NOTE TO USERS

The original manuscript received by UMI contains pages with slanted, light, and/or indistinct print. Pages were microfilmed as received.

This reproduction is the best copy available

UMI

Performance Optimization for Distributed-Shared-Data Systems

by

Craig Steven Bruce

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 1998

©Craig Steven Bruce 1998

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-32819-8

Canada

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

This thesis introduces an all-user-level-library approach to using distributed shared data in a network-of-workstations environment, which has a relatively high-latency, low-bandwidth communication network. The approach is based upon managing arbitrary distributed user-defined dynamic data structures. This approach is conceptually straightforward and provides a programmer with a high degree of control over the system. The organization of the system also provides many opportunities for optimizing data access and migration to hide communication latency. The available techniques include using direct memory pointers for accessing data objects, prefetching, protocol tuning, invalidation versus updating for consistency management, optimistic locking, "tagged" objects, and control of many lower-level issues and trade-offs.

The basic mechanisms of the system also provide a suitable foundation for higher-level mechanisms. These mechanisms provide convenient and efficient matrix processing, linked-structure processing, and programming-language interfaces. Techniques for making the best use of the system and translating programs from other distributed-shared-data systems are also discussed.

Obtaining good performance from distributed-shared-data systems is also important. Several example applications are used to demonstrate the performance of the system. As well, the examples provide an opportunity to discuss techniques that are useful in tuning applications for the environment.

Acknowledgments

I would like to thank my supervisor, Dr. David Taylor, for his guidance, wisdom, patience, and sense of humor during the course of my studies. Thanks also to Dr. Wayne Loucks, Dr. Jay Black, and the other members of my examining committee for their suggestions for improvements to this thesis document.

I also thank all of the people who helped along the way, in a personal way, to one degree or another, whether they knew it or not: David Taylor (again), Keith Pomakis, Mike Galluchon, Luise Bacher, Derek Euale, Paul Ward, Mark Fox, Jamie Hodge, Doreen Galli, Stuart Wachsberg, and, of course, my family, Bill, Marilyn, Randy, and Crystal Bruce, not to forget Niko and Cleo. I also thank for inspirational music during my studies: U2, Sarah McLachlan, Van Halen, and Pink Floyd.

Finally, thanks to Natural Sciences and Engineering Research Council (NSERC), Information Technology Research Centre (ITRC), Institute for Computer Research (ICR), the University of Waterloo (UW), and the Shoshin Research Group for financial support ($$$).

Contents

1 Introduction **1**

1.1 Distributed Systems 1

1.2 Networks of Workstations 2

1.3 Interprocess Communication 4

1.4 Distributed Shared Data 5

1.5 Thesis Objectives 7

1.5.1 Transport Layer 8

1.5.2 Basic Objects and Synchronization 10

1.5.3 High-Level Mechanisms 12

1.5.4 Performance & Optimization 14

1.5.5 Applications 15

1.6 Thesis Organization 16

2 Related Research **17**

2.1 Classic Distributed Shared Memory 17

2.1.1 Using Shared Memory 17

2.1.2 Implementation 19

2.1.3 Evaluation 23
2.2 Memory-Consistency Models 26
2.3 Selected DSM Systems 30
2.3.1 TreadMarks 30
2.3.2 CVM 31
2.3.3 Munin 33
2.3.4 Midway 34
2.4 Shared Data-Object Systems 35
2.4.1 Linda 35
2.4.2 Orca 37
2.4.3 ABC++ 39
2.4.4 Problems with the Object-Oriented Approach 40
2.4.5 Shared Regions 41
2.5 Message Passing 42
2.5.1 Remote Procedure Call 43
2.5.2 Multicasting and Broadcasting 43
2.6 Conclusions 45

3 Basic DSDS System **48**

3.1 Motivation & System Concepts 48
3.2 Primitives 51
3.3 System Data Structures 57
3.3.1 Shared-Memory Heap 57

3.3.2 Named Objects 59
3.4 Object-Management Protocol 59
3.4.1 Data-Object Control Information 60
3.4.2 Acquiring a Lock 65
3.4.3 Handling Invalidations 68
3.4.4 Granting Locks 69
3.4.5 Asynchronous-Protocol Evaluation 75
3.4.6 Barrier Implementation 76
3.5 Discussion of Design Issues 77
3.6 Conclusions 82

4 Performance Enhancements **86**

4.1 Prefetching 87
4.1.1 Motivation 87
4.1.2 Explicit Prefetch 88
4.1.3 Prefetch Links 89
4.1.4 Asynchronous Grant 92
4.1.5 Pre-Invalidation 94
4.1.6 Local Invalidation 95
4.1.7 Prefetching in General 96
4.2 Non-Stalling Invalidations 97
4.2.1 Stall on Lock 97
4.2.2 Stall on Next Operation 99

