SUPPORT FOR DATA MANAGEMENT IN OBJECT-ORIENTED DATA-PARALLEL SCIENTIFIC AND ENGINEERING SIMULATIONS

Jacob K. Gotwals

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

August 1996

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

_____

Dennis B. Gannon, Ph.D.

_____

Randall Bramley, Ph.D.

Doctoral Committee

_____

Franklin Prosser, Ph.D.

_____

Dirk Van Gucht, Ph.D.

_____

Robert Glassey, Ph.D.

August 6, 1996

Copyright © 1996

Jacob K. Gotwals

ALL RIGHTS RESERVED

Dedicated to My Family

Acknowledgments

I would like to thank my advisor Professor Dennis Gannon for all the support, encouragement, and advice he provided during my years at Indiana University. He has created a research group that provides an excellent academic environment in which his students can mature as they pursue their research interests. I would also like to thank the rest of my thesis committee: Professor Randall Bramley, for many interesting insights into academia and advice about Scientific Computing; Professor Robert Glassey, for some enjoyable discussions related to Numerical Analysis; Professor Franklin Prosser, for an excellent Hardware class and some good advice about giving presentations; and Professor Dirk Van Gucht, for many interesting discussions and much advice and encouragement related to the Database field. Thanks to Professor Peter Ortoleva in the Chemistry Department, for his support during my first two years at Indiana and for many interesting discussions. Thanks to the Advanced Research Projects Agency and the National Science Foundation for indirectly funding much of my research.

I would like to thank my parents Clayton K. Gotwals M.D. and Emily W. Gotwals for everything, my siblings Anne, John, and Sarah Gotwals for lots of good times on many weekend trips home to Cincinnati, and the rest of my family for being there.

There were several friends with whom I spent a significant amount of time during my years at Indiana. In chronological order: thanks to Siew-Khim Tan for (among other things) lots of good times in Bloomington and the S.F. Bay Area and a great guided tour of Malaysia and Singapore; thanks to Jon Kahrs for some good meteorological discussions and many good times in Indiana and the Bay Area; thanks to (Clark)

(Changxing) Qin and his family for lots of great food and many good times; thanks to my office mates Hui (Jeff) Ma and Suresh Srinivas for good times, for great discussions of all sorts of interesting things, and for helping me hunt for jobs; and thanks to Cathy Cowan, Phil Candee and Aaron Brubaker for many movies and good times in Bloomington.

Thanks to all of the great friends who have made Bloomington feel like a home, not just a place to study. In alphabetical order, thanks to: Lindsay Alexander, Curt Anderson, Shelly Barnard, Phil Delphey, Laurie Ervin, Lucia Fouts, Laurie Forsman, Cathy Hamaker, Madeline Hunt, Karen Jewell, John Jones, Tim Jones, Ann Kamman, Stephanie Koutek, Jennifer Lackey, Julianne Marley, David May, Vladimir Menkov, Susie Reitbauer, Juan Villacis, Heather Whipple, and Jim Wooley.

Thanks to the people of the Unitarian Universalist Church of Bloomington for being there, and thanks to the staff, especially Mary Ann Macklin, for helping to create an environment in which students feel welcome.

Thanks to the people I met through Extreme Computing, for some good times and good discussions (especially at conferences!): Pete Beckman, Francois Bodin, Shridar Diwan, Liz Johnson, Kate Keahey, Jenq Kuen Lee, Bernd Mohr, Sekhar Sarukkai, Suresh Srinivas, Neel Sundaresan, Juan Villacis, Beata Winnicka, and Shelby Yang.

Thanks to the past and present members of the administrative and systems staff of the I.U. C.S. Department. Special thanks to Bruce (Shing Shong) Shei of the systems staff, and Pam Larson, Nat McKamey and Julia Fisher of the administrative staff.

Abstract

Many data-parallel scientific and engineering simulations require the coordinated movement of large sets of data between multiple storage devices and multiple processors on distributed memory parallel machines. These data management requirements are not met by conventional file systems, so parallel file systems have been developed to support I/O in these programs. Current parallel file systems support I/O on simple data structures such as distributed rectangular arrays of primitive data types; however they do not provide adequate support for I/O on the complex distributed data structures often found in object-oriented parallel programs. This dissertation describes how class libraries can be used to extend the functionality of both parallel file systems and object-oriented database management systems to support the complex I/O requirements of object-oriented data-parallel simulations. The contributions of this dissertation are summarized below:

Several abstract models of parallel file systems are developed; the experimentally measured performance of two commercial parallel file systems is explained in the context of those models. This analysis should help application and library developers obtain high I/O performance from parallel file systems.

Three new abstractions for data management within the file I/O model are proposed. These abstractions can be used to extend parallel file systems to meet the special I/O requirements of object-oriented data-parallel simulations. Implementations of each abstraction are described.

A new abstraction for data management within the persistent data model is proposed. This abstraction can be implemented on object oriented database systems using language-specific class libraries; an implementation of this abstraction is described as well. This work shows how persistence can be used to support several common data management tasks in object-oriented simulations.

Finally, the file I/O model and the persistent data model of data management are compared, in the context of implementations of systems supporting these models for I/O in object-oriented parallel simulation programs. A number of metrics are applied to these systems to perform the comparison: performance, space requirements, productivity effects, ease of implementation, maintainability, and portability.

Brief Contents

Acknowledgments . . . . . *v*

Abstract . . . . . *vii*

1 Introduction **1**

2 I/O Performance in Parallel File Systems **16**

3 Parallel Filestreams **42**

4 Object-Parallel Filestreams **56**

5 Distributed Array Streams **70**

6 Persistent Collections **90**

7 Analysis and Comparison **105**

8 Conclusion **122**

A Introduction to Parallelism **128**

B Introduction to Database Management Systems **134**

Bibliography *146*

Contents

Acknowledgments . . . . . *v*

Abstract . . . . . *vii*

1 Introduction **1**

1.1 Overview 1

1.1.1 Scientific and Engineering Simulations 1

1.1.2 Data Management in Simulations 2

1.2 Data Management Tasks in Parallel Scientific and Engineering Simulations . . 3

1.2.1 Entire Database Transfer Tasks 3

1.2.2 Concurrent Data Access Tasks 4

1.2.3 Tasks Requiring Access to Large Databases 5

1.3 Two Models for Systems Supporting Data Management Tasks 5

1.3.1 File I/O Model 6

1.3.2 Persistent Data Model 7

1.4 Contributions 9

1.5 Related Work 11

1.5.1 Parallel File System Performance 12

1.5.2 Support for File I/O in Parallel Programs 12

1.5.3 Support for Persistent Data in Parallel Programs 13

1.6 Summary and Dissertation Outline 14

2 I/O Performance in Parallel File Systems. 16

2.1 Introduction to Parallel File Systems 17

2.1.1 An Example Program 17

2.1.2 Performance Measures 22

2.2 Models Explaining the Impact of Buffering and Virtual Memory on PFS Performance 24

2.2.1 Unlimited Compute Node Memory and No I/O System Memory 25

2.2.2 Finite Physical Memory Buffer in the I/O System 25

2.2.3 Virtual Memory on Compute and I/O Nodes 27

2.3 Experimentally Measured Parallel I/O Performance 29

2.3.1 Intel Paragon 30

2.3.2 Thinking Machines CM-5 37

2.4 I/O Optimizations 39

2.5 Summary 40

3 Parallel Filestreams 42

3.1 Goals 43

3.2 Design 44

3.3 Implementation 49

3.3.1 Implementation of Insertion and Extraction Functions 49

3.3.2 Internal Architecture 51

3.3.3 Lessons Learned 54

3.4 Summary 54

4 Object-Parallel Filestreams 56

4.1 Goals 56

4.2 Design 58

4.3 Implementation 64

4.3.1 Interface 64
4.3.2 Internal Architecture 68
4.4 Summary 69

5 Distributed Array Streams. 70
5.1 Goals 70
5.2 Design 73
5.3 Implementation 77
5.3.1 Interface 78
5.3.2 Internal Architecture 83
5.4 Summary 89

6 Persistent Collections 90
6.1 Goals 91
6.2 Design 93
6.3 Implementation 93
6.3.1 Interface 93
6.3.2 Internal Architecture 101
6.3.3 Lessons Learned 103
6.4 Summary 104

7 Analysis and Comparison 105
7.1 Performance 105
7.1.1 Distributed Array Streams 106
7.1.2 Persistent Collections 108
7.1.3 Performance Summary 114
7.2 Storage Requirements 114
7.2.1 Distributed Array Streams 114
7.2.2 Persistent Collections 115

7.3 Productivity Effects 115
7.3.1 Distributed Array Streams 116
7.3.2 Persistent Collections 116
7.4 Ease of Implementation 117
7.5 Maintainability 118
7.6 Portability 118
7.7 Limitations of this Study 119
7.8 Summary 120

8 Conclusion 122
8.1 Contributions 122
8.2 Directions for Future Research 125
8.2.1 File I/O Model 125
8.2.2 Persistent Data Model 126
8.3 Conclusion 127

A Introduction to Parallelism 128
A.1 Parallel Hardware 128
A.2 Parallel Programming Systems 130
A.2.1 Virtual Machines 130
A.2.2 Levels At Which Parallelism Can Be Extracted 131
A.2.3 Data-Parallelism 132
A.2.4 Task Parallelism 133

B Introduction to Database Management Systems 134
B.1 Overview 134
B.2 Data Models 136
B.2.1 The Entity-Relationship Data Model 136

B.2.2 The Relational Data Model 138
B.2.3 The Object Data Model 140
B.3 DBMS Client-Server Architectures 142
B.3.1 One-Representation 142
B.3.2 Two-Representation 144
B.4 Families of Currently Available DBMS 144

Bibliography *146*

List of Illustrations

1 Introduction. 1

Figure 1.1: A Range Of I/O Mechanisms 11

2 I/O Performance in Parallel File Systems. 16

Figure 2.1: Architectural Elements of Distributed Memory I/O Systems. 17

Figure 2.2: Using a Parallel File System for Input 18

Figure 2.3: Using a Parallel File System for Output 21

Figure 2.4: Three Parallel File System Models 24

Figure 2.5: Comparing Output Performance With and Without Virtual Memory on the I/O System 27

Figure 2.6: Comparing Paging Algorithms for the Model with Virtual Memory on the I/O Nodes 28

Figure 2.7: Paragon Input Performance. 32

Figure 2.8: Comparing the Paragon's Application and Actual Output Performance 33

Figure 2.9: Paragon Application Output Rates for Output Operations and Their Constituent Writes 34

Figure 2.10: Using fsync() to Avoid Paging on the I/O Nodes 35

Figure 2.11: Using fsync() in a Large Output Operation 36

Figure 2.12: Paragon Application Output Rates for Varying Numbers of Processors 37

Figure 2.13: Input and Application Output Performance of the Maryland CM-5. . . 38
Figure 2.14: Paragon And CM-5 Application Output Performance 39

3 Parallel Filestreams 42
Figure 3.1: Parallel Filestream Architecture 44
Figure 3.2: Comparing Ordinary and Parallel Filestreams 45
Figure 3.3: Parallel Filestream Methods 46
Figure 3.4: Parallel Filestream State Diagram for Output. 47
Figure 3.5: Parallel Filestream State Diagram for Input. 48
Figure 3.6: Recursive Insertion and Extraction Operators 51
Figure 3.7: Class Inheritance Structure in a C++
Parallel Filestream Implementation 52

4 Object-Parallel Filestreams 56
Figure 4.1: Object-Parallel Filestream Architecture 58
Figure 4.2: Comparing parallel and object-parallel filestreams 59
Figure 4.3: Object-Parallel Filestream Methods 60
Figure 4.4: Object-Parallel Filestream State Diagram for Output. 62
Figure 4.5: Object-Parallel Filestream State Diagram for Input. 63
Figure 4.6: Class Inheritance Structure in a pC++ Implementation of
Object-Parallel Filestreams. 68

5 Distributed Array Streams. 70
Figure 5.1: Architecture of the D/stream Abstraction. 73
Figure 5.2: Comparing Parallel and Object-Parallel Filestreams 74
Figure 5.3: D/stream Methods. 75
Figure 5.4: D/stream State Diagrams 77
Figure 5.5: Declaring a Distributed Array of Lists of Particles in pC++ 79
Figure 5.6: Summary of Checkpointing Using D/streams 82

Figure 5.7: Class Inheritance Structure in a pC++ D/streams Implementation. . . . 83
Figure 5.8: A Possible File Format for D/streams Implementations. 84

6 Persistent Collections 90

Figure 6.1: Persistent Collection Architecture 90
Figure 6.2: Sketch of the Persistent Collection Interface. 91
Figure 6.3: Persistent Collection State Diagram 92
Figure 6.4: Using Persistent Collections for I/O in an Astrophysics Simulation . . 97
Figure 6.5: Visualization of the Current State of an SCF Computation 98
Figure 6.6: Using Persistent Collections for I/O in SCFBOB. 99
Figure 6.7: Class Structure in a Persistent Collection Implementation. 100

7 Analysis and Comparison 105

Figure 7.1: Performance of a D/streams Implementation 107
Figure 7.2: Raw Performance Data 109
Figure 7.3: Checkpoint Time per MByte 110
Figure 7.4: Relative Checkpoint Time per MByte 111
Figure 7.5: Shore Architecture 113
Figure 7.6: Summary of Analysis 120

8 Conclusion 122

A Introduction to Parallelism 128

Figure A.1: Parallelism. 128
Figure A.2: Pipelined SIMD Architecture 129
Figure A.3: Parallel SIMD Architecture 129
Figure A.4: Distributed Memory MIMD Architecture 130
Figure A.5: Shared Memory MIMD Architecture 130
Figure A.6: Parallelism Extracted 131

B Introduction to Database Management Systems 134

Figure B.1: Single-Representation DBMS Architecture 142

Figure B.2: Two-representation DBMS Architecture 143

Chapter 1: Introduction

1.1 Overview

1.1.1 Scientific and Engineering Simulations

For the purposes of this dissertation, let a *scientific / engineering simulation* be defined as a program that simulates the behavior of a physical system by iteratively solving a discretized system of equations that models the physical system by describing the ways in which relevant physical variables affect each other over time and through space[1]. Let the equations in such a system be called *model equations*. Analytic solution of model equations may be difficult, time consuming, or impossible, especially when complex (i.e. realistic) boundary and/or initial conditions are involved. For this reason, since the advent of computers, numerical solution of equations modeling physical systems (i.e. simulation) has played an increasingly important role in science, allowing scientists to develop and test complex models of natural processes.

Once a given physical system is well understood, simulation becomes a useful tool for engineers. An engineer can combine a mathematical model of a structure being designed with a mathematical model of the physical environment in which that structure is intended to operate. A simulation based on the resulting (combined) model allows detailed measurements of the structure's predicted performance to be obtained, often at a

1. Throughout this dissertation, *italics* are used to denote definitions; underlining is used to denote emphasis.

much lower cost than the alternative (building an actual small-scale or full-scale version of the structure, along with a system to measure its performance).

Parallelism is an important consideration in many scientific and engineering simulations (see Appendix A for an introduction to parallelism), for two reasons:

1) There tends to be a large potential for the exploitation of parallelism in these programs, since they generally simulate physical processes that act in parallel at many or all points in space simultaneously.

2) Scientists and engineers benefit from the exploitation of parallelism in simulations, since applying more computational power can allow:

   - a given simulation to run in a shorter amount of time, or
   - a given set of model equations to be discretized more finely and solved in the same amount of time, resulting in a more accurate solution, or
   - a more complex model to be simulated in a given amount of time.

1.1.2 Data Management in Simulations

The terms *input/output* (*I/O*) and *data management* are used interchangeably in this dissertation to refer to the controlled movement of data between transient storage (i.e. physical memory) and stable storage (i.e. disk). Many parallel scientific and engineering simulations have complex I/O requirements; they require the coordinated movement of large sets of data between multiple storage devices and multiple compute nodes on distributed memory parallel machines. These requirements are not met by conventional file systems, so parallel file systems have been developed to support I/O in these programs. Current parallel file systems support I/O on simple data structures such as distributed rectangular arrays of primitive data types; however they do not provide adequate support for I/O on the complex distributed data structures often found in object-oriented parallel simulations. But object-orientation is becoming an increasingly

important issue in the development of simulations, as scientists and engineers seek to obtain the benefits of object-orientation [10] in the development process. This dissertation describes how class libraries can be used to extend the functionality of both parallel file systems and object-oriented database management systems to support the complex I/O requirements of object-oriented parallel simulations.

The remainder of this chapter describes data management tasks common in parallel simulations, identifies two models for data management systems supporting those tasks, summarizes the contributions of this dissertation, and reviews related research.

1.2 Data Management Tasks in Parallel Scientific and Engineering Simulations

This section identifies several types of I/O tasks commonly required in scientific and engineering simulations.

1.2.1 Entire Database Transfer Tasks

In most simulations that predict the dynamic behavior of physical systems (i.e. the manner in which physical systems change over time given a set of initial conditions), simulation proceeds iteratively in discrete time steps; during the computation for each time step, a set of data that defines the predicted state of the simulated physical system is updated to reflect changes in the physical system that are predicted to occur in a certain amount of actual time (the time step size). Let a simulation's *data set* or *database* be defined as the set of data representing the current state of a simulated physical system. Let an *entire database transfer task* be defined as a task that requires the transfer of an entire data set between memory and disk.

Several types of entire database transfer tasks commonly required in simulations are described below:

- **Initial and Final I/O**: *Initial input* is the transfer into memory of a data set representing initial conditions at the start of a simulation. *Final output* is the transfer of results from memory to permanent storage as a simulation progresses. Final output is required to obtain a permanent copy of a simulation's predictions. In addition, final output is often used during debugging as follows. Many parallel simulations are manually parallelized versions of sequential programs. During the parallelization process developers often need to compare the predictions of the parallel and sequential versions (the predictions should be the same), to confirm that parallelization has not introduced bugs. The comparison of these results is often performed by outputting corresponding data sets from each program to permanent storage, and using a program such as UNIX's `diff` to compare them.

- **Checkpointing and Restart**: Many simulations are exceptionally long-running. In order to avoid losing hours or days of of results due to program and system crashes, these programs may periodically *checkpoint*, i.e. save their state, so that the state can be reloaded and computation *restarted* if a program or system crash occurs.

1.2.2 Concurrent Data Access Tasks

Let a *concurrent data access task* be defined as an I/O task in which a data set is accessed by both a simulation and a second program concurrently. Several types of concurrent data access tasks that are commonly performed in simulations are described below:

- **Concurrent Visualization and Computation**: Since parallel simulation applications tend to be long-running, users frequently wish to visualize partial results as a simulation progresses. Visualization facilities may be provided by a

separate program, which then requires a mechanism for accessing the data set concurrently with the simulation.

- **Computational Steering**: Users sometimes wish to change a simulation's parameters as it executes. For example if a simulation is executing too slowly, a scientist may wish to reduce the precision of an approximation. Changing this parameter without stopping and restarting the simulation can be accomplished if a second *steering program* can concurrently access (and modify) the simulation's parameters. This is one form of *computational steering*.

1.2.3 Tasks Requiring Access to Large Databases

Some simulations operate on data sets that are too large to fit *in core,* i.e. in physical memory. Some data sets are too large to fit even within address spaces supported by virtual memory systems. Support for access to large databases is an important issue currently being addressed by researchers in both the computational science and database communities. This dissertation focuses on support for entire database transfer tasks and concurrent data access tasks; however, the parallel stream abstractions introduced in Chapters 3 through 5 could be applied to large database access tasks in which a simple sequential data access pattern is required.

1.3 Two Models for Systems Supporting Data Management Tasks

Several systems and abstractions for data management are developed in subsequent chapters; in this section, terminology is defined that will be used to describe these systems. Data management systems can be classified into several broad groups by data management model. A *data management model* specifies two things: the relationships

between the abstractions involved in data management (i.e. the relationships between programs, the data management system, program variables, transient storage, and stable storage), and the programmer interface to the data management system (i.e. the ways in which the programmer interacts with the data management system in order to accomplish data management tasks such as those identified in the previous section: entire database transfers, concurrent data access, and access to large data sets).

Two common data management models, the file I/O model and the persistent data model, are described in the following subsections.

1.3.1 File I/O Model

The data management systems discussed in chapters 3 through 5 operate within a *file I/O model* of data management; let such systems be called *file I/O systems*. In the file I/O model, program variables are transient, i.e. their values are always destroyed when the program terminates. *Files* (strings of bytes) are the units of stable storage and communication between programs. The file I/O system performs the work of moving data between variables and files, but the programmer must manage the variable-to-file mapping. The file I/O system gives the programmer primitives for initiating movement of data between program variables and files, and for *flattening* and *unflattening* variables (i.e. converting them to and from strings of bytes).

Data management abstractions in the file I/O model tend to be passive, performing no action until the programmer explicitly identifies an action to be performed. In the file I/O model, I/O tasks are generally coded as follows:

- **Entire database transfers** are coded with explicit commands that move data between files and memory.

- **Concurrent visualization and computation** can be performed by having the simulation program periodically write its data set to a file, and having the visualization program read this file when visualization is to be performed.

- **Computational steering** of a simulation can be coded similarly, by instructing the program to periodically read a file containing any parameters that are to be dynamically modifiable. If a steering program changes a parameter in this file as the simulation runs, the copy of the parameter in the simulation's transient memory is eventually changed when the file is read.

- **Access to large databases** can be coded by simply storing the entire database in a single file. File I/O systems generally provide primitives that allow programs to sequentially or randomly read and write to files. In the file I/O model it is the programmer's responsibility to manage the mapping between regions of the file and regions of a data set; in some cases this is a complex task, especially in simulations where the size or structure of the data set changes dynamically at run time.

1.3.2 Persistent Data Model

Chapter 6 introduces a data management system that operates within an alternate data management model: the persistent data model. In this alternate model, a program variable may be designated as *persistent*, meaning that it is to be stored in memory like an ordinary variable, but in addition it is to be associated with a second copy that is kept on stable storage. The data management system is responsible for the movement of data between memory and stable storage, and manages the mapping between the two. Persistent data on stable storage may be accessed by more than one program

concurrently. So units of typed persistent data are the units of both stable storage and communication between programs in this model.

In the persistent data model, the data management system gives the programmer a means of identifying which program variables are to be persistent, a way of associating those variables with persistent data on stable storage, and a way of initiating, committing, and aborting *transactions* on those variables. Persistent variables can only be modified within transactions. When the programmer *commits* a transaction, this tells the data management system that the program's persistent variables are in a consistent state, so at that point their values may be shared with other programs that may have requested access to those variables. When a transaction is *aborted*, the data management system must ensure that persistent variables are returned to the values they had when the transaction was initiated. Transactions may be aborted explicitly by the programmer or implicitly when a program or system failure occurs. So the data management system is responsible for returning all persistent data to a consistent state when a program is restarted after a program crash or a system failure.

Persistent data systems tend to be more active than file I/O systems in that they may spontaneously (without programmer intervention) initiate the transfer of data between stable storage and memory when this is needed; this is known as *transparent I/O*.

In the persistent data model, the I/O tasks identified in Section 1.2 can be coded as follows:

- **Initial and final I/O** are automatically performed by the persistent data system when persistent objects are declared and when the program terminates, respectively.

- **Checkpointing**[1] (i.e. saving a simulation's state) is performed transparently when the programmer commits a transaction; in the persistent data model, committing a

1. Do not confuse this computational science term with its homonym, the database systems term "checkpointing" meaning writing database buffers to disk.

transaction within which a data set has been modified causes the data management system to ensure that the modified data is written to stable storage.

- **Concurrent visualization and computation** can be performed by using the persistent data system from within the visualization program to access the persistent data within the simulation. The persistent data system's transaction mechanism manages the concurrent data access using a locking mechanism such as the one described in Figure 6.3.

- **Computational steering** of a simulation can be coded similarly; the persistent data system can be used within the steering program to concurrently access and modify the simulation's persistent data as the simulation runs.

- **Access to large databases**: Many persistent data systems are built on object-oriented database management systems (*OODBMS*), some of which have special *query languages* that support optimized access to subsets of large databases. A detailed discussion of OODBMS query languages is beyond the scope of this dissertation; see [39] for more information.

1.4 Contributions

This dissertation makes several contributions to the understanding of how data management can be supported in object-oriented data-parallel simulations; these contributions are summarized in this section. An expanded description of these contributions is presented in Section 8.1 of the Conclusion chapter.

Several abstract models of parallel file systems are developed in Chapter 2, and the experimentally measured performance of two commercial parallel file systems is explained in the context of these models. This analysis is intended to help application and library developers obtain high I/O performance from parallel file systems.

Three new abstractions for data management within the file I/O model are proposed in Chapters 3 through 5. These abstractions can be implemented on parallel file systems using language-specific class libraries; implementations of each abstraction are described. These abstractions provide functionality lacking in current parallel file systems, functionality that is required if the special I/O requirements of object-oriented parallel simulations are to be met. Chapter 3 introduces *parallel filestreams*, an extension of ordinary C++ filestreams that supports node-buffered parallel I/O with flattening and unflattening in SPMD parallel C++ programs. *Object-parallel filestreams*, introduced in Chapter 4, extend parallel filestreams to support the element buffering required for file-format-independent I/O on distributed arrays of objects in languages such as pC++. *Distributed array streams*, described in Chapter 5, extend object-parallel filestreams to support a simplified programmer interface as well as automatic bookkeeping of array distribution meta-data in special-format files.

A new abstraction for data management within the persistent data model is proposed in Chapter 6. This abstraction can be implemented on object oriented database systems (*OODBMS*) using language-specific class libraries; an implementation of this abstraction is described. This work shows how persistence can be used to support several common data management tasks in object-oriented parallel simulations.

Chapter 7 carries out a comparison of how well file I/O systems implemented on parallel file systems and persistent data systems implemented on OODBMS support I/O in object-oriented parallel simulations. The comparison is based on experience gained in the implementation of the abstractions described in Chapters 3 through 6, and involves a number of metrics: performance, space requirements, productivity effects, ease of implementation, maintainability, and portability.

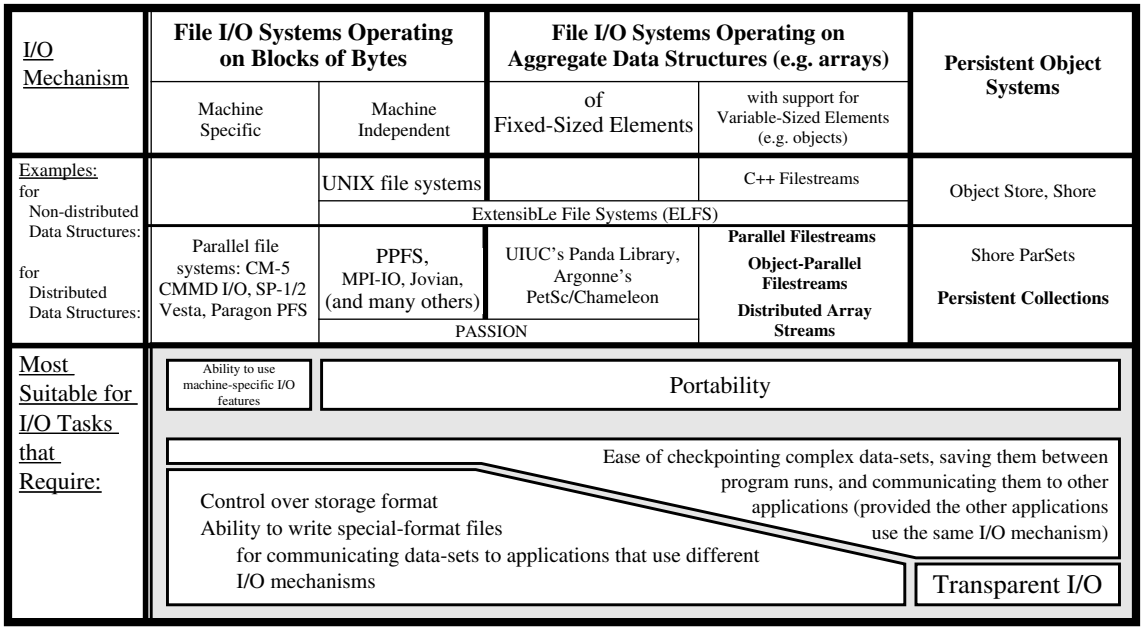


Figure 1.1: A Range Of I/O Mechanisms

Those mechanisms offering a higher degree of control over low-level I/O details are grouped toward the left; those offering greater ease of use and supporting I/O primitives operating on more complex types are grouped toward the right. (The thickness of the shapes under a given I/O mechanism in this figure is intended to represent the suitability of that mechanism for the indicated I/O task.)

1.5 Related Work

Figure 1.1 presents an overview of several research efforts involving data management in parallel programs, and shows where the systems described in this dissertation fit in. Parallel filestreams (Chapter 3), object-parallel filestreams (Chapter 4), and distributed array streams (Chapter 5) differ from ordinary C++ filestreams in that they support I/O on distributed data structures on distributed memory parallel machines; as discussed in Section 3.1, the single-buffered design of ordinary C++ filestreams make them incompatible with existing parallel file systems. These three new parallel stream abstractions also differ from the many research efforts that have been directed towards support for parallel I/O on distributed data structures, in that (like C++ filestreams) the

parallel stream abstractions introduced in this dissertation support I/O on arrays having arbitrarily complex, variable sized elements. The persistent collection abstraction introduced in Chapter 6 differs from all of the abstractions in the left two thirds of Figure 1.1 in that it operates within a persistent data model of I/O, rather than a file I/O model.

The following three subsections describe in more detail several of the research efforts related to the work described in this dissertation.

1.5.1 Parallel File System Performance

A number of studies of parallel file system performance have been undertaken. Kwan and Reed [40] describe the performance of the Scalable File System of the CM-5 at NCSA. Their motivation was to compare the performance of the CM-5 parallel I/O interfaces of CM-Fortran and C/CMMD. Bordawekar, Rosario and Choudhary present an experimental performance evaluation of the Intel Touchstone Delta Concurrent File System in [5]. Chapter 2 differs from this work in that in addition to describing and analyzing the performance of commercial systems, a set of abstract parallel file system models are developed and the performance of two commercial systems is explained in relation to those models. Unlike most of the related work, the emphasis in Chapter 2 is not on measuring the performance of specific I/O systems, but in identifying more general relationships between parallel file system architecture, application I/O behavior, and I/O performance.

1.5.2 Support for File I/O in Parallel Programs

A large number of experimental systems have been developed to support file I/O in parallel programs. The libraries PetSc/Chameleon [29] from Argonne and Panda [46] [45] from UIUC, and the software system PASSION [15] from Syracuse, support I/O on distributed arrays of fixed-sized elements in the context of the distributed-memory

data-parallel programming model. PetSc/Chameleon supports I/O on block-distributed arrays. Panda supports more general HPF-style array distributions and interleaving, as does d/streams. PASSION is a large effort at Syracuse that provides support at the language, compiler, runtime, and file system level for I/O on distributed arrays as well as for computation over out-of-core distributed arrays. (It should be noted that distributed array streams can only be used for out-of-core computation in the simple case where the simulation's data access pattern is sequential.) The two-phase access strategy described in PASSION for parallel access to files, where data is first read in a manner conforming to the distribution on disk and then redistributed among the processors, is similar to the implementation of the distributed array streams **load** method. Language extensions for parallel I/O on distributed arrays have been discussed in the HPF Forum [26].

The distributed array streams implementation described in Chapter 5 differs from these systems in that it supports buffered I/O on distributed arrays of objects whose size may vary over the array itself, for example distributed arrays of variable-density grids or distributed arrays of lists of particles, in the context of a portable object-parallel language.

