Compilation, Locality Optimization, and Managed Distributed Execution of Scientific Dataflows

DISTRIBUTION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Daniel Liwei Wang

Dissertation Committee:
Professor Stephen F. Jenks, Chair
Professor Pai Chou
Professor Charles S. Zender

2008
The dissertation of Daniel Liwei Wang is approved and is acceptable in quality and form for publication on microfilm and in digital formats:

______________________________

______________________________

______________________________

Committee Chair

University of California, Irvine
2008
DEDICATION

To my parents, Ike and Teresa.
# TABLE OF CONTENTS

LIST OF FIGURES viii

LIST OF TABLES x

ACKNOWLEDGMENTS xi

CURRICULUM VITAE xii

ABSTRACT OF THE DISSERTATION xv

1 Introduction 1
  1.1 Preface .......................... 1
  1.2 Overview .......................... 2
  1.3 Background and Related Work .............. 3
    1.3.1 The Geoscience Domain ............. 3
    1.3.2 Parallel Programming .............. 3
    1.3.3 Code Mobility .................. 5
    1.3.4 High-Performance Scientific Computing ....... 5
    1.3.5 I/O Parallelism ................ 6
  1.4 Strategies ........................ 7
    1.4.1 Geoscience Data Analysis and Reduction ............ 7
    1.4.2 Shell Compilation ................ 7
    1.4.3 I/O-Constrained Efficiency and Parallelism ........ 8

2 Geoscience Data Reduction and Analysis 9
  2.1 Introduction ........................ 9
  2.2 Problem .......................... 10
    2.2.1 Large data volume ................ 10
    2.2.2 High data-intensity .............. 11
  2.3 Approach .......................... 12
    2.3.1 File granularity ................ 12
    2.3.2 Implicitly-defined workflows .......... 13
    2.3.3 Shell interface ................ 14
    2.3.4 Grid-inspired .................. 15
    2.3.5 Usage Model ................... 16
  2.4 Design .......................... 17
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.1</td>
<td>Lightweight Computation</td>
<td>17</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Shell Language</td>
<td>19</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Locality Sensitivity</td>
<td>20</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Administration</td>
<td>21</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Security</td>
<td>21</td>
</tr>
<tr>
<td>2.4.6</td>
<td>Dynamic Scheduling</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>Performance-oriented architecture</td>
<td>23</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Compilation</td>
<td>23</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Partitioning</td>
<td>24</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Execution</td>
<td>26</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Disk Parallelism</td>
<td>28</td>
</tr>
<tr>
<td>2.6</td>
<td>Experiment</td>
<td>30</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Setup</td>
<td>30</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Ideal Performance</td>
<td>33</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Overall Performance</td>
<td>34</td>
</tr>
<tr>
<td>2.6.4</td>
<td>Clustered and Multi-core Scalability</td>
<td>34</td>
</tr>
<tr>
<td>2.6.5</td>
<td>Clustered Partitioning</td>
<td>37</td>
</tr>
<tr>
<td>2.6.6</td>
<td>I/O Dependence</td>
<td>37</td>
</tr>
<tr>
<td>2.6.7</td>
<td>Overhead and Limitations</td>
<td>39</td>
</tr>
<tr>
<td>2.7</td>
<td>Conclusion</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Shell Compilation</td>
<td>41</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>41</td>
</tr>
<tr>
<td>3.2</td>
<td>Shell Language</td>
<td>42</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Definition</td>
<td>42</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Purpose</td>
<td>42</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Features</td>
<td>43</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Execution Model</td>
<td>45</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Limitations</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Shell Compilation</td>
<td>46</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Applicability</td>
<td>46</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Definition</td>
<td>48</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Semantic differences</td>
<td>49</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Advantages</td>
<td>51</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Challenges</td>
<td>54</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Execution Model</td>
<td>56</td>
</tr>
<tr>
<td>3.4</td>
<td>Prototype</td>
<td>58</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Application domain</td>
<td>58</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Design goals</td>
<td>59</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Features and Limitations</td>
<td>60</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Experiences</td>
<td>60</td>
</tr>
<tr>
<td>3.5</td>
<td>Discussion</td>
<td>63</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Syntax subset</td>
<td>63</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Partial evaluation</td>
<td>64</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Filename Abstraction</td>
<td>66</td>
</tr>
</tbody>
</table>
3.5.4  Files as Variables ........................................... 67
3.6  Related Work .................................................... 68
3.7  Conclusion ....................................................... 69

4  I/O-Constrained Efficiency and Parallelism 71
4.1  Introduction ....................................................... 71
4.1.1  The Problem of I/O ........................................... 71
4.1.2  Defining I/O Performance .................................... 72
4.2  Performance Characterization .................................. 72
4.2.1  Test Setup ...................................................... 73
4.2.2  Multicore ....................................................... 74
4.2.3  Multicomputer .................................................. 77
4.2.4  Disk Parallelism ................................................ 78
4.3  Scheduling ........................................................ 82
4.3.1  Related Work .................................................... 82
4.3.2  Design .......................................................... 83
4.3.3  Operation ....................................................... 84
4.3.4  Possible Refinements ........................................... 88
4.4  Partitioning ........................................................ 89
4.4.1  Concepts ......................................................... 89
4.4.2  Related Work .................................................... 91
4.4.3  InSeP Algorithm ............................................... 95
4.4.4  Complexity Analysis .......................................... 100
4.4.5  Evaluation ..................................................... 101
4.5  Conclusion ........................................................ 106

5  Analytical Extension 108
5.1  Hardware Advancements ....................................... 108
5.1.1  Increased Network Bandwidth ............................... 108
5.1.2  Increased Core Multiplicity ................................ 109
5.1.3  Recommendations ............................................. 109
5.2  Software Advancements ........................................ 110
5.2.1  Application Parallelism ..................................... 110
5.2.2  Filesystem Parallelism ....................................... 111
5.2.3  Recommendations ............................................. 112

6  Areas of Further Study 113
6.1  Resource Constrained Performance Modeling ............... 113
6.1.1  Introduction .................................................... 113
6.1.2  Single-Factor Performance .................................. 114
6.1.3  Multiple-factor Performance ................................ 116
6.1.4  Finite resource modeling .................................... 118
6.1.5  Conclusion ..................................................... 119
6.2  Pervasive Shell Compilation .................................... 119
6.2.1  Definition ...................................................... 119
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A sample script</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>Simplified timeline for a SWAMP task</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>A parallelizable script that searches for a parameterized string</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Components of total wall-clock time for the benchmarks</td>
<td>23</td>
</tr>
<tr>
<td>2.5</td>
<td>Simplified flowchart (maybe not needed with the timeline)</td>
<td>27</td>
</tr>
<tr>
<td>2.6</td>
<td>Benchmark 1: resample–Data resample</td>
<td>31</td>
</tr>
<tr>
<td>2.7</td>
<td>Benchmark 2: ipcc–Data intercomparison</td>
<td>32</td>
</tr>
<tr>
<td>2.8</td>
<td>Plot of ipcc data(refined)</td>
<td>32</td>
</tr>
<tr>
<td>2.9</td>
<td>Speedup by node count for PPN ∈ {1,2,3,4}</td>
<td>34</td>
</tr>
<tr>
<td>2.10</td>
<td>Speedup in three cluster configurations</td>
<td>36</td>
</tr>
<tr>
<td>2.11</td>
<td>ipcc speedup by PPN for various cluster configurations</td>
<td>38</td>
</tr>
<tr>
<td>3.1</td>
<td>Graphical representation of a simple script dataflow</td>
<td>47</td>
</tr>
<tr>
<td>3.2</td>
<td>Steps in shell compilation</td>
<td>48</td>
</tr>
<tr>
<td>3.3</td>
<td>Filenames can be rewritten to lessen disk access</td>
<td>52</td>
</tr>
<tr>
<td>3.4</td>
<td>Ratio of computation time(gray) to input fetch time(black) for two scripts</td>
<td>61</td>
</tr>
<tr>
<td>3.5</td>
<td>Speedup vs. processor count</td>
<td>61</td>
</tr>
<tr>
<td>3.6</td>
<td>Speedup vs. node count</td>
<td>62</td>
</tr>
<tr>
<td>4.1</td>
<td>Speedup vs. Process Parallelism</td>
<td>74</td>
</tr>
<tr>
<td>4.2</td>
<td>ipcc and resample 1x1 Statistics</td>
<td>75</td>
</tr>
<tr>
<td>4.3</td>
<td>ipcc and resample 1x6 Statistics</td>
<td>76</td>
</tr>
<tr>
<td>4.4</td>
<td>Speedup vs. Multicomputer Parallelism</td>
<td>77</td>
</tr>
<tr>
<td>4.5</td>
<td>Speedup vs. Multicomputer Parallelism</td>
<td>78</td>
</tr>
<tr>
<td>4.6</td>
<td>ipcc and resample 4x1 Statistics</td>
<td>79</td>
</tr>
<tr>
<td>4.7</td>
<td>ipcc and resample 1x1 w/o Explicit Caching Statistics</td>
<td>80</td>
</tr>
<tr>
<td>4.8</td>
<td>ipcc and resample 1x4 w/o Explicit Caching Statistics</td>
<td>81</td>
</tr>
<tr>
<td>4.9</td>
<td>Illustration of prolific commands</td>
<td>86</td>
</tr>
<tr>
<td>4.10</td>
<td>Abstracted workflow from ipcc benchmark</td>
<td>90</td>
</tr>
<tr>
<td>4.11</td>
<td>Simplified workflow graph</td>
<td>91</td>
</tr>
<tr>
<td>4.12</td>
<td>KLFM bipartitioning result</td>
<td>92</td>
</tr>
<tr>
<td>4.13</td>
<td>A workflow that does not benefit from partitioning</td>
<td>93</td>
</tr>
<tr>
<td>4.14</td>
<td>DSC partitioning result</td>
<td>94</td>
</tr>
<tr>
<td>4.15</td>
<td>Step 1: Find roots</td>
<td>95</td>
</tr>
<tr>
<td>4.16</td>
<td>Step 2: Find root clusters</td>
<td>97</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>4.17</td>
<td>Step 3: Compute intersections</td>
<td>97</td>
</tr>
<tr>
<td>4.18</td>
<td>Step 4: Eliminate overlap</td>
<td>98</td>
</tr>
<tr>
<td>4.19</td>
<td>Step 5: Connect resultant clusters</td>
<td>99</td>
</tr>
<tr>
<td>4.20</td>
<td>Performance at command-granularity and cluster-granularity</td>
<td>102</td>
</tr>
<tr>
<td>4.21</td>
<td>ipcc 4x4 Clustered Execution Statistics</td>
<td>103</td>
</tr>
<tr>
<td>4.22</td>
<td>resample 4x4 Clustered Execution Statistics</td>
<td>104</td>
</tr>
<tr>
<td>B.1</td>
<td>Function Signatures</td>
<td>138</td>
</tr>
<tr>
<td>B.2</td>
<td>Description</td>
<td>139</td>
</tr>
</tbody>
</table>
LIST OF TABLES

2.1 Computational performance sorted by CPU count. (Input Rate= \( \frac{\text{Input size}}{t_{\text{compute}}} \). \( t_{\text{compute}} \) does not include parse/workflow generation overhead: 
\( \approx 300 \) seconds for \textit{resample} and \( \approx 0.5 \) seconds for \textit{ipcc}.). . . . . . . 35
I would like to thank my advisor, Dr. Stephen Jenks, who began supporting my
doctoral journey even before I arrived at this university. His encouragement and
confidence in my abilities has been especially crucial to developing an ability for
creative, independent, and significant scientific research. His critique of my research
and its writings has always been fair, constructive, and invaluable.

I would like to thank Dr. Charlie Zender not only for his introduction to a compelling
research area, geoscience computing, and its corresponding funding, but also for his
great help in focusing my research as it applies to non-computer scientists. His well-
timed humor has helped keep my dissertation work in perspective. Dr. Pai Chou has
also helped me in numerous ways, helping me search for a research topic, rigorously
critiquing my work, and adopting my as a surrogate student.

I would like to thank the EECS administrators for their exceptional patience, gen-
erosity, and particularly their unofficial coffee break support.

I have been blessed with membership in several choral groups, and I would like to
thank them for providing me with creative outlets that have comforted and inspired
me at the low and high points of research. I have also been blessed by faith groups,
particularly the Mustard Seeds, which have regularly fed my soul.

I have made many friends in this journey, and I would like to thank them for their
presence and perhaps unconscious achievements in maintaining my sanity and hap-
piness in this long process.

Financial assistance for this work was provided primarily by grants ATM-0231380
and IIS-0431203 from the National Science Foundation.

I acknowledge the modeling groups, the Program for Climate Model Diagnosis and
Intercomparison (PCMDI) and the WCRP’s Working Group on Coupled Modelling
(WGCM) for their roles in making available the WCRP CMIP3 multi-model dataset.
Support of this dataset is provided by the Office of Science, U.S. Department of
Energy.
CURRICULUM VITAE

Daniel Liwei Wang

EDUCATION

Doctor of Philosophy in Computer Engineering 2008
University of California, Irvine

Master of Science in Electrical Engineering 2002
Stanford University

Bachelor of Science in Engineering 1999
Duke University

TEACHING EXPERIENCE

Teaching Assistant
University of California, Irvine Irvine, California
Systems Software¹ (EECS111) Fall 2005
Digital Logic² (EECS31,EECS31LB) Summer 2005
Computer Organization² (EECS112,EECS112L)
Computer Organization Lab³ (EECS112L) Spring 2005
Intro to Programming⁴ (EECS012) Winter 2005
Systems Software¹ (EECS111) Fall 2004

Teaching Assistant
Stanford University Stanford, California
Computer Architecture⁵ (EE282) Fall 2001
Computer Organization⁶ (EE182) Spring 2001
Computer Architecture⁵ (EE282) Winter 2001
Computer Architecture⁷ (EE282) Fall 2000

¹ Prof. Stephen F. Jenks; ² Prof. Alireza Kavianpour ³ Prof. Pai Chou; ⁴ Prof. James Arvo; ⁵ Dr. Andrew Wolfe; ⁶ Tom Fountain; ⁷ Prof. Mendel Rosenblum.

REFEREED CONFERENCE PUBLICATIONS


**EXTENDED ABSTRACTS**


**OTHER PRESENTATIONS**


PAPERS UNDER REVIEW


WORK EXPERIENCE

**Research Software Developer**
Implemented BaBar event store redesign. Speed and memory optimization for object schema. Designed and implemented load-on-demand infrastructure for BaBar event data. (Supervisor: Jacek Becla)

**Research Associate**
OODBMS (Objectivity) measurement and optimization of BaBar event storage.
(Supervisor: Jacek Becla)

**Software Engineer Intern**
Inktomi Corporation, *Foster City, CA* Jun.-Sep. 2001
XML Search engine development—Implemented document compression, caching, numbered-field detection (Supervisor: Ram Subbaroyan)

**Software Engineer Intern**
Inktomi Corporation, *Foster City, CA* Jun.-Sep 2000
Search core dev.—Built non-alphanumeric word parser for searching, crawling, and indexing. (Supervisor: Ram Subbaroyan)

**Junior Programmer Co-op**
Built an automated multi-computer regression testing tool for IBM Payment Server 1.2. (Supervisor: Dallas Wrege)

PROFESSIONAL MEMBERSHIPS

Institute of Electrical and Electronics Engineers (IEEE)
Association for Computing Machinery (ACM)
American Geophysical Union
American Meteorological Society
ABSTRACT OF THE DISSERTATION

Compilation, Locality Optimization, and Managed Distributed Execution of Scientific Dataflows

By

Daniel Liwei Wang

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2008

Professor Stephen F. Jenks, Chair

In order to address the quickly-growing amount of data and the growing desire to share and use each other’s data, this research has made three major contributions. One, shell compilation is introduced as a feasible method for optimizing, sandboxing, and porting shell scripts, which are programs of programs. The application of standard compilation techniques at this higher-level is described, noting the new semantic differences and potential benefits (automatic program-level parallelism) that arise. Two, the ability to compile scripts is applied in geoscience to automatically convert scripts to scientific workflows, resulting in the ability to transparently distribute computation to remote data servers and reduce or eliminate unnecessary data download. The resulting system, the Script Workflow Analysis for MultiProcessing (SWAMP) system dynamically schedules and executes workflows, dispatching commands among cluster machines paying particular attention to data locality and minimizing internal data transfer—a feature particularly important for data-intensive workloads. Performance is shown effective in real geoscience data reduction analysis scripts. Third, the characteristics of I/O-constrained workloads are analyzed and described, along with a technique for explicitly caching files in-memory and a new partitioning algorithm, Independent Set Partitioning (InSeP), whose simple high-level approach based on set
operations can be applied on dynamically-scheduled workflows.
Chapter 1

Introduction

1.1 Preface

With technology advancing at an unquestionably rapid, amazing, and almost predictably surprising pace, one may be surprised to learn that the ever-present quest for “better and faster” is slowing.

The computing field has seen tremendous advancement in the last twenty years, since the introduction of microprocessors made general-purpose computing available to everyone. Every successive year has brought advancements in hardware, software, and networking. Hardware has gotten faster, running existing software faster and enabling more advanced software. New programming paradigms have been developed to manage the expanding complexity of new software. Networking throughput has increased by orders of magnitude. Applications have benefited without programmer effort, for the most part. Although hardware changes have occasionally necessitated slight programming changes, programmers needing more performance in their applications needed only to wait for the next hardware generation.
This free lunch is now over. The computing community has now entered an era where existing application performance increases slowly with hardware. Clock speeds and transistor density have been hampered by semiconductor physics which operate differently at today’s 65nm scales than the 250nm scales of ten years ago. Although the 3GHz clock speed of high-end microprocessors is far higher than their counterparts ten years ago at 500MHz, they are roughly the same as those from five years ago.

1.2 Overview

As interested consumers of computing technology, scientists constantly pursue larger and larger problems, at ever increasing resolution and precision. Plunging data storage costs have made it practical to acquire larger datasets, but efficiently processing them remains a problem.

This research brings scalable, parallelized high-performance data-intensive computing to the desktop, allowing users to utilize its benefits with minimal involvement. Motivated by the problem of large-scale remote data reduction and analysis in the geosciences, it exploits inferred script-level and domain-specific parallelism on multiprocessor and multicomputer systems. The I/O constraints of data-intensive workflows is studied, and novel optimizations for these workloads are presented, balancing computational and bandwidth efficiency while hiding as much of the details as possible. Shell compilation allows scientists to reuse existing scripted analyses, applying the idea that the best learning curve is no curve at all.
1.3 Background and Related Work

1.3.1 The Geoscience Domain

Earth system science analysis commonly involves two types of computation: climate model simulation and data reduction. The first type produces large datasets of variables’ values over grids of points over the earth. The first type is typically produced in batch fashion, having a runtime ranging from several hours to one or more days. These climate models are typically coded in Fortran and run at large data production sites. Simulation often produces large amounts of data, with some scientists’ runs producing 65GB in a couple days. This research targets the problem of analyzing and processing the resultant data volume, whose sheer volume makes high performance computing techniques desirable, but remote data transfer far from trivial.

1.3.2 Parallel Programming

Whereas programming is the means for humans to encode behavior, parallel programming additionally enables humans to encode parallel behavior, i.e. instances where actions may proceed simultaneously. Parallel programming exists in computing systems on several levels.

At the lowest level, microprocessor instruction sets may allow specification of parallel instructions, facilitating instruction-level parallelism (ILP). Examples of these instruction sets are those of the Very Long Instruction Word (VLIW) family [20], of which Intel’s IA-64 [27] is the most recent variety. Parallelism in this form is specified explicitly, in contrast to the more popular implicit ILP detected automatically by current microprocessors. Once promoted as the next inevitable step for performance,
VLIW architectures are now all but ignored—compiler technology that would unlock their performance potential never materialized.

At a higher level, programming languages may facilitate parallel specification. Parallel languages such as NESL[6] have been implemented, as have parallel versions of popular languages, such as Unified Parallel C [10], but none have become popular. Functional languages such as Haskell[28] have been extended with parallel features[32], but have also not been popular. Dataflow languages, which focus on specifying progressions of data through operations rather than sequences of operations that coincidently operate on data, are naturally parallel like functional languages, but have also not been adopted(TFDL[49], Id, SISAL[38]). Parallel languages have been considered “inevitable” for decades, but continue to be ignored by most programmers.

