead to high overheads because of communication latencies. Thus, choosing a suitable packet size is important for performance.

In this chapter, we develop and validate analytical models for determining the packet size that will result in the lowest execution time. Our work builds on the work of Wang et al. on packet size optimization for multi-link communication pipeline [144]. The key difference in our work is that computation is involved in the pipeline and needs to be modeled. Moreover, because of computation, the frequency and/or the volume of communication at different stages can be different.
We consider two distinct models, *fixed-frequency* and *fixed-size* communication. In the fixed-frequency model, all computation stages except the last one send out one packet after getting one as input. The size of the output packet may be different from the size of the input packet. In the fixed-size model, the computation stages always communicate same size or almost same size packets. To meet this requirement, the stages may need to wait and process more than one input packets, so the frequency could be different. We have developed analytical models for determining the optimal packet size for both of the above models.

We have carried out a detailed evaluation of our work using three applications, executed with different parameters and datasets. The main observations from our experiments are as follows. First, we show that the choice of packet size can make a large difference in the execution time. Second, we have shown that the packet sizes chosen by our model result in the lowest or very close to the lowest possible execution time. With little extra effort, our models can be applied to applications running on the Grid environment.

Furthermore, these models are also integrated into a compilation system and worked with real applications. The system applies static analysis as well as program profiling techniques to find out the characteristics of the application and its execution environment, then uses them as parameters in the model to estimate the best communication granularity. The details of the system are presented in Chapter 5.

The rest of the chapter is organized as follows. The packet size problem and our basic approach are discussed in Section 4.2. The details of analytical model for the different cases are presented in Section 4.3. Section 4.4 focuses on the experimental validation of our models. We summarize in Section 4.5.
4.2 Packet Size Optimization Problem

This section formulates the packet size optimization problem that we are addressing in this chapter. We also illustrate the various cases we are considering. Detailed results for each of these cases are presented in Section 4.3.

4.2.1 Overview of the Problem

As we had stated previously, in our target setting, the application is decomposed into several stages and mapped onto a pipeline of computing and communication units. The first stage of the pipeline is always on the site hosting the data repository and is responsible for reading, subsetting, and forwarding the data to the following stages. The last stage is typically on a user’s desktop and in charge of collecting and viewing the results.

Suppose the entire operation involves reading and processing a data file of size $B$. One possibility is to transmit and process this file as one big packet. This option is likely to be impractical, because of the limited memory that may be available at different stages of the pipeline. Moreover, this option does not allow overlapping of the operations at different stages of the pipeline. Now, consider another extreme, where a single data-item is received, processed, and forwarded at each processing stage. Again, this option is likely to give poor performance, because of the high communication latency and the overhead involved in switching between communication and computation at each node. Thus, an important optimization parameter for getting high performance from the pipeline is the granularity at which packets are received, processed, and forwarded at each pipeline unit.
4.2.2 Solution Approach

Optimization of packet size in a multi-link communication pipeline has been researched by Wang et al. [144]. While their formulation can form the basis for our analysis, their results are not directly applicable to the problem we are considering. This is because of the following two reasons:

- Our pipeline includes both communication and computation stages. Both of them need to be modeled for determining the impact of packet size on the overall execution time.
- Because of computation, the volume and/or the frequency of communication between the different stages can vary. This is not directly handled by Wang et al.’s work.

Consider a pipeline with \( m \) filters or computing stages. There are \( m-1 \) streams connecting these stages. For our analysis, we consider it as a pipeline with \( n = 2m-1 \) stages. Similar to Wang et al., we consider the communication time to be an affine function of the size of the packet communicated, i.e., it comprises a fixed overhead per message and a per-byte cost. Based upon our experiences with our target class of applications, we find that the same model can be used for computation stages. There is usually a fixed overhead associated with receiving a packet and starting the processing. The remaining cost is usually linear in the size of the packet.

To present our analysis, we use the following terminology.

- \( n \): the number of pipeline stages, including both computation and communication stages
• $G_i$: the fixed per-packet overhead for stage $i$

• $g_i$: the per-byte cost for stage $i$

• $B$: the size of the entire data file

• $k$: the number of packets

• $t_{ij}$: the time the $i$th packet spends in the $j$th stage, $t_{ij} = \text{size of packet } i \times g_j + G_j$

We still need to be able to handle the fact that the volume and/or the frequency of communication between different stages may not be the same. This is because an output packet of a computation stage is most likely different from its input packet, in both content and size. We introduce $\alpha_i$ as the ratio of output size to input size for the stage $i$. Obviously, this value is equal to 1 for all communication stages. We also assume that the value is same for all packets passing through stage $i$.

In our analysis, we will consider two models of execution. These models are fixed-frequency and fixed-size communication. In the fixed-frequency model, all computation stages except the last one send out one packet after getting one as input. The size of the output packet may be different from the size of the input packet. In the fixed-size model, the computation stages always communicate same size or almost same size packets. That is, the computation stages might need to wait for more than one input packets in order to prepare an output packet of the required size. Note that it is possible to execute the pipeline in a way that both the frequency and size of packets communicated may be different. We do not consider this possibility in this chapter.
To choose the packet size to minimize the execution time, we need expressions for the total execution time. Before elaborating the different cases we are handling, we review the basic results from Wang et al. [144]. The expression for execution time is based upon the notion of a bottleneck stage. This is the stage which is always busy after the first packet is received, and before the last packet exits it. We assume that the \( b^{th} \) stage is the bottleneck stage, and is characterized by the two-tuple \((G_b,g_b)\). Then the total execution time of a program can be stated as:

\[
T = T_f + T_b + T_l
\]  

(4.1)

where \( T_f, T_b, \) and \( T_l \) are defined as follows:

- \( T_f \): the time the first packet takes to reach the bottleneck stage.
- \( T_b \): the time all packets spend in the bottleneck stage.
- \( T_l \): the time the last packet takes to exit the pipeline after leaving the bottleneck stage.

Figure 4.1 shows the execution time-line of a pipeline consisting of three computing stages and two communication stages.

Note that the above formulation assumes that there is only a single bottleneck stage in the pipeline. However, this does not limit our analysis in any way. Suppose, there are two bottleneck stages in a pipeline. Then, the amount of time spent by all packets in these two stages will be identical. Therefore, we will derive the same expression by considering either of these two as the bottleneck.
Figure 4.1: Execution Time-line of a Pipeline with Three Computation Stages and Two Communication Stages. Stage 3 is the bottleneck stage.

4.3 Choosing Optimal Packet Sizes

In this section, we derive the expressions for choosing the optimal packet size. Recall that $B$ is the total size of input file, $k$ is the number of packets used, and $p = \lceil \frac{B}{k} \rceil$ is the size of each packet. We will aim at determining the value of $k$ in our analysis. In the next two subsections, we focus on fixed-frequency and fixed-sized communication patterns, respectively.

4.3.1 Fixed-Frequency Communication Pattern

The expression for minimizing the execution time depends upon which stage is the bottleneck stage. Here the bottleneck stage is defined as the pipeline stage which has the largest processing time for each incoming packet. Considering the packet size change pattern shown in Figure 4.3, the formal definition of bottleneck stage $b$ under the fixed-frequency communication model is as follows.
Figure 4.2: Execution Time-line of A 1st-Stage Bottleneck Pipeline under Fixed-Frequency Communication Pattern

Stage$_1$ Stage$_2$ \ldots \text{Stage}_i$ \ldots \text{Stage}_n

size of packet $\frac{B}{k} \alpha_1 \frac{B}{k} \ldots \alpha_i \frac{B}{k} \ldots \alpha_{i-1} \frac{B}{k} \alpha_1 \ldots \alpha_{n-1} \frac{B}{k}$

Figure 4.3: Changing Size of the Packet Flowing Through the Pipeline Shown in Figure 4.2

\[
b = \left\{ i \left| \left(\prod_{j=1}^{i-1} \alpha_j\right) \frac{B}{k}g_i + G_i = \max \left\{ \left(\prod_{l=1}^{j-1} \alpha_l\right) \frac{B}{k}g_j + G_j \right\}, j = 1, \ldots, n \right. \right\}.
\]

We separately consider two different cases, corresponding to bottleneck at the first stage, and bottleneck at any of the other stages.
4.3.1.1 Bottleneck at the First Stage

Figure 4.2 shows the execution time-line for a 1st-stage bottleneck pipeline, and Figure 4.3 illustrates how the size of a packet changes in the pipeline. We rewrite $T_f$, $T_b$ and $T_i$ as follows.

$$T_f = 0$$

$$T_b = B * g_1 + k * G_1$$

$$T_i = (\alpha_1 \frac{B}{k} * g_2 + G_2) + (\alpha_1 \alpha_2 \frac{B}{k} * g_3 + G_3) + \cdots$$

$$+ (\alpha_1 \alpha_2 \cdots \alpha_{i-1} \frac{B}{k} * g_i + G_i) + \cdots$$

$$+ (\alpha_1 \alpha_2 \cdots \alpha_{n-1} \frac{B}{k} * g_n + G_n)$$

(4.4)

Substituting equations 4.2, 4.3 and 4.4 in the equation 4.1, we get

$$T = T_f + T_b + T_i$$

$$= B * g_1 + k * G_1 +$$

$$+ (\alpha_1 \frac{B}{k} * g_2 + G_2) + (\alpha_1 \alpha_2 \frac{B}{k} * g_3 + G_3) + \cdots$$

$$+ (\alpha_1 \alpha_2 \cdots \alpha_{i-1} \frac{B}{k} * g_i + G_i) + \cdots$$

$$+ (\alpha_1 \alpha_2 \cdots \alpha_{n-1} \frac{B}{k} * g_n + G_n)$$

$$= B * g_1 + k * G_1 + \sum_{i=2}^{n} \left( \prod_{j=1}^{i-1} \alpha_j \right) \frac{B}{k} * g_i + G_i$$

(4.5)

Since $\frac{d^2 T}{dk^2} > 0$, to obtain the value of $k$ such that $T$ is minimized, we differentiate the above equation with respect to $k$, and set the result to 0.
Figure 4.4: Execution Time-line of a Pipeline With Bottleneck at Second or Later Stages: Fixed-Frequency Communication Pattern

\[
\frac{dT}{dk} = -B \sum_{i=2}^{n} \left( \Pi_{j=1}^{i-1} \alpha_j \right) g_i \frac{k}{k^2} + G_1 = 0
\]  

(4.6)

Solving equation 4.6, we can get

\[
k = \left[ \frac{B \sum \left( \Pi_{j=1}^{i-1} \alpha_j \right) g_i}{G_1} \right].
\]  

(4.7)

4.3.1.2 Bottleneck at Second or Later Stages

The execution time-line of a pipeline with bottleneck at second or later stages is given in Figure 4.4. The change in the size of the packet is already depicted by Figure 4.3. Now Stage_0 is the bottleneck stage, where \( b \neq 1 \). We rewrite \( T_f, T_b \) and \( T_l \) as follows.

\[
T_f = \left( \frac{B}{k} * g_1 + G_1 \right) + \left( \alpha_1 \frac{B}{k} * g_2 + G_2 \right) + \cdots + \\
\left( \alpha_1 \alpha_2 \cdots \alpha_{i-1} \frac{B}{k} * g_i + G_i \right) + \cdots +
\]
\[(\alpha_1 \alpha_2 \cdots \alpha_{b-2} \frac{B}{k} g_{b-1} + G_{b-1})\]  \hspace{1cm} (4.8)

\[T_b = \alpha_1 \alpha_2 \cdots \alpha_{b-1} B g_b + k \cdot G_b\]  \hspace{1cm} (4.9)

\[T_l = (\alpha_1 \alpha_2 \cdots \alpha_b \frac{B}{k} g_{b+1} + G_{b+1}) + \cdots + (\alpha_1 \alpha_2 \cdots \alpha_{n-1} \frac{B}{k} g_n + G_n)\]  \hspace{1cm} (4.10)

Substituting equations 4.8, 4.9 and 4.10 in the equation 4.1, differentiating the resulting equation with respect to \(k\), and setting the result to 0, then solving the equation, we get

\[k = \left[ \sqrt{\frac{B (g_l + \sum_{i \neq b} (\prod_{j=1}^{i-1} \alpha_j g_i) + G_b)}{G_b}} \right]. \]  \hspace{1cm} (4.11)

### 4.3.2 Fixed-Size Communication Pattern

This scheme differs from the previous one in that there are fixed-size packets flowing on the communication links, except the last packet. Hence, communication frequency varies depending on the packet content and the filtering condition. Such pattern may save us some overhead. On the other hand, it can also introduce more idle time for later stages, which in turn increases the total runtime.

\[b = \left\{ \begin{array}{l} i \prod_{j=1}^{l-1} \alpha_j \left( \frac{B}{k} g_i + G_i \right) = \\ \max \left\{ \prod_{j=1}^{l-1} \alpha_j \left( \frac{B}{k} g_i + G_i \right) \right\}, i = 1, \ldots, n \end{array} \right\} \]  \hspace{1cm} (4.12)

As mentioned earlier, \(\alpha_i\) is the ratio of output size with respect to input size for stage \(i\). So to have an output packet whose size is same as that of the input, stage

67
Figure 4.5: Execution Time-line of a Pipeline With Bottleneck at Second or Later Stages: Fixed-Size Communication Pattern

\( i \) needs to process \( \left\lceil \frac{1}{\alpha_i} \right\rceil \) packets before sending out a packet. Here, we define the bottleneck stage \( b \) as in \( 4.12 \).

Intuitively, we compare the time spent at each stage to process the original packet, and the slowest one is the bottleneck stage. We again consider the two cases, corresponding to bottleneck at the first stage, and bottleneck at second or later stages. The latter is discussed first.