1.5.3 Support for Persistent Data in Parallel Programs

Several systems other than persistent collections have been developed to support persistence in parallel programs, but not nearly as much work has been done in this area as in the area of support for file I/O. ParSets [21], a component of the Shore [7] OODBMS developed at the University of Wisconsin at Madison, is similar to the pC++ implementation of persistent collections described in Chapter 6, in that it provides support for data-parallelism in the form of parallel application of methods of persistent objects. The pC++ programming system, when combined with persistent collections, differs from Shore ParSets in that it supports programmer-controlled alignment and distribution of the elements of aggregate data-structures, and the parallel application of

operators acting simultaneously on multiple aggregate persistent data-structures. It may be possible to provide these facilities in a ParSet abstraction; however, the initial ParSets paper does not describe how such support could be provided.

The Raven [1] operating system and programming language supports task-parallelism and persistent objects, but unlike pC++ persistent collections, no facilities are provided for data-parallelism. Shared Data Abstractions [12], developed at the University of Vienna and ICASE at NASA Langley, extend the support for data-parallelism in FORTRAN 90 with support for task-parallelism and for simplified I/O on data modules, but no support is provided for concurrent multi-program access to data (unlike the persistent collection implementation described in Chapter 6, which does support concurrent access).

A large amount of work has been done on object-oriented database programming systems providing concurrent distributed access to persistent data; however, the emphasis has been on distribution supporting inherently distributed data or inherently distributed users, rather than distribution supporting parallelism for speedup (as is supported in pC++ persistent collections). In the domain of relational DBMS, a large body of work exists on parallel implementation of internal DBMS operations such as the processing of joins. However, this work does not address explicit parallelism in application code, which is useful for exploiting parallelism available in the problem domain (explicit parallelism is supported in pC++ persistent collections).

1.6 Summary and Dissertation Outline

This introductory chapter has identified the data management tasks common to parallel scientific and engineering simulations, described two possible models for systems supporting the data management requirements of those tasks, described the contributions

of this dissertation in terms of those models, and referenced other research work having similar goals.

The remainder of this dissertation is organized as follows. Chapter 2 introduces parallel file systems and relates their performance to their internal architecture and to application I/O patterns. Chapters 3 through 6 describe designs and implementations of four new abstractions for data management within the file I/O and persistent data models in SPMD C++ and object-parallel programs. Chapter 7 evaluates how well file I/O systems implemented on parallel file systems and persistent data systems implemented on OODBMS support I/O in object-oriented parallel simulations. Chapter 8 summarizes the results of this dissertation and describes directions for future research, and Appendices A and B present brief introductions to parallelism and database management respectively.

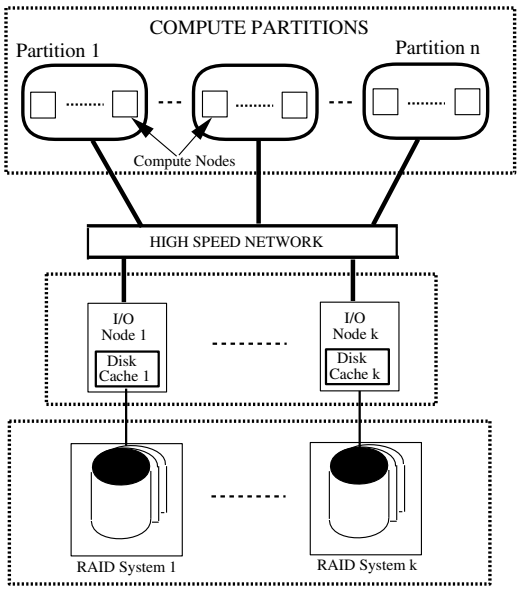
Chapter 2: I/O Performance in Parallel File Systems

Restricting a parallel machine to the use of a single I/O device would cause I/O to become a bottleneck, since I/O power would be unable to scale with computational power. Therefore, many current distributed memory parallel machines provide a scalable *parallel file system* (*PFS*) consisting of several I/O processors each connected to a high speed I/O device. This enables high bandwidth cooperative I/O i.e. *parallel I/O* to be performed from the compute nodes [47] [36] [17]. Many large-scale parallel applications require the use of parallel I/O to achieve high performance [44].

This chapter introduces parallel file systems, defines several measures of PFS performance, and relates PFS performance to application I/O patterns and to various elements of PFS internal architecture, such as the use of buffering and virtual memory on the I/O nodes. In addition, experimental results are presented that relate the performance of PFS on the Intel Paragon and the Thinking Machines CM-5 to the PFS models. The models and results presented in this chapter should be useful for optimizing I/O performance in applications and I/O libraries, configuring existing PFS and designing new ones. This work was first described in [32].

Figure 2.1: Architectural Elements of Distributed Memory I/O Systems

A variable number of I/O nodes share a scalable network with the compute nodes, allowing I/O power to be scaled with compute power. Each I/O node has a disk buffer and is connected to a disk device, which could be a RAID (Redundant Array of Inexpensive Disks). The compute nodes in the parallel machine are grouped into compute partitions. The I/O nodes are shared by all the compute partitions.



2.1 Introduction to Parallel File Systems

Figure 2.1 shows the architectural elements of the PFS of several current distributed memory parallel machines [43] such as the Intel Paragon and Thinking Machines CM-5. These PFS support an interface similar to that of a normal UNIX file system. In addition, they are accessible through special system calls provided for parallel I/O. The interface for parallel I/O is similar to the interface for normal UNIX I/O, but an *I/O mode* associated with each file descriptor coordinates the way in which the compute nodes access and modify files.

2.1.1 An Example Program

This section describes an example program that shows how parallel file systems are used to support the I/O requirements of "polygon overlay", a pC++ [30] program that reads in two lists (or "vectors") of polygons (the "left" and "right" vectors), computes a new polygon vector by overlaying the left and right vectors, then outputs the overlay vector. This program is discussed in detail in [49]. (For a brief description of pC++ see "The

```
void Polygon::readPolygon() {
  {
    // Read the local part of a DISTRIBUTED vector of polygons
    // into leftVec on each node
    leftVec = (polyVec_t *)malloc(sizeof(polyVec_t));

    // Open a read-only input pfstream (s) on each node
    ipfstream s(leftFilename, pfilebuf::rdonly);

    // Read the total size of the polygon vector
    s.setparmode(pfilebuf::unixpar);
    s.unbufread((char *)&totalLeftLength, sizeof(int));

    // Calculate the local size of of the vector, store in leftVec->len
    ...

    // Read the local part of the vector into leftVec->vec
    leftVec->vec = (poly_t *) malloc(sizeof(poly_t)*leftVec->len);
    s.setparmode(pfilebuf::syncpar);
    s.unbufread((char *)leftVec->vec, leftVec->len * sizeof(poly_t));
  }

  {
    // Read a COMPLETE COPY of a second vector of polygons
    // into rightVec on EACH NODE
    rightVec = (polyVec_t *)malloc(sizeof(polyVec_t));

    ipfstream s(rightFilename, pfilebuf::rdonly);

    s.setparmode(pfilebuf::unixpar);
    s.unbufread((char *)&(rightVec->len), sizeof(int));

    rightVec->vec = (poly_t *) malloc(sizeof(poly_t)*rightVec->len);
    s.unbufread((char *)rightVec->vec, rightVec->len * sizeof(poly_t));
  }
}
```

Figure 2.2: Using a Parallel File System for Input

pC++ Language" on Page 133.) The parallel file system interface described here is supported by the *parallel filestream* abstraction, which is described in more detail in Chapter 3. Parallel filestreams support additional functionality not found in current parallel file systems; however, the polygon overlay program does not require the use of this additional functionality, so in this case parallel filestreams function purely as *I/O middleware*, supporting application portability by providing a single I/O interface over multiple parallel file systems. The more advanced features of parallel filestreams are described in Chapter 3.

An early version of this program that used ordinary UNIX I/O system calls and no parallel I/O had 200 lines of code devoted to I/O, approximately 10% of the total code in the program. Rewriting the original UNIX I/O using a parallel file system reduced the I/O code from 200 to 70 lines, a 65% reduction. Most of this reduction was due to the fact that in the original program, a significant amount of code was required to copy pieces of the (distributed) output polygon vector from their home nodes to node zero, which performed all of the (sequential) file output. When using a parallel file system, all of the nodes of a parallel machine can participate in I/O, so this communication is unnecessary.

Input

Figure 2.2 shows how parallel filestreams (*pfstreams*) can be used to input the left and right polygon vectors in the function `readPolygon()` of polygon overlay. After allocating memory for the left vector, before reading it in from the file, an input pfstream `s` is declared as follows:

```
ipfstream s(leftFilename, pfilebuf::rdonly);
```

This opens the file `leftFilename` and attaches the stream `s` to it.[1]

The left vector is a distributed data structure, i.e. it doesn't appear in its entirety on any one node, but is distributed across the nodes of the parallel machine. On the other hand the right vector is copied in its entirety into each node. The parallel file systems of the Paragon and CM-5 support I/O modes that allow the programmer to use parallel I/O for each of these types of input. (The names used for these modes in the parallel filestreams abstraction correspond to the I/O mode names used in the Paragon's parallel file system.) In *`unixpar`* mode, each processor has its own file pointer and can

1. The programmer interface to parallel filestreams is similar to that of ordinary C++ filestreams. A complete description of the parallel filestream interface is presented in Chapter 3. For the purposes of this discussion, the important issue is not parallel filestream syntax, but the way in which the underlying parallel file system's I/O modes can be used to coordinate parallel I/O from within a parallel application.

perform I/O independently of the other processors. This mode is useful for reading a copy of a single block of data in the file into each processor in parallel, and for reading pC++ global variables. The total number of polygons in the left polygon vector is to be read into the pC++ global variable `totalLeftLength`, so at this point the parallel I/O mode of the stream `s` is set to `unixpar` mode:

```
s.setparmode(pfilebuf::unixpar);
```

Next an unbuffered read of `sizeof(int)` bytes is performed from the stream `s` directly into the global variable `totalLeftLength`:

```
s.unbufread((char *)&totalLeftLength, sizeof(int));
```

The parallel filestream abstraction also supports *buffered* I/O, which can yield higher I/O performance when I/O is to be performed on large data sets stored non-contiguously in memory. However, when data are mostly contiguous, as in this program, unbuffered I/O is more efficient.

The left polygon vector is to be distributed across the elements of the `Polygon` collection using a block mapping from the file to the collection elements. Now that the total size of this vector is known, the number of polygons to be read into each element is computed. Then after allocating memory to store the local segment of the vector, the parallel I/O mode of the stream is set to *`syncpar`* mode, in which all the processors use a single file pointer, reads and writes are synchronizing operations, and data is distributed from the file to the nodes using a block distribution strategy.

```
s.setparmode(pfilebuf::syncpar);
```

In this I/O mode, each node must specify the amount of data it requires from the file (different nodes may request different amounts). The data blocks for the nodes are read from the file and are sent to each processor in parallel. Next the local block of data of size

```
leftVec->len * sizeof(poly_t)
```

```
void Polygon::writePolygon()
{
  int totalOutVecLength;

  // Compute total size of the distributed vector outVec
  totalOutVecLength = outVec->len;
  pcxx_ReduceIntAdd(&totalOutVecLength, 1);

  // Open a write-only output pfstream
  opfstream s(outFn, pfilebuf::wronly | pfilebuf::creat | pfilebuf::trunc, 0644);

  // Use pfilebuf::unixpar mode since we're writing a global variable.
  s.setparmode(pfilebuf::unixpar);
  s.unbufwrite((char *)&totalOutVecLength, sizeof(int));

  // (See the description of pfilebuf::syncpar mode in readPolygon())
  s.setparmode(pfilebuf::syncpar);
  s.unbufwrite((char *)outVec->vec, outVec->len * sizeof(poly_t));
}
```

Figure 2.3: Using a Parallel File System for Output

is read from the stream `s` directly into the memory pointed to by `leftVec->vec`:

```
s.unbufread((char *)leftVec->vec,
            leftVec->len * sizeof(poly_t));
```

At this point input is completed; the file is closed automatically when the program block containing the declaration of the stream `s` is exited.

Next the "right" vector is read from `rightFilename` in a similar manner, except that this vector is not a distributed data structure; a copy of the entire vector is to be stored on each processor. For this reason, `unixpar` mode is used to read this vector, rather than `syncpar` mode (which was used for `leftVec`).

Output

Later in the program, the overlay (output) vector `outVec` is computed and then output by the `writePolygon()` method (see Figure 2.3). The overlay vector is a distributed data structure having a block distribution. Before it is output, its total size is first computed using a reduction operation. Next an output parallel filestream is declared, and the total size is written to the file using `unixpar` mode (since only one copy of the

total size should appear in the file, and each node's file pointer should be advanced past the just-written size prior to the next write).

Next the local part of the output vector is written to the file using `syncpar` mode. In `syncpar` mode, writes are synchronizing operations, and the parallel file system assumes the data structures being written have a block distribution across the nodes; the blocks of data (one per node) are written to the file in order of node number. The file is closed automatically by the stream's destructor at the end of the block in which the stream was declared.

2.1.2 Performance Measures

This section defines several parallel file system performance measures. In later sections, these measures are used to describe the performance both of models of PFS and of commercial PFS.

Assume a parallel machine has p compute nodes and one I/O node. Also assume that:

- An application, when run on the compute nodes, issues a parallel input or a parallel output request to the I/O system.
- The parallel I/O request consists of a system call executed by each compute node, which causes a total of n bytes to be concurrently read or written (n/p bytes from each compute node).
- The time for I/O is the same on each compute node.

For such a system let the following I/O performance measures be defined:

- The *application I/O time* denoted by $Time_{app}(n)$ is the time the system call takes to move data between the application's buffers and the I/O system.

- The *actual I/O time* denoted by $Time_{actual}(n)$ is the actual time for the I/O operation to complete, i.e. the time taken for the I/O system to move data all the way between the application and the disk.

These times are significant because while data is being transferred between the application's buffers and the operating system, the application must not disturb the application memory holding the data, and until the I/O operation completes, the data cannot be used by the application (in the case of input) or by other applications (in the case of output).

A rate can be associated with each of these times:

- The *application I/O rate* denoted by $Rate_{app}(n)$ is:

  $n \ / \ Time_{app}(n)$

- The *actual I/O rate* denoted by $Rate_{actual}(n)$ is:

  $n \ / \ Time_{actual}(n)$

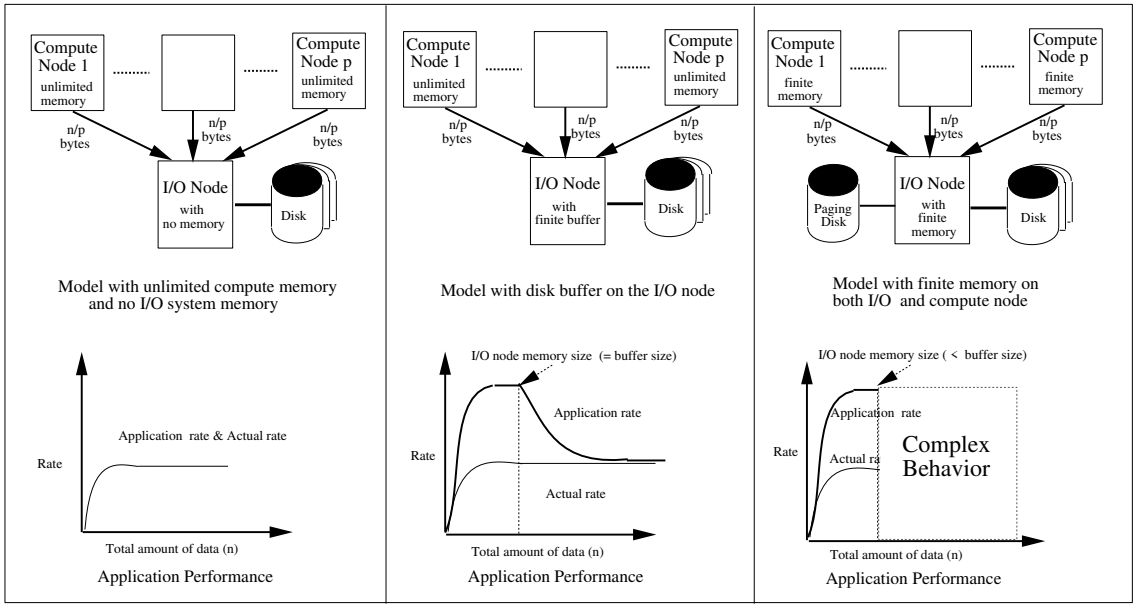


Figure 2.4: Three Parallel File System Models

2.2 Models Explaining the Impact of Buffering and Virtual Memory on PFS Performance

In this section a simple model of a PFS is described and successively refined in three steps. The initial model is a system with unlimited memory on the compute nodes and no I/O system memory. In the second model, an I/O buffer is added to the I/O node, and in the third model, virtual memory is added on both the compute and I/O nodes. Figure 2.4 will be referred to throughout the following three subsections; it depicts the internal architecture of these three models, along with the expected I/O performance at each stage. The models described in this section are meant to explain how the interaction of buffering, virtual memory, and application I/O patterns can affect I/O performance.

2.2.1 Unlimited Compute Node Memory and No I/O System Memory

The first parallel file system model that will be analyzed is a system with unlimited memory on the compute nodes and no memory on the (single) I/O node (see the left side of Figure 2.4). In this case, the I/O node moves data directly between the application's memory (on the compute nodes) and the disk. So by the definition of $Time_{app}(n)$ and $Time_{actual}(n)$,

$$Time_{app}(n) = Time_{actual}(n)$$

and this implies that

$$Rate_{app}(n) = Rate_{actual}(n).$$

Let this rate be called *Rate(n)*.

As n increases, one would expect to see Rate(n) start low at first due to latency, and then quickly increase to some limiting rate corresponding to the raw I/O rate of the physical disk device (see the first graph in Figure 2.4). After this point there should be no change in I/O performance when the the size of the data set is scaled upwards. So if the size of the program's data set is increased by some factor f, the time for I/O on that data set should increase by the same factor f.

2.2.2 Finite Physical Memory Buffer in the I/O System

Now assume the I/O node has an I/O buffer of finite size b bytes of physical (not virtual) memory (see the central part of Figure 2.4). In this model and the next the discussion is limited to output for clarity.

Parallel output in this model consists of two concurrent processes. Data is transferred 1) from the application through the operating system to the disk buffer, and 2) from the disk buffer to the disk. Assume the transfer of data from the application's

buffer to the disk buffer happens more quickly than the transfer from the disk buffer to the disk.

As the center graph in Figure 2.4 shows, when the total amount of output (n) that the application performs is less than the size of the disk buffer (b) then the application output rate will be much higher than the actual output rate, since the operating system can quickly copy the contents of the application's buffers into the I/O node's buffer. Therefore the system calls can complete before the data is actually written to the disk. When n is very small, latency will slow the application output rate, so a graph of the application output rate as a function of n will start low, rise quickly, then asymptotically approach a value corresponding to the peak rate at which the operating system can transfer data from the compute nodes into the I/O node's buffer.

However when the amount of output performed is greater than the I/O node’s buffer size, the system calls will start to take longer to return, for the following reason. After the disk buffer fills, the transfer rate from the compute nodes to the I/O buffer will be the same as the rate at which data is transferred from the I/O buffer to the disk. Therefore as the output size is increased past b bytes, the graph of the average application output rate will start dropping, asymptotically approaching the actual output rate.

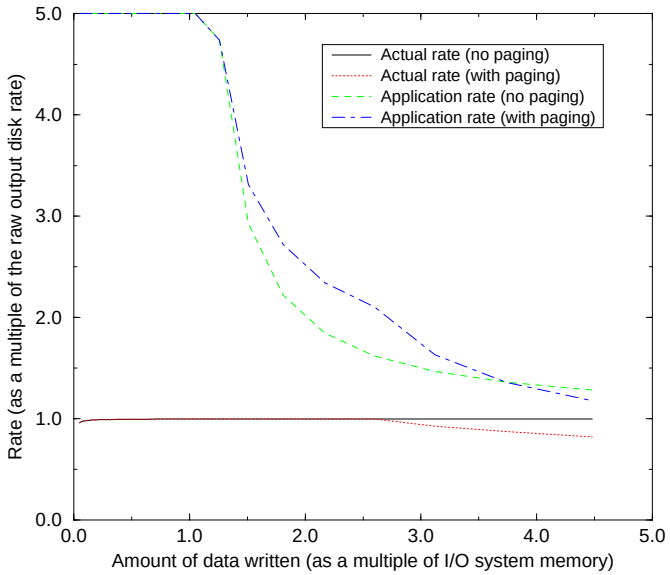


Figure 2.5: Comparing Output Performance With and Without Virtual Memory on the I/O System

(This figure compares the performance of the models in Sections 2.2.2 and 2.2.3). This data was obtained from a simple virtual memory simulator, in which it was assumed that the raw output disk bandwidth is one fifth the interconnection network bandwidth, and that the raw paging disk bandwidth is one half the raw output disk bandwidth. This graph suggests that the use of virtual memory for the I/O system can improve application output rates at the expense of actual output rates.

2.2.3 Virtual Memory on Compute and I/O Nodes

Actual parallel systems have finite physical memory on both the compute and I/O nodes but provide a larger virtual memory through paging mechanisms. In this section the model from the previous section is extended to include virtual memory on both the compute and I/O nodes (see the right portion of Figure 2.4). Assume the size of the I/O node's buffer is *b* bytes and the physical memory size on both the compute nodes and the I/O node is *m* bytes, where b > m (so the I/O node's buffer is too large to fit into the I/O node's physical memory).

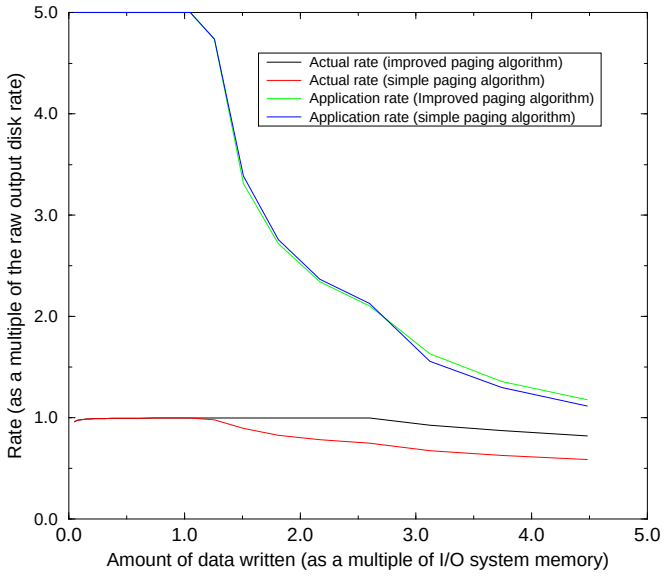


Figure 2.6: Comparing Paging Algorithms for the Model with Virtual Memory on the I/O Nodes

(Simulator data). Both actual and application performance start to drop (due to paging) when the total amount of data written equals the total physical I/O system memory. The severity of the drop depends on the paging algorithm used. With a simple demand paging algorithm, the actual rate drops to 60% of the raw disk rate when 5 times the I/O system memory size is written. With an improved paging algorithm that incorporates early page-in of paged blocks to unused I/O system memory, the actual rate drops to 85% of the raw disk rate, a smaller drop than for the simple algorithm.

For n < m the behavior of this model is the same as that of the model in the previous section. When n > m paging starts on the I/O node. At this point the system's behavior will be complex, and will be determined by the paging algorithm and the raw I/O rates of the paging disk and the actual disk. Also when the total output size is greater than the total size of the compute nodes' physical memory (n > p*m), paging will occur on both the I/O node and the compute nodes. Analytic modeling of this behavior is difficult, so a simple virtual memory system simulator was developed to examine this behavior.

Figures 2.5 and 2.6 depict the results of experiments run on this simulator. Figure 2.5 shows that the use of virtual memory for the I/O system can improve application output rates at the expense of actual output rates. Figure 2.6 shows that if I/O system virtual memory is used then specialized paging algorithms, such as early page-in to unused I/O system memory, can improve actual output performance compared to the performance that can achieved with simpler paging algorithms.

So far it has been assumed that the paging disk device and the destination disk device have the same raw I/O rate. Consider the case where the paging disk is slower than the actual disk. For a write of n bytes where n is large, by the time all the data has been transferred out of the compute nodes, a large amount of data will have been transferred onto the paging disk. After this, the amount of data in the I/O node's buffer will begin to drop, since that data will be transferred out of the buffer by the actual disk faster than it is transferred in to the buffer from the paging disk. At the point where the amount of data in the I/O node's buffer drops to zero, the rate at which data is written to the actual disk will drop to the paging disk's raw I/O rate. Therefore, when the paging disk is slower than the actual disk, for large n, it becomes important to avoid causing paging on the I/O node during output. Paging on the I/O node can be avoided by writing only enough data to fill the I/O node's memory, and then waiting for the memory to be written to the actual disk before writing more data.

2.3 Experimentally Measured Parallel I/O Performance

This section describes the performance of parallel file systems on two commercial parallel machines, and relates their measured performance to the performance patterns predicted by the PFS models described in the previous section.

Parallel file systems on some commercial parallel machines (such as the Intel Paragon) perform better when a large parallel write is broken into a series of smaller ones. Let a *parallel write operation* be defined as a single parallel write system call performed on each compute node of a parallel machine, and a *parallel output operation* as a series of one or more parallel write operations with no intervening computation. As mentioned in the previous section, there are two significant amounts of time associated with a given parallel output operation: the *application output time* and the *actual output time*. Also there are two data transfers rates, the *application output rate* and the *actual output rate*, associated with these times. Let *parallel read operation*, *parallel input operation*, *application input rate*, and *actual input rate* be defined similarly.

The following experimental results for PFS input performance are all in terms of the actual input rate, for the following reason: after initiating a read operation, there will be some period of time during which no data is being transferred into the user's program, while the data is being read from the disk and sent through the network. But neither the Paragon nor the CM-5 provide a way to measure that time. So there is no way to measure the remaining time when data is being transferred into the application's buffers. Therefore, even though it makes sense to talk about an application input rate for a parallel read operation, there is no way to measure that rate on the Paragon or the CM-5.

2.3.1 Intel Paragon

This section describes the results of PFS performance experiments that were run on the Intel Paragon at Indiana University in the Fall of 1994. This machine had 92 compute nodes and 2 I/O nodes. Each I/O node had 2 RAID-3 arrays; each array had 5 disks for a total of 10 disks. The programs described here were the only ones running on the machine during most of these experiments. The I/O nodes on this machine were in a service partition. This is not the most optimal configuration since the service partition

nodes also run all the UNIX commands issued by users; these commands use up potential I/O buffer memory.

The test program that was used performed repeated I/O operations, linearly increasing the total amount of I/O. The following optimizations, suggested by the Paragon User's Guide [36], were incorporated into the test program:

1) Read/write buffers were aligned to page boundaries. If properly aligned buffers are not used, the system must copy the data to new, aligned buffers, worsening performance.

2) The Paragon's record (`M_RECORD`) parallel I/O mode was used. This mode requires that the same amount of data be read or written by each processor in any given parallel I/O operation.

3) Large write operations were broken into multiple smaller operations, where the amount of data written by each processor is either equal to the stripe unit size, or is an integer multiple of the full stripe size. This makes optimum use of file striping, keeping all the I/O nodes busy at once.

4) Read and write requests were initiated with the file pointer located on file system block boundaries.

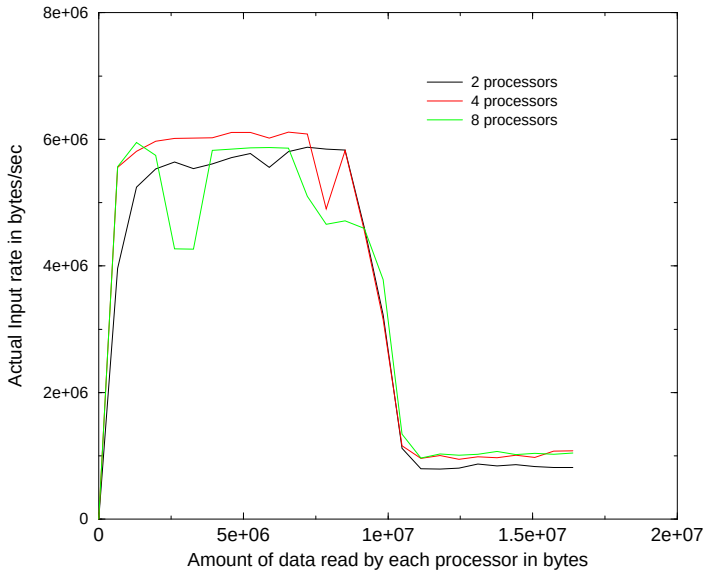


Figure 2.7: Paragon Input Performance (in terms of the actual input rate). Input performance drops when the input size exceeds the physical memory of the compute nodes, causing paging.

Parallel Input Performance

Figure 2.7 is a graph of the IU Paragon's actual input rate for varying sizes of parallel input operations, each comprised of a single parallel read operation, for 2, 4, and 8 processors. The input performance suddenly drops at approximately 8 MBytes of data input per node; at this point the input size exceeds the physical memory of the compute nodes, causing paging.

The Paragon User's Guide indicates that parallel write performance can be improved by breaking large parallel write operations into a series of smaller operations, where the amount of data written by each operation is equal to the stripe unit size. During the course of this experimentation it was determined that the corresponding optimization for input (breaking up large parallel read operations into smaller read operations) does not improve input performance on the Paragon.

Parallel Output Performance

Figures 2.8 through 2.12 describe the empirically measured parallel output performance of Indiana's Paragon. Parallel output operations in these experiments were comprised of

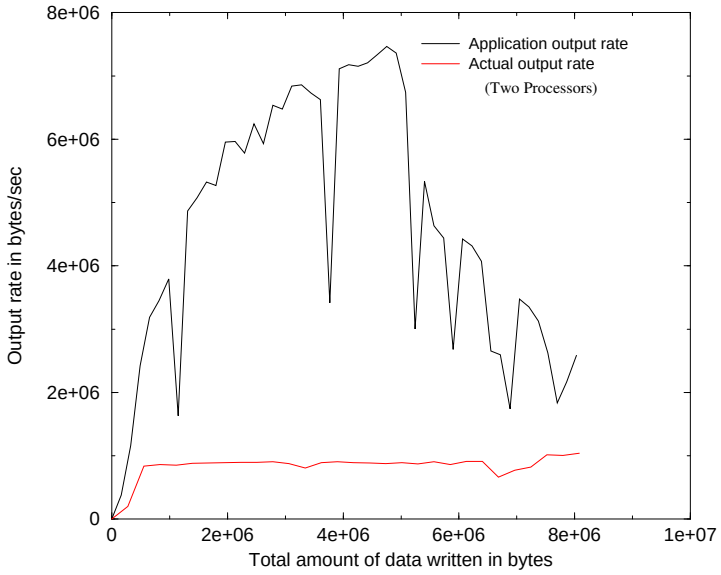


Figure 2.8: Comparing the Paragon's Application and Actual Output Performance measured using two compute nodes on IU's Paragon. The graph shows that the actual output rate remains nearly constant, no matter how much data is written. The graph of the application output rate, on the other hand, rises to a peak around 5MB then suddenly begins to drop. A possible cause of this behavior becomes apparent in Figure 2.9.

a series of smaller parallel write operations of size equal to the I/O system's stripe unit size.

Figure 2.8 compares the Paragon's application and actual output rates for output operations of varying size. Note the sudden drop in the application output rate for output operations larger than 5 MBytes. Figure 2.12 shows that this same peaked shape appears after about the same amount of data has been written, regardless of the number of compute nodes involved in the output operation.