A popular approach to parallel programming is the usage of parallel libraries like OpenMP[14] and MPI[23] in mainstream general purpose languages like C and Fortran. Though they are often maligned as kludges, their approach allows the easiest transition to parallelism since it allows the greatest code reuse. This approach is dominant in supercomputing applications, but used little elsewhere due to its complexity.

For general applications, the dominant form of parallel programming is through multithreading. This approach, though most used, is also perhaps the most complex and error-prone to use, since application logic must be decomposed manually into independent components and synchronized with custom code.

With so many means of parallel programming available, and parallel architectures studied almost as long as their sequential counterparts, the relative disuse of parallel programming is more than a little odd. This can be attributed to the proposition that parallel programming is hard. The only forms of parallelism that have become ubiqu-
uitous have been those where no explicit parallel specification was required. Where explicitly parallel instructions have failed, techniques for automatic ILP have flourished. It is naturally difficult for humans to think in parallel terms—while sequential logic has many metaphors in daily life, parallel logic has few, aside from vectors.

### 1.3.3 Code Mobility

Traditionally, the role of data has been seen in terms of input and output to computation. Indeed, most programming paradigms are based around sequences of operations parameterized with desired inputs and outputs. Object-oriented programming shifts the model towards passing messages to objects (structured data), but stops short of a model of code transfer rather than data transfer. This computation-centered model is appropriate where it is difficult to move code (for reasons of portability, compatibility, bulk, or a combination), but easier to move data.

The idea of code mobility is not new. “Web 2.0” browsing has well-established the practice of downloading Java, Javascript, or Flash code from servers for client-side execution. Database management systems are key examples where code is regularly submitted to servers for execution. In these cases, the cost of data movement is too high in terms of latency or throughput, and languages have been created especially to be portable and mobile.

### 1.3.4 High-Performance Scientific Computing

High-performance computing is typically defined as applications of computing technology on computationally expensive problems. This focus on computational power is seen in supercomputer rankings, where the most powerful computers are ranked in
terms of floating point operation execution rate (flop/s). I/O performance is often mentioned but only as it is useful in explaining the gap between measured and theoretical flops. This focus has led to impressive achievements in computation, with peak throughput sometimes growing faster than Moore’s Law, and recently reaching 1 petaflop/s ($10^{15}$ flops). Simulations can be run at ever finer resolutions, producing ever increasing amounts of data. However, the resulting abundance of data difficult to manage. While data gathering and production has received much attention, data analysis has received relatively little, due to its lower computational requirements which make their solutions less interesting for researchers.

1.3.5 I/O Parallelism

There is little work available on I/O parallelism. Research on I/O performance has focused on peak throughput or latency, just as research on computational performance has focused on achieving peak throughput. However, while CPU performance can be efficiently divided to provide performance to multiple simultaneous tasks without significant loss in overall throughput, I/O performance more typically experiences significant degradation with increasing task multiplicity. Magnetic disk is the archetypical example, and the mitigation of this degradation is only possible through sophisticated algorithms in modern operating systems and controller hardware. Redundant Array of Inexpensive Disk (RAID) technology [41] is a popular example of parallelizing disk access that improves peak throughput for serial accesses but otherwise reduces a disk array’s potential for concurrent access.
1.4 Strategies

The contributions of this thesis can be organized in three larger themes.

1.4.1 Geoscience Data Analysis and Reduction

Geoscience data analysis represents an class of workloads whose properties suggest suitability for high performance computation (HPC), but whose needs have largely been ignored (although geoscience simulations are well-supported). Chapter 2 will discuss the particular needs of geoscience data analysis, explaining exactly how its data-intensive, large-scale needs are not addressed by existing HPC solutions. I will describe an implemented system, which addresses those needs, and conclude with a treatment of why this class of workloads matters for not only all scientific disciplines, but any discipline which works with data on a large scale.

1.4.2 Shell Compilation

In appreciation of the reluctance of the programming community to pervasively adopt parallel languages, I propose a method to automatically optimize and parallelize programs of programs, that is, workloads specified by existing shell scripts. In Chapter 3, I define what shell compilation means and what it does not mean. I describe its enormous potential for facilitating high performance, explain why its study has been largely ignored, and show how its difficulties can be overcome for a significant class of applications.
1.4.3 I/O-Constrained Efficiency and Parallelism

As technological advancement continues to provide more ways to observe and produce data more quickly, data-intensive workloads and efficient methods for their execution become increasingly important. In Chapter 4, I describe the ways in which computing (both high-performance and mainstream) and other technology have contributed to a relative imbalance between produced and analyzed data. Using examples from geoscience, I analyze the ways in which data-intensive workloads are unsuitable for existing high-performance computation systems. In light of these factors, a method for avoiding I/O will be described, along with a novel partitioning algorithm which allows workflow schedulers to minimize data transfer and achieve high-performance in these workloads.
Chapter 2

Geoscience Data Reduction and Analysis

2.1 Introduction

While new technologies have spurred a glut in produced data, geoscience data utilization remains hampered by the physical separation between producer and user—data transfer and analysis is always too slow, too expensive, or both. Suppose a researcher is interested in a particular property X, which can be easily computed over her terabyte dataset in a few minutes. Transfer costs make computing over an overseas colleague’s own terabyte dataset already cumbersome, and computing over a consortium’s hundred terabyte collection practically impossible. In this paper, we explore a method to make the former easy and the latter possible. Our system provides a method of safely and efficiently executing script-defined analyses on remote data.

By making large scale geoscience data analysis and reduction accessible to scientists with only modest network and local computational capacity, our approach will enable
geoscientists to evaluate more and larger data sets, enhancing their research ability. Our approach does not require such scientists to learn new, complicated toolsets, but rather enhances script-based data reduction and analysis in common use today.

2.2 Problem

The growing gap between the amount of data produced and the amount analyzed is a significant problem. Computing advances have made it possible for individual scientists to run their own simulations with their own parameters and data at wider scales and finer resolutions, but data analysis remains hampered by the difficulty of moving bulky data and the scarcity of software tools that scale to large volumes. Observed data from local and remote sensing has grown in volume similarly, while presenting additional challenges like new sensing types, environments, and scales.

Most geoscience analysis and visualization tools, for example, are designed to operate with single variables or two-dimensional plots, but struggle when performing larger, bulk data processing. With individual scientists not uncommonly producing nearly 100 gigabytes in single runs, handling large data volumes has become increasingly important.

2.2.1 Large data volume

Although network bandwidth records are broken regularly, dataset sizes have also grown, keeping data movement a challenge. Because managing high volumes is so difficult, scientists typically avoid the problem, executing high-resolution simulations infrequently, either as followups to exhaustive lower-resolution testing or as team efforts in larger collaborative experiments. Analysis is done tediously, often using
custom Fortran code to analyze or to reduce the data to scales manageable by other tools. For example, 100 gigabytes of data can be generated in a few days of atmospheric simulation, but such runs are infrequent due to the difficulty in managing the results. Because each study is unique, each correspondent body of processing code is frequently unique and written with performance optimization as a low priority. As a result, scientists’ work is hampered by an implicit directive to keep data sizes small enough for their workstations to handle. Geoscience analysis and visualization tools often assume that the entire dataset can be kept in random-access memory, aiming for interactive usage rather than volume processing.

Even when large data volumes are produced and centralized for larger collaborative studies such as those undertaken by the Intergovernmental Panel on Climate Change (IPCC), usage is limited by the large effort required for users to download and manage the data. Working around this problem requires extreme measures—the Sloan Digital Sky Survey [50] distributed their 1.3 terabyte compressed dataset by shipping complete computers.

### 2.2.2 High data-intensity

Recent work on high-performance computing research has recognized the importance of data-intensive workloads [57], but serious study remains rare regarding high data-intensive workloads where computation is small relative to data size. High data-intensive workloads can be defined as having very low (between 0 and 10) floating point operations per byte of input data (flop/byte). For example, computing an average, a common task in geoscience, requires less than one flop/byte. In these workloads, computational costs are small relative to data movement costs due to disk or network I/O. Whereas traditional compute-bound execution performance is at most
influenced by I/O considerations, performance is nearly defined by I/O considerations in these high data-intensive cases.

2.3 Approach

Our solution avoids the data movement problem by enabling scientists to use existing desktop-oriented analysis tools on remote data centers, and optimizes execution with techniques sensitive to data movement. This strategy moves data analysis and reduction operations to the remote data center where the data reside, taking advantage of the computational, networking, and storage infrastructure of the data center to perform the I/O-centric data-intensive operations quickly and efficiently. By exploiting locality and parallelism, our system dramatically improves overall performance and significantly reduces data transfer sizes, which benefits both end users and the data centers.

2.3.1 File granularity

Many desktop analysis tools like [25, 30] are able to access and manipulate data in terms of individual values at grid points. Complementary to these are those tools that work larger granularities, that help provide insights into trends and statistics in the data that help scientists decide when to apply the fine-grained tools. The files exchanged in geosciences represent logical datasets, and are frequently broken apart to ease storage management or to cope with technological limitations. Data operations of significant intensity are well-matched to files or sets of files. Granularities smaller than files are easily hyperslabbed from files. Files, therefore, are a logical level of granularity for a system designed for bulk processing.
In some cases, files are aggregated into logical datasets that are published by data servers such as OPeNDAP[11] or THREDDS[17]. These help free scientists from managing inconsistencies in file splitting used by different colleagues. Fortunately, these aggregations are easily mapped to files and are thus complementary to file-based analysis.

### 2.3.2 Implicitly-defined workflows

Workflows are differentiated from other batch workloads through the interdependency of their constituent parts. Because they contain internal dependencies, they may perform poorly on generic grid frameworks designed for large volumes of independent tasks. Workflows can be executed on grid workflow engines, which can utilize the dependency information to provide maximum parallel performance over grid resources. However, careful work is required to design and specify each workflow with each particular grid framework. With a detailed specification, the workflow scheduler can then provide an optimal schedule. [16] describes a system which leverages job performance characteristics to produce efficient execution plans.

Unfortunately, workflow specifications are too complex for individual scientists to construct. While they rarely possess grid workflow hardware, they commonly perform workflow processing, that is, tasks composed of interdependent components, in the form of custom scripts that they execute on whatever data they are interested in. Our approach directly leverages their scripts to construct implicit workflows. Once converted, scripts can benefit from workflow optimization, enabling additional parallelism beyond application-level multithreading.
2.3.3 Shell interface

Instead of choosing data-focused domain language such as one used in GrADS[30] or FERRET[24], we chose shell language (used by standard shell command interpreters). Shell language is used to define an ordered sequence of command invocations and operates naturally at the file-level granularity mentioned above.

The choice of shell language offers three key advantages over the use of workflow languages: universal familiarity, applicability, and scalability. Familiarity comes naturally, since nearly every user who processes data of significant volume regularly utilizes shell scripts of some form to automate processing. The remaining users who do not are familiar with executing programs at command-lines. In leveraging these existing scripts, our approach minimizes retraining, a common and significant barrier of adoption in many computing frameworks.

Shell language is applicable for a wide variety of tasks, due to its ability as a glue language, which connects programs written in other languages. Though uniqueness in scientists’ usage is seen in custom Fortran (or other language) code and in personal scripted sequences, internet-scale collaboration has resulted in common data formats and common toolsets for manipulation. By supporting a popular tool set, the netCDF Operators (NCO)[59] which support processing the most common format for exchanging geoscience data, netCDF[43], our approach covers a wide variety of geoscience data analysis and reduction tasks.

Scalability is achieved primarily by parallelization and data locality optimization. Script-detected parallelization allows performance to scale with parallel hardware. This provides an additional level of parallelization beyond the programs’ own use of parallel techniques such as OpenMP and MPI. Should the programs be optimized further for performance[60], a script based system benefits automatically. Combining
both script-level and application-level parallelism better positions such a system to exploit increasingly parallel hardware.

2.3.4 Grid-inspired

Grid technologies represent the latest promise in high-performance computing. By flattening access to disparate, homogeneous resources, grid technologies should yield such an abundance of raw computational power which should be irresistible to anyone desiring high performance computing. While grid projects such as [12], [39], [51], and [44] each enjoy success in their own goals, no single implementation has dominated. Though all aim for generality, all have features that are tuned to their own application domains, and are not sufficiently generic enough for each other’s purposes. All target large-scale computation, but only one maintains an interface simple enough for lightweight tasks from casual users[51]. The few that are aware of data locality require specialized tools to construct tasks.

In some ways, grid technologies are unsuitable to computation that is largely data-bound instead of compute-bound. Indeed, an underlying assumption is that the availability of larger computational pool will enhance performance, an assumption which does not hold when a task performance is dependent on disk transfer rates rather than CPU speed. These data-bound tasks are generally under-researched, given that high performance computing prizes teraflops rather than secondary storage (disk) bandwidth. Data-bound tasks are little studied, because users do not perform them often, which is because data-bound performance is ignored in their tools, which is because the developers lack references for data-bound task performance, and so on. Some computational grid models consider data movement costs and schedule data movement coupled with computation[57], but still assume that computation
cost dominates.

Our solution borrows the grid concept of distributing work among loosely coupled machines, but differs from a grid computation service in two ways. First, it provides a computational service necessarily bound to data locality, with an interface similar to grid computational services, but tailored for data rather than computational performance. Two, its execution model groups execution to minimize data movement. While scaling to leverage computer cluster resources, it discards the flexible resource mapping because its workload does not benefit from the availability of remote computational resources. Its tasks can be usually be completed in less time than the time to stage the inputs to a large, fast, but remote compute cluster.

2.3.5 Usage Model

Because terascale data remains too cumbersome to download, a practical solution must reduce or eliminate the initial data download. This suggests that the computation itself should be relocated to where the data itself resides. Therefore, our solution implements a computational service to complement existing data access service. By re-interpreting unmodified desktop analysis shell scripts, scientists’ own custom analyses can be safely relocated to the data server, without requiring user accounts, familiarity with foreign environments, or parallel or advanced computing skill.

The sample script in Figure 2.1 computes the time-varying spatial mean and deviation from the year 2000 for two different climate models, along with the ensemble average and its deviation. It executes unmodified on a desktop as well as on our remote computing service. Because the system re-interprets the script, it can ensure that only safe programs are referenced and only permissible files are read, and that program
for mdl in ccma_cgcm3_1 ncar_ccsm3_0; do
    ncwa -a lat,lon sresa1b_$mdl.nc avg_$mdl.nc
    ncwa -d time,0,11 avg_$mdl.nc 2000_$mdl.nc
    ncdiff avg_$mdl.nc 2000_$mdl.nc anm_$mdl.nc
done

# Create model ensemble
ncea avg_*.nc sresa1b_avg.nc

# Ensemble mean of year 2000
ncwa -d time,0,11 sresa1b_avg.nc sresa1b_avg_2k.nc

# Create ensemble anomaly
ncdiff sresa1b_avg.nc sresa1b_avg_2k.nc sresa1b_anm.nc

Figure 2.1: A sample script

outputs are safely mapped to isolated areas. It is worth noting that [51] provides similarly simple computational service, but requires tasks to be specified in a custom Perl-like language[52].

2.4 Design

We designed our system, which we call SWAMP (Script Workflow Analysis for MultiProcessing), as a computational service designed to be deployed alongside a data service. SWAMP was initially implemented as a plugin to a leading geoscience data server, OPeNDAP, but outgrew the the protocol and operating limitations in OPeNDAP Server 3. Basic SWAMP operation is illustrated in the timeline diagram in figure 2.2.

2.4.1 Lightweight Computation

SWAMP is distinguished from existing grid frameworks and data servers in how and what sort of computational service it provides. By providing a scripting interface, it offers analytical function beyond simple subsetting that is provided by existing data
servers. While its simple, lightweight computational capability may seem unjustified in an era of commodity high-performance workstations, its capabilities are matched towards providing access to scientific results from input data that is too large to download. Its parallelization capability enhances scalability, and its sandboxed and lightweight execution make it more feasible for open-access environments than generic grid computation systems. Scripts like the one illustrated in figure 2.1 can be submitted to a SWAMP instance using a simple SOAP[7] or XML-RPC[56] call which returns an identifying token to be used to check job status and download results.
2.4.2 Shell Language

Computation in SWAMP is specified with the same syntax as ordinary command interpreters, shell language. Direct, unmodified re-use of existing user scripts is therefore possible in SWAMP. While other parallel systems require users to explicitly annotate their code to exploit parallelism, SWAMP reveals how much can be gained in situations where the user does not. It intentionally provides the analysis subset of a desktop environment so that remote data can be processed just as simply. Debugging a SWAMP script, for example, can usually be done outside SWAMP using the operating system’s own shell.

SWAMP supports most syntactical features used by scientists in analysis scripts. Variables can be defined and referenced. Control structures like for-loops and conditional branches can be used, easing volume processing and portability. Safe shell “helper” programs like `seq` and `printf` can be used, as can filename wildcards like `?` and `*`. Wildcards are intelligently expanded using the script’s local context, properly accounting for files written earlier. Parameter variables, i.e. those referenced by `$1` or `$2`, will be supported in a later revision. For security and implementation tractability, program selection is restricted to a set of programs whose argument and file input/output semantics are programmed into the system. Without semantic understanding, the system would not be able to detect inter-command dependencies or to sandbox script execution. In its current implementation the program set is composed of the NCO tools, which are regularly used in the geoscience domain.

The use of shell syntax allows the system broader applicability than geosciences. The system can operate in any domain where scripts are used by adding support for its programs, given that the programs are well-behaved, i.e. their inputs and outputs are files, and can be determined solely by argument parsing. Once the system is in place,
for f in *; do
grep $1 $f
done

Figure 2.3: A parallelizable script that searches for a parameterized string.

parallel execution of scripts becomes a question of “Why not?” rather than “Why?”
As an example, UNIX grep could be parallelized in such a fashion without significant effort, once written in a script as in Figure 2.3.

2.4.3 Locality Sensitivity

Without a locality-sensitive design, a system cannot provide computational service to large datasets, because large data sizes make input pre-staging and output post-staging too expensive to be practical. An initial design goal of our system is to enable user analyses to scale to large data volumes. Because long-haul network bandwidth will always fall short with respect to data volumes produced from observation or simulation, any solution for providing computation on such data must not move data unless absolutely necessary. Such is the primary motivation for providing computational service at a data repository. Additionally, SWAMP’s execution engine dispatches processes with locality sensitivity in two ways. First, it partitions workflows into clusters with reduced inter-cluster dependencies. This minimizes communication between worker nodes. installations. Secondly, clusters are preferentially dispatched to nodes to minimize the need for input file staging. Without these methods, data-intensive performance scales poorly.
2.4.4 Administration

SWAMP has a number of features which ease its deployment both at data centers and lab group installations. The server is implemented in the Python language as a self-contained, standalone daemon accessible over SOAP or XML-RPC protocols. It can operate in clustered configurations and is able to add and remove worker instances without restarting. Each installation is configured by editing one file or two files in the case of clustered instances. For enhanced I/O performance, it can exploit ramdisks on machines with abundant physical memory. Built-in logging facilities can be used for troubleshooting or usage tracking.

2.4.5 Security

Because our system is designed to provide computation access as widely as plain data access, security issues must be considered. SWAMP resists overall system intrusion by using an interpreted language that resists buffer overflows, and by restricting scripts to a small set of well-understood programs. Program arguments must pass basic validation tests, and invocations occur directly without implicit subshell use, making it difficult to invoke untrusted programs. Filenames in scripts are remapped first to logical filenames in static single assignment form[1], a feature which also enhances execution parallelism. Virtualizingfilenames in such a manner not only isolate tasks from each other, but also from accessing system files. Scripts are also executed one at a time, which allows a long-running script to monopolize resources and potentially deny service, but prevents a flood of requests from overloading and crashing the system.
2.4.6 Dynamic Scheduling

Our system schedules tasks on-demand as processing resources are available, whereas other workflow frameworks schedule tasks statically before beginning execution. This choice reflects the practical realities of accepting script-specified user workflows. Static scheduling is able to create more optimal end-to-end workflow execution times, i.e. makespans, but relies heavily on accurate performance prediction. In operational environments, where workflows are used as part of a well-defined processing pipeline, tasks have approximately deterministic execution times, and their performance can thus be predicted for future iterations. However, while each scientist may focus on a limited number of scripts, the number of scripts used by the larger community is large. Therefore, the execution times of the contained commands cannot be predicted without support in each command for some sort of “dry run” mode where its predicted performance is computed[60], along with sufficient metadata to allow dependent programs to predict performance. Such a feature is difficult to implement, and may not be possible for all prospective programs. With this in mind, the more reactive approach, dynamic scheduling, is more appropriate.