4.3.2.1 Bottleneck at Second or Later Stages

Figure 4.5 depicts the execution time-line for a pipeline with bottleneck at second or later stages. The size of a packet flowing in the pipeline is \( \frac{B}{k} \). Note that longer delays for later stages are possible here, due to the fixed-size communication requirement. Each stage \( i \) needs to process \( \left\lceil \frac{1}{\alpha_i} \right\rceil \) input packets when preparing the output. Taking into account these factors, \( T_f, T_b \) and \( T_i \) could be rewritten as follows.

\[
T_f = \left\lceil \frac{1}{\alpha_1} \right\rceil \left( \frac{B}{k} \cdot g_1 + G_1 \right) + \cdots +
\]

68
Figure 4.6: Execution Time-line of a 1st Stage Bottleneck Pipeline under Fixed-Size Communication Pattern

\[
\begin{bmatrix}
\frac{1}{\alpha_i} & \left(\frac{B}{k} \cdot g_i + G_i\right) \\
\frac{1}{\alpha_{b-1}} & \left(\frac{B}{k} \cdot g_{b-1} + G_{b-1}\right)
\end{bmatrix}
\]

(4.13)

\[T_b = \alpha_1 \alpha_2 \cdots \alpha_{b-1} k \left(\frac{B}{k} \cdot g_b + G_b\right)\]  

(4.14)

\[T_i = \left(\frac{B}{k} \cdot g_{b+1} + G_{b+1}\right) + \cdots + \left(\frac{B}{k} \cdot g_n + G_n\right)\]  

(4.15)

Substituting equations 4.13, 4.14 and 4.15 in the equation 4.1, then differentiating the resulting equation with respect to \(k\), and setting the result to 0, we get

\[
\frac{dT}{dk} = -B \left(\sum_{i<b} \frac{g_i}{\alpha_i} + \sum_{i>b} g_i\right) k^2 + \left(\prod_{i=1}^{b-1} \alpha_i\right) G_b = 0
\]

(4.16)

Solving equation 4.16, we can get

\[k = \sqrt{\frac{B \left(\sum_{i<b} \frac{g_i}{\alpha_i} + \sum_{i>b} g_i\right)}{\left(\prod_{i=1}^{b-1} \alpha_i\right) G_b}}.
\]

(4.17)
<table>
<thead>
<tr>
<th>Communication Pattern</th>
<th>Bottleneck Stage $b$</th>
<th>$k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed-frequency</td>
<td>$b=1$</td>
<td>$k = \sqrt{\frac{B \sum_{i=1}^{n} (\prod_{j=1}^{i-1} \alpha_j) g_i}{G_1}}$</td>
</tr>
<tr>
<td></td>
<td>$b \neq 1$</td>
<td>$k = \sqrt{\frac{B (g_1 + \sum_{i \neq b} (\prod_{j=1}^{i-1} \alpha_j) g_i)}{G_b}}$</td>
</tr>
<tr>
<td>fixed-size</td>
<td>$b=1$</td>
<td>$k = \sqrt{\frac{B \sum_{i=1}^{n} g_i}{G_1}}$</td>
</tr>
<tr>
<td></td>
<td>$b \neq 1$</td>
<td>$k = \sqrt{\frac{B (\sum_{i &lt; b} \frac{g_i}{\alpha_i} + \sum_{i &gt; b} g_i)}{(\prod_{i=1}^{b-1} \alpha_i) G_b}}$</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of All Models

### 4.3.2.2 Bottleneck at First Stage

This is a special case of Section 4.3.2.1, as there is no idle time before the bottleneck stage begins to work. The execution time-line of such a pipeline is given in Figure 4.6. We rewrite $T_f$, $T_b$ and $T_i$ as follows.

\[
T_f = 0
\]  

(4.18)

\[
T_b = B \cdot g_1 + k \cdot G_1
\]  

(4.19)

\[
T_i = \left( \frac{B}{k} \cdot g_2 + G_2 \right) + \cdots + \left( \frac{B}{k} \cdot g_n + G_n \right)
\]  

(4.20)

Substituting equations 4.18, 4.19 and 4.20 in the equation 4.1, and following the steps shown in section 4.3.1.1, we get

\[
k = \sqrt{\frac{B \sum_{i=2}^{n} g_i}{G_1}}. \tag{4.21}
\]
Figure 4.7: T can be Represented as Summation of Two Simple Linear Functions

We summarize results from all models in table 4.1. There are several observations from these results.

1. The total execution time $T$ can be represented by the summation of two simple linear functions. $T = ak + \frac{b}{k}$, where $a, b > 0$. Figure 4.7 plots these two functions. The cross point $(A)$ of these two lines gives the optimal value of $k$. Also if $a$ is small, the plot will be more like the dotted line, then the variation of $T$ will not be very significant over the range right to $A$, and not too far from $A$. Otherwise, if $a$ is large, a sharper curve could be expected.

2. The final optimal solution only depends on the value of $G_b, \alpha_i$, and the values of $g_i$ for non-bottleneck stages. Factors such as the per-byte cost for bottleneck stage and per-packet cost for non-bottleneck stages do not have any impact of the optimal value of $k$. 

71
4.3.3 Applicability to the Grid Environment

An obvious question is, how can the analytical models we have derived be used in a grid environment, given the variability of the resources in such an environment. Clearly, our models will need the fixed \((G_i)\) and per-byte \((g_i)\) cost for both computing and communication phases. We believe that the use of coarse-grained pipelined model will require the availability of dedicated computing resources. Thus, the fixed and per-byte cost for computing phases will not change during the execution of an application. These costs can be obtained during an initial execution of the application on the same or similar environment and used as part of our models.

The communication bandwidth and latency are more likely to vary dynamically in a grid environment. To obtain information about these, existing tools such as the Network Weather Service (NWS) [146] can be used. The closed-form expressions we have derived can be used with newly obtained parameters and optimal packet size can be computed without a significant runtime overhead.

4.4 Experimental Results

This section reports a series of experiments we conducted for validating our work. Our experiments have focused on two aspects: 1) demonstrating that the choice of packet size can make a significant difference on the overall execution time of our target applications, and 2) showing that the packet sizes suggested by the expressions we have derived give best or very close to the best performance.
4.4.1 Experimental Setting and Applications

In the long run, we expect that pipelined parallelism can be exploited in wide-area networks. However, this is going to require high bandwidth networks and certain level of quality of service support. Recent trends are clearly pointing in this direction, for example, the five sites that are part of the NSF funded Teragrid project expect to be connected with a 40 Gb/second network [125]. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single cluster. The cluster we used had 700 MHz Pentium machines connected through Myrinet LANai 7.0.

In the set of applications we have focused on, the following four computation stages are common:

- Read (R) stage, which is responsible for reading a chunk of points from the data file and prepares packets for sending to subsequent stages.

- A filtering or subsetting stage, denoted by F, which filters out points based upon a relatively inexpensive test.

- A local processing stage, denoted by L, which performs the processing independently on each packet.

- A global processing stage, denoted by G, which combines local results to compute the final results.
It is obvious that $\alpha_1 = 1$ since the R stage does not perform any processing on the input data before it forwards them to next stage. Similarly, for all communication stages $\alpha_i = 1, i = 2, 4, 6$.

We use three applications to test our model, two of which are algorithms implementing isosurface rendering. They are, z-buffer based isosurface rendering and active pixels based rendering, referred to as ZBUF and ACTP, respectively. Isosurface rendering is a key visualization problem. The inputs to the problem are a three-dimensional grid, a scalar isosurface value, and a two-dimensional viewing screen with an angle associated with it. The goal is to view a surface, as seen from the given viewing angle, which captures the points in the grid where the scalar value matches the given isosurface value.

Our third application is k-nearest neighbor search, referred to as KNN. It is one of the basic data mining problems [77]. Here, the training samples are described by an n-dimensional numeric space. Given a new point and a range, the goal is to find the $k$ training samples that are closest to the new point within the specific range. This application also involves four stages.

We created two versions for both KNN and ZBUF, based upon certain choices of parameters. KNN-1st, ACTP and ZBUF-1st are versions in which the first phase of the pipeline is the bottleneck stage, while in KNN-2nd and ZBUF-2nd, the second computation stage is the bottleneck stage. These versions allow us to validate expressions for different cases that we had listed in the previous section.

In our experiments, we have run these applications under both fixed-frequency and fixed-size communication schemes with various input sizes.
4.4.2 Measuring Pipeline Parameters

For using the expressions we have derived for optimal packet sizes, we need to know the fixed overhead and the per-byte cost associated with each stage of an application. We now describe how these values are obtained.

Note that we need to execute an application at least once to obtain these values. However, this is still a lot easier than iteratively finding the optimal packet size. Moreover, the values associated with a communication phase are independent of the application and can be determined just once for the target execution environment. Similarly, if some phases are common between multiple applications, the parameters can be determined once and used later.

To determine the fixed ($G_i$) and per-byte ($g_i$) cost for a computation phase, we identify the program components that represent these costs. This is illustrated by the following example, taken from the Range_Query filter code of the K-nearest neighbor search algorithm. The functionality shown here is to get an input packet from the input stream, and process it point by point.

```
while (1)
{
    if ((pbuf = arg.ins[0].read()) == NULL) break;

    pbufptr = pbuf->getPtr();
    memcpy(&nump, pbufptr, sizeof(int));
    pbufptr += sizeof(int);

    for (int i=0; i<nump; i++)
    {
        t = new KPOINT();
```

75
<table>
<thead>
<tr>
<th>Communication Pattern</th>
<th>Applications</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-frequency</td>
<td>KNN-1st</td>
<td>$G_1$, $g_3$, $g_5$, $\alpha_3$</td>
</tr>
<tr>
<td></td>
<td>ZBUF-2nd</td>
<td>$G_3$, $g_1$, $g_5$, 75.85%</td>
</tr>
<tr>
<td></td>
<td>ACTP</td>
<td>$G_1$, $g_3$, $g_5$, $\alpha_3$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.41, 0.054, 0.013, 75.85%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.1606, 0.1289, 0.7569, 3.42%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5338, 0.1494, 0.2191, 2.95%</td>
</tr>
<tr>
<td>Fixed-size</td>
<td>KNN-2nd</td>
<td>$G_3$, $g_1$, $g_5$, $\alpha_3$</td>
</tr>
<tr>
<td></td>
<td>ZBUF-1st</td>
<td>$G_1$, $g_3$, $g_5$, 1</td>
</tr>
<tr>
<td></td>
<td>ACTP</td>
<td>$G_1$, $g_3$, $g_5$, $\alpha_3$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.4087, 0.0872, 0.0127, 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3848, 0.1207, 1.0296, 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.4552, 0.1356, 0.1937, 1</td>
</tr>
</tbody>
</table>

Table 4.2: Parameters for Computational Phases for All Applications

```java
...
}
... ...
}
```

In the above code, the cost to execute statements (2), (3) and (4) does not change with the input packet size, and is considered as part of the fixed overhead. The cost for executing for statement is determined by the value of `nump`, which is the number of points contained in an input packet. So, the execution time of all statements inside for loop is considered as the per-byte cost. We instrument the code to measure both of the above. The parameters we obtained for the computational phases for all applications are summarized in Table 4.2.

We used the following procedure to determine the parameters associated with a communication phase. It is important to note that this is not strictly the time
required over the communication link. The time spent in the middleware in preparing or receiving a message is also included in the communication time. This is illustrated through the Figure 4.8. The communication time starts when sender finishes the send operation, and ends when receiver finishes the receive operation, i.e. it is of duration \( t_2 - t_1 \) in Figure 4.8.

For the communication links in the cluster we used, \( G_i \) was 32.6 \( \mu s \) and \( g_i \) was 1.73 \( \mu s \), for very communication phase \( i \).

### 4.4.3 Evaluation of Fixed-Frequency Model

We now present experimental results validating the expressions we have derived for the fixed-frequency model.

#### 4.4.3.1 K-Nearest Neighbor Search

In this experiment, we test the KNN-1st with \( K = 3 \), \( B = 108M \), and \( \alpha_3 = 75.85\% \). In Figure 4.9, we show how the execution time of this application varies as the packet size is changed. As can be seen from the figure, execution time can vary by up to a factor of 3 as the packet size is changed. However, the lowest execution time can be achieved from choosing any packet size from a significant range of values.
The packet size suggested by our model can be calculated as follows. Using the parameters from Table 4.2 and the Equation 4.7, we get

\[
k = \sqrt{\frac{108 \times 10^6 (1.73 + 0.054 + 75.85\% \times (1.73 + 0.013))}{2.41}}
\]

\[
= 11798
\]

then

\[
\left\lfloor \frac{B}{k} \right\rfloor = 9154 bytes
\]

From Figure 4.9, we can see that this value of packet size can give the lowest execution time. Thus, our model suggests a packet size that minimizes execution time.

4.4.3.2 Zbuffer Based ISO-Surface Rendering

We test the application ZBUF-2nd here. The size of data file \(B\) is about 145M, the output image size is 512x512, and value of \(\alpha_3\) is 3.42\%. Calculating \(k\) by taking parameters in Table 4.2 and using the Equation 4.11, we get

\[
k = \sqrt{\frac{152729808 (0.1289 + 1.73 + 3.42\% \times (1.73 + 0.7569))}{2.1606}}
\]

\[
= 11726
\]

then

\[
\left\lfloor \frac{B}{k} \right\rfloor = 13025 bytes
\]

The impact of packet size on the execution time is shown in Figure 4.10. Two things can again be observed from this Figure. First, the choice of the packet size can have a large impact on execution time. Second, the packet size suggested by our model minimizes the execution time.
Figure 4.9: Runtime of KNN-1st under Fixed-Frequency Scheme

4.4.3.3 ActivePixel Based ISO-Surface Rendering

The size of data file $B$ is about 145M, the output image size is 2048x2048, and value of $\alpha_3$ is 0.0295. Calculating $k$ by taking parameters in Table 4.2 and using the Equation 4.7, we get

$$k = \left[ \frac{1}{1.5338} \sqrt{\frac{152729808(1.73 + 0.1494 + 0.0295 \times (1.73 + 0.2191))}{1.5338}} \right]$$
\[
\frac{B}{k} = 10997\text{bytes}
\]

The variation in execution time as the packet size is changed is shown in Figure 4.11. Again, we see that execution time can vary significantly and the packet size suggested by our model achieves the lowest execution time.

4.4.4 Evaluation of Fixed-Size Model

We now validate the expressions we have derived for the fixed-size communication model.

4.4.4.1 K-Nearest Neighbor Search

In this experiment, we test the KNN-2nd with \(K = 200\), \(B = 108M\), and \(\alpha_3 = 1\). Using the parameters in Table 4.2 and the Equation 4.17, we get

\[
k = \left\lfloor \frac{108 \times 10^6(0.0872 + 1.73 + 1 \times (1.73 + 0.0127))}{2.4087} \right\rfloor = 12634
\]

then

\[
\left\lfloor \frac{B}{k} \right\rfloor = 8549\text{bytes}
\]

which falls in the experimental optimal range shown in Figure 4.12.

4.4.4.2 Zbuffer Based ISO-Surface Rendering

The size of data file \(B\) is about 597M, the output image size is 2048x2048, and value of \(\alpha_3\) is 1. Using the parameters from Table 4.2 and the Equation 4.21, we get

\[
k = \left\lfloor \frac{610919236(1.73 + 0.1207 + 1 \times (1.73 + 1.0296))}{1.3848} \right\rfloor
\]
then

\[
\left\lfloor \frac{B}{k} \right\rfloor = 13546\text{bytes}
\]

Again, the value suggested here gives the lowest execution time, as validated in Figure 4.13.

4.4.4.3 ActivePixel Based ISO-Surface Rendering

The size of data file \( B \) is about 597M, the output image size is 512x512, and value of \( \alpha_3 \) is 1. Using the parameters from Table 4.2 and the Equation 4.21, we get

\[
k = \left\lfloor \frac{610919236(1.73 + 0.1356 + 1 \times (1.73 + 0.1937))}{1.4552} \right\rfloor
\]

\[
= 39885
\]

then

\[
\left\lfloor \frac{B}{k} \right\rfloor = 15317\text{bytes}
\]

which falls in the experimental optimal range shown in Figure 4.14.

4.5 Summary

One important factor that impacts the performance of applications executing with the coarse-grained pipelined execution model is packet size, i.e., the unit of data transfer from one pipeline stage to another. Within this chapter, we have derived analytical expressions for determining packet size that will achieve the best performance.

Though our work builds on top of the work of Wang et al. on packet size for multi-link communication pipelines, we also addressed two new challenges. First, we had to model computation phases. Second, because of computation, the frequency
and/or the volume of communication between the different communication stages can be different. We have considered two models, fixed-frequency and fixed-size communications, and derived expressions for choosing the packet size.