4.2.3 Stall On Release 100
4.2.4 Asynchronous Invalidation 101
4.3 Update Protocol 102
4.3.1 Motivation 102
4.3.2 Application Interface 103
4.3.3 Implementation 103
4.3.4 Broadcasting and Multicasting 104
4.3.5 Tuning 106
4.4 Optimistic Locking 107
4.4.1 Motivation 107
4.4.2 Application Interface 108
4.4.3 Implementation 110
4.5 Tagged Objects 113
4.5.1 Concept 113
4.5.2 Application Interface 114
4.5.3 Implementation 116
4.6 Protocol Issues 118
4.6.1 Exploiting Semantics 118
4.6.2 Light-Weight Locking 119
4.6.3 Limiting Data Transfer 120
4.6.4 Object-Owner-Locating Cost 122
4.6.5 Reader Lock Granting 125
4.6.6 Deadlocks 125
4.7 Conclusions 126

5 High-Level Mechanisms and Usage **131**

5.1 Motivation 131

5.2 Matrix Processing 132

5.2.1 Data Organization 132

5.2.2 Possible Approaches 135

5.2.3 Matrix Creation 141

5.2.4 Matrix Accessing 143

5.3 Linked-Structure Processing 144

5.3.1 Link Cursors 144

5.3.2 Concurrency Control 146

5.4 Higher-Level Interface 148

5.4.1 Block-Structured Languages 148

5.4.2 Data-Structure Models 150

5.4.3 Transactions 156

5.5 Techniques for Using DSDS 157

5.5.1 Mapping Data Structures to DSDS 157

5.5.2 Classic DSM 159

5.5.3 Munin 159

5.5.4 Linda 160

5.5.5 Orca 161

5.5.6 Shared Regions 162

5.6 Conclusions 162

6 Test Applications **165**

6.1 Test Environment 165

6.2 Matrix Multiply 168

6.2.1 Sequential Implementation 169

6.2.2 Distributed Implementation 170

6.2.3 Distributed Performance 172

6.3 All-Pairs Shortest Path 176

6.3.1 Sequential Implementation 176

6.3.2 Distributed Implementation 177

6.3.3 Distributed Performance 179

6.4 Successive Overrelaxation 180

6.4.1 Sequential Implementation 181

6.4.2 Distributed Implementation 182

6.4.3 Distributed Performance 183

6.5 Gaussian Elimination 185

6.5.1 Sequential Implementation 185

6.5.2 Distributed Implementation 186

6.5.3 Distributed Performance 187

6.6 Game-Tree Searching: Othello 188

6.6.1 Sequential Implementation 189

6.6.2 Distributed Implementation 191

6.6.3 Distributed Performance 192

6.7 Performance Comparison to Other Systems 194

6.7.1 Matrix Multiply with Message Passing 195
6.7.2 CVM 195
6.7.3 Orca 198
6.8 Conclusions 200
6.8.1 Summary 200
6.8.2 Distributed Algorithms 203
6.8.3 Future Work 208

7 Conclusion **209**
7.1 Research Summary 209
7.1.1 Basic DSDS System 209
7.1.2 Performance Enhancements 210
7.1.3 High-Level Mechanisms and Usage 211
7.1.4 Test Applications 212
7.2 Contributions to Computer Science 213
7.3 Future Work 213
7.4 Conclusion 215

A Prototype-Implementation Details **216**
A.1 Design Restrictions 216
A.2 Unix-Implementation Implications 218