The model described in the previous section predicts that if small parallel write operations are performed repeatedly, then the application output rates of those operations should be high until the memory of the I/O nodes is filled, at which point the application

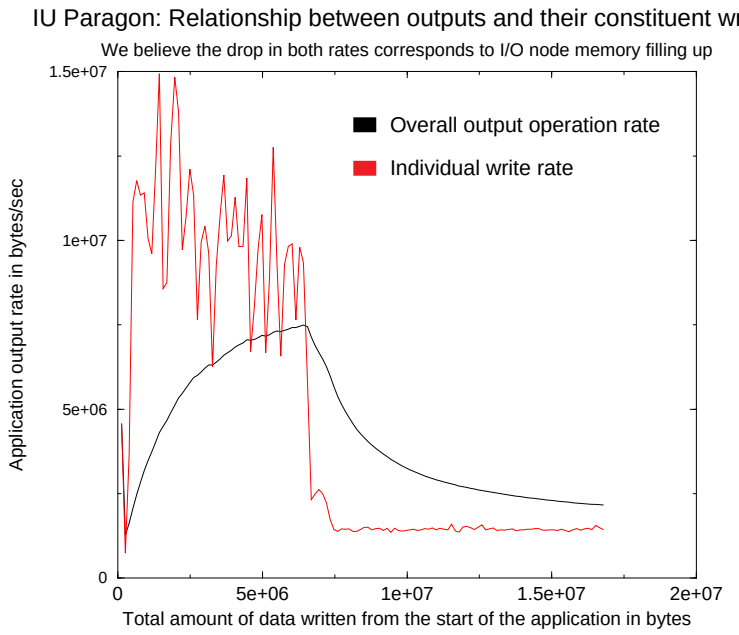


Figure 2.9: Paragon Application Output Rates for Output Operations and Their Constituent Writes

The application output rate for an output operation of a given size is shown in black directly above that size; the rates for the individual write system calls making up that output operation appear in red to the left of that size.

output rates should suddenly drop due to the extra time required for paging on the I/O nodes. This appears to be the cause of the drop in application output rate observed in Figure 2.8. See Figure 2.9, which relates the empirically measured application output rate of parallel output operations of various sizes to the application output rates of the individual parallel write operations that make up those parallel outputs. After 6 Mbytes have been written, the application output rate of the individual parallel write operations suddenly drops. This point roughly corresponds to the total size of the memory on the Paragon's I/O nodes, so it is likely that this drop is caused by the I/O system's memory becoming filled. The sudden drop in the rate for the individual writes is what causes the observed peak in the rate for the output operations that those writes comprise.

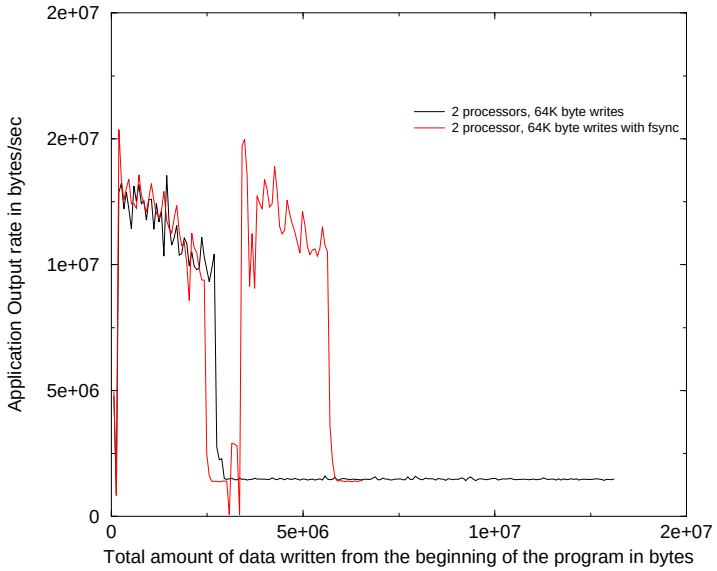


Figure 2.10: Using fsync() to Avoid Paging on the I/O Nodes

Results for two runs are shown, one in which an `fsync()` call was performed after a total of 3MBytes of data were output, and one without the `fsync()`. (The individual writes were all of equal size; the rate for the write that brings the total amount of data output in a given run to b bytes appears at b on the x-axis.) The rate starts high in both graphs, then suddenly drops when the I/O system's memory is filled, due to the onset of paging. But in the run with `fsync()`, the rate jumps up after the `fsync()`.

It should be possible to avoid a drop in the application output rate on the Paragon by making sure not to cause paging on the I/O nodes. This could be accomplished by repeatedly writing just enough data to fill the I/O nodes' memory, then pausing until that data is completely transferred to disk. To test this theory, the application output rates of each of a series of many small writes were measured in two runs (see Figure 2.10). In one run output was paused using the UNIX `fsync()` system call after roughly 3MBytes of data were written. (`fsync()` forces a pause until all data is completely written to disk.) In the second run, output was not paused. In both runs there is a drop in the application output rate after a total of about 2.5MBytes of data has been written. In

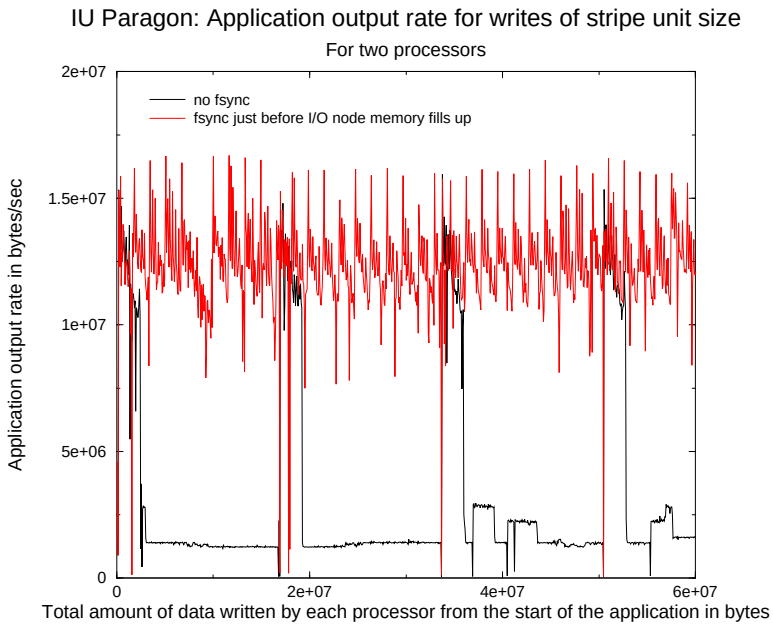


Figure 2.11: Using fsync() in a Large Output Operation

Results of an experiment similar to Figure 2.10, but for a greater total output size. `fsync()` was called periodically in one run after enough data had been written to fill the I/O nodes' buffers; `fsync()` was not called in the other run.

the run with the `fsync()`, the rate jumps up again at 3MBytes after writing is resumed, while the rate remains low in the run without `fsync()`. So it would appear that one can avoid a drop in the application output rate by using `fsync()` to wait for the I/O nodes' memory to clear after writing enough data to fill it.

Figure 2.11 shows the results of an experiment similar to the one described in Figure 2.10, but where three times as much data was written. In one of the runs in Figure 2.11, `fsync()` is periodically called before the I/O nodes' memory fills (instead of just being called once as in Figure 2.10). The model described earlier predicts that as data continues to be written to the I/O nodes, in the run without `fsync()`, the application output rate should remain low indefinitely after the point where the I/O nodes' memory fills up. The behavior of the non-`fsync()` run in Figure 2.11 is more complex than the model predicts; the rate jumps up periodically, as if the I/O nodes'

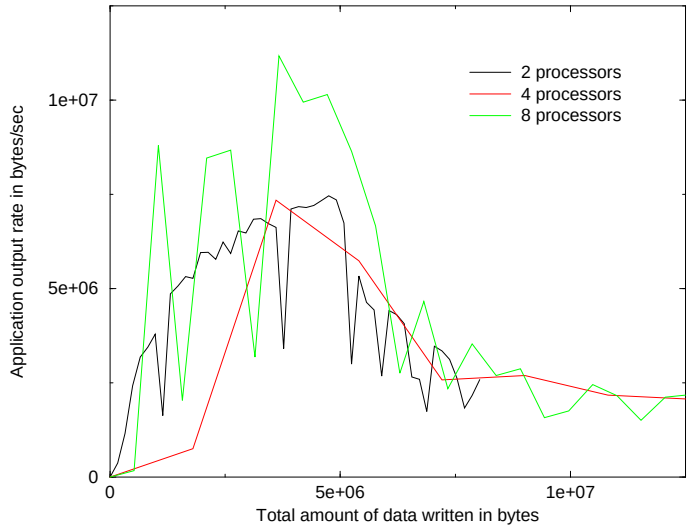


Figure 2.12: Paragon Application Output Rates for Varying Numbers of Processors Notice that the performance drop happens at around the same point (close to 5MB) regardless of the number of processors from which the output takes place. It appears that this drop occurs when the I/O system's memory becomes filled, causing the onset of paging on the I/O nodes.

buffers had spontaneously and periodically been cleared. Perhaps this behavior is due to perturbations caused by I/O from other applications or by I/O from the operating system itself.

2.3.2 Thinking Machines CM-5

This section describes results of experiments run on the CM-5 at UMIACS, Maryland in the Fall of 1994. This machine had 32 compute nodes and 2 I/O nodes. A wall clock timer was used to obtain these measurements, even though there were other users on the system, since the other timers on the CM-5 cannot be used for I/O measurements. Figure 2.13 shows the UMIACS CM-5's input and application output performance. The drop in performance for I/O operations greater than 6 MBytes per node is probably due to paging caused by virtual memory on the compute nodes. Figure 2.14 compares the

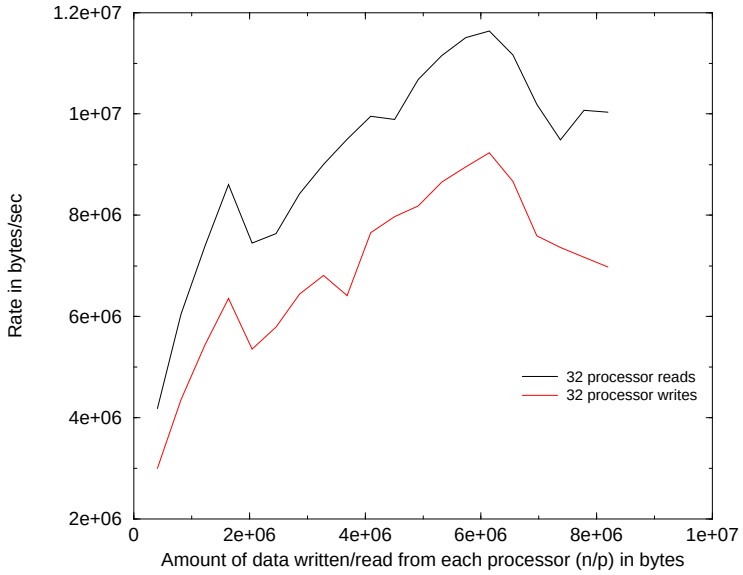


Figure 2.13: Input and Application Output Performance of the Maryland CM-5. Notice the performance drop for both input and output rates beyond 6MB.

application output performance of Maryland's CM-5 to that of Indiana's Paragon. Following the publication of [32] (in which this work was first described), Intel reconfigured the I/O system of the Paragon at Indiana, improving its I/O performance.

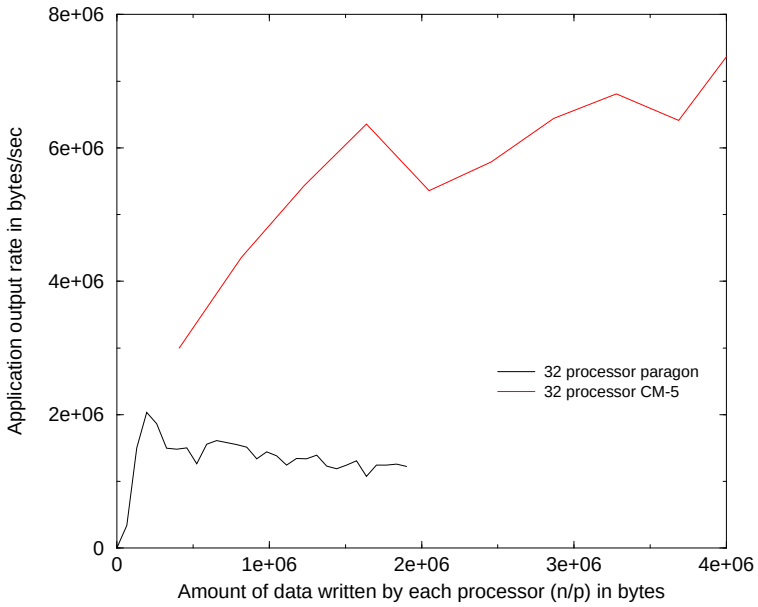


Figure 2.14: Paragon And CM-5 Application Output Performance

for IU's Paragon and Maryland's CM-5. Both measurements are for 32 processors. Each Paragon output operation was comprised of a series of parallel write operations, each of a size equal to the stripe unit size; each CM-5 output operation was comprised of a single parallel read/write operation.

2.4 I/O Optimizations

The I/O optimizations that were identified during the course of these experiments are summarized below. These optimizations should be applicable on the Paragon and the CM-5, as well as other machines having similar I/O architectures.

- Blocks of data to be output from the application should be properly aligned in memory so that the operating system doesn't have to make an aligned copy of the data during output.

- Striping mechanisms provided in parallel I/O systems should be used to balance the I/O load among the components of the parallel I/O system. This may require

carefully setting the size and number of the individual I/O requests making up I/O operations.

- If virtual memory is used for the I/O system then the programmer can improve the actual output rate if I/O system paging is avoided by writing only enough data to fill the I/O system memory, then waiting until that data is completely written to the disk before writing more data. This technique will improve the application output rate as well.

- The simulation results that were described suggest that the use of virtual memory for the I/O system can improve application output rates at the expense of actual output rates. If I/O system virtual memory is to be used, then specialized paging algorithms, such as early page-in to unused I/O system memory, can improve actual output performance.

2.5 Summary

In this chapter, two important measures of parallel file system performance were identified: the application I/O rate and the actual I/O rate. The *application I/O rate*, the rate at which data is transferred between the application's memory and the operating system, determines the time after an I/O system call during which the application must leave its I/O memory undisturbed. The *actual I/O rate*, the rate at which the data is transferred between the application's memory and the disk, determines the time that must elapse after an I/O system call before the application receives the data (during input) or before another application can access the data from the file (during output).

Three successively more detailed models of parallel file systems were developed, and the I/O performance of a simple I/O intensive application was described for each model. These models were used to explain the effects of application I/O patterns,

buffering, and I/O system virtual memory on I/O performance. Then the performance patterns predicted by these models were related to the results of experiments performed on two commercial parallel I/O systems (on the Intel Paragon and the Thinking Machines CM-5). Several I/O optimizations identified during the course of this experimentation were described.

This chapter described techniques that parallel application and library developers can use to obtain high I/O performance from parallel file systems. The following three chapters describe three abstractions that support the special I/O requirements of *object-oriented* parallel programs; these abstractions can be implemented on parallel file systems. Such implementations should incorporate the I/O optimizations identified in this chapter to increase I/O performance.

Chapter 3: Parallel Filestreams

Parallel file systems, described in the previous chapter, effectively support parallel I/O in parallel programs that have relatively simple distributed data structures, such as distributed arrays of floating point numbers. However, *object-oriented* parallel programs have special data management requirements that are not met by parallel file systems.

Object-parallel languages such as pC++ [30] support distributed arrays of arbitrary objects. Complex objects (such as tree-structured objects) often have very fragmented memory usage patterns; parallel file systems only support I/O on blocks of contiguous bytes. So before these programs can perform I/O on these distributed arrays using parallel file systems, some mechanism needs to be provided for aggregating or *flattening* this data into contiguous blocks of memory on each node before output, and for *unflattening* the data during input. So, buffering is required at the application level, along with a class-specific method for programmers to specify how object data is to be flattened and unflattened.

A further complication that arises in object-parallel programs is the fact that distributed arrays of objects may be redistributed between output and input. For many I/O tasks it is important that data written from a given object be read back into the same object, even if the array distribution or number of processors changes between output and input. So some mechanism needs to be provided to keep track of the meta-data describing the source of each data item in a file.

In the absence of programming system support for the above functionality, object-parallel application developers must hard-code this support into each program they write. A better approach is to factor this code out of applications and into language-specific class libraries within object-parallel programming systems. Such libraries can implement new I/O abstractions supporting the I/O functionality object-oriented parallel programs require. In addition, such libraries can function as *I/O middleware*, improving application portability by insulating applications from the diverse programmer interfaces of underlying parallel I/O systems on different parallel machines, and improving I/O performance by incorporating machine-specific I/O optimizations.

This chapter and the two chapters following describe designs and implementations of three new abstractions supporting I/O in object-oriented parallel programs. This chapter introduces *parallel filestreams*, an extension of ordinary C++ filestreams supporting node-buffered parallel I/O with flattening and unflattening in SPMD parallel C++ programs. *Object-parallel filestreams*, introduced in Chapter 4, extend parallel filestreams to support the element buffering required for file-format-independent I/O on distributed arrays of objects in languages such as pC++. *Distributed array streams*, described in Chapter 5, extend object-parallel filestreams to support a simplified programmer interface as well as automatic bookkeeping of array distribution meta-data within special-format files.

3.1 Goals

As mentioned in the previous section, object-oriented parallel programs require mechanisms for flattening, unflattening, and buffering object data at a level above parallel file systems. In C++, these facilities are provided by the *filestream* (*fstream*) abstraction, which is part of the standard C++ *iostream* library [6]. However, the

single-buffered design of C++ filestreams make them incompatible with existing parallel file systems; the fact that parallel I/O systems support multiple sources and sinks of data (i.e. multiple I/O buffers) spread across multiple nodes is precisely what allows them to support I/O parallelism. The goal of the work described in this chapter was to design and implement an extension of the filestream abstraction that would support parallel I/O in SPMD parallel C++ programs, and that could be implemented on existing parallel file systems.

3.2 Design

The *parallel filestream* (*pfstream*) is the abstraction that was designed to meet the above goals. Figure 3.1 depicts the architecture of this abstraction: a parallel filestream provides a single buffer on each compute node of a distributed memory parallel machine.

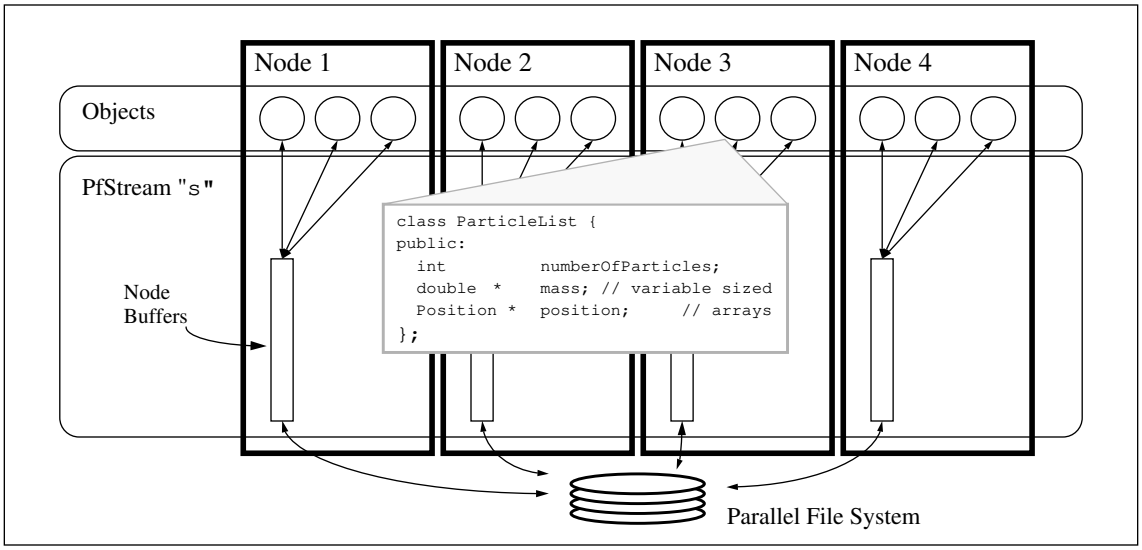


Figure 3.1: Parallel Filestream Architecture

This diagram shows the architecture of the parallel filestream abstraction, along with a set of objects on four nodes of a distributed memory parallel machine.

Ordinary Filestreams	Parallel Filestreams
Associated with a single thread	Associated with one thread per node
Single buffer	One buffer per node
Attached to a single file	
Ordinary file system	Parallel file system
Flush/load controlled by the stream	Flush/load controlled by programmer
Static buffer size	Dynamic buffer size

Figure 3.2: Comparing Ordinary and Parallel Filestreams

There are several differences between ordinary C++ filestreams and parallel filestreams; these are summarized in Figure 3.2, and described in more detail in the following paragraphs. Both abstractions support one buffer per associated thread — for filestreams, this means a single buffer, whereas parallel filestreams have a buffer per node. A parallel filestream's buffers can be flushed and loaded concurrently — this is what allows a parallel filestream to take advantage of the underlying parallel file system.

Since filestreams have one buffer associated with a single file, for filestreams, the timing of buffer flushes and loads has no effect on the contents of the file. During writing, data appears in the file in the order it was written to the buffer, no matter when flushes take place; during reading, data is loaded into the buffer in the order it was written in the file. Thus for filestreams, buffer flushing and loading can be controlled automatically by the filestream implementation itself.

In parallel filestreams as well, during writing, considering any particular node, placement of data in the file from that node depends on the order in which data is written to that node's buffer; however, since a parallel filestream has multiple buffers associated with a single file (one buffer per node), the placement of data within the file also

```
class PfStream {
 public:
  //    Opening/closing files
  void  open(const char *fileName,
             int oflag,
             int mode = 0644);
  void  attach(int fd);
  void  close();

  //    Parallel I/O Mode
  void  setParMode(ParModeType m);
  int   parMode();

  //    Loading this node's buffer
  //    from the file
  void  load();
        // (# of bytes to load)
  void  setBlockSize(int s);
  int   blockSize();
  void  load(int blockSize);

  //    Flushing this node's
  //    buffer to the file
  void  flush();

  //        Insertion functions
  //        (for buffered output)
  void insert(char* x, int nbytes);
  PfStream& operator>>(char &x);
  PfStream& operator>>(int &x);
  PfStream& operator>>(long &x);
  PfStream& operator>>(short &x);
  ...

  //        Extraction functions
  //        (for buffered input)
  void extract(char* x, int nbytes);
  PfStream& operator<<(char x);
  PfStream& operator<<(int x);
  PfStream& operator<<(long x);
  PfStream& operator<<(short x);
  ...

  //    Direct (unbuffered) I/O
  void  unbufWrite(char *x,
                   unsigned nbytes);
  void  unbufRead(char *x,
                  unsigned nbytes);
};
```

Figure 3.3: Parallel Filestream Methods

depends on the amount of data written to the buffers on the other nodes, as well as on the relative timing of buffer flushes on the nodes. So in order to have control over data placement within the file, users of parallel filestreams require control over the timing of buffer flushes (flushes cannot be performed automatically by the parallel filestream implementation, as they are in filestreams). Furthermore, during reading programmers need control over both the relative timing of buffer loads and over the amount of data to be read into each buffer during each load, since in parallel input both factors have an impact on the manner in which data in the file is distributed to the nodes.

Ordinary C++ filestreams generally have fixed sized buffers. But since parallel filestream buffers are flushed manually by the programmer during output, and since the programmer may not know the size of objects that are to be inserted into the buffers

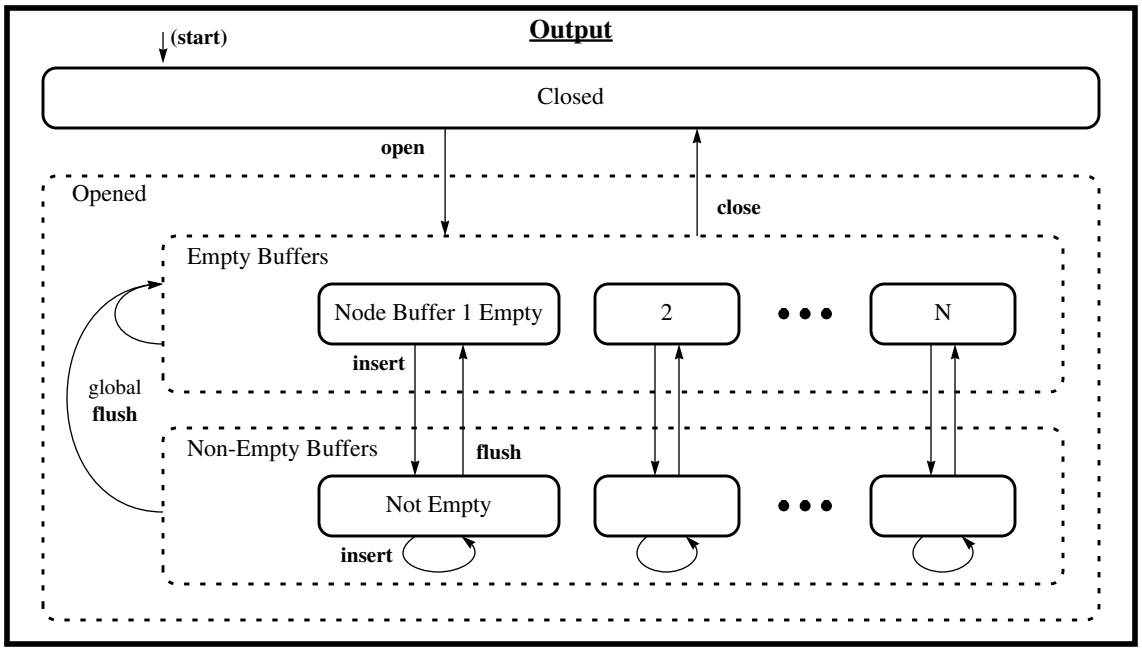


Figure 3.4: Parallel Filestream State Diagram for Output.

before flushing, parallel filestreams should have variably-sized buffers that grow (dynamically and automatically) as data is inserted into them.

Figure 3.3 lists per-node methods that implementors of parallel filestreams may wish to support, using a C++ class interface description style. (Of course the parallel filestream abstraction could be implemented in any language.) The state diagrams in Figures 3.4 and 3.5 show the order in which these methods are intended to be used. It may be useful to refer to these figures as the methods are described in more detail in the following paragraphs. `Flush()` flushes the buffer of the node(s) on which it is called, and `load()` causes the node's buffer to be loaded with data from the disk. (These will be synchronizing operations when they initiate parallel input or output). The `blocksize` methods set and get the current blocksize, which specifies how much data the calling node should read when `load()` is called. This needs to be specified by the user since on machines like the Intel Paragon, in some parallel I/O modes, different data

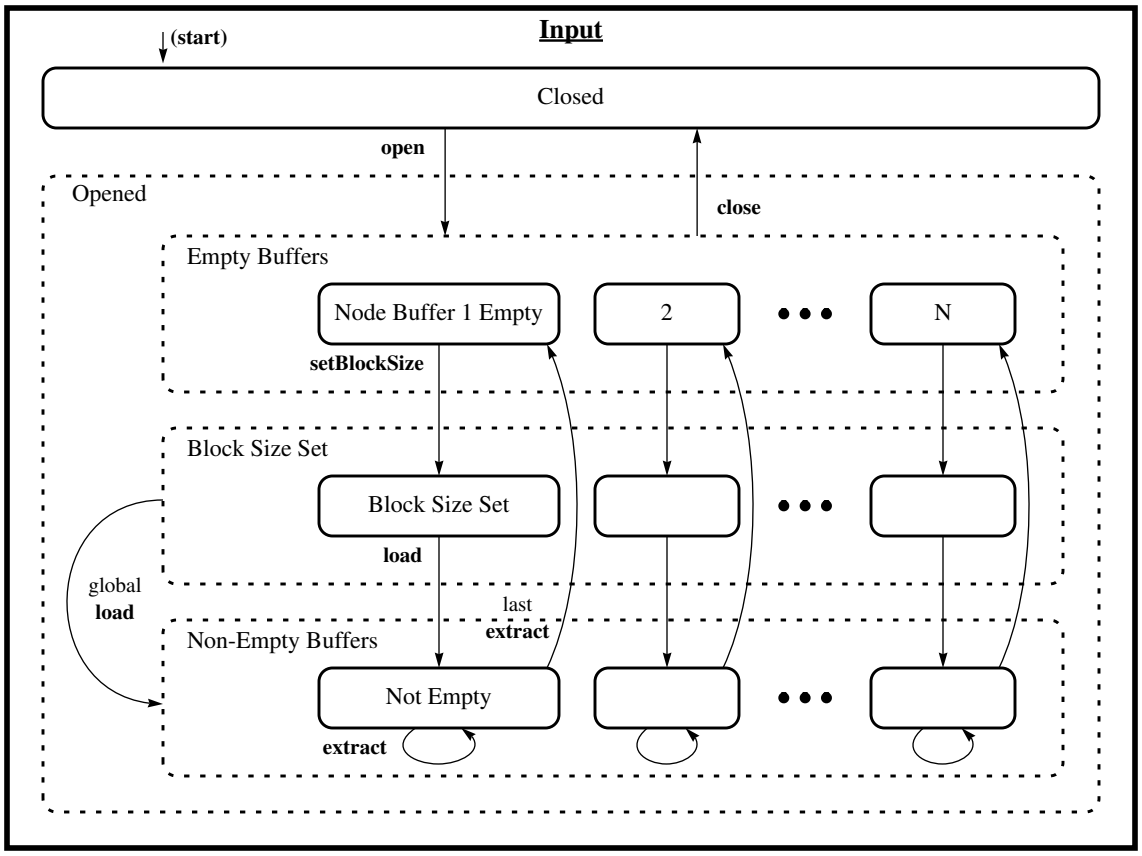


Figure 3.5: Parallel Filestream State Diagram for Input.

is is sent to a given node's buffer during input depending on the amount of data read by other nodes participating in the parallel read. In addition, methods are listed for:

- Opening and closing the file associated with a stream
- Attaching a stream to the file descriptor of a previously opened file
- Setting the parallel I/O mode, which controls the way in which data is mapped between the buffers and the file, and
- Performing unbuffered parallel I/O directly to the underlying parallel file system

The programmer uses *insertion functions* and *extraction functions* to copy data from an object into a buffer and from a buffer into an object (respectively). Insertion and extraction functions should be defined by parallel filestream implementors for all of the

fundamental (predefined) types in the language, as well as for untyped blocks of memory. In addition, programmers need a means of specifying their own insertion and extraction functions for new types they create themselves. A programmer-defined insertion function decomposes an insertion of a new type into a series of insertions of simpler types for which insertion functions have already been defined, and likewise for programmer-defined extraction functions.

3.3 Implementation

This section describes an implementation of parallel filestreams that I constructed in the summer of 1994 to support parallel I/O in SPMD C++ programs on the Intel Paragon, the Thinking Machines CM-5 and the SGI Power Challenge. The implementation is in the form of a C++ class library; details of the library's interface and internal architecture are explained. This information should be useful to those planning to implement I/O abstractions similar to parallel filestreams.