The current implementation dispatches script commands for execution on nodes according to each node’s configurable maximum parallel processes per node (PPN). Maximum performance is normally achieved when the number of parallel processes equals the number of physical processors, with an important exception described in section 2.6.6. Another attractive possibility, given the system’s target of data-bottlenecked scripts, is to schedule according to I/O load. By limiting parallel disk access, seek frequency can be reduced to bring disk bandwidth higher to its theoretical maximum sustained read rate.
2.5 Performance-oriented architecture

By providing computation at the data source, our system avoids the primary problem in using large scale remote data: the lengthy data download time. Figure 2.4 illustrates the contribution of data transfer to overall task execution time for our two benchmarks (see section 2.6.1). In both cases it is clear that transfer cost is just as important, if not more important than computation cost. Additional performance benefits come from the following areas.

2.5.1 Compilation

By parsing command semantics, our system is able to transform an otherwise serial script into a directed-acyclic-graph workflow. Each command-line becomes a graph node, and its parent and child nodes are determined by matching input and output filenames. The parser derives these filenames through code that is customized for the supported executables, the netCDF Operators. Because it is intractable to determine argument semantics for all arbitrary programs automatically, support is restricted to programs common in a domain, and thus make the required custom code more tractable. NCO tools are easily combined in scripts to form useful geoscience data reduction and analysis tasks, and support for their semantics alone yields a rich
computational capability.

By treating filenames as variable names and commands as basic operators, more traditional compiler techniques can be applied. Live variable analysis allows the system to intelligently guess which files are results and which are intermediate “temporary” files. “Dead” files can be allocated to memory instead of disk, reducing disk contention during execution and bandwidth when returning results to clients.

2.5.2 Partitioning

Multi-computer performance in earlier implementations suffered from high synchronization overhead between master and worker nodes. Commands were dispatched individually, promoting an even balance of commands on available worker nodes, but limiting scalability due to the frequency of updating the master node with command progress. To reduce this problem, the workflow is partitioned into groups of nodes, while minimizing the dependency between groups. By dispatching work in these partitions, multi-node execution avoids unnecessary data transfer and requires master-worker communication overhead, at the risk of work imbalance from the coarser scheduling granularity.

The partitioning algorithm is as follows. Consider a directed graph $G = (V, E)$ with the set of vertices $V$ (representing script commands) and the set of directed edges $E$ (representing derived dependencies between the commands). Define a root of $G$ as a vertex $r$ which has no predecessors in $G$, where $r \in V, \emptyset = \{ v : v \in V, v \prec r \}$.

- Compute roots of $G$. Let $n_r$ be the number of roots.
- For each root $r_i$, compute its corresponding root cluster $C_{r_i}$, the set including
the root and all of its direct and indirect successors.

\[ C_{r_i} = r_i \cup \{ v : v \in V, v \succ r_i \} \] (2.1)

- For each root cluster \( C_{r_i} \), compute its intersections with other root clusters, resulting in intersection clusters \( C_{r_{i,j}} \).

\[ C_{r_{i,j}} = C_{r_i} \cap C_{r_j} \] (2.2)

(Note that for equation 2.2, computing \( C_{r_{i,j}} \) for \( i, j \in [0, n_r], i < j \) is sufficient since \( C_{r_{i,j}} = C_{r_{j,i}} \) and we are not interested in \( C_{r_{i,i}} = C_{r_i} \).)

- Construct \( C'_{r_i} \) by removing descendants shared with other root clusters. These modified root clusters can be dispatched independently, in parallel.

\[ C'_{r_i} = \{ v : v \in C_{r_i}, v \notin C_{r_{i,j}}, \forall j \} \] (2.3)

- Construct \( C'_{r_{i,j}} \), by removing elements not shared with other intersection clusters. These modified intersection clusters can be dispatched independently of other intersection clusters, but may have dependencies on root clusters.

\[ C'_{r_{i,j}} = \{ v : v \in C_{r_{i,j}}, v \notin C_{r_{x,y}}, \forall x < i, x = i, y < j \} \] (2.4)

This provides \( n_r + |C'_{r_{i,j}}| \) clusters, with \( r \) initial parallel clusters to execute, and up
to $r \times (r - 1)$ non-root independent clusters. If additional parallelism is needed, $C'_{r_i,j}$ can be recursively split according to the original algorithm, since they may have multiple roots. Otherwise if there is only one root, the cluster may be split by cutting the cluster after the first (smallest depth) command which has multiple successors. The algorithm avoids cutting between a parent node and its only-child, because that confers no parallelization benefit and risks penalties from file staging delays if the child were to be scheduled outside its parent’s node.

This algorithm has performed adequately in our benchmarks, although standard bi-partitioning or k-partitioning algorithms were considered. These alternate partitioning algorithms operate on flow networks and are common in the computer-aided design (CAD) community for integrated circuit layout purposes. Though attractive in algorithmic complexity, their minimization constraints were found to be unsuitable. They are designed to balance nodes between partitions for physical layout, while our algorithm focuses solely on branching and “fan-out” points.

### 2.5.3 Execution

Attaining maximum performance for compute-intensive workloads is a well-researched topic in high-performance computing. Our system was designed to reduce or eliminate the prohibitive transfer time cost for scientific data analysis, and since our workloads are correspondingly I/O rather than compute-intensive, maximizing their performance involves different issues.

We define I/O intensive workloads as those with relatively low floating point operation (flop) counts per input byte. For these workloads, parallel performance is limited by data transfer between worker nodes in a cluster, disk contention, and overall disk bandwidth. Management overhead (compilation, scheduling, and dispatch) is an
additional bottleneck, but is only important after I/O optimization.

Figure 2.5 illustrates our simple execution model. Scripts enter the system through a web service RPC interface, and are queued for execution. If no other script is in progress, the script is prepared for execution by parsing, validation, verification, workflow generation, and cluster partitioning. The resultant workflow clusters are then dispatched to the available workers in round-robin fashion, until the list of ready clusters is exhausted or each worker has a number of dispatched clusters equal
to its PPN. Each worker then executes ready commands from its set of clusters, favoring most-recently-ready commands whose inputs have just become available. This most-recently-ready policy exploits cache hotness for performance and shortens temporary file lifetimes so that space in the in-memory filesystem can be reclaimed earlier. As workers complete commands, they report results to the master, except for commands without children outside the cluster–their results are batched and reported when their parent cluster completes. This trades off some accuracy in workflow state by only allowing critical messages that affect global execution scheduling (by causing command clusters to become ready) to be reported synchronously. Batching and deferring less important events drastically reduces master-worker synchronization overhead.

2.5.4 Disk Parallelism

In [53] we showed the sensitivity of data-intensive workloads on disk performance. In those cases, we found that simply parallelizing execution did not improve performance–rather performance was penalized due to the increasing disk seek rate.

Two techniques have been implemented to improve I/O performance in our system. The first, introduced in [53], improves performance by converting a significant fraction (in some cases, the majority) of disk accesses to memory accesses. While single program executions in isolation are difficult to optimize–input data must be fetched and output data must be written–workflows contain much more promise. Internal edges between nodes in the workflow graph (program invocations) denote data that are produced and consumed during workflow execution. While a general-purpose operating system would certainly perform file buffering, our system explicitly avoids disk writing in those cases by utilizing a large in-memory filesystem. Our execution
engine remaps command output files to memory while free space is available, spilling to disk when memory is almost full. Because in-memory filesystem space is so precious, intermediate files there are deleted as soon as they are dead (unneeded for future commands).

The second technique for improving I/O performance is enabled through multi-computer execution and replication. While many cluster installations utilize centralized head-node storage or storage-area networks (SANs), our results show that maintaining input data replicas on cluster nodes yields far greater performance, with aggregate read performance closer to the total available disk bandwidth. By splitting execution among multiple computers, the system can also write output data using the aggregate disk write bandwidth available, rather than contending for write bandwidth on centralized storage.

Despite this advantage, practical installations may find difficulty in providing cluster nodes with private data replicas. In those cases, we believe that aggressive opportunistic caching of data on node-local storage can provide good performance by minimizing concurrent access to a centralized storage system.

Disk reorganization may also improve performance. While magnetic disks have historically been treated as random access devices compared to sequential devices like tape media, processor clock frequencies have increased and memory latencies have dropped, increasing the relative latency for random access and making disks seem more like sequential media. Hsu has proposed a system [26] for reorganizing disk data layout to bring average random read performance closer to sequential read performance. We expect implementation of such a system would also help our workloads, although performance would still be limited to disk bandwidth.
2.6 Experiment

We began development with the hypothesis that there would be significant process-level parallelism inherent in scientific scripts, but without a quantitative estimate. Intuitively, we predicted that eliminating input data download would make the greatest difference, but we expected that exploiting parallelism would have some impact as well.

2.6.1 Setup

We tested our system using two benchmark scripts in both standalone and clustered configurations. The first subsamples high temporal-resolution global wind predictions to obtain the predictions closest to twice-daily satellite overpasses[9], while the second normalizes surface air temperature predictions from sixteen different models for intercomparison. Results from the first are pending publishing, but both are used for climate research.

The first benchmark resample, is conceptually shown in figure 2.6. Its 2MB script of 14,640 unique command-lines represents a practical upper-bound on the length of script expected and can be considered a “stress-test.” The second benchmark ipcc, shown in figure 2.7 illustrates a task which was one of the motivating problems for this work. The task generates global temperature anomaly trends from 19 different simulations from the World Climate Research Programme’s (WCRP’s) Coupled Model Intercomparison Project phase 3 (CMIP3) multi-model dataset and compares each trend versus the mean trend of the entire data ensemble. Requiring approximately 30GB of input data this workflow is computationally simple, and results in roughly 500kB of output data. ipcc output data can be processed further to create
the temperature trend chart shown in Figure 2.8.

The benchmarks were tested on a cluster of dual Opteron 270s with 16GB of memory and dual 500GB SATA drives, running CentOS 4.3 Linux. We tested execution with one master and one, two, or three worker worker nodes, all configured identically. We varied the number of parallel processes per node (PPN) allowed on each node in order to study the benefit of using using more cores (four available per-node) versus the increased I/O contention. Multi-node performance is compared in configurations with and without cluster partitioning in order to better understand scheduling overhead.

Performance is compared with a control setup where the same script is executed at the GNU bash command line. We computed the baseline wall-clock script-execution time including the estimated time to download the input data set. Transfer times
100 years of model data
19 model runs
1 file/run
1 timestep/month

For each run:
- `ncwa` Compute spatial means
- `ncwa` Compute year 2000 mean
- `ncdiff` Compute anomaly vs year 2000

Temperature anomaly trends for 19 simulations and their ensemble mean

Plot model anomalies vs ensemble anomaly

Source data:
- 28,405MB in 19 files
- 463KB in 20 files

Spatial means:
- 90KB in 20 files

Year 2000 means:
- 90KB in 20 files

Finish:
- 452KB in 20 files

Figure 2.7: Benchmark 2: \textit{ipcc}–Data intercomparison

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure2_7.png}
\caption{Benchmark 2: \textit{ipcc}–Data intercomparison}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure2_8.png}
\caption{Plot of \textit{ipcc} data (refined)}
\end{figure}
are estimated assuming 3MiBytes/s \( (3 \times 2^{20}) \) bandwidth, based on NPAD \textit{pathdiag} \cite{37} measurement of 30Mbits/s bandwidth between our workstation at UCI and the National Center for Atmospheric Research (NCAR).

These benchmarks are highly parallelizable, but are not “embarrassingly parallel” in the classical sense. Many of the individual operations, e.g. simple averaging, could be considered “embarrassingly parallel,”, but parallelization at that fine granularity is independent to our method (and can have debatable benefit due to movement costs). Our contribution to parallelism does not consider the semantics of each executable except for its file I/O, and parallelizes execution similarly as machine instructions can be parallelized on superscalar CPUs.

2.6.2 Ideal Performance

In general, the greatest possible speedup of a parallelized implementation relative to a serial implementation of the same algorithm can be computed by applying Amdahl’s law\cite{2}:

\[
\frac{1}{S + \frac{1}{N}},
\]

where \( S \) is the fraction of time spent in serial (or non-parallel) execution and \( N \) is the number of parallel processors applied. For \textit{resample}, the entire workflow is parallelizable into 10 independent flows, and each flow contains a stage where 730 independent subtasks are available. Thus the maximum possible speedup, ignoring I/O, for \( N \leq 10 \) is \( N \), and somewhat less for \( N > 10 \) (a small portion can exploit 730×10 parallelism).

For \textit{ipcc}, the workflow contains a non-parallelizable portion, which should account for approximately 1/20 of execution time. If we assume \( S = 0.05 \), then the maximum speedup should be \( \frac{1}{0.05 + \frac{1}{N}} \), or approximately 3.5, 5.9, 7.7, and 9.1, for \( N = \{4, 8, 12, 16\} \), respectively.
2.6.3 Overall Performance

Both benchmarks show significant benefits from SWAMP. Comparing SWAMP’s performance in a 4-node x 4 core configuration versus the baseline non-SWAMP case, we find that overall end-to-end time can be reduced from approximately 99 minutes to about 10 minutes in the resampling case, and from 165 minutes to 3 minutes in the IPCC case, giving roughly 10x and 64x reductions, respectively. Eliminating transfer time alone accounts for roughly 2x, and 22x, with efficient parallel execution accounting for the remaining savings.

Raw computational performance is shown in table 2.1, excluding transfer transfer time and job preparation overhead (parsing, compilation, and workflow generation), which are both dependent on static workload and not on execution configuration.

2.6.4 Clustered and Multi-core Scalability

Figures 2.9a and 2.9b show plots of speedup versus node counts for resample and ipcc respectively. For completely CPU-bound workloads, ideal scalability should be reflected in linear speedup in the total number of CPUs, without regard to the number
<table>
<thead>
<tr>
<th># CPUs</th>
<th># Nodes</th>
<th>PPN</th>
<th>$t_{\text{compute}}$s</th>
<th>Input Rate MB/s</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>resample</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3202.36</td>
<td>2.69</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1710.29</td>
<td>5.05</td>
<td>1.87</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1264.77</td>
<td>6.82</td>
<td>2.53</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1264.53</td>
<td>6.82</td>
<td>2.53</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1022.67</td>
<td>8.44</td>
<td>3.13</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>987.24</td>
<td>8.74</td>
<td>3.24</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1052.91</td>
<td>8.2</td>
<td>3.04</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>694.84</td>
<td>12.42</td>
<td>4.61</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>744.33</td>
<td>11.59</td>
<td>4.3</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>4</td>
<td>584.29</td>
<td>14.77</td>
<td>5.48</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td>687.97</td>
<td>12.54</td>
<td>4.65</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>487.03</td>
<td>17.72</td>
<td>6.58</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>4</td>
<td>432.41</td>
<td>19.96</td>
<td>7.41</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>3</td>
<td>383.83</td>
<td>22.49</td>
<td>8.34</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>4</td>
<td>317.68</td>
<td>27.17</td>
<td>10.08</td>
</tr>
<tr>
<td><em>ipcc</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>455.55</td>
<td>65.38</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>494.60</td>
<td>60.22</td>
<td>0.92</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>232.57</td>
<td>128.07</td>
<td>1.96</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>496.58</td>
<td>59.98</td>
<td>0.92</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>152.59</td>
<td>195.2</td>
<td>2.99</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>520.33</td>
<td>57.24</td>
<td>0.88</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>246.67</td>
<td>120.75</td>
<td>1.85</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>138.62</td>
<td>214.86</td>
<td>3.29</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>326.07</td>
<td>91.35</td>
<td>1.4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>181.50</td>
<td>164.1</td>
<td>2.51</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>4</td>
<td>323.72</td>
<td>92.01</td>
<td>1.41</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td>139.00</td>
<td>214.28</td>
<td>3.28</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>252.2</td>
<td>118.1</td>
<td>1.81</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>4</td>
<td>235.91</td>
<td>126.26</td>
<td>1.93</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>3</td>
<td>154.98</td>
<td>192.19</td>
<td>2.94</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>4</td>
<td>152.72</td>
<td>195.03</td>
<td>2.98</td>
</tr>
</tbody>
</table>

Table 2.1: Computational performance sorted by CPU count. (Input Rate=$\frac{\text{Input size}}{t_{\text{compute}}}$), $t_{\text{compute}}$ does not include parse/workflow generation overhead: $\approx$ 300 seconds for *resample* and $\approx$ 0.5 seconds for *ipcc*.)
of CPUs per node. In both benchmarks, speedup increases linearly with the number of nodes but only in resample does speedup increase with the number of CPUs (see subsection 2.6.6). From these results, we infer that larger systems with more processor cores per node, or more physical nodes, will have proportionally greater performance, except for extremely I/O bound workflows, where performance benefits come only with more physical nodes that can provide more read and write disk bandwidth.

In figure 2.10, we compare performance in three different cluster configurations while the total number of CPUs is held constant to 4. For resample, we see that performance is roughly constant, meaning that performance is effectively dependent on CPU count and is unaffected by master-worker management effects or resource contention. The 2x2 configuration seemed best, likely due to the increase in available disk bandwidth from 1x4 combined with low file dispersion (which impacts the 4x1 configuration). In contrast, ipcc performance is clearly dependent on node count rather than CPU count for reasons elaborated in subsection 2.6.6.
2.6.5 Clustered Partitioning

In an earlier implementation [54], execution performance suffered from communication between master and worker nodes. As expected, performance improved by using coarser-grained scheduler, which guaranteed that blocks of commands could execute without data staging and thus significantly reduced worker synchronization requirements. Without batching, we observed that while data traffic was reduced, scheduling traffic increased with the higher execution throughput, which limited scalability. In some cases, workers reported results to the master scheduler at an aggregate bandwidth of 500kB/s.

In [54] where a non-partitioned implementation was tested, we observed increasing performance as more worker nodes participated, but maximum speedup in a 4-PPN, 3 node configuration was limited to 3.5 ($t_{\text{compute}} = 906$ seconds). With partitioning, the same configuration yields a speedup of 7.4, which is significantly closer to the theoretical ideal speedup, which should be close to 12 (see subsection 2.6.2).

2.6.6 I/O Dependence

Because our workloads are far more data-intensive than compute-intensive, overall performance should be highly influenced by I/O performance, and this expectation is well supported by our tests. Intuitively, highly data-intensive computation (with more than one flop per input-byte) is usually bottlenecked by bandwidth to the disk (or other subsystems), and benefits little from increased processor core counts.

In [53], we demonstrated the benefits of redirecting file I/O to an in-memory filesystem for single-machine installations, showing mild speedup (1.2 for resample) even without parallelization, and more significant speedup (1.5 for resample) with four
parallel processes, relative to using only an on-disk filesystem. We observed similar results in testing the current system with and without using the in-memory filesystem. High data-intensive workloads thus incur a drastic penalty as processor parallelism increases without corresponding increases in parallelism of the I/O subsystems. For *resample*, redirecting file I/O to an in-memory filesystem proved effective, because of the majority of file I/O occurred on intermediate (temporary) files rather than input files.

The *ipcc* benchmark illustrates I/O dependence more clearly. Consider figures 2.9b and 2.11 which plot the same speedup data for *ipcc* showing speedup versus node count and PPN, respectively. For *ipcc*, performance is almost completely dependent on the machine count rather than the processor-count. Note that the Input Rate, or the average read bandwidth to the input, ranges between 48MB/s and 65MB/s, which is close to the disk manufacturer’s specified maximum sustained transfer rate of 65MB/s[^46]. Indeed, additional process-parallelism without changing disk count reduces performance slightly, which is consistent with the expectation that additional disk contention lowers overall disk read bandwidth. In this case, disk access patterns were voluminous, but sequential, and OS buffering seems to have limited the penalty from contention.

[^46]: disk manufacturer’s specification.
2.6.7 Overhead and Limitations

The parsing stage incurs significant overhead in the resampling benchmark, due to shell and environment variable handling in pure Python code. The excessive script length of 14,000 commands in 22,000 lines translates to roughly 5 minutes of overhead, pointing to a need for optimization in the parsing and command generation code, should such lengthy scripts become commonplace. The parsing overhead can be also reduced to insignificance for long scripts by implementing an early start mechanism that dispatches commands as soon as they are discovered as ready, before parsing completes. We expect that with tasks that are mostly or completely I/O bound, an early start mechanism would have the same effect on performance (or, in some cases, greater) as a fast parser.