Bibliography **220**

List of Tables

4.1 Conservative Object-Locking Sequence 129

5.1 DSDS Equivalences to Munin Coherence Annotations 160

6.1 DSDS-Primitive and Sequential-Operation Times 167

6.2 Network RPC Times 167

6.3 Sequential Matrix-Multiply Performance 170

6.4 Basic Distributed Matrix-Multiply Performance 173

6.5 Option Distributed Matrix-Multiply Performance 174

6.6 Start-Barrier Distributed Matrix-Multiply Performance 175

6.7 All-Pairs Shortest Path Distributed Performance 179

6.8 Successive Overrelaxation Distributed Performance 184

6.9 Gaussian-Elimination Distributed Performance 187

6.10 Distributed Othello Performance 193

6.11 Rees and Black Message-Passing Matrix-Multiply Results, n=256 . . 196

6.12 Ward's CVM Performance Results 197

6.13 Orca Performance Results 199

6.14 Optimizations Applied to Applications 207

List of Figures

1.1 Network of Workstations 2

1.2 DSD System Software Layers 9

2.1 DSM VM-Page-Access Transitions 22

2.2 DSM with Release Consistency 29

2.3 TreadMarks Multi-Writer Lazy-Release Consistency 31

2.4 RPC Primitives and Messages 44

3.1 Structure of an Allocated Object 52

3.2 In-Core Object Record 61

3.3 Object States on a Single Host 64

3.4 Basic Remote-Lock-Request Algorithm 66

3.5 Lock-Message Format 66

3.6 Object Invalidation 69

3.7 Granter Algorithm: Owner, X-lock Requested. 70

3.8 Granter Algorithm: Owner, S-lock Requested. 71

3.9 Granter Algorithm: Not Owner. 72

3.10 Detection of Lost Grant Message 74

4.1 Prefetching-Links Example 92

4.2 Storage Structure on a Host of an Object with Tags 116

5.1 Examples of Dividing a Matrix into Tiles 133

5.2 Tile-Oriented Approach to Matrix Multiply 137

5.3 Structure of Mapped Objects 139

5.4 Structure of an Automatically Mapped Matrix 142

5.5 Link Cursor example in C 145

5.6 Link Cursor example in C++ 145

5.7 Block-Structured Shared Variable Access 149

5.8 Possible Language-Level DSDS Types 149

5.9 Classic Linked-List Structure 151

6.1 Splitting and Transfer of Matrices for Matrix Multiply 172

6.2 Sequential All-Pairs Shortest-Path Algorithm 177

6.3 All-Pairs Shortest Path Row Assignment 178

6.4 Sequential Othello Tree-Search Algorithm 189

6.5 Examples of Data Distribution for Matrix Multiply 205

Chapter 1

Introduction

1.1 Distributed Systems

Advances in hardware technology have frequently been the driving force behind advances in system-software technology. In the early days, centralized batch-processing systems were used, then large multiprogramming systems, and today, distributed systems based on interconnected local-area networks (LANs) are popular. System software and applications have evolved significantly to utilize the potential of these different types of systems.

A *distributed system* is a collection of independent computers that are interconnected by a communication network and are capable of working together, plus all system software required to make effective use of such a hardware configuration. These independent computers have no physically shared memory, so the only physical means of communication and synchronization between computers is passing messages through the network.

Distributed computing systems have become more widely used in recent years because of their advantages over centralized systems: a higher performance/price ratio ("more bang for the buck"), potentially increased reliability and availability because of

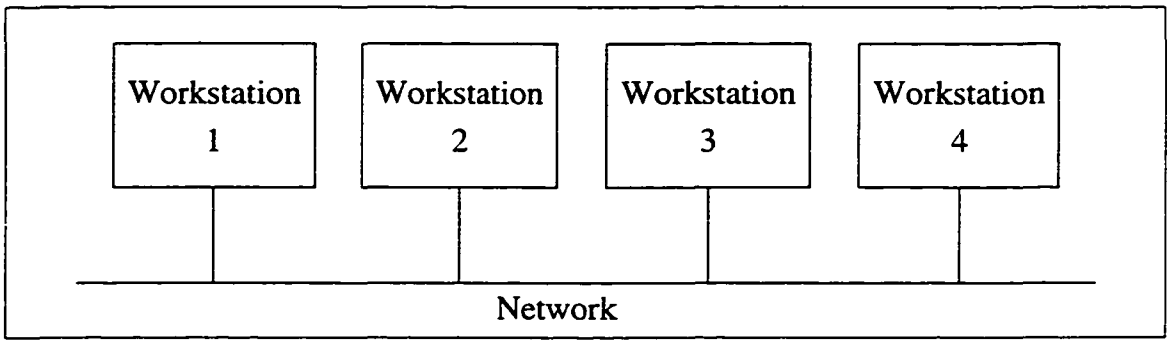


Figure 1.1: Network of Workstations

partial failure modes (if protocols are implemented appropriately), shared resources, incremental growth and online extensibility, and the fact that some applications are inherently distributed [Tanenbaum92]. People also find it very convenient to have a fairly powerful machine all to themselves, since it provides fast and consistent response times for interactive computing.

This thesis introduces an all-user-level-library approach to provide a shared-data programming environment for distributed computing on networks of workstations. The approach is based upon managing arbitrary distributed user-defined dynamic data structures. This approach is conceptually straightforward and provides a programmer with a high degree of control over the system. The organization of the system also provides many opportunities for optimizing data access and migration.

1.2 Networks of Workstations

A Network of Workstations (see Figure 1.1) is a specific type of distributed system in which each computer in the system has more or less equal computing capacity, where this computing capacity is greater than that of a personal computer but which is fairly moderate compared to that of mainframe and supercomputers. They are typically intended to be used by one user but have full multiprogramming capability

and spare cycles when no user is signed on or when the user is not making full use of the machine. They typically have one main processor, although some have a small number of processors (up to four). A major reason that workstations are popular is economics, as networks of workstations are significantly less expensive than mainframe and supercomputers, and the capacity of workstations is growing at a faster rate than that of supercomputers, because of their larger-volume market [Anderson95].

Network-data-transfer rates for workstations are typically on the order of 10 Mbps to 100 Mbps (and rates are increasing). This is relatively fast; however, communication latency is a significant problem in a workstation environment. Message latency is the period of time between the instant that one process makes a system call to send a message and the instant that another process returns from a system call to receive it. "Communication cost is typically on the order of milliseconds [in Workstation-LAN systems]" [Bal89]. This delay is not likely to decrease significantly in the foreseeable future, since it represents user-process, kernel, context-switching, device-driver, hardware-interface, and network-access-time overheads, in addition to the actual "wire time" of the message being sent.

