Efficient I/O for Computational Grid Applications

Ron A. Oldfield
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/dissertations

Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.dartmouth.edu/dissertations/5

This Thesis (Ph.D.) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Ph.D Dissertations by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.
Efficient I/O for Computational Grid Applications


A Dissertation
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of
Doctor of Philosophy
in
Computer Science
by
Ron A. Oldfield

DARTMOUTH COLLEGE

Hanover, NH
May 2003

Examinining Committee:

__________________________
(chair) David F. Kotz

__________________________
Thomas H. Cormen

__________________________
Robert S. Gray

__________________________
David E. Womble

__________________________
Carol L. Folt
Dean of Graduate Studies
Abstract of the Dissertation

Efficient I/O for Computational Grid Applications


by

Ron A. Oldfield

Doctor of Philosophy in Computer Science
Dartmouth College, Hanover, NH,
May 2003
Associate Professor David F. Kotz, Chair

High-performance computing increasingly occurs on “computational grids” composed of heterogeneous and geographically distributed systems of computers, networks, and storage devices that collectively act as a single “virtual” computer. A key challenge in this environment is to provide efficient access to data distributed across remote data servers. This dissertation explores some of the issues associated with I/O for wide-area distributed computing and describes an I/O system, called Armada, with the following features: a framework to allow application and dataset providers to flexibly compose graphs of processing modules that describe the distribution, application interfaces, and processing required of the dataset before or after computation; an algorithm to restructure application graphs to increase parallelism and to improve network performance in a wide-area network; and a hierarchical graph-partitioning scheme that deploys components of the application graph in a way that is both beneficial to the application and sensitive to the administrative policies of the different administrative domains. Experiments show that applications using Armada perform well in both low- and high-bandwidth environments, and that our approach does an exceptional job of hiding the network latency inherent in grid computing.
Acknowledgments

Thanks to my parents for their love and encouragement over the years. I can only hope to be as much of an inspiration to my children as they were to me.

Thanks to David Kotz for his guidance and friendship. He has a great instinct for knowing when to provide direction, when to provide encouragement, and when to get out of the way—all are qualities of a great advisor.

Thanks to Tom Cormen for his car, two pairs of tennis shoes, his pit smoker, and for teaching me that nothing is more important than good barbecue.

Thanks to the many graduate students that made my time at Dartmouth an enjoyable experience. Special thanks to Clint Hepner (my fishing companion), B.J. Premore (my hiking companion), and Senthil Periaswamy, a great friend and colleague.

Thanks to the many people at Sandia National Laboratories that provided guidance (and funding) for my research. Special thanks to Jeff Nelson, Bill Camp, and David Womble. They are each great mentors and deserve much of the credit for my success.

Thanks to Jay Lepreau, and the students and staff who run the Emulab at the University of Utah, for allowing us to use their facility to run our experiments.

Most of all, thanks to my wife Susan for her loving support over the past six years.
# Contents

Abstract ii  
Acknowledgments iii  
List of Tables vii  
List of Figures ix  
List of Algorithms x  

1 Introduction 1  
1.1 What are computational grids? 2  
1.2 Data-intensive grid applications 3  
1.3 Contributions of this dissertation 3  

2 The Armada Framework for Parallel I/O 5  
2.1 Background and Motivation 5  
2.1.1 Evolution of parallel I/O systems 5  
2.1.2 The need for flexibility 6  
2.1.3 The need for remote application code 7  
2.2 The Armada Framework for Parallel I/O 7  
2.3 Armada ships 8  
2.3.1 Types of ships 10  
2.3.2 Properties of ships 11  
2.4 Armada graphs 13  
2.5 Implementation 13  
2.5.1 Ships 15  
2.5.2 Graphs 17  
2.5.3 Harbors 21  
2.6 Related Work 22  
2.6.1 The data-flow computation model 22  
2.6.2 Parallel I/O systems 22  
2.6.3 Support for remote execution of application code 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Restructuring an Armada Graph</td>
<td>26</td>
</tr>
<tr>
<td>3.1 The Restructure Algorithm</td>
<td>26</td>
</tr>
<tr>
<td>3.2 Restructuring series-connected nodes</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1 A swap must produce an SP-tree</td>
<td>28</td>
</tr>
<tr>
<td>3.2.2 Configurations that do not produce an SP-tree</td>
<td>29</td>
</tr>
<tr>
<td>3.2.3 Commutativity of series-connected ships</td>
<td>35</td>
</tr>
<tr>
<td>3.2.4 A swap should benefit the application</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Algorithm summary</td>
<td>40</td>
</tr>
<tr>
<td>3.4 Analysis</td>
<td>40</td>
</tr>
<tr>
<td>3.5 Related Work</td>
<td>43</td>
</tr>
<tr>
<td>3.5.1 Automatic parallelization of user code</td>
<td>43</td>
</tr>
<tr>
<td>3.5.2 Operation ordering to improve data flow</td>
<td>44</td>
</tr>
<tr>
<td>4 Placement of Armada Ships</td>
<td>46</td>
</tr>
<tr>
<td>4.1 Graph partitioning</td>
<td>46</td>
</tr>
<tr>
<td>4.2 The Chaco partitioning software</td>
<td>47</td>
</tr>
<tr>
<td>4.2.1 Spectral partitioning with KL local refinement</td>
<td>47</td>
</tr>
<tr>
<td>4.2.2 Chaco parameters used by Armada</td>
<td>47</td>
</tr>
<tr>
<td>4.2.3 Adapting Chaco to work with Armada</td>
<td>48</td>
</tr>
<tr>
<td>4.3 Placement of the Armada ships</td>
<td>48</td>
</tr>
<tr>
<td>4.3.1 Creating a model of the data flow</td>
<td>49</td>
</tr>
<tr>
<td>4.3.2 Partitioning to administrative domains</td>
<td>50</td>
</tr>
<tr>
<td>4.3.3 Assigning Armada ships to hosts within each domain</td>
<td>51</td>
</tr>
<tr>
<td>4.4 Summary</td>
<td>51</td>
</tr>
<tr>
<td>5 Evaluation</td>
<td>53</td>
</tr>
<tr>
<td>5.1 The network testbed</td>
<td>53</td>
</tr>
<tr>
<td>5.2 Representative application</td>
<td>54</td>
</tr>
<tr>
<td>5.2.1 Experiments</td>
<td>55</td>
</tr>
<tr>
<td>5.2.2 Results and discussion</td>
<td>57</td>
</tr>
<tr>
<td>5.3 File transfer and permutation</td>
<td>60</td>
</tr>
<tr>
<td>5.3.1 Design</td>
<td>60</td>
</tr>
<tr>
<td>5.3.2 Implementation</td>
<td>61</td>
</tr>
<tr>
<td>5.3.3 Experiments</td>
<td>62</td>
</tr>
<tr>
<td>5.3.4 Results and discussion</td>
<td>64</td>
</tr>
<tr>
<td>5.4 Remote seismic imaging</td>
<td>74</td>
</tr>
<tr>
<td>5.4.1 Design</td>
<td>75</td>
</tr>
<tr>
<td>5.4.2 Implementation</td>
<td>80</td>
</tr>
<tr>
<td>5.4.3 Experiments</td>
<td>84</td>
</tr>
<tr>
<td>5.4.4 Results and discussion</td>
<td>87</td>
</tr>
</tbody>
</table>
5.5 Remote analysis of fMRI data ............................................ 93
  5.5.1 Design ................................................................. 93
  5.5.2 Conclusion ......................................................... 95

6 Conclusions and Future Work ............................................. 97
  6.1 Contributions ......................................................... 97
  6.2 Limitations of Armada ................................................ 98
    6.2.1 Applications need to make large requests ..................... 98
    6.2.2 Limitations in the graph representation ...................... 98
  6.3 Future work ........................................................... 98
    6.3.1 Building a complete I/O system for computational grids ........ 98
    6.3.2 Improving placement ............................................. 100
    6.3.3 Armada on a cluster of workstations .......................... 101
  6.4 Conclusion ........................................................... 101

Bibliography ............................................................... 103
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Configurations of series-connected ships in an SP-tree.</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>Summary of experiments.</td>
<td>54</td>
</tr>
<tr>
<td>5.2</td>
<td>Average size of data output queues lengths for selected ships from the restructured Armada graph for different WAN bandwidths.</td>
<td>87</td>
</tr>
</tbody>
</table>
## List of Figures

2.1 Evolution of parallel file systems. ........................................... 6
2.2 Simple Armada graph. ......................................................... 9
2.3 Hierarchy of Armada ships. ................................................. 10
2.4 Examples of structural ships. .............................................. 10
2.5 Example of a non-structural ship. ...................................... 11
2.6 Transformations of parallelizable ships. ............................... 12
2.7 SP-tree representation. ..................................................... 14
2.8 Architecture of an Armada ship. ......................................... 15
2.9 The RemoteShip interface. ............................................... 16
2.10 Document Type Description for Armada graphs. ....................... 18
2.11 XML document for a simple Armada graph. ............................ 19
2.12 Class hierarchy for SP-trees in Java. .................................. 20
2.13 The Harbor interface. ..................................................... 21
3.1 Restructured Armada Graph. ............................................. 27
3.2 Four cases for swapping neighboring ships. ............................ 30
3.3 The forbidden subgraph $N$. ............................................. 34
3.4 Forbidden configurations. .................................................. 34
4.1 Topological-sort order of the representative application. ............ 50
4.2 Simulated data flow through an Armada graph. ......................... 50
4.3 A partitioning of an Armada graph to three administrative domains. 51
4.4 A placement of Armada ships to three administrative domains. .... 51
5.1 Representative application. ............................................... 55
5.2 Four configurations of the filtering application. ....................... 56
5.3 Topology of network illustrating links between LANs. ................. 57
5.4 Execution time for the representative application. ..................... 58
5.5 Effective throughput for the representative application ................ 59
5.6 Armada graphs for a distributed file copy application. .............. 60
5.7 Restructured Armada graph for a distributed file copy application. 61
5.8 Logical data layout for the distributed file copy application. ......... 62
5.9 Virtual network topology for the baseline application. ............... 63
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.10</td>
<td>Two domain-level placements for the single file copy application.</td>
<td>64</td>
</tr>
<tr>
<td>5.11</td>
<td>Network topology for the distributed file copy application.</td>
<td>65</td>
</tr>
<tr>
<td>5.12</td>
<td>LAN placements for the original distributed file copy application.</td>
<td>66</td>
</tr>
<tr>
<td>5.13</td>
<td>LAN placements for the restructured distributed file copy application.</td>
<td>67</td>
</tr>
<tr>
<td>5.14</td>
<td>Performance of reading compressed data.</td>
<td>68</td>
</tr>
<tr>
<td>5.15</td>
<td>Performance of reading compressed data.</td>
<td>69</td>
</tr>
<tr>
<td>5.16</td>
<td>Execution time for file copy applications without compression.</td>
<td>70</td>
</tr>
<tr>
<td>5.17</td>
<td>Effective throughput of file copy without compression.</td>
<td>71</td>
</tr>
<tr>
<td>5.18</td>
<td>Execution time of file copy applications with different placements.</td>
<td>72</td>
</tr>
<tr>
<td>5.19</td>
<td>Throughput for the file copy application with different placements.</td>
<td>73</td>
</tr>
<tr>
<td>5.20</td>
<td>Description of seismic imaging.</td>
<td>74</td>
</tr>
<tr>
<td>5.21</td>
<td>Compute partition for Salvo.</td>
<td>76</td>
</tr>
<tr>
<td>5.22</td>
<td>Armada graph for the input phase of Salvo.</td>
<td>77</td>
</tr>
<tr>
<td>5.23</td>
<td>Restructuring an Armada graph for the input phase of Salvo.</td>
<td>78</td>
</tr>
<tr>
<td>5.24</td>
<td>Armada graph for reading velocity data.</td>
<td>79</td>
</tr>
<tr>
<td>5.25</td>
<td>Armada graph for writing an image plane.</td>
<td>81</td>
</tr>
<tr>
<td>5.26</td>
<td>C++ class hierarchy for distributed datasets using Armada.</td>
<td>83</td>
</tr>
<tr>
<td>5.27</td>
<td>C++ declaration for the Dataset class.</td>
<td>83</td>
</tr>
<tr>
<td>5.28</td>
<td>C++ declaration for the DistributedDataset class.</td>
<td>84</td>
</tr>
<tr>
<td>5.29</td>
<td>C++ declaration for the TraceDataset class.</td>
<td>85</td>
</tr>
<tr>
<td>5.30</td>
<td>Emulab network topology for the seismic experiment.</td>
<td>86</td>
</tr>
<tr>
<td>5.31</td>
<td>Original and restructured graphs with LAN placement.</td>
<td>88</td>
</tr>
<tr>
<td>5.32</td>
<td>Execution time for reading seismic data.</td>
<td>89</td>
</tr>
<tr>
<td>5.33</td>
<td>Effective throughput for reading seismic data.</td>
<td>90</td>
</tr>
<tr>
<td>5.34</td>
<td>Execution time for seismic application with different WAN latencies.</td>
<td>91</td>
</tr>
<tr>
<td>5.35</td>
<td>Throughput of seismic application for different WAN latencies.</td>
<td>92</td>
</tr>
<tr>
<td>5.36</td>
<td>Feature-space clustering description.</td>
<td>93</td>
</tr>
<tr>
<td>5.37</td>
<td>Armada graph for statistics calculations on a fMRI brain data.</td>
<td>94</td>
</tr>
<tr>
<td>5.38</td>
<td>Restructured Armada graph for fMRI data analysis.</td>
<td>95</td>
</tr>
<tr>
<td>5.39</td>
<td>Placement of the restructured Armada graph for fMRI analysis.</td>
<td>96</td>
</tr>
</tbody>
</table>
List of Algorithms

1. INITIALIZE($N$) ................................................................. 27
2. RESTRUCTURE($N$) .......................................................... 28
3. SLIDELEFT($B$) ............................................................... 31
4. SLIDERIGHT($A$) ............................................................. 32
5. PARALLELIZELEFT($P, B$) ................................................ 33
6. PARALLELIZERIGHT($A, P$) ........................................... 33
7. COMMUTATIVE($A, B$) .................................................... 35
8. BENEFICIALSWAP($A, B$) ............................................... 39
9. DIRECTION($A$) ............................................................. 40
10. CONVERTTOBINARY($N$) ................................................. 41
11. PROPAGATE-FLOW($G$) .................................................. 49
Chapter 1

Introduction

A recent trend in high-performance computing is the construction of “computational grids” composed of a heterogeneous collection of networks, computers, storage devices, and various other devices that collectively act as a single geographically distributed “virtual” computer [FK98]. Examples of such environments include Alliance’s National Technology Grid, NPACI’s Metasystem, NASA’s Information Power Grid, and DOE ASCI’s DisCom\(^2\) program. Applications for these environments use high-speed networks to logically assemble collections of resources such as scientific instruments, supercomputers, databases, and people into one “virtual” location.

Since computational grid applications may be geographically distributed, an important feature of a grid environment is an efficient parallel input/output (I/O) infrastructure for access to remote datasets. The problem, however, is that most prior work on parallel I/O and parallel file systems targeted tightly-connected homogeneous systems, and there are several fundamental differences between a grid environment and a tightly-connected parallel machine:

- A tightly-connected machine has “static” resources, meaning the number and location of compute nodes and I/O nodes remain fixed. The compute and disk resources on the grid are continually changing, which means a grid environment should be able to dynamically add and remove resources as they become available.

- The performance of the various devices on the grid vary dramatically. A tightly-connected parallel machine typically has consistent performance from the processors, the network, and the I/O subsystem. An application on the grid could be made up of processors and storage systems with varying abilities. In addition, the network performance may fluctuate dramatically during the lifetime of the application. Policies that assume consistent performance from the available resources will not work effectively in a grid environment.

- In a grid environment, computational resources may migrate, but large datasets cannot migrate or be duplicated without significant overhead. A dataset may span several different sites and the I/O system must define each portion of the distributed dataset in terms of the specific host locations where the data resides. A well-designed I/O system will abstract this site-specific information to provide a single logical view of the dataset.

- Unlike a tightly-connected parallel machine, the grid environment consists of resources that reside in different administrative domains. The local resources that were once controlled by a single group of
Chapter 1. Introduction

System administrators will now be available to anyone with access to the grid. Resource naming and access control will have to be consistent throughout the grid.

- It is likely that individual grid resources will each have an associated cost. People are not going to simply make their resources available to outside users unless there is some benefit to the resource owner. Application users will want to consider these costs when deciding which grid resources to use for their application. This additional constraint makes the task of allocating resources much more complicated for a grid environment than on a tightly-connected parallel machine.

With these differences in mind, the goal of our research is the design and implementation of an efficient parallel I/O infrastructure for grid applications.

1.1 What are computational grids?

In the 1998 book The Grid: Blueprint for a New Computing Infrastructure, Ian Foster and Carl Kesselman define a computational grid as “a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities” [FK98]. The word “grid” is meant to be analogous to the electric power grid, which provides pervasive access to power. The goal is to view computing as a resource for all areas of society, not just scientists and academics. Foster and Kesselman argue, for example, that university researchers use computing to study impact of land use on biodiversity, but city planners that select routes for new roads or plan for new zoning ordinances typically do not—and yet city planners have a large impact on our society.

The ability to “plug in” to a computational grid is appealing because it has several interesting applications. For example, computational grids can provide real-time computation for demand-driven applications, that is, applications that have infrequent requirements for large computing resources (e.g., a medical diagnosis system during surgery or a seismic simulation after an earthquake); they can use idle compute resources; and they can provide a means to share data that was computed or gathered at geographically distributed locations, for example, weather prediction simulations that use data gathered from many satellites and stored at locations around the world.

To get some practical insight into the value of computational grids, consider a data-intensive scientific application that analyzes data generated by a remote scientific instrument (for example a particle accelerator, radio telescope, or satellite weather monitor). The current standard practice for this kind of application has the four following steps:

1. Download the remote dataset to a temporary (local) storage location using FTP.
2. Pre-process the data to prepare it for computation, which may include extracting a subset, converting the data format, or distributing data to a local parallel file system.
3. Execute the computationally intensive data analysis on a local supercomputer, writing output to a temporary file system.
4. Post-process the output and store it on an archive.

All of the steps except the third step usually require direct human interaction. While this process may optimize the use of the supercomputing resource, it is clearly not a time-optimal solution if you consider the amount of human time used for steps 1, 2, and 4. The root problem with applications of this type is
that while the devices used to generate, store, and compute the data are connected by a common network (namely the Internet), until recently, no infrastructure existed to logically co-locate the distributed components. Computational grid frameworks provide this function by adding a software layer that abstracts the details associated with accessing distributed resources.

There are two primary challenges associated with building computational grids: heterogeneity and performance. Heterogeneity distinguishes computational grids from other platforms like supercomputers and clusters of workstations. Grid applications cross administrative domains, causing problems with security, code portability, and accounting [HK00]. Hardware interfaces and access protocols vary with the manufacturer of the device, requiring middleware to convert one access protocol to another. Projects developing middleware for grid computing include the Globus project [FK97], Legion [GWtLT97], and the Storage Resource Broker [BMRW98].

The second challenge for computational grid developers is performance. Networks between the resources are typically high-latency and low-bandwidth, so the bottleneck for many grid applications is the network. Complicating the problem is the fact that performance and reliability of the various grid resources can change dramatically over short periods of time. The performance barrier is arguably the more challenging of the two and is the primary focus of our research.

1.2 Data-intensive grid applications

An important class of applications for computational grids is the set of data-intensive grid applications. These applications typically require access to large (terabyte to petabyte) remote datasets and have computational requirements that can only be met by high-performance supercomputers or visualization servers. In addition, the datasets are often stored in formats different than what is needed by the application, requiring conversion steps between acquiring the data and performing the computation. Examples of this type of application include seismic processing [OWO98], climate modeling [DIS96, LEG+97], astronomy\(^1\), computational physics [GJM+97], computational biology and chemistry [WCF+99, YFHL96], visualization [FM99, FL99, LJD+99], and others [Old01, OK01].

1.3 Contributions of this dissertation

This dissertation examines issues associated with developing an I/O system, called Armada, designed to improve I/O performance for data-intensive grid applications. The contributions of this dissertation are the following:

- A flexible framework, based on a data-flow programming model, that allows the application programmer and the dataset provider to describe and deploy a network of application-specific and dataset-specific functionality across a computational grid. Chapter 2 describes the design and implementation of this framework.

- An algorithm to restructure a data-flow application graph to improve data flow across a wide-area network, based on programmer- and user-assigned properties that describe the behavior of the nodes within the graph. We present the design, analysis, and implementation of this algorithm in Chapter 3.

\(^1\)Digital Sky Survey (http://www.npaci.edu/Research/DI/digital-sky.html)
A hierarchical graph-partitioning scheme used to place individual application components onto resources of a computational grid. Chapter 4 presents an approach to placement of application components that leverages existing graph-partitioning software.

Performance evaluation. Chapter 5 describes I/O performance of three applications: an application that models a typical remote scientific data analysis, a remote-copy application that allows arbitrary permutations between distributed datasets, and a seismic imaging application. We also describe the design of an application for remote feature extraction and analysis of an fMRI brain-image dataset.

The system described in this dissertation demonstrates that a flexible design along with careful attention to data-flow performance can lead to efficient I/O for grid applications. Nonetheless, Armada is not a fully-featured I/O system. It lacks support for data management, security, accounting, fault tolerance, and other features typically found in a production-ready I/O system. In Chapter 6 we discuss some of these issues, along with a general summary of the Armada system, limitations of our approach, future directions for this research, and conclusions.
Chapter 2

The Armada Framework for Parallel I/O

The Armada framework provides a flexible programming model that allows the application and dataset provider to describe the I/O system as a data-flow graph of distributed application- and dataset-specific components. This chapter describes the design and implementation of this framework.