We have carried out detailed evaluation of our models using three applications, executed with different parameters and datasets. Our experiments have shown that the choice of packet size makes a significant difference in the execution time, and the packet sizes suggested by the model result in the lowest or very close to the lowest possible execution time. With a little extra effort, the models can be applied to applications running on the Grid environment.
Figure 4.10: Runtime of Zbuffer Based ISO-Surface Rendering (ZBUF-2nd) under Fixed-Frequency Scheme
Figure 4.11: Runtime of ActivePixels Based ISO-Surface Rendering (ACTP) under Fixed-Frequency Scheme
<table>
<thead>
<tr>
<th>Packet Size (bytes)</th>
<th>Execution Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>240</td>
<td>&lt; 9</td>
</tr>
<tr>
<td>960</td>
<td>9</td>
</tr>
<tr>
<td>9600</td>
<td>10</td>
</tr>
<tr>
<td>96000</td>
<td>10.5</td>
</tr>
<tr>
<td>960000</td>
<td>11</td>
</tr>
<tr>
<td>9600000</td>
<td>11.5</td>
</tr>
<tr>
<td>96000000</td>
<td>12</td>
</tr>
<tr>
<td>960000000</td>
<td>12.5</td>
</tr>
<tr>
<td>960000000</td>
<td>13</td>
</tr>
<tr>
<td>9600000000</td>
<td>13.5</td>
</tr>
<tr>
<td>96000000000</td>
<td>14</td>
</tr>
<tr>
<td>960000000000</td>
<td>14.5</td>
</tr>
<tr>
<td>9600000000000</td>
<td>15</td>
</tr>
<tr>
<td>96000000000000</td>
<td>15.5</td>
</tr>
<tr>
<td>960000000000000</td>
<td>16</td>
</tr>
<tr>
<td>9600000000000000</td>
<td>16.5</td>
</tr>
<tr>
<td>96000000000000000</td>
<td>17</td>
</tr>
<tr>
<td>960000000000000000</td>
<td>17.5</td>
</tr>
<tr>
<td>9600000000000000000</td>
<td>18</td>
</tr>
<tr>
<td>96000000000000000000</td>
<td>18.5</td>
</tr>
<tr>
<td>960000000000000000000</td>
<td>19</td>
</tr>
<tr>
<td>9600000000000000000000</td>
<td>19.5</td>
</tr>
<tr>
<td>96000000000000000000000</td>
<td>20</td>
</tr>
<tr>
<td>960000000000000000000000</td>
<td>20.5</td>
</tr>
<tr>
<td>9600000000000000000000000</td>
<td>21</td>
</tr>
<tr>
<td>96000000000000000000000000</td>
<td>21.5</td>
</tr>
<tr>
<td>960000000000000000000000000</td>
<td>22</td>
</tr>
<tr>
<td>9600000000000000000000000000</td>
<td>22.5</td>
</tr>
<tr>
<td>96000000000000000000000000000</td>
<td>23</td>
</tr>
<tr>
<td>960000000000000000000000000000</td>
<td>23.5</td>
</tr>
<tr>
<td>9600000000000000000000000000000</td>
<td>24</td>
</tr>
<tr>
<td>96000000000000000000000000000000</td>
<td>24.5</td>
</tr>
<tr>
<td>960000000000000000000000000000000</td>
<td>25</td>
</tr>
<tr>
<td>9600000000000000000000000000000000</td>
<td>25.5</td>
</tr>
<tr>
<td>96000000000000000000000000000000000</td>
<td>26</td>
</tr>
<tr>
<td>960000000000000000000000000000000000</td>
<td>26.5</td>
</tr>
<tr>
<td>9600000000000000000000000000000000000</td>
<td>27</td>
</tr>
</tbody>
</table>

Figure 4.12: Runtime of KNN-2nd under Fixed-Size Scheme
Figure 4.13: Runtime of Zbuffer Based ISO-Surface Rendering (ZBUF-1st) under Fixed-Size Scheme
Figure 4.14: Runtime of ActivePixel Based ISO-Surface Rendering (ACTP) under Fixed-Size Scheme
CHAPTER 5

LANGUAGE AND COMPILER SUPPORT FOR EXPLOITING COARSE-GRAINED PIPELINED PARALLELISM

5.1 Introduction

This chapter presents a compilation system supporting applications executed in the coarse-grained pipelined fashion. As explained in previous chapters, the coarse-grained pipelined execution model is very suitable for deploying distributed data-intensive applications. However, to realize the pipelined execution, there is a number of challenges to be overcome. The computation associated with an application needs to be decomposed into stages, paying attention to both the amount of computation in each of the stages and the volume of communication between them. Moreover, the decomposition decisions are dependent on the environment in which the application will be executed. The algorithms presented in Chapter 3 tackled these two challenges. Another problem is that the code for each of the processing stages needs to be written to iterate over a packet or buffer of data that is received from the previous stage. Finally, we need to exploit shared or distributed memory parallelism which may be available at one or more of the stages.
Thus, we believe that high-level language and compiler support for coarse-grained pipelined execution of data-intensive applications is clearly needed. We are not aware of any previous work in this direction. Recently, some work has been done on runtime support and scheduling for pipelined parallelism [27, 126, 149], but not at the language/compiler level.

This chapter describes and evaluates a compilation system we have been developing for exploiting coarse-grained pipelined parallelism for data-intensive applications. For a given application and its corresponding execution environment, at the preparation stage, a one-pass algorithm will be applied to determine the required communication between consecutive atomic filters. As the core, the system decomposes the application, making use of the algorithms and models presented in Chapter 3 and 4. In the end, the code are generated for each individual filter.

Our language dialect exposes both the pipelined and data parallelism. Our work is based upon the observation that data-driven applications from many domains, including scientific data analysis, data mining, visualization, and image analysis, frequently involve generalized reductions. Thus, much of the processing associated with different data elements is independent. Such applications are, therefore, not only suitable for execution in a pipelined fashion, but it is also feasible to exploit such parallelism through a compiler. For code execution, we use a user-level middleware called DataCutter [29, 27].

We have carried out a detailed evaluation of our current compiler using four data-driven applications. Two of these applications implement algorithms for isosurface rendering, which is one of the common visualization tasks. The two algorithms are based on z-buffer and active pixels, respectively. Another applications is a key data
mining kernel, k-nearest neighbor search, and the last application is *virtual microscope*, which processes digitized images.

The summary of the results obtained from our experiments is as follows. We compared compiler decomposed versions against versions that use pipelined parallelism in a *default* fashion, i.e., copy all data from source nodes to compute nodes, and perform all processing there. The compiler decomposed version was faster by between 10% and 30% for isosurface cases, by nearly 150% for k-nearest neighbors, and by nearly 40% for virtual microscope. The speedups obtained from increasing the width of the pipeline were near-linear, with the exception of a small query on virtual microscope. For k-nearest neighbors and virtual microscope, we also compared the compiler generated versions with manual implementation. For k-nearest neighbors, there was no significant difference in performance. For virtual microscope, the compiler generated version was slower by between 10% and 50%.

The rest of the chapter is organized as follows. Section 5.2 sketches our language dialect. In Section 5.3, we focus on the key compiler analysis phases. A number of issues related to code generation are briefly discussed in Section 5.4. Section 5.5 details the experimental results we have obtained. We summarize in Section 5.6.

### 5.2 Data Parallel Java

We propose to use a data parallel dialect of Java that we had previously used for the same class of applications on a cluster environment [62]. In this dialect of Java, a programmer can specify a computation assuming that all data is available in a flat memory and all computations are done on a single processor.
Interface **Reducinterface** {
  /* Any object of any class implementing */
  /* this interface is a reduction variable */
}

public class **KmPoint** {
  double x1, x2, x3;
}

public class **Kcenter** implements **Reducface** {
  static double [] x1, x2, x3;
  static double[] meanx1, meanx2, meanx3;
  static long[] count;
  static double MES;
  void **Finalize**() {
    for (i=0; i<5; i++) {
      x1[i] = meanx1[i]/count[i];
      x2[i] = meanx2[i]/count[i];
      x3[i] = meanx3[i]/count[i];
    }
  }
  void **Assign**(KmPoint point, int i, double dis) {
    meanx1[i] += point.x1;
    meanx2[i] += point.x2;
    meanx3[i] += point.x3;
    count[i] += 1;
    MES += dis;
  }
}

**Figure 5.1**: Parallel k-means clustering in Data Parallel Java (Part 1)

We now describe the extensions and limitations on Java that our dialect uses. Our constructs include both data parallel constructs, and a special construct to denote the feasibility of performing pipelined processing. For the first group of extensions, we borrow two concepts from object-oriented parallel systems like Titanium [150], HPC++ [31] and Concurrent Aggregates [47].

91
public class Kmeans {

    public static void main(String[] args) {
        RectDomain< 1 > querybox;
        Point< 1 > lowend = ..;
        Point< 1 > hiend = ..;
        RectDomain< 1 > InputDomain= [lowend:hiend];
        KmPoint<1d> Input=new KmPoint[InputDomain];

        while(not_converged) {

            foreach (p in InputDomain) {
                min=9.999E+20;
                for (i=0; i < 5; i++) {
                    int dis = Kcenter.distance(Input[p],i);
                    if( dis < min) {
                        min=temp;
                        minindex=i;
                    }
                }
                Kcenter.Assign(Input[p],minindex,min);
            }
            Kcenter.Finalize();
        }
    }
}

Figure 5.2: Parallel k-means clustering in Data Parallel Java (Part 2)

- Domains and Rectdomains are collections of objects of the same type. Rectdomains have a stricter definition, in the sense that each object belonging to such a collection has a coordinate associated with it that belongs to a pre-specified rectilinear section of the domain.

- The foreach loop, which iterates over objects in a domain or rectdomain, and has the property that the order of iterations does not influence the result of the
associated computations. We further extend the semantics of foreach to include the possibility of updates to reduction variables, as we explain later.

We have introduced a Java interface called Reducinterface. Any object of any class implementing this interface acts as a reduction variable [80]. The semantics of a reduction variable are analogous to those used in version 2.0 of High Performance Fortran (HPF-2) [80] and in HPC++ [31]. A reduction variable has the property that it can only be updated inside a foreach loop by a series of operations that are associative and commutative. Furthermore, the intermediate value of the reduction variable may not be used within the loop, except for self-updates. The goals of these extensions is to give the compiler information about independent collections of objects (with no aliasing between the elements), parallel loops and reduction operations.

The outline of the data parallel Java code for k-means clustering is shown in Figures 5.1 and 5.2. There are several advantages associated with specifying the analysis and processing over datasets in this fashion. The above model assumes that all data is available at a single site. It also assumes that the data is available in arrays of object references, and is not in persistent storage. It is the responsibility of the compiler to locate individual elements of the arrays from disks.

To explain the set of constructs, we use isosurface construction with z-buffers as an example. The outline of the code using this example is shown in Figures 5.3 and 5.4. Isosurface rendering is a key visualization problem [104]. The input to an isosurface construction algorithm is a three-dimensional grid, a scalar isosurface value, and a two-dimensional viewing screen with an angle associated with it. A scalar value is associated with every point in the grid. Typically, it denotes certain physical property such as density or pressure. The goal is to view a surface, as seen from the given
Interface `Reducinterface` {
    // Any object of any class implementing
    // this interface is a reduction variable
}
class CUBE {
    VERTEX[] cube = new VERTEX[8];
    {* member functions ... *}
}
class TRIANGLE {
    TRI_Point[] tri;
    {* member functions ... *}
}
class TRILIST {
    TRIANGLE[1d] data;
    int tri_cnt;
    {* member functions ... *}
}
class ZBUFFER {
    Points[2d] data;
    int winx, winy;
    {* member functions ... *}
}
class ZBUFFER_Reduce Extends ZBUFFER {
    Implements ReducInterface {
        void MergeTwoZbuffers(ZBUFFER recvzbuf) {
            int i, j;
            for (i=0; i<recvzbuf.winy; i++) {
                for (j=0; j<recvzbuf.winx; j++) {
                    if (data[i].row[j].z > recvzbuf.data[i].row[j].z)
                        data[i].row[j] = recvzbuf.data[i].row[j];
                }
            }
        }
    }
}

Figure 5.3: Outline of Z-buffer Based Iso-surface Extraction Code (Part 1)
public class isosurface {
public static void main(String arg[]) {
  float iso_value;
  RectDomain<1> CubeRange = [min_n:max_n];
  InputCube InputData = new InputCube(CubeRange);
  TRI_LIST tri_list = new TRI_LIST();
  Point<1> p;
  int runtime_define_num_packets;
  int n = (max_n - min_n)/runtime_define_num_packets + 1;
  RectDomain<1> PacketRange = [1:runtime_define_num_packets];
  RectDomain<1> EachPacket = [1:n];

  ZBUFFER.Reduce zbuffer = new ZBUFFER(screen.winx,screen.winy);
  ZBUFFER[1d] zbuffer1 = new ZBUFFER(PacketRange);

  Pipelined_loop(b in PacketRange) {
    zbuffer1[b] = new ZBUFFER(screen.winx,screen.winy);
    foreach (p in EachPacket) {
      * extract triangles *
      InputData.getCube(p).ISO_SurfaceTriangles(iso_value, tri_list);
    }

    RectDomain<1> TriRange = [1:tri_list.size()];
    foreach (p in TriRange) {
      * process each triangle *
      * and compute an edge list *
    }

    * Update a Z-Buffer *
    edgetable.PZBUFSIZEetable(zbuffer1[b], screen);

    * Merge into a single z-buffer *
    zbuffer.Merge_two_zbuffers(zbuffer1[b]);
  }
}
}

Figure 5.4: Outline of Z-buffer Based Iso-surface Extraction Code (Part 2)

viewing angle, which captures the points in the grid where the scalar value matches the given isosurface value. Such a surface is referred to as the isosurface.
The following steps are used. The grid is processed as a set of cubes, where a scalar value is associated with each corner point. If the isosurface value is greater or lower than the values at each of the eight end points of the cube, it is assumed that the isosurface does not pass through this cube. Otherwise, a set of triangles is generated to approximate the surface passing through the cube. Next, each such triangle is transformed from the original grid coordinates to the viewing coordinates. Then, it is projected to a two-dimensional image plane (the screen) and clipped to the screen boundaries. Subsequently, the triangles (polygons after the above processing) are accumulated onto a z-buffer. A z-buffer stores a color and distance associated with each pixel or point in the 2-dimensional viewing screen. As polygons are accumulated onto a z-buffer, the color associated with the least distance is chosen.

Overall, this algorithm is well suited for execution in a pipelined fashion. The processing associated with different cubes and triangles is independent. Moreover, after triangles are extracted from a set of cubes, these triangles can be processed independent of the other cubes. Similarly, after triangles are processed to extract polygons, these polygons can be accumulated independent of other triangles or polygons. The accumulation function is associative and commutative, i.e., the final result obtained does not depend upon the order in which triangles or polygons are processed.

In our isosurface example, one-dimensional Rectdomains are used to denote collections of cubes, triangles, or polygons. The foreach loop is used to imply that the order of processing each cube, triangle, or polygon, does not impact the correctness of the result. Reducinterface is to denote that accumulation onto a z-buffer involves associative and commutative operations.
The goal of the above three extensions is to give the compiler information about independent collections of objects (with no aliasing between the elements), parallel loops and reduction operations. Though it is possible that advanced compilation techniques may be able to infer some or all of this information automatically, for the purpose of focusing our research effort on issues related to filter decomposition and pipelined execution, we have chosen an experimental framework that explicitly provides this information.