The wire time is the physical-network-transmission time of a message (from the time the first bit leaves the source until the last bit arrives at the destination), which can be calculated as the message size divided by the bit rate, plus the network propagation time. Wire time has been decreasing significantly (since network data rates are increasing), but this time is only a small part of the total cost [Thekkath91] [Parsons97]. Real progress will involve the removal of a substatial amount of the software from the process of sending and receiving messages, which goes beyond the scope and economics of typical workstations. Instead, effort can be made to reduce the impact of message latency on distributed computation.

1.3 Interprocess Communication

Programming distributed applications using the raw network features of a distributed system would be inconvenient and non-portable, so systems have been devised that make network-communication features available to programmers in a convenient and hardware-independent way. The abstractions are made available to processes rather than processors, so the mechanisms are referred to as *interprocess communication*.

Interprocess communication (*IPC*) refers to how *processes* interact with one another and exchange information. In a distributed system, processes must cooperate, communicate, and synchronize with one another to achieve their intended function. The arrangement of processes and of their interactions define how effectively and efficiently a distributed application will operate. A large number of communication mechanisms and process-structuring schemes have been devised in the past.

An important challenge in devising IPC schemes for use with distributed systems is to achieve a maximal amount of parallelism (simultaneous execution by different processors) between the processes of an application. Provided most of this is useful computation, this allows for the *speedup* of an application, which is defined as the ratio of the wall-clock execution time of the distributed version divided by execution time of an appropriately tuned sequential (one-processor) version of the application. A speedup of greater than 1.0 means that the distributed version indeed executes faster. The maximum speedup expected by a distributed application is equal to the number of processors, although in practice speedup is less than this because of system overheads and internal application synchronization and communication. These factors need to be minimized for best performance.

Automatic parallelism (parallelism determined and generated by a compiler) is not likely to be very effective on general problems on MIMD (Multiple-Instruction-stream Multiple-Data-stream) machines [Bal91]. (Distributed systems are a type of MIMD machine.) Thus, manually organized and tuned distributed applications will

generally give the best performance.

Other important measures of an IPC scheme include expressiveness, simplicity, convenience, familiarity, ease of conceptualizing process interactions, and ease of handling partial failures. Expressiveness refers to how few declarations and invocations are required to perform complicated operations. Simplicity means introducing "a minimum number of concepts with simple rules for their combination" [MacLennan87]. Familiarity means using mechanisms and concepts that are similar to and consistent with what programmers are already trained and experienced in using. Conceptualizing process interactions allows a programmer to intelligently choose the best methods and algorithms for writing correct and efficient programs by knowing the inherent costs and other implications of the actual underlying system operations. Partial failures result when parts of a distributed system (usually machines or networks) malfunction or stop while the rest of the system continues to function properly. Production IPC systems should be able to detect and recover from these partial failures and assist applications in recovering their internal consistency.

1.4 Distributed Shared Data

There are two general approaches to interprocess communication used in any type of system: shared memory and message passing. With shared memory, a value is placed in a certain memory location and all processes can access it directly. With message passing, a message (value) is sent from one process to another (or others) using explicit operations. Each of these approaches has its advantages and disadvantages compared to the other. In a system with physically shared memory, shared-memory communication is instantaneous and cheap whereas message passing is relatively expensive, especially if operating-system overhead and bulk data copying are involved. On the other hand, shared memory requires auxiliary mechanisms for coordination whereas message passing has synchronization integrated with communication.

The shared-memory paradigm is more appropriate for some applications and the message-passing paradigm is more appropriate for others. Applications that need global state information are hard to program using message passing; programs based on asynchronous messages are hard to understand and debug; and a remote procedure call is two to four orders of magnitude slower than reading local data [Bal91]. Therefore, there are weaknesses with each of the approaches for certain applications.

In a distributed system there is no physically shared memory between processors, so all interprocess communication, at some level, is based on message passing. (Shared-memory semantics can be emulated in this environment.) However, the issue of whether a shared-memory or message-passing paradigm is better is still open to debate. Depending on the type of application, it may be easier and more efficient to use the semantics of shared memory. Emulated shared memory provides the property of caching memory variables, used during computation, on the local host. For example, the cost of accessing up-to-date information that is cached is insignificant compared to the cost of requesting up-to-date information from a server in a message-passing scheme. Emulated shared memory can also provide "active receivers," where only processes that are currently interested in receiving a message (access to shared-memory variables) will incur the cost of doing so. Synchronization is also more flexible, since specific messages do not need to be generated and received in a rigid order at specific points during computation; the hosts may access shared variables at any time.

In order for a distributed application to work well with a shared-data paradigm, the data-access pattern must have a high degree of locality, a very low proportion of updates to reads, or a large amount of user work performed, on average, for each access. In fact, if an application does not have at least one of these properties, we believe it is very unlikely that the distributed-shared-data paradigm is going to allow the application to work efficiently. For such an application, we believe that no distributed implementation will work efficiently. (We use the term "distributed shared data" (DSD) to refer generally to all shared-data systems and approaches for

distributed systems, including distributed shared memory and shared user-defined objects.)