2.1 Background and Motivation

Our investigation of scientific applications using parallel I/O [OK01] revealed that data-intensive applications benefit from explicit control over policies and functionality related to parallel I/O. Developers with particular interest in high-performance I/O often customize the I/O interface [MMD98, OWO98, SR98, NFK98, TLG98], design application-specific policies for caching and prefetching [RPR99, CDZ*97, Bel88], or enable remote execution of application code to filter, compress/decompress, or reorganize data [DH99, LJD*99, CMA*97, DLY*98, FMH*97, OWO98].

In this section, we discuss the inadequacies of traditional parallel file systems and why, in a computational grid setting, they should “evolve” to adopt a more flexible approach. We explain why an I/O system should allow application control over the interface and functionality of the system and why remote execution of application code should be supported.

2.1.1 Evolution of parallel I/O systems

Traditional operating systems and file systems restrict the management of system resources to the kernel and privileged servers. Applications not trusted by the operating system are limited to the interface and implementation provided by the privileged software. The problem with this scheme is that not all applications have the same demands. For example, the standard page-replacement policies of a UNIX operating system perform poorly for applications with random or non-sequential data accesses [CD94]. The standard network protocol offered by most operating systems is usually inadequate for the high-performance demands of parallel applications [BSP*95]. Still other applications such as databases and real-time systems may require functionality that is not available from standard operating systems. These applications often implement their own policies and functionality at the user level causing their application to compete, rather than cooperate, with the kernel for system resources [SSS95].

The Galley parallel file system [NK97] proposed that the traditional functionality of parallel file systems be separated into two components: a fixed core that is standard on all platforms, encapsulating only primitive abstractions and interfaces, and a set of high-level libraries to provide a variety of abstractions
CHAPTER 2. THE ARMADA FRAMEWORK FOR PARALLEL I/O

Figure 2.1: Our proposed evolution of I/O system structure. Traditional systems depend on a fixed “core” file system that attempts to serve all applications. The Galley Parallel File System reduced the size of the core system, leaving the API and many of the features of the I/O system to an application-selectable library. In the Armada I/O System, we shrink the core further and allow user-selected code to run on compute nodes, I/O nodes, and intermediate network nodes.

and application-programmer interfaces (APIs). The idea was to simplify the parallel file system and allow application libraries to provide a more appropriate interface and functionality to the user application. While this approach is more flexible than the conventional parallel file system, the application still has no control over the policies implemented at the data servers. Armada adds this flexibility by allowing user code to run on processors used by the client application (compute nodes), on processors used by the data servers (I/O nodes), or on intermediate processors in the network (network nodes). The system decides where the user code should run based on resource availability, performance, and user preference. Figure 2.1 illustrates the evolution from conventional parallel file systems to Galley and then to Armada.

The Armada core I/O system is very simple: there is no caching, prefetching, or remote access. A server executing on top of the existing operating system (e.g., UNIX) provides a local interface to open, close, read, and write data and arbitrate among Armada programs competing for processor time, memory, disk access, and network access. In short, the core system focuses on the shared aspects of the I/O system. Thus, Armada application libraries implement nearly all features of the system, including the API, caching, prefetching, data distribution, synchronization and consistency.

Although we expect most applications to choose from pre-defined libraries, we encourage the use of application-specific libraries designed by application programmers, generated automatically by compilers, or generated at run time [PAB+95]. We refer to all of these choices as “application-selected code,” and to all user code running on the I/O nodes as “application libraries.”

2.1.2 The need for flexibility

As applications and computing environments become more complex, the need for I/O system flexibility becomes increasingly important. To improve individual application performance, the I/O system should support functionality that enables efficient parallel access to applications with a variety of application needs and access patterns. In particular, the data-access interface and the file-system policies like caching, prefetching, and data distribution should be tailored to fit the access patterns of the application.
Application control of the interface

Our motivation for application control of the interface comes from previous research efforts showing that scientific applications clearly benefit from using an interface that enables advanced parallel I/O techniques such as collective I/O, prefetching, and data sieving [MMD98, OWO98, SR98, NFK98, TLG98]. The problem, however, is that it is difficult to provide a single interface that meets the needs of every application. Attempts to provide such an interface lead to a bulky system that cannot adapt to the needs of new applications. In addition, many application developers would rather not deal with the low-level intricacies of how to best access the data, but would rather access data in a way that makes the most sense to the application. For example, computational chemistry applications may prefer to use an I/O interface designed specifically for their application domain [NFK98, FN96]. In some cases, the same data might be used by several applications, each wanting to use a different interface to access the data. For example, a sorting application wants a Parallel Disk Model [VS94a, VS94b] view of the data, but a computational fluid dynamics code wants a 3D matrix view of the data. The system needs to provide application libraries with the flexibility to define an interface that can both provide a meaningful high-level interface to the application and exploit the full capabilities of the available resources.

Application control of system policies

In addition to providing an application-specific interface, the application library should decide which optimization policies best fit the access patterns for an application. For example, prefetching and caching policies should be tailored to match the application’s access patterns to reduce latency and avoid unnecessary data requests [GP91, KE01, PGG+01], and data-distribution policies should match the application’s access patterns to optimize parallel access to distributed disks [CK93, WGWR93].

2.1.3 The need for remote application code

There are many reasons to support remote execution of application-selected code in an I/O system for grid computing. The application can apply application-specific caching and prefetching policies at the data servers. Developers can implement mechanisms like disk-directed I/O [Kot97] using application-specific data-distribution information. Applications can distribute file data to compute nodes using a data-dependent mapping function, for example, in applications with a data-dependent decomposition of unstructured data [Kot95]. I/O nodes can filter data in an application-specific way, passing only the necessary data on to the compute node, saving network bandwidth and compute-node memory [Kot95, BP88, FM99, KCFS99]. I/O nodes can exchange blocks without passing the data through compute nodes, for example, to rearrange blocks between disks during a copy or permutation operation. Format conversion, compression, and decompression also are possible. In short, there are many ways to optimize memory and disk activity, reduce network latency, and reduce disk and network traffic, by moving what is essentially application code outside of the compute nodes.

2.2 The Armada Framework for Parallel I/O

The Armada framework provides a flexible programming model that allows the application programmer and the dataset provider to deploy a network of application-specific and dataset-specific functionality across the grid. Applications access remote data by sending data requests through a graph of distributed application
objects called ships.\footnote{1} Data is pushed toward the client for reads and pushed toward the servers for writes. (Currently, a graph can be used only for reading or writing, but not both. Since workload studies [NKP+96] indicate the vast majority of scientific applications do not simultaneously read and write the same file, this restriction is not a serious limitation.)

While it is possible for an application to construct an entire graph from scratch, we believe a typical graph will consist of two distinct portions: a portion that describes the layout of the data (usually from a data provider), and an application-specific portion that describes the interface and preprocessing required by the application. Figure 2.2, for example, shows a graph for an application with read access to a replicated and distributed dataset. The portion from the data provider describes the layout of two replicas, each stored as a distributed file. The application prepends a graph to the data provider’s portion that includes a filter and three interface ships (one for each client processor).

The Armada framework is specifically designed to provide a latency-tolerant way for data-intensive grid applications to access to large, distributed datasets. Armada is not a parallel file system, nor does the system itself store any data. The set of data segments that make up a dataset are each stored in conventional data servers, as files, as databases, or the like. Indeed, a dataset provider can build a graph on top of legacy files and databases, present a collection of similar datasets through a standard interface, or provide transparent access to derived virtual datasets—either cached or calculated as needed. The graph encodes most functionality provided by the I/O system, including the programmer’s interface, data layout, caching and prefetching policies, and interfaces to heterogeneous data servers. We discuss details of how to encode such functionality in Section 2.5. A graph, once deployed on a computational grid, appears to the application as an object providing access to a specific type of data through a high-level interface.

## 2.3 Armada ships

An Armada graph is a set of interconnected ships that form two directed acyclic graphs (DAGs): one for the flow of requests (request graph) and one for the flow of data (data graph). As described above, ships provide nearly all essential I/O functionality except storage. From an operational perspective, requests flow (in a pipelined manner) from the client processors, through the ships in the request graph, to the data servers. Data flows back to the client (through the ships in the data graph) for reads, and flows toward the data servers for writes. In a typical read-only scenario, an application would process only portions of the data at a time, thus alternating between sending requests and getting data back. From a theoretical perspective, we treat a ship as if the application’s entire sequence of requests arrives at once, and we reason about a ship’s effect on the sequence of requests or on the sequence of data passing through.

In this context, a sequence is an ordered collection of elements (either requests or data) written $X = (x_1, x_2, \ldots, x_k)$. Since a single ship may process and generate multiple sequences, we use the notation $S^n = \{S_1, S_2, \ldots, S_n\}$ to describe the set of sequences $S_1, S_2, \ldots, S_n$. The processing of sequences of requests or data as it passes through a ship is called a mapping, which has the following notation:

- $S^n \xrightarrow{A} T^m$ denotes the request mapping of ship $A$ from $n$ sequences in $S^n$ (input) to $m$ sequences in $T^m$ (output).
- $D^n \xrightarrow{A} E^m$ denotes ship $A$’s data mapping (for writes) from $n$ sequences in $D^n$ to $m$ sequences in $E^m$.

\footnote{The terms “Armada” and “ship” come from an analogy with the 18th century Spanish Armada. A Spanish Armada consisted of a fleet of strategically positioned war ships collectively trying to achieve a single goal.}
Figure 2.2: The figures show the Armada graph for an application with read-only access to a replicated and distributed dataset. (a) shows the request flow, (b) shows the data flow. $M$ is a merger, $filt$ is a filter, $rep$ is a replica manager, $dist$ is a distribution manager, and $seg$ is a segment storage ship.
CHAPTER 2. THE ARMADA FRAMEWORK FOR PARALLEL I/O

Figure 2.3: Hierarchy of Armada ships.

```
Armada Ships

  Structural
    Distribute (partition, select, copy)
    Merge
    Data-Processing
      Filter (> , < , =)
      Transform (FFT, unit conversion)
      Reduce (min, max, sum)
      Permute (sort, transpose)
    Optimization
      Cache
      Prefetch
    Interface
      Client (Matrix, Line, String, stdio)
      Storage (File, Query)
```

For reading, $D^n \rightarrow_d E^m$ denotes the data mapping from $m$ sequences in $E^m$ to $n$ sequences in $D^n$.

2.3.1 Types of ships

Our framework includes a rich set of ship classes (shown in Figure 2.3) divided into two primary categories: structural and non-structural.

**Structural** ships (illustrated in Figure 2.4) allow one-to-many and many-to-one connections in a graph. A distribute ship maps the elements of a single sequence to $k$ sequences. It has request mapping $R^1 \rightarrow \mathcal{A} S^k$ and data mapping $D^1 \rightarrow \mathcal{A} E^k$ for writing or $D^1 \rightarrow \mathcal{A} E^k$ for reading. One could imagine using distribute ships in several ways. For example, a distribute ship could read from a dataset partitioned into $k$ pieces. Another example is a select ship used to read from distributed replicas of a dataset. A select ship could choose a single path to forward all requests, perhaps based on network conditions, or it could partition the requests and send each partition to a separate replica (thus providing parallel access). In yet another instance, a copy ship could forward write requests to all output paths so that $R_i = S_i$ and $D_i = E_i$ for all $i = 1, 2, \ldots, k$. This ship could be used to update replicated datasets.

A merge ship interleaves $k$ sequences. It has a request mapping $R^k \rightarrow \mathcal{A} S^1$ and data mappings $D^k \rightarrow \mathcal{A} E^1$ for reading.
2.3. ARMADA SHIPS

Figure 2.5: A non-structural ship processes data or requests as it travels through the graph. They have a single input and a single output. In the example, $A$ has the request mapping function $R_A \rightarrow S$.

Figure 2.5: A non-structural ship processes data or requests as it travels through the graph. They have a single input and a single output. In the example, $A$ has the request mapping function $R_A \rightarrow S$.

for writing or $D^{k_A} E^1$ for reading. In cases where the ordering of requests is not important, we expect a merge ship to arbitrarily interleave sequences; however, some applications may choose to arrange incoming elements in a particular order. For example, an application performing a MERGE sort would take $k$ sorted data sequences as input, and it would output elements (based on value) to a single sorted data sequence. Another example is an application receiving write requests for potentially overlapping regions. A POSIX-compliant application would have to order requests based on a time stamp embedded in the request [CF96, CPdM+93].

Non-structural ships (illustrated in Figure 2.5) process and generate single sequences of requests and data. A non-structural ship $A$ has request mapping $R^1 A \rightarrow S^1$ and data mapping $D^{1-A} E^1$ for writing or $D^{1} A \rightarrow E^1$ for reading. Figure 2.3 shows three types of non-structural ships: data-processing ships, optimizing ships, and interface ships.

A data-processing ship manipulates data elements, either individually, or in groups, as they pass through the ship. Data-processing ships are likely to be useful for “on-the-fly” preprocessing in scientific applications. Our hierarchy from Figure 2.3 identifies four types of data-processing ships. A filter ship outputs a subsequence of its input; for example, to select interesting observations from a spatial dataset. A transform ship changes the content of individual data elements; for example, a Fast Fourier Transform (FFT) ship transforms complex data from time values to frequency values. A reduction ship applies a function to a collection of elements and returns a single result; for example, to sum all of the elements. A permute ship rearranges the elements in a collection; for example, to sort or transpose a dataset.

Optimization ships improve I/O performance through latency-reduction techniques like caching and prefetching.

The interface ships typically form the right and left end points of the Armada graph. On the far left, client-interface ships convert an application’s method calls to a set of data requests into the rest of the armada ships. We expect library programmers to develop client-interface ships that match the semantics of a particular class of applications, such as computational chemistry applications [NFK98], out-of-core data-parallel programming [CC98], or a POSIX interface for access by legacy software. On the far right, storage-interface ships process Armada requests and access low-level data servers to either load or store data based on the request. They are essentially “drivers” for the many available storage systems, for example, a file ship to store a data segment in a UNIX file, or a database ship that “queries” data from a relational database.

2.3.2 Properties of ships

Encoded in a ship’s description are programmer-assigned properties used as directives by the restructuring and placement algorithms. These properties provide a simple way for the programmer to describe the capabilities and expected behavior of the ship once deployed on the grid.
Two important properties are request and data equivalence. A ship that is request-equivalent produces sequences of requests that are equivalent to the input. Similarly, a data-equivalent ship produces sequences of data that are equivalent to its input. We declare two sequences $S^n$ and $T^m$ to be equivalent (written $S^n \equiv T^m$) if $T^m$ is a permutation of $S^n$, or if $T^m$ is a set of sequences that partition $S^n$. For example,

$$\{1, 2, 3, 4, 5\} \equiv \{2, 3, 5, 1, 4\},$$
$$\{1, 2, 3, 4, 5\} \equiv \{2, 3\}, \{1, 4, 5\},$$
$$\{1, 2, 3, 4, 5\} \equiv \{2, 3\}, \{1, 5, 4\}.$$

In other words, order does not matter. We chose to define equivalence in this manner because enforcing a strict ordering involves synchronization that the application may not require. Note also that the equivalence relation is transitive, symmetric, and reflexive.

We expect most structural ships to be both request and data-equivalent, because although they may interleave sequences (as in a merge ship) or partition sequences (as in a distribution ship), the requests or data being operated on do not actually change. For example, a request-equivalent distribution ship has request mapping $S^n \xrightarrow{A} T^m$, where each $T_1, T_2, \ldots, T_m$ is a subsequence of $S^n$. A request-equivalent merge ship has a request mapping $S^n \xrightarrow{A} T^1$, where each $S_1, S_2, \ldots, S_n$ is a subsequence of $T_1$.

Another important property describes the expected behavior with respect to data flow. A ship with the data-reducer property reduces the amount of data flowing back to the client for reads or toward the data servers for writes. For example, a filter is a ship with the data-reducer property. Inversely, a ship with the data-increaser property increases the size of the data as it flows through the ship. A ship that caches data is a data increaser for reads and a data reducer for writes.

The parallelizable property identifies ships that can transform into a collection of ships that operate on subsequences of requests and data in parallel. To have the parallelizable property, the parallelized version must produce output equivalent to the original graph. Ships become parallelized by trading positions with a structural ship (Section 3.2.1 describes when such a “swap” is legal). A ship that is allowed to parallelize...
toward the clients is left-parallelizable. In this case, the parallelized version includes a new ship on every input path of the merge ship. A right-parallelizable ship moves to the right of an adjacent distribution ship by placing a new ship on every output path of the distribution ship. There are two types of parallelizable ships (illustrated in Figure 2.6) supported by Armada: replicatable and recursive.

A ship is replicatable if each parallelized ship is identical to the original. Replicatable ships operate on requests and data objects independently. For example, a filter that discards integer data with a negative value, or a distribution ship with a mapping function that operates on individual requests.

A ship with the recursive property may “split” into a manager ship and a collection of worker ships that are identical to the original ship. Data-reduction ships that calculate a sum, min, or max are recursive. Merge sort is also a recursive operation.

2.4 Armada graphs

We use a series-parallel tree (SP-tree) to describe the composition of an Armada graph. An SP-tree derives from a series-parallel directed acyclic graph (SP-DAG), which is recursively defined as follows [VTL82]:

1. A DAG with a single vertex and no edges is an SP-DAG.

2. If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are two SP-DAGs, so is the DAG formed by one of the following operations:

   a) Parallel composition: $G_p = (V_1 \cup V_2, E_1 \cup E_2)$

   b) Series composition: $G_s = (V_1 \cup V_2, E_1 \cup E_2 \cup (N_1 \times R_2))$, where $N_1$ is the set of sinks of $G_1$ and $R_2$ the set of sources of $G_2$.

We can represent the composition of an SP-DAG as a series-parallel tree, where s-nodes represent a series composition, p-nodes represent a parallel composition, and leaves are the vertices in the original graph.

A contracted SP-tree (CSP-tree) is one in which p-nodes only have s-nodes and leaves as children, and s-nodes only have p-nodes and leaves as children. It is easy to show that for each SP-tree, there exists a unique CSP-tree, composed by contracting children with the same type as the parent [BdF96].

We chose to use a CSP-tree to describe the composition of an Armada graph because a CSP-tree is syntactically easy to describe (we use XML), the SP-tree structure is easy to manipulate internally, and (as we discuss in Chapter 3) we understand how to restructure SP-trees in a way that does not corrupt the integrity of the data flow through an Armada graph.

Figure 2.7-a shows an annotated version of the Armada graph from Figure 2.2 on page 9, illustrating nodes that have either series (labeled S) or parallel (labeled P) connections. Figure 2.7-b shows the SP-tree representation for the graph in Figure 2.7-a. Armada ships form the vertices of the graph in (a) and the leaves of the SP-tree in (b).

2.5 Implementation

In this section, we discuss the implementation of Armada. A quick overview: Armada ships are Java objects instantiated on hosts in the grid. Each host has a persistent server, called a harbor, that provides an execution environment for Armada ships.
Figure 2.7: An application graph and its SP-tree representation. (a) shows an annotated Armada graph, illustrating Armada ships that have series (labeled $S$) or parallel (labeled $P$) connections. Figure 2.7-b shows the SP-tree representation. Armada ships form the vertices of the graph in (a) and the leaves of the SP-tree in (b).
2.5. IMPLEMENTATION

We chose Java for several reasons: it provides a “sandbox” [WLAG93] for executing untrusted client code, it is reasonably efficient now that just-in-time compilers are available, it is increasingly popular among HPC programmers, it has convenient mechanisms for remote execution and communication (RMI), and it can interface to application code in other languages through the Java Native Interface (JNI). Only ships and harbors need to be written in Java; client code could be in C, C++, or any other language that interfaces with Java.

In this section, we describe the ship and harbor implementations, and describe three interfaces (Java, C++, and XML) for constructing Armada graphs.

2.5.1 Ships

Ships are the foundation of the Armada framework. The abstract Ship class, the base class for all other ships, provides mechanisms that manage connections between ships, and provide methods that allow the communication of requests and data along those connections. An application or library developer extends the Ship class to implement application-specific functionality.

Figure 2.8 shows the architecture of an Armada ship. The architecture consists of four layers: a core system that represents the underlying operating system, a network-protocol layer that provides mechanisms for communicating with distributed objects, a basic-functionality layer that manages request and data connections, and an application-specific layer.

Network Protocols

The network-protocol layer of the architecture provides a level of abstraction that allows the application developer to concentrate on the most important functions of the application-specific ships, namely processing requests and data. The developer of the Ship class can change the protocol without requiring changes from the extended classes. This feature is helpful when testing a variety of network protocols.

In the current implementation, ships communicate using a combination of RMI and TCP sockets. They use RMI for administrative tasks, such as establishing a connection, sending and receiving information about
public interface RemoteShip extends java.rmi.Remote {

    public void controlConnect(RemoteShip dest, int index);
    public void dataConnect(RemoteShip dest, int index);
    public ObjectgetID();
    public StringgetHost();
    public intgetRequestServerPort();
    public intgetDataServerPort();
    public void close();
}

Figure 2.9: The RemoteShip interface.

the ship, or shutting down a ship. We discuss the remote interface implemented by the Ship class in the
next subsection. Although RMI is convenient, it has notoriously poor performance when used for large and
frequent network transfers. For that reason, we chose to use persistent TCP sockets to communicate requests
and data between ships. Our current implementation uses TCP sockets for request and data connections
located inside the same harbor, although there are faster means for communication between processes on
the same host (e.g., shared memory). A potential optimization to the current network protocol would be to
exploit these faster communication methods for Armada ships connected inside the same harbor.

Basic functionality

The Ship class implements most of the basic-functionality layer of the architecture. The responsibilities
for this layer include implementing the remote RMI methods for administrative tasks, managing connec-
tions to adjacent ships in the graph, and implementing methods for sending data and requests along those
connections.