The construct `PipelinedLoop` is used to capture the processing associated with each `packet`. The processing associated with each packet is assumed to be independent, except for updates to any object that belongs to a class that implements the ReducInterface. The size of each packet can be chosen at runtime. This is expressed through the prefix `runtime_define` to the variable `num_packets`.

In the code shown in Figures 5.3 and 5.4, the set of input cubes is divided into `runtime_define_num_packet` packets. Each such packet is processed independently, and a z-buffer is allocated and updated for each packet. Subsequently, the z-buffers computed from each of the packets are merged together. The z-buffer onto which all intermediate z-buffers are merged is defined to implement the ReducInterface, denoting that this merge is associative and commutative.

Though the above constructs make pipelined and data parallelism known to the compiler, the compiler input code is much simpler in several ways than the filter-stream code that DataCutter requires. First, the boundaries between the filters are not specified in the input code. These can be chosen by the compiler with the knowledge of the target execution environment. Second, the input code assumes a global address space. In comparison, the code for each filter that our compiler generates
involves iterating over a buffer (a linear set of bytes) that is received from another filter. Third, the programmer does not need to worry about what values will be communicated if a particular boundary is used, instead, this is determined by the compiler.

5.3 Compiler Analysis

This section focuses on various phases of analysis performed by our compiler. We give an overview of the compilation problem, describe our algorithm for communication analysis, state our cost model, and then describe an algorithm for filter decomposition.

5.3.1 Overview of the Compilation Problem

We focus on a single loop that iterates over a set of packets. Within this loop, there may be several foreach loops, including one or more loops that update a reduction object. We have a pipeline of available computing resources. The first element in this pipeline is always the data repository or data source, i.e., the computing unit where data is available. The last element in this pipeline is where the final results are desired. We denote the computing units in the pipeline by $C_1, \ldots, C_m$. The connection between units $C_i$ and $C_{i+1}$ is denoted by $L_i$.

Given such a loop over a set of packets and a computing environment described above, we want to compute a mapping between the computation in the loop to the computing units in our pipeline. Stated in terms of our target runtime system DataCutter, we want to decompose the computation into a set of $m$ filters, and place one filter on each unit in the pipeline.
Several problems need to be addressed here. First, we need to select a set of candidate points within the loop, each of which can serve as a filter boundary. Second, we have to choose \( m - 1 \) filter boundaries among the candidate filter boundaries. The compiler may have to extract information from the program to make the above decision. Finally, given the \( m - 1 \) filter boundaries, we need to generate code for each of the \( m \) filters. We have already discussed the filter decomposition problem in Chapter 3, and will address the first and third problem in this section.

We currently consider three types of candidate filter boundaries: 1) Start and end of a foreach loop, 2) A conditional statement, either inside or outside a foreach loop, and 3) Start and end of a function call within a foreach loop. We also assume that any loop that is not a foreach loop must be completely inside a single filter. In our target class of applications, such loops typically did not enclose a significant amount of computation, and therefore, this restriction did not limit us in anyway.

We represent the set of candidate filters boundaries and the code between those by a candidate filter boundary graph. The nodes in this graph are the candidate filter boundaries, with the exception of a start node that pre-dominates all other nodes, and an end node that post-dominates all other nodes. An edge in this graph connects two candidate filter boundaries that are adjacent, i.e., control in the original code can flow from the first boundary to the second boundary, without reaching any other candidate filter boundary. If there are candidate filter boundaries within a foreach loop, we perform loop fission and create separate foreach loops. This ensures that there are no candidate boundaries inside a foreach loop. With such loop fission, and because we require any other kind of loop to be completely enclosed inside a filter,
the candidate filter boundary graph is always acyclic. A flow path in this graph is
defined to be any path from the start node to the end node.

Given any flow path in our graph, we want to select \( m - 1 \) filter boundaries. Our
objective in selecting these \( m - 1 \) filter boundaries is to minimize the total execution
time for the pipeline. For this purpose, the compiler needs to extract the set of values
that will be communicated if a particular candidate filter boundary is selected. The
same information is also required for code generation. Analysis for this is presented
in the next subsection.

Note that in our current work, we only perform static decomposition of the problem, i.e., the mappings of the tasks to the computing units is not changed during the
execution. While this is reasonable for our current set of target applications, it could
limit performance in some cases. Considering dynamic decomposition of the problem
will be a topic for our future work.

### 5.3.2 Required Communication Analysis

We now present a one-pass algorithm for determining the required communication
across all candidate filter boundaries.

Consider two consecutive candidate filter boundaries, \( f_1 \) and \( f_2 \), and let \( b \) be the
code between these. We use the following terms. \( Gen(b) \) denotes the set of values
that have been defined in the code section \( b \) and are still live at the end of \( b \). \( Cons(b) \)
is the set of values that are used in \( b \), and are not defined in \( b \). \( ReqComm(f_1) \) and
\( ReqComm(f_2) \) denote the communication required at potential filter boundaries
\( f_1 \) and \( f_2 \), respectively. In our model, all communication occurs only between two
consecutive filters. Then, we have

\[ \text{ReqComm}(f_1) = \text{ReqComm}(f_2) - \text{Gen}(b) + \text{Cons}(b) \]

At the end of the last filter in the candidate filter boundary graph, \( \text{ReqComm} \) is initialized to be the null set. Thus, using the above equation, and computed values of \( \text{Gen}(b) \) and \( \text{Cons}(b) \) for code segment between each pair of consecutive filter boundaries, we can determine \( \text{ReqComm} \) for each candidate filter boundary using just a single pass over the candidate filter boundary graph.

After computing these terms, we select only a subset of the filter boundaries. An important observation is that the term \( \text{ReqComm}(f_1) \) computed as stated above still holds correct even if a filter boundary is not inserted at \( f_2 \). To explain this, we consider a series of three consecutive candidate filters in our graph, \( f_1, f_2, \) and \( f_3 \), with code segments between them denoted as \( b_1 \) and \( b_2 \). \( \text{ReqComm}(f_2) \) is computed using \( \text{ReqComm}(f_3) \) and the code segment \( b_2 \). Subsequently, \( \text{ReqComm}(f_1) \) is computed using \( \text{ReqComm}(f_2) \) and the code segment \( b_1 \).

Now, suppose, after our analysis, we insert filter boundaries only at \( f_1 \) and \( f_3 \), i.e., the code segments \( b_1 \) and \( b_2 \) are put in the same filter. The computed value of \( \text{ReqComm}(f_1) \) is still correct. This is because any value required for correct execution for the code in the segment \( b_2 \) is either generated inside the code segment \( b_1 \), or in code segment before \( f_1 \). In the former case, it does not need to be communicated any more, and in the latter case, it is captured in \( \text{ReqComm}(f_1) \) as computed originally.

Computing \( \text{ReqComm} \) sets requires the terms \( \text{Gen} \) and \( \text{Cons} \) for each code segment that falls between two consecutive filter boundaries. We use a simple one pass algorithm that assumes structured control flow. Thus, the entire analysis for determining \( \text{ReqComm} \) at each filter boundary can be performed using only a single pass.
over the program. Though our current implementation is in an off-line compiler, the analysis of the type described here is likely to be implemented in Just-In-Time (JIT) compilers. Therefore, the efficiency of analysis is important.

```
Analyze(Code Segment b) {
    Initialize Gen(b) and Cons(b) to null
    for all statements s in b in the reverse order {
        if s is an assignment statement {
            Gen(b) = Gen(b) + LHS(s)
            Cons(b) = (Cons(b) - LHS(s)) + RHS(s)
        }
        else if s is a conditional block {
            Analyze(s)
            Cons(b) = Cons(b) + Cons(s)
        }
        else if s is a loop {
            Analyze(s)
            Insert_Loop_Range(Gen(s))
            Insert_Loop_Range(Cons(s))
            Gen(b) = Gen(b) + Gen(s)
            Cons(b) = (Cons(b) - Gen(s)) + Cons(s)
        }
    }
}
```

Figure 5.5: Computing Generated and Consumed Sets Using a Single Pass

The analysis performed on a code segment that falls between two consecutive filter boundaries is shown in Figure 5.5. We view the code segment b as a sequence of statements. A statement can be either an assignment statement, a conditional statement, or a loop. Our algorithm traverses this sequence of statements in the reverse order. Initially, both Gen(b) and Cons(b) are initialized to be null sets.
For an assignment statement $s$, $LHS(s)$ denotes variables modified by $s$ and $RHS(s)$ denotes all variables used by $s$. A variable that is given a value in this statement is added to the $Gen(b)$ set and removed from the $Cons(b)$ set. Any variable that is used in this statement is added to the $Cons(b)$ set. We assume that (potentially conservative) alias information is available. In updating the $Gen(b)$ set, we use the \textit{must alias} information, i.e., a variable can only be added to this set if it is definitely defined. In updating the $Cons(b)$ set, we use the \textit{may alias} information, i.e., any variable that could potentially be used is added to this set.

A conditional statement $s$ inside the segment $b$ is handled as follows. First, we independently analyze the set of statements inside the conditional block. The set of variables $Cons(s)$ is added to the set $Cons(b)$. However, the set $Gen(s)$ cannot be added to the set $Gen(b)$, since the statements in the block $s$ are enclosed in a conditional. Note that a variable that is both defined and used in the block $s$ does not get added to the $Cons(b)$ set.

For handling loops, we assume that a loop cannot be executed for zero iterations. We compute $Gen(s)$ and $Cons(s)$ set for the loop body. If the variables in these sets are accessed using a function of the loop index, we replace these variables by rectilinear sections, derived from loop bounds. Then, the set $Gen(s)$ is added to the set $Gen(b)$ and removed from the set $Cons(b)$. The set $Cons(s)$ is added to the set $Cons(b)$.

There are two important aspects of our analysis that are not shown in Figure 5.5. First, our analysis is applied interprocedurally. The main issue in doing this is changing variable names from actual parameters to formal parameters and vice-versa. Note
that we perform context-sensitive analysis, i.e., a procedure included in multiple code segments is analyzed independently each time.

Second, in storing and manipulating Gen and Cons sets, we use rectilinear sections, whose bounds may only be available symbolically. We also keep track of fields of classes and handle nested classes.

5.4 Code Generation Issues

This section focuses on some of the code generation issues that were addressed in our compiler implementation.

After the set of boundaries has been selected, the compiler needs to generate code for each filter. In our model, each filter has one input stream, with the exception of the filter that reads from the data source itself. Also, each filter has one output stream, with the exception of the filter that generates the final results. The following two challenges arise. First, we need to decide how the data communicated from one filter to another is packed. Second, we need to generate code for each filter that receives and unpacks from the input stream, performs the computations, and then packs data for the output stream.

As described in Section 5.3.2, our communication analysis phase generates the ReqComm sets, which denote the communication required between each pair of consecutive filters. Let the code inside a filter \( f \) comprise of code segments \( b_1, \ldots, b_k \) between consecutive candidate filter boundaries. By processing the sets \( Cons(b_1), \ldots, Cons(b_k) \), we can compute the set \( Cons(f) \), which denotes the values that are consumed inside the filter \( f \).
Figure 5.6: Examples of Unpacking: Instance-wise (a) and Field-wise (b)
\_t = new _T();
Inptr = Inbuf.getPtr();
memcpy(&count, Inptr, sizeof(int));
Inptr += sizeof(int);

{ /* unpacking */
    for(int i=0; i<count; i++) {
        off = 0;
        memcpy(&(_t -> x), Inptr+off+sizeof(int)+0+i*(sizeof(t.x)+sizeof(t.z)), sizeof(int));
        off = 0;
        memcpy(&(_t -> z), Inptr+off+sizeof(int)+sizeof(t.x)+i*(sizeof(t.x)+sizeof(t.z)), sizeof(int));
    }
    for(int i=0; i<count; i++) {
        Outptr += sizeof(int); /* save for size info */
    }
    size=Outptr-head;
    memcpy(head, &size, sizeof(int));
    Outptr += sizeof(int); /* save for size info */
    off = 0;
    for(int k=0; k<1; k++) {
        memcpy(&size, Inptr+off, sizeof(int));
        off += size;
    }
    for(int i=0; i<count; i++) {
        memcpy(Outptr, Inptr+offset+sizeof(int)+0+i*(sizeof(t.y)), sizeof(int));
        Outptr += sizeof(int);
    }
    size=Outptr-head;
    memcpy(head, &size, sizeof(int));
}

Figure 5.7: Examples of Unpacking: Complete Generated Code
Initially, we focus on how the code for each filter is generated. Consider the code in the input language that corresponds to this filter. We primarily focus on collections of objects that are read or written in a foreach loop in this code. Let $T$ be a class whose collection is accessed inside a foreach loop. We initially determine all fields of $T$ that are read or written in the code for this filter. Then, we create a new class $\mathcal{T}$ which includes only these data fields. Before executing an iteration of the foreach loop, we allocate an object of type $\mathcal{T}$. Then, any fields of $\mathcal{T}$ that are communicated from the previous filter are unpacked from the received packet and copied to the allocated object.

Next, we discuss how our compiler decides to pack the elements within the set $\text{ReqComm}$. Suppose, for example, the class $T$ has two fields, $x$ and $y$, that are communicated, and each packet includes $\text{count}$ objects of type $T$. In this simple case, there are two ways in which the packet can be arranged. The first will be

$$< \text{count}, t_1.x, t_1.y, \ldots, t_{\text{count}}.x, t_{\text{count}}.y >$$

and is referred to as the instance-wise method. The second will be

$$< \text{count}, \text{offset1}, t_1.x, \ldots, t_{\text{count}}.x, t_1.y, \ldots t_{\text{count}}.y >$$

and is referred to as the field-wise method. The value $\text{offset1}$ denotes the offset for getting the first instance of the field $y$, and is stored for easing the unpacking. For more general examples, where multiple classes, more than two fields, and/or nested classes are involved, we can use a combination of these two basic methods.

If a field is included in the set $\text{ReqComm}$ at the boundary of two filters $f_1$ and $f_2$, it could be used in $f_2$, or in any of the filters following $f_2$. In the former case, the field belongs to the set $\text{Cons}(f_2)$, and in the latter case, it does not. If two
or more fields are used in $f_2$, then it is clearly more efficient to pack them in the instance-wise fashion. However, if one field is used in the receiving filter, and another is packed again and sent to the next filter, then the field-wise method turns out to be more efficient. In Figure 5.6, we show unpacking code for instance-wise and field-wise methods, respectively.

Thus, for packing fields of objects, we use the following approach. For each filter that has an output stream, we sort the fields of classes by the first filter whose $Cons$ set they belong to. The fields that are used for the first time in the same filter are packed in the instance-wise fashion. For the fields that are used for the first time in different filters, we use the field-wise fashion, sorting by the order in which they are first read.

Figure 5.7 shows code for a filter that involves both unpacking and packing. Here, the input includes three fields $x$, $y$ and $z$ from class $T$, among which $x$ and $z$ are used by the current filter, thus packed together in the instance-wise fashion. Because the field $y$ is used by later filters, so it is packed separately in the field-wise fashion. After processing, the current filter creates a new data-structure $s_list$, whose $x$ field is required by one of the following filters. Together with $T.y$, $s_list.x$ is packed as field-wise and forwarded.

### 5.5 Experimental Results

The techniques described in this chapter have been implemented in a prototype compiler, which is based on the Titanium [150] infrastructure. This section reports the results we obtained from our current compiler. Initially, we describe the applications
we used, the versions we created, and the configurations in which the applications were executed. Then, we present the results obtained from each of the applications.