2.7 Conclusion

We have addressed the problem of scaling data-heavy geoscience analysis by introducing a system that leverages familiar shell-script interfaces and NCO tools to specify safe computation on remote data. Although these analysis workloads have required low amounts of computational power and routinely perform less than one floating point operation per byte of input, their large data sizes ($\approx 30$-$100$GB) has made them cumbersome to execute using other methods. Our implementation has made such computation efficient, by compiling scripts into implicit workflows, partitioning the workflows to reduce overhead and minimize data transfer, and scheduling execution according to data locality. Future work is aimed at implementing more advanced syntax support in the parser, and coarser work delegation for better performance where input data resides at multiple locations. SWAMP is scheduled to be tested at the National Center for Atmospheric Research (NCAR) as a method to provide
server-side analysis capability for their community data portal.
Chapter 3

Shell Compilation

3.1 Introduction

Since the development of “RUNCOM”, a shell for the Compatible Time-Sharing System (CTSS)[45] in the mid 1960s, shell-scripting has enjoyed continued popularity among users, programmers, administrators, and all those who interact with UNIX or BSD systems. Shells, i.e. command interpreters, are the traditional way for humans to interact with these systems, and shell scripting languages extend the basic task of selecting and running programs with higher level programming language constructs. However, their focus on efficiency in interactively running and connecting programs makes them unmatched for short tasks like system initialization, administrative tasks, and compositions of simple tools, and cumbersome for larger tasks, where its compactness and specialization limits readability and generality.

The commands executed by most shell scripts are simple and short-lived, but in the scientific computing community, shell scripts form the backbone of significant data reduction and analysis tasks that can take hours or days to run serially. While parallel
execution of many of the commands is feasible, it is unreasonable to expect scientists from non-computing disciplines to define dependences in a safe and effective manner.

Given their widespread use in the scientific community, we explore the performance potential of shell scripts, beginning with a description of shell language. We then propose an alternative execution model, where scripts are compiled into workflows and executed in parallel. We describe our prototype implementation, illustrating the performance improvements. Our experiences are generalized as we then discuss the important issues to be considered in any implementation of shell compilation. Finally, we describe other work related to shells compilation and parallel program execution and summarize our case for script compilation.

3.2 Shell Language

3.2.1 Definition

Shell command languages exist as a means to direct the loading and execution of programs within a hosting operating system. For the purposes of this paper, we define shell language to mean the language accepted by a command language interpreter, the `sh` utility, as specified by POSIX [29]. Most shells implement the POSIX feature set, and share the same core ideas and purpose, although their syntaxes may differ.

3.2.2 Purpose

The purpose of a shell language is to provide a convenient and flexible means of expressing what program to execute and how it should be executed. Shells date back to the mid-1960s, when Louis Pouzin coined the term to describe a command
language for driving the execution of command scripts (sequences), so that multiple
commands could be executed on a computer in the the absence of an operator[42].
While graphical user interfaces now provide alternate means for humans to specify
command execution, shell language remains the most popular choice to automate the
execution of command sequences. For example, every version of Microsoft Windows
has provided shell-scripting via a batch file facility, a feature included since the earliest
versions of the Windows predecessor DOS. Shell scripts are “programs of programs,”
and it is this usage that motivates our study.

While it is possible to invoke programs in other languages, only shell language is
specialized as such. Program execution is possible in C, for example, but using C
instead of shell language would be unquestionably painful, even in interpreted form
(e.g. Cint[15] and CINT[8]). [47] compares the “hello, world” program in sh versus
C, finding their program sizes to be 29 bytes and 16 kilobytes, respectively.

3.2.3 Features

Shell language has a simple syntax, designed to minimize syntactical clutter in the
most common cases. Execution, for example, is specified by simply naming the desired
program or command followed by its arguments in a command-line. Identically-named
programs are distinguished by their locations on the filesystem and are indicated by
prepending this location (the path name) to the program name. General-purpose
programming features developed out of early macro and variable capabilities, and
more interactive features like command and filename completion developed for user
convenience.

The shell features conditional branching through case-esac and if-elif-else-fi con-
structs, which are evaluated sequentially. Looping is expressed using for-do-done
constructs, which executes a command sequence for each item in a list, and \textit{while-do-done} or \textit{until-do-done}, which repeat sequences as long as a particular expression evaluates as true or false, respectively. These control flow structures let programmers specify flexible and iterative behavior analogous to control flow structures in other imperative languages.

The shell recognizes a single primitive type, string, for variables, which may exist in two namespaces. \textit{Shell} variables are defined and exist within a particular running shell context. They exist for command and parameter substitution, as well as for control flow structures, but are not otherwise visible to other programs unless they are \textit{exported} to the environment. \textit{Environment} variables are managed by the operating system for the shell instance. The environment variable namespace context is inherited by programs launched by the shell, but environment variables are otherwise used similarly as shell variables. Both shell and environment variables are syntactically referenced identically, with precedence given to shell variables.

Functions or subroutines can be defined in the shell, but do not occupy separate contexts, and do not have local variables except for their positional arguments. Variable references are always dynamically-scoped—each line is interpreted and executed one at a time.

The shell provides an interface for I/O-redirection which facilitates directing a program’s standard input or output to or from a file or another program.

Job control is achieved by a syntax for background launching of programs, coupled with built-in shell commands to list jobs and to switch them between foreground and background. These facilities allow explicit specification of parallel programs (or program sequences), albeit without built-in support for synchronization.
3.2.4 Execution Model

Shell language exports an immediate, serial, and local execution model. Commands, whether programs or shell built-ins, are expected to be executed immediately, one after another, without any sort of batch queuing delay, on the same machine as the running shell instance. Commands are executed in the context of the running shell (including shell variables) as well as the “environment,” which includes operating system-managed per-process context such as user privileges, environment variables, and filesystem context. Each running shell instance maintains separate context, and children shell instances, e.g. those used for executing shell scripts, inherit a copies of their parent contexts. While programs can be run in parallel, the shell commands which launch and toggle jobs between foreground and background are executed immediately and serially.

3.2.5 Limitations

While shell language has many features in common with more general-purpose languages, its limitations make it unsuitable for programming long, complex programs and applications. Only scalar variables are supported, and while similar type-less approaches exist in other languages, shell arithmetic is integer-only and limited. Objects, records, containers, or array data structures are not supported (arrays are supported in other popular shells). Variables exist in a single global scope, making recursion difficult at best. Unsurprisingly, shells also lack synchronization primitives for concurrent programming.

Additional evidence of shell language’s limitations is seen in the popular use of shell “one-liners” that rely on tool languages like AWK or sed, or more full-featured languages like Perl. These languages, freed need for interactivity and a terse program
launching syntax, let a shell programmer perform the text processing often needed to match one program’s output to another’s input. While such common use of external languages reflects shell language’s strength as a glue language, it also reflects an expectation that computation will be done through external programs rather than within the shell.

3.3 Shell Compilation

3.3.1 Applicability

Our initial motivation was based on the widespread use of shell scripting to automate data processing and analysis. No other language provides as easy or as familiar a mechanism to reuse existing scientific tools. These scientific scripts form a large class which share three characteristics. First, these scripts are composed of programs which are well-behaved, i.e. their behavior is highly predictable, or their unpredictable characteristics are ignorable. This means that behavioral models can be written which allow a compiler to reason about program behavior, such as file access patterns. Second, these scripts are written to process data, so their intrinsic operation consists of data flowing through a chain (or web) of steps. Dataflow systems possess natural concurrency, so these scripts, given that they specify dataflow behavior, contain significant potential for parallelism. Third, they make use of a limited set of programs—common tools for data processing usually exist in each scientific community. Implementing program behavior specifications for a compiler to reason about such scripts is therefore tractable—not all possible programs need be considered.

For concreteness, consider the following shell script, which is an example of a scientific processing script.
Given the availability of data stored in files according to a “YYYY-MM-DD.nc” naming convention, this script creates a yearly time-series of years 1999 and 2000, along with the average difference between the two years, using the netCDF Operators (NCO), a suite for data manipulation and analysis common in geosciences[59]. Its side-effects are insignificant or ignorable, it clearly represents a sequence of processing steps on data, and it easily accomplishes its job using a limited set of programs. We illustrate its dataflow graphically in figure 3.1.

While these program set limitations may seem to constrain shell compilation to irrelevance, for many application domains, these limitations are small and easily handled. The benefits we describe below, especially in performance, cannot be easily obtained otherwise. Shell compilation leverages the largest advantage of shells, namely its ease in running and connecting together programs regardless of their implementation languages.
3.3.2 Definition

We define shell *compilation* as the transformation from shell script representation to a dataflow program, or workflow specification. Traditional script execution, in contrast, involves no intermediate analysis of the script, consisting purely of line-by-line parsing and execution. We illustrate a possible script compilation sequence in figure 3.2. The process begins with lexical and syntax analyses like most compilers, but the remaining steps are different. At the end, where another compiler would produce a program that can be loaded and executed, the resultant workflow can then be optionally optimized and executed on a workflow engine.

Unlike other compilers, the Syntax Parser builds a parse-tree representation reflecting shell language control flow constructs. The Partial Script Evaluator uses this representation to evaluate the shell language portion of the script. In this stage, the compiler executes variable assignments, evaluates control flow constructs such as loops and conditional statements, and performs command and parameter substitution. Partial Script Evaluation is not intrinsic to shell compilation, but we believe its presence provides significant performance benefits (see section 3.5.2).

The Dependency Analyzer applies programmed knowledge of program behavior to determine inputs and outputs of each program invocation and to build a graph rep-
resenting all data dependencies between the program commands in the script. The resulting graph is free of loops and conditionals and may be used for execution on a workflow engine. At this stage, the compiler may verify that the script makes no references to invalid programs and no accesses to unpermitted or invalid files. Because this step relies on models of program behavior that cannot be generated automatically, it relies on the manual effort to support each program, preventing shell compilation from being possible for all scripts.

This definition of shell compilation leverages a shell language property shared only with other wrapper languages or glue languages, that they function only to provide a high-level organization for the more interesting behavior implemented externally. Because script interpretation speed is rarely a bottleneck, optimization of only shell constructs is of limited utility. Performance is rarely affected by slow variable assignment or shell arithmetic. The scope of compilation must extend, therefore, beyond the semantics of the shell language to include program behavior. Compilation leverages models of individual program behavior to model and optimize behavior of entire scripts.

3.3.3 Semantic differences

In order to achieve performance benefits, some differences in execution semantics are necessary. Normal script execution with a standard shell interpreter implies inorder, sequential execution of each command line. Within the limits of this model, compilation could at best, convert a script into an executable consisting of the proper sequence of exec system calls. Again, no performance improvement would be seen unless interpretation were a bottleneck. We propose loosening execution correctness to require merely the correct final state rather without defining how execution should
progress.

Having a correct final state means execution should terminate with a state which is indistinguishable or equivalent from the perspective of the user. For simplicity, we restrict the output state to consider only filesystem contents—correct contents written to output files. This excludes file metadata such as timestamps or file access order, because those do not affect data correctness in most applications. Intermediate files, i.e. files written during execution but are not part of the user’s output, are also not considered. While intermediate files may be deterministic, their value to the user is similar to the value of files created by the the standard C `tmpfile` function, or files residing in “/tmp” or “C:\TEMP”—useful for forensic analysis, but generally disposable. Correct output files have bit-for-bit identical contents, excepting parts which are non-deterministic relative to input files (such as timestamps or process identifiers). With intermediate state now unspecified, transient disk or memory footprint or disk access patterns are no longer under direct programmer control—responsibility lies with the compiler whose resource management can be automatic if not better.

With this looser definition of correctness, programs may be launched out-of-order and in parallel. Program execution may even be migrated to other physical machines, or avoided entirely, as long as correct results are obtained, perhaps through caching. To account for this flexibility, an execution model should not guarantee system state, but rather leave it undefined, between execution start and completion. Interruption of execution under this looser definition may leave the system in an undefined state, although practical implementations should allow the filesystem to be rolled back to equivalent initial conditions. The freedom gained by loosening correctness is similar to that gained by not requiring precise interrupts in a particular microprocessor design.
3.3.4 Advantages

Altering the definition of correctness to be based on results rather than implementation, and using information related to program semantics confer a few key advantages.

Parallel execution is the most obvious benefit. Shell scripts may be parallelized by simply instructing the interpreter to start all programs as background processes (for example, by appending “&” to all program invocations), but such a method is unlikely to produce correct results without utilizing knowledge of program semantics and inserting additional programs or sophisticated shell constructs to function as barriers or other forms of synchronization. With knowledge of program semantics, parallelism can be exploited while still achieving correctness. This allows shell-scripted tasks to leverage parallel hardware without being ported to shared-memory or message-passing libraries or being rewritten in parallel languages.

With knowledge of program semantics, a compilation system can detect the inputs and outputs of programs from their command-lines. Knowing each command-line’s inputs and outputs, it can construct dependence relations between commands, and thus detect where commands can be executed in parallel while maintaining correct results. Coupled with the loose correctness guarantee, program semantic knowledge enables the system to alter filenames in program arguments and to effectively abstract filesystem access. The resulting layer of indirection provides locality and interference optimization, isolation, and portability. Optimizing locality and interference in script execution has significant benefits to overall execution performance, especially in I/O bound scripts. Some scripts show little benefit from parallelization without such optimization. Scripts, especially those whose sequential execution alone saturates disk bandwidth, can benefit from a system which can rewrite commands so that they access different disks, thus preventing disk contention from degrading performance.
Freedom to redirect file access also allows the system to convert disk accesses to in-memory filesystem accesses. Given sufficient memory capacity, intermediate files need never be written to disk (see figure 3.3). In contrast, operating system caching, while effective, is unlikely to eliminate disk access to the same degree due to metadata consistency requirements. Files which are detected to reside remotely, e.g. via NFS, can be prefetched into an in-memory filesystem while other parts the script are run. Similarly, files to be written remotely may be written first to memory, deferring remote transmission, and decoupling program performance from network performance problems.

Abstracting file access allows a system to isolate script execution to prevent scripts from accessing inappropriate files. Because the compilation system understands program behavior, it can flexibly allow accesses by some programs but not by others, giving each program different privileges, perhaps even depending on context (such as whether its inputs are tainted). This differs from the type of isolation provided by the chroot system call which provides a sanitized view of the filesystem. Executing scripts may also be prevented from interfering with each other, even if they reference identical filenames, provided that their output files are destined for different locations, e.g. in environments where servers execute scripts on behalf of users. Script programmers are also freed from manually choosing or stochastically generating non-conflicting names for temporary files.
Portability is also enhanced by abstracting file access. A compilation system can
detect when scripts reference platform- or system-specific pathnames and substitute
equivalent pathnames where possible. This facilitates automatically distributing ex-
ecution among multiple machines, provided the machines can be given, or already
have access to the files needed.

Finally, because correctness only depends on final state, a compilation system can
remap, substitute, add, or eliminate commands in execution. Program names can be
remapped for portability reasons. Unsafe program references may be replaced with
references to safe or hardened equivalents. A single command-line can be replaced
with multiple command-lines. Suppose a script specifies a command which aggre-
gates and processes data from many files and whose execution exceeds the available
memory resources of the running system. If the compiler knows of an equivalent com-
mand sequence which has lower memory requirements, e.g. by processing the files
in smaller batches and aggregating the results, it may choose to substitute the new
sequence. Conversely, the compiler could detect where multiple command-lines could
be combined into a single command. Each time the compiler finds a match in its ta-
ble of equivalence productions, it can choose to substitute based on some measure of
goodness (perhaps using a utility function). Caching can also be used to reuse results
from previous executions, or to eliminate redundant command-lines within scripts.

To better understand the potentials of these command transformations, consider this
following example.

\texttt{wc /export/server1/home/user/paper.tex}

Assuming that /export/server1 were a remote NFS mount, this command performs
a word-count of /home/user/paper.tex using \texttt{wc}. Consider this alternate command
which appears functionally equivalent to the user (assuming the appropriate permis-
rsr server1 "wc /home/user/paper.tex"

We can express this equivalence as:

\[ wc /export/<machine>/<path> \leftrightarrow rsh <machine> "wc /<path>" \]

although a compiler should note other information about \texttt{wc} and \texttt{rsh}, such as how to compute the input and output filenames and model its behavior. The goodness of the transformation can be evaluated using a corresponding utility function either statically at compile-time, or dynamically at run-time, since some factors which affect the goodness, e.g. available network bandwidth, \texttt{rsh} transaction latency, may vary significantly over time.

### 3.3.5 Challenges

There are several difficulties in implementing shell compilation as described so far. The first difficulty relates to the implementation of shell language syntax itself. Shell languages are typically documented assuming an implicit knowledge of the shell interpreter basics, and are sometimes described by comparison to other shell languages. Documentation usually focuses on usage and features rather than specification, although the POSIX specifications for command shells\cite{POSIX} is an important exception. Achieving full functionality in a new implementation is a significant task, even without the complex features mentioned above.

Another difficulty is the difficulty of expressing and utilizing knowledge of program behavior. Script compilation is most easily understood in the context of constituent
programs which can be modeled as performing transformations of a given number of input files and writing outputs to a given set of output files. Clearly not all programs can be easily characterized this way. Examples of programs which cause difficulty are interactive programs such as terminal programs (e.g. `telnet`) or text editors (e.g. `vi`), or shell programs themselves. In these cases, program behavior file access is difficult to predict by analyzing the command-line alone, or the desired “result” behavior of the program is not adequately captured by a model of transforming inputs and writing outputs. Automatically determining which programs follow this model is similar to the halting problem and is therefore intractable. To support a program therefore requires hand-coded specifications or other manual assistance. Just as compilers in other programming languages consider function calls opaque, a shell compiler must otherwise consider program invocations to have unknown side effects that prevent re-ordering.

Fault or exception handling in out-of-order shell execution can be problematic. To parallelize, the system must dispatch a given command-line before its script-order predecessor command-lines have completed. For example, consider a script whose first command line hangs the script in an infinite loop. Parallel execution would speculate ahead, according to its model of expected behavior, and thus produce results distinguishable from the non-parallel, or serial interpreted version. In out-of-order microprocessors, similar issues from faulting instructions are handled by delaying register or memory writes by speculated instructions until their predecessors have completed (committed) normally. Upon failure, speculated instructions are squashed and their results are not persisted in register or memory state. Parallel script execution can perform similarly, by virtualizing file writes of speculated program calls, and discarding results upon parent failures. Program side effects (behavior not predicted or modeled by the shell compiler), however, would still persist, and must therefore be assumed to be negligible or effectively nonexistent.
Finally, compilation is less effective with scripts including programs that require user interaction. Because their behaviors are not predictable, their presence in scripts may make the script unsuitable for compilation, or at best require a significantly more complex specification of behavior, and be scheduled with constraints that may prevent execution parallelization. In multicomputer execution, it is unclear whether the scheduler should dispatch interactive programs locally, or arrange to forward user interaction between the invoking workstation and a remote node. User interaction during script execution presents a significant complication to parallel or distributed execution (and to batch execution, similarly), and should not be supported in those cases.

3.3.6 Execution Model

A compiled execution model implies transformation of original source code into another format which is more directly executable. Because many benefits of shell compilation depend on an execution engine which leverages the additional semantic knowledge, we describe both compilation and execution.

Compilation Phase

Instead of line-by-line interpretation, compilation begins by scanning for shell language constructs and generating a parse tree. After this first pass, a second pass evaluates conditional branches and unrolls loops and other built-in constructs (those typically evaluated within the shell) in order to present a simpler, cycle-free, hazard-free workflow to the execution engine. Once the script is flattened into a simpler sequence, its command-lines are matched against the set of allowed programs and their arguments are parsed to determine input and output filenames. Inputs are
matched against a hash table which stores output file mappings of previous lines, and against the catalog of available source files. Outputs are inserted into the hash table, creating new mappings if entries already exist. These mappings replace the original script-specified filenames. The hash table facilitates wildcard globbing to match files that should exist at that point in execution. We discuss file abstraction further in section 3.5.3. At this point, the compiler can check input and output to enforce file access privileges. Once each command is matched, remapped, and validated, the compiler builds a corresponding object and links it with parent commands according to file inputs. The resulting set of commands form a workflow.