Figure 2.9 shows the RemoteShip interface implemented by the Ship class that allows administration
of remote Armada ships through Java RMI. A ship establishes a TCP socket connection with a specified
RemoteShip by calling thecontrolConnect or dataConnect method. The controlConnect
and dataConnect methods assign the new socket connection to the output path identified by the index
parameter. Establishing a new request or data connection involves calling the remote methods getHost
and getRequestServerPort (or getDataServerPort) of the destination ship to get the host and
port of the server socket for the remote ship. The getID method returns a unique identifier assigned to the
ship, a java.rmi.dgc.VMID object. A VMID combines a java.rmi.server.UID (a guaranteed
unique object identifier within a single virtual machine) with the host address to try to get a system-wide
unique identifier (we assume at most one VM per host). The close method shuts down the ship by closing active connections and removing the ship from the RMI runtime environment.

The Ship class manages control connections (for requests) and data connections (for data) separately. In each case, it manages existing connections with threads (one per connection) that either read from the TCP socket (for input connections) or write to the TCP socket (for output). Each thread contains a fixed-length queue of pending requests or data. The Ship class uses additional processing threads (one for requests and one for data) that remove elements from input connection queues and call the application-implemented methods for processing requests and data (discussed in the next subsection).

As a debugging feature, we added a separate thread that monitors activity of the various queues. At designated intervals (the default is every second), the monitor checks the size of each queue. When the ship closes, we output (to a log file) the average size of each of the queues. Each queue also outputs the idle time waiting because the queue was empty, the idle time waiting because the queue was full, and the amount of time spent processing elements. As we demonstrate in Section 5.4.4, these statistics provide a nice way to quickly find bottlenecks in an Armada graph.

Two extensions of the Ship class, the StructuralShip and NonStructuralShip classes, implement protected methods (called by subclasses) for sending data and requests to adjacent ships in the Armada graph. The method sendRequests sends an array of requests along one of the output control paths (identified by an index parameter) by appending the array of requests to the appropriate output queue. Similarly, sendData sends an array of data elements along one of the output data paths by appending the array to the appropriate output queue. Non-structural ships do not use the index parameter since they have only one path for data and one path for requests.

Application extensions

An application or library developer extends the StructuralShip or the NonStructuralShip class to implement application-specific functionality. In both cases, the application implements the processRequests method to process an array of incoming requests, and the processData method to process an array of incoming data elements. For extensions of the StructuralShip, these methods include an index parameter that identifies the source. Inside the methods, the application performs the application-specific processing and then sends the resulting array of requests or data to the next ship in the graph through calls to the methods sendRequests and sendData implemented by the StructuralShip and NonStructuralShip classes.

The developer labels an Armada ship with properties by implementing a null interface for that property. For example, a ship with the data-reducer property implements the DataReducer interface. Similar interfaces exist for data-increaser, left-replicatable, right-replicatable, left-recursive, and right-recursive. The restructuring and placement algorithms use the Java Reflection API to identify ships that implement a particular property.

2.5.2 Graphs

As we mention in Section 2.4, we use a series-parallel tree to describe the composition of an Armada graph. In this section we describe the XML representation used to describe an Armada graph using an SP-Tree, and we describe how to construct and deploy an Armada graph using either Java or C++.
CHAPTER 2. THE ARMADA FRAMEWORK FOR PARALLEL I/O

<?xml version='1.0' encoding='UTF-8'?>
<!ELEMENT Arg EMPTY>
<!ATTLIST Arg
    value CDATA #REQUIRED
    name CDATA #REQUIRED
>
<!ELEMENT Class (Arg)*>(
<!ATTLIST Class
    name CDATA #IMPLIED
>
<!ELEMENT Leaf (Class)>(
<!ATTLIST Leaf
    domain CDATA #IMPLIED
    host CDATA #IMPLIED
    id CDATA #IMPLIED
>
<!ELEMENT Parallel ((Leaf|Series),(Leaf|Series)+)>
<!ELEMENT Series ((Leaf|Parallel),(Leaf|Parallel)+)>
<!ELEMENT SPTree (Leaf|Series)>
<!ATTLIST SPTree
    type (READ|WRITE) #REQUIRED
>
Figure 2.10: Document Type Description for Armada graphs (stored as SPTree.dtd).

XML representation of an Armada graph

We use the Extensible Markup Language (XML) to encode descriptions of Armada graphs for persistent storage or for transmission across the network. We chose XML because XML's hierarchical structure fits our SP tree needs well and there are existing class libraries (e.g., the Simple API for XML Parsing (SAX)) that provide mechanisms for creating and manipulating XML documents.

The structure of an XML document for an Armada graph follows the Document Type Description (DTD) in Figure 2.10. The primary components are the Leaf, which requires a single Class description; the Parallel, composed of a Leaf or Series followed by one or more Leaf or Series descriptions; the Series, composed of a Leaf or Parallel followed by one or more Leaf or Parallel descriptions; and the SPTree description, which has a single child (either a Leaf or Series).

Figure 2.11 shows XML for an Armada graph that describes a dataset striped to two separate files. In this
2.5. IMPLEMENTATION

Armada graph

SP-tree

Figure 2.11: XML document for a simple Armada graph. The top two figures illustrate the Armada graph and SP-tree described by the XML document. A file in the illustration corresponds with the BlockReaderShip in the XML. The distribution ship dist corresponds to the StripeReaderShip in the XML.
example, each storage interface ship (BlockReaderShip) has an assigned host and domain, along with an argument that provides a path to a local UNIX file. The structural ship StripeReaderShip has a *stride* argument that tells StripeReaderShip how to distribute the blocks to the files. StripeReaderShip does not have an assigned placement until after graph restructuring (described in Chapter 3) and ship placement (described in Chapter 4).

Representing an Armada graph in Java

The methods for constructing, restructuring, and deploying, and, in general, managing an Armada graph are all implemented in the Java classes associated with SP-trees. In particular there are four classes (illustrated in Figure 2.12) that extend the abstract Node class for an SP-tree node: the SPTree, the ParallelNode, the SeriesNode, and the Leaf. The SPTree class contains a single pointer to the Node (either a SeriesNode or a Leaf) that represents the root of the tree. The ParallelNode and SeriesNode classes each contain a linked list that points to child nodes.

An application constructs an SP-tree in one of two ways: by loading an XML description of an Armada, either from a file or transmitted across the network; or by constructing an SP-tree manually.

The XML interface for loading SP-trees uses the SAX parser for Java to construct an SP-tree. The SAX parser uses an event-based scheme in which the parser notifies the application handler (through callback functions) when the parser finds a particular symbol.

The manual construction of an SP-tree progresses in a bottom-up fashion by first constructing the necessary Leaf objects that represent Armada ships. The Leaf class constructor requires the class name of the associated Armada ship and a list of arguments (name, value pairs) to use when deploying the ship. Next, the application constructs parallel or series nodes by successively calling the appendChild or prependChild methods.

Once constructed, the application restructures the SP-tree by calling the restruct method of the SPTree class. The restruct method recursively restructures the SP-tree following the algorithm described in Chapter 3.

The final step is to deploy the Armada graph by calling the deploy method of the SPTree class. The first thing the deploy method does is decide a placement for each of the Armada ships, based on the hierarchical graph partitioning scheme described in Chapter 4. We then recursively traverse the SP-tree and construct the Armada graph in a bottom-up fashion, first installing each Armada ship on its assigned host, and then connecting the ships by calling RMI methods of associated ships.
public interface Harbor extends java.rmi.Remote {

    /** Create and install a new Armada Ship */
    public RemoteShip install(String classname, String args[])
            throws RemoteException;

    /** Register an existing Armada Ship */
    public RemoteShip register(RemoteShip ship)
            throws RemoteException;

    /** Find a registered Armada Ship */
    public RemoteShip find(Object id)
            throws RemoteException;

    public void uninstall(Object id) throws RemoteException;

    public void unregister(Object id) throws RemoteException;

    public void ping() throws RemoteException;

    public void close() throws RemoteException;
}

Figure 2.13: The Harbor interface.

Representing an Armada graph in C++

Since we expect many scientific applications to use C or C++, we implemented the identical functionality of the Java classes for SP-trees in a set of C++ classes. The current implementation of these classes wraps the Java class objects into the C++ class objects using the Java Native Interface.

2.5.3 Harbors

Each available processor has a persistent server to host Armada ships, called a harbor. The Harbor class provides methods (executed through RMI) that allow an application to connect to an existing ship, upload and install a new ship, or to register a local ship on the harbor.

Figure 2.13 shows the Harbor interface. The install method constructs an object for the given class name (along with the provided arguments) and returns a remote reference (via the RemoteShip interface) to the newly installed Armada ship. Arguments for the install method represent a concatenation of the name/value pairs represented in the XML. For example, the argument “file=/tmp/test.dat” corresponds to the name/value pair name=“file”, value=“/tmp/test.dat” in the XML. The register method allows a previously instantiated object to register itself with the local harbor. This method is convenient for applications that construct a client-interface ship as part of the application interface. The find method returns a remote reference to a previously installed (or registered) Armada ship.
Implementations of the Harbor interface run on top of the host’s operating system to provide a secure execution environment that allows ships to access the CPU, memory, network, and storage resources of the host machine. To provide the necessary services for a prototype implementation of the Armada system, we designed the SimpleHarbor class, an implementation of the Harbor interface that manages a set of registered Armada ships as a hashtable, using the ship’s unique identifier as the key. The implementation of the install method uses the Java Reflection API to find the definition of the provided class name and the class constructor that takes an array of String objects as input. It then creates a new instance of the Ship class, registers the ship by calling the register method, and returns a reference of the ship to the caller. The register method simply adds the new ship to the underlying hashtable.

The SimpleHarbor class provides no security features beyond what is provided by the Java runtime environment. Java includes support for a wide range of security policies to control domain-level preferences (typically through a policy file) that the administrator can apply through the command line when starting a SimpleHarbor. In most cases, Java interjects a permission check between the user code and the system resource before granting access to a resource (such as a file or a socket). While this level of security is fine for a prototype, it has some flaws that need to be addressed. The primary issue is what happens after the system grants access to the resource. Java has no way to revoke access. We discuss a more secure implementation of the Harbor interface in future work section (Section 6.3 on page 98).

2.6 Related Work

Inspiration for the Armada framework came from a large body of work including stackable file systems [HP94, KS97, Ros92, SW93, ZBS99], active storage [KPH98, RFGN01, WCV02], streaming computation [Mes99, PS00, VV96], and component-based systems [APGG00, BFK00, HS99, RMR00]. This section discusses the projects, systems, and ideas that had the largest influence over the design.

2.6.1 The data-flow computation model

The Armada graph abstraction is based on the data-flow computation model proposed by Dennis et al. in 1974 [Den74]. In the Dennis data-flow graph, applications exist as a graph of “actors” that process “tokens” traveling along arcs of the graph. At the time, the data-flow model provided an efficient solution to two fundamental problems with von Neumann computers: memory latency and synchronization. These problems are critical performance issues when transferring data across geographically distributed computation resources. The pipeline parallelism inherent in the data-flow model allows for a latency-tolerant solution to wide-area I/O.

2.6.2 Parallel I/O systems

Our framework gets much of its design features from the stackable file systems and component-based approaches, which provide flexibility by representing open files as a collection of application-specific “stackable” building blocks. This flexibility enables the system to provide only the necessary functionality to suit the application, allowing good performance for a wide variety of applications. For example, the Hurricane file system [KS97] (designed for clusters of workstations) provides the full bandwidth of parallel disks to some parallel applications. Another stackable file system, WrapFS [ZBS99], uses a file system template in which kernel-level modules with a common vnode interface are stacked to form complex file systems. They demonstrate flexibility by implementing four separate file systems using WrapFS. In a later paper [ZN00],
Zadok et al. describe FiST, a language used to describe a stackable file system. The FiST compiler generates the necessary vnode modules used by the system. The Armada framework extends the stackable file system concept to a grid environment and uses a data-flow computation model for latency tolerance. A data provider, using Armada, describes complex arrangements and distributions of data with a graph of mobile objects. The application then layers application-specific processing on top of the graph from the data provider.

Another file system that had an influence on the Armada design was the Galley parallel file system [NK97], also developed at Dartmouth College. Galley separates the functionality of the system into two components: a fixed core that is standard on all platforms, and a set of high-level libraries. The core system provides the application-library with a low-level interface for parallel I/O. Application libraries provide an appropriate interface to the application by layering their interface on top of the core system. The Armada approach is the same, except we reduce the core system even further to allow the application library to control the behavior of the system even on the I/O nodes.

TPIE [Ven94, VV96] is a system designed for the application of I/O-optimal algorithms on a variety of parallel architectures. TPIE is a set of C++ templates and class libraries to process streams of data. TPIE library functions perform fundamental operations such as distribution, merging, filtering, and permutation. The application programmer inserts custom functionality by extending the library classes, implementing call-back functions that operate on a set of input and output streams, calling the application’s function once for each set of input records. Even though (conceptually) there is a single thread of control, the system may execute some operations in parallel. For example, a filter applied to records of a striped file may execute on each stripe of the data file independently. TPIE is an example of an efficient data-flow system with user control over part of the functionality within the system.

Another interesting system that uses a data-flow model is the Parallel Storage-and-Processing Server (PS²) [Mes99], from École Polytechnique Fédérale de Lausanne. PS² uses the Computer-Aided Parallelization tool (CAP) to express the parallel behavior of the I/O intensive applications at a high-level. The CAP system constructs a data-flow computation graph with “actors” as nodes of the graph. The actors are computational units that provide application-specific functionality. For I/O-intensive applications, the actors provide application-specific data distribution, prefetching, or filtering that potentially execute near the data storage devices.

In both PS² and TPIE, as with many other parallel-processing systems, the developer typically uses the same programming model for the entire application. A data-flow programming model works well for applications where the communication between parallel components is well defined, but many scientific applications have complex communication patterns that do not perform well in a distributed data-flow environment. In contrast, Armada treats I/O as a separate data-flow-based application that provides a data-access service to the client application. The client, which may exist on a supercomputer or a tightly-connected cluster, could use lower-level parallel programming models (e.g., MPI or HPFortran) more fitting to the performance requirements of the application. This approach also allows easier integration of legacy applications to datasets on the grid, since only an interface between the Armada graph and the client is needed.

Another difference is the granularity of computation. In PS² the degree to which the data is divided and processed is decided by the compiler. In TPIE, computation occurs on data streams that match the number of data sources. Armada, like TPIE, parallelizes computation to match the data sources, but it also creates data streams (with embedded computation) to match the number of client processors, providing an end-to-end parallel data-flow graph.
2.6.3 Support for remote execution of application code

As mentioned in Section 2.1, many projects support remote execution of application code to filter, compress/decompress, or reorganize data to improve I/O access to the application. Here, we discuss a few of these systems.

Active storage systems [KPH98, RFGN01, WCV02] require a processor associated with each storage component. These systems treat computation as a “side-effect” of I/O, processing data at the disk. A key problem with active-storage systems is resource sharing. Multiple applications accessing the same active-storage disk may overload the disk’s processor, slowing down access for all. In such situations, processing on a nearby processor may provide an effective load-balancing solution. Armada allows for such a solution.

DataCutter\(^2\) [BFK00], developed at the University of Maryland, is middleware used to explore and analyze scientific datasets stored on archival storage systems across a wide-area network. DataCutter provides a query-based interface with support for accessing subsets of datasets and for performing user-defined transformations of large datasets in archival storage. The processing structure is composed of a set of processes called “filters” that typically execute close to the data source. While the processing structure and data filtering ideas of the DataCutter are similar to ours, Armada has significant differences. Armada provides more flexibility in the interface, and it has support for efficient placement of file structure objects using algorithms that consider the underlying structure of the system and the performance requirements of the application.

A paper by Franke and Magee [FM99] describes efforts to improve the performance of data visualization applications that use remote data generators (disks or running application) and data consumers (visualization stations) for visualization applications. They deal with network bottlenecks by using a distributed and redundant data cache to hold intermediate data between the data generator and the data consumer. They also reduce network traffic by applying data filters to the data at the distributed cache processors. The main argument is that since the data must be filtered before it is visualized, it makes more sense to perform the filter at the data cache so the computation can be distributed and to reduce the amount of data that needs to be transferred across the network.

A similar paper by Leigh et al. [LJD99] discusses methods for interactively reducing the size of datasets so they can be viewed at remote sites. The methodology employs feature detection, partitioning, summarization, and decimation at or near the data source. The reduced image is then sent to the tele-immersion server and then to various tele-immersion clients (viewers).

Mocha (Middleware based On a Code SHipping Architecture) [RMR00] is a database middleware system designed to connect hundreds of heterogeneous data sources over a wide area network. Mocha supports dynamic code shipping to the remote data source. The remote code objects include operators for data filters that can significantly reduce the amount of data sent through the network. The relocatable objects exist as Java classes stored in one or more repositories and are migrated on demand by the MOCHA system. MOCHA is written entirely in Java and has database support for Oracle, Informix, file servers, and XML repositories.

The major distinction between Armada and the systems mentioned above is the way we decide placement. All of these systems place filtering code on the data server to reduce the amount of network traffic across the wide-area network, but Armada is the only system that pays special attention to where (even within each domain) the filtering codes should run. In some cases, for example, it is better to place a filter on a processor near the I/O node, to reduce the computational burden on the I/O node. Also, as we show in the next chapter, restructuring an Armada graph can often increase the placement options for filtering operators and remove bottlenecks in the data flow graph.

2.6.4 Other related research efforts and resources

There are many other examples of applications that have added support for remote application code to improve I/O. These applications use ad-hoc methods, specific to their application domain, designed to work around the lack of support for remote code execution by the system. Such applications exist in ocean and climate modeling codes [DH99], satellite image databases [CMA+97], virtual microscopes [FMH97], and seismic processing [OWO98]. For a more complete description of these applications, see [OK01] and [Old01].

2.7 Conclusions

The Armada framework provides a flexible programming model that enables data providers to describe arbitrary distributions of data, and application programmers to describe preprocessing and the layout of the data to compute nodes. The reason why we separate the data-provider’s portion from the application is because the application does not need to know the details of the data distribution scheme (which may arbitrarily change) to efficiently access the data. The restructuring optimizations, discussed in the next chapter, interleave the application and data-provider portions of the graph to allow an efficient placement of application-specific operations and provide end-to-end parallelism between distributed data servers and the parallel application.
Chapter 3

Restructuring an Armada Graph

Although the Armada framework is flexible and convenient for application developers and dataset providers, two issues prevent the efficient mapping of an application graph to a computational grid. First, the connection between the application’s portion of the graph and the data-provider’s portion forms a bottleneck. Second, the configuration of the graph restricts the placement alternatives for the application-supplied code. Recall the simple application from Chapter 2 (illustrated in Figure 3.1-a). If the network between the two replicas is slow, or the network between the replicas and the clients is slow, there is no placement of the filter that allows us to significantly reduce the amount of data transferred over a slow network. Figure 3.1 shows a restructured graph that mixes the application’s portion of the graph and the data-provider’s portion of the graph. The new graph has a parallelized version of the replica-select ship to match the number of clients, and it has a parallelized version of the filter to match the number of data servers. The result provides end-to-end parallelism and allows the filters to be placed near (in terms of connectivity) to the data servers, thus reducing the amount of data transferred over a potentially slow network.

In this chapter, we describe an algorithm to improve performance of data-intensive applications by restructuring an Armada graph to increase parallelism and to arrange processing ships so as to minimize traffic across a wide-area network.

3.1 The Restructure Algorithm

Armada restructures a graph by swapping (exchanging the position of) adjacent ships. It increases parallelism by swapping parallelizable ships with structural ships. It reduces network traffic by moving data-reducing ships toward the data-source and data-increasing ships toward the data destination.

The restructuring algorithm manipulates an Armada graph by restructuring its SP-tree. Recall that an SP-tree node is either a series node, parallel node, or leaf node that corresponds to a ship. The process of restructuring an SP-tree requires two tasks: initializing the tree, and recursively traversing the tree swapping series-connected nodes as needed.

The INITIALIZE procedure (Algorithm 1) verifies that root node $N$ is indeed an SP-tree, compresses the tree to form a CSP-tree (if necessary), and initializes internal nodes (series and parallel nodes) to dirty. The dirty flag identifies nodes that need to be recursively restructured.

The RESTRUCTURE algorithm (Algorithm 2) traverses the CSP-tree in a depth-first manner, revisiting subtrees when necessary. The base case occurs if $N$ is a leaf or if $N$ is marked clean (meaning it has already
3.1. THE RESTRUCTURE ALGORITHM

![Diagram showing original and restructured application graphs]

(a) Original.

(b) Restructured.

**Figure 3.1:** Armada restructures the application graph to increase parallelism and to allow the filters to execute close to the data servers.

**Algorithm 1** \textsc{Initialize}(N)

1: verify $N$ is an SP-tree
2: compress $N$ (as necessary) to make $N$ a CSP-tree
3: mark all internal nodes of $N$ dirty
Algorithm 2 \textsc{Restructure}(N)

1: if (N is clean or a leaf) then [base case]
2: \hspace{1em} return
3: end if
4: if (N is a parallel node) then
5: \hspace{1em} for all child $\in$ children of N do
6: \hspace{2em} \textsc{Restructure}(child)
7: end for
8: else \{N is a series node\}
9: \hspace{1em} New $\leftarrow$ new series node
10: \hspace{1em} for all child $\in$ children of N do [from left to right]
11: \hspace{2em} append child to New
12: \hspace{2em} \textsc{SlideLeft}(child) \{slide child left to proper position\}
13: end for
14: N $\leftarrow$ New \{N is now restructured\}
15: end if
16: mark N clean

been restructured). If N is a parallel node, the algorithm recursively calls \textsc{Restructure} on each child of N. For series nodes, \textsc{Restructure} individually removes and properly aligns each child of N onto a new series node (labeled New in Algorithm 2) by iteratively “sliding” the child (by swapping adjacent nodes where necessary) from right-to-left using the \textsc{SlideLeft} algorithm (Algorithm 3). After properly aligning each child, \textsc{Restructure} sets N to point to the new node and marks N clean. The process of restructuring series-connected nodes (described in the next section) forms the core of our restructuring algorithm.