5.5.1 Applications

We have used four data-driven applications. Two of these applications implement algorithms for *isosurface rendering*, which is one of the common visualization tasks. The third algorithm is a key data mining kernel, k-nearest neighbor search, and the last application is *virtual microscope*, which processes digitized images.

We now describe these applications in more details. The basic isosurface rendering problem was described in Section 5.2. Z-buffer and active pixels are two of the popular algorithms used for this purpose [104]. The initial steps, i.e. extraction of triangles and transformations on triangles, are identical in these two algorithms. However, they differ significantly in how the triangles or polygons are projected to a screen. In the z-buffer approach, an image is created corresponding to a set of triangles in a packet or on a processor. Then, these z-buffers or images are merged to create the final image. The active pixel algorithm removes some of the computational and memory inefficiencies associated with the z-buffer algorithm. Essentially, it uses a sparse representation of the dense z-buffer, and avoids allocating, initializing, or communicating a full z-buffer.

We implemented these two algorithms using our language dialect. In our discussion, they are referred to as *z-buffer* and *active-pixel*, respectively. The code in our input language dialect was nearly 1400 lines for each of these. Our third application is k-nearest neighbor search, referred to as *knn*. It is one of the basic data mining problems [77]. Here, the training samples are described by an n-dimensional numeric
space. Given a new point, the goal is to find the $k$ training samples that are closest to the new point. The final application we used is virtual microscope [61], denoted by \texttt{vmScope}. A virtual microscope is designed to interactively view and process digitized data arising from tissue specimens. The input data is a high-resolution digitized image. This application emulates the usual behavior of a physical microscope, including continuously moving the stage and changing magnification or resolution. The code in our language dialect for both \texttt{knn} and \texttt{vmScope} was relatively small, i.e., under 200 lines.

5.5.2 Experiment Design

Our overall goal was to demonstrate that compiler generated pipelined code is efficient, and can effectively use a pipeline of computing units. In the long run, we expect that pipelined parallelism can be exploited in wide-area networks. However, this is going to require high bandwidth networks and certain level of quality of service support. Recent trends are clearly pointing in this direction, for example, the five sites that are part of the NSF funded Teragrid project expect to be connected with a 40 Gb/second network [125]. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single cluster. We assumed that input data is available on only a subset of the nodes, and final results are required or viewed on a single node. The cluster we used had 700 MHz Pentium machines connected through Myrinet LANai 7.0.

We used three different pipeline configurations. In the first configuration, the data is available at a single node, another node is available for computations, and the
results are required at yet another node. This configuration is designated as the 1-1-1 configuration. In the second configuration, data is available at 2 nodes in the cluster, another 2 nodes are available for computations, and the final results are required at another node. This configuration is designated as the 2-2-1 configuration. In the third configuration, referred to as the 4-4-1 configuration, data is available at 4 nodes, another 4 nodes are available for computations, and the final results are required at another node. These three configurations are also referred to as configurations with pipeline widths of 1, 2, and 4, respectively.

To evaluate our compiler, we created two or three versions for each of our applications. One version is denoted as Default, and reflects a simple way of using pipelined parallelism. Here, the nodes hosting the data only read the data and transmit it to the compute nodes. The final results are generated on these compute nodes, and then copied on the node where they are viewed. Our experiments contrasted these versions with versions in which more intelligent decomposition is performed by the compiler. Such latter versions are referred to as Decomp. Typically, these version performed more computations on either or both of the data source nodes and the node where the results were required, and reduced the volume of data that had to be communicated between the nodes. For knn and vmscope, we also compared the performance of compiler generated code with the manually written DataCutter codes, where similar decomposition was performed. For these two applications, the compiler and manual versions are referred to as Decomp-Comp and Decomp-Manual, respectively.
Figure 5.8: Results from Z-Buffer Based Isosurface Rendering, small dataset

Figure 5.9: Results from Z-Buffer Based Isosurface Rendering, large dataset

112
Figure 5.10: Results from Active Pixel Based Isosurface Rendering, *small* dataset

Figure 5.11: Results from Active Pixel Based Isosurface Rendering, *large* dataset
5.5.3 Isosurface Algorithms

We now describe the results we obtained from the two isosurface algorithms, z-buffer and active-pixels. We used two datasets to evaluate both of these algorithms. These datasets were generated by an environmental simulator ParSSim [17] and were previously used by Beynon et al. in the context of DataCutter [28]. These datasets comprised grid data for 10 time-steps, and were 1.5 GB and 6 GB, respectively. The two datasets are referred to as small and large datasets. We report experiments on processing a single time-step, the data corresponding to which is 150 MB and 600 MB, respectively.

As we stated previously, we created two versions, Default and Decomp, for both the algorithms. We did not have access to comparable manual versions. The Default version reads data in packets, each of which comprises a set of cubes, from the nodes hosting data and copies them to compute nodes. Compute nodes receive data in packets, and perform all processing. In the Decomp version, the compiler places some preprocessing of the cubes at the nodes where the data is. In particular, a loop that checks whether all corners of a cube have an isosurface value greater or lower than the specified isosurface value is placed at data nodes. Besides reducing the amount of computation that needs to be performed on compute nodes, this reduces the volume of data that needs to be copied from data nodes to compute nodes.

Figure 5.8 shows experimental results from z-buffer, using the small dataset. We show results on three configurations, 1-1-1, 2-2-1, and 4-4-1. Compiler based decomposition consistently gives 20% improvements on all three configurations. By increasing the width of the pipeline, we get significant speedups. With a width of 2,
the compiler decomposed version gives a speedup of 1.92, and with a width of 4, the speedup is 3.34.

Figure 5.9 shows experimental results from \texttt{z-buffer}, using the larger dataset. Again, the performance improvements from the compiler decomposed versions are between 20\% and 25\%. For the compiler decomposed version, a speedup of 1.99 is obtained with the pipeline width of 2, and a speedup of 3.82 is obtained with the pipeline width of 4.

The results from \texttt{active-pixel} algorithm are presented in Figures 5.10 and 5.11. We see almost the same trends. The \texttt{Decomp} versions outperform the \texttt{Default} versions by between 15\% and 25\%. The speedups obtained from increasing the width of the pipeline are quite close to linear.

5.5.4 k-nearest Neighbor

For our experiments with \texttt{knn}, we used a 108 MB dataset comprising 4.5 million three-dimensional points. We used two test cases, with values of $k$ being 3 and 200.
Figure 5.13: Results from k-nearest neighbors, k = 200

We used three versions. In the Default version, all input data is forwarded in packets to compute nodes. In both Decomp-Comp and Decomp-Manual versions, the volume of data to be transmitted is reduced. These two versions only differed in how the data received in a packet was iterated on.

The results are presented in Figures 5.12 and 5.13. In both the test cases, the performance difference between Decomp-Comp and Decomp-Manual versions is insignificant. The Default version is consistently slower by almost 150% in all cases. This is because of the large volume of data that is transferred by this version. All versions achieve good speedups as the width of the pipeline is increased. With the Decomp-Comp version and $k = 3$, the speedups are 1.89 and 3.38 with pipeline width of 2 and 4, respectively. Similarly, with $k = 200$ on the same version, the speedups are 1.87 and 3.35, respectively.
Figure 5.14: Results from Virtual Microscope, Small Query

Figure 5.15: Results from Virtual Microscope, Large Query
5.5.5 Virtual Microscope

The last application we experimented with is vmscope. We used a 800 MB image for our experiments. A query specifies a rectangular region and a sub-sampling or resolution factor for the desired output. We used two queries. The first, referred to as the small query, required a $512 \times 512$ output with a sub-sampling factor of 2. The second, referred to as the large query, required a $2048 \times 2048$ output with a sub-sampling factor of 4.

Like knn, we generated three versions. Decom-Manual and Decom-Comp versions reduced the volume of data that had to be copied from the data sources to compute nodes. These two versions again differed on how the data received in a packet is iterated on.

The results from the small and large queries are presented in Figures 5.14 and 5.15, respectively. For small query, the amount of data processed is quite small. It is hard to achieve a good load balance between the different data nodes and the different compute nodes. As a result, the speedups are very limited. With a pipeline width of 1, the Decom-Comp version is slower than the Decom-Manual version by nearly 20%, but is faster than the Default version by nearly 40%. The same trend is seen between these versions as the pipeline width is increased, but the relative differences are smaller. This is because load imbalance is the biggest factor contributing to the performance in these cases.

The main difference between compiler generated and manual code is as follows. In this application, only every $\text{subsample}^{th}$ element along each dimension is picked from the original image. In the compiler generated version, a conditional is used for this purpose whereas the manual version simply uses a stride in reading from input
buffers. Since the application does not involve a lot of computation, this made a significant difference in the performance.

For large query, a good load-balance could be obtained, and therefore, good speedups were obtained. However, because of a larger subsampling factor, the differences between manual and compiler versions are even larger. The compiler decomposed version is consistently faster than the default version by nearly 40%.

5.6 Summary

In this chapter, we have addressed a number of design and algorithmic issues in providing high-level programming support for coarse-grained pipelined parallelism. Our basic premise has been two-fold. First, we believe that the availability of data repositories and data collection instruments on the internet is making this form of parallelism desirable and feasible for a variety of data-driven applications. Second, we believe that language and compiler support will be crucial for exploiting such parallelism, as manual decomposition of the processing is time-consuming and error-prone.

The results from our current compiler have been very encouraging. We have shown that compiler-based decomposition of applications can achieve significantly better performance as compared to what we considered the default way of using pipelined parallelism. Note, however, that it is currently not easy to implement such default pipelined parallelism, either. Most often, all data is downloaded on compute nodes, and then processing is performed. This is generally much slower than what we used as the baseline in our experiments. By using the support for transparent copies in
our target runtime system DataCutter, we have also achieved good speedups from increasing the width of the pipeline.
CHAPTER 6

LANGUAGE AND COMPILER SUPPORT FOR ADAPTIVE APPLICATIONS

6.1 Introduction

In recent years, there has been much interest on adaptive or autonomic computing. Adapting applications or programs has been studied by many, and a variety of solutions have been proposed, including those through new algorithms [101], runtime/middleware [87, 42, 15, 135, 119, 124], and language/compilers [58, 53].

In this chapter, we expand our system to support program adaptivity for data-driven applications where users have significant flexibility in the output they desire. At the same time, there are other constraints, such as the need for real-time response or limit on the consumption of certain resources, which are more crucial. For example, while visualizing simulation data, the output can be viewed at different granularities, i.e., the output image can be $512 \times 512$, $1024 \times 1024$, or $2048 \times 2048$, etc. While it is preferable to view the image at the finest level, constraints such as the need for real-time response or interactivity, or limit on the battery consumption, could be more important. Examples of applications where the users can have some flexibility in the output arise in multimedia (including video/audio streaming applications), image
processing, scientific visualization, data mining/analysis on simulation data, among others.

Similarly, there are several environments in which one or more resource constraints could be crucial. Battery life, network bandwidth, and response time can all be important in a mobile or sensor-network based environment. Memory can often be a significant constraint when computing on embedded systems [30]. Meeting the real-time constraint for processing the data is a necessity in the streaming data model [19]. While visualizing or analyzing data from a remote data repository, it is very desirable to maintain interactivity. In all such cases, application users want to achieve the most accurate or most fine-grained output from the applications, while still meeting the resource or time constraints.

This chapter proposes a language model for specifying applications which have flexibility in their output. The key idea in our language extensions is to have the programmers specify adaptation parameters, i.e. the parameters whose values can be varied within a certain range. The constraints that need to be observed are also specified. In our current implementation, a data parallel dialect of Java is taken as the basis [55]. A special class is defined and it is required that all the adaptive parts of the application must extend this class.

For supporting program adaptation, we use a hybrid methodology, which combines compile-time analysis with runtime feedback. A program analysis algorithm states the execution time of an application component as a function of the values of the adaptation parameters and other runtime constants. These constants are determined by initial runs of the application in the target environment. Once these constants are known, we know which adaptation parameters can be modified and by how much to
achieve a given change in execution time, or to maintain the same execution time for a given change in available resources. To summarize, the main contributions of this work are as follows.

- We have designed a small set of language extensions which can easily expose the adaptive features of an application.
- We suggest a novel way to achieve program adaptation by taking advantages of both static analysis and runtime information.
- A new static analysis algorithm is presented, which states the execution time of a program component as a function of adaptation parameters and other runtime constants.
- This adaptation approach is implemented and evaluated for data-intensive grid-based applications. The experimental results show the efficacy of the approach.

The rest of the chapter is organized as follows. The description of our target applications is presented in Section 6.2. Section 6.3 describes the language extensions for supporting adaptation. In Section 6.4, we present the system architecture, and the compiler and runtime support. The experimental results are discussed in Section 6.5. We summarize in Section 6.6.

6.2 Target Applications

We consider the following examples:

Video/Audio Streaming Applications: Video, which usually consumes a large portion of the bandwidth required by a streaming multimedia application, is also
flexible in the bandwidth needs. A distributed video stream server [119] needs to maintain a specific frame rate and can do so by altering the levels of fidelity for the video data. Similarly, distributed visual tracking systems [99] need on-the-fly adaptation as the available bandwidth varies.

**Real-Time Visualization:** Large scale volume rendering involves an increasing amount of data, which makes interactive rendering infeasible. With the advanced visualization hardware support and the recent more efficient rendering algorithms, the speed of volume rendering has increased significantly. Yet, the average size of a volumetric dataset continues to grow even faster [101]. To tackle this problem, one can sacrifice image quality to some extent for speed, by a flexible level of detail control. Computer assisted medical surgery [34] and real time volume navigation [83] are good examples in this category. Likewise, a map display application [44] will be more attractive if it can adapt to system variations and offer different data services taking into account preferences of various users.

**Hot-list Query Processing:** Imagine a nationwide company like Wal-Mart, where transactions are generated every moment at each local store site, forming streaming data [72]. The manager at the headquarter wants to query top 100 selling items at each time interval. So, the problem is of determining frequently occurring values from a stream, sub-streams of which arrive from different places. One option for solving this problem is to communicate all sub-streams to a single location, and then apply the original algorithm. However, bandwidth limitations may not allow this. An alternate solution will be to create a summary structure for each sub-stream, and then communicate these to a central location. We can expect that larger the size of the summary, more accurate the final results will be. Thus, the number of
frequently occurring values at each sub-stream is the *adaptation* parameter used in this application.

**Image Processing:** Virtual Microscope [13] is an application for providing a realistic emulation of a high power light microscope. In this application, users can specify a view screen and an acceptable range of resolution. Within a given time period, the server should be able to provide an image for view, while the resolution of the image could be varied depending on the resources availability. That is, the image quality might be traded for performance when necessary. Map viewer [66] is yet another adaptive image processing application.

**Data Mining/Analysis on Simulation Data:** In scientific simulations, simulation data is continuously generated at each time-step for a grid of points. A variety of insights can be gained by analysis or mining of this data [137]. Due to the enormous volume, it is impossible to store information for each point on the grid at every time-step. Therefore, introducing adaptability to the processing of such data is a necessity. One can reduce the data quantity by down-sampling the time dimension, space dimension, or both.

### 6.3 Language Extensions

This section presents the language constructs we have introduced to support adaptivity.