Execution Phase

Once the workflow graph is complete, the system can begin executing commands. In a simple implementation, the workflow may be executed by simply invoking the commands in the order that they were parsed and expanded during compilation. Parallel execution, in contrast, requires more careful bookkeeping of commands. A basic method is to maintain four classifications of commands—unready, ready, running, and complete. Most commands enter the execution phase as unready. Once their dependencies are satisfied, they become ready. The roots of the workflow, i.e. commands with no parents, begin ready. Ready commands can be invoked whenever there are free resources, and, once invoked, become running. The running classification exists to ensure commands are not called twice (alternatively, a more sophisticated scheduler could allow replicated executions for increased data locality in multicomputer environments). Finally, once a running command successfully finishes, it becomes complete. At this point, its children (dependent commands) become ready if they have no other non-complete parents. Once all commands are complete, the script itself is considered complete. The resulting execution is topologically ordered, and
approximates a dataflow computation. Output files can be published to the owning user in one batch, or individually as they are written.

The prototype described in the following section combines both compilation execution phases, presenting an interface that mimics interpreted execution.

### 3.4 Prototype

We have implemented a prototype compilation and execution system for shell scripts in the domain of geoscience data analysis. It directly addresses the need for larger scale analysis tools by scaling existing tools and parallelizing their execution. Data movement challenges are addressed by filename virtualization and portability, which allows scripts to be safely and efficiently executed at remote data centers, all but eliminating the need to download input data. Its use of shell scripts to specify custom analyses allows use by individual scientists with little or no retraining.

#### 3.4.1 Application domain

The geosciences community faces a growing imbalance between the rate of data production and the rate of data analysis and utilization. Although data are expected to be viewed, processed and analyzed many times more frequently than they are produced, the lack of scalability in these tools places severe limits on their usage. Scientists often limit resolution in simulations and observation because of data manipulation and processing challenges, rather than supercomputing limits (in simulations) or hardware technology (in observations). Because they typically apply data analysis using shell-scripted sequences of these tools, they can directly benefit from the performance advantages of script compilation.
This domain represents an ideal opportunity to demonstrate benefits of script compilation for the following reasons. First, only a limited number of programs need support. NCO [59] is a suite of orthogonal processing primitives that are frequently used in scripted sequences that perform a variety of both simple and complex operations on (usually geoscience) data stored in the netCDF format[43]. Because of the challenges (see section 3.3.5) of supporting large numbers of programs, restricting support to NCO simplifies implementation.

Second, geoscience data analysis makes use of programs whose behavior is generally well-behaved, well-understood, and well-documented—programs typically read in data from files and output derived data or statistics. At the same time, NCO operators employ advanced file selection syntax, which requires a compilation system to recognize argument semantics when virtualizing file access.

Third, analysis scripts are usually non-interactive and represent implicit workflows whose dataflow and parallel execution potential can be exposed automatically using compilation. While constituent programs may be parallelized via MPI or OpenMP, script compilation can expose higher-level parallelism that does not require reimplementation.

3.4.2 Design goals

The prototype is designed to meet the data analysis and reduction needs of the geoscience community while illustrating the potential benefits offered by shell compilation. It supplements existing data service software by providing computational analysis service. As a public-facing service, it must isolate tasks for security. It must support enough shell syntax to be useful; however, implementing the full standard is unnecessary since most analysis scripts use only a small subset of available syntax.
End user benefit is the goal, rather than elegance or complete syntax coverage. Scientists should be able to *interchangeably* run shell scripts via either compiled execution or plain POSIX interpretation.

### 3.4.3 Features and Limitations

The implementation limits support for shell constructs to those most often encountered in geoscience scripts: for-loops, if-branching, and shell variables. While the syntax for handling environment variables is supported, the system does not maintain a distinction from shell variables and does not apply them in the environment during execution—they are not necessary for the supported program set. Shell scripts that execute using the Bourne shell are able to run unmodified in our system, but can detect the system (and handle portability issues) by testing a predefined shell variable. The system is intended for remote access, allowing clients to submit scripts for execution and to download results afterwards. By user request, we have applied a heuristic that defines *results* files as those not referenced as inputs after being written (i.e. they are *leaf* files). All other files written by script execution are marked as intermediate and can be optimized differently. In the event that non-leaf files are desired as results, they can be marked in the script via special command lines (i.e. dummy pass-through commands)—a more elegant scheme is currently unnecessary.

### 3.4.4 Experiences

Experiences with the prototype from a systems perspective are detailed in [55]; we present here the highlights that relate to shell compilation issues.
Figure 3.4: Ratio of computation time (gray) to input fetch time (black) for two scripts

![Figure 3.4: Ratio of computation time to input fetch time](image)

Figure 3.5: Speedup vs. processor count

**Performance**

The original intent of the prototype system was to scale shell script performance by eliminating unnecessary data download and by exploiting latent parallelism in geoscience data reduction scripts. Our tests have shown that the transfer costs alone account for a significant fraction (≈50% in one case, ≈95% in another, assuming 30Mbit/s bandwidth) of overall end-to-end time, and are practically eliminated by sending the computation to the data source (see figure 3.4). Script compilation provides the sandboxing and portability that allows a data server to accept and safely execute existing desktop shell scripts. Script compilation was also shown effective at exposing parallelism without re-implementing existing analysis tools or porting analysis shell scripts to explicitly parallel formats such as workflow languages [40]. While a script’s potential parallelism width varies over the course of its execution (sometimes numbering in the hundreds), the average width was between 10 and 19.
Figure 3.6: Speedup vs. node count

Figure 3.5 illustrates the overall performance achieved on a single SMP machine by compiler-exposed parallelism, relative to ideal speedup (zero overhead, zero non-parallelizable components). The resample script was easily parallelized, giving a speedup close to ideal, but suffered from bookkeeping overhead in the 4-way case where the command completion rate was about one every 70ms. The ipcc script did not benefit from SMP parallelism, since its sequential execution because of its data intensity which saturates the disk manufacturer’s maximum transfer rate at 65MB/s. The disk bottleneck is verified in multicomputer execution, which is plotted in figure 3.6. Overall, speedup was substantial, but limited by increased overhead due to the relatively large latency of network versus shared memory for synchronization. However, the ipcc script was able to leverage the increased disk bandwidth available from replicating files across all machines. A more comprehensive performance analysis is available in [55].

Although disk-intensive tasks often bottleneck with increasing CPU parallelism, the system can reduce contention by transparently remapping files to an in-memory filesystem. This explicit caching behavior provided performance benefits even without parallelism (14%), but enabled a computational speedup of 3.6 compared to 2.4
when commands were allocated to all four cores of a 4-way SMP workstation.

Finally, we should note that compilation time was excessive on one benchmark, whose lengthy script of 14,640 unique command-lines exposed the inefficiencies of our parser implementation. However, we expect that most scripts can be compiled in far less time. Another benchmark of similar data intensity was compiled in roughly 0.5 seconds, owing to its simpler and more compact script about two pages long.

Usability

One of shell compilation’s biggest advantages is its ability to bring parallel processing to non-computer scientists. Because shells are nearly universal in scientific workstations, no specialized knowledge is needed. Many users already maintain their own data analysis scripts, making porting to a shell compiling system simple. Usually, only minor modifications to path specifications are necessary. The loosened correctness guarantee and unspecified execution order have little importance in data analysis scripts, since it is more important to have correct, useful results quickly. System administrators appreciate the restricted program set in open-access systems, following the principle of least-privilege.

3.5 Discussion

3.5.1 Syntax subset

One implementation difficulty was the choice of exactly what subset of shell syntax to support. The initial implementation supported neither variables nor control flow constructs such as branching or looping, though both were demanded and added as the
potential and realized speedups became evident. Some other features that are worth supporting include variables, wildcard expansion, and I/O redirection. Variables are required in order to support looping, and their presence is essential to writing portable scripts. Wildcard expansion significantly simplifies working with multiple files (see section 3.5.3). I/O redirection allows programs that work with standard input and standard output to be parallelized within the more generic file-based model.

Some features of shell programming should not be supported. In a compiled system, where parts of the script may be executing on different machines, job control command semantics are not easily defined or implemented. This is a consequence of the shift in a script’s meaning from a specification of execution order towards a specification of desired output. Compiled scripts should also not support all programs—exclusion of interactive programs and other programs whose behavior cannot be determined a priori from their command-lines is natural and discourages use of compilation for scripts which will not benefit. Although scripts containing these less deterministic programs can be supported by splitting compilation and execution at these occurrences, the unclear benefits do not justify the additional complexity. In open-access environments, restricting the program set is doubly important to reduce the possibilities for unknown security holes.

### 3.5.2 Partial evaluation

One important issue was deciding which script elements should be evaluated during compilation and which should be used to develop the workflow. Consider the following script lines, which copy a file:

```bash
SOURCE=foo.data  # (1) set SOURCE
cp $SOURCE bar.data  # (2) copy the file
```
cp $SOURCE foobar.data  # (3) copy the file again

Dataflow analysis easily detects that the second line depends on the first line, and could produce a workflow graph looking like:

\( (1) \text{SOURCE=...} \rightarrow (2) \text{cp $SOURCE...} \rightarrow (3) \text{cp $SOURCE...} \)

However, we can produce an optimized graph with fewer nodes by evaluating the variable assignment in (1) during compilation and substituting in (2) and (3).

\( (1') \text{cp foo.data...} \rightarrow (2') \text{cp foo.data...} \)

The latter graph has fewer nodes and has no non-parallel components.

Similarly, evaluating control-flow structures eliminates control hazards and enables dependencies to be evaluated over the whole script. The resulting dependency graph lacks unused script portions (dead code elimination), expands iterative portions (loop unrolling and parallelism), and can be scheduled for execution with greater freedom. Specifically, if-then-else and for-do-done constructs are prime candidates for pre-evaluation. These branch and loop evaluations rely on the assumption that their conditional or looping expressions do not depend on outputs of previous lines. In our application domain, this assumption generally holds. Eliminating control hazards and providing the entire execution graph aids a system’s admission control as well. Scripts that execute too many commands, or reference invalid or unauthorized files can be rejected as a whole instead of aborting in the midst of execution.

Partial evaluation processes the “meta” portion of the script, that is, the portion that supports the main purpose of the script. A standard shell interpreter distinguishes
between built-in and external commands, evaluating the former internally and executing programs specified by the latter. Compilation should process these built-ins (or a subset), but some external programs should be processed as well. These include `seq` and `printf` which are often used to generate loop ranges and format filename strings in command lines. In general, referentially transparent portions of the script are good candidates for partial evaluation, but should be restricted to extremely lightweight programs because their partial evaluation would occur during compilation and without optimization. ¹

### 3.5.3 Filename Abstraction

While standard shell interpretation does not provide abstraction from filesystem names, our experience has identified at least two additional useful layers. First, we separate the script file layer from the physical filesystem. This provides the benefits introduced in section 3.3.4: locality and interference optimization, isolation, and portability. This abstraction is achieved by remapping all references to filenames in the script. This includes not only filenames in program arguments, but other arguments from which the program may compute filenames (as in the NCO tools [59]), and wildcards originally expanded against the filesystem.

The second layer of abstraction is inserted between script filenames and physical filenames in order to eliminate write-after-read and write-after-write data hazards (anti, and output dependencies). Read-after-write hazards, or true dependencies are resolved by command ordering. The new layer, consists of logical names and is identical to the script layer, except that logical names must refer to files that are immutable, once written. This converts command-lines into static single assignment

---

¹Overall, partial evaluation serves to evaluate the parts of the program which are referentially transparent, that is, they can be replaced by their values without changing the program.
(SSA)[13] form, and prevents the aliasing that occurs when scripts write to the same filename more than once. The table below summarizes these levels.

<table>
<thead>
<tr>
<th>File Namespace</th>
<th>Multiplicity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script</td>
<td>[1]</td>
<td>user-visible</td>
</tr>
<tr>
<td>Logical</td>
<td>[1,n]</td>
<td>SSA form</td>
</tr>
<tr>
<td>Physical</td>
<td>[0,1]/logical</td>
<td>local copy</td>
</tr>
</tbody>
</table>

*Note that logical files may have multiple physical copies.

### 3.5.4 Files as Variables

To evaluate the language characteristics of shells, it is useful to consider files as the analog of variables in other languages. A particular filesystem is like a memory system, and the ability to mount other filesystems like mapping in other segments in “virtual memory.” Like memory pages and segments, parts of the filesystem can be marked with a combination of read, write, and execute privileges. Symbolic links are like reference variables. Files do not need to be declared or fixed in size when they are created, and offer the same flexibility as variables in loosely-typed languages. Files have global scope, but are organized hierarchically in directories, just as variables in nested data structures. Although shells do not have local variables(files) with scope-limited lifetime as other languages, the concept of a “working directory” simplifies naming similarly. Unlike other languages, however, shells have a separate search path for referencing executable variables(files) whereas other languages unify the namespace of data and code.

Considering files in the same vein as variables, may they be garbage collected? Shells do not implement garbage collection—although the reclaiming of unlinked inodes
is similar, it occurs in repair contexts. In considering shell scripts as programs, garbage collection would mean automatic deletion of unreachable (dead) files, but since files are always reachable unless overwritten or explicitly deleted, it is unclear how to identify which files are effectively dead. Standard garbage collection techniques like mark-and-sweep do not seem effective, since all files are reachable through the filesystem hierarchy. We can infer that files only written-to but not read-from in the script are “live,” and desired as useful output, and the contrapositive—that files written-to and read-from are dead—is true often enough to be a good heuristic, though not always the case. Our prototype uses this heuristic since it holds true for most scripts of our domain.

Considering platform portability in light of this file-based model, shell language experiences similar portability problems as other languages. Just as standard system libraries may vary in location and cause portability problems in C programs, filesystem paths and layout vary and may cause similar problems for shell programmers.

Wildcard expansion is a shell language feature without analog in the realm of variables. For example, `cp * ..` copies all files in the current directory to the parent directory. We are not aware of the equivalent concept, copying all variables in local scope to another (e.g. parent) scope in a non-shell language, although some languages can achieve similar behavior using introspection or a variable representing the current context.

### 3.6 Related Work

Work in shell compilation has been relatively scarce, favoring instead development of new, more programmable shells, such as `rc` for Plan 9 [19], or other languages
that provide shell-like functionality within a more general-purpose syntax \[22\]. \[47\] provides a good description of the weakness of the Bourne shell as a programming language, and proposes the addition of program-running and program-connecting capabilities to a general-purpose programming language. The \texttt{make} system \[31\] can accomplish a similar performance capability in more modern parallel \[4\] or distributed \[35\] implementations, but at the cost of a vastly different syntax that forces user specification of command inter-dependencies.

Grid workflow research has explored similar issues in parallel and distributed execution. GridAnt \[3\] provides similar abstraction in its workflow specification and offers similar flexibility in scheduling and distributing work. GRID superscalar\[5\] and its successor COMP superscalar similarly convert sequential applications to dependency-aware workflows, but require use of a C-like syntax uses function signatures to signify dependence relations. All grid workflow systems, including GridAnt, require explicit user specification of dependencies, which script compilation avoids.

### 3.7 Conclusion

Shell languages are special programming languages which contain many difficulties and lack many features in comparison to other languages. Their specialized purpose of facilitating and automating execution of other programs has led to design choices that make their compilation useless or practically impossible in the general case. However, by restricting the program choice and loosening the correctness model, compilation becomes possible, and the resulting execution flexibility enables significant advantages to security, portability, and most of all, performance. Compilation exploits a characteristic shared by a significant class of shell scripts—that they define data flows in terms of ordered command-line sequences processing files. Our proto-
type, implemented for this class of scripts in the domain of geoscience data reduction, illustrates the significant performance potential as well as the issues that prevent shell compilation from being universally applicable. Results show that shell compilation may be used as an alternative method of leveraging parallel hardware—by parallelizing sequences of applications rather than rewriting the applications themselves. With the increasing importance of parallelism as a means for achieving high performance, shell compilation merits further study since it provides performance benefits to applications without requiring deep study of their internal algorithms.
Chapter 4

I/O-Constrained Efficiency and Parallelism

4.1 Introduction

4.1.1 The Problem of I/O

Although supercomputer and distributed computer performance is most commonly compared with respect to Dongarra’s LINPACK Benchmark, the author cautions [18] that the benchmark “must not be used indiscriminately to judge the overall performance of a computer system.” Given that the benchmark consists of solving dense systems of equations, computers that excel at such a benchmark may not excel at data-intensive workloads.

This chapter examines the performance characteristics of two data-intensive benchmarks, each representing real geoscience workloads. The data-intensity of each is quantified, along with the performance problems that a I/O-insensitive scheduler
would incur. A scheduler implementing basic I/O-sensitivity is described. Two techniques, explicit file caching and work partitioning are described. The partitioning algorithm is described in particular detail—its development was necessary to satisfy certain constraints lacked by existing partitioning algorithms. Workloads are assumed to be representable by directed acyclic graph (DAG) workflows. The DAG workflow model is the most popular model for defining workflow applications and is well-studied as a result.

### 4.1.2 Defining I/O Performance

The performance of a given system is most unambiguously defined by its overall execution time for a given known task. In CPU-bound applications, computational throughput measured in operations per second (for general applications) or floating-point operations per second (for mathematical or scientific applications) is a sufficient proxy for performance. Supercomputers, which are designed for CPU-bound applications, are compared in terms of computational throughput.

The performance of I/O bound applications can be approximated by the performance of the bottlenecking I/O resource. Consider an application which computes the minimum and the maximum of integer values stored in a particular file. Because value comparison is computationally simple, application performance can reasonably be expected to depend on the peak sustained read bandwidth of the source disk.

### 4.2 Performance Characterization

Intuitively, I/O-bound workflows should exhibit different characteristics than their CPU-bound counterparts. For example, one expects an I/O-bound workflow to ben-
efit less from additional flops than from I/O bandwidth. This section summarizes observations of I/O-bound workflows when parallelized in different ways on the SWAMP system.

4.2.1 Test Setup

Each workflow was tested using four CentOS 4.2 Linux-based dual-Opteron 270 systems equipped with 16GB of DDR memory and 500GB SATA disks. Each Opteron 270 processor consists of two complete 64-bit processors, sharing only a memory controller and cache coherence logic. Tests were conducted using the SWAMP system which allows the number of parallel processes and machines to be tuned, enabling performance observation in configurations between 1 and 16 processors. Although the number of physical CPUs available is not directly controlled, varying the number of offered parallel processes is a popular and well-understood means of controlling CPU involvement.

Test results are shown for two benchmarks the resample benchmark and the ipcc benchmark, both of which are described and tested in section 2.6. The shell compilation cost of SWAMP execution is ignored since it is an artifact of the system rather than a characteristic of the workload. Both workloads are I/O intensive: resample processes 8GB of input data, creates 26GB in intermediate files, and writes 230MB of output; ipcc processes 30GB of input data, creates about 500kB in intermediate files, and writes about 500kB of output.
4.2.2 Multicore

For CPU-bound workflows, performance is expected to scale with processor count, especially within a single machine, where network latency is not a concern. Figure 4.1 summarizes overall speedup versus a baseline script execution outside of the SWAMP system. Despite its I/O intensity, resample scales well with increasing process count, saturating when the number of parallel processes equals the number of CPUs as would be expected from CPU-bound workload. This scalability is enabled by the explicit caching optimizations made possible by SWAMP’s shell compilation. Disk parallelism is further discussed in section 4.2.4.

Explicit caching is unable, however, to compensate for the I/O-boundedness of ipcc. To better understand this, compare its runtime characteristics to those of resample (figure 4.2). Comparing their CPU usage (figures 4.2a and 4.2b), ipcc spends nearly all time in the iowait state, compared to resample which spends nearly all time in user code. Comparing their disk activity statistics, ipcc’s disk usage is sustained whereas resample’s is bursty, with long periods of idle time.

From the single-process traces, one can predict the relative un-scalability and scal-
Figure 4.2: *ipcc* and *resample* 1x1 Statistics
Figure 4.3: *ipcc* and *resample* 1x6 Statistics
ability of ipcc and resample, respectively. This is confirmed in the statistics for the configuration using six parallel processes (figure 4.3).

ipcc’s CPU profile appears roughly the same, showing only differences where one of its processes can iowait while another sustains a burst of user time. resample, on the other hand, is able to saturate the CPUs after completing the parse stage (time≈300). Similarly ipcc’s disk activity is similar, maintaining a disk transaction rate of roughly 200Hz and a read bandwidth less than 10MiB/s in both cases, although the additional in-flight processes have smeared the spikes together.