3.2 Restructuring series-connected nodes

As mentioned above, the \textsc{Restructure} procedure restructures a series node by iteratively “sliding” each child into the correct position of a previously restructured series node using the \textsc{SlideLeft} procedure (shown later in Algorithm 3 on page 31). \textsc{SlideLeft} attempts to “swap” the designated node with the node to its left. The algorithm decides whether or not to swap two series-connected nodes (labeled A and B) based on three conditions: the graph formed by swapping A and B must produce an SP-tree, A and B must be commutative, and swapping A and B must benefit the application. We describe each of these conditions in order.

3.2.1 A swap must produce an SP-tree

Since our algorithm relies heavily on the structure of series-parallel DAGs, our first condition requires that a graph formed by swapping two series-connected ships in a series-parallel DAG is also a series-parallel DAG. Since we use an SP-tree to represent an SP-DAG, we satisfy this condition by only allowing a swap when the configuration of A and B in the SP-tree matches one of four configurations guaranteed to produce another SP-tree, if A and B are swapped.

Table 3.1 lists all possible SP-tree configurations of two series-connected ships and identifies configurations that are allowed to swap. The rows represent the left side (ship A) of a connection and the columns represent the right side (ship B). The three types of ships are non-structural, merge, and distribution. Since a merge ship takes multiple inputs it must either follow a parallel node (labeled p-merge in Table 3.1), or it must be the first ship in a series node (labeled merge); thus, it cannot be on the right side of a series connec-
3.2. RESTRUCTURING SERIES-CONNECTED NODES

Table 3.1: Possible configurations of series-connected ships in an SP-tree. The cells identify whether swapping is allowed (subject to the commutativity requirement).

<table>
<thead>
<tr>
<th>Ship A</th>
<th>non-struct</th>
<th>distrib</th>
<th>distrib-p</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-struct</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>merge</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>p-merge</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

The SLIDELEFT and SLIDERIGHT procedures (Algorithms 3 and 4) implement a swap for the four allowed configurations. Lines 11–12 implement case 1, lines 14–21 implement case 2, lines 23–30 implement case 3, and lines 32–36 implement case 4. SLIDELEFT moves a node from right to left, and SLIDERIGHT moves the node from left to right.

The additional helper routines PARALLELIZELEFT and PARALLELIZELEFT (Algorithms 5 and 6) perform the operations necessary to create parallelized versions of the ships and place them in the correct branch of the SP-tree.

3.2.2 Configurations that do not produce an SP-tree

Table 3.1 shows five configurations that do not produce series-parallel DAGs after a swap. These configurations occur either when a distribution ship on the right is not series-connected to a parallel node, or when a merge ship on the left does not follow a parallel node. We show that these configurations
CHAPTER 3. RESTRUCTURING AN ARMADA GRAPH

Figure 3.2: Four cases for swapping neighboring ships.
3.2. RESTRUCTURING SERIES-CONNECTED NODES

Algorithm 3 SLIDELEFT(B)

1: A ← node to the left of B {else null}
2: if (B is a parallel node) then {A may now be able to slide left}
3:  SLIDELEFT(A)
4:  return
5: end if
6: while (A ≠ null) do {iterate left until we reach the end}
7:  if (COMMUTATIVE(A, B) and BENEFICIAL(A, B)) then
8:    L ← node to the left of A {else null}
9:    R ← node to the right of B {else null}
10:  if (A and B are non-structural ships) then {case 1}
11:     reverse the order of A and B in the series node
12:     A ← L {iterate left}
13:  else if (A is right-parallelizable and non-structural, B is a distribute ship, and R is a parallel node) then {case 2}
14:    remove A from the series
15:    PARALLELIZERIGHT(A, R) {R is a parallel node}
16:    if (A is recursive) then
17:      create a new manager ship M
18:      place M to the left of B
19:      B ← M {see if M needs to move left—it cannot move right}
20:     end if
21:  end if
22:  else if (L is a parallel node, A is a merge ship, and B is left-parallelizable and non-structural) then {case 3}
23:    remove B from the series
24:    PARALLELIZERIGHT(B, L) {L is a parallel node}
25:    if (B is recursive) then
26:      create a new manager ship M
27:      place M to the right of A
28:      SLIDERIGHT(M) {see if M needs to move right—it cannot move left}
29:     end if
30:    return {everything else is already sorted}
31:  else if (L is a parallel node, A is a right-parallelizable merge ship, B is a left-parallelizable distribute ship, and R is a parallel node) then {case 4}
32:    remove A
33:    remove B
34:    PARALLELIZERIGHT(A, R) {R is a parallel node}
35:    PARALLELIZERIGHT(B, L) {L is a parallel node}
36:    return {everything else is already sorted}
37:  else {not a swappable configuration of nodes}
38:    return {everything else is already sorted}
39: end if
40:  else
41:    return {not commutative and beneficial}
42: end if
43: end while
Algorithm 4 SLIDE\textsc{Right}(A)

1: \(B \leftarrow \) node to the right of \(A\) \{else null\}
2: \(\text{if} (A \text{ is a parallel node}) \text{ then} \{B \text{ may now be able to slide right}\}
3: \quad \text{SLIDE\textsc{Right}}(B)
4: \quad \text{return}
5: \text{end if}
6: \text{while} \ (B \neq \text{null}) \text{ do} \text{[iterate right until we reach the end]}
7: \quad \text{if} (\text{COMMUTATIVE}(A, B) \text{ and } \text{BENEFICIAL}(A, B)) \text{ then}
8: \quad \quad L \leftarrow \text{node to the left of } A \text{ \{else null\}}
9: \quad \quad R \leftarrow \text{node to the right of } B \text{ \{else null\}}
10: \quad \text{if} \ (A \text{ and } B \text{ are non-structural ships}) \text{ then} \{\text{case 1}\}
11: \quad \quad \text{reverse the order of } A \text{ and } B \text{ in the series node}
12: \quad \quad B \leftarrow R \text{ \{iterate right\}}
13: \quad \text{else if} \ (A \text{ is right-parallelizable and non-structural, } B \text{ is a distribute ship, and } R \text{ is a parallel node}) \text{ then} \{\text{case 2}\}
14: \quad \quad \text{remove } A \text{ from the series}
15: \quad \quad \text{PARALLELIZE\textsc{Right}}(A, R) \text{ \{R is a parallel node\}}
16: \quad \quad \text{if} \ (A \text{ is recursive}) \text{ then}
17: \quad \quad \quad \text{create a new manager ship } M
18: \quad \quad \quad \text{place } M \text{ to the left of } B
19: \quad \quad \quad \text{SLIDE\textsc{Left}}(M) \text{ \{see if } M \text{ needs to move left—it cannot move right\}}
20: \quad \quad \text{end if}
21: \quad \text{return} \text{ \{everything else is already sorted\}}
22: \quad \text{else if} \ (L \text{ is a parallel node, } A \text{ is a merge ship, and } B \text{ is left-parallelizable and non-structural}) \text{ then} \{\text{case 3}\}
23: \quad \quad \text{remove } B \text{ from the series}
24: \quad \quad \text{PARALLELIZE\textsc{Left}}(B, L) \text{ \{L is a parallel node\}}
25: \quad \quad \text{if} \ (B \text{ is recursive}) \text{ then}
26: \quad \quad \quad \text{create a new manager ship } M
27: \quad \quad \quad \text{place } M \text{ to the right of } A
28: \quad \quad \quad A \leftarrow M \text{ \{see if } M \text{ needs to move right—it cannot move left\}}
29: \quad \quad \text{end if}
30: \quad \text{B \leftarrow node to the right of } A \text{ \{iterate right\}}
31: \quad \text{else if} \ (L \text{ is a parallel node, } A \text{ is a right-parallelizable merge ship, } B \text{ is a left-parallelizable distribute ship, and } R \text{ is a parallel node}) \text{ then} \{\text{case 4}\}
32: \quad \quad \text{remove } A
33: \quad \quad \text{remove } B
34: \quad \quad \text{PARALLELIZE\textsc{Right}}(A, R) \text{ \{R is a parallel node\}}
35: \quad \quad \text{PARALLELIZE\textsc{Left}}(B, L) \text{ \{L is a parallel node\}}
36: \quad \quad \text{return} \text{ \{everything else is already sorted\}}
37: \quad \quad \text{else} \text{ \{not a swappable configuration of series nodes\}}
38: \quad \quad \text{return} \text{ \{everything else is already sorted\}}
39: \quad \text{end if}
40: \quad \text{else}
41: \quad \quad \text{return} \text{ \{not commutative and beneficial\}}
42: \quad \text{end if}
43: \text{end while}
3.2. RESTRUCTURING SERIES-CONNECTED NODES

Algorithm 5 PARALLELIZELEFT($P, B$)

1: $n$ ← the number of children of $P$ {P is a parallel node}
2: parallelize $B$ creating $B_1, B_2, ..., B_n$, each marked as dirty
3: for $(i = 1 \ldots n)$ do
4:   $C_i$ ← $i$th child of $P$
5:   if $(C_i$ is a series node) then
6:     append $B_i$ to $C_i$
7:     SLIDELEFT($B_i$) {slide $B_i$ to the proper position in $C_i$}
8:   else {C_i must be a leaf}
9:     $C_i$ ← a new series node with the original $C_i$ followed by $B_i$
10:    RESTRUCTURE($C_i$) {restructure the new series node}
11:   end if
12: end for

Algorithm 6 PARALLELIZERIGHT($A, P$)

1: $n$ ← the number of children of $P$ {P is a parallel node}
2: parallelize $A$ creating $A_1, A_2, ..., A_n$
3: for $(i = 1 \ldots n)$ do {prepend each $A$ to the children of $P$}
4:   $C_i$ ← $i$th child of $P$
5:   if $(C_i$ is a series node) then
6:     prepend $A_i$ to $C_i$
7:     SLIDERIGHT($A_i$) {slide $A_i$ to the proper location in $C_i$}
8:   else {C_i must be a leaf}
9:     $C_i$ ← a new series node with $A_i$ followed by the original $C_i$
10:    RESTRUCTURE($C_i$) {restructure the new series node}
11: end if
12: end for
CHAPTER 3. RESTRUCTURING AN ARMADA GRAPH

Figure 3.3: The forbidden subgraph $N$.

Figure 3.4: Two configurations of series-connected ships that produce non-series-parallel graphs when swapped. Bold lines highlight the forbidden subgraph $N$ in the restructured portions.

do not produce series-parallel DAGs after a swap by using the forbidden subgraph characterization of Valdes et al. [VTL82]. They proved that a DAG is series-parallel if and only if its transitive closure does not contain the graph $N$ (shown in Figure 3.3) as an induced subgraph. An induced subgraph is obtained by deleting some vertices and all edges incident with deleted vertices.

Figure 3.4-a shows an example of the case when a distribution ship $B$ is not series-connected to a parallel node. In this particular example, the left-most ship $A$ is non-structural. Figure 3.4-b shows a similar example of the second case, where a merge ship does not follow a parallel node. In each example, bold lines indicate the forbidden subgraph $N$ in the restructured portions.
### 3.2. RESTRUCTURING SERIES-CONNECTED NODES

#### Algorithm 7  COMMUTATIVE($A, B$)

1: if ($A$ or $B$ is data-equivalent) and ($A$ or $B$ is request-equivalent) then
2:     return true
3: else
4:     return false
5: end if

#### 3.2.3 Commutativity of series-connected ships

The second condition for making a swap requires two connected ships to be request-commutative and data-commutative. Two series-connected ships are request-commutative if the graph created by swapping the two ships generates a sequence of requests equivalent to the requests produced by the original graph (see Section 2.3.2 for our definition of equivalence). Similarly, two series-connected ships are data-commutative if the graph created by swapping the two ships generates a sequence of data equivalent to the data produced by the original graph. The **COMMUTATIVE** function (Algorithm 7) uses user-provided ship properties to decide commutativity. It returns *true* if and only if at least one ship is request-equivalent and at least one ship is data-equivalent. Intuitively, **COMMUTATIVE** returns *true* when the request-mapping function for at least one ship is the request-identity,\(^1\) and the data-mapping function for at least one ship is the data-identity. The following theorems provide a more theoretical proof of commutativity.

Theorem 3.2.1 proves that two series-connected ships are request-commutative if at least one of the ships is request-equivalent. Theorem 3.2.2 proves that two series-connected ships are data-commutative if at least one of the two ships is data-equivalent.

**Theorem 3.2.1.** Two series-connected ships that produce an SP-DAG when swapped, are request-commutative if either ship is request-equivalent.

**Proof.** The first clause in the theorem allows us to restrict our proof to the four allowable cases from Section 3.2.1. Below, we prove request-commutativity for each case by showing that the results from the initial configuration are equivalent to the results from the swapped configuration.

- Two non-structural series-connected ships with request mappings $R^1 \xrightarrow{A} s^1 \xrightarrow{B} T^1$ for the initial graph, and $R^1 \xrightarrow{B} U^1 \xrightarrow{A} V^1$ for the swapped graph. The two ships are request-commutative if $V^1 \equiv T^1$.

\[
R^1 \xrightarrow{A} s^1 \xrightarrow{B} T^1 \quad R^1 \xrightarrow{B} U^1 \xrightarrow{A} V^1
\]

**Lemma.** If $A$ is request-equivalent, $V^1 \equiv T^1$.

**Proof.**

1. $S^1 \equiv R^1$: definition of request-equivalence,

\(^1\)A request-identity function returns an “equivalent” sequence of requests, which may not be identical to the original sequence of requests.
2. $V^1 \equiv U^1$: definition of request-equivalence,
3. $U^1 \equiv T^1$: $U^1$ is the output of ship $B$ with input $R^1$, $T^1$ is the output of ship $B$ with input $S^1 \equiv R^1$, and
4. $V^1 \equiv T^1$: transitivity from 2 and 3.

\[ \square \]

**Lemma.** If $B$ is request-equivalent, $V^1 \equiv T^1$.

**Proof.**
1. $U^1 \equiv R^1$: definition of request-equivalence,
2. $S^1 \equiv T^1$: definition of request-equivalence,
3. $V^1 \equiv S^1$: $S^1$ is the output of ship $A$ with input $R^1$, $V^1$ is the output of ship $A$ with input $U^1 \equiv R^1$, and
4. $V^1 \equiv T^1$: transitivity from 2 and 3.

\[ \square \]

- A series connection of a right-parallelizable non-structural ship $A$, a distribution ship $B$, and a parallel node. The request mappings are $R^1 \xrightarrow{A} S^1 \xrightarrow{B} T^n$ for the initial graph and $R^1 \xrightarrow{B} U^n \xrightarrow{A} V^n$ for the restructured graph. The ships are request-commutative if $V^n \equiv T^n$.

\[ \text{(a) original.} \]

\[ \text{(b) swapped.} \]

**Lemma.** If $A$ is request-equivalent, $V^n \equiv T^n$.

**Proof.**
1. $S^1 \equiv R^1$: definition of request-equivalence,
2. $V^n \equiv U^n$: definition of request-equivalence,
3. $U^n \equiv T^n$: $U^n$ is output from $B$ with input $R^1$, $T^n$ is output from $B$ with input $S^1 \equiv R^1$, and
4. $V^n \equiv T^n$: transitivity from 2 and 3.

\[ \square \]
Lemma. If $B$ is request-equivalent, $V^n \equiv T^n$.

Proof.
1. $U^n \equiv R^1$: definition of request-equivalence,
2. $T^n \equiv S^1$: definition of request-equivalence,
3. $S^1 \equiv V^n$: definition of parallelizable (see Section 2.3.2), $S^1$ is the output of $A$ with input $R^1$, $V^n$ is the output of parallelized $A$ with input $U^n \equiv R^1$, and
4. $V^n \equiv T^n$: transitivity from 2 and 3.

Lemma. If $A$ is request-equivalent, $V^1 \equiv T^1$.

Proof.
1. $S^1 \equiv R^n$: definition of request-equivalence,
2. $V^1 \equiv U^n$: definition of request-equivalence,
3. $U^n \equiv T^1$: definition of parallelizable, $U^n$ is the output of parallelized $B$ with input $R^n$, $T^1$ is the output from $B$ with input $S^1 \equiv R^n$, and
4. $V^1 \equiv T^1$: transitivity from 2 and 3.

Lemma. If $B$ is request-equivalent, $V^1 \equiv T^1$.

Proof.
1. $U^n \equiv R^n$: definition of request-equivalence,
2. $T^1 \equiv S^1$: definition of request-equivalence,
3. $S^1 \equiv V^1$: $S^1$ is the output of $A$ with input $R^n$, $V^1$ is the output of $A$ with $U^n \equiv R^n$ as input, and
4. $V^1 \equiv T^1$: transitivity from 2 and 3.

- A series connection of a parallel node, a right-parallelizable merge ship $A$, a left-parallelizable distribution ship $B$, and another parallel node. The request mappings are $R^n \xrightarrow{A} S^1 \xrightarrow{B} T^k$ for the initial graph and $R^n \xrightarrow{B} U^{nk} \xrightarrow{A} V^k$ for the restructured graph. The ships are request-commutative if $V^k \equiv T^k$.

\[ \begin{align*}
R^n &\xrightarrow{A} S^1 & T^k \\
R^n &\xrightarrow{B} U^{nk} & V^k
\end{align*} \]

**Lemma.** If $A$ is request-equivalent, $V^k \equiv T^k$.

**Proof.**

1. $S^1 \equiv R^n$: definition of request equivalence,
2. $V^k \equiv U^{nk}$: definition of request-equivalence,
3. $U^{nk} \equiv T^k$: definition of parallelizable, $U^{nk}$ is the output of the parallelized version of $B$ with input $R^n$, $T^k$ is the output of $B$ with input $S^1 \equiv R^n$, and
4. $V^k \equiv T^k$: transitivity from 2 and 3.

**Lemma.** If $B$ is request-equivalent, $V^k \equiv T^k$.

**Proof.**

1. $U^{nk} \equiv R^n$: definition of request-equivalence,
2. $T^k \equiv S^1$: definition of request-equivalence,
3. $S^1 \equiv V^k$: definition of parallelizable, $S^1$ is the output of ship $A$ with input $R^n$, $V^k$ is the output of the parallelized version of ship $A$ with input $U^{nk} \equiv R^n$.
4. $V^k \equiv T^k$: transitivity from 2 and 3.
Algorithm 8 \textsc{BeneficialSwap}(A, B)

1: if \textsc{Direction}(A) > \textsc{Direction}(B) then
2: return \textit{true}
3: else
4: return \textit{false}
5: end if

Therefore, all configurations of two series-connected ships that produce an SP-DAG when swapped are request-commutative if at least one of the two ships is request-equivalent.

\hfill \Box

\textbf{Theorem 3.2.2.} Two series-connected ships $A$ and $B$ that produce an SP-DAG when swapped, are data-commutative if either $A$ or $B$ is data-equivalent.

\textit{Proof.} Theorem 3.2.1 shows that for each of the four configurations that produce SP-DAGs when swapped, if either $A$ or $B$ is request-equivalent, the two ships are request-commutative. Since the data-flow and request-flow are treated independently and the algebra involved in proving commutativity for request mappings is identical to the algebra required to prove commutativity for data-mappings, it follows that if $A$ or $B$ is data-equivalent, then $A$ and $B$ are data-commutative.

\hfill \Box

3.2.4 A swap should benefit the application

Although two series-connected ships may be commutative, a swap of the two ships may not improve the performance of the application. The boolean function \textsc{BeneficialSwap} (Algorithm 8) provides this last step in deciding whether to perform a swap of two series-connected ships. The goal is to accurately predict an increase or decrease in the overall application performance resulting from a swap. Our initial heuristic is a greedy approach based on two expectations: increased parallelism leads to improved performance, and moving data-reducing ships closer (in terms of the number of ships) to the data source and data-increasing ships closer to the data destination results in improved performance by reducing the amount of data transferred through slow portions of the network.

\textsc{BeneficialSwap} uses the \textsc{Direction} function (Algorithm 9) to assign a preferred direction to each ship (1 for right, --1 for left, and 0 for no preference). As in our diagrams, we use the convention that storage is on the right and clients are on the left. Based on the expectation that increased parallelism leads to improved performance, the merge ship always prefers to move right and the distribution ship always prefers to move left. If a ship is a \textit{data-reducer} (a property assigned by the programmer) the ship prefers to move toward the data source (right for reads, left for writes). Similarly, ships with the \textit{data-increaser} property prefer to move right for writes and left for reads. The \textsc{BeneficialSwap} function takes as input two connected ships $A$ and $B$, in which $A$ is currently to the left of $B$, and returns \textit{true} if the preferred direction of $A$ has a greater value than the preferred direction of $B$. This result can only happen if $A$ wants to go right and $B$ wants to go left, if $A$ wants to go right and $B$ has no preference, or if $A$ has no preference and $B$ wants to go left.
CHAPTER 3. RESTRUCTURING AN ARMADA GRAPH

Algorithm 9 DIRECTION(A)

1: if (A is a merge ship) then
2: return 1 {want to go right to increase parallelism}
3: else if (A is a distribution ship) then
4: return −1 {want to go left to increase parallelism}
5: end if
6: if (read-only armada) then
7: if A is a data-reducer then
8: return 1 {want to move closer to the data servers}
9: else {A is a data-increaser}
10: return −1 {want to move closer to the clients}
11: end if
12: else {write-only armada}
13: if (A is a data-increaser) then
14: return 1 {want to move closer to the data servers}
15: else {A is a data-reducer}
16: return −1 {want to move closer to the clients}
17: end if
18: end if
19: return 0

3.3 Algorithm summary

The goal of the restructuring algorithm is to take an SP-tree that represents a distributed graph of application objects, and restructure the tree (adding nodes where necessary) to create a new SP-tree with increased parallelism and an arrangement of data-processing objects that allows a more efficient deployment of processing objects to grid resources. The algorithm works by traversing the tree depth-first and reordering (through a sequence of “swaps”) series-connected nodes in the tree when

- two series-connected nodes are commutative,
- swapping two series-connected nodes benefits the application, and
- the graph resulting from a swap of two series-connected nodes is an SP-tree.