3.3.1 Implementation of Insertion and Extraction Functions

This section describes in detail one way of allowing programmers to define their own insertion and extraction functions for flattening and unflattening programmer-defined types. Insertion and extraction functions are implemented by overloading the operators << and >>, allowing a style of I/O similar to formatted text I/O in iostreams [6]:

```
buffer << object; // copy the object into this node's buffer
buffer >> object; // retrieve the object
```

The implementation defines << and >> for all the fundamental C++ types. Programmers can define their own >> and << operators for new types, as in iostreams. For example, for the following type:

```
class ParticleList {
  int       numberOfParticles;
  double*   mass;              // dynamically allocated
  Position* position;          // arrays
};
```

the programmer could define insertion and extraction operators as follows:

```
pfstream& operator<<(pfstream& s, ParticleList& p) {
  // Insert the numberOfParticles field
  // of p (an integer):
  s << p.numberOfParticles;

  // Insert the mass and position fields,
  // variable-sized arrays
  // of size numberOfParticles:
  s << array(p.mass, p.numberOfParticles);
  s << array(p.position, p.numberOfParticles);

  return s;  // allows chaining insertions (s << a << b << c;)
}

pfstream& operator>>(pfstream& s, ParticleList& p) {
  s >> p.numberOfParticles;
  s >> array(p.mass, p.numberOfParticles);
  s >> array(p.position, p.numberOfParticles);
  return s;
}
```

Pointers (such as `mass` and `position` above) require special treatment. Pointers are generally used either to reference dynamically allocated arrays or to reference dynamically allocated objects. For arrays, a macro called `array()` is defined, that takes a pointer and an integer specifying the size of the array, and returns an object of type `MemBlock` containing that information. Insertion and extraction operators for `MemBlocks` are defined by the implementation. Pointers to objects are handled by (possibly recursively) inserting/extracting the object pointed to (see Figure 3.6).

```
class MyTree {
 public:
  int data;
  MyTree *leftChild;  // NULL if no left child
  MyTree *rightChild; // NULL if no right child
};


PfStream& operator<<(PfStream& s,     PfStream& operator>>(PfStream& s,
                     MyTree* &t) {                         MyTree* &t) {
  if (t) {                              char thisNodeExists;
    s << (char)1;
    s << t->data;                       s >> thisNodeExists;
    s << t->leftChild;                  if (thisNodeExists) {
    s << t->rightChild;                   t = new MyTree;
  }                                       s >> t->data;
  else                                    s >> t->leftChild;
    s << (char)0;                          s >> t->rightChild;
  return s;                             }
}                                       else t = 0;
                                      }
```

Figure 3.6: Recursive Insertion and Extraction Operators

3.3.2 Internal Architecture

Figure 3.7 shows the implementation's class inheritance structure. There are actually two families of classes:

- `DynamicStreamBuf` plus the four other "`DynamicStream`" classes implement insertion operators, extraction operators, and a dynamically allocated buffer for a general purpose stream that can be attached to any source and sink of bytes
- `ParallelFileBuf` plus the four "`PfStream`" classes add the methods and logic necessary to attach that stream to an underlying parallel file system.

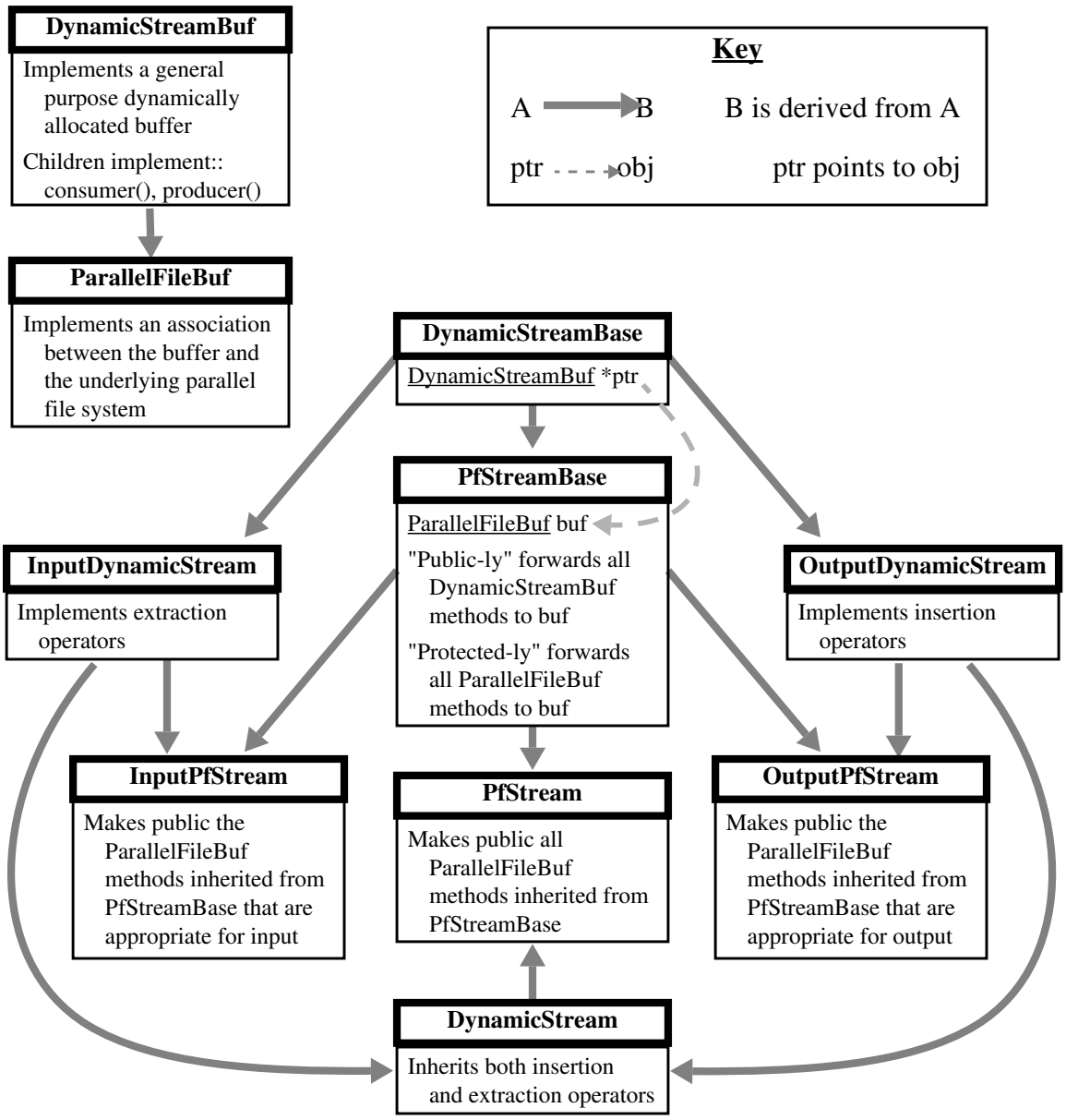


Figure 3.7: Class Inheritance Structure in a C++ Parallel Filestream Implementation

Separating the dynamic stream implementation from the parallel file system "glue" code allows the dynamically buffered stream implementation to be easily reused. The only classes in Figure 3.7 that the programmer needs to be aware of are `PfStream`, `InputPfStream`, and `OutputPfStream`; these classes make up the programmer's interface to the library. `InputPfStream` and `OutputPfStream` are similar to

PfStream but are intended to be used for either input or output but not both, so their programmer interface can be simplified somewhat.

The DynamicStreamBuf class includes the following member functions:

```
producer(char *buf, int nbytes);  // stores nbytes at buf
consumer(char *buf, int nbytes);  // copies nbytes from buf
```

These functions act as sources and sinks of bytes, respectively. A DynamicStreamBuf calls these functions when it needs to load or flush its buffer. Of course, since DynamicStreamBufs are meant to be general purpose, these functions cannot actually be implemented within DynamicStreamBuf itself, so they are "pure virtual" functions, implemented within classes derived from DynamicStreamBuf.

DynamicStream also provides the programmer with the following functions for fine-tuning the behavior of the buffers:

```
//    Dynamic buffer fine tuning
void  setBufSize(int s);
int   bufSize();
int   bytesInBufferNow();
void  setBufAdd(int a);
int   bufAdd();
void  setBufFactor(float f);
float bufFactor();
```

These allow the programmer to set the size of the buffer and to control how fast the buffer should grow when its capacity is exceeded. Buffactor is a factor by which the size should be multiplied, and bufadd is a fixed number of bytes that should be added to the size after the multiplication. BufSize is the current maximum size of the buffer, and bytesInBufferNow() returns the number of bytes that are actually in the buffer.

Middleware Functionality

In addition to supporting the parallel filestream abstraction, this library functions as parallel I/O *middleware*, encapsulating machine- and operating system-specific code and supporting a uniform abstraction for parallel I/O across disparate underlying machines and file systems. The parallel I/O modes supported by this implementation are nearly

identical to the those supported by the Paragon's and CM-5's parallel file systems, so implementation was straightforward on these machines, and was slightly more involved on machines such as the SGI Power Challenge, which lacks a parallel file systems. The implementation encapsulates machine-specific code within the `ParallelFileBuf` methods

```
void parWriteImpl(int fd, char *buf, unsigned nbytes);
void parReadImpl(int fd, char *buf, unsigned nbytes);
```

which are responsible for supporting Paragon-like parallel write and read semantics based on the current I/O mode, whether or not the underlying file system supports parallel I/O.

3.3.3 Lessons Learned

A lesson learned during the course of this implementation is that the parallel I/O model is fundamentally incompatible with the stream I/O model employed in the C++ streams library [6]. A C++ stream is a single source or sink of bytes, but parallel I/O requires multiple sources and sinks. So while a syntax similar to the C++ iostreams syntax has been used for the parallel filestreams interface, the semantics of the two interfaces are somewhat different. An initial intention was to reuse portions of the C++ filestreams library in this implementation, but the differences in the two I/O models made this impractical.

3.4 Summary

This chapter introduced issues involved in the use of parallel file systems for I/O in object-oriented parallel programs, and described the design and implementation of *parallel filestreams*, an abstraction supporting node-buffered parallel I/O in SPMD-parallel C++ programs. The following chapter extends the parallel filestream

model to support element-buffering, required for I/O in in object-parallel (pC++) programs.

Chapter 4: Object-Parallel Filestreams

This chapter describes the *object-parallel filestream* abstraction, which extends the abstraction described in the previous chapter to support parallel I/O in *object-parallel* programs. (*Object-parallelism* is a special type of object-oriented parallelism supported in the language pC++.) First the special I/O requirements of object-parallel programs are identified, then the design and an implementation of object-parallel filestreams is described.

4.1 Goals

In SPMD (Single Program Multiple Data) style parallelism, several copies of a program run concurrently on the nodes of a parallel machine, one copy per node, and each copy gets a single thread of control. The programmer may wish to perform I/O operations from any of these threads. To support buffered I/O in these programs, the parallel filestream abstraction (described in the previous section) associates one buffer with each thread — and there is one thread per node, so this means one buffer per node as well. It is important for there to be no more than one thread associated with a given buffer, since this way the threads do not interfere with each other as they read from and write to their respective buffers. And it is useful to associate exactly one buffer with each node, since

this maps well to current parallel file systems, which expect exactly one block of data to be contributed from (sent to) each node, during parallel output (input).

In pC++ [30] style *object-parallelism* (see "The pC++ Language" on Page 133) there is a single main thread of control from which parallel operations can be performed on *collections* (distributed arrays of *element* objects) by invoking a member function of the element objects on the collection itself — initiating one thread per object to execute that function. As in SPMD programs, programmers may wish to perform I/O operations from any of these element threads. However, unlike SPMD programs, if there are more elements than processors, there may be several threads per node in an object-parallel program. This means that parallel filestreams cannot easily be used for I/O in object-parallel programs, since they have only one buffer per node, and as explained above, it is important for there to be no more than one thread associated with a given I/O buffer. The goal of the work described in this chapter was to develop an abstraction that extends the functionality of parallel filestreams to support parallel I/O in object-parallel programs.

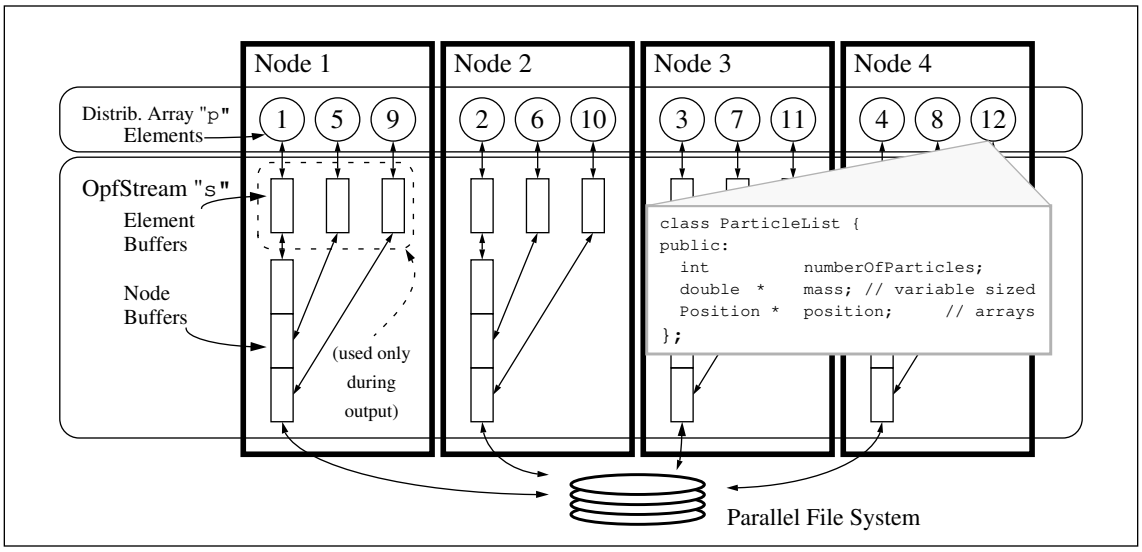


Figure 4.1: Object-Parallel Filestream Architecture

This diagram shows the architecture of the object-parallel filestream abstraction, and also depicts `p`, a one-dimensional distributed array of 12 `ParticleList` objects distributed cyclicly over four nodes of a distributed memory parallel machine.

4.2 Design

This section describes *object-parallel filestreams* (*OpfStreams*), abstractions designed to support parallel I/O on objects in object-parallel programs. Figure 4.1 depicts the architecture of object-parallel filestreams. An object-parallel filestream is similar to a parallel filestream (*pfstream*), but has not only one node buffer on each node (as in parallel filestreams), but also an *element buffer* for each element of the collection on which I/O is to be performed. So object-parallel filestreams support a two-level buffering system: during output, data is first inserted into the element buffers by the element threads, then the element buffers are flushed into the node buffers. Each node buffer accumulates data from its local element buffers; when the node buffers are flushed, parallel output to the file takes place as in parallel filestreams. During input, data is loaded from the file into the node buffers, then again loaded from each node

Parallel Filestreams	Object-Parallel Filestreams
Controlled by one thread per node	Controlled by one main thread that can invoke one thread per distributed array element
One buffer per node	One buffer per node, plus one buffer per distributed array element
Attached to a single file residing on an underlying parallel file system	
Flush/load controlled by the user of the stream	
Dynamic buffer size	

Figure 4.2: Comparing parallel and object-parallel filestreams

buffer into its local element buffers, from which it can be extracted into data structures in each element. (An implementation note: loading a group of element buffers from a local node buffer can be implemented by simply positioning the element buffers within the node buffer's memory — so during input no data movement between the node and element buffers is actually required.) This two level buffering strategy allows an object-parallel filestream to meet the dual requirements of having both one thread per buffer (so the threads can insert/extract data from their local element buffers in parallel, without synchronizing) and having one buffer per node (this is required by current parallel file systems, which expect exactly one buffer per node for parallel reads and writes).

If programmers are to have control over the order in which data appears in an object-parallel filestream file during output, and if they are to have control over the element into which a given datum in the file is loaded during input, then they must have explicit control over both the timing of buffer flushes and loads and the size of loads, for both the element buffers and the node buffers. The explanation for these requirements

```
Collection OpfStream {
 public:
  PfStream nodeBuffers;   //(defines one buffer
                          // per node)
};
```

```
class ElementStream {
 public:
  //    Loading this
  //    element buffer from the
  //    associated node buffer
  void  setBlockSize(int s);
  int   blockSize();
  void  load();
  void  loadAll();

  //    Flushing this element
  //    buffer into the
  //    associated node buffer
  void  flush();
  void  flushAll();

  // Insertion functions
  // (for buffered output)
  void insert(char* x, int nbytes);
  ElementStream& operator>>(char &x);
  ElementStream& operator>>(int &x);
  ElementStream& operator>>(long &x);
  ElementStream& operator>>(short &x);
  ...

  // Extraction functions
  // (for buffered input)
  void extract(char* x, int nbytes);
  ElementStream& operator<<(char x);
  ElementStream& operator<<(int x);
  ElementStream& operator<<(long x);
  ElementStream& operator<<(short x);
  ...
};
```

Figure 4.3: Object-Parallel Filestream Methods

The class ElementStream is intended to be the element type of collections of type OpfStream.

follows the same reasoning as the explanation for the similar requirements for parallel filestreams (Page 45). Figure 4.2 compares parallel and object-parallel filestreams.

Figure 4.3 lists methods that implementors of object-parallel filestreams may wish to support, using a pC++ style class and collection interface listing. The state diagrams in Figure 4.4 and 4.5 show the order in which the methods are intended to be used. These methods are described in more detail in the following paragraphs.

At the node level, object-parallel filestreams need to support exactly the same methods that parallel filestreams support for opening and closing files and for loading and flushing the node buffers. Programmers should be able to initiate SPMD-style parallel operations on the node buffers from the main thread of object-parallel programs.

(In Figure 4.3 this is indicated by the inclusion of a data member `nodeBuffers` of type `PfStream` in the public (SPMD) section of the `OpfStream` collection.)

At the element level, object-parallel filestreams need to support methods for loading the element buffers from (and flushing them to) the node buffers and for inserting (extracting) data into (out of) the elements of collections on which I/O is to be performed. Since the element buffers on a given node share a node buffer, in order to insure deterministic behavior during object-parallel flushes and loads, versions of element flush and load need to be provided that serialize the element buffers' access to the local node buffer during object-parallel flushes and loads. The `flushAll` and `loadAll` methods support this functionality, allowing the elements access to their node buffer in order of their element index.

Insertion and extraction should be controllable from both the main thread and from the element threads; see "Implementation of Insertion and Extraction Functions" on Page 49 for an example implementation.

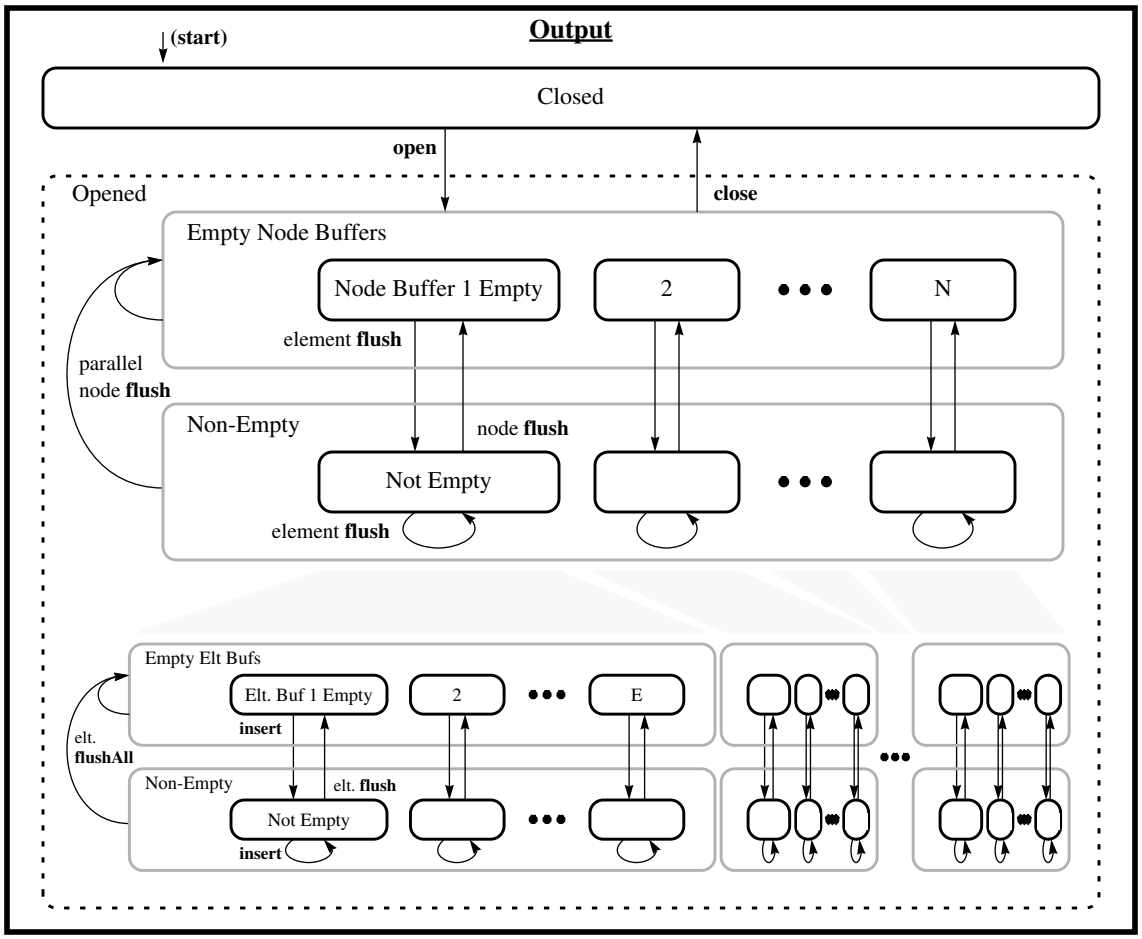


Figure 4.4: Object-Parallel Filestream State Diagram for Output

(Assuming N nodes and E element buffers per node). A state is associated with each buffer (both node and element buffers). Element buffer states are listed below the corresponding node buffer's states. All buffers start in the empty state.

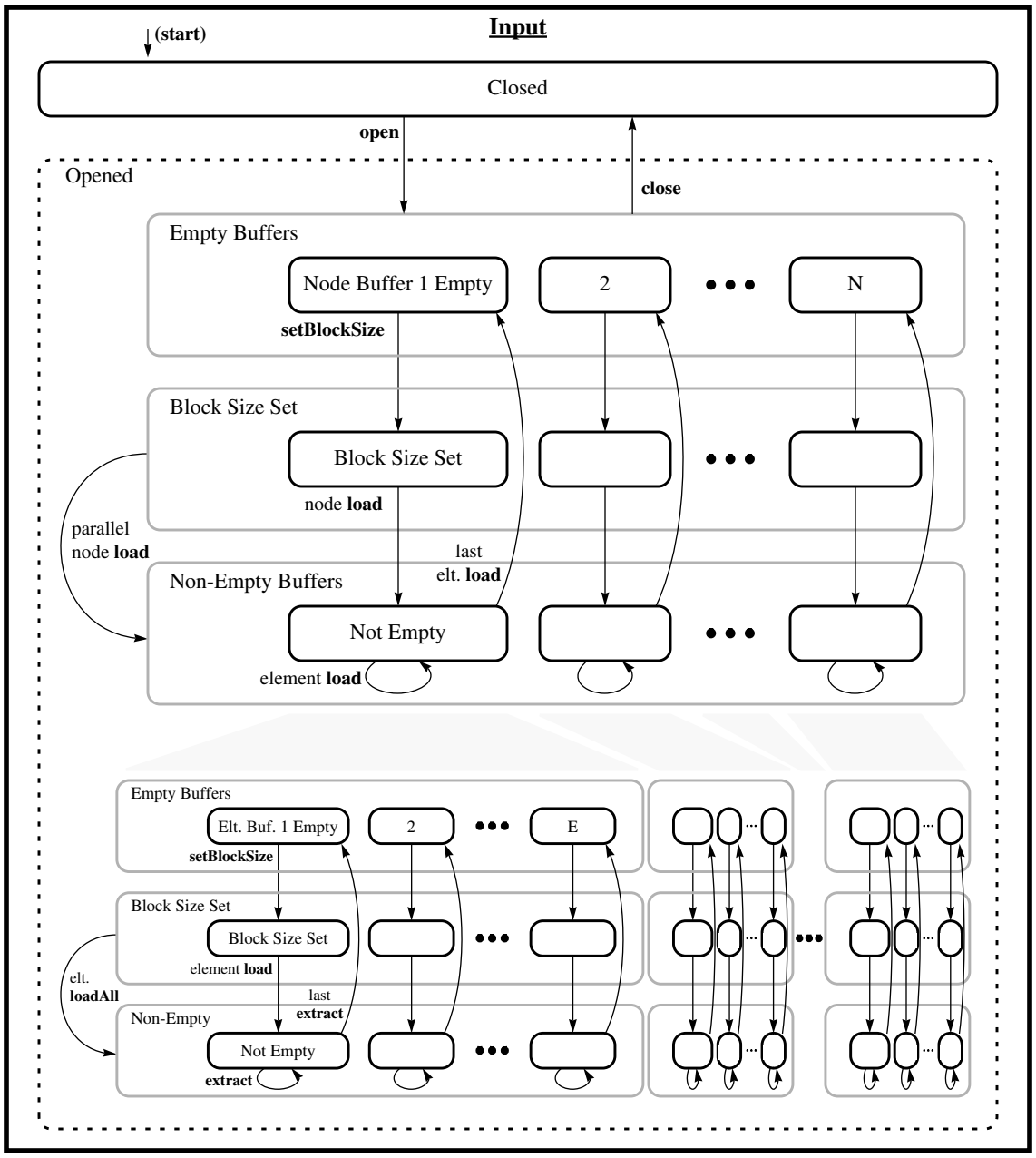


Figure 4.5: Object-Parallel Filestream State Diagram for Input.

(See the caption of Figure 4.4 on Page 62 for further explanation.)

4.3 Implementation

This section describes an implementation of object-parallel filestreams that I constructed in the summer of 1994 to support parallel I/O in pC++ programs. The implementation is a pC++ class library that builds upon and reuses a large portion of the parallel filestreams implementation described in the previous chapter. In this section details of this library's interface and internal architecture are explained.

4.3.1 Interface

Parallel Operations on Node and Element Buffers

When working with object-parallel filestreams (*OpfStreams*), programmers need mechanisms for specifying parallel operations (such as load and flush) on the stream's node and element buffers. This section describes a straightforward way that control of these operations can be supported from the main thread in pC++ implementations of object-parallel filestreams.

To support parallel operations on the node buffers, the object-parallel filestream can be implemented as a pC++ collection of element buffers. One parallel filestream (*PfStream*) object per node can be incorporated into the object-parallel filestream by placing it in the "public" part of the `OpfStream` collection.

```
Collection Opfstream {
public:
   PfStream n;
...
};
```

Then using pC++ syntax, the programmer can initiate SPMD-parallel operations (such as flush) on the node (parallel filestream) buffers using a single statement in the main thread:

```
OpfStream<ElementStream> s;
...
s.n.flush();
```

To support parallel operations on the element buffers using a similar syntax, a slightly different implementation technique is required. The element buffers can be implemented by a class `ElementStream`. A virtual reference to an `ElementStream` can be included within the MethodOfElement section of the `OpfStream` collection:

```
Collection Opfstream {
...
MethodOfElement:
   virtual ElementStream &e;
...
};
```

indicating to the compiler that a similar (but non-virtual) reference to an `ElementStream` exists in the `ElementStream` class:

```
class ElementStream {
public:
   ElementStream &e;
...
};
```

The `ElementStream` constructor should set this reference to refer to `*this` (the constructed `ElementStream` itself). Then programmers can specify object-parallel operations on the element buffers with a single line of code in the main thread:

```
OpfStream<ElementStream> s;
...
s.e.flushAll();
```

Here the programmer specifies that all element buffers should be flushed to their respective node buffers.

Insertion and Extraction Functions for Collections

This section describes pC++ implementation techniques for object-parallel filestreams that allow programmers to insert (or extract) elements of an entire collection to (or from)

an object-parallel filestream's element buffers with a single statement. (The following examples are framed in terms of insertion; extraction can be implemented similarly.)

pC++ allows parallel operations on collections to be expressed using a data-parallel style syntax. If `s` and `c` are the collections:

```
OpfStream<ElementStream> s;
MyCollection<MyElement> c;
```

and if there is an insertion operator << defined on the element types of `s` and `c`:

```
ElementStream& operator<<(ElementStream& es, MyElement& me);
```

then the meaning of the following statement in the main thread

```
s << c;
```

is that an element thread should be created for each corresponding pair of elements from `s` and `c`, and each of these threads should carry out the application of

```
ElementStream& operator<<(ElementStream& es, MyElement& me);
```

on its pair of elements. But this can be an ordinary programmer-defined insertion operator that inserts the data associated with an object of type `MyElement` into the element buffer of its associated `ElementStream`. These operators are used and defined the same way as insertion operators for parallel filestreams, described in "Implementation of Insertion and Extraction Functions" on Page 49.

The following strategy can be used to save memory during output. As data is inserted into the element buffers, in cases where the amount of data to be inserted is larger than the size of the representation of a pointer, a pointer to the data can be inserted into the element buffer instead of the actual data (since the data doesn't actually have to be aggregated until it is sent to the node buffers). The size of the data pointed to must be inserted also, and the size information must be interpreted intelligently when data is being copied from the element buffers into the node buffers.

Programmers can insert (or extract) a single field of the elements of an entire collection with a single statement in the main thread as well. Assume the following collections are defined:

```
OpfStream<ElementStream> s;
MyCollection<MyElement> c;
```

and assume `MyElement` has an integer member `intMember`. Then the `intMember` field of each of the elements of `c` can be inserted into the element buffers of `s` using the following syntax in the main thread:

```
s << c.intMember;
```

This works because in pC++, `c.intMember` is a collection of type `MyCollection<int>` having the same distribution and alignment as `c`, and whose elements are the integer members of the elements of `c`. So this is just the insertion of an entire (simpler) collection of integers into the `OpfStream`.

Now assume `d` is a second collection aligned with `c` and containing an integer field `intMemberOfD`. If the programmer invokes

```
s << c.intMember;
s << d.intMemberOfD;
```

this will cause the corresponding `intMember` and `intMemberOfD` fields of the elements of `c` and `d` to be placed contiguously in the element buffers of the `OpfStream` (so they will be written contiguously in the file), even if they are scattered in memory. This functionality is called *interleaving*.

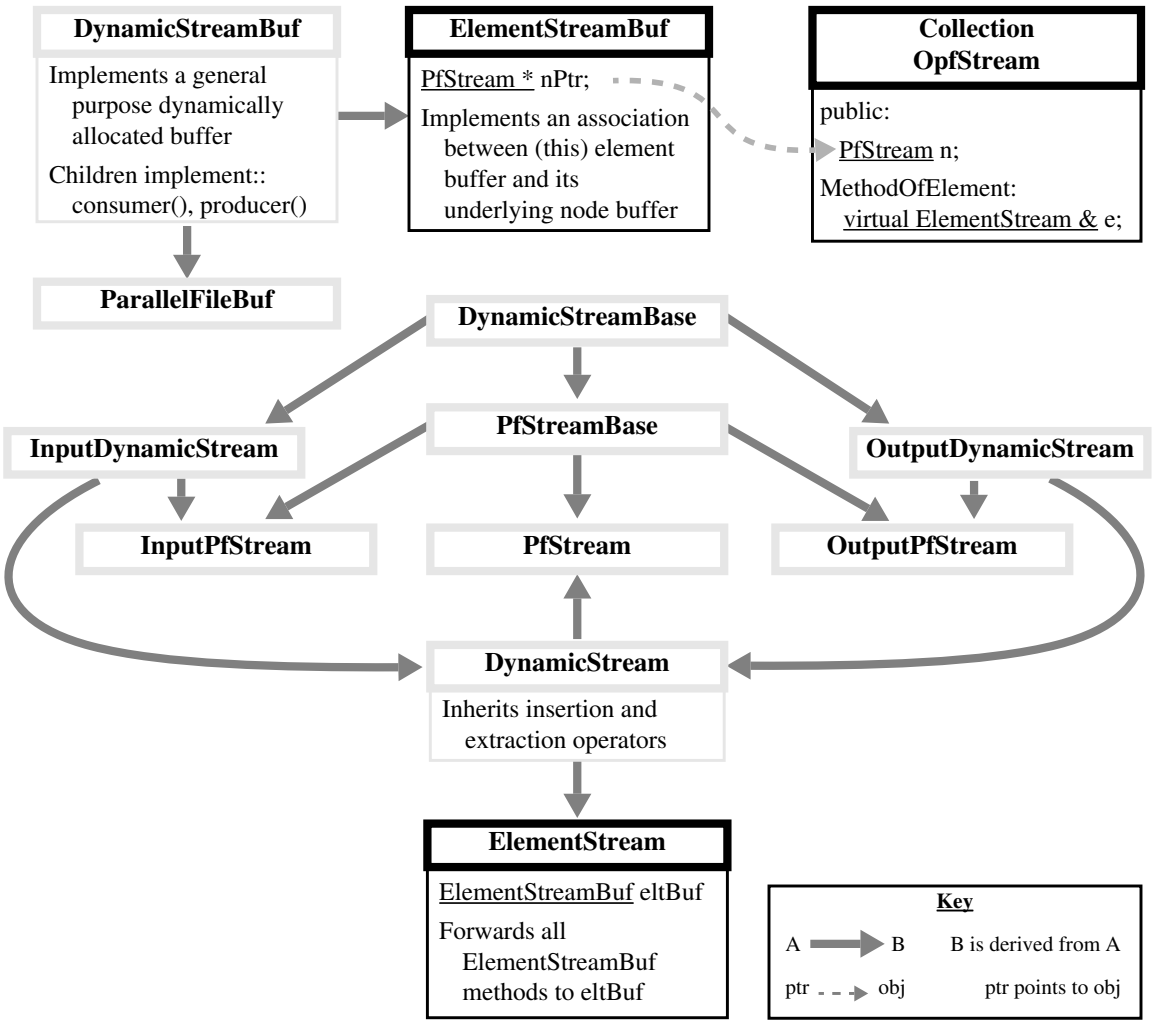


Figure 4.6: Class Inheritance Structure in a pC++ Implementation of Object-Parallel Filestreams

(Classes with gray borders are described in more detail in Figure 3.7 on Page 52.)