4.2.3 Multicomputer

The same trends seen in the SMP multiprocessing configurations hold when the workflows are parallelized across multiple machines. One difference, however, is that more disks are available, which should allow I/O parallelism for ipcc. This is confirmed in a plot of speedup versus parallel processes in figure 4.4, which illustrates similar conditions as 4.1 except that processes are spread among multiple machines rather than cores on the same machine.
Another speedup plot is shown in figure 4.5, where the number of processes per node (PPN) is set to 4 rather than 1.

The multicomputer CPU and disk traces are largely unsurprising and are shown in figure 4.6. Configured with 4 nodes, and 1 process per node, visual inspection of the trace reveals that each node is loaded similarly as the unparallelized case shown in figure 4.2. This provides ipcc with the disk bandwidth it needs that was not available when work was parallelized over CPU- but not I/O-resources as shown in 4.3.

### 4.2.4 Disk Parallelism

A workflow executing on a single machine primarily has a single bottleneck, its disk subsystem. In order to quantify the I/O bottlenecks, CPU and disk statistics were gathered for ipcc and resample benchmarks in serial and 4-way multicore execution, both with SWAMP’s explicit in-memory file caching disabled. The resulting traces are illustrated in figure 4.7, and can be compared with figure 4.2, which illustrates the same traces but with explicit caching enabled.

The ipcc benchmark is I/O bound in disk bandwidth to its input data (see section
Figure 4.6: *ipcc* and *resample* 4x1 Statistics

(a) *ipcc* CPU usage

(b) *resample* CPU usage

CPU% is cumulative over 4 CPUs

(c) *ipcc* Disk transaction rate

(d) *resample* Disk transaction rate

(e) *ipcc* Disk bandwidth

(f) *resample* Disk bandwidth
Figure 4.7: *ipcc* and *resample* 1x1 w/o Explicit Caching Statistics
4.2.2), and, being little affected by the cache disabling, experiences little difference in execution time. resample, on the other hand, suffers considerably, spending about 25% more time in computation (3850 seconds vs. 3091 seconds). Looking at figures 4.7d and 4.7f, one can see that far greater transaction rates are needed, and that disk bandwidth is now dominated by output writing rather than source reading. Despite these increases, resample’s disk usage pattern remains bursty, meaning that the disk should be able to support additional concurrency from parallel resample execution.

The CPU traces in figure 4.7b corroborate the increased disk utilization, showing a significant iowait fraction—something absent with explicit caching.

To evaluate disk concurrency in uncached multicore execution, ipcc and resample
were rerun with PPN=4. The corresponding traces in figure 4.8 can be compared with the PPN=6 in figure 4.3. Again, ipcc’s execution is little affected by the lack of explicit caching, but resample is significantly affected. Where explicit caching allowed CPU time to be spent mostly in user code (figure 4.3b), the lack of caching forces the CPU to spend more time in iowait than user code. The dramatic increases in transaction rate (figures 4.8d and 4.3d) further support the conclusion that resample at PPN=4 is limited by disk concurrency. Its computational execution time of 2443 seconds is far slower than the 1022 seconds in cached (PPN=4) mode. This dramatic difference underscores the advantages of explicit caching that are difficult to eliminate with operating system disk caching.

4.3 Scheduling

Efficient workflow scheduling involves many factors. This section describes the SWAMP system’s scheduling system, beginning with some highlights from the larger workflow scheduling community and following with a more detailed description of this work’s scheduler.

4.3.1 Related Work

Workflow scheduling, at its most basic level, is the allocation of a set of interdependent tasks on a set of resources. It is distinguished from other forms of scheduling by its scheduling of distinct work units whose correct execution depends on results produced by other work units within the workflow. For example, thread schedulers in operating systems operate without regard for dependencies between threads, leaving interdependencies to be expressed through each thread’s use of other OS structures
or explicit synchronization primitives.

Although workflow scheduling and its relative dataflow scheduling have been studied for many years, it has been revitalized by recent interest in grid scheduling. Grid technologies provide a normalized interface to heterogeneous resources across administrative boundaries, potentially providing access to an extremely large and diverse set of resources to an end user. Grid technologies have, so far, claimed strengths in raw computational power (through computational grids) and raw data capacity (through data grids). Input and output in grid frameworks occurs by \textit{staging}[21] data to and from computation resources as matters of preparation and cleanup. Little attention is paid to raw data movement performance.

One grid project whose system considers data movement cost is the Pegasus project. The Pegasus framework [36, 48, 16] leverages grid technology for complex data-dependent scientific workflows. Scientists use tools to specify workflows as directed acyclic task graphs containing data dependencies. Pegasus implements advanced resource allocation and locality-aware scheduling, but does not integrate with data services or apply automatic dependence extraction.

\subsection{Design}

The SWAMP system is designed to enable users to simply and easily utilize symmetric multiprocessing (SMP), cluster-level, and perhaps wide-area parallelism. With a long-term goal of making gigascale and terascale data analysis matters of seconds or minutes rather than hours or days, its scheduler must remain lightweight with low minimum end-to-end turnaround latencies.

A basic understanding of the tradeoffs involved in scheduler design will provide con-
text for SWAMP scheduler. Its scheduler is dynamic, allocating tasks to resources
during runtime rather instead of a preparation step. The dynamic approach pro-
vides several advantages. Dynamic schedulers can respond and adapt to changing
conditions, for example, making use of new resources available after start, or work-
ing around resources which have failed. This is especially important where execution
costs, communication costs, and overall running times are not predicted. Where costs
are known or estimable, a static scheduler can find efficient schedules similarly as an
omniscient kernel thread-scheduler can find efficient schedules, even though optimal
scheduling is NP-hard. Despite the power of static scheduling, the difference between
good and optimal schedules is small, and likely small compared to the time variations
encountered in workflow execution on clusters.

4.3.3 Operation

The SWAMP scheduler has three primary components which are involved in overall
workflow scheduling. When workflows are submitted to the system, they enter a
FIFO queue, which buffers workflows so that only one workflow is in-flight at once.
Since the system is designed to support data-intensive tasks which are bottlenecked
on precious I/O resources, support of multiple workflows in flight was eschewed to
avoid potential contention and cache pollution.

If there are no running workflows, the system removes a workflow from the head of the
queue, parses and compiles it, and partitions it. The partitioning algorithm, described
in the next section (section 4.4), divides the task graph into subgraphs, using the
coarse-granularity to reduce bookkeeping and synchronization overhead and forcing
most interdependent commands to be scheduled together, increasing data locality.
With partitioning, the workflow is transformed from a DAG of individual commands
to a DAG of clusters (subgraphs).

The roots of the DAG of clusters are then used to initialize a ready-list. The scheduler is now ready to begin dispatching. Machines in a cluster are organized into a master-worker arrangement, where the master (configurably) may execute commands as well. Each machine is classified as busy or non-busy, where busy is defined to mean that the machine has a number of clusters in flight equal to its configured number of parallel processes (processes-per-node, PPN). Thus, when a worker is busy, it is guaranteed to have at least as many ready or running commands as its configured PPN. Clusters may be linear or non-linear, where linearity is defined as having no two mutually-independent tasks.

To begin execution, the master dispatches clusters to non-busy machines in round-robin fashion, until there are no non-busy machines left or the ready-list has been exhausted. Again, the master itself may be included as a worker machine. Each cluster is dispatched as an graph of commands. Two HTTP URLs are bundled with each command, which are used by the worker to report command results, one for success and one for failure. These URLs are used as a form of callback for workers to report conditions, and upon success, URLs for retrieving the command’s output files are bundled as GET data. In providing URLs for output files, workers allow the master to include pointers to source data when dispatching subsequent clusters.

The master retains only loose synchronization with its workers, initiating communication only to dispatch clusters, and expecting only callbacks (through URLs) upon successes of prolific commands or failures of any command. Prolific commands are defined as commands in a cluster whose outputs necessarily impact execution of other clusters or the workflow itself. Figure 4.9 illustrates prolific commands (marked by shading) in the context of clusters in a workflow. It may be unclear as to why the concept of prolific commands is necessary—since commands are dispatched at cluster-
Figure 4.9: Illustration of prolific commands

- prolific

granularity, why are results not reported at the same coarseness? Reporting results of prolific commands allows unready clusters to become ready more quickly. Because these commands have impacts outside their containing cluster, their completion either (a) fulfills a dependency for a waiting unready cluster or (b) provides an output file desired directly by the end user. In both cases, earlier reporting of prolific command state should speed overall execution in return for a small cost.

One small disadvantage of loose synchronization and the concept of prolific commands is that the master loses some bookkeeping and state-tracking accuracy during runtime. Since it is only updated with “important” information, it generally underestimates the number of commands completed, since the completion of un-prolific commands is only learned indirectly through report of their prolific successors.

The master does not specify or control the ordering of execution within clusters. Each worker independently schedules the clusters that have been allocated to it. This design
choice encourages looser coupling, anticipating future implementation of support for work distribution to independent SWAMP-compatible system instances at other data centers.

After the master performs its initial cluster dispatch, it idles, allowing its internal worker component to execute commands while awaiting HTTP callbacks from workers. Each time a callback is received, the master updates its internal command tracking state, checking the command’s cluster-level successors to determine if any have been subsequently made ready. Should a new cluster become ready, it is dispatched to (a) the same machine that performed the callback, provided the machine is not busy, (b) an un-busy machine which hosts at least one of the cluster’s input files, or (c) any un-busy machine, in that order of priority. If there are no un-busy machines, the cluster is put on the ready list, which is checked again as soon as the master receives the last callback for a cluster (recall that machines are measured for busy-ness according to their numbers of clusters in-flight). This scheme for cluster dispatch keeps workers as busy as possible, while trying to keep chains of clusters executing on the same machine for better locality.

Within each worker, a fine-grained command scheduler maintains an ordered list of its allocated clusters (each of which is an mutually-independent DAG), and populates its own ready-list with ready commands from each cluster. This scheduler dispatches commands from its ready-list until the number of running commands equals its configured PPN. Upon each command’s completion, the scheduler attempts to maximize cache hotness by dispatching one of the completing command’s successors, if one has become ready, or the next command on the ready-list if one has not. At this granularity the other locality-dependent decisions focus on mapping command outputs to the in-memory scratch space to reduce disk contention.
4.3.4 Possible Refinements

Although the current scheduling scheme has been shown effective in testing, there are a few areas for refinement. One refinement is concerned with the master’s tracking of worker busy-ness. Currently, worker instances register themselves to a master instance, indicating their configured PPN, which is used to limit the number of clusters in flight. After registration, workers never report busy-ness nor update their PPN. It is conceivable that administrators may wish to limit worker activity and adjust PPN based on time of day or other factors. Generally, worker resources may vary over time, and an update mechanism may be useful so that changing resource availability can be reported without interrupting a running workflow.

Another refinement is in the definition of worker busy-ness. Section 4.2 discussed the importance of I/O considerations, and described an example of a workflow which did not benefit from additional CPU resources. With this in mind, it may be beneficial to track I/O utilization. More concretely, this means watching statistics such as transactions per second or read/write bandwidth to disks. A consistently high disk transaction rate and sustained read/write bandwidth would indicate disk busy-ness. In that case, a worker may gain overall performance by dispatching less commands than its configured PPN, and thus leaving CPU resources idle and reducing command parallelism. Combined with the first refinement, the master could dispatch less clusters to workers, and still keep worker resources fully-utilized.

Currently, the master dispatches as many ready clusters as it can upon start-up. A refinement would be to delay further dispatch of clusters after one round-robin iteration. The master could potentially re-poll workers for their busy-ness before dispatching more clusters, or simply enforce some time quantum of delay after dispatching a large cluster before dispatching another cluster to the same machine.
Finally, coarse dispatch of work to clusters inherently contains risk of imbalancing load among machines. In the current scheme, individual clusters, since they are allowed to be non-linear, may contain sufficient parallelism on their own to saturate a worker’s resources, yet the master tries to dispatch enough clusters to a worker to guarantee busy-ness in the case that all clusters are linear. To address this, the master could steal unexecuted clusters from busy workers and reassign them to workers that are completely idle. This has no negative impact on locality, except in cases where mutually independent clusters share common successor clusters. In those cases, locality may be optimized by dispatching those to the same machine. Still, the benefits of that are unclear, since dispatch to different machines could provide each cluster with more in-memory scratch space, more disk bandwidth, and more parallel processing capability.

4.4 Partitioning

Because of the noticeable synchronization overhead and inefficient file dispersion observed in dynamically-scheduled command-at-a-time multicomputer workflow execution, workflows should be partitioned and dispatched in coarse clusters to both reduce synchronization overhead and increase file locality.

4.4.1 Concepts

Basic workflows can be modeled as directed acyclic graphs (DAGs), where each vertex corresponds to the execution of a command, incoming edges connect to parent (ancestor) commands, and outgoing edges connect to child (successor) commands. To illustrate, consider the ipcc benchmark discussed earlier in Chapter 2. An abstracted
view of this workflow is shown in figure 4.10. A simplified version shown in figure 4.11 will be used for reference in this chapter.

An important concept in workflows and other DAGS is cost. In workflows, the cost of a particular component subtask is often known, as may be the transfer or other overhead cost for preparing the result of one command to be used as input for another command. Many algorithms for partitioning, that is, dividing workflows into two or more components, utilize these costs to generate optimized partitions. These costs, the execution and communication costs, are often available for workflows that represent operational work, work that is re-executed regularly and frequently with different parameters or inputs. In these cases, although each execution generates
new results, the meta-characteristics of the workflow which relate to execution or communication cost are approximately or effectively constant.

A workflow’s cost information is generally difficult to obtain before its execution. Even estimating workflow costs using input metadata requires at least a pseudo-execution or dry-run, and although it is possible under certain conditions, estimation is equivalent to the halting problem and is thus generally undecidable.

4.4.2 Related Work

Considerable research has been done on graph partitioning. The computer-aided design (CAD) community makes heavy use of partitioning algorithms in order to assist or automate the placement of circuit elements in integrated circuits, or physically discrete components on circuit boards. Min-cut partitions, or partitions which minimize the division of networks between, are used in layout to minimize the number and electrical load of wires between different areas on chips. The Kernighan-Lin-
Fiduccia-Mattheyes (KLFM) algorithm, an iterative linear-time min-cut heuristic for bi-partitioning a flow network, is a popular example in the CAD community, noted for its good results despite only linear scaling of execution time in the number of cells and networks. Figure 4.12 illustrates the results of applying this algorithm on the workflow from figure 4.11.

KLFM is a partitioning algorithm that finds the minimum cut, parameterized with a chosen tolerance for imbalance. This result was generated using a tolerance of 0.2 so that the largest partition was contained at most a $0.5+0.2=0.7$ fraction of nodes. Since KLFM is a bipartitioning algorithm, only two partitions result, making recursive application necessary if more partitions were desired. Alternatively, $k$-partitioning algorithms can be used to divide graphs into a chosen number of partitions, with complexity sometimes greater and sometimes lesser than recursive KLFM application[34].

Although KLFM and its variants are mature and well-understood, they present one
potential disadvantage when applied to partition work among machines in a cluster. They partition a graph into a predetermined number of partitions, regardless of the graph’s topology. Choosing the number of partitions may be difficult—too few could limit parallelization, and too many may cause inefficient results. This means that graphs whose partitioning would present no benefits in execution would be still be partitioned. A simple graph with a shape as shown in figure 4.13 would not benefit from partitioning, so the algorithms’ forced partitioning would merely add additional synchronization overhead. Should the workflow be implemented using distinct hardware elements for each task, these algorithms would be more appropriate.

Partitioning is also studied in the parallel processing community, where algorithms are needed to divide DAGs of tasks among processors. There are many such heuristic algorithms, intended for statically scheduling, differing in time complexity, use of nonlinear clusters, or allowance of task duplication. One such algorithm is the dominant sequence clustering algorithm (DSC) [58]. Its results on this chapter’s reference workflow are shown in figure 4.14.

The DSC algorithm partitions a DAG for an unbounded number of processors, with the expectation that they may be merged if the number of physical processors is less than the number of resultant clusters. Partitioning for an unbounded number of processors is more suitable since the graph will not be forcibly split for a predetermined number of processors. Instead, the graph will be split into a number of partitions according to the graph’s own topology. The graph in figure 4.13, for example, would not be split.
For the example workflow, the DSC algorithm results in four clusters. Inspection determines this partition to be reasonable. Execution on an appropriate four-processor system would be predicted efficient. One disadvantage is that cluster 2 contains multiple dependencies which force it to delay execution until all dependencies or satisfied, or, alternatively, its execution can be suspended between completion of its first task and the readiness of the remaining two external dependencies are satisfied. For the scheduler described in section 4.3, clusters can only be dispatched completely ready to execute, and thus DSC’s partition is less performant. This minor disadvantage is potentially present in other statically-scheduled approaches—processors usually assumed fully-connected, and synchronization between processors is assumed possible.

A minor issue with algorithms such as DSC is their assumption of uniform communication cost among processors. A cluster of multi-core machines has parallelism among multiple computer and among the processors within each computer, and in such a configuration, communication cost among processors is not uniform. The linear cluster restriction of DSC and other heuristics may produce clusters which are
less efficient at exploiting cheap *intra*-computer communication while eschewing more expensive *inter*-computer communication.

### 4.4.3 InSeP Algorithm

The Independent Set Partitioning (InSeP) algorithm used by the SWAMP system is briefly introduced in section 2.5.2. Its heuristic operates in a top-down fashion, identifying the independent tasks of a graph, and attempting to cluster each task’s successors together. It was conceived using set operations, and has been effective both results quality and running time.

InSeP is designed for ad-hoc workflows, where execution and communication costs are rarely available a priori. Scheduling decisions are better deferred until runtime, where execution statistics can be used by a dynamic scheduler.

InSeP is described as follows. Its operation is illustrated in the context of the work-
flow in figure 4.11 used above. Consider the workflow as a directed graph $G = (V,E)$ with the set of vertices $V$ (representing script commands) and the set of directed edges $E$. A node’s incoming edges signify a command’s dependence on another commands (parents), whereas its outgoing edges represent its role as a provider to other commands.

- **Step 1: Find roots.** Define a root of $G$ as a vertex $r$ which has no predecessors in $G$, where $r \in V, \emptyset = \{v : v \in V, v \prec r\}$. By construction, the roots of a workflow graph represent commands which can begin execution without regard for the progress of other commands in a workflow. Roots are named $r_i$, where $0 < i \leq n_r$ and $n_r$ is the number of roots. (Figure 4.15)

- **Step 2: Find root clusters.** Define a root cluster $C_{r_i}$ of a root $r_i$ as the set consisting of $r_i$ and all its direct and indirect successors. Clusters signify a unit of work to be dispatched and executed without interruption. These initial root clusters $C_{r_i}$ do not qualify because some vertices may be shared between other clusters. (Figure 4.16)

- **Step 3: Compute intersections.** Because $C_{r_i}$ may not be disjoint, compute their intersections $C_{r_{i,j}} = C_{r_i} \cap C_{r_j}$. Because $C_{r_{i,j}} = C_{r_{j,i}}$ (set intersection is unordered) and $C_{r_i,i} = C_{r_i}$, it is sufficient to compute intersections for $i, j \in [0, n_r], i < j$. Note that in figure 4.17, $C_{r_{1,2}} = C_{r_{1,3}} = C_{r_{2,3}}$ and are therefore not disjoint.

- **Step 4: Eliminate overlap.** Construct modified root clusters $C'_{r_i}$ by removing shared members, which are contained in the union of sets $C_{r_{i,j}}, j \neq i$. Sets $C'_{r_i}$ are disjoint, and represent clusters that can be dispatched without dependence on other clusters. Because the intersection sets may not be disjoint, construct $C''_{r_{i,j}}$ as the members of $C_{r_{i,j}} \notin C_{r_{i,k}}, \forall k < j$. Some $C''_{r_{i,j}}$ may have no members,
Figure 4.16: Step 2: Find root clusters

Figure 4.17: Step 3: Compute intersections
i.e. \( C'_{r_{i,j}} = \emptyset \). In figure 4.18, which illustrates step 4, note that \( C'_{r_{1,3}} = C'_{r_{2,3}} = C'_{r_{3,1}} = \emptyset \) and are eliminated. Recall that \( C_{r_{i,j}} \forall i \geq j \) are not considered, so it is also unnecessary to compute \( C'_{r_{2,1}}, C'_{r_{3,1}}, \) or \( C'_{r_{3,2}} \).