Note that our solution is not optimal, since there are cases in which both ships are commutative and a swap benefits the application (by our definition of beneficial), but swapping the two ships will produce a non-SP-tree. We believe these cases to be rare; however, the benefits of the keeping the graph series-parallel outweigh the benefit of covering those cases.

3.4 Analysis

The total time to restructure a graph is the sum of the time to initialize the nodes in the graph and the time to recursively traverse and restructure the SP-tree.

Initialization includes verifying that the tree is series-parallel, compressing it to form a CSP-tree, and initializing internal nodes of the tree to dirty. A paper from Valdes et al. [VTL82] describes a linear-time algorithm for recognizing series-parallel DAGs and trees, so recognizing a SP-tree requires only the linear running time of the Valdes algorithm. To convert the SP-tree to a CSP-tree, we traverse the tree depth-first and combine nodes where the children and the parent are of the same type. We also visit each internal node
Algorithm 10 ConvertToBinary(N)

1: if (N is a leaf) then
2:     return {base case}
3: else {N is either a series node or a parallel node with children a₁, a₂, ..., aₙ}
4:     right ← a new node (of the same type as N) with a₂, a₃, ..., aₙ as its children
5:     ConvertToBinary(a₁) {recursive traversal}
6:     ConvertToBinary(right) {recursive traversal}
7:     N ← a new node with a₁ and right as its only children {N is now binary}
8: end if

once to initialize them to dirty. Since each task runs in linear time, with respect to the number n of nodes in the SP-tree, the initialization step has a running time of \(O(n)\).

Our analysis of graph restructuring has two parts: first, we calculate the size of the largest possible tree (based on the number of leaves in the initial tree) and the number of operations used by ParallelizeLeft and ParallelizeRight to generate the tree, and then we calculate the number of operations (based on the number of nodes in the largest possible tree) used by Restructure, SlideLeft, and SlideRight to decide whether or not to swap series-connected nodes. An upper bound on the total cost of restructuring is the sum of the number operations use to generate the largest possible tree, and the number of operations used to traverse and decide which nodes to swap.

The first set of theorems calculate upper bounds on the size of the SP-tree before restructuring.

**Theorem 3.4.1.** There are at most \(v - 1\) internal nodes (parallel and series) in an SP-tree with \(v\) leaves.

**Proof.** Convert the SP-tree to a binary SP-tree (using Algorithm 10), creating series and parallel nodes where necessary. The binary SP-tree has exactly \(v\) nodes of degree two. Since all internal nodes of a binary SP-tree are degree two (a degree-one internal node is unnecessary), there are exactly \(v - 1\) internal nodes in the binary tree and at most \(v - 1\) internal nodes in any SP-tree.

**Theorem 3.4.2.** There are at most \(2v - 2\) edges in an SP tree with \(v\) leaves.

**Proof.** Theorem 3.4.1 states that there are at most \(v - 1\) internal nodes of an SP-tree with \(v\) leaves, which means the tree has at most \(2v - 1\) total nodes. Since a tree with \(n\) nodes (internal nodes and leaves) has \(n - 1\) edges (every node has one parent except the root), there are at most \(2v - 2\) edges in an SP-tree with \(v\) leaves.

The previous set of theorems establish upper bounds on the size of the original SP-tree based on the number of leaves. Now we use properties of the algorithm to calculate upper bounds on the size of the restructured tree based on the number of leaves in the original tree. Notice that our algorithm only creates new series nodes and leaves. The number of parallel nodes remain constant. The following lemmas calculate upper bounds on the number of series nodes and leaves created by the restructuring algorithm.

**Theorem 3.4.3.** The restructuring algorithm creates at most \(v\) series nodes, where \(v\) is the number of leaves in the original SP-tree.

**Proof.** The ParallelizeRight and ParallelizeLeft procedures create a new series node only if the child of a parallel node is a leaf. This creation happens at most once for each leaf in the original SP-tree since the restructuring algorithm does not create parallel nodes. Therefore, restructuring creates at most \(v\) new series nodes, where \(v\) is the number of leaves in the original SP-tree.
Corollary 3.4.4. There are $O(v)$ internal nodes in the restructured SP-tree, where $v$ is the number of leaves in the original tree.

Proof. Theorem 3.4.1 states that there are at most $v-1$ internal nodes in the original tree, and Theorem 3.4.3 states that restructuring creates at most $v$ new series nodes. Since the number of parallel nodes remains constant, the number of internal nodes in the restructured tree is at most $2v-1 = O(v)$.

Theorem 3.4.5. Restructuring creates at most $2v^2 - 2v$ leaves, where $v$ is the number of leaves in the original SP-tree.

Proof. When parallelizing a ship (leaf of the SP-tree), line 2 of the PARALLELIZERIGHT or PARALLELIZERIGHT procedure on page 33 creates one copy of the ship (thus creating a new leaf) for each edge of the parallel node to the right of the series-connected distribution ship (for PARALLELIZERIGHT) or for each edge of the parallel node to the left of the adjacent merge ship (for PARALLELIZERIGHT). From Theorem 3.4.2, we know there are at most $2v-2$ edges in the original SP-tree. Since PARALLELIZERIGHT and PARALLELIZERIGHT do not create parallel nodes or increase the number of edges from parallel nodes, there can be at most $2v^2 - 2v$ new leaves created as a result of restructuring a single ship, which results in a maximum of $v(2v-2) = 2v^2 - 2v$ new leaves. Therefore, there can be at most $2v^2 - 2v$ leaves created by the restructuring algorithm.

Theorem 3.4.6. The restructured tree has $O(v^2)$ nodes, where $v$ is the number of leaves in the original tree.

Proof. The maximum size of the restructured tree is the sum of the internal nodes from the original tree and the nodes (both leaves and series nodes) created during restructuring. Theorem 3.4.1 proves that there are at most $v-1$ internal nodes for an SP-tree with $v$ leaves; therefore, there are at most $2v-1$ total nodes (including leaves) in the initial SP-tree. Theorem 3.4.3 proves that restructuring creates at most $v$ internal nodes. Theorem 3.4.5 proves that restructuring creates at most $2v^2 - 2v$ new leaves. If we sum the initial number of nodes, the number of new internal nodes, and the number of new leaves, we get $(2v-1) + (v) + (2v^2 - 2v) = 2v^2 + v - 1 = O(v^2)$ for the maximum size of the restructured SP-tree.

Since we now have upper bounds on the size of the restructured tree, we calculate upper bounds on the running time of the entire restructuring algorithm by summing the number of comparisons between series-connected ships with the number of operations used to create the new SP-tree (operations executed after a successful comparison). We define a comparison as the test used by SLIDELEFT or SLIDERIGHT to decide whether to swap two series-connected ships.

Lemma 3.4.7. There are at most $O(v)$ children in any series node of the restructured SP-tree, where $v$ is the number of leaves in the original tree.

Proof. There can be no more than one copy of each original leaf in any series node. Therefore there are at most $v$ leaves in a series node of the restructured tree, and at most $O(v)$ children (leaves and internal nodes) of a particular series node.

Theorem 3.4.8. There are $O(v^3)$ comparisons of series-connected nodes, where $v$ is the number of leaves in the original SP-tree.

Proof. The SLIDELEFT and SLIDERIGHT procedures place a node into a previously sorted list of series-connected nodes by comparing nodes to the left (for SLIDELEFT) or to the right (for SLIDERIGHT) of the new node. If the two nodes cannot swap, then the node must be correctly placed, and so the procedure
3.5. RELATED WORK

exits. In the worst case, the algorithm makes one comparison between each child and every other child in the series node. Lemma 3.4.7 proves that there are $O(v)$ children in a series node, so restructuring a single series node requires $O(v^2)$ comparisons. Since Corollary 3.4.4 proves that there are $O(v)$ internal nodes in the restructured SP-tree, restructuring an SP-tree requires $O(v^3)$ comparisons. □

**Theorem 3.4.9.** The running time of the restructuring algorithm is $O(v^3)$, where $v$ is the number of leaves in the original tree.

**Proof.** Our runtime analysis calculates an upper bound on the running time by summing the following three items: the number of operations used by \textsc{Restructure} to traverse the restructured SP-tree, the number of operations used to create nodes (leaves and series nodes), and the number of operations used to compare series-connected nodes.

Since the \textsc{Restructure} procedure traverses the tree depth-first, visiting each node exactly once, the number of operations used to traverse the tree is linear with respect to the size of the restructured tree, which is $O(v^2)$ from Theorem 3.4.6.

Since creating a node and appending (or prepending) it to a series node requires a constant number of operations, the number of operations used to create nodes is the sum of the number of new series nodes and the number of new leaves. From Theorems 3.4.3 and 3.4.5, restructuring creates at most $v$ series nodes and $2v^2 - 2v$ leaves, resulting in a running time of $O(v^2)$.

Theorem 3.4.8 shows that there are $O(v^3)$ comparisons of series-connected nodes. A comparison consists of a check for commutativity between the two ships, a check to see if a swap of the two ships is beneficial to the application, and a check to see if the configuration of the ships matches one of the four allowed configurations from Table 3.1. Each operation has constant running time, so the total running time of comparing series-connected nodes is $O(v^3)$.

Summing all three items, we get a total running time of $O(v^2 + v^2 + v^3) = O(v^3)$. □

3.5 Related Work

As mentioned above, Armada builds a data-flow graph by combining two graphs: one from the data provider that describes the layout of the data, and an application-defined graph that describes the interface and processing required by the client. The Armada runtime system restructures the combined graph to improve data flow between the data servers and the client. The two primary features of our restructuring algorithm are the automatic parallelization of ships and the reordering of ships to improve data-flow performance. This section describes a few projects with related goals.

3.5.1 Automatic parallelization of user code

Since parallel computation became popular in the 1980’s, many projects have tried to make the task of programming parallel applications easier for the developer [BST89, Bal92]. One approach attempts to automatically convert sequential programs into parallel programs. A Carnegie Mellon University web page provides a comprehensive reading list on parallel programming languages and systems that support automatic parallelization. Here we discuss three systems of particular relevance to Armada because of their focus on parallel processing of I/O streams.

\footnote{CMU Reading List on Parallel Programming Languages, \url{http://www-2.cs.cmu.edu/~scandal/parallel-lang/reading-list/reading-list.html}}
The Parallel Storage-and-Processing Server (PS^2) [Mes99], from École Polytechnique Fédérale de Lausanne, is designed specifically for I/O-intensive applications. It uses the Computer-Aided Parallelization tool (CAP) [GMMH98] to express the parallel behavior of the application. The CAP system constructs a data-flow computation graph with “actors” as nodes of the graph. The actors are computational units that provide application-specific functionality. For I/O-intensive applications, actors provide application-specific data distribution, prefetching, or filtering. CAP parallelizes a task by inserting split actors to create and distribute sub-tokens and merge actors to gather and synchronize processed sub-tokens after computation.

A project at Duke University is investigating extensions to the streaming computation model that map computation to active disks [WCV02]. They specify primitive computations as “functors” that perform simple computations as a side-effect of I/O access. Functors have bounded compute and memory requirements. The application exposes functors and their compute costs to the system and the system decides (based on analytic models) a mapping of functors to active disks in a way that balances load. They are primarily interested in applications that implement I/O-efficient external memory algorithms discussed in [AV99, Vit99].

DataCutter^3 [BFK+00], developed at the University of Maryland, is middleware used to explore and analyze scientific datasets stored on archival storage systems across a wide-area network. DataCutter provides a query-based interface with support for accessing subsets of datasets and for performing user-defined transformations of large data sets in archival storage. The processing structure is composed of a set of processes called “filters” that typically execute close to the data source. A recent paper by Spencer et al. describes how DataCutter can replicate filters to distribute computation inside a pipelined data-flow graph [SFB+02]. The scheme uses an analytic model, based on measured performance, to decide how many additional filters to create, then it distributes data elements to filters using either a round robin, weighted round robin, or demand-driven distribution policy. The system then merges results from the distributed filters and forwards the result to the next stage in the data-flow graph.

The systems described above all implement a common algorithmic technique known as partition-and-merge. In this technique, the system first divides the problem into subtasks that execute in parallel, then it merges results in a single synchronous step. The limitation of this approach is the assumption of a single input and single output. Armada takes a slightly different approach. The application portion of the Armada graph describes the layout of data to parallel compute nodes with the assumption of a single data source. The data provider describes its layout of the data to parallel storage servers with the assumption of a single compute node for the application. The Armada system combines the two graphs and restructures to provide end-to-end parallelism in an attempt to remove synchronization points.

### 3.5.2 Operation ordering to improve data flow

Another feature of the Armada system is the ability to re-order operations in the graph to improve data flow to or from the client. This approach extends the work of projects in the distributed database community that have investigated data flow optimization for many years [DGG+86, ADAT+99]. Two systems of particular interest are the dynamic QUery OBject (dQUOB) system, and River.

The dynamic QUery OBject (dQUOB) system [PS00] is a runtime system for managing and optimizing large data streams in which the users request remote data with SQL-like queries, and computations are performed on the data stream between the remote data servers and the client. The computation is encapsulated in a piece of compiled code called a “quoblet.” After the dQUOB compiler converts an SQL query to a “query tree,” the tree is evaluated at runtime to determine which portions of the query tree apply the most filtering. The tree is then restructured to move the high-filtering portions closer to the data.

---

River [ADAT+99, AD03] is a data-flow programming environment for database query processing applications. River is specifically designed for clusters of computers with heterogeneous performance characteristics. The goal of the River runtime system is to adapt to “performance faults”—portions of the system that perform poorly—by dynamically adjusting the transfer of data through the data-flow graph. River uses two constructs to build applications: a distributed queue that deals with performance faults by consumers and graduated declustering that deals with performance faults of producers. A distributed queue pushes data through the data-flow graph at a rate proportional to the rate of consumption and adapts to changes in consumption rates. Graduated declustering deals with producer performance faults by reading from replicated producers. Although River is designed specifically for query processing, the authors briefly discuss how one might adapt scientific applications to work in their framework [AD03].

Much of the success of distributed query processing systems can be attributed to the relational model. Relational operators have well-defined properties that are ideal for parallel processing execution [DG92] and easy to describe mathematically. Armada presents a more general approach that allows the reordering of arbitrary functions (not just query processing functions) based on programmer-supplied properties that help the system to decide commutativity between adjacent functions.
Chapter 4

Placement of Armada Ships

Ships that make up the Armada graph execute on processors near the client, processors near the data, or intermediate network processors. An effective placement (especially for ships that increase or decrease data flow) has a significant impact on the overall performance of the application. Our approach is to treat placement as a hierarchical graph-partitioning problem. We first partition the graph into administrative domains in an attempt to minimize data transferred between domains. Then we partition ships in each domain to processors provided by domain-level resource managers.

This chapter consists of three parts: a brief description of graph partitioning and its applications; a description of a software library, called Chaco, that provides a set of heuristic solutions to graph partitioning; and a detailed description of how we use Chaco to implement a hierarchical placement algorithm for Armada graphs.

4.1 Graph partitioning

A large class of high-performance applications can conveniently be described in the form of a graph. Many of these applications benefit by partitioning the graph into sets that are, by some measure, as disjoint as possible. Specific examples include divide-and-conquer algorithms for designing efficient circuit layouts [KAKS99], the construction of nested-dissection orderings for sparse matrix factorizations [HR99], and parallel discrete-event simulations [Liu03]. In general, many parallel applications that map large data structures to distributed processors benefit by such a partitioning.

We now state the general graph-partitioning problem. Given a graph with \( n \) weighted vertices and \( m \) weighted edges, find a partitioning of the vertices that minimizes flow across the partitions, and balances the sum of the vertex weights in each partition. Unfortunately, the graph-partitioning problem is NP-complete [GJS76].

Although no known efficient solution exists (or is likely to exist) for the graph-partitioning problem, there are several efficient heuristic solutions. The difficulty for the application designer now becomes choosing an appropriate heuristic. Some solutions, for example, disregard the identity of the partition and try to minimize the number of edge crossings (size of the cuts). Such solutions do not work well for circuit design or parallel applications where the identity of the set matters. For these types of applications, solutions exist that try to minimize the number of connections between architecturally distant partitions. There are a number of other trade-offs to consider when choosing an appropriate algorithm [Pot95]; however, a complete discussion of these topics is beyond the scope of this work.
4.2. THE CHACO PARTITIONING SOFTWARE

Over the past decade, much work has gone into the development of software libraries that provide implementations of various heuristic solutions to graph partitioning. The most well known of these are METIS [KK98], PARTY [PD96], JOSTLE [WC02, WCE97], SCOTCH [PR96], and Chaco [HL94]. Although each of these libraries provides sufficient functionality for our placement scheme, we chose to use Chaco, partly because of our familiarity with its authors, but also because of the flexibility of their approach and the breadth of options available to the user.

4.2 The Chaco partitioning software

Chaco is a software library for graph partitioning that implements five classes of partitioning algorithms: simple, spectral [HL95a, PSL90, Sim91], inertial [Sim91, Wil91, LH94], Kernighan-Lin (KL) [KL70, SK88], and multilevel-KL [HL95b]. Each algorithm uses a recursive decomposition that partitions the graph into two, four, or eight sets at each level of recursion. For details of the individual algorithms, see [HL94] or the previously cited papers. In this section, we focus on the algorithms and parameters used by Armada to partition an Armada graph.

4.2.1 Spectral partitioning with KL local refinement

After careful consideration of the various algorithms available, we decided to use a combination of the spectral and Kernighan-Lin algorithms for partitioning.

Spectral methods partition a graph based on eigenvectors of a matrix constructed from the graph. The details about how and why this approach works are in [HL95a]. The particular method used by Armada is spectral bisection. In this method, Chaco uses the second lowest eigenvector of the Laplacian matrix to divide the graph into two pieces. Chaco also provides several eigen solvers with different trade-offs of speed and robustness to calculate the eigen vectors. We use the Lanczos eigen solver with selective orthogonalization. This solver is the default and is recommended for graphs with fewer than 10,000 vertices. Since we expect Armada graphs to be relatively small (by these standards), the default solver is an appropriate choice.

Spectral methods do a particularly good job of finding the right general area to make a cut, but they often fail to make good decisions at the set boundaries. To deal with this limitation, Chaco provides a two-pass method that sends the result of spectral partitioning to a Kernighan-Lin method.

Kernighan-Lin partitioning [KL70] uses a greedy, local optimization strategy that moves vertices between sets to try to reduce the number of edge crossings. The authors of Chaco generalized the original algorithm to work with vertex and edge weights [HL95b].

In the Chaco user manual [HL94], Hendrickson and Leland state that KL finds good partitions only if given a good initial partition. Since spectral methods provide a good initial partitioning (but fail at the set boundaries), and KL methods only work well when given a good initial partitioning, using spectral methods for global optimization and KL methods for local refinement produces a much better partitioning than either method used on its own.

4.2.2 Chaco parameters used by Armada

Chaco comes with a rich set of parameters that allow the user either to fine tune the algorithm used for partitioning or to extend the default functionality to include some special feature. Although the default settings are sufficient for most parameters, we manually configure two of the parameters: one controls the

---

1Chaco is a product of Sandia National Laboratories, which is the primary source of funding for the Armada project.
importance placed on balancing the partitions, the other is a special feature that improves the placement of certain types of graphs by using a technique known as *terminal propagation*.

**Balancing partitions**

There is usually a trade-off between minimizing the edge weights crossing partitions and balancing the sum of the vertex weights in each partition. By default, Chaco tries to partition sets to make the sum of the vertex weights in each set as equal as possible; but by modifying the KL_IMBALANCE parameter (specifically for the KL algorithm), the user specifies the importance of providing a balanced partition. The values of KL_IMBALANCE range from 0 to 1 (default is 0). For wide-area distributed environments, it is often more important to minimize the flow across partitions than it is to have balanced partitions. We set the value of KL_IMBALANCE to 0.7.

**Terminal Propagation**

Terminal propagation [DK85] is a modification to the Kernighan-Lin algorithm designed to improve data locality by incorporating information about outgoing edges (or terminals) into the partitioning algorithm. It was originally designed for VLSI chip design, but may be useful for any application where the mapping to specific sets is important. In Armada the client and storage interface ships, which typically have a fixed host assignment, often form the end-points of the graph. Terminal propagation allows the host assignments of the interface ships to remain fixed during the graph partitioning process.

Chaco includes terminal propagation for the bisection mode of spectral partitioning (the algorithm used by Armada) and for KL and multilevel-KL algorithms. The parameter TERMINAL_PROP enables terminal propagation when set to 1 (the default value is 0). For details on the incorporation of terminal propagation into Chaco algorithms, see [HLD96].

### 4.2.3 Adapting Chaco to work with Armada

Adapting Chaco to work with Armada was easy. The Chaco software consists of a main driver application (for command-line operation) and an application interface to the library (all C code). The original version of Chaco included the driver and the interface in the same file. To get the C code of Chaco to work with Armada (written in Java), we separated the driver from the interface and compiled the entire Chaco code (excluding the driver) into a shared library object. We then created an Armada interface to Chaco using the Java Native Interface. The Armada placement algorithm dynamically links to the Chaco library at runtime.