We use a digital image retrieval application, the virtual microscope [13], to illustrate the constructs. In this application, the user specifies a portion of an image that needs to be viewed at a remote location. This image can be retrieved and transferred over the network at one of many different resolution levels. Thus, the resolution level
class Adapt_Spec {
    String constraints;
    String adapt_vars;
    Adapt_Spec(String c, String v) {
        constraints = new String(c);
        adapt_vars = new String(v);
    }
}
}

interface Reducinterface {
    // Any object of any class implementing
    // this interface is a reduction variable
}

class Image {
    VMPixel[2d] data;
    RectDomain<2> size;
    Image(RectDomain<2> size) {
        data = new VMPixel[size];
    }
}

class Image_Reduce extends Image implements Reducinterface {
    void Merge(Image i) {... ... }
}

class VMScope extends Adapt_Spec {
    ...
    public static RectDomain<2> VMslide = [lowpoint : hipoint]; (1)
    public static Image VScope = new Image(VMslide); (2)
    ...
}

Figure 6.1: Illustration of the language construct extensions: Code segments from the Virtual Microscope application (Part 1).

is a parameter whose value can be adjusted, subject to other constraints such as the response time and network bandwidth. The code for this application is shown in Figures 6.1 and 6.2.
public void run() {
    ...
    pipelined_loop(b in part_range){
        ...
        foreach( p in eachpart){
            if(VScope.data[p].is_sampled(lowend, sub_factor))
            {
                q = (p - lowend)/sub_factor; // mapping
                Output_t[b].data[q].Assign(VScope.data[p]);
            }
        }
    }
    foreach(b in part_range){
        Output.Merge(Output_t[b]);
    }
}
public static void main(String[] args)
{
    ...
    Image.Reduce Output;
    while(true){
        ...
        constraints = "RESP_TIME:0ms:30ms";
        adapt_vars = "sub_factor:1:32";
        part_range = [1 : runtime_define_num_partition];
        Output_t = new Image(part_range);
        Output = new Image(Outputdomain);
        VMQuery = new VMScope(constraints,adapt_vars,
                                lowend,hiend,sub_factor,Output,
                                part_range,Output_t);
        VMQuery.run();
    }
}

Figure 6.2: Illustration of the language construct extensions: Code segments from the Virtual Microscope application (Part 2).
6.3.1 Adaptation Constructs

Two extensions are added to support program adaptation in addition to the extensions made in chapter 5.2. The class `Adap_Spec` is a super class to all classes for which adaptation is desirable. All the adaptive classes must extend this class, and include `super()` in its own constructor. This requirement helps expose the adaptive classes to the compiler, and also ensures that both the constraints and the adaptation variables are specified.

The definition of this class is shown in Figure 6.1. To define a constraint, the type should be specified, followed by the lower and upper bound of its value. As an example, in the line(7) of Figure 6.2, it is stated that the response time should be less than 30 ms. To define an adaptation variable, one needs to state the variable name and the range of the acceptable values. The line(8) is showing that the application developer can accept the value of `sub_factor` within range [1,32].

One obvious question is, “What is the part of the code on which the constraint applies?” For this purpose, each adaptive class must provide a `run` function which encloses a unit of work, e.g. the amount of computation and communication needs to be finished within a specified time period. Thus, the compiler only needs to insert code to measure the response time (or other resource consumption) of the `run` method for each adaptive class.

6.4 Adaptation Approach and System Architecture

In the previous section, we discussed how an application developer can expose the constraints and the adaptation parameters to the system. We now focus on the use of program analysis and runtime feedback for supporting the desired adaptation.
Consider an application which is executed for the first time in a given environment, or if the resource availability (e.g. available CPU cycles or network bandwidth) in the environment has changed recently. It is possible that with the initial or existing values of the adaptation parameters, the constraint(s) specified in the program may be violated. The key question is, how do we choose the adaptation parameter(s) whose value should be modified, and by how much do we modify these parameters. For this purpose, we need to compute the relationship between the execution time and the set of adaptation parameters.

We use a hybrid methodology, which combines compile-time analysis with runtime feedback. A program analysis algorithm states the execution time of an application component as a function of the values of the adaptation parameters. Suppose the processing at a particular stage involves two adaptation parameters, $x$ and $y$. A program analysis module analyzes the code and constructs an expression for execution time for this stage as follows:

$$T = \sum_k C_k \times x^{ik} \times y^{jk} + C$$

Here, $C$ and $C_k$ are constants that do not depend upon any of the adaptation parameters, though their specific values may not be determined by the program analysis module. Thus, the expression for execution time is a sum of a constant factor and several terms that include powers of $x$ and/or $y$ (note that $ik$ or $jk$ could be zero).

Once such an expression for execution time has been constructed, initial runtime runs are used to compute the values of the constants involved. These runs involve executing the processing phase with different values of adaptation parameters, and the resulting execution times are noted. By substituting the values of adaptation
parameters and execution times in the above expression, a set of linear equations is generated. Solving these equations gives the value of the constants.

Once these constants are known, we know which adaptation parameters can be modified and by how much to achieve a given change in execution time, or to maintain the same execution time for a given change in available resources. We believe that our approach is more practical than most previous approaches to compiler-based modeling and prediction [12, 20, 148, 64, 88, 21, 79], since we do not expect the compiler to estimate the constants.

As we had stated earlier, our approach has been implemented in the context of grid-based data-intensive applications using the coarse-grained pipelined execution model. The execution time for an application using this model typically depends upon the slowest or the bottleneck stage. This bottleneck stage can be a computational stage or a communication stage. In our earlier work, we had developed an algorithm for determining the communication that needs to be performed between the different stages in this model [55]. The output from this algorithm can be used for determining how the change in adaptation parameters can change the communication volume. Therefore, our discussion in this chapter will focus on estimating the dependence of computation time on adaptation parameters.

Our overall system architecture is shown in Figure 6.3. The system comprises three basic units, in addition to the the DataCutter Runtime system used for supporting pipelined parallel execution. The program analysis module takes as input the Data Parallel Java source code, and performs analysis to determine the relationship between execution time and the adaptation parameters. This module also inserts the profiling code that obtains execution times for the run routines. The filter-stream code for
the DataCutter runtime system is generated by the code generation module. The adaptation module examines the execution log output by the runtime system and the expression computed by the program analysis module, and determines the runtime constants in the expression for execution time. This module is also responsible for modifying the values of adaptation parameter(s) when the constraints are not met. Our system assumes that change in resource availability can be detected, using tools such as the Network Weather Service (NWS) [146].

The next two subsections describe the techniques used in the program analysis and adaptation modules, respectively. Details of the code generation module were presented in our earlier work [55].
6.4.1 Program Analysis for Supporting Program Adaptation

This subsection focuses on the program analysis component of our adaptation framework. As we stated earlier, our goal is to state the execution time of an application component as a function of the values of the adaptation parameters and other constants. The constant values must be independent of the values of the adaptation parameters, but need not be determined till runtime.

A number of projects have focused on predicting the execution time for a program [12, 20, 79, 148, 64, 88, 22] as accurately as possible. With growing complexity of applications and machine architectures, it is becoming increasingly hard to make such predictions. However, since we use runtime runs to determine the constants, the problem we are addressing is much simpler than the general execution time prediction problem.

Our algorithm is presented in Figure 6.4. Our algorithm performs interprocedural analysis but assumes structured control flow. Thus, the only control structures in a procedure are for or foreach loops, and if-then statements. The critical factors impacting the execution times of an application or application component are the iteration counts for the loops and the frequency of executing the body of a conditional.

We classify each conditional or loop into two classes, on the basis of whether the condition or the iteration count depends upon one or more adaptation parameters or not. The frequency of executing the body of a if statement whose condition does not depend upon any of the adaptation parameters is assumed to be constant. Similarly, the iteration count of a loop whose loop bounds or the termination condition does not depend upon any of the adaptation parameters is assumed to be constant. Note that neither of the assumption may hold true in practice. For example, the conditionals
FindRel
Input: A statement \( s \) to be analyzed, and a list \( varList \) containing variables related to the adaptation parameters.
Output: a string representing the execution time.
algorithm:
switch \((s \rightarrow \text{type})\)
{
case assignment:
  if \( rhs \) is related with \( varList \)
    record \( lhs \) in \( varList \);
  if \( rhs \) is a method call
    \( \exp = \text{FindRel}(s \rightarrow rhs, varList) \);
  \( \exp = \text{"CONST"}; \)
case method call:
  if arguments are related with \( varList \)
    \( \exp = \text{FindRel}(s, varList) \);
  else \( \exp = \text{"CONST"}; \)
case if:
  if \( \text{cond} \) is related with \( varList \)
    \( \exp = \text{FindRel} (\text{cond}, varList) \);
  else \( \exp = \text{"CONST"}; \)
    \( \exp = \exp \ \text{concat} \ "\ x\" \ \text{concat} \text{FindRel}(s \rightarrow \text{body}, varList) \);
case for/foreach:
  if loop count is associated with \( varList \)
    \( \exp = \text{FindRel} (\text{loop count}, varList) \);
  else \( \exp = \text{"CONST"}; \)
    \( \exp = \exp \ \text{concat} \ "\ x\" \ \text{concat} \text{FindRel}(s \rightarrow \text{body}, varList) \);
case block:
  \( \exp = \exp \ \text{concat} \ "(\text{CONST})"; \)
  for each statement \( s_i \) in the block
    \( \exp = \exp \ \text{concat} \ "+\" \ \text{concat} \text{FindRel}(s_i, varList) \);
  \( \exp = \exp \ \text{concat} \ "\); \}
return \( \exp \);

Figure 6.4: Algorithm to Determine the Relationship between Execution Time and Adaptation Variables.

or the iteration counts could depend upon the input data. However, our goal is to infer the best that is possible at compile-time.

We also make the following two assumptions. First, the cost of execution of a block of assignment statements is assumed to be a constant. Second, the cost of execution
of an if-then construct is the frequency of executing the then part multiplied by the cost of one execution of the then part. Similarly, the cost of execution of a loop is the iteration count multiplied by the cost of one execution of the loop body. Again, in the presence of deep memory hierarchies or complex pipelines, none of these assumptions may always be true.

Thus, the main task for the algorithm is to determine where the iteration counts or the conditionals could depend upon the adaptation parameters, and how. A data structure varList is maintained to record all the variables directly or indirectly related to the adaptation parameters, along with how they are related to the adaptation parameters. When processing a function call, the mapping of the formal parameters to the argument list should also be noted down in varList. In addition, by utilizing a different version of varList for a new block, the scoping rules are also taken into account.

Note that the boolean expressions in the conditionals should be given more consideration. The input data of our target applications can be considered as a collection of objects with same type, as suggested by the data parallel construct Rectdomain. There is a coordinate associated with each object in the collection. Hence, any sampling based adaptation can be projected to the coordinate space, i.e. sampling rate 2 for the input data is comparable to step size 2 in the index array. Also worth mentioning is that this rectilinear section is continuous, which can help ease the branch prediction tasks.

In the following part, we will take two example codes to illustrate the algorithm. Figure 6.5 shows a part of the code of the K-means clustering algorithm. For this application, it and k are two adaptation variables. At the first glance, the only part
affected by these two variables are the loop counts. However, the cost associated with $\text{Center.renew()}$ is also affected by the value of $k$, since $k$ is passed as an argument to the constructor of class $KCenter$ when $\text{Center}$ is initialized. So, after analyzing line (1), $\text{Center.num\_centers}$ is recorded in the $\text{varList}$, along with its value $k$.

The number of iterations in the outer-most loop is bounded by the value of $it$. The cost for executing (3) is related to $k$ since it is associated with $\text{Center.num\_centers}$, and is denoted as $k \times \alpha$. The loop count for the $\text{foreach}$ statement in line (4) is unrelated with $it$ or $k$ and thus can be regarded as a constant. For the line (5), $k$ is identified as its iteration count, and for the statements (6) and (7) in its loop body, the cost can be represented as $\beta$. Performing line(10) requires constant time with respect to the adaptation parameters. Overall, the execution time can be approximated as

$$T = it \times (k \times \alpha + C \times (k \times \beta + C_1 + C_2) + C_3$$

$$= it \times k \times (\alpha + C \times \beta) +$$

$$it \times (C \times C_1 + C_2) + C_3$$

(6.1)

We can rewrite 6.1 as

$$T = it \times k \times \alpha + it \times \beta + C$$

(6.2)

Furthermore, if only one variable is an adaptation variable, we have

$$T = it \times \alpha + C$$

(6.3)

or,

$$T = k \times \alpha' + C'$$

(6.4)

The code for $\text{is\_sampled}$ method in Figure 6.6 shows the processing of conditionals in our algorithm. The execution frequency of the if statement is $1/\text{samp}^2$ as one out of $\text{samp}^2$ pixels will be sampled from the original input image.
6.4.2 Adaptation Module

We now briefly describe the functionality of the adaptation module. This module is responsible for two major tasks. First, it carries out a number of initial runs of the application to determine the values of the constants in the expression computed by program analysis module. Second, after the values of these constants are known, it modifies the values of adaptation parameters to meet the constraints specified by the programmer.

Consider the first task. The number of initial runs is typically equal to the number of unknown values in the expression that has been computed. This is because by supplying the initial values of the adaptation parameters and obtaining the execution times, we can obtain linear equations. The number of distinct equations that are required is equal to the number of unknown values.

Let $x$ and $y$ be the adaptation parameters, and let $C_i$ be the constants. Consider an execution time expression of the form

$$T = x^i y^j C_1 + C_0$$

Examples of expressions that have this form are $T = x C_1 + C_0$, $T = C_1 / x + C_0$, $T = x y C_1 + C_0$, etc. In each of these cases, we need to have initial runs with distinct values of $x$ (and $y$, if applicable). The distinct values we use are typically among the minimum, the maximum, or the median value of the adaptation parameter.

Similarly, consider any expression of the form

$$T = x^i y^j C_2 + x^k y^l C_1 + C_0$$

Three initial runs are required for finding the values of the three constants that are involved.
Once these constants are known, it is easy to select new values of the adaptation parameters to meet the given constraint.

6.5 Experimental Results

In this section, we report on the experiments we conducted to evaluate our approach for supporting adaptive execution. We had two specific goals in designing our experiments. First, we wanted to evaluate if our combined approach, which combines static analysis and runtime feedback, can accurately predict execution times for our target applications. Second, we wanted to demonstrate that this approach can be used effectively with the compiler-generated code for achieving coarse-grained pipelined parallelism.

Before discussing the results we have obtained, we first describe the experimental setting, the applications we have used, and how adaptation plays a role in these applications.

6.5.1 Experimental Settings and Applications

In the long run, we expect that pipelined parallelism can be exploited in wide-area networks. However, this is going to require high bandwidth networks and certain level of quality of service support. Recent trends are clearly pointing in this direction, for example, the five sites that are part of the NSF funded Teragrid project expect to be connected with a 40 Gb/second network [125]. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single cluster. We assumed that input data is available on only a subset of the nodes, and final
results are required or viewed on a single node. The cluster we used had 700 MHz Pentium machines connected through Myrinet LANai 7.0.

All the applications used in the experiments had three stages of processing. The data is read from the data repository, application-specific processing is performed on the dataset, and final results are forwarded to the end user. We used three different pipeline configurations. In the first configuration, the data is available at a single node, another node is available for computations, and the results are presented at yet another node. This configuration is designated as the 1-1-1 configuration. In the second configuration, data is available at 2 nodes in the cluster, another 2 nodes are available for computations, and the final results are required at one distinct node. This configuration is designated as the 2-2-1 configuration. In the third configuration, referred to as the 4-4-1 configuration, data is available at 4 nodes, another 4 nodes are available for computations, and the final results are required at another node.