4.3.2 Internal Architecture

Figure 4.6 shows the class inheritance structure of the implementation. Much of the implementation of parallel filestreams (described in section 3.3 on Page 49) is reused; only three new classes were added. The `OpfStream` collection class implements the interface to the node buffers, which can conveniently be implemented by containing a single parallel filestream within the object-parallel filestream. (Note: This works in

pC++ 1.0 because in this version of pC++, non-MethodOfElement data members of collections are always duplicated one per node.) An `ElementStreamBuf` class (derived from `DynamicStreamBuf`) implements a single element buffer. Recall that the `producer()` and `consumer()` functions of `ParallelFileBuf` read from and wrote to an underlying parallel file system (see Page 53); in `ElementStreamBuf`, these functions read from and write to their associated node buffer in the `OpfStream` collection. To facilitate coding the data movement between the element and node buffers, each `ElementStreamBuf` maintains a pointer to its associated node buffer. An instance of `ElementStreamBuf` is contained in the `ElementStream` class, whose main purpose is to inherit insertion and extraction operators from `DynamicStream`. `ElementStream` contains facilities for dynamic buffer fine-tuning similar to those in the parallel filestream implementation (see Page 53).

To instantiate a complete `OpfStream`, the programmer declares a collection of type `OpfStream<ElementStream>` (an `OpfStream` collection having elements of type `ElementStream`), and gives this collection the same distribution and alignment as the collection(s) on which I/O is to be performed. This ensures there is a one-to-one correspondence between the `ElementStreams` and the elements of the collection on which I/O is to be performed.

4.4 Summary

This chapter discussed the design and implementation of the object-parallel filestream abstraction, an extension to parallel filestreams that can be used to support parallel I/O in object-parallel programming systems such as pC++. The following chapter describes several ways in which object-parallel filestreams can be extended to make their use easier for programmers.

Chapter 5: Distributed Array Streams

This chapter describes the design and an implementation of the *distributed array stream*, an abstraction that extends the object-parallel filestream to support a simplified programmer interface as well as automatic redistribution of element data (which is necessary when the element distribution or number of processors changes between output and input).

5.1 Goals

The use of parallel filestreams (Chapter 3) and object-parallel filestreams (Chapter 4) is somewhat more complex than the use of ordinary C++ filestreams. Ordinary filestreams behave like simple first-in first-out queues that reside on persistent storage. Parallel and object-parallel filestreams behave somewhat like queues as data is being inserted and extracted from a given element or node buffer; however, in addition to being concerned with the order in which data are inserted and extracted from each buffer, the programmer must also control the flushing and loading of these buffers. Could this complexity be reduced by an improved design? Probably not, given the requirements of parallel I/O:

1) **Buffering is required for efficient I/O**: If I/O is to be efficient, I/O hardware requires that data be aggregated into relatively large chunks before output is performed. Likewise, I/O hardware returns data to memory in large chunks, from

which it must be de-aggregated. So I/O abstractions must support buffering for aggregation and de-aggregation.

2) **Multiple buffers are required for parallel I/O**: In order to support simultaneous I/O involving multiple nodes of a parallel machine, there must be at least one buffer for aggregation/de-aggregation on each node participating in the I/O. In the case of object-parallel filestreams, if multiple elements on a single node are to be able to aggregate/de-aggregate data in parallel (without synchronizing), then object-parallel filestreams must support an additional buffer per element.

3) **If the parallel I/O is to store or access data in a (single) file, and if the programmer is to control the file's format, then the mapping between the buffers and the file cannot be controlled automatically;** there must be some mechanism by which the programmer specifies this mapping. Giving the programmer control over buffer flushes and loads, as the file is written or read from beginning to end, is one way for the programmer to specify this mapping; this is the approach that has been taken in the design of parallel and object-parallel filestreams.

On the other hand, if the above requirements are relaxed, then it may be possible to simplify the coding of parallel I/O in application programs:

- **If data can be stored in multiple files**, then each file can be associated with a single thread (i.e. a single processor), so each processor can use a simple, separate abstraction (such as an ordinary C++ filestream) for I/O. This is often the first I/O approach supported by developers of new parallel systems, since it is relatively simple to implement. This is a workable approach for I/O supporting checkpoint/restart tasks, but there are several potential drawbacks to using this approach for initial input or final output. One problem is that SPMD programs

running on a large number of nodes will generate and consume a large number of files, and the work involved in managing all of these files may not be trivial. A more serious problem is the fact that parallel programs often need to be able to be run with widely varying degrees of parallelism, which often means using a widely varying number of processors. So if there is one file per processor then the number of input files required and output files generated will vary from run to run, and the user must redistribute the input data into separate input files for each degree of parallelism desired. Furthermore, most visualization tools and other programs used to analyze output data expect the data to reside in a single file, so multiple output files will need to be consolidated before analysis can take place.

- **If the requirement that data be stored in files is dropped**, then alternate persistent storage mechanisms such as object-oriented database management systems could be used to support I/O tasks. This approach is explored further in Chapter 6, "Persistent Collections".

- **If the requirement that the programmer have complete control over the file format is relaxed** then some aspects of buffer flushing and loading can be automated, making things easier for application developers, while maintaining the benefits of storing data in a single file.

The goal of the work described in this section was to simplify the programmer's interface for parallel I/O in object-parallel languages like pC++, taking the latter approach above, i.e. relaxing the requirement that the programmer have complete control over the file format.

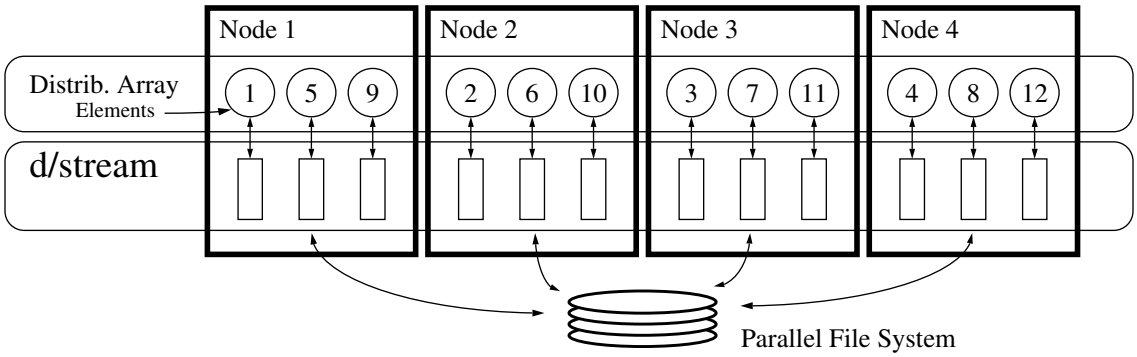


Figure 5.1: Architecture of the D/stream Abstraction

5.2 Design

The *distributed array stream* (*d/stream*) is an abstraction for parallel I/O in object-parallel languages such as pC++. Like object-parallel filestreams, d/streams support parallel I/O between distributed arrays of objects and a single file; however, d/streams require a that a fixed (d/stream specific) file format be used. This allows d/stream implementations to automatically record information within the file about the size and distribution of the objects stored in the file, as the file is being written. Then when reading, d/stream implementations can automatically route data to the correct distributed array elements, even if these elements appear on different nodes than when the file was written (this may occur if the distribution of the array or the number of nodes has changed since the array was written). In addition, since information about the size of objects in d/stream files is stored within the files themselves, d/streams do not require programmers to specify the amount of data to load into each buffer during input (as is required in parallel and object-parallel filestreams; see "Goals" on Page 43 for further explanation). Programmers can simply invoke "load" and the d/stream implementation determines the amount of data that should be loaded into each buffer.

Figure 5.1 illustrates the architecture of the d/stream abstraction, which is described further below. Like object-parallel filestreams, d/streams have one buffer per distributed

Object-Parallel Filestreams	D/streams
Can read and write any file format	Must read and write a special file format
One buffer per node, plus one buffer per distributed array element	One buffer per distributed array element
Buffer flushing/loading can be individually controlled	Buffer flushing/loading methods act on all buffers as a group
User must manage mapping from writing elements to file to reading elements using fine control of buffer flushing and loading	Stream implementations manage the mapping automatically; users have only coarse control over buffer flushing/loading
Controlled by one main thread that can invoke one thread per distributed array element	
Attached to a single file residing on an underlying parallel file system	
Dynamic buffer size	

Figure 5.2: Comparing Object-Parallel Filestreams and D/streams

array element. Unlike object-parallel filestreams, loads and stores on these element buffers are synchronizing operations, and the mapping between the element buffers and the file is fixed (the programmer cannot vary the mapping by varying the timing of loads and stores of the element buffers). Though d/stream implementations may make use of node buffers, node buffering is transparent to programmers using d/streams. So node buffering is not part of the architecture of the d/streams abstraction, though it may be a part of a given d/streams implementation architecture. Figure 5.2 summarizes these differences between object-parallel filestreams and d/streams.

Figure 4.3 summarizes the methods that implementors of d/streams may wish to support, using a pC++ style interface listing; these methods are described in more detail below. Insertion and extraction of distributed array data are controllable both from element threads and from the main thread (for insertion of entire distributed arrays), as in object-parallel filestreams (see "Insertion and Extraction Functions for Collections" on

```
Collection DStream {
 public:
  void open(char* name, int oflag, int prot=0644);
  void close();

  void flush();
  void load();
  void unsortedLoad();
}
```

```
class ElementDStream {
 public:
  // Insertion functions
  // (for buffered output)
  void insert(char* x, int nbytes);
  ElementStream& operator>>(char &x);
  ElementStream& operator>>(int &x);
  ElementStream& operator>>(long &x);
  ElementStream& operator>>(short &x);
  ...

  // Extraction functions
  // (for buffered input)
  void extract(char* x, int nbytes);
  ElementStream& operator<<(char x);
  ElementStream& operator<<(int x);
  ElementStream& operator<<(long x);
  ElementStream& operator<<(short x);
  ...
};
```

Figure 5.3: D/stream Methods.

The class `ElementDStream` is intended to be the element type of collections of type `DStream`. Both `load` and `unsortedLoad` transfer a block of data (written by a corresponding `flush`) from a file into an input d/stream's buffer. Implementations of `unsortedLoad` may have better input performance when reading un-ordered distributed arrays of objects.

Page 65 for an example of how this can be implemented). The d/streams buffer management methods (`flush()`, `load()`, and `unsortedLoad()`) are callable from the main thread only. `Flush()` flushes all of the data in all of the element buffers to the file using a file format that should be clearly defined by implementations. `Load()` and `unsortedLoad()` read the data written by a corresponding `flush()` back into

the element buffers. So the programmer needs to be aware of element buffering, but not node buffering.

Note that unlike parallel and object-parallel filestreams, d/streams do not require the programmer to specify the amount of data that should be loaded, since this is recorded in the file by `flush()`. Furthermore, the d/stream `load` and `flush` methods are much more intelligent than the corresponding methods in parallel and object-parallel filestreams, in the following respect. In parallel and object-parallel filestreams, the mapping of data from threads performing output to threads performing input is affected by the pattern of insertions and extractions, the pattern of `flush`es and `load`s, the distributions of the data structures from which output and input are performed, and the number of nodes that participate in output and input — so the programmer must take all these factors into account when using these abstractions for I/O. When using d/streams, the only variable the programmer needs to be concerned with is the pattern of insertions and extractions; this pattern must be the same for output and input, i.e. the same types must be inserted/extracted from the element threads in the same order during output and input.

The `flush` and `load` methods exist only to allow programmers to control interleaving (see Page 67) and buffer memory usage. Implementations in which these are not important issues may choose to fully automate loading and flushing. For a given `flush/load` pair, d/stream implementations should guarantee that `load` returns blocks of bytes to exactly the same element buffers from which they were `flush`ed, even if the array distribution or number of nodes (and thus the number of elements per node) changes between output and input. For `unsortedLoad`, implementations must guarantee only that each of these blocks was written from some (single) element, though not necessarily the one with the same element index. Thus `unsortedLoad` can be faster than load in some implementations. `UnsortedLoad` is intended to be used to

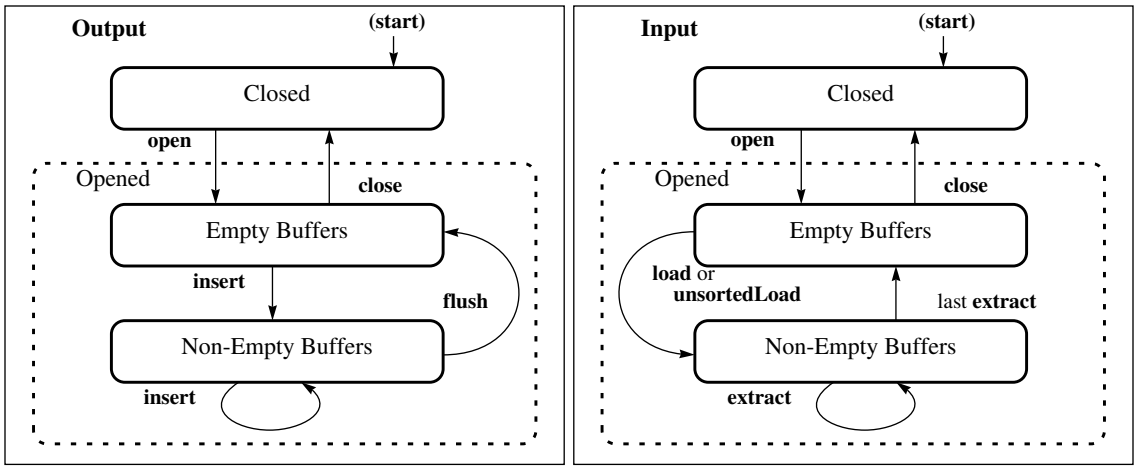


Figure 5.4: D/stream State Diagrams

Specify the order in which the d/stream methods can be used.

read array data in which the element indices perform no important role in the computation.

The state diagrams in Figure 5.4 show the order in which the d/stream methods are intended to be called; this is similar to the usage of ordinary C++ iostreams. Several constraints on the use of the methods cannot be indicated in the state diagrams so they are discussed here. Data must be read in the same order it was written. Specifically:

- When a d/stream file is read by an input d/stream, every `load` or `unsortedLoad` must correspond to a `flush` that occurred when the file was written, and every `extract` must have a corresponding `insert`.
- Each `extract`ed object or array must have the same size and type as the corresponding array that was `insert`ed.

5.3 Implementation

This section describes a pC++ implementation of distributed array streams that I constructed in the summer of 1994. Like implementations of the abstractions described

earlier, this implementation is a pC++ class library built on a more primitive library (the object-parallel filestreams library described in the previous chapter).

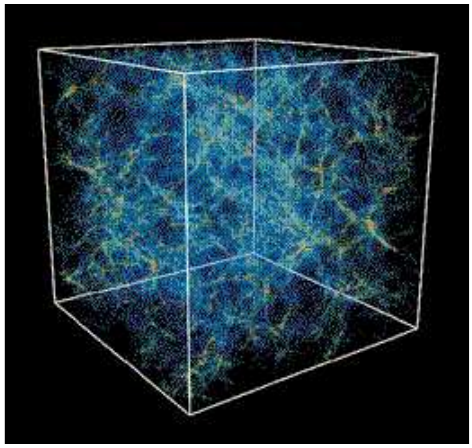
5.3.1 Interface

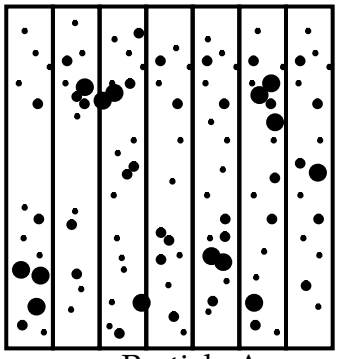
This section illustrates how the d/streams interface was implemented in pC++ by showing how the library described in this section would be used to code checkpoint/restart tasks in an example application. Assume the application simulates the behavior of particles in 3D space. Information about the particles can be stored in a distributed array of particle lists, where each list keeps track of the particles in a 3D slab of space. Figure 5.5 shows the declarations that would be used to declare this data structure (called `particleArray`) in pC++. Using d/streams to write a procedure to checkpoint the entire particleArray from the main thread is fairly straightforward:

```
#include "particleArray.h"
#include "pc++streams.h"

void saveParticleArray() {
  DStream stream(&d, &a, "myCheckpointFileOne");
  stream << particleArray;
  stream.flush();
}
```

It is also possible to checkpoint a single field of the elements of the particleArray:

```
#include "particleArray.h"
#include "pc++streams.h"

void saveNumberOfParticles() {
  DStream stream(&d, &a, "myCheckpointFileTwo");
  stream << particleArray.numberOfParticles;
  stream.flush();
}
```






```
class Position {
     double              x, y, z;
};

class ParticleList {
     int                 numberOfParticles;
     double  *           mass;                    // variable sized
     Position*           position;                // arrays
};

Processors          P;
Align a(12,"[ALIGN(collection[i], template[i])]");
Distribution d(12, &P, CYCLIC);

Collection DistributedArray {
     updateParticles();
};

DistributedArray<ParticleList> particleArray(&d,&a);
```

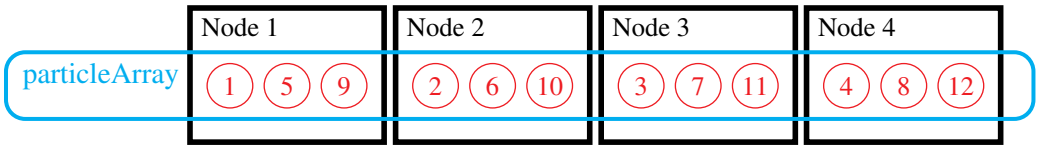


Figure 5.5: Declaring a Distributed Array of Lists of Particles in pC++

(These declarations are referred to as particleArray.h subsequently).

Functions to restore the chekpointed data can be written similarly:

```
#include "particleArray.h"
#include "pc++streams.h"

void loadParticleArray() {
  DStream stream(&d, &a, "myFileOne");
  stream.load();
  stream >> particleArray;
}

void loadNumberOfParticles() {
  DStream stream(&d, &a, "myFileTwo");
  stream.load();
  stream >> particleArray.numberOfParticles;
}
```

In the procedures `saveParticleArray()` and `loadParticleArray()` above, the statements

```
stream << particleArray; // and
stream >> particleArray;
```

apply the operators << and >> to corresponding pairs of elements of the collections `stream` and `particleArray`, i.e. they invoke the operators

```
DElementStream& operator<<(DElementStream& eltStr,
                           ParticleList& p);
DElementStream& operator>>(DElementStream& eltStr,
                           ParticleList& p);
```

on each pair of elements (see Page 65 for more details). The programmer needs to define these operators to tell the d/streams implementation how `ParticleLists` should be inserted into and extracted from element streams.

```
class ParticleList {
public:
  int  numberOfParticles;
  double* mass        // dynamically allocated,
  Position* position; // variable sized arrays
};
```

The programmer can accomplish this by making the following declarations in particleList.h:

```
declareStreamInserter(ParticleList &p) {
  eltStr << p.numberOfParticles;
  eltStr << array(p.mass, p.numberOfParticles);
  eltStr << array(p.position, p.numberOfParticles);
}

declareStreamExtractor(ParticleList &p) {
  eltStr >> p.numberOfParticles;
  eltStr >> array(p.mass, p.numberOfParticles);
  eltStr >> array(p.position, p.numberOfParticles);
}
```

In this d/streams implementation, `declareStreamInserter()` and `declareStreamExtractor()` are macros defined in pC++streams.h; the compiler transforms these into the appropriate declarations:

```
DElementStream& operator<<(DElementStream& eltStr,
                           ParticleList& p)
```

and

```
DElementStream& operator>>(DElementStream& eltStr,
                           ParticleList& p)
```

simplifying the required syntax for the programmer. The `array()` macro is used to insert/extract dynamically allocated array data; see Page 50 for further details.

Figure 5.6 summarizes the actions that occur when `saveParticleArray()` is called. The statement

```
stream << particleArray;
```

in the main thread invokes the parallel application of

```
DElementStream& operator<<(DElementStream& eltStr,
                           ParticleList &p)
```

in the element threads, on corresponding pairs of elements from `stream` and `particleArray`. This operator is defined by the programmer in particleArray.h, using the macro `declareStreamInserter(ParticleList &p)`. The insertion operators (and the `array()` macro) used there are defined by the d/streams implementation (in pc++streams.h). If the `ParticleList` class had contained data

```
void saveParticleArray() {
   DStream<DElementStream>       stream(&d, &a, "myFileOne");
   DistributedArray<ParticleList> particleArray(&d,&a);

   stream        << particleArray; // in the main thread, invokes:
   DElementStream << ParticleList;  // in each element thread
}
```

in particleArray.h:

```
declareStreamInserter(                              ParticleList &p) {
DElementStream& operator<<(DElementStream& eltStr, ParticleList &p) {
     eltStr << p.numberOfParticles;
     eltStr << array(p.mass, p.numberOfParticles);
     eltStr << array(p.position, p.numberOfParticles);
}
```

in pc++streams.h:

```
MemBlock array(int    *base, int size) {...}
MemBlock array(float  *base, int size) {...}
MemBlock array(double *base, int size) {...}
...

class DElementStream {
     public:
        DElementStream& operator<<(int      x) {...}
        DElementStream& operator<<(float    x) {...}
        DElementStream& operator<<(double   x) {...}
        ...
        DElementStream& operator<<(MemBlock x) {...}
        ...
}
```

Figure 5.6: Summary of Checkpointing Using D/streams

members of other programmer-defined types, then programmer-defined insertion functions for those types (defined by other macros similar to `declareStreamInserter(ParticleList &p))` would be called within `declareStreamInserter(ParticleList &p)`.

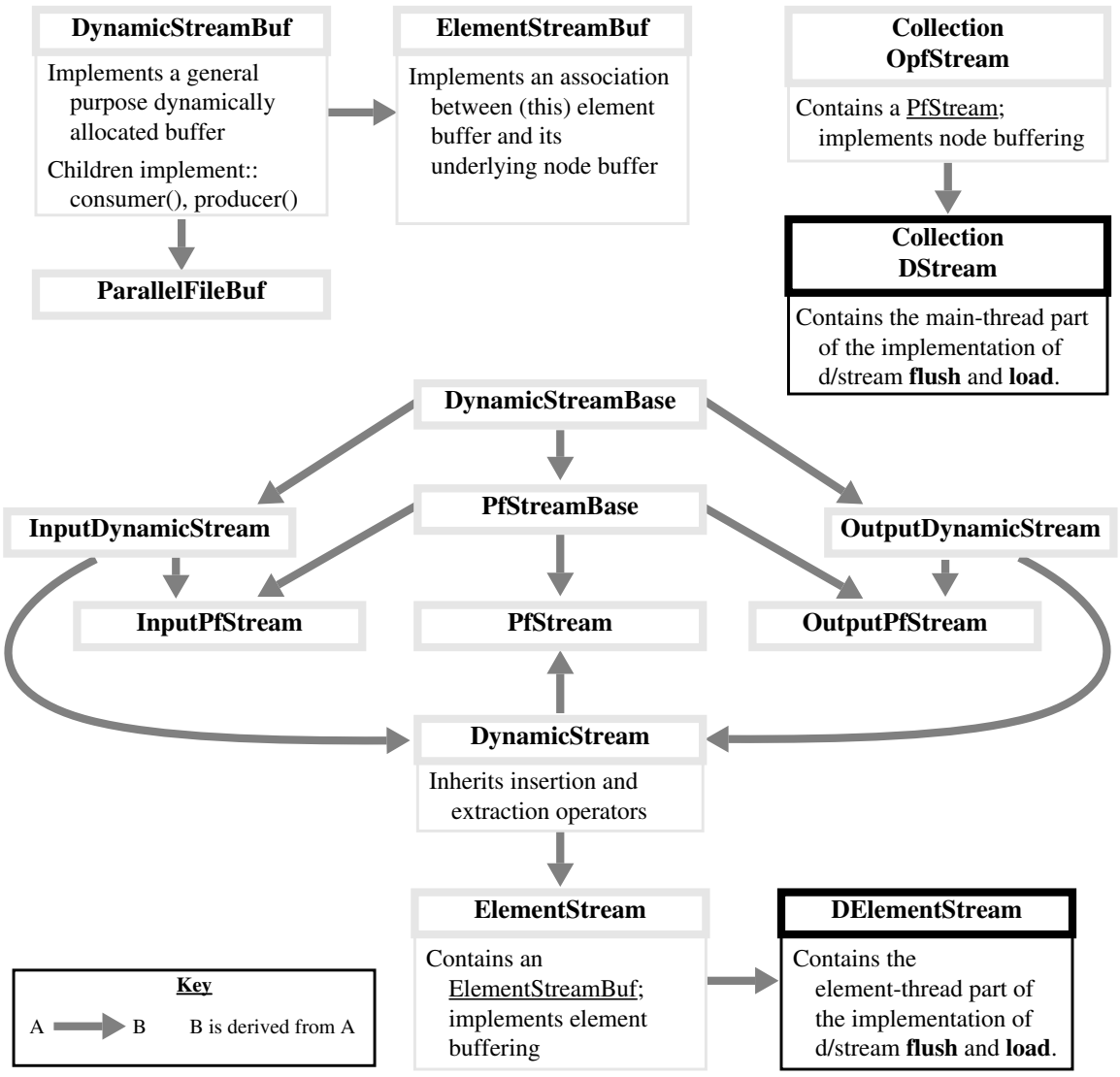


Figure 5.7: Class Inheritance Structure in a pC++ D/streams Implementation (Classes with gray borders are described in more detail in figures on pages 52 and 68.)

5.3.2 Internal Architecture

Figure 5.7 shows the class inheritance structure of my pC++ implementation of d/streams. Note that since this implementation is built directly on the implementation of object-parallel filestreams described in the previous chapter, it was possible to reuse both the object-parallel filestreams implementation (Section 4.3, Page 64) and the parallel

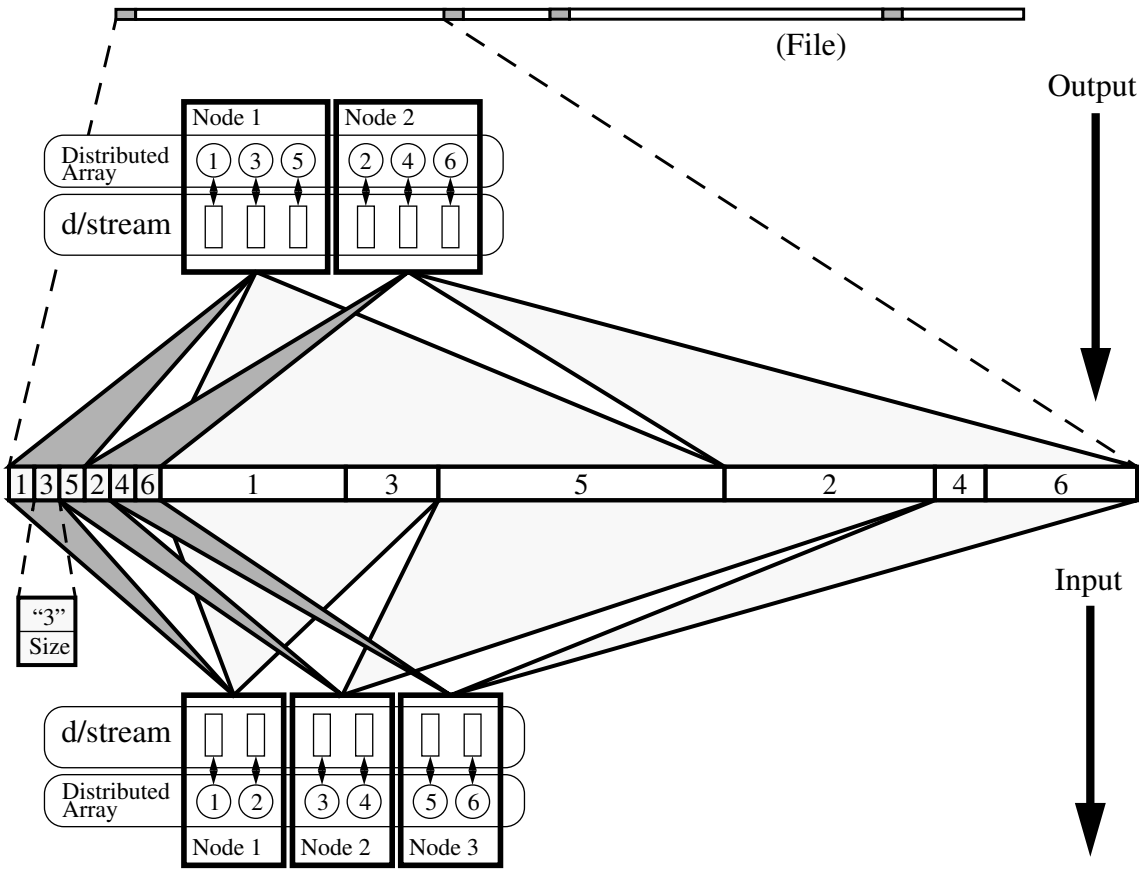


Figure 5.8: A Possible File Format for D/streams Implementations

filestreams implementation (Section 3.3, Page 49). There are two new classes. The `DStream` collection class is derived from the object-parallel filestream collection, and contains the main-thread part of the implementations of the flush and load methods. `DElementStream`, derived from `ElementStream`, contains the element-thread part of the implementation. So to instantiate a complete d/stream, the programmer declares a collection of type `DStream<DElementStream>` (a `DStream` collection having elements of type `DElementStream`), and as with the object-parallel filestream implementation, the programmer must give this collection the same distribution and alignment as the collection(s) on which I/O is to be performed, so there is a one-to-one correspondence between element buffers of the stream and elements of the source/destination collection.

Figure 5.8 depicts a file format appropriate for d/streams; this is described in greater detail below. Assume the programmer declares

```
DStream<DElementStream> s(...);
```

in the main thread. The **flush** method is invoked from the main by simply calling

```
s.flush();
```

which causes the stream implementation to perform the following steps:

1) **Writing distribution and size information**: Information about the distribution of the collection `s` (and thus the distribution of all collections that could have inserted data into `s`), and about the size of the data to be output from each element, needs to be written to the file prior to the actual data, so that the implementation of `load` will know how much data is needs to be read and where the data belongs. The implementation writes this data from all the nodes concurrently, using a parallel write operation. For collections having a large number of elements (and thus a large amount of distribution and size information to write), this is a good strategy. For smaller collections, due to the latency concerns involved in writing small chunks of data, it may be more efficient to communicate this information to a single node (node zero on the Paragon and CM-5 would be an appropriate choice). Then instead of writing this information in a separate parallel output step, it can be placed at the head of that node's buffer so that it can be written with the actual data in step 2 (this data needs to appear before the actual data in the file; thus node zero should be the choice on the Paragon and CM-5).