### 4.3 Placement of the Armada ships

With some background on graph partitioning and the Chaco graph-partitioning software, we now describe how Armada uses Chaco to assign ships in an Armada graph to hosts in a computational grid. Note that placement in our current implementation is a static operation that occurs as a pre-processing step, before deployment of the Armada ships. We discuss some of the implications of a dynamic placement algorithm as future work in Section 6.3.2.

The placement process requires three steps: constructing a model of the data flow, partitioning the model into sub-graphs that represent administrative domains, and partitioning each sub-graph to decide host assignments for each of the Armada ships. We describe each of these processes in turn.
4.3. PLACEMENT OF THE ARMADA SHIPS

Algorithm 11 \textsc{Propagate-Flow}(G)
\begin{algorithmic}[1]
\State initialize \textit{inFlow} of each vertex to zero
\State \textit{inFlow}(\text{root}) = 100
\State \textsc{Topological-Sort}(G) \{takes }O(V + E)\text{ time\}
\For{each vertex }u \in V \text{ in topological-sort order}
\State \textit{perEdge} = (u﻿.dataRatio \times u﻿.inFlow)/u﻿.numAdj
\EndFor
\end{algorithmic}

4.3.1 Creating a model of the data flow

The first step in deciding a placement is to construct a model of the Armada graph that includes weighted vertices and weighted edges. We designed a generalized \texttt{Graph} class for this purpose. The constructor of the \texttt{Graph} class creates an adjacency-list representation of the Armada graph from the series-parallel tree. Each vertex in the adjacency list contains a pointer to the represented \texttt{Leaf} from the \texttt{SPTree}, an assigned weight (a floating-point value), and a linked list of edges to adjacent vertices. Each edge contains a floating-point weight value and a pointer to the adjacent vertex. Vertex weights represent the expected computation (per data unit) of the associated ship. Edge weights represent the data flow across the edge.

Since, initially, we are not particularly interested in balanced sets (see Section 4.2.2) we assign equal weights to the vertices. An implementation concerned with balancing computation would likely assign vertex weights based on statistics from previous runs of the application, or an approximation by the programmer.

We assign edge weights by propagating an artificial data flow from a “super-source” (a virtual vertex with an edge to all the actual sources) to all of the sinks. Algorithm 11 details the \textsc{Propagate-Flow} algorithm. \textsc{Propagate-Flow} calculates an \textit{inFlow} variable for each vertex based on the sum of the weights of the edges flowing into the vertex. It also assigns weights to outgoing edges based on the \textit{inFlow} variable and \textit{dataRatio} variable. The \textit{dataRatio} value (assigned by the programmer) represents the ratio of the amount of input data to the amount of output data for a particular ship. For example, a filter that reduces data by half would have \textit{dataRatio} = 0.5.

\textsc{Propagate-Flow} starts by initializing the \textit{inFlow} of every vertex to 0. It then initializes the \textit{inFlow} of the super-source to 100 (it could actually be any value). \textsc{Propagate-Flow} next performs a \textsc{Topological-Sort} on the input graph. Then, in topological-sort order, \textsc{Propagate-Flow} calculates and assigns a weight to each outgoing edge, and updates the \textit{inFlow} variable for each adjacent vertex. The weight assigned to each edge is the \textit{dataRatio} times the \textit{inFlow} divided by the number of adjacent vertices.

\textsc{Topological-Sort} ranks the vertices of a directed acyclic graph (DAG) in order of the reverse finishing times of a depth-first search [CLR90]. For example, Figure 4.1 shows the topological-sort ordering of the restructured Armada graph for the representative application. A topological-sort ordering guarantees that a vertex with rank \( k \) has no incoming edges from vertices with rank higher than \( k \). Therefore, we know the \textit{inFlow} variable for a particular vertex will not be modified after \textsc{Propagate-Flow} visits the vertex. \textsc{Propagate-Flow} runs in \( O(V + E) \) time, since it updates each edge exactly once.

The \textsc{Propagate-Flow} algorithm makes two assumptions about the graph: all sources generate the same amount of data, and data flows equally along all data paths. Both assume the best case; however, given
CHAPTER 4. PLACEMENT OF ARMADA SHIPS

Figure 4.1: Topological-sort ordering of the restructured Armada graph for the representative application. The vertex on the far right is the super-source.

Figure 4.2: Simulated data flow through the restructured Armada graph of the representative application with a filter that reduces data by one half. Values inside the vertices represent the vertex weights (all identical, in the current implementation), values outside a vertex identify an edge weight.

no knowledge of the application access patterns, these are reasonable assumptions. A more accurate model might incorporate historical information about the application.

Figure 4.2 shows the edge and vertex weights for the restructured Armada graph of the representative application. In this particular case, the filter reduces the data flow by half \( \text{ioRatio} = 0.5 \).

4.3.2 Partitioning to administrative domains

After assigning edge and vertex weights to the graph, partitioning the graph to administrative domains requires a call to Chaco. Our current implementation assumes that the computational grid has no intermediate administrative domains. That is, we assign Armada ships to processors either in the administrative domain(s) of the client application, or in the administrative domains of the data servers. Figure 4.3 illustrates the domain-level partitioning of the representative application.
4.3.3 Assigning Armada ships to hosts within each domain

The final step is to assign Armada ships within each administrative domain to local processors. This involves calling Chaco for each sub-graph assigned to an administrative domain. Figure 4.4 shows the final placement of Armada ships for the representative application.

4.4 Summary

In this chapter we describe a placement method for Armada ships that leverages the Chaco graph partitioning software to first partition the Armada graph to administrative domains, and then partition the ships within each domain to available processors. The goals are to decide a placement that reduces data transferred across a wide-area network by placing data-reducing (or data-increasing) Armada ships into appropriate domains, and to decide a placement within administrative domains that adheres to policies set by system administrators.

Since Armada uses the Chaco graph-partitioning software, the role of the Armada system is to choose...
an appropriate partitioning algorithm (and parameters), and to provide Chaco with a model of the data flow graph use by the application. We chose an algorithm that combines two separate methods: a spectral method that provides a good initial partitioning, and a greedy method for local refinement along the partition boundaries. We calculate a data-flow model by propagating an artificial flow through a graph that uses programmer-supplied values for each Armada ship (representing the ratio of the input flow to the output flow).

In the next chapter, we evaluate the effectiveness of the restructuring and partitioning algorithms by measuring the performance of several applications. Although our algorithm produces reasonably good placements for each application, there is much room for improvement. For example, our model of the data flow assumes that all sources generate the same amount of data, and data flows equally along all data paths. A more accurate model would have to either get more information from the user or programmer, or rely on historical statistics about the data flow through each Armada ship. We discuss more about improving the placement algorithm in Section 6.3.2.
Chapter 5

Evaluation

Two important features of Armada are the ability to restructure the application graph to distribute computational or network load, and the effective placement of application objects to reduce network traffic. This chapter presents an evaluation of the performance benefit of restructuring and placement based on the I/O performance of applications using Armada.

The first section describes the testbed used to perform the experiments. The next three sections describe the design, implementation, and performance of three applications: a representative application, an application for third-party file transfers and data permutation, and a seismic processing application. The final section describes the design of an application that uses Armada for remote analysis of fMRI brain images.

5.1 The network testbed

There are three types of environments suitable for evaluating the performance of Armada: a computational grid testbed, network simulation, and network emulation. A computational grid testbed provides the most realistic environment, but does not allow direct control over network parameters, and thus no chance to experiment with a wide range of conditions. Simulation provides the most flexibility with respect to control over network parameters, but producing an accurate network and computational model is often difficult, requiring a significant amount of additional work on the part of the researcher. Emulation provides a nice mix between a real network testbed and a simulated environment and was our choice for evaluating Armada. In an emulated network environment, code executes on real computational resources that communicate through a simulated network.

The Emulab network testbed [WLS+02] is a software system that provides an emulated network environment for academic research in distributed systems and networking. Emulab works by allocating local nodes in a cluster and connecting them through a Virtual LAN (VLAN) that restricts traffic to the subnet defined by the user. Each node allocated by the user functions as either an application node, a simulated router, or a traffic generator. To enforce control over bandwidth, latency, and packet loss, Emulab routes controlled network traffic through automatically allocated additional processors that use Dummynet [Riz97] (a network emulator). The Dummynet nodes act as an Ethernet bridge between nodes in the virtual network and are transparent to experimental traffic.

Users describe the network topology with a script file based on the ns-2 (network simulator) language. The script file defines the processors needed, their configuration in the virtual network, and the network parameters for each link. After submitting the script file to Emulab through a web interface, Emulab allo-
Table 5.1: Summary of experiments using the Emulab cluster at the University of Utah. Total run time is the sum over all experiments (idle time not included).

<table>
<thead>
<tr>
<th></th>
<th>Representative</th>
<th>Copy</th>
<th>Seismic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of experiments</td>
<td>115</td>
<td>545</td>
<td>299</td>
</tr>
<tr>
<td>Total run time</td>
<td>6.42 hrs</td>
<td>6.3 days</td>
<td>13.8 hrs</td>
</tr>
<tr>
<td>Number of nodes used</td>
<td>15</td>
<td>30, 10</td>
<td>42</td>
</tr>
<tr>
<td>Average run time</td>
<td>201 sec</td>
<td>16.7 min</td>
<td>167 sec</td>
</tr>
<tr>
<td>Min run time</td>
<td>28.3 sec</td>
<td>57.1 sec</td>
<td>54.3 sec</td>
</tr>
<tr>
<td>Max run time</td>
<td>24.3 min</td>
<td>1.47 hrs</td>
<td>18.7 min</td>
</tr>
</tbody>
</table>

cates necessary processors for the experiment (including Dummynet nodes). Once allocated, the user has exclusive access (including root access) to the processors.

A useful feature of Emulab is the ability to dynamically change network parameters through the command line. This capability enables efficient parameter studies through batch-file processing for experiments that use the same network topology. In one instance, for example, we executed a batch job that ran five instances of a single application using seven different bandwidths, five different latencies, and three different application parameters, totaling over 500 experiments.

The Emulab software currently controls two clusters: one at the University of Utah, and one at the University of Kentucky. The Utah cluster consists of 168 processors, each connected to high-speed Cisco switches through five 100Mbps interfaces. The cluster in Kentucky has a nearly identical configuration, but has 50 processors.

The ability to easily configure network topologies and control network parameters (such as bandwidth and latency) make the Emulab network testbed ideal for examining the effectiveness and the flexibility of Armada. All experiments presented in this chapter used the Emulab network testbed at the University of Utah to produce results.

Table 5.1 shows a statistical summary of the experiments presented in this chapter. For each experiment we measured the end-to-end run time of the application across a variety of network parameters. Each data point is the mean of at least 4 (often 5) independent trials. The standard deviation for independent trials was measured to be at most 0.3 seconds.

### 5.2 Representative application

In Chapter 2, we introduced a representative application (see Figure 5.1-a) that reads data from a replicated and distributed dataset. The portion from the data provider describes the layout of two replicas, each stored as a distributed file. The application prepends a graph to the data provider’s portion that includes a filter and three interface ships (one for each client processor). In Chapter 3, we again used the representative application to illustrate the process of graph restructuring. We show the resulting restructured graph in Figure 5.1-b. Previous chapters thoroughly covered the design of this application. In this section, we present performance results for an implementation of the representative application.
5.2. REPRESENTATIVE APPLICATION

Figure 5.1: A representative application for reading a replicated and distributed dataset. (a) shows the original graph, (b) shows the restructured graph.

5.2.1 Experiments

We demonstrate the performance benefits of graph restructuring by measuring the execution time of four different configurations of the representative application. For each configuration, the network topology consists of a wide-area network (WAN) connecting three local-area networks (LANs).

Figure 5.2 illustrates the four configurations that use manual placements of Armada ships to processors. Each “blob” in the figure encompasses the ships deployed to a single LAN. Except for restruct1, each ship is installed on a separate processor within the LAN. The first configuration (labeled orig1) uses the original graph and places the filter on the same LAN as the client. In the second configuration (orig2) we use the
CHAPTER 5. EVALUATION

Figure 5.2: Four configurations of the filtering application.

original graph, but place the filter on the same LAN as one of the remote files. This particular assignment of Armada ships to domains matches the result of using our placement algorithm for the original graph. The third (restruct1) and fourth (restruct2) configurations use the restructured graph. In restruct1, each adjacent filter and segment ship share a host. In restruct2, we place each filter on a host in the same LAN different than that of its adjacent segment ship. In both restruct1 and restruct2, the assignment of Armada ships to domains matches that of our placement algorithm for the restructured graph.

The area between the blobs represents the WAN. Each LAN is connected to the WAN by a single router, which has a link to each of the other two LANs. We illustrate these links in Figure 5.3. Regardless of the number of graph edges crossing the WAN, each WAN link has limited capacity. In configuration orig2, note that the client LAN uses only one of its WAN connections, so the client/server cut has half the bandwidth of the other configurations.

Using Emulab, we configured each LAN in our topology to have five 850 MHz Intel Pentium III processors connected by a 100 Mbps switched network with a network latency of 0.15 msec. As mentioned above, the three LANs were connected by a triangle of three WAN links, each with equal bandwidth, and latency 2.0 msec. We varied the WAN link bandwidths to understand the performance of the application under different network conditions. For orig1, restruct1, and restruct2, we varied the WAN link bandwidths from 1 to 50 Mbps, so that the bandwidth available between client and servers varied from 2 to 100 Mbps. Since orig2 used only one client/server WAN link, we generously varied its WAN link bandwidths from 2 to 100 Mbps.
5.2. REPRESENTATIVE APPLICATION

![Network Topology Diagram](image)

**Figure 5.3**: Topology of network illustrating links between LANs.

5.2.2 Results and discussion

Figures 5.4 and 5.5 show timing and throughput measurements for each configuration, as the total client/server WAN bandwidth varied. In this particular application, the filter removed exactly fifty percent of the data. In Figure 5.5, we also show the optimal throughputs for orig1 (lower solid line) and the others (upper solid line).

For bandwidths below 30 Mbps, the network was the bottleneck for all configurations. Placement of the filtering code on the server side of the WAN allowed a near-doubling in performance over orig1, due to the filter’s halving of the WAN traffic. When the client/server WAN bandwidth was above 30 Mbps, computation associated with Java serialization and the filter code became the bottleneck. The restructured graph’s distribution of the filter across four processors provided a significant performance gain over the original graph.

With the original graph, the orig2 placement was faster than orig1 only because its WAN links were twice as fast. When computation was the bottleneck, orig2 and orig1 had equivalent performance. With the restructured graph, restruct2 was equivalent to restruct1 at low WAN bandwidths, but was faster at high WAN bandwidths because the filter in restruct1 shares a processor with its adjacent segment ship.

The overhead of computation for this simple application was unexpectedly high. Although restructuring helps by distributing some of the computation, further performance tuning of the Armada system would also improve performance. These improvements, however, would not change the nature of the results. That is, we could perhaps raise the throughput curves in Figure 5.5 and shift the transition point between network-bound and computation-bound operation, but the shape of the plots would remain roughly the same.
Figure 5.4: Execution time for the representative application with latency=2ms and bandwidth ranging from 0–100 Mbps. (a) shows the full range of the timings, (b) shows the range between 0 and 400 seconds. Each point is the mean of four independent trials.
5.2. REPRESENTATIVE APPLICATION

Figure 5.5: Measured effective throughput for the representative application with latency=2ms and bandwidth ranging from 0–100 Mbps. Each point is the mean of four independent trials.
5.3 File transfer and permutation

Application control over distribution of data to parallel disks is an important feature for a large class of data-intensive applications. For out-of-core applications especially, application control over data distribution benefits codes that require LU factorization [WGRW93], large FFTs [CN98], matrix transposition [CH97], and sorting [CC03]. Also, for large distributed databases, the layout of blocks on disk can have a significant impact on performance, and it is often beneficial to occasionally redistribute data to match application access patterns. Chang et al. describe such a system to manage data from remote-sensing satellites [CMA + 97].

The flexibility of the Armada framework allows the application or data-provider to describe arbitrary data distribution ships that, when connected in a graph, could efficiently permute data from one distributed system to another. In this section, we describe the design, implementation, and performance of such an application. In addition to permutation, this application demonstrates the ability to perform remote third-party transfers using Armada, and the automatic compression and decompression of data sent across a wide-area network.

5.3.1 Design

The copy application copies fixed-sized blocks of data from one distributed file to another, permuting blocks as necessary to match the distribution scheme of the output file. Figure 5.6 shows an Armada graph for this application.

For block I/O, we use a storage-interface ship (labeled file in Figure 5.7) that reads or writes blocks of byte data. We use two different structural ships to describe the layout of the data. For the data source, we use a stripe reader (labeled srd). For the data destination, we use a stripe writer (labeled swr). Although the structural ships could implement any type of block-distribution scheme, we chose standard block striping because of the simple implementation. In both cases, the structural ships convert global block indices to local block indices for their respective block I/O ships. We also insert a data compression ship (labeled cmp) and a decompression ship (labeled dec) into the data-flow path to reduce traffic across a wide-area network.

Driving the application is a single client-interface ship (labeled cpy) that executes on the client processor. It sends the initial transfer request to the I/O ships on the left, which then forward the requests through the Armada graph to the I/O ships on the right. The cpy ship then waits for acknowledgment that the transfer completed from the I/O ships on the left. Since no block-data actually travels through the client, the copy application provides third-party transfer capabilities, an important feature for the management of remote
5.3. FILE TRANSFER AND PERMUTATION

Figure 5.7: Restructured Armada graph for a distributed file copy application.

datasets.

The stripe reader and writer and the compression and decompression ships are all replicatable. For restructuring, we use the standard policies for swapping series-connected ships discussed in Section 3.2. That is, structural ships that distribute data (stripe writer) and data-reduction ships (the compression ship) prefer to move toward the data source. Structural ships that merge data (the stripe reader) and data-increasing ships (the decompression ship) prefer to move toward the data destination. Figure 5.7 shows the result of restructuring the original graph given these properties.

5.3.2 Implementation

The implementation of the copy application includes implementations of all the ships mentioned in the previous section. Here is a short summary of implementation details for each ship.

**BlockIOShip** (*file*) is a storage interface ship that reads or writes fixed-size blocks (our blocks are 32KB) of byte data from a local disk. The parameters passed into the ship by the dataset creator identify a single UNIX file and whether to read or write from the file.

**StripeReaderShip** (*srd*) represents a distributed input file as a logical sequence of blocks. It receives requests for individual block indices, or it receives a “transfer all” request, used for the copy application, that it passes to all output paths. An input parameter (called the *striping unit*) defines the number of consecutive blocks that were stored on each disk when the file was striped.

**StripeWriterShip** (*swr*) represents a distributed output file as a logical sequence of blocks. It forwards data blocks to an output path based on a global index in the logical file. An input parameter (called the *striping unit*) tells *StripeWriterShip* how many consecutive blocks to write to each output path when striping data.

**CompressionShip** (*cmp*) uses the *java.util.zip* package to convert a fixed-size byte block into a gzip-compressed block of data; gzip is GNU zip.

**DecompressionShip** (*dec*) uses the *java.util.zip* package to convert a gzip-compressed block of data into raw byte data.

---

1. [http://www.gzip.org](http://www.gzip.org)
CHAPTER 5. EVALUATION

Figure 5.8: Logical layout of data for the distributed file copy application. The figures show the first eight blocks.

**Copy** (cpy) is a client-interface ship that manages an SPTree object (see Section 2.5.2 about managing Armada graphs using Java) that represents the ships in the Armada graph. It constructs the Armada graph given command-line inputs for the source and destination datasets (XML descriptions of Armada graphs), tells Armada to restructure and deploy the combined graph, then it initiates a file copy by sending a single “transfer all” request to each `BlockIOShip` on the left side of the Armada graph.

### 5.3.3 Experiments

To investigate the performance of the copy operation, we created experiments for two types of applications. The first application (the baseline test) copies a single 600 MB file (using Armada) across a WAN. The second application copies a distributed file to another distributed file (with a different layout of the data) across a WAN. Figure 5.8 shows the logical block layout of the first eight blocks for the two distributed files. The input file is a distributed version of the original 600 MB file, striped to six I/O servers with a striping unit of 2 blocks. The application copies the input to a file distributed to four I/O servers using a striping unit of 4 blocks.

**Remote single file copy**

Figure 5.9 illustrates the virtual network topology used for the baseline application. The first LAN (labeled `lan0` in Figure 5.9) contains one processor for the client, one processor for the destination file (the I/O server), and one extra processor for Armada ships. The second LAN (labeled `lan1`) has two processors: one that hosts the I/O server for the source file, and an extra processor for Armada ships.
Figure 5.9: Virtual network topology for the baseline application. The client uses a single processor from lan0 to copy a single file from lan1 to a single file on lan0. The extra processors on each LAN are available as hosts for Armada ships.
Figure 5.10: Two domain-level placements for the single file copy application.

Figure 5.10 illustrates the Armada graph, with two different domain-level placements for the single file copy application. In the first placement, each Armada ship uses a separate processor. In the second placement, the Armada ships for compression and decompression each share a processor with the I/O server (to simulate a system with no additional processors available for Armada ships).

**Remote distributed file copy**

For the distributed file copy application, we configured a virtual network (shown in Figure 5.11) with 30 processors from two LANs. The client executes on a processor in lan0. The source file exists on six I/O servers on lan1, the destination file exists on four I/O servers on lan0. The remaining processors are available as potential hosts for Armada ships.

We ran experiments for three different domain-level placements for both the original (in Figure 5.12) and the restructured (in Figure 5.13) Armada graphs. In the first placement, we provide enough processors for each Armada ship to have its own host. In the second placement, we provide 2 processors for each client or data server (one for the interface ship, an one for other Armada ships). In the final placement, Armada ships share a host with one of the interface ships (no additional processors are available for Armada ships).