We used three different applications in our experiments. Our goal was to create a set that is representative of the class of applications discussed in Section 6.2. Virtual Microscope is an emulation of a microscope which allows users to view part of the original image with different resolutions [13]. A query specifies a rectangular region and a range of values for the subsampling or the resolution factor for the desired output. This factor is used as the adaptation parameter. When the resources are adequate and stable, users can view the same image with a better resolution. When the resources such as the network bandwidth or CPU cycles are limited, a lower resolution image may have to be displayed to meet the constraint on the time.

The second application is comp-steer, based around the use of data stream processing for scientific data analysis. Here, a simulation running on one computer
generates a data stream, representing intermediate values at different points in the mesh used for simulation. These values are sampled, communicated to another machine, and then analyzed. The processing time in the analysis phase is linear in the volume of data that is output after the sampling. The sampling rate, denoting the fraction of original values that are forwarded, is the adaptation parameter used in this application.

K-means clustering is a very commonly used clustering (data mining) technique [77]. An implementation of k-means was used as the third application. In this algorithm, a value or a range of values of \( k \) is taken as input. The goal of the algorithm is to partition a given set of multi-dimensional points into \( k \) clusters, using proximity within the multi-dimensional space as the criteria. Three steps in this algorithm are as follows: 1) start with \( k \) given centers for clusters; 2) for each data instance (point), find the center closest to it, assign this point to the corresponding cluster, and then move the center of the cluster closer to this point; and 3) repeat this process for a fixed number of iterations or until the assignment of points to the cluster does not change. There are a number of ways in which this algorithm can be adapted. Obviously, the more the iterations performed, the algorithm will give partitions which are closer to the optimal. So, the number of iterations needed could be varied when the resource availability changes. The other potential adaptation parameter is the number of clusters desired, since it also impacts the execution time.

6.5.2 Results from Virtual Microscope

We used a 800 MB image for this application. The query specifies a 28961 × 27808 range, and the value of the adaptation parameter, i.e., the resolution level \( (r) \), can
range from 1 to 10. Our program analysis algorithm determines the execution time as \( T = C_1/r^2 + C_0 \). Two initial runs are required for computing the two constants. Here, we select the minimum and maximum value of \( r \) for these runs. Figures 6.7, 6.8 and 6.9 compare the predicted execution time with the actual running time for the three different pipeline configurations. The difference between predicted and observed execution times are within 15%. Comparing the execution time across the different configuration shows that good speedups are achieved through the compiler and runtime support for coarse-grained pipelined parallelism.

6.5.3 Results from Comp-Steer

A dataset of size 1.2 GB was used. The adaptation parameter, sampling rate (s), can vary between 1 and 100. Our program analysis algorithm states the execution time as \( T = C_1/s + C_0 \). Two runs are required for determining the constants, and are carried out using the minimum and maximum values of the sampling rate. Figures 6.10, 6.11 and 6.12 compare the predicted execution time with the actual running time for different pipeline configurations. The difference between predicted and experimental results is within 7.2%. Comparison of execution times also shows good speedups from increasing the pipeline width.

6.5.4 Results from K-Means Clustering

The dataset used here comprises 9,000,000 3-dimensional points, with total size 108 MB. The application is run with different number of clusters (k) and different number of iterations (it). We carried out a number of experiments with this application.
Number of Iterations as the Only Adaptation Parameter: The program analysis module determines the execution time as \( T = it \times C_1 + C_0 \). We select the minimum and maximum value of \( it \) for the two required initial runs. Figures 6.13 and 6.14 compare the predicted execution time with the actual running time for \( k = 5 \) and \( k = 10 \), respectively. The difference between predicted and actual results is within 5%. One of the question we were interest in is, how accurate does our prediction stay when the choice of parameter values for initial runs is changed. In Figure 6.15, we show the predicted results, calculated with the minimum and median values of \( it \). The difference between these two sets of predicted execution times is within 5%.

Number of Clusters as the Only Adaptation Parameter: The program analysis module computes the execution time as \( T = k \times C_1 + C_0 \). Again, we used the minimum and maximum values of \( k \) for the initial runs. Figure 6.16 compares the predicted execution time with the actual running time with \( k \) ranging from 5 to 20. The difference between predicted and experimental results is within 3%.

Considering Two Adaptation Parameters: The program analysis module computes the execution time as \( T = it \times k \times C_2 + it \times C_1 + C_0 \). Three runs are required to obtain values of the three constants involved. Figure 6.17 shows the the actual execution times with different number of iterations and different number of clusters. In Figures 6.18 and 6.19, we show two sets of predicted values, obtained by different sets of initial runs. The actual execution times and the two sets of predicted execution times are always within 8%.

6.5.5 Summary of Results

To summarize, the main observations from our results are as follows.
• Our combined program analysis and runtime feedback based approach is quite precise in practice. The actual execution times are within 8% of the predicted execution times in almost all cases. This is quite impressive, considering that our model does not take memory hierarchy into account.

• Our approach is effective irrespective of the choice of the set of parameters for initial runs.

• Our approach works well with the coarse-grained pipelined parallel execution model and data-intensive applications.

6.6 Summary

In this chapter, we have focused on the support of adaptivity aspect of our target class of applications, where the users have some flexibility in the output they desire, while under some performance constraint. We have designed language, compiler, and runtime support for this class of applications. The main idea in our language support is to allow specification of adaptation parameters. At compile-time, our program analysis algorithm states the execution time as a function of adaptation parameters and other runtime constants. An initial set of runs are used for determining these runtime constants.

Our experiences with three data-driven applications have shown the feasibility of our approach. The adaptive nature of the applications can be expressed through our simple language extensions. The execution time model we derive through combined compile-time and runtime analysis is quite effective in practice, even though it does not explicitly account for memory hierarchies.
class KCenter
{
    KCenter(int k)
    {
        num-centers = k;
        ... ... }
    void Renew()
    {
        for (int i=0; i<num-centers; i++)
        {
            if (count[i]!=0)
            {
                center[i].setx((int)(sumx[i]/count[i]));
                center[i].sety((int)(sumy[i]/count[i]));
                center[i].setz((int)(sumz[i]/count[i]));
            }
            sumx[i] = 0; sumy[i] = 0; sumz[i] = 0;
            count[i] = 0;
        }
    }
    public void run() {
        ... ...
        Center = new KCenter(k);
        for (i=0; i<it; i++)
        {
            if (i>0) Center.renew();
            foreach (p in InputDomain)
            {
                for (j=0; j<k; j++)
                {
                    dis = Point[p].distance(Center.get(j));
                    if (j==0) {min = dis; ind = j;}
                    else
                    {
                        if (dis<min) {min = dis; ind = j;}
                    }
                    Center.update(Point[p],ind);
                }
            }
        }
    }
}

Figure 6.5: Example Code from K-means Clustering: Code Segment from run function and its related class.
public class VMPixel {
    int x;
    int y;
    boolean is_sampled(Point<2> low, int samp)
    {
        if (((x - low[0]) % samp == 0) &&
            ((y - low[1]) % samp == 0))
            return true;
        return false;
    }
    ...
}

Figure 6.6: Example Code from Virtual Microscope: The is_sampled method of class VMPixel.

Figure 6.7: Comparison of predicted and actual execution times, 1-1-1 pipeline configuration
Figure 6.8: Comparison of predicted and actual execution times, 2-2-1 pipeline configuration

Figure 6.9: Comparison of predicted and actual execution times, 4-4-1 pipeline configuration
Figure 6.10: Comparison of predicted and actual execution times, 1-1-1 pipeline configuration

Figure 6.11: Comparison of predicted and actual execution times, 2-2-1 pipeline configuration
Figure 6.12: Comparison of predicted and actual execution times, 4-4-1 pipeline configuration

Figure 6.13: Comparison of predicted and actual execution times, $k = 5$
Figure 6.14: Comparison of predicted and actual execution times, $k = 10$

Figure 6.15: Comparison of predicted and actual execution times, $k = 10$, minimum and median value of $it$ for initial runs
Figure 6.16: Comparison of predicted and actual execution times, \( it = 10 \)

Figure 6.17: Actual execution time for K-Means Clustering: Two Adaptation Parameters
Figure 6.18: Predicted execution time for K-Means Clustering: First Set of Runs

Figure 6.19: Predicted execution time for K-Means Clustering: Second Set of Runs
CHAPTER 7

RELATED WORK

In this chapter, we review the literature for previous research efforts related to our work. The study will mainly focus on the following topics: support for large-scale data-intensive applications, support for exploiting coarse-grained pipelined parallelism, job scheduling algorithms, performance modeling and tuning, support for application adaptation.

7.1 Support for Large-Scale Data-intensive Applications

Large-scale distributed data-intensive applications play an increasingly important role in advancing the human society. For example, the high energy physics experiments can foster the understanding of the form of universe. The biomedical informatics analysis can facilitate discovery of myths of life. Hence, this class of applications attract a large body of scientists and researchers from both computer science and other domains such as physics, chemistry, and biology, among all. Under such unprecedented scale collaboration, numerous software systems have been developed in the past decade.
7.1.1 Grid Middleware Systems

Distributed data-intensive applications serve as the main motivating force of the Grid technologies. To support this class of applications, a wealth of Grid middleware systems has been developed. Among them, Globus [67] toolkit consists of a set of components which provide basic services for resource management, communication, information access, security, and so on. Legion [76] addresses issues of scalability, faulty tolerance, security and site autonomy by performing resource management, scheduling, and other system-level tasks. By code migration and CPU co-scheduling techniques, Condor [59] provides mechanisms to make use of idle cycles on a cluster of workstations connected by a network.

Others focus on more specific services, such as NetSolve [41], AppLES [42] and Network Weather Service (NWS) [147]. NetSolve [41] takes the assumption that large computational problems are generally solved by making calls to available software packages. It offers users the ability to utilize remote computational resources easily and efficiently. As a user-level grid middleware, AppLES [42] adopts application-level scheduling techniques to support efficient execution of parameter sweep applications over the grid. At the network level, Network Weather Service (NWS) [147] aims at forecasting dynamically changing performance characteristics of a set of distributed resources, using network monitoring and performance prediction, resource reservation and network-level adaptation. A similar project is introduced in [52]. The Remos system is intended to provide resource information about a variety of network environments to distributed applications [52]. They employ different types of collectors on each particular network environment to gather topology and flow information about
the system. These information are then organized by modelers and presented to the application in the form of logical topology [111].

Another effort worth mentioning is the Grid Application Development Software (GrADS) Project, which aims at simplifying the application development tasks while taking into account the performance goals under resource variations. They considered an application as a configurable object program, which allows the rapid adaptation. They also proposed the telescoping language for constructing such programs, for which a library of Grid-aware components should be available [24].

Comparison with our work: We also work on supporting distributed large-scale data-intensive applications.

We provide advanced support for this class of applications, incorporating language and compilation techniques. With our system, application developers can write a sequential code with some time constraints. Then, the application can be deployed on a pipeline of computing units and execute on a dynamic changing environment, meeting the performance requirements.

7.1.2 Workflow Systems

In some cases, distributed data-intensive applications might be multidisciplinary, and not monolithic codes. They consist of existing application components. Under such situation, we can regard the applications as being defined by workflows, with each step being an individual component fulfilling certain tasks or satisfying a given constraint. These steps are then connected reflecting their data/control flow dependencies.
Usually, high level description languages are provided to allow the specification of workflows [9, 97]. For example, xWFL is a simple XML-based workflow language for constructing applications [152]. The Virtual Data Language (VDL) of Chimera project can be used to describe a logical workflow [69]. With the WebFlow system, the users can even take advantage of a visual programming tool to compose a new application with existing components using drag and drop capability [14, 109].

After the workflow construction stage, we only get hold of an abstract workflow. Next step is to transform the abstract workflow to a concrete workflow by mapping each component of the entire application onto certain resources. To fulfill this task, mechanisms for resource discovery, resource selection, and resource monitoring must be available. Numerous workflow management systems make use of the services provided by the Globus toolkits, such as the Globus Monitoring and Discovery Services (MDS) [65], Globus Replica Location Service (RLS) [46], Grid Resource Allocation and Management (GRAM) [49] and Grid Security Infrastructure (GSI) [145]. For example, Pegasus consults RLS to locate the replicas of required data, queries MDS to get the available resource characteristics, and uses GSI for authentication purposes [51]. Other projects taking the similar approach include GridFlow [40], GridAnt [16] and Nimrod-G [36].

The tasks specified by workflows will be executed on a large-scale distributed or grid-like environment. Due to the dynamic nature of such execution environment, a full-ahead-plan may not always be suitable. For this reason, Pegasus also provides the just-in-time planning. The original abstract workflow will be partitioned before the instantiation step. A partition of the workflow will be scheduled only after all its
preceding tasks has been completed [51]. With such partial workflow submission, the system can adjust the resource mapping in case of failure at execution time.

In [152], Yu takes a decentralized event-driven approach to allocate and schedule resources at run time. For this purpose, a task manager (TM) is created for each task in the workflow. All task managers are controled by workflow coordinator (WCO). Communication between TM and WCO is via the event service server (ESS). The ESS is implemented using tuple spaces. All TMS and WCO need to subscribe to ESS in order to get invoked on a particular event. Tuples are created and sent to ESS by TMs with the event information such as task status and output location. ESS is in charge of notifying the interesting party of a particular event. If all the input data is available for a given task, the corresponding TM will proceed to handle the task.

7.1.3 Other Efforts

Several other projects offer support for distributed applications which is somewhat similar to our target runtime system, DataCutter, as introduced in Section 2.4. An example project is the active streams and related efforts from Georgia Tech [35, 92, 122].

The Active Streams project implements a middleware approach to build adaptive distributed applications. They consider the distributed system being composed of applications, services and data streams. Data streams are made active when attached with streamlets, which is very similar to the filter concept in the DataCutter system. Streamslets are location-independent functional units performing operations on the incoming data items and generating results on output streams. The adaptation to
dynamic underlying environment is realized through attachment, parameterization
and re-deployment [35, 92, 122].

In the past, there are also efforts focusing on the use of heterogeneous environments
for parallel computing. A representative project is HenCE [23]. HenCE is built
on top of PVM, and provides an integrated graphical interface for programmers to
construct a parallel programs. A HenCE program consists of two parts. One part is a
program graph, with nodes in the graph representing procedures and arcs denoting the
dependencies. The other part is that for each procedure in the program, there is a set
of implementations associated with it for different architectures. HenCE then writes
necessary PVM code and compiles this code on the user requested configuration.
When comes to the execution, HenCE can dynamically balance the work load among
different machines, considering the heterogeneity in both algorithm implementations
and hardware performance.

**Comparison with Our Work:** Our work is distinct from the previous efforts
in two aspects. First, we provide high-level language support that hides heterogeneity
from the programmers. In addition, programmers do not need to consider how to
partition the program and explicitly specify the parallelism within the applications.
Second, the application adaptation is realized through parameterization by an ap-
proach combining both static program analysis with runtime feedback. In all, our
approach indeed allows the programmers to focus on the algorithmic aspect of the
problem, since it helps hide all other implementation details.
7.2 Support for Coarse-Grained Pipelined Parallelism

Several efforts have targeted runtime support for coarse-grained pipelined parallelism. The Stampede project [126] has focused on interactive multimedia applications, which have several common characteristics with the applications we have targeted. The support offered is in the form of cluster-wide threads and shared objects. They do not include high-level language or compiler support, but can handle more dynamic applications. Yang et al. have developed a scheduler for vision applications which are executed in a pipelined fashion within a cluster [149]. They include support for meeting real-time constraints, but require low-level programming of applications.