2) **Writing the actual data**: Next the data in the element buffers (or referenced by pointers in the element buffers) is aggregated into the node buffers, then all the node buffers are written to the file in a single parallel write operation, using the parallel I/O primitives of the underlying parallel file system. On the Intel Paragon

and Thinking Machines CM-5, the implementation uses parallel I/O primitives that transfer a block of data from each compute node to the file system simultaneously, and that write those blocks to the file in order of node number.

The programmer invokes the d/stream `load` method calling `s.load()`, and the `unsortedLoad` method by calling `s.unsortedLoad()`. When either method is invoked, the implementation first reads the distribution and size information preceding the actual data, using one parallel read operation, then reads the actual data into the node buffers using a second parallel read. (Note that no information about the distribution or size of the data to be read needs to be passed to the library by the programmer when reading, since that information is stored directly in the file, preceding the data itself.) If `unsortedLoad` was invoked, then at this point the implementation sets up each element buffer so that it falls within the appropriate region in its local node buffer, and the data is ready to be `extract`ed. If `load` was invoked, then if the number of nodes or the distribution of the array is different than when the array was written, the blocks of data that were read into the node buffers may need to be sent to the actual owner nodes before the element buffers can be set up.

Implementing `flush` and `unsortedLoad` using the facilities provided by the underlying object-parallel filestreams implementation is a fairly straightforward task. `Flush` is implemented by the following main-thread procedure:

```
void DStream::flush() {
  // set the parallel I/O mode to syncpar
  // meaning all nodes share a single file pointer,
  // and in a parallel flush, a block of data is sent from
  // each node to the file in node number order
  n.setparmode(pfilebuf::syncpar);

  // write the size of each element buffer
  // into its local node buffer
  n_referencedFromElement() << e.bytesInBufferNow();
  // flush the node buffers to the file
  n.flush();
```

```
  // flush the actual data in each element buffer
  // into its local node buffer
  e.flushAll();
  // flush the node buffers to the file
  n.flush();
}
```

The object-parallel filestream implementation makes it simple to specify parallel stream operations from the main thread; SPMD-parallel operations on all of the node buffers can be specified using `n.operation()`, and object-parallel operations on all of the element buffers can be specified using `e.operation()`. Within an object-parallel operation on the element buffers, the function `n_referencedFromElement()` can be used to refer to the local node buffer. (See Page 64 for more details). The call `e.flushAll();` flushes the element buffers to their respective node buffers in the order of their element indices on each node.

`UnsortedLoad` is implemented as follows:

```
void DStream::unsortedLoad() {
  // set the parallel I/O mode to syncpar
  // meaning all nodes share a single file pointer,
  // and in a parallel load, a block of data is sent from
  // the file to each node in node number order
  n.setparmode(pfilebuf::syncpar);

  // on each node, load a single integer from the file for
  // each local element (this is the size of the block of
  // data to be loaded for each local element)
  n.setBlockSize(elementsOnThisNode * sizeof(int));
  n.load();
  // on each node, for each local element, read an integer
  // from the local node buffer and set the element buffer's
  // block size accordingly
  e.setBlockSizeFromNodeBuffer();

  // on each node, sum the block sizes of the local element
  // buffers so the node's blocksize can be set to the
  // total amount of data to load into the node
  int nodeBufferSize = 0;
  ResetLocal(this);
  for (int i=FirstLocal(); i>=0; i=Local(i))
    nodeBufferSize += (*this)(i)->e.blocksize();
  n.setBlockSize(nodeBufferSize);
```

```
    // for each node buffer, load that node buffer's (blocksize)
    // bytes from the file (in order of node number)
    n.load();
    // for each node, for each local element buffer,
    // load that element buffer's blocksize bytes from the
    // node buffer, in order of element index
    e.loadAll();
  }
```

(Note: in both `flush` and `unsortedLoad`, the movement of the elements' buffer-size/blocksize data between the elements and node buffers needs to be serialized in order of element index on each node. No explicit code is needed to perform this serialization on the current implementation of pC++, since pC++ itself automatically serializes object-parallel operations on each node in order of element index.) `UnsortedLoad` requires one object-parallel helper function in `DElementStream` for reading each element's blocksize from its local node buffer:

```
void DElementStream::setBlockSizeFromNodeBuffer() {
  int size;

  // extract this element's blocksize from the local node buffer
  n_referencedFromElement >> size;
  setblocksize(size);
}
```

Note that the above code does not write the element indices (distribution information) to the file with the sizes of each element buffer, nor does it read the distribution information in `unsortedLoad`. `UnsortedLoad` does not require distribution information, since it reads element data in whatever order it appears in the file. However, `load` does require distribution information. `Load` can be implemented as follows. In `flush`, each element should send two integers to the node buffer before the actual data is flushed: the element buffer size (as above) and the element's index. In `load`, the element index data should be read along with the element size data; then after the actual data has been read, each node knows the actual element index corresponding to each of the data blocks in the node buffer. In addition, each node knows the current distribution of the receiving array (which has the same distribution as the stream itself). So each node should communicate the sizes of its element data blocks to the correct

owner nodes. Then each node should allocate a new node buffer of the appropriate size to hold the element data it is about to receive, after which each node can send its element data blocks to the correct owner nodes, where they can be stored in the newly allocated node buffers.

5.4 Summary

For the programmer, coding parallel I/O on distributed arrays is much simpler with distributed array streams than with object-parallel filestreams. The price for this simplicity is the programmer's loss of control over the file format. Object-parallel filestreams is the more general I/O abstraction, since it supports parallel I/O on files of any format. But just as object-parallel filestreams can simplify the implementation of d/streams, it can also simplify the implementation of application-specific parallel I/O mechanisms having application-specific file formats. So d/streams should be viewed as just one possible simplified special-format parallel I/O abstraction that can be built on object-parallel filestreams.

The next chapter looks at a totally different approach to I/O in object-oriented parallel simulations: persistence.

Chapter 6: Persistent Collections

Normally, elements of pC++ collections are transitory, i.e., their data disappears when the program terminates. In order to preserve transitory data, the programmer must write code to output that data to a file before the program terminates, using either an operating system supported file I/O mechanism or a higher-level abstraction such as distributed-array streams (described in the previous chapter).

This chapter explores an alternative to the file I/O model for supporting data management tasks in object-parallel simulations: the persistent data model. A new object-parallel I/O abstraction is proposed, the *persistent collection*. Elements of a persistent collection can contain persistent data in addition to ordinary transitory data.

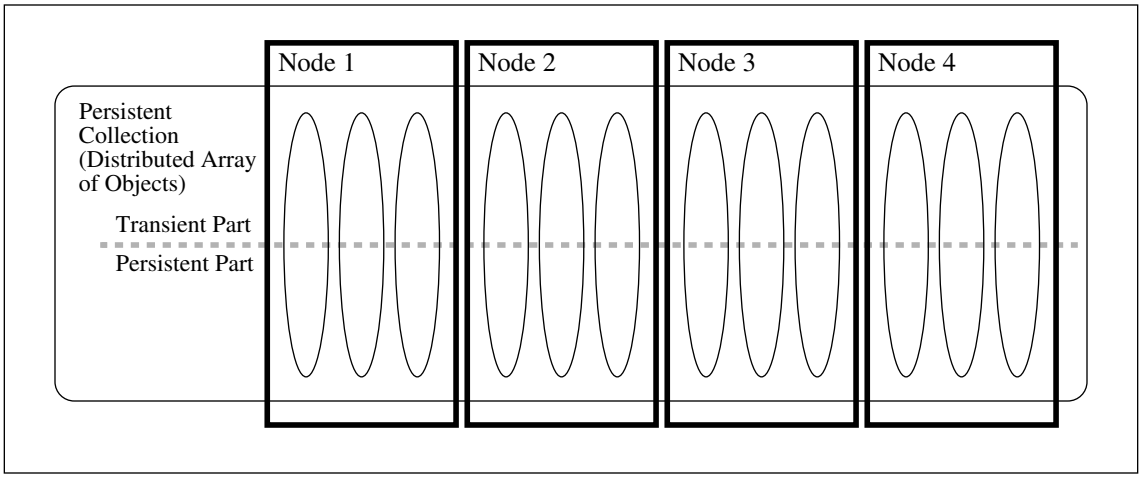


Figure 6.1: Persistent Collection Architecture

A persistent collection is a distributed array of objects, where each object has a persistent part and a transient part.

```
Collection PersistentCollection {
 public:
  // Mechanism for associating this collection
  //   with a collection on stable storage
  static void beginTransaction();
  static void commitTransaction();
  static void abortTransaction();
};

Collection MyCollection: public PersistentCollection {
  ...
};

class PersistentElement {
  // Mechanism for accessing data in the transient part
  // Mechanism for accessing data in the persistent part
};

class MyElement: public PersistentElement {
  // Transient Data
  // Persistent Data
};
```

Figure 6.2: Sketch of the Persistent Collection Interface.

The persistent section of each element is automatically preserved across program executions; no application I/O code is required to save or load this data. A transaction mechanism is supported, allowing programmers to checkpoint persistent data with a single line of code that commits a transaction. In addition, the persistent part of a collection is concurrently accessible by multiple pC++ programs, with no explicit code for communication required. The following sections describe the design and an implementation of the persistent collection abstraction, and discuss an example program using persistent collections for I/O.

6.1 Goals

The goal of the work described in this chapter was to investigate how persistent data systems supporting data management in object-oriented parallel languages can be

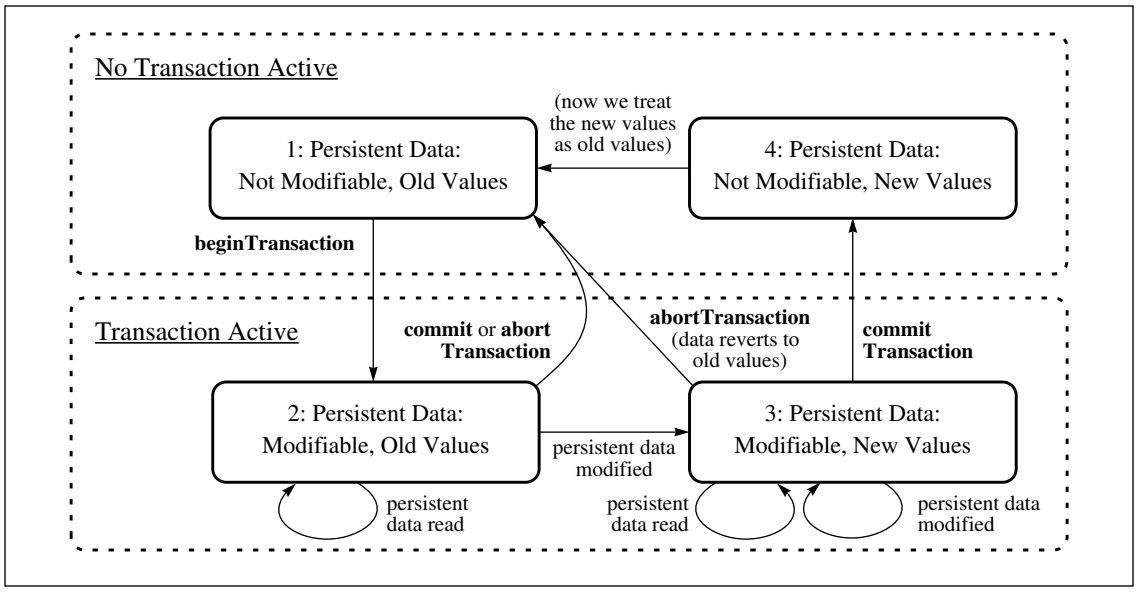


Figure 6.3: Persistent Collection State Diagram

Specifies the dynamic behavior of persistent collections. Several constraints not shown in the diagram are described here. There are (at least) two types of locks on persistent data: shared and exclusive. In state 2, the first attempt to read persistent data causes a shared lock to be acquired on that data before it can be read. Acquisition of the shared lock will succeed immediately as long as no other program has an exclusive lock on the same data (in which case the thread will block until the other program releases the exclusive lock). Similarly, in state 2, an attempt to modify persistent data causes an exclusive lock to be acquired on that data before it can be modified (and before the transition to state 3). Acquisition of the exclusive lock will succeed immediately as long as no other program(s) have exclusive or shared locks on the same data (in which case the thread will block until the other program(s) release the locks). Implementations may include mechanisms for dealing with deadlocks, or may leave deadlock avoidance up to the programmer. A persistent collection must be associated with a collection on stable storage before its persistent data can be accessed or modified.

designed, and to investigate how object-oriented database management systems (*OODBMS*) can be applied in the implementation of these systems.

6.2 Design

Figure 6.1 sketches the architecture of the persistent collection abstraction, which is described further below. A *persistent collection* is a distributed array of objects, where the data within each object is divided into a transient part and a persistent part, and access to the persistent data is regulated by a transaction mechanism. So to use a persistent collection, the programmer needs mechanisms for identifying the transient and persistent parts of data within the element class and mechanisms for beginning, committing, and aborting transactions on the persistent data. Figure 6.2 (which lists methods that implementors of persistent collections may wish to support, using pC++ notation) shows these mechanisms being inherited by the programmer's collection from generic `PersistentCollection` and `PersistentElement` classes, but other implementation techniques may be possible. Figure 6.3 describes the transaction and locking mechanisms underlying the dynamic behavior of persistent collections.

6.3 Implementation

I implemented persistent collections for the pC++ language in the summer of 1995 as a pC++ library built on a beta release (version 0.9.3) of the Shore [7] object-oriented database management system. The next section describes the pC++ interface to this library, and the section following describes details of the implementation's internal architecture.

6.3.1 Interface

This section first describes how programmers can define persistent collection elements, then describes how persistent collections themselves can be instantiated and used.

Elements of persistent collections are defined as follows. The programmer defines the data that is to reside in the persistent part of the elements using the interface definition language SDL (Shore Data Language), in an "interface" called `PersistentElementData`. This definition takes place in a special file `PersistentData.sdl`, which is transformed (by Shore) into a C++ header file `PersistentData.h` that the user includes in pC++ source files that access the persistent data. The following is the SDL definition the programmer would use to obtain a single persistent long integer per element (`myPersistentLong`):

```
module MyElement {

  interface PersistentElementData {
   public:
    attribute long myPersistentLong;
  };

}
```

All SDL interfaces must be defined within an SDL module. Implementors of persistent collections on Shore may wish to use the module name (`MyElement`) to distinguish between `PersistentElementData` definitions for different element classes. (So in the above example `PersistentElementData` would define the persistent part of pC++ collection elements of type `MyElement`.)

The programmer defines both the transient part of a persistent collection's elements and the element's methods in an ordinary pC++ element class (`MyElement` in this example), deriving this class from the class `PersistentElement` (which is defined in `PersistentElement.h`, part of the implementation):

```
#include "PersistentElement.h"

class MyElement : public PersistentElement {
 public:
  long myTransientLong;
  void P_initialize();
  void hello();
};
```

The class PersistentElement contains a data member P through which the element's persistent part can be accessed; data in the transient part is accessed normally:

```
void MyElement::hello() {
  printf("Hello world!\n");
  printf("My persistent data is %ld\n", P->myPersistentLong);
  P.update()->myPersistentLong += 1;
  printf("My transient data is %ld\n", myTransientLong);
}
```

P->myPersistentLong is used for read-only access to myPersistentLong; P.update()->myPersistentLong is used when myPersistentLong is to be updated.

Persistent collection "classes" are defined exactly like ordinary collections, except that they must be derived from PersistentCollection (which is defined in PersistentCollection.h, part of the implementation):

```
#include "PersistentCollection.h"

Collection MyCollection: public PersistentCollection {
  public:
    MyCollection(Distribution *D, Align *A,
                 char *persistentCollectionName);
  MethodOfElement:
    virtual void hello();
};

MyCollection::MyCollection(Distribution *D, Align *A,
                           char *persistentCollectionName)
:PersistentCollection(D, A, persistentCollectionName) {};
```

Note that the constructor MyCollection::MyCollection() takes a string argument persistentCollectionName, which it must pass on to the constructor of PersistentCollection.

The programmer can instantiate a persistent collection of type MyCollection<MyElement> having SIZE elements distributed in BLOCK fashion over the processors in the usual way:

```
void Processor_Main(int argc, char **argv) {
  Processors P;
  Distribution D(SIZE, &P, BLOCK);
  Align A(SIZE,"[ALIGN(V[i], T[i])]");

  MyCollection<MyElement> X(&D, &A,
                            "/myPersistentCollectionName");

  beginTransaction();
  X.hello();
  commitTransaction();
}
```

In this example, the programmer passes the string `"/myPersistentCollectionName"` into the collection's constructor; this string is passed on to `PersistentCollection`, where it identifies a persistent collection on stable storage (in the Shore persistent object namespace) that is to be associated with the instantiated collection. If no collection named `"/myPersistentCollectionName"` exists when the program is run, then one is created. `PersistentCollection` contains a function

```
int isPersistentNew();
```

that returns true if a persistent collection on stable storage is created when the collection is instantiated; the programmer can use this function to perform initialization on the persistent parts of a collection's elements when the persistent parts are first created.

Access to the persistent part of a collection must be made within a transaction (see Figure 6.3 on Page 92 for a detailed explanation). `beginTransaction()` initiates a transaction, `commitTransaction()` ends a transaction normally, and `abortTransaction()` (or a system or program crash) ends a transaction and rolls back all persistent data modified during the transaction to their pre-transaction values. So to checkpoint the persistent part of a collection, the programmer can simply call `commitTransaction()`. In addition, if other programs will access the persistent data concurrently, the points at which `commitTransaction()` are called define the points at which the other programs may gain access to the collection's persistent data.

SelfConsistentField.h:

```
#include "PersistentCollection.h"

Collection SelfConsistentField : public PersistentCollection {
  ...
};
```

scf.h:

```
#include "SelfConsistentField.h"

void Processor_Main(int argc, char **argv)
{
  Processors P;
  Distribution D(num_of_elements, &P, BLOCK);
  Align A(num_of_elements,"[ALIGN(T[i], D[i])]");

  // Declare the persistent collection X and perform initial input
  //   (Persistent data is initialized in the constructor for X)
  //   (If this is a restart, X will automatically contain its old values)
  //   (If this is not a restart, the (programmer-provided) constructor for X will
  //     initialize the collection)
  beginTransaction();
  SelfConsistentField<Segment> X(&D,&A, "/scf");

  // Initialize non-persistent data
  // ...

  // Main computation loop
  while (X.n < nsteps) {
    X.n++;

    // Computation for time step n
    // ...

    if (X.n < nsteps) {
      // Checkpoint all persistent data
      commitTransaction();

      // At this point the data in X is read/write accessible by other programs
      // (e.g. visualization or computational steering programs)

      beginTransaction();
    }
  }

  // Final output of all persistent data
  commitTransaction();
}
```

Figure 6.4: Using Persistent Collections for I/O in an Astrophysics Simulation

Example Programs

To test the persistent collection implementation, I used it to implement checkpointing and inter-program communication for concurrent visualization and computation, in a version of the astrophysics simulation "Self Consistent Field" [35], henceforth referred to as *SCF*. Figure 6.4 sketches how initial and final I/O and checkpointing/restarting is

coded in this version of SCF. The important point to note in this figure is the way in which `commitTransaction()` is used to specify checkpoints.

To test the concurrent data access facilities of my Persistent Collection implementation, I wrote a program *SCFBOB* that acts as an interface between SCF and the volume visualization tool *BOB* (Block Of Bytes) [33]. SCFBOB can be run at any time to dynamically extract the current SCF data set from the SCF program, writing the data set to a specially formatted file, for visualization using BOB (see Figure 6.5).

scfbob.pc

```
#include "SelfConsistentField.h"

void Processor_Main(int argc, char **argv)
{
  Processors P;
  Distribution D(num_of_elements, &P, BLOCK);
  Align A(num_of_elements,"[ALIGN(T[i], D[i])]");

  // Declare the persistent collection;
  //    associate it with a persistent collection "/scf" on stable storage
  //    (Input/communication is automatic)
  // "/scf" may be concurrently accessed by the SCF program, in which case
  //    this program will gain access to the persistent data between SCF transactions
  beginTransaction();
  SelfConsistentField<_ScfbobElement> X(&D,&A, "/scf");

  // Analyze data in the collection X
  //   and write the output file (to be read by BOB for visualization)
  // ...

  commitTransaction();
}
```

Figure 6.6: Using Persistent Collections for I/O in SCFBOB for concurrent visualization and communication.

Figure 6.6 sketches how concurrent data access is coded in SCFBOB. In this figure, at the point at which the persistent collection `X` is first accessed, Shore attempts to acquire a read lock on the collection; if the SCF simulation is in the middle of a transaction (i.e. a time step) at this point (and so has a write lock on `X`), the persistent collection implementation will force SCFBOB to block until SCF commits its transaction (at the end of the time step), at which SCFBOB will gain access to the collection.

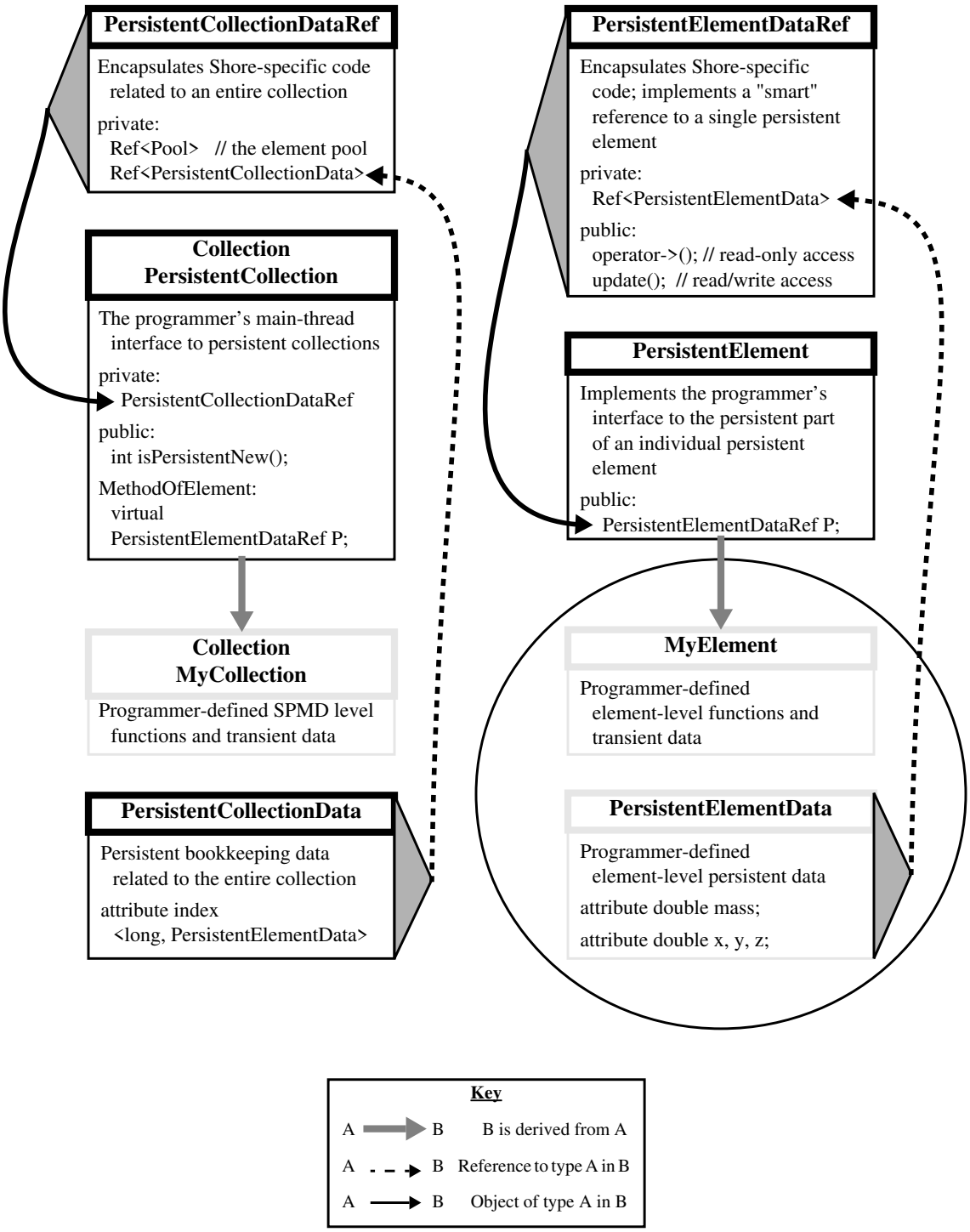


Figure 6.7: Class Structure in a Persistent Collection Implementation

Gray borders denote programmer-defined classes. Elements have both a transient part and a persistent part: the upper and lower classes (respectively) in the circle.

6.3.2 Internal Architecture

Figure 6.7 depicts the pC++ class structure of my implementation of persistent collections. The implementation consists of two more or less separate parts: an "element" part and a "collection" part.

The "collection" part of the implementation (see the left side of Figure 6.7) contains code related to the instantiation of persistent collections. The constructor for `PersistentCollection` implements the mechanism for associating a persistent collection with a named collection on stable storage (i.e. in a Shore database); if the named collection does not exist, then it is created and `isPersistentNew()` will return true (allowing the programmer to perform initialization, etc.) The programmer's collection (`MyCollection` in the figure) is derived from `PersistentCollection`. `PersistentCollection` contains a data member of type `PersistentCollectionDataRef`, a class that encapsulates the Shore-specific code required for these operations.

The "element" part of the implementation (see the right side of Figure 6.7) implements mechanisms for defining and accessing persistent data within individual collection elements. The programmer defines the persistent part of a collection's elements in SDL (Shore Data Language, an interface description language) within `PersistentElementData`; the transient part is defined in pC++ within the element class (`MyElement` in the figure), as usual. The element class is derived from the class `PersistentElement`, from which it inherits `P`, a "smart" reference to the persistent part of the element. Element methods are defined within the element class as usual, but may access data in the persistent part of the element using `P->mass` for read-only access and `P.update()->mass` for read/write access (using `mass` as an example). `P` is of type `PersistentElementDataRef`; this class implements `operator->()` and `update()`, encapsulating (as much as possible) the Shore-specific code and data required for accessing the persistent part of an element.

In addition to the classes described above, the implementation contains three global functions for managing transactions:

```
void beginTransaction();
void commitTransaction();
void abortTransaction();
```

The implementation of each of these functions simply calls the Shore function of the same name.

This implementation maps persistent collections to the Shore data model as follows. The stable storage version of a persistent collection has a name which corresponds to a directory (of the same name) in the Shore object namespace. The persistent parts of the elements of the collection are represented as anonymous Shore objects of type `PersistentElementData` within a pool in the collection's directory. A single registered Shore object of type `PersistentCollectionData`, also in this directory, contains a Shore index that maps element numbers to OID's of the `PersistentElementData` objects (implementing persistent part of the corresponding elements) in the pool.

When a persistent collection is instantiated, the programmer passes a name into the collection's constructor to associate the collection with a stable storage version. This name is passed to the constructor for the `PersistentCollectionDataRef` class, which uses it to go to the directory with the same name in the Shore object namespace, where it finds the element pool and a `PersistentCollectionData` object, and remembers their OIDs for later use. Next the constructor of `PersistentCollection` makes an object-parallel call to an initialization method within the `PersistentElementDataRef` object contained in each element. The initialization method uses the its own element number and the Shore index (referenced within `PersistentCollectionDataRef`) to find (and remember) the OID of the corresponding `PersistentElementData` object within the element pool, which contains the persistent part of the element. Finally, when the programmer invokes

`P->myPersistentData` or `P.update()->myPersistentData` in an element method, `PersistentElementDataRef` invokes the corresponding Shore method (`operator->()` or `update()`) on the Shore reference to the persistent part of the object, Shore obtains an appropriate lock on the data if necessary, Shore brings the persistent part of the object into its local object cache if it is not present, and Shore returns the requested data to the programmer.

6.3.3 Lessons Learned

This section discusses issues that should be considered by those planning to develop libraries similar to persistent collections, and those planning to use such libraries for data management in scientific/engineering applications:

- **Language / OODBMS type incompatibility issues:** The type representation used by a persistent collection implementation's OODBMS must mesh well with the type representation used by the target languages; otherwise problems will result. By far the most significant problem I encountered was an incompatibility between the Shore and FORTRAN representations of arrays. My version of the SCF code is a combination of pC++ and FORTRAN, with arrays stored in FORTRAN format. The persistent arrays supported by the beta version of Shore are an opaque type that appears to be some type of linked-list structure, fundamentally incompatible with the FORTRAN representation. If I had used Shore arrays for persistent storage, I would have had to copy the entire data set back and forth between the two representations at every SCF main loop iteration, causing decreased performance and an increased line count. I solved this problem by using Shore strings to store the FORTRAN arrays; this hack forced an awkward syntax to be required for array access.

- **Some file I/O cannot be avoided:** Most existing post-processing tools (including visualization tools such as BOB) expect files as input, so libraries such as persistent collections cannot be used directly for communication with these programs; an intermediary program (such as SCFBOB) is required to generate a file from the persistent data. However, my experience has been that these intermediary programs are fairly simple and modular.

- **Restarts must be handled with care:** Designing the code for handling restarts was a difficult part of porting SCF to the use of persistent collections for I/O. I found that it was not sufficient to support persistence for collections only; SCF stored a significant amount of state in non-collection data structures, so I was forced to either use files to store this information or to support independent persistent objects as well as persistent collections. I chose to support persistent objects, which turned out to be fairly straightforward using Shore.

6.4 Summary

This chapter discussed the design and implementation of the persistent collection abstraction, which supports I/O in object-parallel programs using the the persistent data model. The next chapter compares the file I/O and persistent data models, based on experience gained during the design and implementation of the I/O systems discussed in the last four chapters.

Chapter 7: Analysis and Comparison

The file I/O model has been the conventional data management model for parallel scientific and engineering simulations. Chapters 2 through 5 discussed four abstractions for file I/O in these programs: parallel file systems, parallel filestreams, object-parallel filestreams, and distributed array streams. Chapter 6 introduced an alternate model for data management in these programs, the persistent data model, and explored the use of object-oriented database management systems in the implementation of persistent data systems. This chapter compares the suitability of file I/O abstractions implemented on parallel file systems to the suitability of persistent data systems implemented on OODBMS for the support of I/O in object-oriented parallel scientific and engineering simulations. The comparison is based on experience gained during the work described in the previous chapters. Two representative implementations are compared: distributed arrray streams (Chapter 5) and persistent collections (Chapter 6). Several metrics are used for comparison: performance, storage requirements, productivity effects, ease of implementation, maintainability and portability.

7.1 Performance

The strategy taken in this section is to use file system I/O performance as a baseline for comparison of the I/O performance of d/streams and persistent collections. Section 7.1.1 compares d/streams I/O performance to the performance of various underlying file systems, Section 7.1.2 compares file system I/O performance to the performance of

persistent collections implemented on an OODBMS, and Section 7.1.3 summarizes the results of these comparisons.

7.1.1 Distributed Array Streams

D/streams make parallel I/O coding easier for application developers by automatically managing the bookkeeping required for keeping track of the amount of data written from each element during output and the amount to be read into each element during input. It is important for programmers to be aware of the performance overhead required for this bookkeeping, so that if this overhead is unacceptable, more highly optimized I/O mechanisms can be employed.