For implementing distributed shared data, a straightforward simulation of shared variables is unattractive. Therefore, a more sophisticated, lower-cost approach is required, which provides the same single-copy memory-access semantics as physically shared memory. The challenge is to provide shared-data access efficiently and provide a convenient application interface. Distributed Shared Memory (DSM), using virtual-memory hardware, is a popular way of providing DSD, and all-user-level-library systems have advantages, too. The objective is to provide effective parallelism in an environment of a network of workstations with relatively high-latency/low-bandwidth communication compared to specialized parallel computing systems.

1.5 Thesis Objectives

Distributed systems provide many advantages over centralized systems, including incremental scalability, shared resources, and economics. In particular, networks of workstations have become popular because of their economic advantages and their appropriateness for single-user interactive computing. An unfortunate property of networks of workstations for distributed computing is that while they can have high-bandwidth networks, they also have rather high-latency communication.

Despite the advantages of distributed systems, distributed applications are difficult to write. Interprocess communication mechanisms are needed to abstract out hardware differences between systems and present a programmer with a convenient and powerful programming interface. Distributed shared data is an approach to IPC that is analogous to using real shared memory, which is an appropriate paradigm for many applications. Shared memory must be emulated by software protocols, and the mechanism must to be sophisticated and dynamic in order to give acceptable performance. Additionally, the mechanism must allow the programmer to optimize

data exchanges for performance-critical portions, since automatic tuning cannot be counted upon to deliver the best results.

The objective of this thesis is to devise an all-user-level-library approach to DSD to provide a shared-data programming environment for distributed computing on networks of workstations. The approach is based upon managing arbitrary distributed user-defined dynamic data structures. The mechanisms provided should be conceptually straightforward and provide the programmer with a high degree of control over the system. The organization of the system should also provide many opportunities for optimizing data access and migration. The applicability of these methods and optimizations to other distributed-shared-data systems must also be considered.

We became interested in this subject because it is both important and interesting. IPC and shared-data are obviously important for distributed applications. Our approach is interesting because, while it has a simple basis and simple concepts, it offers many possibilities for implementing and tuning applications.

The organization of this and other DSD systems can be viewed as comprising four layers of software: a message-transport level, basic objects and synchronization, higher-level mechanisms, and applications (see Figure 1.2). (DSM systems may have operating-system-kernel and signal-handling layers also.) Performance and optimization are important to all four layers and to the system as a whole.

1.5.1 Transport Layer

We want our transport layer to be both responsive and simple. In order to guarantee maximum responsiveness, we want to use a fully asynchronous transport layer. That is, one that can send a single message to another process without waiting for acknowledgments or responses and without delaying for any interprocess synchronization. The transport system also needs to be tuned for small objects, since this is part of the system model that we propose. The system should work well with larger

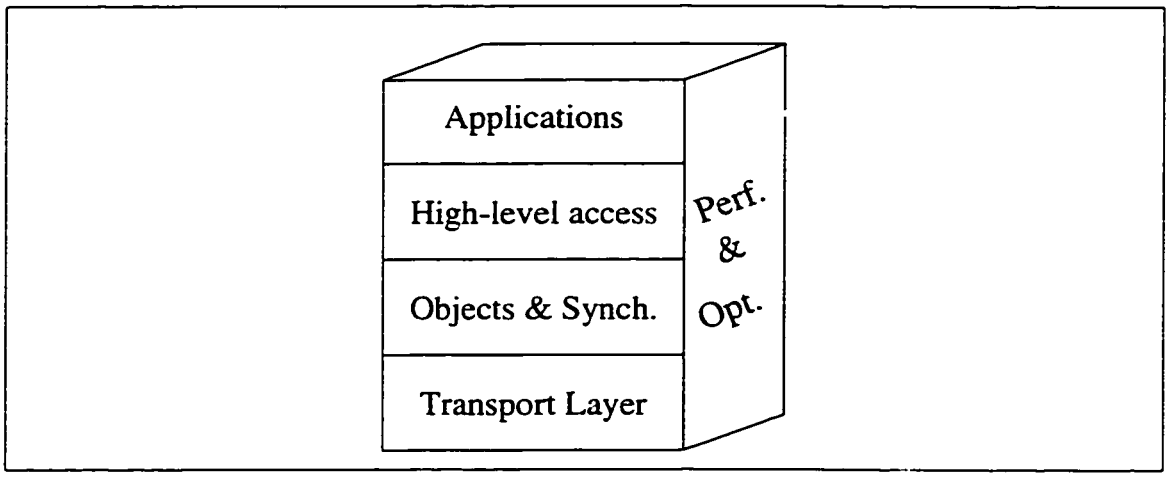


Figure 1.2: DSD System Software Layers

objects also, but this is a less critical concern since larger objects are significantly more efficient to transport than smaller ones, because message-transport overheads are amortized over a larger number of bytes for each message passed.