After Step 4, the entire workflow is contained in disjoint sets \( C'_{r_i} \) and \( C_{r_{i,j}} \) for \( 0 < i, j \leq n, i < j \).

- **Step 5: Connect resultant clusters.** To prepare the resultant clusters for dynamically scheduled execution, add directed edges between clusters to signify dependencies between clusters. (Figure 4.19)

- **Step 6: (Optional) Cut and recurse InSeP.** An application of steps 1-5 results in a number of clusters and a degree of parallelism according to the workflow’s topology. To expose greater parallelism, recursive application of InSeP may be desired. Because InSeP relies on the presence of multiple root clusters, and each \( C'_{r_i} \) has only one root node by construction, recursive application on \( C'_{r_i} \) is non-trivial, but straightforward. Cut points may be found in intersection
Figure 4.19: Step 5: Connect resultant clusters
clusters similarly.

- Step a: Find a cut point of $C'_{r_i}$ by traversing the graph from the root until a node with true fan-out is encountered. Define true fan-out as a node’s possession of at least two children $c_i, c_j$ which are not direct descendants of each other. If no nodes have true fan-out, $C'_{r_i}$ should not be split since no parallelism would be gained.

- Step b: Given a node $x$ with true fan-out found in step a, split $C'_{r_i}$ into one parent cluster $p = v : v \in r_i, x \cup u : r_i \prec u \prec x$, and one children cluster consisting of $C_x = x \cap v : x \prec v, v \in C'_{r_i}$.

- Step c: Recurse InSeP over the children cluster $C_x$, which has multiple roots by construction.

InSeP may be recursively applied to the resultant clusters as

4.4.4 Complexity Analysis

InSeP’s algorithmic complexity is presented below.

- **Step 1: Find roots.** A node’s classification as root may be tested by checking its parents’ membership in the graph. A node has $\frac{|E|}{2|V|}$ parents on average, and each check is $O(1)$ hash table lookup over vertices $v$ in $V$. Building the table is $O(|V|)$. Step 1 is therefore $O(|V| + |E|)$.

- **Step 2: Find root clusters.** Finding root clusters is $O(r_n|E|)$, and $O(|E|^2)$ in the worst case.

- **Step 3: Compute intersections.** Set intersection is quadratic in the number of members, so Step 3 is $O(|V|^2)$. 
• **Step 4: Eliminate overlap.** An efficient implementation can process Step 4 in conjunction with Step 3 with sufficient bookkeeping, so Steps 3-4 take \( O(|V|^2) \) together.

• **Step 5: Connect resultant clusters.** Step 5 can be accomplished using existing bookkeeping information from Steps 3-4.

• **Step 6: (Optional) Cut and recurse InSeP** Finding the cut point is approximately \( O(|E|) \). On average, the recursion depth is at most \( O(log|V|) \).

Overall, InSeP’s complexity is \( O(|V|^2log|V|) \) including the recursive step. Although InSeP is not the fastest algorithm, it is also conceptually simple and easy to perform by inspection, making implementation and debugging simple. In practice, InSeP’s partitioning cost is observed negligible with respect to the other costs in the system like parsing costs and especially execution cost.

### 4.4.5 Evaluation

Many DAG partitioning algorithms designed to divide work among a number of processors produce linear clusters. When the scheduling target is a set of processors, non-linear clusters, which are defined as containing at least two mutually independent tasks, potentially waste parallelism. InSeP does not attempt to produce linear clusters because it is intended to divide work among a set of processing instances which have their own parallel execution resources. Breaking InSeP’s clusters to produce linear clusters may cause reduce performance—dependent commands are separated and the loss of guaranteed locality between them may increase execution time.

InSeP’s performance and the general benefit of coarse-grained scheduling enabled by partitioning can be quantified by comparing the performance of InSeP-enabled
SWAMP to a command-at-a-time-dispatched SWAMP. Figure 4.20 compares SWAMP’s performance in the resample benchmark. Speedup is plotted versus node-count, using PPN$\in\{1,2,3\}$ in command-at-a-time (dashed lines) and clustered(solid lines) configurations. Comparing the clustered family of curves with the command family of curves, the clustered configuration has universally steeper slopes, indicating better performance with increasing worker count. Ideal performance would be indicated by line slopes that are equal to PPN, and it is clear that the non-clustered configuration exhibits nowhere near ideal performance due to intense overhead costs. Details on the overhead costs can be found in [54].

The quality of InSeP’s partitions can be examined by considering InSeP-assisted execution traces. Consider execution with PPN=4 over and 4 workers. If InSeP’s
Figure 4.21: IPCC 4x4 Clustered Execution Statistics

(a) CPU usage

(b) Disk transaction rate
Figure 4.22: resample 4x4 Clustered Execution Statistics

(a) CPU usage

(b) Disk transaction rate
partitions were balanced, all workers would have similar CPU usage curves, being busy and idle at roughly the same time. In a perfectly balanced case, they would also finish their allocated work simultaneously, after accounting for imbalance from the workflow’s specific topology. For ipcc, since execution is divisible into approximately 19 parallel clusters and 1 approximately-serial cluster, the allowed imbalance is estimated at 1/20th the baseline serial execution time, or ≈23 seconds. resample has significantly more tasks than ipcc, numbering ≈ 14,000 compared to ipcc’s 60, and should be enable near elimination of imbalance. However, InSeP’s conservative partitioning, especially without aggressive recursion, means that resample is only divided into its 10 large clusters, giving an expected imbalance of roughly 1/10th the baseline, or ≈340 seconds.

Figure 4.21a plots the non-idle CPU fraction for the ipcc benchmark in the 4-PPN, 4-worker case, where it is clear that one worker has completed especially early. One worker is expected to complete 23 seconds earlier than most, due to imbalance, and one worker is expected to complete 23 seconds after most, to complete the non-parallel portion. Worker compute-0-2 is the earlier finisher, but appears, visually, to complete about 30 seconds earlier than most. Worker pbs (co-located with the master process) finishes roughly 20-25 seconds afterward, which corroborates with the expectation that one node finish later due to the serial cluster. These timings are corroborated in figure 4.21b, which plots disk transaction rate.

Figure 4.22 plots the same traces for the resample benchmark. resample has no serial portion like ipcc, but again, one may expect, due to conservative partitioning, an imbalance of ≈340 seconds between earlier and later finishing workers. later finishers. Visual inspection of figure 4.22a shows an imbalance between 70-90 seconds. This can be explained by the large amount of parallelism available within resample’s clusters. Since the nodes are configured with PPN=4, one might estimate the imbalance under
ideal, 4-way parallelism to be \( \frac{340}{4} \) seconds, or 85 seconds, which corroborates with the visual inspection. The total execution time with an ideal balance can be estimated by computing the average of the finish times of each worker, which yields approximately \( t=540 \) and a total execution time of about 530 seconds. With an actual execution time of 602 seconds, InSeP’s imbalance costs roughly 13% of overall execution time. Subtracting out the parsing and compiling components, the estimated balanced computation time is 220 seconds, and given an actual computation time of 293 seconds, InSeP imbalance costs roughly 33%.

Figure 4.22b plots disk transaction rate for resample, but is not useful for illustrating finish times. This is because SWAMP’s aggressive ramdisk optimization has eliminated all disk access except for initial source reading and final output writing. Final output writing itself may be avoided by aggressive operating system caching—a SWAMP client, by default, downloads its results immediately after its workflow completes and subsequently sends a message releasing the workflow’s resources (including output data).

### 4.5 Conclusion

This section considers the problems of I/O-constrained parallel workflow performance under data-intensive tasks. Benchmarks and execution traces quantify the impacts of high data-intensity for execution on various configurations of parallel hardware. Testing parallelism using multiple processor cores versus multiple discrete computers, shows that data-intensive workloads generally favor multiple machines due to their added I/O resources, despite the potential reduction in data locality caused by spreading execution across more machines. An implementation of a dynamic hierarchical workflow scheduler is also described, noting the advantage of dynamism.
when execution and communication costs are unknown. Finally, the InSeP DAG workflow partitioning algorithm is described, comparing its results with other partitioning algorithms, evaluating its algorithmic complexity, and testing its efficacy under data-intensive workflows.
Chapter 5

Analytical Extension

This research is motivated strongly by the inability of current hardware and software to address the needs of the geoscience community. This chapter examines the relevance of this work in light of expected developments in hardware and software technology.

5.1 Hardware Advancements

The techniques studied in this work are designed to address two key deficiencies in hardware—limited long-distance bandwidth and the gap between I/O and CPU performance.

5.1.1 Increased Network Bandwidth

Advances in networking technology have been frequent and significant. Commodity machines now ship with 1Gbit/s networking connectivity, and 10Gbit/s connectivity
is nearing production. Backbone providers cite lack of demand for bandwidth as a
double cause for the alleged prevalence of “dark fiber,” or unused fiber-optic links. The shell
compilation methods presented were developed to ease the portability of data analysis
scripts from a workstation to a remote data center. Bandwidth local and distant
will always grow, perhaps leading one to question the general theme of relocating
computation rather than data.

Despite continual advance in networking, the growing popularity and production rate
of huge datasets will ensure the advantage of moving computation rather than data.
Computational and data storage capabilities advance with, and perhaps faster than,
networking bandwidth, thus encouraging larger datasets. For example, the Large
Hadron Collider collaboration is expected to record approximately 10 petabytes of
data per year after filtering and processing.

5.1.2 Increased Core Multiplicity

While microprocessor peak performance has continued to increase as according to
Moore’s law, achieving peak performance now requires efficient use of multiple pro-
cessor cores. This research has demonstrated the benefits of exploiting high-level
parallelism from scripts of unparallelized programs, and is well-positioned to take
advantage of future trends in microprocessors.

5.1.3 Recommendations

Although drastically increased long-haul bandwidth would undoubtedly please scien-
tists and others interested in bulky data, such a solution is not economically feasi-
ble. Challenges in semiconductor physics will continue to limit future increases in
processor performance for single-threaded applications. With computing hardware advancements in core multiplicity and storage density of limited benefit to end users, an alternate proposal is development of commodity storage parallelism.

Commodity computers are commonly sold with four processors, but only a single disk, which limits parallelism potential. The development of more parallelism in storage systems and other peripherals is recommended.

### 5.2 Software Advancements

#### 5.2.1 Application Parallelism

Although shell compilation provides additional ways to exploit hardware parallelism, there is a possibility that applications themselves will implement enough parallel code to a sufficient degree that the need for higher levels of parallelism become rare. Grid frameworks already expose massive amounts of general computation capability which may be exploited through MPI and other parallel APIs. With multi-core processors still a recent development, one may explain the current lack of application parallelism as caused by a temporary lack in parallel programming expertise which will disappear as programmers are retrained. Yet anecdotal experience suggests that programmers consider parallel code only necessary for high-performance computing, and prefer to defer parallelism to operating system developers and the high-performance computing niche.

With shell compilation’s parallelism efficacy dependent on the manual specification of program behavior, it may not be obvious why no similar effort was undertaken to extract parallelism over function calls whose unpredictable side-effects present similar
problems as program unpredictability. One possibility is that function calls whose execution times are long enough to justify a scheduling overhead are relatively rare and unique in nearly every program. On the other hand, scripts that contain calls to only “expensive” programs are quite common—their execution is invoked from a script to reduce user boredom and inefficiency from waiting for them to complete.

5.2.2 Filesystem Parallelism

Although the filename remapping technique is admittedly primitive in comparison to true filesystem virtualization, its effectiveness for performance is real. Because filesystems greatly influence disk access and nobody doubts that disks are bottle-necks, future filesystem research may develop alternate techniques for reducing disk contention and enforcing security.

However, a more advanced filesystem or layer on top of a filesystem such as Parrot[33] is somewhat limited because its only knowledge of the workload comes from the sequence of filesystem API system calls. Although its ability to control disk access is direct and not generally bypassable, it does not have access to semantic information about the larger patterns of disk accesses. Compilers of general purpose languages also have little understanding of disk access semantics, despite access to far greater information than the filesystem.

Shell compilation is disadvantageous in characterizing and optimizing disk access because it must rely on manual specification of program behavior, which may not (it does not, currently) include access pattern characterization. Yet on a higher level, it can determine file lifetimes, over a sequence of program executions and make optimizations accordingly. This is a distinct advantage of shell compilation in file access optimization that is not easily duplicated at lower levels.
5.2.3 Recommendations

Despite their disadvantages in optimizing disk access, filesystems have significant potential in using structures that support parallel access. One possibility is the idea of having filesystem structures that support concurrent reading and writing. Such a capability is allegedly offered in Sun Microsystem’s ZFS, which supports parallel reads and writes to the same file.
Chapter 6

Areas of Further Study

6.1 Resource Constrained Performance Modeling

6.1.1 Introduction

Experiences in analyzing performance of the SWAMP system have shown that it is difficult to model performance of systems whose performance does not exclusively depend on a single linearly-sharable resource like processing capability or whether parts of the program are optimizable. How can the performance of a I/O-constrained workflow be estimated? In developing a model for estimation, it is instructive to start by calculating the upper-bound on speedup, that is, the maximum speedup we can obtain from the workflow.
6.1.2 Single-Factor Performance

Amdahl’s Law considers only a single factor for performance, that is, a generic factor of “processing performance”. In practice, CPU processing performance is used, and the Law is therefore used to estimate benefit from CPU parallelism. Alternatively, for completely I/O-bound applications, evaluation proceeds using a measurements of I/O performance to estimate the benefit from I/O parallelism.

**Trivially-parallelizable Estimation**

When all nodes are independent, the workflow is considered “embarrassingly parallel.” Let the workflow be defined as a set of $n$ subtasks where each takes $t_i$ time to execute. The upper-bound of speedup is then given as:

$$ speedup = \frac{\sum_{i=1}^{n} t_i}{\max(t_i)} \quad (6.1) $$

This reduces quickly if $t_i = t_j = c$ for all $i$.

$$ speedup = \frac{\sum_{i=1}^{n} c}{c} = \frac{n \times c}{c} = n \quad (6.2) $$

If the times are not uniform, then speedup is better computed using:

$$ speedup = \frac{\sum_{i=1}^{n} t_i}{\max(t_i)} \quad (6.3) $$

This simply means that execution is limited by the longest subtask.

Unfortunately, these trivially-parallelizable conditions do not exist for workflows, since they do not allow subtasks to be interdependent as they are in workflows.
Staged Estimation

A better way to estimate workflow performance is to first divide the workflow into stages by laying the subtasks out as nodes in a graph, where their height/depth is measured by their maximum path to a root subtask. In this case, speedup can be calculated as:

\[
\text{speedup} = \sum_{i} n_s w_i * n_i
\]  

(6.4)

where \(n_s\) is the number of distinct stages, equal to the depth of the deepest subtask, \(w_i\) is the fraction of total execution time spent in that stage, and \(n_i\) is the maximum speedup of that stage, equal to the number of subtasks in that stage.

This is much closer than the trivially-parallelizable case, and is a nice, practical, understandable way to look at performance for logically-structured workflows. Most human-generated workflows are structured thusly for practical reasons.

Amdahl's law can be compared to a two-staged estimation. Using \(n_i\) as speedup for stage \(i\) is troublesome, since it assumes, again, that each subtask in a stage has equal execution time. What if the times are non-uniform?

Critical-path Estimation

In considering non-uniform task estimation time, the upper-bound performance by considering the workflow’s critical path. Define the critical path of a workflow to be the longest (measured in execution time) root-leaf path, where a root-leaf path is a
path from a root subtask to one of its leaf subtasks. Then calculate speedup as:

\[
speedup = \frac{\sum_{i}^{n} t_i}{\sum_{\text{critical-path subtasks}} t_i}
\]  \hspace{1cm} (6.5)

where \( t_i \) is the execution time of a particular subtask.

The simplicity of this calculation overlooks the complexity involved in determining the critical path from all root-leaf paths. Critical path calculation is possible through standard CAD tool algorithms that calculate circuit critical paths.

### 6.1.3 Multiple-factor Performance

Single-factor estimations are useful in most cases, but are unable to cope where performance is impacted significantly by multiple factors. Most grid applications can use single-factor estimation, but with the increasing importance of I/O, single-factors are no longer sufficient.

I/O is under-studied as an performance consideration, other than in maximizing peak throughput. However, many systems that provide peak throughput suffer greatly when handing multiple independent requests.

**Trivially-parallelizable Estimation**

Amdahl’s law can be adapted to support multiple factors.

\[
speedup = \min(speedup_i)
\]  \hspace{1cm} (6.6)

where \( speedup_i \) is the speedup calculated by one particular factor.
Consider the case where a program fully utilizes the CPU and the disk, and one wishes to run two instances of the program. Intuitively, one expects that on a the original system, the total time is $T = t + t = 2 \times t$. If another CPU is added, without adding another disk, one intuitively expects that the total time should remain the same (neglecting seek overhead, which will be discussed later). More precisely, the total time can be modeled as:

$$T = \max(t_{\text{disk}}, t_{\text{cpu}})$$  \hspace{1cm} (6.7)

This generalizes to:

$$T = \max(t_i)$$  \hspace{1cm} (6.8)

for all resources $i$, which corresponds to the speedup model in equation (6.6).

This models resource usage in terms of the amount of time required on each resource. Thus if another CPU is added, one can expect:

$$T = \max(t_{\text{disk}}, \frac{1}{2}t_{\text{cpu}})$$  \hspace{1cm} (6.9)

which indicates an unchanged overall execution time.

**Staged Estimation**

The idealized multi-factor estimation can be extended to a staged estimation similarly as with a single-factor model.
Accounting for non-ideal resource characteristics

Thus far, this discussion has considered resources to have no concurrence penalties. This means that single resources, a single CPU core or a single disk, provide the same throughput regardless of the multiplicity of demand. In practice, this does not hold—for example, single CPUs accomplish multitasking through periodic context switching. Context switches require finite non-zero time, during which the CPU is not making progress in program execution. Disks suffer greater concurrence penalties, since the time to switch, i.e. the time to move the disk arm from a location servicing one program to another serving the other, is large with respect to the time to read/write data of average length.

6.1.4 Finite resource modeling

The above analysis considers modeling only maximum speedup. To estimate performance with limited resources, some assumptions must be altered. Corresponding to the equation presented in (6.1), we now have:

\[
\text{speedup} = \left\lceil \frac{\sum^n_i t_i}{\sum_{a=0}^{\lceil \frac{n}{n_r} \rceil} \max_{i=(a)n_r..(a+1)n_r} (t_i)} \right\rceil \tag{6.10}
\]

For constant subtask time \( t_i \), this reduces to:

\[
\text{speedup} = \frac{n}{\lceil \frac{n}{n_r} \rceil} \tag{6.11}
\]

We note that (6.10) depends on a particular scheduling scheme which dispatches in groups of \( n_r \) subtasks, waiting until all \( n_r \) subtasks in a group complete before dispatching the next group. This scheme is sensitive to the selection of subtasks for each dispatch group and performs best when each group is composed of subtasks of
similar length (achievable by sorting subtasks in descending order by time).

A more performant scheduler would schedule subtasks resources to independently so that each execution resource is given approximately the same amount of work. However, finding an optimal placement of subtasks among resources is a form of the knapsack problem, a classic maximization problem in combinatorics which is NP-complete. Even so, an exponential-time algorithm may be applied if $n_r$ is sufficiently small.

### 6.1.5 Conclusion

The modeling problem presents significant challenges, but as changing hardware technologies encourage parallel rather than serial algorithms, more complex models are needed to better describe performance. Amdahl’s law has survived decades of technological progress, but the time is nigh for the development of another model which can address complex hardware factors while being simple enough to be intuitive.

### 6.2 Pervasive Shell Compilation

#### 6.2.1 Definition

Pervasive shell compilation is defined as shell compilation in the context of everyday shell usage. This implies that parallelization may occur without the user’s explicit specification. Since implicit program parallelization generally violates user expectations of interactive shell behavior, care must be taken to minimize or eliminate surprising results. One strategy is to require users to whitelist commands for parallelization in the same way as they are whitelisted and supported for shell script
compilation. Including visual feedback in-band at the console should also remind the user of parallelism.