**5.3.4 Results and discussion**

Figures 5.14 and 5.15 show plots of the execution time and effective throughput for the single-file copy, the original Armada graph for distributed file copy, and restructured Armada graph for distributed file copy. For these plots, we assigned each Armada ship to a separate host. We calculated results of the remote transfers for WAN bandwidths ranging between 1 Mbps and 100 Mbps and network latency fixed at 2ms.

We expected better results from the restructured graph (especially for low bandwidths), because of the
compression; however, after further analysis, we noticed that we were getting less than 10% compression from the compression ships. The reason was the choice of dataset. Since we were also running experiments for the seismic application (Section 5.4) we reused the seismic data for the copy experiments. Unfortunately, seismic data does not compress well with gzip compression.

Despite the poor compression, we see significant improvement of the restructured Armada graph over the original graph. This improvement is due primarily to parallelizing the compression and decompression
Figure 5.12: Three different placements for the original distributed file copy application.

 ships. The original graph becomes compute-bound soon (at around 10 Mbps), since all data converges at the compression ship. The restructured graph becomes compute-bound near 30 Mbps and becomes completely limited by the decompression ships near 50 Mbps.
5.3. FILE TRANSFER AND PERMUTATION

Figure 5.13: Three different placements for the restructured distributed file copy application.
Figure 5.14: Execution time for the file copy application with latency=2ms and bandwidth ranging from 1–100 Mbps. (a) shows the full range of the timings, (b) shows the range between 0 and 1000 seconds. Each data point is the mean of five independent trials.
5.3. FILE TRANSFER AND PERMUTATION

Figure 5.15: Effective throughput of file copy application with latency=2ms and bandwidth ranging from 0–100 Mbps.

Figures 5.16 and 5.17 show plots of a similar application in which we remove the compression and decompression ships. We see that without the overhead of the compression, the restructured graph transfers data near the speed of the network. Even without compression, there is overhead associated with serialization of objects as they pass through the graph. The restructured graph adds enough parallelism to keep pace with the network, but the single file and the original graph become bound by Java serialization at around 50 Mbps.

The previous set of tests assume that the administrative domain assigns each ship to a separate processor; however, it is likely that only a few processors will be available. Figures 5.18 and 5.19 show execution time and effective throughput for the restructured Armada graphs (for the remote copy with compression) for the three different processor assignments from Figure 5.13. The three placements are no extra processors available for Armada (np=0 in Figures 5.18 and 5.19), one additional processor available for Armada ships (np=1), and enough extra processors for each ship to have its own host (np=all).

It is interesting to note that although one additional processor makes a large difference, more than one does not provide much benefit. In fact, above 70 Mbps, the single-node version slightly outperforms the multi-node version. We attribute this divergence to the difference between data transfer rates between processors within the same LAN (100 Mbps) and processes running on the same processor (3000 Mbps). This conclusion may seem counter-intuitive since the application is compute-bound above 50 Mbps, but consider the work being done by the ships. The compression/decompression ships are compute intensive, the storage interface ships are file I/O intensive, but the structural ships do little computational work. Placing the compression ship on the same processor with a structural ship achieves better performance if the compute time (per unit of data) on the structural ship is greater than the latency and transfer time required to send the data across the LAN.
Figure 5.16: Execution time of file copy applications without data compression.
Figure 5.17: Effective throughput of the distributed file copy application without data compression.
Figure 5.18: Execution time of restructured Armada graph for compressed data transfer using three different placements: a no additional processors for Armada ships (np=0), one additional processor for each input and output file (np=1), and enough processors for each Armada ship to have its own host (np=all).
Figure 5.19: Effective throughput of the restructured Armada graph for compressed data transfer using three different placements.
5.4 Remote seismic imaging

The goal of seismic imaging is to identify sub-surface geological structures that may contain oil. Seismic imaging is both computationally intensive (often requiring months to process a single dataset) and data intensive. A seismic dataset can be large, sometimes more than a terabyte in size, and is stored as a collection of files. For example, the SEG/EAGE Synthetic Seismic Dataset (SSD) \cite{ABN+94, OSV95} is a multi-terabyte, synthetically generated dataset consisting of several thousand files. Each file consists of recorded pressure waves, gathered by a set of receivers distributed across the surface and generated by a single acoustic source, also located on the surface. The dataset consists of data collected from the same receivers for thousands of different source positions. We refer to the data collected by a single receiver for a single source as a “trace,” and the file associated with a single source position as a “shot file.” Figure 5.20 shows a 2-D slice of a propagating acoustic wave from a single source (demonstrating the acquisition of data), and the calculated image of the SEG/EAGE overthrust model \cite{ABN+94}.

Post-stack migration \cite{Yil87} is a technique that significantly reduces the amount of processing by “stacking” (i.e., summing) co-located traces from each shot file before the computation phase. If the result of the post-stack computation shows promise, the scientist may perform the more computationally intensive pre-stack \cite{OOW+97} method that calculates an image for each shot file before combining the results into a single image.

Both applications are ideal for demonstrating the effectiveness of Armada: they require efficient access to large (potentially remote) datasets that require significant preprocessing, and they are both computationally intensive. For post-stack migration, an Armada graph can apply the data-reducing “stack” operator near
the data servers, reducing network traffic between the data and the computation. For pre-stack migration, Armada can overlap computation of the current image with reading and preprocessing the data for the next image, and combining and writing results from the previous image.

We modified a seismic imaging application called Salvo [OOW+97, OWO98] to use an Armada graph for its I/O. Salvo is a 3-D, finite-difference, pre-stack (and post-stack), depth imaging code designed to run on the Intel Paragon, Cray T3D, SGI Power Challenge, the IBM SP2, or a network of workstations. Salvo consists of portions written in C and FORTRAN, and it uses the Message Passing Interface (MPI) for communication between processors.

Salvo is unique because, in addition to allocating processors required for computation, Salvo allocates a few extra processors just for I/O. These processors make up the “I/O partition.” The I/O partition acts as a programmable interface between the system I/O nodes and the compute nodes. I/O requests that were previously made to the system I/O nodes with blocking UNIX calls are now made to the I/O partition with non-blocking communication calls. Collective I/O routines used by the I/O partition decrease the load on the system by combining many small read and write requests [SACR96], which also reduces the dependence on the caching system. Since the I/O partition is programmable, any pre- or post-processing of the data can also be done on the I/O partition. Some of the overhead operations required by seismic application include Fast Fourier Transforms (FFT), interpolation, coordinate transforms, de-convolution, and stacking.

In many ways, the I/O partition of Salvo and Armada are alike. They both allow I/O-specific application code to execute on processors other than the compute nodes. In fact, the success of the I/O partition inspired the development of the Armada framework. Armada provides a general approach to the techniques used in Salvo, making it easy to apply the same ideas to many applications.

In this section, we describe the design of an Armada graph to replace the I/O partition of Salvo.

5.4.1 Design

The processing structure of Salvo can be broken into a trace-input phase and computation phase. The trace-input phase reads the seismic dataset, stacks data (if a post-stack computation), transforms the data to the frequency domain, and distributes frequencies to processors for computation. The computation phase performs the imaging, or “migration,” of the trace data. This phase reads and distributes a velocity model (an estimation of the subsurface geology), computes an image, and writes the image to an output file.

Reading and processing traces

Before computing an image, Salvo first reads and pre-processes traces in a seismic dataset. For post-stack calculations we read every shot file and perform a vector sum (stack) on each co-located trace across each shot file. Next, we transform the time data to the frequency domain by performing a one-dimensional Fast-Fourier Transform (FFT) on each time trace. After the FFT, we distribute frequencies ($\omega$) to the compute nodes for computation. Salvo uses a 3D decomposition of the frequency data (shown in Figure 5.21) that distributes data across rows along the $x$-axis, columns along the $y$-axis, and towers along the $\omega$-axis. The distribution scheme identifies the tower for an individual trace based on its $x$ and $y$ position, and then evenly distributes the frequencies to processors along the $\omega$-axis.

Figure 5.22 shows the interaction between the compute partition and an Armada graph for the input phase of Salvo. Notice that we do not distribute frequencies to compute nodes using the Armada graph. Instead, we distribute entire frequency traces to the base node of each tower of the compute partition and then the Salvo code distributes the frequencies using MPI. We used this approach for two reasons. First,
Figure 5.21: Salvo uses a 3D decomposition of frequency data for computation. The distribution scheme identifies the tower for an individual trace based on its $x$ and $y$ position and then distributes the frequencies to processors along the $\omega$-axis.

it allows us to pass intact traces (whether in the time-domain or frequency-domain) throughout the entire Armada. This method of distributing frequency traces makes design of the ships simpler because we have to deal only with one type of object. Second, an efficient logarithmic scatter algorithm exists in MPI to perform the frequency distribution, and although we could re-implement the same functionality in the Armada ships, it was easier just to use the existing implementation. A side-effect of this approach is that only the base nodes of the compute partition (highlighted in Figure 5.22-a) interact with the Armada graph.

In our design, the graph from the data provider consists of a replicatable structural ship that manages distributed shot files (labeled $sdst$ in Figure 5.22-b) and storage interface ships (labeled $file$) that read seismic time traces from a disk. In addition to the raw trace data, a seismic trace includes a header that identifies (among other things) the spatial coordinates of the trace. Although the figure shows only one structural ship, one could imagine a far more complex distribution that layers distribution ships to access portions of the dataset that reside in different administrative domains, or portions of an individual shot file scattered to distributed disks.

For the application, we first prepend to the data-provider’s graph a processing ship (labeled $stk$ in Figure 5.22-b) that computes a vector sum of co-located traces. We designed a replicatable processing ship (labeled $fft$) that computes the Fourier transform of a time trace, a replicatable structural ship (labeled $tdst$) that distributes incoming traces to compute nodes, and finally a client-side interface (labeled $api$) that interacts directly with the base nodes of the compute partition to send requests and receive data through the Armada graph.

Figure 5.23 shows the steps used by Armada to restructure the initial graph. We show only cases where a swap comparison between two series-connected ships returned $true$ (see Section 3.2 for a more thorough explanation of restructuring series-connected ships). The $tdst$ and $fft$ ships were the first to swap, since $tdst$ preferred to move right to increase parallelism and we assigned no preferred direction to the $fft$ ship (it did not change the flow of data by a significant amount). The second swap (Figure 5.23-c) was between the stack ship $stk$ and the shot-distribution ship $sdst$. Since $stk$ is right-recursive, it became a manager ship $smg$
Figure 5.22: The Armada graph is used by the base nodes \((p_w = 0)\) of the compute partition: (a) highlights the base nodes of the compute partition, and (b) shows an Armada graph for reading, transforming, and distributing traces to those compute nodes.

I/O in the compute phase

Salvo computes an image by modeling the propagation of pressure waves through the earth, one depth step at a time. Unlike the trace-input phase, the migration (or compute) phase is primarily a computational operation; however, performing the I/O directly from the compute nodes may take 20\%–30\% of the overall run time for a typical production run [OWO98]. Our goal is to overlap computation and I/O for by moving the I/O portions of the compute phase into Armada graphs.

The compute phase has three primary operations for each depth step:

- read a plane of data from the velocity model (defined below),
- compute an image, and
- output resulting image.

Here, we describe the design of Armada graphs to read a plane of velocity data and to write the resulting image to a distributed file.

The velocity model is an estimation of the sub-surface geology represented as a three-dimensional array of floating point values that represent the speed of sound at a particular grid point. Since the velocity model typically represents a region much larger than the region represented by the trace data, the first preprocessing step is to extract only the necessary region from the velocity model. Also, if the grid spacing of the velocity...
model does not match that of the grid spacing for the image, we need to interpolate the grids of the velocity model. Once in the correct format, we distribute the velocity plane to the base nodes of the compute partition, where it is copied to processors along the $\omega$-axis. That is, each plane of the processor mesh contains a copy of the velocity plane. Like the input phase, only the base nodes of the compute partition interact with
5.4. REMOTE SEISMIC IMAGING

![Diagram of Armada graph for reading velocity data]

(a) Original graph.

![Restructured Armada graph]

(b) Restructured graph.

Figure 5.24: Armada graph for reading velocity data. (a) shows the original graph. (b) shows the restructured graph.

Armada.

Figure 5.24-a shows the original Armada graph for reading, extracting, interpolating, and distributing velocity data to the compute nodes. The graph from the data provider includes a distribution ship (labeled \(vdst\)) that describes the layout of the velocity model to disks and a storage interface ship (labeled \(vfile\)) that
reads a segment of the velocity data from a file. The application-specific portion of the graph includes the application-interface (api) that executes directly on the compute node, a replicatable distribution ship (cdst) that describes the decomposition of the velocity plane to the compute nodes, a replicatable interpolation ship (int), and a replicatable data extraction ship (ext).

Figure 5.24-b shows the restructured graph. Since all non-interface ships are replicatable, the restructuring process is rather simple. The data extraction ship ext moves to the right, since it reduces the data flow; the interpolation ship int moves to the left (it has no preference about direction); and the two structural ships vdst and cdst swap to provide end-to-end parallelism.

Each processor in the compute partition calculates an image for every frequency in its domain and then sums the results (using MPI_All_reduce) to the base nodes of the processor mesh, creating a single plane of image data. We then write the image data to a distributed file using the Armada graph shown in Figure 5.25.

The data-provider portion consists of a replicatable distribution ship idst that describes the layout of the image to multiple disks, and a storage interface ship ifile that writes a segment of the image plane to a file. The application-specific portion includes an application interface api that executes on the compute partition, and a replicatable structural ship imrg that merges data from the compute partition. The restructuring algorithm creates the graph in Figure 5.25-b by swapping idst and imgr.

While there are no processing ships in the output graph, the distribution and merge functions provide essential caching and reorganization of data to allow large blocks to be written to disk. The combination of the data distribution by idst and the merging of data by imrg form a synchronization step that gathers blocks of contiguous data before writing it out to a file. This is essentially the same functionality provided by two-phase methods that first gather and organize data from all the compute nodes before writing to a distributed file [TC96]. The difference between traditional two-phase methods and Armada is that Armada processes the data outside the compute nodes.

### 5.4.2 Implementation

Although we present designs for both the trace-input phase and the compute phase, we implemented an Armada graph only for the trace-input phase of the post-stack imaging application. Since the trace-input phase requires substantial processing and filtering of a remote dataset, it is ideal for evaluating the effectiveness of Armada.

For the trace-input phase, we implemented two dataset-specific ships (expected to come from the data provider): a storage-interface ship TraceFileShip, and a structural ship ShotDistributeShip.

**TraceFileShip (File)** reads seismic time traces stored in the Stanford Exploration Project (SEP) format. TraceFileShip reads blocks of floating-point values using the new I/O (NIO) package, introduced in version 1.4 of the Java Developer Kit (JDK). The NIO package allows bulk transfers [BDT01] (and automatic byte-swapping if necessary) of floating-point values. It is much improved from the regular I/O interface that reads non-byte values one value at a time.

**ShotDistributeShip (Sdst)** is a replicatable structural ship that distributes requests for traces to the proper TraceFileShip based on two input arrays that describe the layout of the data: one that defines the number of shot files for each output path, and one array that includes shot-id offsets for each of the output paths. One could layer multiple ShotDistributeShips in a hierarchical manner, for example to direct requests to distribution ships on each administrative domain.

---

2SEP Format Description for Numerical Data (http://research.seg.org/3dmodel/sepformat.html)

---

80
5.4. REMOTE SEISMIC IMAGING

We also designed five application-specific ships for the Salvo application.

**TraceStackShip** is a data-reducing, processing ship that vector-sums co-located time traces from different shot files. Shot requests that pass through the ship on their way toward the data servers identify

---

**Figure 5.25**: Armada graph for writing an image plane. (a) shows the original graph, (b) shows the restructured graph.
trace data that needs to be stacked its way to the application.

**RRTraceStackShip (Stk)** is a right-recursive extension of the TraceStackShip. The restructuring algorithm generates a new LRTraceStackShip (discussed next) after parallelizing RRTraceStackShip toward the data source.

**LRTraceStackShip (Smg)** is a left-replicable extension of the TraceStackShip. It acts as the manager ship for the right-recursive RRTraceStackShip.

**TraceFFTShip (FFT)** is a replicatable processing ship that computes a one-dimensional real-to-complex Fast Fourier Transform (FFT) of a seismic time trace. It uses a Java-enabled version of the FFTW [FJ98] library from MIT that calls FFTW through the Java native interface.

**TraceDistributeShip (Tdst)** is a replicatable structural ship that directs trace data to a client-interface ship. TraceDistributeShip calculates the output path based on input parameters for the global dimensions of the dataset and the processor mesh used by the application. It assumes a 3D block decomposition of the data.

**TraceAPIShip (API)** is a client-side interface ship that converts method calls to read a seismic dataset into shot requests that travel through the Armada graph. The client application instantiates this object directly on the client processor and registers the ship with the local Armada harbor (see Section 2.5.3 for information on the harbor interface and functionality).

### Application Interface

The original Salvo implementation consists of a combination of C and FORTRAN code. It uses FORTRAN for the computational kernel and C for all other system functions. The developers designed the code to be “modular” so that different implementations of a module could easily be plugged into Salvo. One such module reads, processes, and distributes seismic traces. For this work, we created a replacement module for Salvo that reads, processes, and distributes trace data using Armada. The rest of the Salvo application was unchanged.

Although Salvo could use the C++ version of the SPTree class (see Section 2.5.2) to access an Armada graph, we designed a higher-level interface (in the form of a C++ class hierarchy) that represents a distributed dataset and transparently manages an Armada graph. Figure 5.26 illustrates the C++ class hierarchy for distributed datasets. Figures 5.27, 5.28, and 5.29 show the class definitions of the Dataset, DistributedDataset, and TraceDataset classes. Below, we outline the process used by Salvo to access seismic trace data using Armada.

The user starts by constructing a TraceDataset object. Figure 5.29 shows the application interface for the TraceDataset. Constructor arguments include the MPI communicator of the processors involved, the processor mesh describing the processor configuration, and the URL for the data provider. Inside the constructor, the root node (node 0 in our case) connects to the data provider to get dimensions for the requested dataset (the number of traces along the x and y axes, and the number of frequencies in the ω-axis) and to get the Armada SP-tree that describes the physical layout of the data. For the current implementation, since we do not have a real data provider, the data provider URL is simply a path to an XML file that contains an Armada graph description of the input (we also hard-code the dimensions of the dataset). The root node then broadcasts the global dimensions of the dataset to other processors through the MPI communicator. The constructor for the DistributedDataset class calculates a local region for each processor. The
5.4. REMOTE SEISMIC IMAGING

![Class hierarchy diagram]

**Figure 5.26**: C++ class hierarchy for distributed datasets using Armada. The DistributedDataset class used a standard block-decomposition approach to partition the dataset to processors. TraceDataset and ModelDataset construct and manage an Armada graph using the SPTree C++ interface.

```cpp
class Dataset {
public:
    /** dims represents the size of the global dataset. */
    Dataset(const int dims[]);

    /** Returns global dimensions of the dataset */
    const int *getdims() const;
};
```

**Figure 5.27**: C++ declaration for the Dataset class.

The DistributedDataset class uses a standard 3D decomposition approach that evenly divides each of the three dimensions to processors based on a given three-dimensional processor mesh.

After defining the distributed dataset, the user calls the prependOp method of the TraceDataset class to explicitly prepend a stack operator and an FFT operator to the dataset, effectively prepending the Armada SP-tree from the data provider with the application-specific operators.

With the preprocessing of the dataset specified, the next step is to call the open method, which has the following tasks:

- On all processors, open installs a TraceAPIShip that later calls the read or readStacked method.

- On the root node, open does the following:
  - creates an SP-tree Leaf for the TraceDistributeShip, used to route traces to correct processors based on the data distribution scheme;
  - creates an SP-tree Leaf for each of the previously installed TraceAPIShips and places the leaves inside an SP-tree ParallelNode;
  - prepends the leaf for the TraceDistributeShip and the parallel node for the TraceAPIShips to the SP-tree; and finally,
The final step is to call the read or readStacked method of TraceDataset. Calling the read method generates an Armada request for trace data in a single shot file. The readStacked method generates a request for every shot file identified in the input parameters. Both methods immediately return an ArmadaRequest (see Section 2.5.2) that the application accesses to check the progress of the pending I/O, or wait for the I/O to complete. This “non-blocking” functionality allows the application to pre-fetch data for some future computation, thus overlapping computation and I/O. This overlap is especially important for pre-stack seismic imaging, an application that reads and processes each shot file independently.

5.4.3 Experiments

Using the Emulab network testbed, we ran experiments measuring the performance of our implementation of the trace-input phase of Salvo. Figure 5.30 shows the Emulab virtual network used for our experiments. The Salvo application used six processors in the first LAN (lan0). These processors represent the base nodes of the compute partition (see Figure 5.22-a). The compute nodes accessed a remote dataset distributed across five I/O servers in lan1 and five I/O servers in lan2. The remaining processors were available as potential hosts for Armada ships.

The dataset consisted of 10 shot files, each with roughly 25 MB of trace data (111 × 111 traces with 500 samples per trace). Note that our dataset was much smaller than a real dataset that may contain thousands of shot files. The main reason we chose a smaller dataset was that we did not have the physical resources to

---

```cpp
class DistributedDataset : public Dataset {
public:
    /** Calculates and stores dimensions for local dataset */
    DistributedDataset(const MPI_Comm &comm,
                       const int dims[3],
                       const int pmesh[3]);

    /** Returns the processor mesh. */
    const int *getpmesh() const;

    /** Returns processor coordinates. */
    const int *getcoords() const;

    /** Returns dimensions of local array. */
    const int *getlocalDims() const;

    /** Returns local offsets to distributed array. */
    const int *getlocalOffsets();

    /** Returns the size (in bytes) of local dataset */
    int getlocalSize() const;
};
```

Figure 5.28: C++ declaration for the DistributedDataset class.