Our work has some similarities with the StreamIt effort at MIT [136]. StreamIt projects implement language and compiler level support for streaming applications. They introduced a structured model for streams, and included stream-specific abstraction and representation. A streaming application can be represented by a set of filters connecting by structs such as Pipeline, SplitJoin and FeedbackLoop, which is demonstrated in a stream graph. With such explicit presentation, StreamIt compiler performs stream-specific analysis and optimizations to improve the application performance. Though the set of applications that have been experimented with are different, streaming applications have some similarities with the data-driven applications we are targeting. Our language is at a higher-level, but we can only handle applications that involve generalized reductions. Also, the target architecture of the two efforts are quite different.

Pipelined parallelism has also been considered for communication-exposed or pipelined FPGA architectures. Ziegler et al. have developed compiler analysis for mapping a
program to such an architecture [155, 153, 154]. While they consider different granularity of communication between the FPGAs, the evaluation of different choices is done through a design tool, and not analytically.

7.3 Scheduling Chained-tasks on Chained Processors

The filter decomposition problem studied in Chapter 3 is somewhat similar to the problem of scheduling chained-tasks on chained processors. Bokhari et al. initially provided a $O(n^3m)$ solution to this problem [32, 33]. This was subsequently improved to $O(n^2m)$ by Hansen [78] and Nicol [118]. Our MIN_BOTTLENECK algorithm has the same complexity.

Also, many researchers have considered somewhat different versions of the problem. A faster but approximate algorithm for the same problem was developed by Iqbal [91]. Sheu and Chiang considered a generalization [133], whereas, Hsu presented an algorithm having a merge phase and an assignment phase [85]. Chan and Young even answered the decision version of the problem and found the optimal solution with minimal number of processors [103, 151].

Our main contribution in this work is the overall treatment of the problem and the other algorithms we have presented, as well as experimental evaluation of each of these algorithms.

7.4 Communication Analysis and Optimization

A significant part of the analysis our compiler performs is communication analysis. Though communication analysis has been studied extensively as part of SPMD
compilers [11, 81], our analysis is different because of supporting a different form of parallelism.

An important part in exploiting distributed memory parallelism is optimizing communication. With the owner computes rule, all computations associated with a data item must be performed on the processor owning it. Any other values involved in the computation which are not available locally have to be communicated from other processors. Hence, reducing the communication cost is always a optimization target. For this purpose, an abundance of optimization techniques have been developed, including message vectorization [71], message coalescing [38], message aggregation [82], collective communication [100], message pipelining [130], and redundant communication elimination [73].

A large body of work exists on compiler support for instruction-level or fine-grained pipelined parallelism [96]. Early efforts on ILP compilation consider instructions in a basic block for their independence. However, most basic blocks can only present very limited ILP [142, 98]. Several approaches have been proposed to enlarge the scope of consideration. Exemplars are to form a trace [63, 134] or a superblock [90], which is a region composed of a sequence of basic blocks along a frequently executed path. These two techniques are very effective when a single frequently executed path exists. With multiple paths, predicated execution will be used to form a larger region [86, 128]. Loop unrolling and software pipelining also can be used to generate larger collection of instructions for optimizations [105, 56, 127, 138].

There are different kinds of dependences between instructions which hinder the parallelization, namely data dependences and control dependences. A rich variety of techniques have been developed for overcoming these dependences. For example,
register renaming is used to overcome false data dependences [89], and many prediction techniques are used to overcome true data dependences [131, 37, 102, 132]. In the case of loops, a group of loop transformation techniques are developed to remove loop-carried dependences [106].

Our work considers a very different target environment. We focus on coarser grain or program level parallelism, and our goal of the analysis is to find the required communication data items across each candidate filter boundary. Apart from the identity of these items, for further tuning the application performance, the granularity of the communication should also be considered.

Our work in choosing the optimal communication granularity is closely related to the work on optimizing packet size in a multi-link communication pipeline by Wang et al. [144]. In comparison, our contributions are two folds. First, we have derived expressions for cases where computation is involved and as a result, frequency and/or the volume of communication across different links is different. Second, we have carried out a detailed validation of the model for our target applications and their execution scenario. Authors of [115] also applied Wang’s idea to improve throughput of remote storage access through pipelining. Obviously, they targeted at a different problem than ours.

7.5 Support for Application Adaptation

7.5.1 Middleware Support

A number of efforts exist on runtime or middleware support for adaptation. DART [129] is a system facilitating quick development of adaptive applications. It
provides compile-time support for generating *glue code* on the basis of high-level keywords in the source code. A runtime component is responsible for making adaptation decisions following a set of selected policies. Moura *et al.* present software support in the component-base programming context for construction of auto-adaptive applications [50]. It leaves applications the option to dynamically choose the most beneficial components. ROAM implements resource-aware runtime adaptation for device heterogeneity in mobile systems [87]. Schwan and his group take into account the runtime resource management issues when supporting adaptable applications [124]. Odyssey [119] proposes an application-aware adaptation, where adaptive decisions are driven collaboratively by both operating system and applications. Task scheduling and migration for Grid computing are utilized in Condor [135] to achieve adaptability. CACTUS-G incorporates sophisticated application-transparent adaptation strategies into applications built on top of CACTUS [15]. In another effort, an architecture-based adaptation strategy is used to monitor an application and direct dynamic changes to it [45]. In comparison, our work is distinct in considering a class of applications where adaptation is based upon change in adaptation parameters, and providing an integrated language, compiler, and runtime approach.

### 7.5.2 Language and Compiler Support

Program Control Language (PCL) is probably the closest to our work. It takes an aspect-oriented programming paradigm to allow adaptive behaviors to be specified separately from the base functionalities [58]. A set of language extensions has been incorporated into PCL for adaptation operations and performance monitoring. It also
applies static task graph as a formal model for adaptation. A somewhat similar approach is taken by Alua [139] in the context of component-based programming. The basic components of the system are written in a compiled language such as C, while the configuration part is specified by an interpreted language Lua. Diniz and Liu have proposed Selector as a language construct for adaptation [53]. Voss and Eigenmann have developed ADAPT, which uses a specialized language to allow the programmers to specify the adaptation that can be done at runtime [141]. Our approach is specifically focused on applications where the adaptation is through the change in values of adaptation parameters. For such applications, our language extensions require significantly less effort from the programmers.

In the Grid Application Development Software (GrADS) project [25], it is proposed that an application is encapsulated as a configurable object program that can be reoptimized for execution at runtime. Their goal is to support performance contracts guided dynamic optimization. Again, the focus and the approach is quite different from ours, and we are not aware of any existing implementation of their approach.

7.5.3 Other Efforts

There is some similarity between our work and the area of dynamic or adaptive compilation. Here, programs are first compiled and initiated, and then when runtime hardware configuration or system load changes, the system might recompile parts of the application. Efforts in this area include Merlin [95], DyC [74], Č [57], and the work from Washington University [123].

Resource-aware programming (RAP) can be viewed as another way to support program adaptation. Systems supporting mobile and real-time applications also need
to meet some resource constraints like response time or space limitation. RTC++ provides active objects to allow timing constraints specification for a task within an object declaration [93]. Hooman and his colleagues extend non-real-time programming languages with timing annotations, which enables timing constraints to be specified on an abstract level [84]. The Lagoona project [70] also adds language extensions to allow defining real-time constraints with a component-oriented paradigm. Other related works include Amulet from CMU [114], Timber from OHSU [30], and Chuck from Princeton [143]. Here, the focus has been on scheduling to meet real-time constraints, as opposed to adapting the output of the applications to meet certain constraints.
CHAPTER 8

CONCLUSIONS

This section summarizes the contributions of the work described in this dissertation.

8.1 Contributions

In this dissertation, we have presented advanced middleware support for developing and executing data-intensive applications on the distributed or Grid environment. The research statement can be reiterated as: 

*Initiating simple language constructs and designing effective compiler algorithms for distributed data-intensive applications to ease the programming tasks, allowing programmers to focus on the already strenuous problem-solving issues, while hiding details unrelated to the applications.*

We have carried out the following tasks:

- **Supporting Execution of grid applications on the coarse-grained pipelined execution model** We use a dialect of Java that exposes both pipelined and data parallelism to the compiler. Our compiler is responsible for selecting a set of candidate *filter boundaries*, determining the volume of communication
required if a particular boundary is chosen, performing the decomposition, and generating code. We have developed a one-pass algorithm for determining the required communication between consecutive filters, a cost model for estimating the execution time for a given decomposition, and several polynomial time algorithms for performing the decomposition.

In a separate study, the filter decomposition problem has been further investigated and formulated in a mathematical way. There are three different algorithms being proposed. Algorithm MIN_ONETRIP is a dynamic programming algorithm, optimizing the one trip cost for a packet passing through the pipeline. Another dynamic programming algorithm, MIN_BOTTLENECK, minimizes the time spent on the bottleneck stage. Finally, MIN_TOTAL is an approximate greedy algorithm which tries to minimize the total execution time. The results show that these heuristic algorithms work quite well in practice, with the possible exception of MIN_ONETRIP when the number of packets became larger.

- **Optimizing grid application performance by selecting the optimal packet size** We develop an analytical model for choosing packet size, i.e., the unit of transfer between the pipeline units. Because the pipeline includes both communication and computation phases, the frequency and/or volume of communication between different phases can be different. We consider two models, fixed-frequency and fixed-size, and derive mathematical expressions for both. The detailed evaluation of our models, conducted with three applications, executed with different parameters and datasets, show that the choice of packet size makes a significant difference in the execution time, and the packet sizes
suggested by the model result in the lowest or very close to the lowest possible execution time.

- **Adapting sample-based grid applications** We provide a combined language/compiler and runtime solution for supporting *adaptive* execution of these applications, i.e., to allow them to achieve the best precision while still meeting the specified constraint at runtime. The programmers have flexibility to specify *adaptation parameters*, i.e., the parameters whose values can be varied within a certain range. A program analysis algorithm states the execution time of an application component as a function of the values of the adaptation parameters and other runtime constants. These constants are determined by initial runs of the application in the target environment. We integrate this work with our previous work on supporting *coarse-grained pipelined* parallelism, and thus support adaptive execution for data-intensive applications in a distributed environment.

### 8.2 Future Work

In concert with the advent of wide area high bandwidth network technologies, the class of distributed data-intensive applications will become increasingly popular and important in almost every domain. Accordingly, system support for developing and executing such applications are in urgent need under the unprecedented scale. For this purpose, the work presented in this dissertation can be expanded in many different ways. Some representative problems I hope to work on are listed as follows.

*Dynamic Decomposition for Application Adaptation*

In the previous work on supporting adaptive applications, the following two assumptions are made. First, adjustment of parameters will not affect the slowest or
bottleneck stage of the pipeline, that is, the most time-consuming phase stays same before and after the parameter adjusting. Second, the performance constraints could be met by adjusting values of the adaptation parameters, i.e. assigning adaptation parameters certain values in the given range will result in the desired response time.

However, it is very possible that the bottleneck will shift when the values of adaptation parameters changed. It is also probable that, with a fixed program partitioning, the constraints cannot be satisfied at all, no matter what the values of adaptation parameters are. This kind of problems can be solved by integrating the flexibility of adaptation parameters into the decomposition strategy. That is, the impact of varying adaptation parameter values on the performance is taken into account when generating partitions for the application. For different parameter values, different partition might be chosen to achieve the best performance.

To reach this goal, a series of questions need to be answered. How many versions of partition are necessary for an application in order to meet the performance constraints under any variation of the resource availability? Which partition should be used for a particular range of the adaptation parameter values? How to decide the partition-parameter value relationship efficiently and effectively? Furthermore, if an application has already been deployed with a partition, and a new time constraint is introduced, how can this new constraint be met?

**Characteristics Modeling of the Grid Applications**

Based on my experience with the grid environment and applications, it is very critical to have accurate knowledge about the atomic filters of an application in order to better support as well as tuning the performance of the grid applications. The knowledge include the computation cost of each individual filter, the communication
volume between two consecutive phases, the relationship between the performance and adaptation parameters for each atomic filter, and the impact each parameter can bring to the performance of a filter.

To make the decomposition algorithms and the adaptation strategies more practical and useful, another two aspects of the applications are to be investigated: 1) estimates of the performance change resulting from combining several atomic filters and avoiding the cost of copying between the buffers; 2) estimates of the impact of data dependence on the performance for each atomic filter. Following this direction, a large body of applications should be studied and a number of delegation applications need to be identified. Both the compile time analysis results and the runtime performance information will be gleaned. In addition, some probabilistic and statistic techniques might be integrated into the model.

**System Supporting Adaptive Distributed/Grid Applications**

As a natural extension of my current work on adapting grid applications to the underlying ever-changing environment, a more extensive set of supports can be provided for adaptive applications to achieve their desired performance on a wider range of variations in resource availability. These supports will come from multiple levels of the software architecture, including programming environment, compiler optimizations, middleware innovations, operating system interventions and networking techniques. With such a system, the adaptability of the application can be exposed more intuitively and explicitly, the performance related features of the applications will be extracted and exploited more accurately, thus, the adaptation policies and strategies will perform more effectively. Meanwhile, other services and supports can be integrated into the system, involving resource monitoring, resource allocation and
task scheduling. I believe this system will ease the development of adaptive applications, improve their efficiency, and will be viable for accommodating a large group of applications from both commercial interest and scientific fields.
BIBLIOGRAPHY


173


[69] Ian Foster, Jens-Soencke Voeckler, Michael Wilde, and Yong Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In 14th International Conference on Scientific and Statistical Database Management (SSDBM 2002).


177


[146] R. Wolski. Forecasting Network Performance to Support Dynamic Scheduling
Performance Distributed Computing (HPDC), 1997.

[147] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distri-
buted Resource Performance Forecasting Service for Metacomputing. Journal

[148] Youfeng Wu and James R. Larus. Static branch frequency and program pro-
file analysis. In Proceedings of the 27th annual international symposium on

[149] M. T. Yang, R. Kasturi, and A. Sivasubramaniam. An Automatic Scheduler
and Distributed Processing Symposium (IPDPS), 2001.

[150] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Libit, A. Krishnamurthy,
P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-
performance Java dialect. Concurrency Practice and Experience, 9(11), Novem-
ber 1998.

[151] Gilbert H. Young and Chi lok Chan. Efficient algorithms for assigning chain-like

[152] Jia Yu and Rajkumar Buyya. A novel architecture for realizing grid work-

architectures. In Proc. IEEE Symp. FPGAAs for Custom Computing Machines,
2002.

[154] Heidi Ziegler and Mary Hall. Evaluating heuristics in automatically mapping
multi-loop applications to fpgas. In FPGA ’05: Proceedings of the 2005
ACM/SIGDA 13th international symposium on Field-programmable gate ar-

[155] Heidi Ziegler, Mary W. Hall, and Pedro Diniz. Compiler Generated Com-
Automation Conference (DAC), June 2003.