The transport service must be able to send a message to any process involved in an application. This may seem like an obvious and easily met requirement, but this requirement implies many problems for a connection-oriented transport service. Most importantly, it implies that a fully connected graph of inter-host connections must be maintained to ensure that all messages can be sent. Either this, or connections must be established and torn down repeatedly between hosts during execution. Also, actual interprocess interactions at the higher levels in the DSD system are connectionless, so adding connection-oriented service gives no increase in usable semantics.

In light of the above requirements, we propose to use UDP. One advantage of UDP is that it will never block on sending, which means that deadlocks are impossible in the transport layer. The main complication that arises out of using UDP is that it is unreliable. This means that the higher-level software in the system must recover from lost messages. However, this is consistent with our goals for the control protocol.

However, there are many alternatives worth considering. These include TCP,

TCP with threads, non-blocking TCP with a state machine, and RPC with threads. Briefly, difficulties with these approach are as follows. TCP is connection-oriented and blocks. TCP with threads is still connection-oriented and has increased internal complexity from thread management and inter-thread coordination. Non-blocking TCP includes management problems and reduces responsiveness from polling. TCP also can slow down operations from its internal buffering, as it can delay on sending a message in some cases. Our higher-level protocol does not guarantee that a message will be returned for each message sent. The RPC paradigm requires reply messages, even though threads can be used to make collecting replies an asynchronous activity, with increased internal complexity.

One difficulty with using a fully asynchronous transport system is that it can push complexity up into higher levels of software. We claim that because of the nature of the system, complexity is unavoidable in the object-management protocol, so only a marginal increase in complexity is encountered which is offset by increased flexibility and responsiveness. On the other hand, complexity can be encountered at higher levels in trying to match an inherently asynchronous protocol to connection-oriented semantics of a transport layer.

1.5.2 Basic Objects and Synchronization

We want a system to manage distributed memory objects. These memory objects should be managed efficiently and relatively automatically, with a programmer indicating where certain object operations should be performed. They should be easy to link into arbitrary dynamic structures. A minimum of work should be performed to transfer the contents of an object between machines.

We need a set of primitives through which the programmer can create and access the distributed objects. The basic concept of locking these objects for use and releasing them after use is similar to other systems and not very complex to understand

or use. Note that parallel and DSM systems are based on the use of locking primitives also, although DSM by itself is not really a peer of the proposed system since it needs more facilities in order to be equivalent. The most-basic operations are Create, Destroy, Lock, Release, and efficient object-data read/write.

The locking semantics provide the basic synchronization of access in the system. The locking must be provided on a per-object basis because coarser granularity locks will greatly restrict concurrency. Shared and exclusive locking is needed in order to allow full concurrency and minimize data transfer for shared-memory operations that do not really conflict. Locking conflicts will be defined in the usual way, i.e., exclusive locks conflict with any other lock, and shared locks do not conflict with other shared locks, and the locks will, of course, be *distributed* locks. Thus, the locking semantics imply that the individual memory objects will be sequentially *coherent*; i.e., the value read out of an object will always be the last value stored into it. The objects should also be *sequentially consistent* and behave as if all locking operations in the distributed system were performed in sequential order. This is the memory-system semantic assumed by all sequential programs.

An object-management protocol must be designed that operates efficiently and enforces the required semantics on the memory objects. The protocol should be fully asynchronous in order to provide maximal responsiveness. Internally, the system should have at least two threads per host: one (or more) to perform the user computation, and one to receive and process external requests for locks on objects stored locally. Internally, the system may need semaphores, etc., to manage its internal operations, but the holding time of these locks should be as short as possible and no blocking operations should ever be performed that might interfere with the responsiveness of the protocol. Also, the only receive operation performed by the secondary thread should be the one to retrieve a new request. This thread design guarantees that no incoming request can ever be delayed for an arbitrary period of time, unless a network failure occurs.

The protocol itself should recover from lost messages without relying on heavy-weight lower-level services like RPC. This is consistent with the principles described in Saltzer, Reed, and Clark [Saltzer84]. The protocol needs to be roughly based on a request-response paradigm, since this is often the most natural organization, but it should also be robust enough to handle situations that may not be possible in that environment, in order to allow for potential performance optimizations. For example, spontaneous asynchronous operations should be permitted without requiring that corresponding requests be received.

The protocol should also integrate the locking and data-flow characteristics of its semantics, in order to avoid unnecessary overhead in performing these tasks independently. Often, simple locking will be all that is needed for concurrency control. We should explore update versus write-invalidation protocols for the basis of the object-management protocol.

We are proud of the control protocol that is presented in this thesis. We envision it as perhaps being an example taken from an unwritten Ph.D. thesis on the subject of methods and tools for building asynchronous protocols, but we do not propose to write that thesis here.

Finally, another important objective for this thesis is to provide an actual implementation of the basic system, not just a simulation or theoretical model, in order to demonstrate that it works correctly and efficiently in a real-machine environment.