To measure this overhead, a simple d/streams benchmark was developed that consists of the file I/O skeleton from the original version of the pC++ computational cosmology application described in Chapter 6, the *Self Consistent Field* code (*SCF*) [35] [23]. SCF is an N-body code in which the primary data structure is a one-dimensional collection of `Segments` where each segment stores data corresponding to several `particles`. There could be several segments on a given processor. Per-particle information includes the x, y, and z coordinates of the particles, their x, y, and z velocities, and their masses.

In the SCF code particle data is periodically written to a file for later analysis (visualization and debugging). The SCF code is mostly an "output only" application, but the benchmark performs both input and output on the particle data. The time required for I/O on the header data (i.e. element size data) and the actual data was measured within the `flush` and `unsortedLoad` functions. This allowed the overhead of keeping track of the size information to be computed for various data set sizes on various machines. The performance of the d/stream implementation, as a percentage of the performance that could have been attained using manual buffering and storing no size information in the file, is summarized in Figure 7.1. The results show that the overhead

Figure 7.1: Performance of a D/streams Implementation

All timings are in seconds; the measurements are for an output operation followed by an input operation on a distributed data structure. The total number of elements in the data structure (i.e., the number of *segments*) is varied to determine the performance for various data set sizes. The d/streams unsortedRead primitive is used for input. The final row in every table gives d/streams' performance as a percentage of the performance that could have been obtained using manually buffered I/O (indicating the overhead of d/streams' automatic bookkeeping of element size information). These results indicate that d/streams' performance is competitive with manually buffered I/O for large-sized data sets. As expected, the buffered I/O supported by d/streams outperforms unbuffered I/O.

Table 1: Benchmark Results on Intel Paragon (4 processors) (in seconds)

I/O Size (# of Segments)	1.4 MB (256)	2.8 MB (512)	5.6MB (1000)	11.2MB (2000)
Unbuffered I/O	7.13 sec.	14.73	283.00	556.78
Manual Buffering	2.14	3.04	5.42	54.17
d/streams	2.47	3.31	5.71	55.00
% of Manual Buf.	86.7%	91.9%	95.0%	98.5%

Table 2: Benchmark Results on Intel Paragon (8 processors)

I/O Size (# of Segments)	1.4 MB (256)	2.8 MB (512)	5.6 MB (1000)	11.2 MB (2000)
Unbuffered I/O	7.53 sec.	14.47	273.77	561.72
Manual Buffering	2.91	3.75	5.72	9.69
d/streams	3.36	4.20	6.16	10.19
% of Manual Buf.	86.5%	89.3%	93%	95.1%

Table 3: Benchmark Results on Uniprocessor SGI Challenge (preliminary)

I/O Size (# of Segments)	5.6 MB (1000)	11.2 MB (2000)	112MB (20000)
Unbuffered I/O	1.68 sec.	3.42	32.20
Manual Buffering	1.05	2.13	20.9
d/streams	1.32	2.71	21.84
% of Manual Buf.	79%	78%	95%

Table 4: Benchmark Results on Multiprocessor SGI Challenge (8 processors) (preliminary)

I/O Size (# of Segments)	5.6 MB (1000)	11.2 MB (2000)	44.8 MB (8000)
Unbuffered I/O	0.55 sec.	1.10	4.95
Manual Buffering	0.22	0.34	2.38
d/streams	0.39	0.75	2.65
% of Manual Buf.	56%	45%	89%

introduced by the library decreases as the I/O size increases — so the performance of the d/streams implementation scales well with data set size. Also listed is the time required for unbuffered I/O on the SCF data set. Application developers sometimes use unbuffered I/O to avoid the extra code required for buffering (parallel file systems

generally do not support buffering), and this can lead to less than optimal I/O performance, since not using buffering can increase total I/O latency time.

7.1.2 Persistent Collections

Methodology

To test the performance of my persistent collection implementation in the context of an actual application, and to compare the performance of persistent collection/OODBMS I/O with that of file I/O, performance was compared in three versions of the SCF program: the persistent collection version, the original (file I/O) version, and a third version in which all code for checkpointing and final output was removed. All three versions of the code were run on the same input data, for 5 iterations, for several data set sizes. Then the I/O time for persistent I/O and file I/O was computed by subtracting the wall clock run time of the persistent I/O and file I/O versions from the run time of the version without I/O.

Two sets of data were obtained for the persistent collection version: one set with Shore's logging mechanism turned on (the default state), and one set with logging turned off. *Logging* is a technique for supporting database failure recovery, in which all changes to a database that occur during a transaction are recorded in a sequential *log file*, in addition to being recoded in an ordinary permanent copy of the database on disk.

The performance tests were performed on a Sun SPARCstation-5 with an 85 MHz microSPARC II CPU and 32 MB of main memory. Disk storage for the file I/O and persistent versions was provided by a large remote disk through NFS. The test program was the only active user process on the machine while the tests were performed; however, there were other users connected to the network through which the NFS disk I/O was performed. To smooth any fluctuations in the results caused by these other users, each measurement was taken multiple times, and the average of these

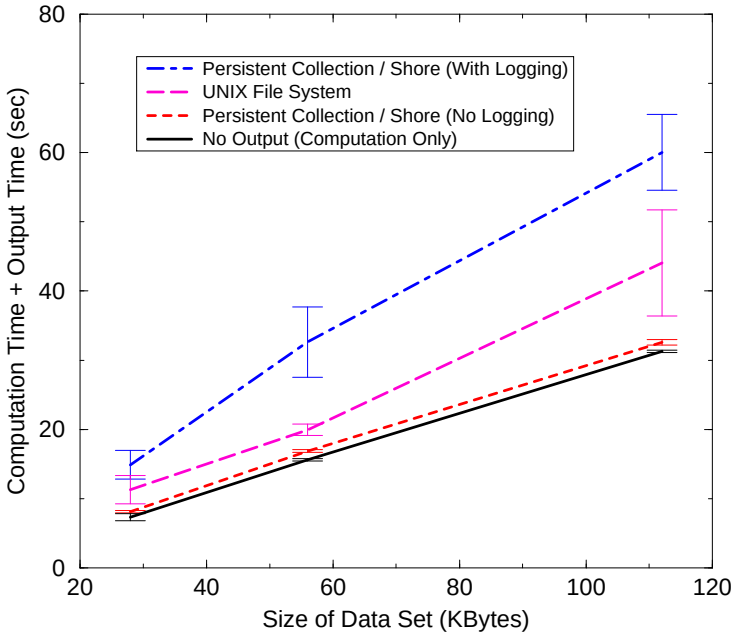


Figure 7.2: Raw Performance Data

measurements was computed after discarding values falling outside two standard deviations from the mean. In the case of the persistent collection version, the Shore server was run on the same machine as the test program. Shore appears to have many parameters that may affect performance; Shore's default parameters were used during all the performance tests.

Performance Results

Figure 7.2 shows the raw performance data that was obtained. Total wall clock run time is graphed for various data set sizes for each of the four versions of the SCF program that were tested. Subtracting the performance graph of the no-output version of SCF from the graphs of the other three versions, dividing by the data set size, and dividing by 6

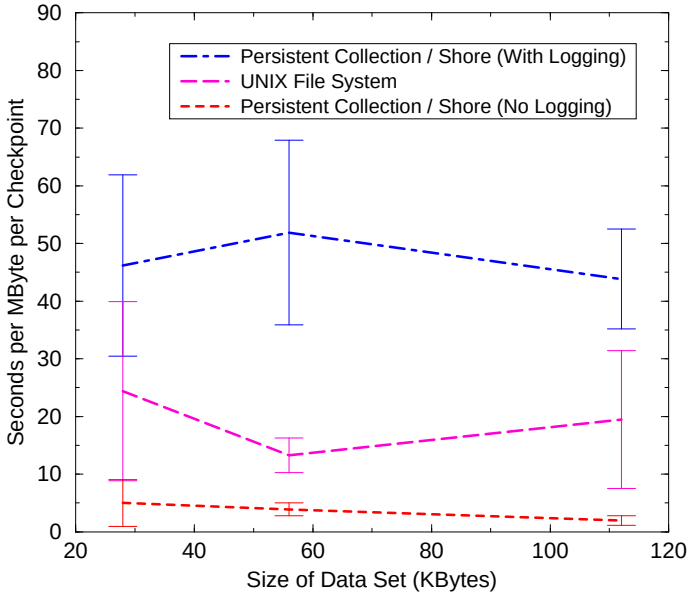


Figure 7.3: Checkpoint Time per MByte

checkpoints (the number performed in each of the runs) gives Figure 7.3, which shows the checkpoint time per MByte of data in the data set. From this graph, it appears that time per MByte for checkpointing remains constant or gradually decreases for all three I/O mechanisms. There definitely seems to be a falling trend in the time cost per MByte for Persistent Collections implemented on Shore (*PC/S*) without logging, which should result in an increasing relative cost for the other two mechanisms as data set size increases. To highlight this effect, each graph in Figure 7.3 was divided by the

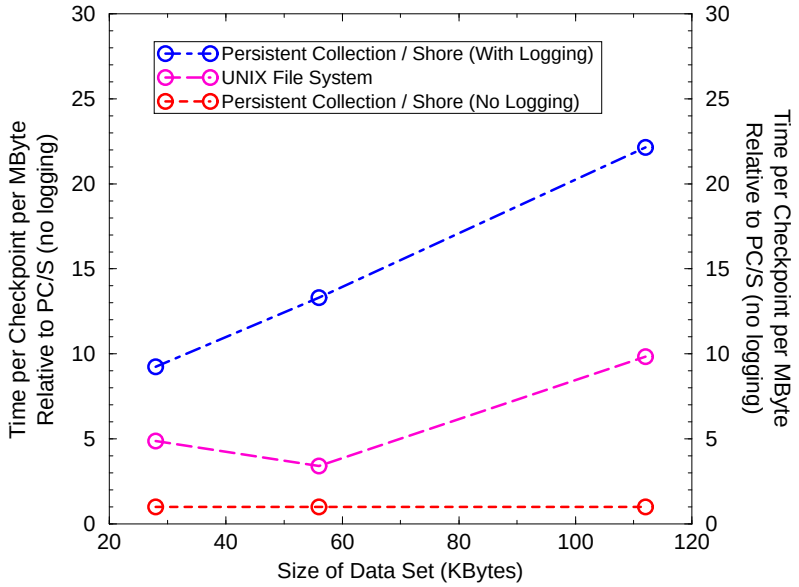


Figure 7.4: Relative Checkpoint Time per MByte

PC/S (no logging) graph, giving Figure 7.4, which shows the checkpoint time per MByte for each version, relative to the PC/S (no logging) version. From this figure it is clear that for the range of data set sizes that were examined, the cost per MByte for file I/O checkpointing and for PC/S checkpointing is significantly greater than the cost for PC/S (no logging) checkpointing, and in fact these relative differences in cost increase significantly as data set sizes increase.

It is likely that the low overhead time measured for PC/S (no logging) is caused by the fact that the entire data set fits in Shore's page cache for these runs; it is likely that no disk I/O takes place in this version, only memory copying, since the Shore process and the application process run on the same machine. With logging turned on, Shore is forced to do disk I/O, since failure recovery mechanisms based on logging require that modifications to the database be periodically written to stable storage.

The discussion so far has focused on checkpointing performance; a performance issue involving inter-application communication should also be noted. Applications using persistent collections with a transaction structures similar to that of SCF, where the

entire database is modified during each transaction, require an exclusive (write) lock on the entire database during each transaction. This means that concurrent access to the database is effectively impossible for simulations such as SCF and visualization programs such as SCFBOB. Visualization programs, which require read-only access to the entire database, must wait until the simulation program checkpoints (and commits a transaction) before they can read-lock the database. Then the simulation program must wait until the visualization program releases its read lock before it can gain the exclusive lock it requires for its next transaction.

Analysis

It appears that Shore's recovery mechanism (logging) is quite time-costly for this application. Turning on Shore's logging feature worsened the performance of Persistent Collection I/O by a factor that ranged from 9 to 22 compared to Shore with no logging. Furthermore, persistent collection I/O with logging turned on was more than twice as time-costly as file I/O.

The version of Shore that was used for these experiments is a beta release; this may have been factor contributing to the relatively poor I/O performance of our application under Shore with logging. But in general, it is questionable whether logging is an appropriate mechanism for failure recovery in scientific and engineering simulation programs.

Logging involves the creation and storage of a sequential list that records each of the modifications to the databases managed by a DBMS. This is an appropriate and efficient recovery mechanism when each transaction modifies only a small fraction of an application's database. But when transactions are used for checkpointing, as in SCF, the application modifies the entire database during each transaction, so the log is guaranteed to grow by an amount at least as large as the database itself during each transaction. In addition, most logging mechanisms record not only the new value of each modified data

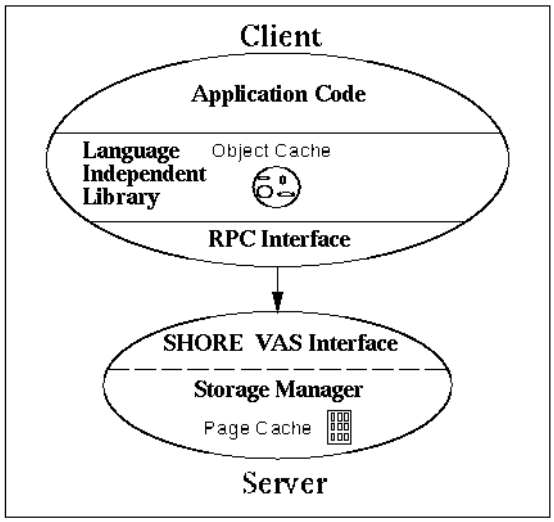


Figure 7.5: Shore Architecture

item in each log entry, but also the old value and additional ancillary information, so during each transaction the log will grow by an amount equal to several times the size of the entire database; all of this log information needs to be transferred to disk following each transaction. One simple optimization that could be performed would be to replace the entire logging procedure with a single backup copy of the entire database written before the transactions in which the entire database is to be modified.

Shore makes no guarantees about preservation of data integrity when failures occur with logging turned off. However, it is interesting to note that in the experiments with PC/S (no logging) I was repeatedly able to successfully restart SCF after killing both the SCF process and the Shore process in mid-transaction. It appears that PC/S (no logging) will restart successfully after process failures caused by most applications errors (segmentation faults, etc), provided that all Persistent Collection data fits within Shore's client-side object cache (refer to Figure 7.5). However, PC/S (no logging) probably will not restart successfully after operating system failures, disk device failures, or process failures caused by errors in Shore itself.

7.1.3 Performance Summary

To summarize:

- Distributed array streams have the same performance characteristics as parallel file systems, with the addition of some overhead for keeping track of element size and distribution information.
- Persistent Collections implemented on the Shore OODBMS can outperform file system I/O when logging is turned off and when the data set fits in the OODBMS' page cache. However when logging is turned on, file I/O outperforms OODBMS I/O by a factor of two.

With logging off, persistent collections can be used for inter-application communication and for protection from process failures caused by errors in the simulation program, but not for protection from other types of failures. Logging does not seem to be an appropriate mechanism for failure protection in scientific/engineering simulation programs such as SCF that modify the entire database during each transaction.

7.2 Storage Requirements

Persistent Collections implemented on Shore has a memory overhead of about 12 MB (which is approximately 120 times the memory overhead of d/streams), and a disk space overhead of about 5 MB (d/streams has no disk space overhead). More details are described below.

7.2.1 Distributed Array Streams

The size of the stripped d/streams library is approximately 100 kB. D/streams' data segment usage depends on the amount of I/O buffering the programmer requires, and

likewise disk space usage depends on the amount of file storage the programmer requires. Typically the programmer will need to store files containing an input data set and one or two checkpoint data sets.

7.2.2 Persistent Collections

The size of the stripped SCF/file executable is .2 MB. The stripped SCF/Shore executable is 1.6 MB, so subtracting the size of SCF itself, Shore's text segment consumes 1.4 MB on the client side. Stripped versions of the Shore server's two executables consume a total of 2.9 MB. Under default parameters, Shore's data and stack segments consume 3.7 MB on the server side and 4.0 MB on the client side (during a 28 kB data set SCF run). So assuming one Shore server per node (there could be fewer), the total memory usage of PC/Shore is 4.3 MB of (fixed) text segment usage and 7.7 MB of (possibly variable) data and stack segment usage, when managing a 28 kB data set.

As for disk space requirements, a Shore server stores data on disk in two places: a primary volume and a log volume (see the discussion of logging on Page 112). Using Shore's default parameters during a run of SCF/Shore with a 28 kB data set, Shore's primary volume initially consumes 5.0 MB of disk storage and grows by 1 MB, and the log volume initially consumes 13 kB and grows by 618 kB.

7.3 Productivity Effects

This section predicts the effects that the use of d/streams and persistent collections would have on application programmer productivity, based on the experience gained in using these systems to support I/O in the example programs described in earlier chapters.

7.3.1 Distributed Array Streams

The use of d/streams (as opposed to parallel file systems) for parallel I/O should have at least as much of a positive effect on productivity as the use of ordinary filestreams (as opposed to ordinary file systems) for ordinary C++ I/O. Both stream abstractions support buffering and flattening/unflattening of complex objects, allowing the code supporting these functionalities to be factored out of applications (increasing productivity). In addition, d/streams supports the bookkeeping required for keeping track of the size and distribution of element objects, allowing still more code to be factored out of applications.

7.3.2 Persistent Collections

The use of persistent collections and a persistent data model (as opposed to parallel file systems and a file I/O model) should have both positive and negative effects on productivity. These effects are explained below, based on the experience gained using persistent collections for data management in the SCF program.

Positive Effects

- **Same line count, increased functionality:** The original SCF program, using file I/O, has 2100 lines of code (including lines in makefiles), 16% of which were I/O related. The persistent collection version has the same number of lines as the original (line count differs by one), and supports automatic restart (unlike the original version). There were actually 60 fewer lines of executable code in the persistent collection version, but this decrease was offset by an increase of 60 lines in the makefile.

Negative Effects

- **Larger makefiles:** The makefile in the persistent collection version of SCF was almost twice as large as the original makefile.
- **Programmers must learn unfamiliar concepts:** The use of transactions, interface definition languages, and database management systems may be unfamiliar to developers of scientific/engineering simulations; learning these new concepts may cause an initial loss in productivity.

7.4 Ease of Implementation

The design and implementation of d/streams was a fairly challenging task. One of the most difficult parts of the implementation involved determining how to automatically manage the complex mapping from the output collection through the parallel file system and back into an input collection, an input collection in which the element distribution or number of processors may have changed since output. However, the strategy described in Section 5.3.2 solves this problem; this should make implementation of d/streams more straightforward in the future. Design and implementation of the d/streams library described in Chapter 5 took approximately 600 programmer hours; the implementation consists of 1550 lines of code, 400 of which support portability across various parallel machines and file systems.

Implementing persistent collections on Shore was fairly straightforward; Shore handles all of the work required for transactions, recovery, and concurrent data access, so the implementation consisted mostly of "glue" code to associate pC++ collections with Shore objects. The most challenging part of the design was the mapping from pC++ collections to Shore's persistent data structures. The design and implementation of Persistent Collections and SCFBOB, and the translation of SCF/file to SCF/Shore,

took a total of approximately 350 programmer hours. The Persistent Collection implementation itself consists of approximately 600 lines of code (not including Shore).

7.5 Maintainability

The implementations of d/streams and persistent collections described in Chapters 5 and 6 are similar, in that both are libraries built on underlying data management systems. D/streams is built on parallel file systems, while persistent collections is built on an object-oriented database system. Since file systems are generally components of operating systems, updates of file systems and operating systems are generally released simultaneously. On the other hand DBMS are large, complex systems that usually are not written or maintained by operating system vendors, so opportunity exists for DBMS to get "out of sync" with the underlying OS, as new versions of the OS are installed. So system maintenance will generally be more complex when OODBMS are used for I/O (as in persistent collections) than when file systems are used (as in d/streams).

7.6 Portability

The portability of systems such as d/streams (implemented on an underlying file system) will be significantly affected by the availability of similar file systems on the target platforms. Similarly, the portability of systems such as persistent collections (implemented on an underlying OODBMS) depends on the availability of similar OODBMS on the target platforms. At this point, there has been slightly more standardization in parallel file systems than there has been in OODBMS, so data management systems built on parallel file systems should tend to be slightly more portable than those built on OODBMS.

7.7 Limitations of this Study

The parallel filestreams implementation described in this chapter only runs on uniprocessors. Originally a parallel version was planned, to be based on a parallel implementation of the Shore OODBMS being developed at the University of Wisconsin, but the parallel version of Shore is not available at the time of this writing. Still, the uniprocessor implementation of persistent collections described here gave some significant insights into the suitability of current OODBMS for the support of a persistent data I/O model in object-parallel simulations.

	d/streams	Persistent Collections
Performance	Faster	Slower (with logging)
Space Requirements Text (per node; Data 28kB data set) Disk	~100 kB (= buffer size) ~30 kB	4.3 MB 7.7 MB 6 MB
App. Prog. Productivity	Positive Effects	Mixed Effects
Ease of Design & Implementation	~600 Hours	~350 Hours
Maintainability & Portability	Depend on OS	Depend on OODBMS
Limitations of this Study		Uniprocessor Implementation Only
Recommendations	Generally preferable	May be useful for apps. requiring OODBMS Explore Alt. Recovery

Figure 7.6: Summary of Analysis

7.8 Summary

Figure 7.6 summarizes the results of the analysis described in this chapter. For the object-oriented parallel simulation applications considered in this dissertation, abstractions such as d/streams (supporting a file I/O model of data management and implemented on parallel file systems) should generally perform better, require less storage, have a better impact on productivity, and be more portable and more easily maintainable than abstractions such as persistent collections (supporting a persistent data model and implemented on object-oriented database systems).

However, there are some conditions under which abstractions like persistent collections are preferable to d/streams. Persistent collections are preferable for supporting checkpoint/restart tasks:

- when the OODBMS' logging mechanism can be turned off, and
- when there is enough physical memory to support both the application's data and the OODBMS in memory (so that the OODBMS can use simple memory copying to support checkpointing), and
- when the expected types of failures are application process failures (i.e. segmentation faults), rather than hardware or media failures.

Persistent collections are preferable for supporting concurrent visualization and computation and computational steering when visualization or steering is to occur only infrequently; the simulation must pause each time the visualization/steering application accesses the simulation's data, but this is preferable the possible file I/O strategy of forcing the simulation to pause at every time step to write an output file (for visualization) or to read a parameter file (for steering). Abstractions like persistent collections, built on OODBMS, should also be preferable for supporting data management in applications that require periodic access to subsets of a large database via query languages or other techniques. This is precisely the type of application that most DBMS are designed to support.

Chapter 8: Conclusion

This chapter describes this dissertation's contributions, identifies several directions for future research, and lists a few final thoughts.

8.1 Contributions

This dissertation makes several contributions to the understanding of how data management can be supported in object-oriented data-parallel simulations. This section restates and expands the description of these contributions first presented in Section 1.4.

Chapter 2 introduced parallel file systems (*PFS*), defined several measures of PFS performance, and related PFS performance to application I/O patterns and to various elements of PFS internal architecture. Two important measures of parallel file system performance were identified: the application I/O rate and the actual I/O rate. Three successively more detailed models of parallel file systems were developed, and the I/O performance of a simple I/O intensive application was described for each model. These models were used to explain the effects of application I/O patterns, buffering, and I/O system virtual memory on I/O performance. Then the performance patterns predicted by these models were related to the results of experiments performed on two commercial parallel I/O systems (on the Intel Paragon and the Thinking Machines CM-5). Several I/O optimizations identified during the course of this experimentation were described.

These models and results should be useful for optimizing I/O performance in applications and I/O libraries, configuring existing PFS and designing new ones.

Parallel file systems effectively support parallel I/O in parallel programs that have relatively simple distributed data structures, such as distributed arrays of simple floating point numbers. However, *object-oriented* parallel programs have special data management requirements that are not met by parallel file systems. Three new abstractions for data management within the file I/O model were proposed in Chapters 3 through 5 to meet these special requirements. These abstractions can be implemented on parallel file systems using language-specific class libraries; implementations of each were described. Chapter 3 introduced *parallel filestreams*, an extension of ordinary C++ filestreams supporting node-buffered parallel I/O with flattening and unflattening in SPMD parallel C++ programs. *Object-parallel filestreams*, introduced in Chapter 4, extend parallel filestreams to support the element buffering required for file-format-independent I/O on distributed arrays of objects in languages such as pC++. *Distributed array streams*, described in Chapter 5, extend object-parallel filestreams to support a simplified programmer interface as well as automatic bookkeeping of array distribution meta-data within special-format files.

In the file I/O model, program variables are transitory, i.e., their data disappears when the program terminates. In order to preserve transitory data, the programmer must write code to output that data to a file before the program terminates, using either an operating system supported file I/O mechanism or a higher-level abstraction such as distributed-array streams. Chapter 6 explored an alternative to the file I/O model for supporting data management tasks in object-parallel simulations: the persistent data model. A new object-parallel I/O abstraction was proposed, the *persistent collection*. Elements of a persistent collection can contain persistent data in addition to ordinary transitory data. The persistent section of each element is automatically preserved across program executions; no application I/O code is required to save or load this data. A

transaction mechanism is supported, allowing programmers to checkpoint persistent data with a single line of code that commits a transaction. In addition, the persistent part of a collection is concurrently accessible by multiple pC++ programs, with no explicit code for communication required.

Chapter 7 compared how well file I/O systems implemented on parallel file systems and persistent data systems implemented on OODBMS support I/O in object-oriented parallel simulations. The comparison is based on experience gained in the implementation of the abstractions described in Chapters 3 through 6, and involves a number of metrics: performance, space requirements, productivity effects, ease of implementation, maintainability, and portability. It was found that for the object-oriented data-parallel simulations considered in this dissertation, abstractions such as distributed array streams (supporting a file I/O model of data management and implemented on parallel file systems) should generally perform better, require less storage, have a better impact on productivity, and be more portable and more easily maintainable than abstractions such as persistent collections (supporting a persistent data model and implemented on object-oriented database systems). The comparatively slow performance of the persistent collections implementation appears to be due to the log-based recovery mechanism employed by the underlying OODBMS. It was found that logging may not be an appropriate mechanism for failure protection in scientific/engineering simulation programs, since these programs tend to modify an entire database during each transaction. However, there are some special circumstances in which abstractions like persistent collections can outperform file I/O systems; these are described in Section 7.8.

8.2 Directions for Future Research

Several approaches to the support of data management in object-oriented data-parallel scientific and engineering simulations have been identified in this dissertation; this section discusses several directions in which this research could be extended.

8.2.1 File I/O Model

Perhaps the most difficult task in supporting the file I/O model in parallel programming systems is the provision of mechanisms that allow programmers to specify the mapping of data between

- multiple I/O buffers on multiple processors, and
- a single file consisting of a one-dimensional string of bytes.

Parallel file systems generally support a small number of hard-coded mappings. Parallel filestream and object-parallel filestream abstractions give the programmer some degree of control over the mapping by allowing the programmer to specify the order and timing with which buffer flushing and loading is performed. Distributed array streams avoid the issue entirely by forcing the programmer to use a fixed mapping, allowing the interface to the d/stream abstraction to be simplified considerably. But in general it is important for there to be mechanisms available with which the programmer can specify this mapping, since special-format files are a widely-used mechanism for communication of information between programs. Therefore continued research into simplified, more general, and more expressive mechanisms for specifying the processor/file mapping should be beneficial.

8.2.2 Persistent Data Model

One of the most significant strengths of database management systems has been their support for access to subsets of large databases. It appears that the major reason for the relatively poor performance of the OODBMS supporting the persistent collection implementation (described in Chapters 6 and 7) is the fact that the application with which this system was tested accesses the entire database within each transaction; DBMS generally have not been designed to support this type of access pattern. Two directions for future research in persistent data systems are the following:

- Data management systems based on current-generation OODBMS should perform much better and should have an improved impact on productivity (relative to systems based on the file I/O model) when they are designed for and tested on applications that require access to a subset of a large database (rather than accessing the entire database) in each transaction. There do exist parallel scientific and engineering applications (e.g. some biology and genetics applications) that require access to subsets of a large database; developers of these applications could benefit from research into data management systems that integrate support for distributed arrays of persistent objects with support for declarative query language access to large databases.

- Developers of more conventional parallel simulations could benefit from research into alternative OODBMS recovery mechanisms (i.e. alternatives to logging) designed to support high performance failure protection in applications that access an entire database in each transaction. Such recovery mechanisms could make possible persistent data systems having performance comparable to or better than that of file I/O systems (for this type of application).

8.3 Conclusion

The development of systems supporting data management in parallel programs is a task made especially complex by the wide range of parallel programming models currently in use. These models collectively employ a diverse set of both ordinary and distributed data structures, and each model is associated with a certain set of hardware architectures. Data management systems tend to be highly data-structure- and hardware-architecture-specific, since they are responsible for the movement of data along the entire path from transient data structures in memory, through the hardware architecture to stable storage devices, and back again. Therefore it seems that the best approach to the design of systems supporting data management in parallel programs is to consider the requirements of each programming model separately, and to design specialized systems for data management in each model.

Hopefully this dissertation makes a small contribution to the understanding of how data management in parallel programming systems can be supported, by focusing on the particular case of data management in object-oriented, data-parallel programming systems. Much work remains to be done in the design of data management systems that will complement parallel programming systems in which other models of parallelism are employed.

Appendix A: Introduction to Parallelism

A.1 Parallel Hardware

Parallelism is a means of reducing the amount of time needed to solve a given problem: the problem is divided into sub-problems that are solved simultaneously by multiple hardware devices (see Figure A.1). A great variety of parallel hardware architectures have been designed over the years. Single Instruction Multiple Data (SIMD)

Figure A.1: Parallelism

A problem is divided into sub-problems that are solved simultaneously by multiple hardware devices.

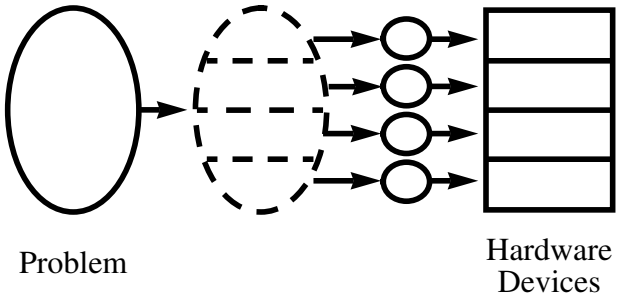
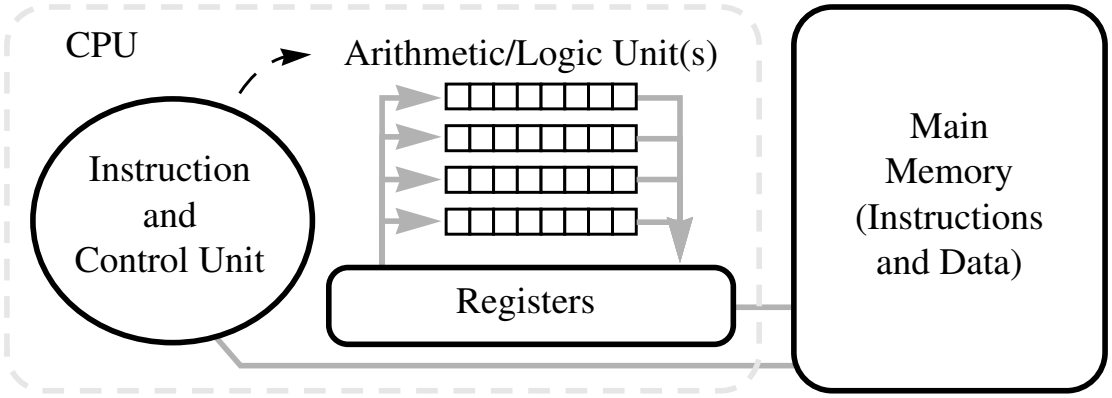


Figure A.2: Pipelined SIMD Architecture

Maximum possible speedup for this architecture is the number of ALU's times the number of pipeline stages.



architectures (see Figures A.2 and A.3) have one instruction unit and multiple devices for doing arithmetic. These architectures work best on regular problems having very simple data structures, such as some types of simulations. The assembly language programming model for these machines has one thread of control, with special instructions for parallel arithmetic on regular data structures. In Multiple Instruction

Figure A.3: Parallel SIMD Architecture

The maximum possible speedup for these machines is the number of ALU's. The ILLIAC IV, the first supercomputer, designed in 1967 by NASA, had a parallel SIMD architecture.