---

- restructures the graph, computes a placement, and deploys the ships in the SP-tree to processors in the grid.
5.4. REMOTE SEISMIC IMAGING

```cpp
class TraceDataset : public DistributedDataset {

public:
    /** Constructs an Armada-enabled distributed dataset for traces */
    TraceDataset(const MPI_Comm &comm,
                 const int pmesh[3],
                 const string &dataProvider);

    /** Returns the data provider URL */
    const string& getdataProvider() const;

    /** Returns the Java virtual machine */
    const JVM& getarmadaVM() const;

    /** Returns a reference to the SPTree */
    SPTree& getspTree();

    /** Prepend an operator to the dataset (i.e., the SPTree) */
    void prependOp(const Operation &op);

    /** Open the dataset (deploys an Armada graph) */
    void open();

    /** Read a single shot of traces */
    const ArmadaRequest read(float dest[],
                              const int shotID);

    /** Read stacked traces (for given shotIDs) */
    const ArmadaRequest readStacked(float dest[],
                                     const int numShots,
                                     const int shotIDs[]);

    /** close the dataset */
    void close();
};
```

Figure 5.29: C++ declaration for the TraceDataset class.

store terabytes of data for our experiments. For this experiment, however, the smaller dataset was sufficient to demonstrate the effectiveness of our approach.

Figure 5.31 shows the original and restructuring Armada graphs (illustrating the partitioning to domains assigned by the placement algorithm) for our experiments. In these experiments, we assumed that there were enough processors so that every Armada ship had its own host. Notice, for the original graph, that the placement algorithm chose to place the FFT and the stack operators on a LAN with the data servers; this placement was correct, given our observations from the manually configured experiments with the representative
Figure 5.30: Virtual network topology used to measure the performance of the trace-input phase of Salvo. The Salvo application used six processors on lan0 to access data distributed to five disks on lan1 and five disks on lan2. The remaining processors on each LAN were available as hosts for Armada ships.
### Table 5.2: Average size of data output queues lengths for selected ships from the restructured Armada graph for different WAN bandwidths.

<table>
<thead>
<tr>
<th></th>
<th>1Mbps</th>
<th>3Mbps</th>
<th>5Mbps</th>
<th>&gt;10Mbps</th>
</tr>
</thead>
<tbody>
<tr>
<td>smg on lan1</td>
<td>9.84</td>
<td>6.01</td>
<td>0.26</td>
<td>0</td>
</tr>
<tr>
<td>sdst on lan1</td>
<td>9.66</td>
<td>4.78</td>
<td>4.59</td>
<td>4.89</td>
</tr>
</tbody>
</table>

application (Section 5.2.1). Also, because the restructure algorithm parallelized the recursive stack operator as far as possible, the right-most stack ships (i.e., the ones adjacent to the TraceFileShips) performed no processing on the data. They were essentially “no-op” ships. The most significant data-reduction (a five-to-one reduction) occurred at the two right-most stack manager ships, because they received co-located traces from each of the five connected I/O servers.

#### 5.4.4 Results and discussion

Figures 5.32 and 5.33 compare the performance of the original and restructured Armada graphs as the WAN bandwidth increases from 1 Mbps to 100 Mbps.

For the restructured graph, we see a dramatic barrier at 5 Mbps where the application changes from being network bound to being compute bound. Since an FFT is a fairly compute-intensive operation, our first guess placed the bottleneck at the ships performing the FFTs; however, after further analysis of the data output queues (see Section 2.5.1 about data queues in Armada ships), we observed the bottleneck to be stacking the data on the data-server LANs. The FFT is, in fact, very fast (we used a Java-enabled version of the FFTW library [FJ98]). The stack operator has to de-serialize and perform a vector sum for each incoming trace, but the FFT only works on the resulting sum. Also, the stack ship acts as a synchronization point for the application, since co-located traces from all inputs must arrive before passing the result on to the FFT ship. For traces that arrive out-of-order, the stack ship caches partial sums and forward the result only after processing all co-located traces.

Table 5.2 shows the average size of data output queues (taken over the full running time of the application with samples every 1 second) for selected ships from the restructured Armada graph. Below 5 Mbps, the output queues of the smg ships on lan1 and lan2 of the restructured graphs fills up, forcing them to wait for data to transfer across the WAN. Above 5 Mbps, smg’s queue remains empty (implying the network is no longer the bottleneck), but sdst’s queue remains partially full because it has to wait for the stack operation to complete.

A similar analysis for the original graph showed the link between the sdst ship on lan1 and the sdst ship on lan2 to be the bottleneck up to 25 Mbps. For bandwidths greater than 25 Mbps, the stack ship was, again, the bottleneck.

Figures 5.34 and 5.35 plot the execution time and effective throughput of the restructured Armada graph for different latencies. These plots demonstrate that pipelining large data transfers effectively negates the effect of latency over a wide-area network. As expected, the different experiments had nearly identical running times. This feature is particularly important; although technology limits the amount of available bandwidth, the speed of light makes latency across large geographic distances unavoidable.
Figure 5.31: Original and restructured graphs (illustrating LAN placement) for reading seismic trace data.
Figure 5.32: Execution time for the original and restructured Armada graphs for reading seismic trace data. Latency is fixed at 2ms, bandwidths range from 0–100 Mbps. (a) shows the full range of the timings, (b) shows the range between 0 and 300 seconds. Each data point is the mean of five independent trials.
Figure 5.33: Effective throughput for the original and restructured Armada graphs for reading seismic trace data.
Figure 5.34: Execution time for seismic application with different WAN latencies.
Figure 5.35: Throughput of seismic application for different WAN latencies.
5.5. REMOTE ANALYSIS OF FMRI DATA

Figure 5.36: Feature-space clustering attempts to identify regions with similar patterns of activation within an fMRI time series. Shaded areas in the resulting image (right) identify areas of activation.

5.5 Remote analysis of fMRI data

Recent advances in technology for high resolution of functional magnetic imaging (fMRI) is increasing the interest in the development of data analysis methods for computational neuroscience. Because of the prohibitive costs of fMRI experimentation and the infrastructure required to acquire the data, research in fMRI analysis is often restricted to large, well funded, medical institutions. To address this problem, the Center for Cognitive Neuroscience at Dartmouth College founded the Functional Magnetic Resonance Imaging Data Center (fMRIDC), a facility for sharing results and data of fMRI studies with a large community of academic and industrial researchers [HGK+01]. The fMRIDC hosts one of the largest repositories of fMRI brain data in the world, with current capacity for nearly 80 TB of data.

In addition to providing access to fMRI data, the fMRIDC recently acquired a large compute engine for local analysis of brain data from the repository. This compute engine provides a valuable capability for remote academic researchers that lack the storage capacity or computation required for analysis. The compute engine also provides a way for remote researchers to investigate large portions of the data (through analysis) before downloading for private computation.

One particularly interesting technique that has potential for remote analysis is a method called feature-space clustering [GHLR01]. Clustering techniques use statistical analysis to identify regions with similar patterns of activation within an fMRI time series. Figure 5.36 shows the result of a feature-space extraction application. Shaded portions of the brain image on the right indicate regions of interest. This section describes an active effort at Dartmouth College to use Armada for remote analysis of fMRI brain images using feature-space clustering. Since this project is currently in the design stage, we have no experimental results. Presented below is our proposed design of an Armada graph for analysis of fMRI data.

5.5.1 Design

A brain image dataset consists of a time-sequence of brain images. Each image is a 3-D spatial volume of voxels that contain a scalar value associated with the activity of a particular portion of the brain at a particular point in time. The feature-space clustering application has two major portions: preprocessing and
Preprocessing the time-series data requires statistical analysis of the brain images, and it may include operations such as spatial smoothing or motion correction (i.e., image registration) to transform each image in the series to a common geometric basis. Statistical analysis starts with voxel-wise calculations of the mean across the time series. The application uses results of the mean to calculate a mask that identifies voxels that might be active. The next step is to calculate variance, skewness, kurtosis, level crossings, linear trend, and outliers. Most of these calculations take the masked image set, the mean, and the variance as input arguments. The final preprocessing step calculates statistics based on Fourier analysis (requires an FFT) of the time series for each voxel.

Although the current implementation of the clustering application at the fMRIDC is a serial application, the operations for both statistics calculation and clustering are single-voxel calculations, making the application highly parallelizable. Figure 5.37 shows our proposed Armada graph for a parallelized version of the clustering application.

The graph in Figure 5.37 consists of I/O servers (labeled file), a structural ship that describes the layout of the data (labeled idst), a structural ship that describes the distribution of voxels to compute nodes (labeled vdst), and nine ships for statistics processing (labeled avg, msk, var, skw, krt, crx, trn, out, and fft). The file ships read blocks of voxel data in parallel (e.g., if a block contained 1000 voxels, each would read the first 1000 voxels at the same time). The idst ship merges data from the I/O servers (effectively a transpose), creating time-sequenced vectors of co-located voxels. Since each I/O server reads voxels in the same order, co-located voxels from each of the image files arrive at idst in the same order, allowing the construction of complete voxel-vectors in an enumerated fashion. That is all voxels from position 1 arrive first (idst creates a voxel vector for position 1), followed by voxels from position 2, and so forth. Producing voxel vectors in this manner allows pipelined processing of independent vectors as they pass through the statistics-processing ships. The voxel-distribution ship directs voxel vectors to the appropriate compute nodes for processing.

Figure 5.38 shows the restructured Armada graph. We made all processing ships left-replicable so we could parallelize the statistics processing to match the cluster application. The two structural ships swap, allowing the distribution of voxels to occur just after reading an image from disk. This optimization step...
5.5. REMOTE ANALYSIS OF FMRI DATA

reduces the load on the *idst* ships that transpose the time-series data as it arrives.

For remote access, the placement algorithm partitions the graph into domains that place the data-reducing processing ship (*msk*) in the same LAN as the data servers. Figure 5.39 shows a partitioning of the restructured graph into two administrative domains. Although we show the boundary to be just left of the *msk* ship, in fact it could be anywhere to the left of *msk*. An intelligent partitioning algorithm would decide based on available computing power where to make the cut (see Section 6.3.2).

5.5.2 Conclusion

The fMRIDC provides a unique opportunity for academic researchers to analyze data from one of the largest collections of fMRI data in the world. The ability to use Armada for remote analysis significantly increases the viability of such experiments for researchers with limited storage, network, or compute capabilities. In addition, the Armada framework makes the process of designing efficient, high-throughput experiments simpler for the scientist.
Figure 5.39: Partitioning of the restructured Armada graph for fMRI analysis into two administrative domains (labeled $lan_0$ and $lan_1$).
Chapter 6

Conclusions and Future Work

The efforts to develop high-performance computational grid applications lead to many new research challenges in the field of distributed computing. One of the greatest of these challenges is providing an efficient I/O infrastructure. Previous development efforts suggest that application control is the key to achieving high-performance I/O. To this end, we designed the Armada parallel I/O system with three key features: application flexibility, support for remote execution of application code, and performance enhancements designed to improve remote access to distributed data.

6.1 Contributions

In this dissertation, we present the Armada I/O system: a new way to develop high-performance data-intensive computational grid applications. In Armada, the application (or application library) controls virtually all aspects of the I/O system through a distributed graph of application components, including the application interface, optimization policies like caching and prefetching, and remote filtering of datasets. Armada supports remote execution by allowing the different components to execute on processors used by the client application, on processors used by storage servers, or on intermediate processors in the network. The primary contributions of this dissertation are the following:

- a flexible framework, based on a data-flow programming model, that allows the application programmer and the dataset provider to deploy a network of application-specific and dataset-specific functionality across the grid;

- an algorithm to restructure a data-flow application graph to improve data flow across a wide-area network, based on programmer- and user-assigned properties that describe the behavior of the nodes within the graph;

- a hierarchical graph-partitioning scheme that leverages existing software to decide where to place individual application components in a way that benefits the application and abides by allocation policies set by individual administrative domains; and

- an evaluation of the I/O performance of a variety of applications using Armada.
6.2 Limitations of Armada

Although we designed Armada to be flexible, there are a number of limitations that prevent some applications from performing well using Armada. This section describes a few of these limitations.

6.2.1 Applications need to make large requests

Armada works well for applications that have detailed knowledge of their access patterns. In particular, applications that make large requests for data take advantage of the latency-tolerant features of the data-flow design. Applications that use the standard POSIX style of accessing small (random) parts of a dataset will not work well in Armada (or any other wide-area environment) because of the network latency involved in processing many small requests.

6.2.2 Limitations in the graph representation

We chose to represent a distributed data-flow graph as a series-parallel tree (SP-tree) because it is syntactically easy to describe and easy to manipulate internally. Although an SP-tree is a convenient initial representation, there may be cases where better performance could be found in a graph that is not an SP-DAG (recall that an SP-tree is also an SP-DAG). We did not investigate these cases. Refer to Section 3.2.2 for examples of configurations not allowed by Armada.

6.3 Future work

In this section, we discuss three potential directions for future work: constructing a complete I/O system using the Armada approach, extensions of the placement algorithm, and adapting Armada to work well on clusters and supercomputers.

6.3.1 Building a complete I/O system for computational grids

Armada is not a fully-featured I/O system. In particular, it lacks support for data management, security, and fault tolerance, which are necessary in a production grid system. It is important to note that there are active research groups working in each of these areas, but there is still much debate over proper solutions. Here, we discuss some of the issues involved with building a complete Armada-enabled system.

Data-manager support

A data manager for computational grid datasets (often called a data grid) provides a data management environment in which data can be manipulated across resources connected by wide-area networks. In our paper *Summary of Existing and Developing Data Grids*\(^1\) [Old01], we identify some of the emerging approaches for dealing with three fundamental requirements of data grids:

- a global name space,
- persistence (or consistency of metadata used for the global name space), and

---

\(^1\)This paper was joint work with several members of the Data Access Group of the Global Grid Forum.
6.3. FUTURE WORK

- common access mechanisms for data discovery and access.

The most common way to represent a global name space is as a logical name space that maps a global name in the logical space to a physical file name used at a particular site. The differences between data grids occur in the generality of the logical name space (such as whether they have a true hierarchy of folders for organizing the data logically) and the degree of independence of the logical hierarchy from the local physical file name hierarchy (whether the physical files can reside at arbitrary path locations, and whether they can have different names at each physical site).

An Armada-enabled data manager would map global names to Armada graph descriptions. The Armada graph structure provides an abstraction to data users that allows the description of arbitrary layouts of distributed datasets, despite the storage format. An Armada graph description could describe a traditional striped or linear data structure, or it could describe a distributed collection of legacy files, stored on UNIX file system, databases, or the like. For derived datasets, the Armada graph may also include components that describe how to construct the dataset, or how to access portions of a pre-generated dataset from a cache.

Persistence and consistency of metadata remains the hardest problem for most data grids. A straightforward way to guarantee consistency is to have the data grid own the data. This requires all manipulations of the data within the grid to be done through data-grid interfaces, making it possible to guarantee consistency of the global name space. Given a consistent name space, it is then possible to build a persistent data-management environment. The metadata for an Armada system is the XML representation of Armada graphs.

Access APIs need to support all manipulations that grid applications require. At a minimum, this requirement implies the need to support UNIX File System semantics for accessing any data object within the grid. A second API is needed for data discovery. Here there are multiple competing systems for the syntax (XML), query language (XQuery, SDLIP, LDAP), and metadata delivery (SOAP, SDLIP, WSDL). There are also multiple choices for the information repository characterization (OMG Common Warehouse Metamodel, OAI, UDDI). The Armada system provides support for access APIs through implementations of the client and server interface ships. A complete system needs implementations of client and server interface ships that match all required APIs and heterogeneous storage systems.

Security

Security in a production grid system is a critical component to its success. In addition to traditional authentication and authorization, a grid system requires safe mechanisms to allow application code to execute on remote processors. System administrators are unlikely to make resources available unless they have the ability to set and enforce security policies for accessing CPU, memory, and storage resources. Although Java allows some control over these policies and provides language-based protection against unauthorized memory access, it does not allow object references to be revoked. As part of a security course at Dartmouth College, we designed and created an initial prototype for a secure harbor implementation (recall from Section 2.5.3 that a harbor provides an execution environment for Armada ships) that uses JKernel [HCC+08] to provide capability-based access to resources available to Armada ships. The essential components of the secure harbor (called JKHarbor) include a security manager and a resource manager.

The security manager is responsible for providing a secure execution environment for application ships. Before installing an untrusted application ship on a harbor, the security manager authenticates the code for the ship and the user wishing to install the ship, and it authorizes use of the host resources based on the user identity and the security policies of the host. After identifying the user, the security manager installs the ship
inside a protected domain, known as a “sandbox” [WLAG93]. Once inside the sandbox, access to resources outside the domain are strictly controlled by the resource manager.

The resource manager provides “capability-based” access [SS75, HCC+98] to system resources (the CPU, network, storage, and memory) outside of the protected domain of the sandbox, and it monitors per-ship and overall usage of the resources available on the host. Capabilities provide the ship with a revocable “ticket” that enables cross-domain access to a resource while the ticket is valid. This scheme prevents abuse of system resources by allowing the harbor to invalidate a ticket if the ship violates a resource usage policy or exceeds a consumption limit assigned to the capability.

The resource manager also monitors and publishes information about the resource consumption of the individual ships and the system as a whole. The harbor requires this information to enforce resource consumption policies, but we make the information available so external programs can use it for their own needs.

Although the design of JKHarbor provided valuable insight into the security needs for a production-ready Armada system, we never completed the implementation, since our primary research focus is not security.

Fault tolerance and adaptability

One of the most difficult challenges in developing efficient computational grid applications is dealing with the dynamic nature of the grid. For long-running applications it is unlikely that resources allocated for the application remain stable with the same performance characteristics throughout the life of the application. Here we suggest changes to the Armada system to deal with three types of faults: random infrastructure failures, performance faults, and faults due to revoked resources. We describe each type of fault in turn.

A random infrastructure failure occurs when either a processor, local storage system, or portion of the network unexpectedly ceases to function. The most obvious way to handle this type of fault is to save application state (checkpoint) periodically, and “roll back” to a previous state when an infrastructure failure occurs. The issue here is where to save state and how to recover data. Atchley et al. [ASP+02] suggest checkpointing data using a “Network Storage Stack,” which is an abstraction that exposes temporary network storage to distributed applications. There are also interesting projects investigating diskless checkpointing [PKD97, PL94]. These methods use memory of neighboring processors to store checkpoint data.

A performance fault is not necessarily a failure, but rather a dramatic change in the system performance, often due to system load, that causes the application to perform poorly. To deal with such faults, we need to incorporate a system monitor and a dynamic placement scheme that occasionally reassigns all or a portion of the Armada graph to different hosts in the network.

The revocation fault occurs when the host of one or more Armada ships revokes a resource (such as a CPU or access to a storage system). There are many reasons why this might happen; for example, the host system may need to shutdown, or another higher-priority job might need the resource. In the event of a resource revocation, the host system should notify the Armada system, allowing it to save the state of the affected ships, free the resource, and reassign the ship to another host. We could treat this type of fault similar to a performance fault, except the system would force a reassignment of the affected ships.

6.3.2 Improving placement

Although experiments show that our placement scheme provides some improvement in performance, there are many ways to improve the way that Armada places application components. For example, our scheme
does not incorporate information about processor availability on different domains, system heterogeneity, or the requirements of individual application components. There are also limitations in the graph-partitioning software itself. Kumar et al. demonstrate that the traditional approach to graph partitioning is inadequate for heterogeneous environments [KDB02]. Although there is a significant amount of current work addressing this problem, there are still many unresolved issues.

### 6.3.3 Armada on a cluster of workstations

We designed Armada specifically for wide-area distributed networks, but there are many ways in which the Armada approach would also benefit applications running in a tightly-connected cluster or a parallel supercomputer. For example, the ability to easily describe and execute data permutations is a useful feature for parallel machines. Also, the end-to-end parallelism gained by restructuring Armada graphs may provide more of a benefit for local parallel machines than to remote machines, since the single network interface connecting the two remote machines is typically the bottleneck. Parallel machines often have multiple network paths allowing data to travel from source to destination efficiently.

Adapting Armada to work well on a cluster or parallel machine requires significant modifications. For example, we used Java because mobile code is easy, and we expect the network (not computation) to be the primary bottleneck for most wide-area applications. On a tightly-connected machine, the overhead of Java would be a substantial factor. An efficient implementation of Armada for clusters would likely use C or C++ for both the harbors and the ships. There would also need to be significant tuning in Armada’s network layer, and perhaps a way to efficiently “join” two ships assigned to the same processor, so they could communicate without using the network layer.

### 6.4 Conclusion

The trend to develop applications for wide-area computing environments (computational grids) is hindered by several performance-related challenges. Unlike traditional parallel computers, grid applications execute in environments with unavoidable latency, low bandwidth, and unpredictable behavior. This dissertation describes a data-flow-based solution in which the application programmer and the dataset provider describe and deploy a network of application-specific and dataset-specific functionality across the grid. Our system then restructures the graph to distribute data flow and computation throughout the graph, and it places individual components using a scheme that is both beneficial to the application and considerate of administrative-domain allocation policies.

Our approach demonstrates that a flexible design along with careful attention to data-flow performance can lead to efficient I/O for grid applications. Our performance results show that the data-flow model does an exceptional job of hiding network latency inherent in grid computing. Applications using Armada perform well in low-bandwidth environments because restructured graphs allow an effective placement of Armada ships. Applications using Armada also perform well in high-bandwidth environments because restructured graphs often include end-to-end parallelism.

---

2 The Global Grid Forum has a Scheduling and Resource Management Area (http://www.gridforum.org/3_SRM/srm.htm), dedicated to solving the placement problem.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