1.5.3 High-Level Mechanisms

The basic mechanisms of the system also provide a suitable foundation for higher-level mechanisms. Some mechanisms should be designed and implemented in order to demonstrate the ability of the system to provide higher-level mechanisms and to simplify application development.

Matrix processing (e.g., solving systems of equations) is an important application

for demonstrating distributed systems and is important in the real world for scientific computing. A method of dividing matrices into "tiles" that have different data-usage patterns is required to access large arrays efficiently. If these tiles are mapped to individual memory objects, our system can manage the objects independently and distribute the objects to various hosts for concurrent processing of the sub-matrices of data they contain. Independent management eliminates thrashing that could occur in a DSM system if independent objects were mapped to the same virtual page.

In addition to object flow and parallelism, matrix processing requires direct, efficient, processor-instruction-level access to the matrix elements, or the system overhead will be far to great for the parallelism to be worthwhile. Compiler-optimization methods will not be effective if the matrix data is not stored in contiguous memory in its usual row-major fashion. These requirements can be reconciled using "mapped objects", which are objects that have their contents stored in memory in an interleaved sub-matrix format but which are managed like regular shared-memory objects. Methods of integrating object locking and matrix-element accessing must also be explored.

Other methods should also be considered. Linked-structure processing is a common application that can be abstracted using "cursors" to handle the memory-object locking and concurrency control. Programming-language mechanisms and interfaces can also be provided to make programming more convenient. The relationship of the techniques used in our system to those available in other DSD systems must also be explored.

The suitability of the basic system to implement other abstract mechanisms also needs to be considered, for example, using it to support a transaction-processing mechanism.

1.5.4 Performance & Optimization

Performance and optimization are important aspects of all layers of the system. The measure of performance that we are most interested in is real execution time, i.e., wall-clock time. An advantage of implementing a real system is that it will demonstrate many performance issues that might otherwise not have been considered. Many performance enhancements can be devised only by studying runs of applications on an actual system to see where the performance bottlenecks are in both the system and the application.

Prefetching is an important performance optimization to consider in a DSD system. Many applications start up by shipping out volumes of data to remote hosts for processing and finish up by shipping volumes of data back to a coordinator for summarization or output. In other cases, processing can be in the form of a pipeline and the next host in the pipeline is known when local processing is complete. In others, a host may be finished with a data object and it may want to release all rights to it.

Several different methods of prefetching are needed to hide communication latency in the various situations that can arise. The flexibility of the asynchronous object-management protocol will allow these to be implemented easily.

Control of waiting for invalidation and of object updating can be used to improve performance. There are several possible points at which the write-invalidate protocol can wait (stall) to collect invalidation acknowledgments for an object. These give different timing characteristics and consistency guarantees. As well, a write-invalidate protocol can be augmented to automatically send out updates to objects using broadcasting hardware if it is available. These interrelated issues also need to be studied.

Optimistic locking is a technique that can hide latency by locking and using an object before it is known whether its data is up-to-date. The correctness of the data is

verified asynchronously and the operation fails if the data is not correct. Light-weight mechanisms that only recover the consistency of the object being optimistically locked will be considered since heavier-weight mechanisms that recover an entire distributed computation would be much to expensive and complex for our purposes. Such simple optimistic locking can be useful in the context of scientific computing.

"Tagged" or "versioned" objects provide a means of escaping the full synchronization of the basic mechanisms when working with iterative algorithms that have fixed data-sharing patterns. Contents of these objects are generated and requested by version number, so the concept is of a type of per-object pipe. Such a mechanism is relatively independent of the basic system, but could be integrated for special types of applications.

Finally, techniques allowing programmers to control lower-level mechanisms and trade-offs for better performance are considered.

1.5.5 Applications

Obtaining good performance from DSD systems is important. Several sample applications, including matrix and linked-structure algorithms, must be used to demonstrate the performance of the system. As well, the samples provide an opportunity to discuss techniques that are useful in tuning applications for the environment.

Example applications to be considered include matrix multiply, all-pairs shortest path, successive overrelaxation, gaussian elimination, and Othello^{TM} (as both a game-tree search and a linked-structure example). These example programs should demonstrate that the system works correctly for a variety of applications and can perform efficiently.

1.6 Thesis Organization

The body of this thesis follows the structure used in the preceding section. Chapter 2 examines the related research in distributed-shared memory, distributed object-based systems, and simple message passing. Chapter 3 introduces the basic system for DSDS and discusses both the protocols used in the system and the underlying design decisions. Chapter 4 introduces several additional mechanisms for enhancing the performance of the basic system. Chapter 5 describes higher-level mechanisms for convenience in programming and explores how to make the best use of the system. Chapter 6 presents several sample applications and examines their performance. Finally, Chapter 7 summarizes the thesis, presents overall conclusions, and indicates areas for possible future work.