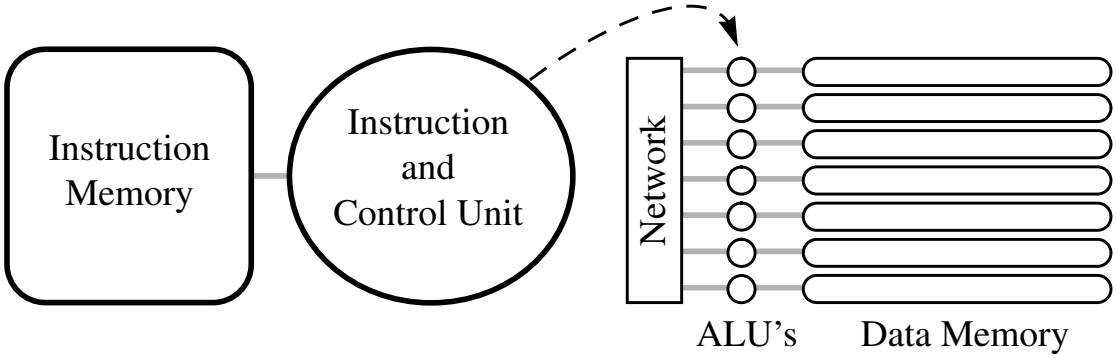
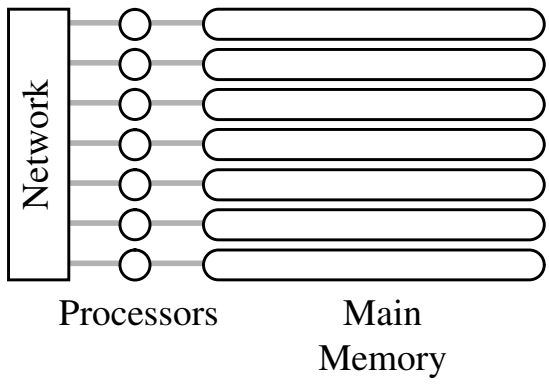


Figure A.4: Distributed Memory MIMD Architecture

Multiple memories, one per processor. The assembly language programming model for these machines is multiple programs, one per processor, communicating by message passing through the network. Often these are multiple copies of the same program, in SPMD (single program multiple data) programs.



Multiple Data (MIMD) architectures (see Figures A.4 and A.5) the maximum possible speedup is the number of processors.

A.2 Parallel Programming Systems

A.2.1 Virtual Machines

The hardware of a machine presents the operating system programmer with a model of parallelism which must be adhered to in writing the operating system. The operating system programmer has the opportunity to define a new virtual machine on top of the

Figure A.5: Shared Memory MIMD Architecture

One memory "shared" by all the processors. Assembly language programming model: multiple programs that communicate through shared variables, or a single program with multiple threads of control.

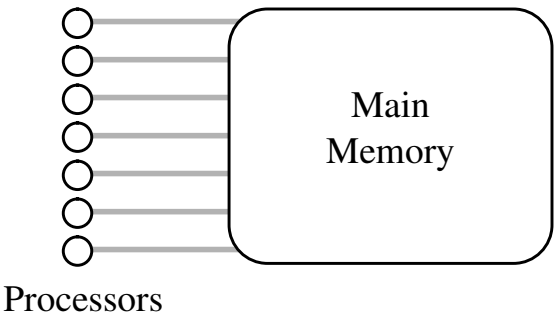
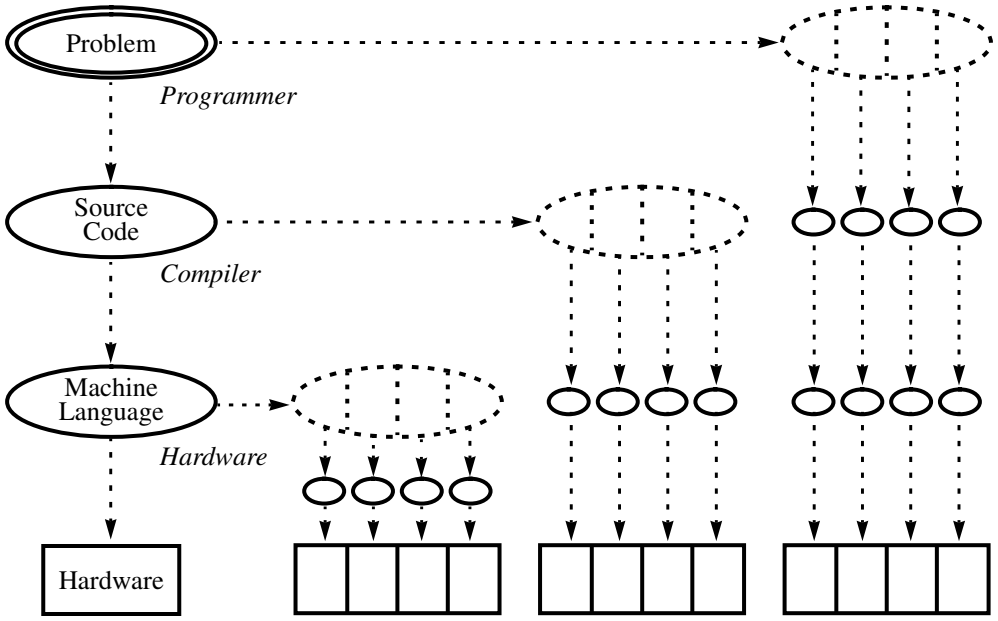


Figure A.6: Parallelism Extracted

by the programmer, the compiler, and the hardware.



hardware. Similarly, programming language designers have an opportunity to implement virtual machines of their own on top of the operating system through the language's compiler and runtime system. If a programming system provides a single virtual machine model across a variety of parallel architectures, then it can be considered to be a type of middleware that allows programs to be easily ported between different machines.

A.2.2 Levels At Which Parallelism Can Be Extracted

A *parallel programming system* is a hardware and software system supporting the development and execution of parallel applications. A great variety of parallel systems have been proposed and developed over the years. An important issue in parallel system design is the question "who is responsible for identifying sub-problems that can be solved in parallel?" (see Figure A.6). For example:

- Very fine-grained parallelism can be dynamically extracted at runtime from a conventional machine language program by a pipelined instruction unit
- Parallelism can be extracted from conventional serial source code by a parallelizing compiler.
- Parallelism can be extracted directly from the problem domain and expressed explicitly in parallel source code by the programmer, using an *explicitly parallel* programming system.

The domain from which parallelism can be extracted for a given problem is the largest in the latter approach, so this approach has the potential to yield the fastest solutions. This approach to parallelism is one discussed in this thesis.

A.2.3 Data-Parallelism

Explicitly parallel programming systems differ in the types of sub-problems for which parallel solutions can be expressed. *Data-parallel* programming systems give the programmer a single thread of control from which identical operations can be applied in parallel to different regions of large data-structures. Mechanisms are often provided for the programmer to explicitly specify how the elements of related data-structures are to be *aligned* with each other (to reduce inter-processor communication during parallel computation involving related data-structures) and how the elements of those data-structures are to be *distributed* to processing elements (to insure a good load balance over the processors during parallel operations). Programming systems supporting data parallelism differ in the nature of the data-parallel operations that can be expressed, the nature of the data-structures over which data-parallel operations can be performed, and the types of alignment and distribution that can be specified.

The pC++ Language

pC++ [4] is a portable object-parallel programming language for both shared-memory and distributed-memory parallel systems. Traditional data-parallel systems are defined in terms of the parallel action of primitive operators on distributed arrays. *Object-parallelism* extends that model to the object-oriented domain by allowing the concurrent application of arbitrary functions to the elements of more complex distributed data structures. This allows the construction of parallel applications having complex dynamic distributed data structures, within an object-oriented framework. pC++ is based on a simple extension to C++ that provides parallelism via the *collection* construct. A collection is a distributed array of objects with additional infrastructure supporting the implementation of arbitrary distributed data structures (e.g. distributed trees of objects) over the distributed array base. pC++ provides facilities for specifying HPF-style distribution and alignment of collections.

A.2.4 Task Parallelism

Data-parallelism in programming systems has proven to be an effective means for the expression and exploitation of parallelism in many applications in science and engineering. However, more general forms of parallelism can support increased performance in a wider range of applications. Programming systems supporting *task-parallelism* allow the programmer to express parallelism involving arbitrary, independent computations on arbitrary data. Task-parallelism is proving to be useful for software modularity in multidisciplinary scientific simulation problems where interaction between separately-developed programs is important, for pipeline parallelism, and for the implementation of dynamic data-driven computation and communication in complex parallel applications [27].

Appendix B: Introduction to Database Management Systems

B.1 Overview

Many applications have data management needs that are not met by conventional programming languages. Database management systems (*DBMS*) have been developed to meet these needs. They provide application developers with data management services such as:

- **Persistence**: Programming languages provide support for the efficient manipulation of transient data — data that disappears when the process in which a program runs terminates. DBMS support manipulation of *persistent* data that can outlive the programs and processes in which it is created, accessed, and modified. A set of related persistent data managed by a DBMS is called a *database*. A single DBMS may be used to manage data in several databases.
- **Recovery**: If a system crashes due to a hardware or software fault while an application is in the process of modifying a database, then that database may be left

in an inconsistent state. A DBMS may support *recovery*, the ability to return a partially modified database to a consistent state after such failures.

- **Concurrent data access**: A DBMS may allow concurrent access to a database from multiple applications, and may provide *concurrency-control mechanisms,* which insure that all applications see a consistent view of the database while it is being concurrently accessed and updated.

- **Distributed data access**: A DBMS may support several types of distributed data access. It may support remote access to a centralized database, using a client-server DBMS architecture, or it may support access to a database in which the data itself is distributed, using a server-server DBMS architecture.

- **Query languages**: Application developers often require access to a specific subset of a database. A DBMS may provide a *query language* which allows developers to declaratively (rather than procedurally) specify what data is needed (rather than how to access it). The DBMS then takes care of extracting the desired data. This allows applications to have *data independence*, meaning that the underlying data storage format can be changed within the DBMS without having to modify the applications that access the data. Furthermore, a declarative query language specification of desired data can often be much simpler than the equivalent procedural specification of how to get it, so the use of a query language can simplify data access coding in applications.

B.2 Data Models

B.2.1 The Entity-Relationship Data Model

A *conceptual data model* is a set of concepts used to describe the structure of application-domain data at a high level of abstraction during the database design phase. The *entity-relationship (ER) model* (see [25], pp. 42, 612) is a commonly used conceptual model described below. Every DBMS has a *representational* (or *implementation*) *data model* which consists of the specific mechanisms the DBMS supports for the representation of application data. Implementing a database requires mapping its schema from a conceptual data model to the DBMS's representational data model. In later sections we describe how the ER model can be mapped to the relational and object representational data models.

In the ER model, an *entity type* is used to describe the structure of information about a set of related application-domain "things". An entity type consists of name (e.g. "person") and a set of *attributes*, which describe properties of those things. Each attribute consists of a name (e.g. "address" or "age") and a *domain* (or *value set*), which is a set of possible *values* for that attribute (e.g. the set of strings of characters (for "address") or the set of non-negative integers (for "age")). An *atomic* (or *simple*) *attribute* has the same domain as a system-provided data type in a given representational model. *Composite attributes* can be divided into several atomic attributes.

A *key attribute* is one that must have a unique value in each entity. A *weak entity type* has no key attributes. If the entities of a weak entity type are distinguished from each other by a relationship with entities in another entity type, then the relationship is called an *identifying relationship* and the other entity type is called the *identifying owner*. For example, an entity type named "interview" might have attributes "date" and "result", none of which might be required to be unique (so "interview" is a weak entity type). If the entities of "interview" are distinguished from each other by a relationship

relating interviews to entities of two other entity types, "candidate" and "company", then "candidate" and "company" are identifying owners of interview, and the relationship is an identifying relationship.

If an entity type A has every attribute of an entity type B, then we say that A *isa* B and A is a *subtype* of B. For example, if the entity type "employee" is a subtype of entity type "person", then the employee type has all the attributes of the "person" type.

An *entity* is a description of a particular application-domain thing (e.g. the person John Doe). Each entity has an *entity type* ("person" in this case) and one value for each of the attributes of that entity type (or several values, in the case of *multivalued* attributes, e.g. the attribute "college degrees earned"). The special value *null* can be used to indicate that an attribute is unknown or not applicable for a given entity. The set of all entities of a given entity type is called the *extension* (or *collection* or *entity set*) of that entity type. An entity type describes the structure of information (i.e. the *intension* or *schema*) of the entities in its entity set.

A *relationship type* is used to describe the structure of information about a type of association that can exist between types of application-domain things. A relationship type consists of a name, a set of attributes, and a set of *roles,* each consisting of a name and an entity type. The *degree* of a relationship type is the number of roles it has. For example, a relationship of degree 2 named "employment" might have the roles "employer" and "employee" with corresponding entity types "company" and "person", and might have an attribute named "date of start of employment".

A *relationship* is a description of a particular association between particular application-domain things. Each relationship has a relationship type, and each of the roles of that relationship type is filled an entity of that role's entity type. Each relationship also has a value for each of the attributes of that relationship. For example, a relationship of type "employment" might consist of the entity "John Doe" filling the

role "employee", the entity "Widgets Inc." filling the role of "employer", and the value "Jan 1 1995" for the attribute "date of start of employment".

A relationship type T among roles r_1, ..., r_n is *many-to-one to role* r_i if given entities e_1, ..., e_{i-1}, e_{i+1}, ..., e_n of appropriate entity types to fill all roles but r_i, there can be at most one entity e_i (filling role r_i), related to e_1, ..., e_{i-1}, e_{i+1}, ..., e_n by a relationship of type T; that is, the relationship is a (partial) function from the entity types of r_1, ..., r_{i-1}, r_{i+1}, ..., r_n to the entity type of r_i. A relationship type is *one-to-one* if it is many-to-one to all of its roles. A relationship type is *many-to-many* if it is not many-to-one to any of its roles. For example, the relationship type "employment" may be many-to-one to its "employer" role, meaning that each employee has at most one employer. If a relationship type "management" having roles "manager" and "department" is one-to-one, then each manager manages at most one department and each department has at most one manager. If a relationship type "works on" with roles "employee" and "project" is many-to-many, then each employee may work on several projects, and each project may have several employees working on it.

B.2.2 The Relational Data Model

The relational model is a representational data model originally developed to support data management in business applications. The only data structures supported in the relational model are named tables with named columns, where each column (called a *field*) can hold data of a user-specified system-defined data type chosen by the database implementor (e.g. integer or string). The tables are called *relations* and their rows are called *tuples* or *records*. (In this section relations are called "tables" to avoid confusion with "relationships".) A *primary key* of a table is a field used to identify the tuples in the table. A *foreign key* of a table is a field used to refer to tuples in another table, by storing

the values of the other table's primary key. The following is a common method of mapping the ER conceptual model to the relational model:

1) **Entity types** are mapped to tables called *entity tables*. Each atomic, single-valued attribute of an entity type is mapped to a field of its entity table. Each composite attribute is broken into its constituent atomic attributes, which are mapped to fields of the entity table. For non-weak entity types, a key attribute is chosen to be the primary key for the corresponding entity table. Each multivalued attribute is mapped to a separate table having one field that holds individual values, and another foreign key field that identifies the owner of each value, by referring to the primary key of the corresponding entity table. Type/ subtype relationships between entity types do not map particularly well to the relational model. See ([25], p. 629) for several possible mappings.

2) **Entities** are mapped to tuples in the corresponding entity tables (and possibly tuples in related tables for multivalued attributes).

3) Relationship types are mapped in various ways, depending on their variety:

   - Each **binary one-to-one** relationship type is mapped into a foreign key field of the entity table E corresponding to one of its roles. That foreign key field is used to refer to the primary key of the entity table corresponding to the other role. Any attributes are mapped to fields of E, as well.
   - Each **binary many-to-one** (to role R) relationship type is mapped into a foreign key field of the entity table E corresponding to the "from" role (i.e., the other role besides R), which is used to refer to the primary key of the entity table corresponding to role R. Any attributes are mapped to fields of E, as well.
   - **Non-binary identifying** relationship types are many-to-one to all roles but

the role corresponding to the weak entity type W that each identifies, and are mapped as above, into the entity table of W. (Binary identifying relationship types are binary many-to-one.)

- **All other non-binary** relationship types and **binary many-to-many** relationship types are mapped as follows: each is mapped to a *relationship table* having a foreign key field for each role, used to refer to the primary key field of the corresponding entity table. Any attributes are mapped to fields of the relationship table.

4) **Relationships** are mapped to tuples in relationship tables, or to portions of tuples in entity tables, depending on how their relationship types were mapped.

B.2.3 The Object Data Model

The object data model originated in object-oriented programming languages. Systems supporting this model allow the application developer to create complex application-specific data types. These systems provide several simple atomic data types, plus mechanisms application developers can use to generate more complex, application-specific data types:

- *Collection types* such as lists, sets, and arrays of any data type can be defined.
- *Class types* group several component data types into a new data type. A class groups several *member* data types into a new data type. A class can *inherit* members of related classes. Values of class type are called *objects*. Each object has a unique identity by which it can be referred to, called an *object identifier* (or *OID*), that is independent of the value it contains. (In contrast, the identity of a tuple in the relational model depends on the value of its primary key field, which is part of the value of the tuple itself.)

- *Reference types* are used to keep track of relationships between objects. A reference to a class is a data type whose values are OIDs of objects of that class. References have functionality similar to foreign keys in the relational model, with the following difference: in the object data model, a reference to a given object will still point to that object after every value in the object changes; in the relational model, a foreign key can lose its meaning if the value of the tuple to which it refers changes, since a foreign key refers to the primary key of that tuple, which is itself a part of the value of that tuple. References also have functionality similar to that of pointers in programming languages, except that reference values cannot become corrupted; they are invalidated automatically when the referenced object is deleted.

The ER model can be mapped to the object data model as follows:

1) Each **entity type** is mapped to an *entity class* having members corresponding to the entity type's attributes. An entity type that is a subtype of another entity type is mapped to a class that inherits members from the class corresponding to the supertype.

2) **Entities** of a given entity type are mapped to objects of the class corresponding to that entity type.

3) **Relationship types** are mapped as follows:

    - Each **binary** relationship type is mapped to a set-valued or list-valued reference component in both of the classes corresponding to the entity types participating in the relationship. If the relationship type is many-to-one to any role r, then the class corresponding to the other role (not r) can have an ordinary (single-valued) reference component. If the relationship type has attributes, then objects of a class having both a reference member and appropriate attribute members are substituted for the reference member(s)

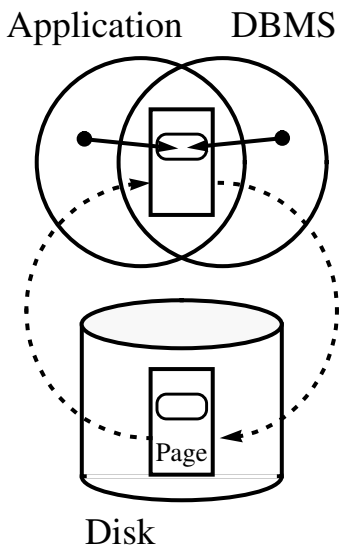


Figure B.1: Single-Representation DBMS Architecture

The representation of data on disk is designed to be as close as possible to the programming language's representation of data in memory.

in one of the classes.

- Each **nonbinary** relationship type is mapped to a set-valued or list-valued reference member in each of the classes corresponding to roles of the relationship, and a *relationship class* containing a single-valued reference member for each of those roles. Any attributes are mapped to appropriately typed members of the relationship class.

4) **Relationships** are mapped to values in relation members of entity objects, and in the case of nonbinary relationships, to relationship objects.

B.3 DBMS Client-Server Architectures

B.3.1 One-Representation

In a DBMS with a one-representation architecture, the DBMS representation of data on disk is designed to be as close as possible to the programming language's representation of data in memory (see Figure B.1). This approach has several benefits. Pages on disk containing data needed by the application can be transferred directly into the

application's memory, with few data representation conversions required. Those that are required are performed "in place" on the data structures affected, overwriting old values. Information in transferred pages can become visible and modifiable by the programming language immediately, through ordinary programming language data structures. So this approach avoids the "impedance mismatch" problem described below, that occurs in DBMS with a two-representation architecture. Pages can be cached in the application's memory for as long as they are needed, then transferred directly back to disk. If the query language executes in the same run-time environment as the programming language, then query results can become available to the programming language immediately after a query completes.

In systems supporting a one-representation architecture and an object model, references to objects on disk are often represented as OIDs, while references to objects in memory are represented by memory addresses, for efficiency. When objects move between disk and memory, conversion between these two representations, or *pointer swizzling*, may be required.

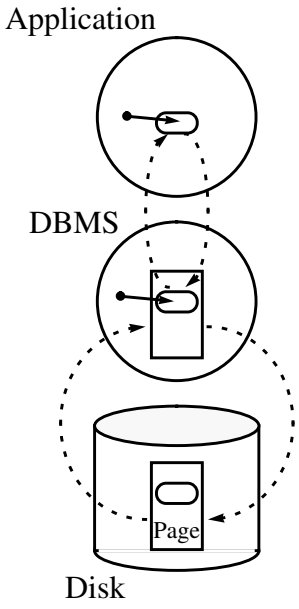


Figure B.2: Two-representation DBMS Architecture

The DBMS representation of data on disk differs significantly from the programming language's representation of data in memory.

B.3.2 Two-Representation

In a DBMS with a two-representation DBMS/programming-language architecture, the DBMS representation of data on disk differs significantly from the programming language's representation of data in memory (see Figure B.2). This type of architecture may be employed when the type system of the programming language differs significantly from the representational data model of the DBMS, or when the DBMS is to be accessed through several different programming languages with incompatible data representations. The drawback to this architecture is that data must be translated between the DBMS representation and the programming language representation every time it crosses between the two systems, resulting in inefficiency and often in inconvenience for the application developer. This problem is known as *impedance mismatch*.

B.4 Families of Currently Available DBMS

It is possible to identify several families of DBMS currently available.

- *Relational DBMS* employ a relational data model and in the past have been implemented using a two-representation architecture.
- *Extended relational DBMS* have a relational data model that is extended with features such as OIDs for tuples, complex application-specific types for fields, and inheritance of properties for relations. These features allow relations to have the semantics of classes and tuples to have the semantics of objects, supporting an object data model on top of the relational model. A two-representation architecture has been used in most of these systems ([11], p. 154).

- *Object-oriented DBMS* have an object-oriented data model and are usually integrated with one or more object-oriented programming languages, using a one-representation architecture.

Bibliography

[1] Donald Acton, Terry Coatta, and Gerald Neufeld. The raven system. Technical Report TR 92-15, University of British Columbia Computer Science, August 1992.

[2] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/ Cummings, second edition, 1994.

[3] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, Oct 1994.

[4] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[5] R Bordawekar, J.M. Del Rosario, and A Choudhary. Experimental performance evaluation of touchstone delta concurrent file system. In *Proceedings of International Conference on Supercomputing 93*, July 1993.

[6] B.Stroustroup. *The C++ programming language*. Addison Wesley, 1986.

[7] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C.K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data*, May 1994.

[8] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the 007 OODBMS benchmarking effort. In *Proceedings of OOPSLA 94 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1994.

[9] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The architecture of the EXODUS extensible DBMS. In *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data*, pages 52–65, May 1986.

[10] Andy Carmichael. *Object Development Methods*. SIGS Books, 1994.

[11] R.G.G. Cattell. *Object Data Management*. Addision-Wesley, revised edition, 1994.

[12] Barbara Chapman, Piyush Mehrotra, John Van Rosendale, and Hans Zima. A software architecture for multidisciplinary applications: Integrating task and data parallelism. In *CONPAR 94*, 1994.

[13] Peter Chen and et. al. RAID: High-performance, reliable secondary storage. Technical Report UCB/CSD-93-778, University of California at Berkeley, Department of Computer Science, 1993.

[14] Alok Choudhary. Parallel I/O systems: Guest editor's introduction. *Journal of Parallel and Distributed Computing*, 17:1–3, 1993.

[15] Alok Choudhary et al. PASSION: Parallel And Scalable Software for I/O. Technical Report NPAC SCCS-636, Syracuse University, September 1994.

[16] P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: A parallel file I/O interface for MPI. Technical Report NAS-95-002, NASA Ames Research Center, January 1995.

[17] Peter Corbett and Dror Feitelson. Design and implementation of the vesta parallel file system. In *Proceedings of Scalable High Performance Computing Conference, SHPCC 94*, May 1994.

[18] D. Culler and et. al. Parallel programming in Split-C. In *Proceedings of Supercomputing 93*, November 1993.

[19] David Culler, Richard Karp, and et. al. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 93*, May 1993.

[20] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, sixth edition, 1995.

[21] David J. DeWitt, Jeffrey F. Naughton, John C. Shafer, and Shivakumar Venkataraman. ParSets for parallelizing OODBMS traversals: Implementation and performance. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, September 1994.

[22] D.Gannon, P.Beckman, F.Bodin, J.Gotwals, S.Narayana, S.Srinivas, and B.Winnika. Sage++: An object oriented toolkit for program transformations. In *Proceedings of Oonski 94*, April 1994.

[23] D.Gannon, S.Yang, S.Srinivas, V.Menkov, and P.Bode. Object-oriented methods for parallel execution of astrophysics simulatons. In *Proceedings of Mardigras 94*, February 1994.

[24] D.Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1:25–42, 1993.

[25] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, second edition, 1994.

[26] Center for Research in Parallel Computing. High Performance Fortran Forum. On the World Wide Web at http://www.crpc.rice.edu/HPFF/home.html.

[27] Ian Foster. Task parallelism and high-performance languages. *IEEE Parallel and Distributed Technology*, pages 27–36, Fall 1994.

[28] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of the concurrent file system of the Intel iPSC/2 hypercube. *Journal of Parallel and Distributed Computing*, 17:115–123, 1993.

[29] N. Galbreath, W. Gropp, and D. Levine. Application driven parallel I/O. In *Proceedings of Supercomputing 93*, November 1993. Also Argonne technical report MCS-P381-0893.

[30] D. Gannon, S. X. Yang, and P. Beckman. *User Guide for a Portable Parallel C++ Programming System, pC++*. Computer Science Department, Indiana University, 1994.

[31] Jacob Gotwals, Suresh Srinivas, and Dennis Gannon. pC++/streams: a library for I/O on complex distributed data structures. In *Proceedings of the 1995 Symposium on the Principles and Practice of Parallel Programming*, July 1995.

[32] Jacob Gotwals, Suresh Srinivas, and Shelby Yang. Parallel I/O from the user's perspective. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, February 1995.

[33] Graphics and Visualization Laboratory of the Army High Performance Computing Research Center. Brick of bytes (BOB) interactive volume renderer.

[34] Mark Henderson, Bill Nickless, and Rick Stevens. A scalable high-performance I/O system. In *Proceedings of Scalable High Performance Computing Conference*, SHPCC94, May 1994.

[35] L. Hernquist and J. P. Ostriker. A self-consistent field method for galactic dynamics. *The Astrophysical Journal*, 386:375–397, 1992.

[36] Intel Supercomputing Systems Division. Paragon Users Guide.

[37] Jr James V. Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain*, July 1995.

[38] John Karpovich, Andrew Grimshaw, and James French. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proceedings of OOPSLA 94, 9th conference on object-oriented programming systems, languages, and applications*, October 1994.

[39] Won Kim. *Modern Database Systems: The Object Model, Interoperability, and Beyond*. Addision-Wesley, 1995.

[40] Thomas Kwan and Daniel Reed. Performance of the CM-5 scalable file system. In *Proceedings of International Conference on Supercomputing 94*, July 1994.

[41] Lamb et al. The ObjectStore database system. *Communications of the ACM*, October 1991.

[42] Edward Lee and Randy Katz. An analytical performance model of disk arrays. In *Proceedings of SIGMETRICS 93*, April 1993.

[43] Bill Nitzberg and Samuel A. Fineberg. Parallel I/O on highly parallel systems. Technical Report NAS-95-022, Numerical Aerospace Simulation Facility, NASA Ames Research Center, M/S 258-6, Moffett Field, CA, 1995. Supercomputing '95 Tutorial M6 Notes.

[44] Juan Rosario and Alok Choudhary. High-performance I/O for massively parallel computers: Problems and Prospects. *IEEE Computer*, pages 59–68, March 1994.

[45] Kent Seamons and Marianne Winslett. An efficient abstract interface for multidimensional I/O. In *Proceedings of Supercomputing 94*, November 1994.

[46] Kent Seamons and Marianne Winslett. Physical schemas for large multidimensional arrays in scientific computing applications. In *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, September 1994.

[47] Thinking Machines Corporation. *CMMD Reference Manual*. Chapter 12.

[48] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1: Classical Database Systems. Computer Science Press, 1988.

[49] Greg Wilson. *Parallel Programming Using C++*. MIT Press, 1996. See the chapter: Object-Parallel Programming with pC++ (Shelby X. Yang, Dennis Gannon, Peter Beckman, Jacob Gotwals, and Neelakantan Sundaresan).

Jacob K. Gotwals

Email: jgotwals@cs.indiana.edu

Education:

Ph.D.	in Computer Science	Indiana University	1996
M.S.	in Computer Science	Indiana University	1992
B.A.	in Computer Science (with Distinction)	Swarthmore College	1990

Experience:

1996-? Intel Corporation, Microprocessor Division — Santa Clara, CA

1993 NASA Numerical Aerodynamic Simulation Center, Parallel Development Tools Group (Summer Internship) — Mountain View, CA

Parallel Performance Tuning Tools (C, Motif): Support visualization and analysis of execution patterns in parallel programs. Part of the AIMS toolkit.

1992-96 High Performance Computing Group — Indiana University, Bloomington, IN

Parallel Filestreams (C++, UNIX, Paragon, CM-5, SGI Power Challenge): C++ library for parallel file I/O. Supports a parallel version of C++ iostreams.

Persistent Collections (C++, Object-Oriented DBMS): C++ library for object-oriented database management in parallel programs.

1991-92 Geochemistry Simulation Group — Indiana University, Bloomington, IN

3D Scientific Visualization Tools (C, AVS): Post-process output of geochemical simulators to generate 3D movies of oil reservoir development.

Parallel Simulation Framework (C, FORTRAN): Application framework for building parallel geochemical simulators

Publications:

pC++/streams: A Library for I/O on Complex Distributed Data Structures.
Jacob Gotwals, Suresh Srinivas, and Dennis Gannon.
(in *Principles and Practice of Parallel Programming*, 1995)

Parallel I/O from the User's Perspective.
Jacob Gotwals, Suresh Srinivas, and Shelby Yang.
(in *Frontiers of Massively Parallel Computation*, 1995)

Performance Indices for Message Passing Parallel Programs.
Sekhar Sarukkai, Jerry Yan, and Jacob K. Gotwals.
(in *International Conference on Supercomputing*, 1994)

Sage++: An Object-Oriented Toolkit for Building FORTRAN and C++ Restructuring Tools.
Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals,
Srinivas Narayana, Suresh Srinivas, and Beata Winnicka.
(in *Proceedings of OONSKI 1994*)

Object-Parallel Programming with pC++.
Shelby X. Yang, Dennis Gannon, Peter Beckman, Jacob Gotwals,
and Neelakantan Sundaresan.
(in *Parallel Programming Using C++*, MIT Press, 1996)