6.2.2 Benefit

Users often employ specify sequences of commands in one-at-a-time form. After each sequential command completes, the next is typed in and issued. This requires significant user attention to minimize idle time between one command’s completion and the its following command’s issuance. The user must monitor command progress for much of the sequence’s overall execution, regardless of whether feedback is needed to adjust the sequence. In current shells, the solution requires users to specify and invoke the sequence as one unit. However, this requires user foresight and additional user precision, since commands in the sequence are not parsed until their predecessors complete, and filename completion is unavailable for files produced during the sequence. Because of those deficiencies, typographical errors in naming commands and filenames (especially those adhering to specific naming conventions) are much more likely.

The significant performance advantages discussed in Chapter 3 are unquestionably useful as well.

6.2.3 Challenges

Implicit compilation and optimization during interactive shell usage contains significant challenges; there are good reasons why interpreted languages like Python do not have always-on compilation and optimization despite the benefits. Yet the benefits of optimization are far higher in a shell context, since program execution is generally
far more expensive and far more hungry for performance tweaks than a function call.

One of the biggest challenges of pervasive compilation is the limited future knowledge: because future file access is unknown, a compiler cannot reason about file lifetime and cannot, for example, garbage-collect (delete) dead files. However, one optimization is still possible—choosing to always write files directly to memory and lazily writing to disk opportunistically like an extra-aggressive cache. The additional write delay is clearly dangerous in interactive situations. Script execution can be restarted, especially if results are batched and committed in a block, but interactive commands are not always repeatable—even if they are, it may be difficult to tell the difference.

A loosened correctness model was important in enabling optimization for shell scripts. It makes sense in a script, because users can overlook intermediate state consistency as long as everything is consistent and correct at the end. Loose correctness is unsafe in an interactive model, however, because the current state can always be inspected (as is probably the expectation). To address this, a system might need to obtain user permission. One way might be to let users explicitly start and complete transactions at the command line (e.g., \texttt{start job1; foo a b; foo b c; foo b d; finish job1;}). This console-level transaction API seems promising, but it is unclear whether it would be sufficiently simple, understandable, and \textit{obviously useful} to be adopted by users.

Lastly, in manipulating file access behavior, a system would need to trap and process external file accesses in order to preserve the illusion of correctness. In order to get a big picture of the dataflow, the system must know when files are accessed. In a script, it is reasonable to expect that the results will not be read until the script terminates; there is much freedom to optimize internally. Without the confines of a script, a program’s results, e.g. its output files, are expected to be visible to any other system process. Likewise, a program’s inputs may come from any other program. It
is especially unclear how to solve this problem of needing system-wide knowledge, extending beyond the confines of the shell compilation process.

### 6.3 Behavioral Specification for Program Parallelism

While I have provided an implementation of a system that parallelizes programs in the geoscience domain, it is unclear that the ad-hoc specification used is scalable to large numbers of programs. The SWAMP system requires support for programs to be programmed in Python and inserted in the appropriate places. However, such a system is not resistant to future developments (such as changes in Python) and may be unnecessarily dependent on arbitrary SWAMP design choices.

In treating shell scripts as programs themselves, and their internal program calls as functions, it is particularly noticeable that there is no widely-accepted means of specifying argument syntax. This is not aided by the wide variety of argument syntaxes (not to mention semantics) supported by binary programs, which makes such specification difficult. However, a basic specification of program argument syntax, i.e. some interface definition language (IDL), would simplify user effort in specifying programs for automatic parallelization.

### 6.4 Adaptive Resource Allocation

Scripts often go through stages of execution need. Ideally, the system would allocate extra devices on demand and deallocate them when unneeded. The memory system already does this, and it should be possible to accomplish for disks and other devices.
Consider a system which allocates additional disks (powering them up if necessary) when additional write bandwidth is needed. Files can be lazily consolidated into their final destination, and the disks deactivated (and powered down, or allocated to other tasks) later. A similar method may manage computers in a cluster. In most parallel execution systems like Sun Grid Engine, each job specifies how many CPUs are required, and the scheduler guarantees those CPUs to the job. However, the job may not need all those processors all the time. The hand-crafted solution is to break the task into portions which each have well-defined constant resource needs and to schedule them separately, but such effort is tedious and error-prone. An automatic system could enable workflow execution, in particular, to conserve resources and still attain maximum performance by dynamically adjusting the number of allocated cluster machines.
Chapter 7

Conclusion

Fueled by the proliferation of cheap and increasingly parallel computing hardware, parallel and distributed computing continues to be an important area of research. At the same time, the climbing data volumes and the demand to share such data necessitate new, effective means of handling such high volumes—volumes where data transfer will always be difficult since their size grows at least as fast as the available bandwidth. The first major contribution of this research is its characterization of a particular class of workloads in terms of the I/O-related factors which limit and drive their performance, along with both proposing and implementing techniques which can optimize their performance where traditional methods are inadequate and perhaps inappropriate. Alternative techniques in both general and grid workflows appear to address these issues, but this research has found them only effective when adapted and combined with less traditional I/O-aware methods.

Geoscience data reduction and analysis typifies workloads of this class. Characteristically data intensive, these tasks are ill-suited for traditional HPC techniques which treat data performance as a dependency rather than a goal on-par or exceeding com-
putational throughput. The second major contribution is the design and implementation of a method for transparently migrating scientists’ computation and transferring results, avoiding tedious and slow data transfer. This server-side processing system (SWAMP) enables users to leverage under-utilized server-side resources using an interface whose semantics appear as simple as existing scripted methods—only faster. The methods and their effectiveness in real data processing performance contradict the idea that only flop-intensive workloads are worth parallelizing and optimizing.

The last major contribution of this work is the study and implementation of a system for the compilation of “shell scripts.” A definition of shell compilation is presented, clarifying what is possible (parallelization) and impossible (universal, automatic optimization). Highlighted are the advantages of treating scripts as dataflow computations, along with the difficulties of automatic characterization of programs. Examples are provided of the potential such a system to provide parallelized execution of sequential programs, an important advantage since the cost of writing a program’s high-level behavioral specification is tiny compared to the cost of rewriting and parallelizing a sequential program (provided it is even possible).

These techniques, which have been tested with real, useful geoscience analysis tasks, have been successful at addressing the initial motivating problem, giving geoscientists the ability to perform terascale data analysis of data more than a thousand miles away, without significantly deviating from local workstation practices, and making the entire analysis process (idea conception, specification, remote and parallel execution, final results) faster than if the data were local, and fast enough to be possible and practical multiple times a day.
Bibliography


Appendices

A Appendix A: Shell Optimization Listing

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant folding and propagation</td>
<td>Variable references are looked up during compile time and substituted</td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>Loop bodies are unrolled to increase optimization scope</td>
</tr>
<tr>
<td>Induction variable analysis</td>
<td>Loop index variables are evaluated during loop unrolling</td>
</tr>
<tr>
<td>Conditional constant propagation</td>
<td>Conditional expressions are evaluated to enable branch-free dataflows</td>
</tr>
<tr>
<td>Dead code elimination</td>
<td>Unreachable branches are eliminated</td>
</tr>
<tr>
<td>Filename abstraction</td>
<td>File access is abstracted to allow optimized filesystem usage</td>
</tr>
<tr>
<td>Live variable analysis</td>
<td>File liveness is detected so that script output may be optimized</td>
</tr>
<tr>
<td>Disk avoidance</td>
<td>Abstracted filenames can be resolved to in-memory locations</td>
</tr>
</tbody>
</table>

B Appendix B: Implementation Guide

B.1 List of supported SWAMP shell syntax

- **Variables:** Shell and environment variables are supported for the purposes of parameter substitution. Unlike standard shells, however, they share a common namespace and are not exported to the OS-tracked per-process environment.
Both $\text{VARNAME}$ and ${\text{VARNAME}}$ syntaxes are supported. Example:

\begin{verbatim}
SRC='\text{/home/wangd/data}'
ncwa $\text{SRC/ccsm01.nc avg01.nc}$
\end{verbatim}

- **Wildcard filename globbing:** Wildcard globbing is supported using the characters "*" and "?" in their standard interpretation (“0 or more characters” and “1 character”, respectively). Globbing is processed before execution in a virtual namespace to facilitate SWAMP’s automatic dependency detection. Example:

A sample expansion of

\begin{verbatim}
ncra ????-1*.nc foo.nc
\end{verbatim}

might be:

\begin{verbatim}
\end{verbatim}

- **Loops:** Simple loop constructions are supported. Example:

\begin{verbatim}
\text{for i in 01 02 ; do}
       ncra cam2.h0.????-$i$.nc cam2_clm$i$.nc
\text{done}
\end{verbatim}

- **Backtick expressions:** Backtick expressions which invoke a whitelisted set of commands (currently \texttt{printf} and \texttt{seq}) are supported. This eases generation of filenames in sequence and according to popular file naming conventions. Example:

\begin{verbatim}
START='01'
END='10'
for $\text{yra in }$ 'seq $\text{START }$ ${\text{END}}' ; do
       yr='printf "%02d" ${\text{yra}}'  \\
       ncwa ccsm${\text{yra}}.nc avg${\text{yra}}.nc
\text{done}
\end{verbatim}
• **Conditional branching:** Simple conditional branching is supported using 
\( =,==, < \) and \( > \) operators. Example:

```bash
typ=1 # 0=test
prd=1#
if [ "${typ}" = '0' ] ; then
    ncra -0 ~/nco/data/in.nc ~/foo.nc
elif [ "${typ}" = "${prd}" ] ; then
    for yra in 'seq 1 4' 'seq 5 12' ; do
        yr='printf "%02d" ${yra}';
        ncra -0 ${yr}.nc foo${yr}.nc
    done
fi # !prd
```

• **Special global variables:** In order to help script writers detect their running environment, SWAMP defines a few global variables.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
<th>Sample value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWAMPVERSION</td>
<td>SWAMP version number</td>
<td>0.1+</td>
</tr>
<tr>
<td>SWAMPHOSTNAME</td>
<td>Running SWAMP instance's hostname.</td>
<td>pbs.ess.uci.edu</td>
</tr>
<tr>
<td>SHELL</td>
<td>Running SWAMP instance's hostname.</td>
<td>pbs.ess.uci.edu</td>
</tr>
<tr>
<td></td>
<td>Usually a fully-qualified domain name (FQDN)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Name of running shell</td>
<td>swamp</td>
</tr>
<tr>
<td></td>
<td>SWAMP: &quot;swamp&quot;;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>other env.: &quot;/bin/bash&quot; or &quot;/bin/sh&quot;</td>
<td></td>
</tr>
</tbody>
</table>

**B.2 Benchmark scripts**

*resample* Benchmark

Abbreviated contents of `full_resamp.swamp`. The complete version as used for benchmarking can be downloaded from swamp.google.com (svn: [http://swamp.googlecode.com/svn/trunk/scripts/full_resamp.swamp](http://swamp.googlecode.com/svn/trunk/scripts/full_resamp.swamp)) *resample* was the first benchmark used. Since loops and shell variables were unsupported at the time of its writing, its several loops are unrolled and its variables are substituted using a specification written in the m4 macro language.
# The lines below are repeated for years 0002 through 0011 (10 years total)
crcat -O camsompdf/camsompdf.cam2.h1.0002*.nc \ %tempf_yr_0002cat.nc% # pack the year into a single series
# AM sampling
ncap -O -s "loctime[time,lon]=float(0.001+time+(lon/360.0))" \ -s "mask2[time,lon]=byte(ceil(0.006944445-abs(loctime-floor(loctime)-0.25)))" \ %tempf_yr_0002cat.nc% %tempf_yrm_0002am.nc%  
ncwa -O --rdd -v WINDSPD -a time -d time,0,71 -B "mask2 == 1" \ %tempf_yrm_0002am.nc% %tempf_yr_0002d0_am.nc.deleteme% 
ncap -O -h -s "time=0.25+floor(time)" %tempf_yr_0002d0_am.nc.deleteme% \ %tempf_yr_0002d0_am.nc% 
# rm yr_0002d0_am.nc.deleteme
ncwa -O --rdd -v WINDSPD -a time -d time,72,143 -B "mask2 == 1" \ %tempf_yrm_0002pm.nc% %tempf_yr_0002d1_am.nc.deleteme% 
ncap -O -h -s "time=0.75+floor(time)" %tempf_yr_0002d1_am.nc.deleteme% \ %tempf_yr_0002d1_am.nc% 
# rm yr_0002d1_am.nc.deleteme
#.
#.
# repeat pattern for all days (0-364).
ncwa -O --rdd -v WINDSPD -a time -d time,26208,26279 -B "mask2 == 1" \ %tempf_yrm_0002d364.am.nc.deleteme% %tempf_yr_0002d364_am.nc.deleteme% 
ncap -O -h -s "time=0.75+floor(time)" %tempf_yr_0002d364_am.nc.deleteme% \ %tempf_yr_0002d364_am.nc% 
# rm yr_0002d364_am.nc.deleteme

# PM sampling
ncap -O -s "loctime[time,lon]=float(0.001+time+(lon/360.0))" \ -s "mask2[time,lon]=byte(ceil(0.006944445-abs(loctime-floor(loctime)-0.75)))" \ %tempf_yr_0002cat.nc% %tempf_yrm_0002pm.nc%  
ncwa -O --rdd -v WINDSPD -a time -d time,0,71 -B "mask2 == 1" \ %tempf_yrm_0002pm.nc% %tempf_yr_0002d0_pm.nc.deleteme% 
ncap -O -h -s "time=0.75+floor(time)" %tempf_yr_0002d0_pm.nc.deleteme% \ %tempf_yr_0002d0_pm.nc% 
# rm yr_0002d0_pm.nc.deleteme
ncwa -O --rdd -v WINDSPD -a time -d time,72,143 -B "mask2 == 1" \ %tempf_yrm_0002pm.nc% %tempf_yr_0002d1_pm.nc.deleteme% 
ncap -O -h -s "time=0.75+floor(time)" %tempf_yr_0002d1_pm.nc.deleteme% \ %tempf_yr_0002d1_pm.nc% 
# rm yr_0002d1_pm.nc.deleteme
#.
#.
# repeat pattern for all days (0-364).
ncwa -O --rdd -v WINDSPD -a time -d time,26208,26279 -B "mask2 == 1" \ %tempf_yrm_0002d364_pm.nc.deleteme% %tempf_yr_0002d364_pm.nc.deleteme% 
ncap -O -h -s "time=0.75+floor(time)" %tempf_yr_0002d364_pm.nc.deleteme% \ %tempf_yr_0002d364_pm.nc% 
# rm yr_0002d364_pm.nc.deleteme
ncrcat -O %tempf_yr_0002d0_am.nc% %tempf_yr_0002d0_pm.nc% \ %tempf_yr_0002d1_am.nc% %tempf_yr_0002d1_pm.nc% \ %tempf_yr_0002d2_am.nc% %tempf_yr_0002d2_pm.nc% 
#
ipcc Benchmark

Contents of ipcctest.swamp. ipcc was written after variable support and loop support were implemented, permitting it to be specified more cleanly and concisely.

#!/bin/bash
# soon, #!/usr/local/bin/swamp_client.py --

# Purpose: Compute time anomaly from homogenized AR4 datasets
# Homogenized means that "raw" PCMDI files were:
# 1. Named with standard scn_mdl_run_YYYYMM_YYYYMM.nc convention
# 2. Have an area-weight variable named "area"

# Usage:
# python swamp_client.py ipcctest.swamp
#
# Another example:
# export SWAMPURL='http://pbs.ess.uci.edu:8080/SOAP'
# python "/swamp/src/swamp_client.py "/swamp/demo/ipcctest.swamp

# Switch between local and swamp path roots.
if [ $SHELL = "swamp" ] ; then
  IPCCROOT=/pcmdi
else
  exit
  exit
# Something like this would work with accounts on PCMDI's server
# if the files were named with the YYYY convention used below
  IPCCROOT=http://user:password@climate.llnl.gov/cgi-bin/dap-cgi.py/ipcc4
fi

models=(cccma_cgcm3_1 cccma_cgcm3_1_t63 cnrm_cm3 csiro_mk3_0 
gfdl_cm2_0 gfdl_cm2_1 giss_aom giss_model_e_h 
...
iap_fgoals1_0_g inmcm3_0 ipsl_cm4 miroc3_2_medres 
miub_echo_g mpi_echam5 mri_cgcm2_3_2a ncar_ccsm3_0 ncar_pcm1 
ukmo_hadcm3 ukmo_hadgem1)

# giss_model_e_r is only available in/for run2
# miroc3_2_hires is not available
# area variable not available in some (all?) of these

variables=tas,pr
scenarios=sresa1b
run=run1

for scn in $scenarios; do
    for mdl in $models; do
        # Should add -w area weight switch when datasets have area
        ncwa -O -v $variables -a lat,lon -p $IPCCROOT/$scn 
        ${scn}_${mdl}_${run}_200001_209912.nc 
        ${scn}_${mdl}_200001_209912_xy.nc
        # Anomalies are relative to year 2000 average
        ncwa -F -d time,1,12 ${scn}_${mdl}_200001_209912_xy.nc ${scn}_${mdl}_2000.nc
        # Compute anomalies for each model
        ncdiff ${scn}_${mdl}_200001_209912_xy.nc ${scn}_${mdl}_2000.nc ${scn}_${mdl}_anm.nc
    done # end loop over model
    # Create model ensemble
    ncea _200001_209912_xy.nc ${scn}_avg_200001_209912_xy.nc
    # Ensemble mean of year 2000
    ncwa -F -d time,1,12 ${scn}_avg_200001_209912_xy.nc ${scn}_avg_2000.nc
    # Create ensemble anomaly
    ncdiff ${scn}_avg_200001_209912_xy.nc ${scn}_avg_2000.nc ${scn}_avg_anm.nc
done # end loop over scenario

# The following are available pre-defined variables that you can
# reference in your scripts.
# SWAMPVERSION : "0.1+" -- version number, perhaps useful in the future.
# SWAMPHOSTNAME" : <hostname> -- the hostname for the swamp service,
# # probably a fully-qualified domain name (FQDN).
# SHELL : "swamp" -- This is always set to "swamp" in the SWAMP environment

B.3 SWAMP Service API

A SWAMP service instance exports the following API through SOAP and optionally
XML-RPC.
<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>newScriptedFlow</td>
<td>script(string), params(list(string))</td>
<td>token(int)</td>
</tr>
<tr>
<td>discardFlow</td>
<td>token(int)</td>
<td>(none)</td>
</tr>
<tr>
<td>pollState</td>
<td>token(int)</td>
<td>state(object)</td>
</tr>
<tr>
<td>pollOutputs</td>
<td>token(int)</td>
<td>outputUrls(list(string))</td>
</tr>
<tr>
<td><strong>Worker</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>registerWorker</td>
<td>cert(string), offer(url(string),slots(int))</td>
<td>token(int)</td>
</tr>
<tr>
<td>unregisterWorker</td>
<td>token(int)</td>
<td>(none)</td>
</tr>
<tr>
<td><strong>Administrative</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td><strong>Debugging</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pyInterface</td>
<td>command(string)</td>
<td>output(string)</td>
</tr>
</tbody>
</table>

Figure B.1: Function Signatures
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td></td>
</tr>
<tr>
<td>newScriptedFlow</td>
<td>Submit a new workflow script</td>
</tr>
<tr>
<td>discardFlow</td>
<td>Release resources of a workflow (e.g. output files)</td>
</tr>
<tr>
<td>pollState</td>
<td>Check workflow state (queued, running, finished)</td>
</tr>
<tr>
<td>pollOutputs</td>
<td>Retrieve URLs of output files</td>
</tr>
<tr>
<td><strong>Worker</strong></td>
<td></td>
</tr>
<tr>
<td>registerWorker</td>
<td>Register a new worker</td>
</tr>
<tr>
<td>unregisterWorker</td>
<td>Remove a registered worker</td>
</tr>
<tr>
<td><strong>Administrative</strong></td>
<td></td>
</tr>
<tr>
<td>reset</td>
<td>Resets service, clearing workflow queues; for administration and debugging</td>
</tr>
<tr>
<td><strong>Debugging</strong></td>
<td></td>
</tr>
<tr>
<td>pyInterface</td>
<td>Evaluate arbitrary Python commands within the instance; unsafe</td>
</tr>
</tbody>
</table>

Figure B.2: Description
B.4 Client Workflow State Diagram

- Start
  - Submit workflow
- Submitted
  - Poll, Job queued/in-progress
  - Poll, Job success
  - Poll, Job failure
- Complete
  - Poll outputs,
    - Download outputs
- Failed
- Finish