Chapter 2

Related Research

This chapter discusses the different approaches and systems for using distributed shared data for programming in distributed systems. The literature abounds with such research; only the more common and important relevant previous work is discussed here.

2.1 Classic Distributed Shared Memory

2.1.1 Using Shared Memory

Using shared memory for Interprocess Communication (IPC) is very efficient in a centralized, uni-processor system because all updates to the shared memory are seen by all other processes instantaneously. This is because there is only a single copy[1] of the updated memory cells of the virtual-memory page in the system. The data communication is instantaneous if the single page is mapped into address spaces of multiple processes.

[1]More precisely, only a single copy is in use at a time, where that single copy may be in a processor cache, main memory, virtual-memory disk storage, or in-transit.

Interprocess synchronization is needed for accessing the shared memory. Otherwise, any process would be free to modify a memory cell at any time, and this could cause inconsistent results in larger data structures because of interleaved partial updates made by other processes. Coordination is necessary to ensure the consistency of user-defined data structures by providing a mutual-exclusion mechanism for non-atomic accessing of the data structures.

Many mechanisms have been proposed for interprocess coordination to protect the accessing of shared memory, including semaphores, event counters, and monitors [Tanenbaum92]. Briefly, a *counting semaphore* uses a variable to store the number of 'units' of some item that are available. The operations Wait and Signal allow a process to lock or release one unit of the semaphore, respectively. If zero units are available during a Wait, the waiting process is blocked until a corresponding Signal is made by some other process. A value of one for the maximum number of units can be used to implement a simple mutual-exclusion mechanism ("mutex").

An *event counter* is a monotonically increasing integer that starts from zero. The operation Read returns the current value of an event counter. Advance increments an event counter by one, and Await waits until an event counter is advanced to a specified value or beyond.

A *monitor* contains code and shared variables. Only one thread is allowed to be active inside a monitor at a time, so there can be no possibility of race conditions corrupting shared data if the code is correct. Condition variables within monitors allow for synchronization. When a thread Waits on a condition variable, another thread is allowed to enter the monitor. This other thread may unblock the waiting thread by signalling a condition variable. A problem with monitors is that their semantics become complicated when invocations are nested [Maekawa87].

Using shared memory for IPC is also efficient for a multiprocessor system where each processor has its own internal cache. Local caches are usually necessary since bus contention would be too great without them, restricting systems to only a few pro-

cessors [Tanenbaum95]. Caches give higher performance by using higher-speed RAM anyway. "However, shared-memory multiprocessors typically suffer from increased contention and longer latencies in accessing the shared memory, which degrades peak performance and limits scalability compared to distributed systems. Memory-system design also tends to be complex" [Protić96]. This is because in this environment, there can be multiple copies of the same data in the system: in the caches of different processors and in the main shared memory. Therefore, a coherence protocol must be employed between the processor caches. The techniques of cache write-throughs, updates and invalidations are used [Tanenbaum95]. These techniques are practical because of the nature of the interconnection network between the processors: a high-speed, short-distance parallel bus with a very short communication latency.

Some of the advantages of loosely coupled "*shared-nothing*" computers over multiprocessors are that they are more economical, can have more aggregate speed, support inherently distributed applications, can give higher reliability, and can grow incrementally [Tanenbaum95]. However, writing effective distributed programs in this environment is a complicated proposition, as communication must be performed exclusively using message passing, which carries with it many difficult restrictions, implications, and tradeoffs. "Herein lies the dilemma. Multicomputers are easier to build but harder to program. Multiprocessors are the opposite: harder to build but easier to program. What we need are systems that are both easy to build and easy to program" [Tanenbaum95]. Distributed Shared Memory is an attempted solution to this problem.

2.1.2 Implementation

Distributed Shared Memory (DSM) is a mechanism for providing the semantics of multiprocessor shared memory in a distributed (loosely coupled) environment. The "designers of the early DSM systems were inspired by the principles of virtual memory

(VM), as well as by cache-coherence maintenance in shared-memory multiprocessors" [Protić96]. The Basic DSM systems use the virtual-memory hardware of modern processors not only to implement the "paging" of virtual pages over the network, but also to enforce coherency semantics by trapping and coordinating access to the data inside the pages. IVY [Li89] was the first such system.

In this way, a shared-nothing environment can provide the exact semantics of an SMP (Symmetric Multi-Processor) parallel environment, and, conceptually, all of the same programs can run in the same way, and all of the familiar parallel/shared-memory interprocess-communication and synchronization techniques can be used too. The system becomes both easy to build and easy to program. However, there is a fundamental problem with this approach, which will be discussed later.

The conceptual implementation of a basic DSM system is relatively straightforward. Such systems use an *invalidation* protocol. The VM hardware must provide three types of access permissions to the VM pages: **read-write**, **read-only**, and **no-access** (i.e., invalid). When a read or write operation is attempted on an address in a page for which a process does not have at least the required permissions, a page fault occurs and the kernel of the operating system of the machine is entered to service the fault. The VM hardware will typically use only fixed-size pages, and the page size will have an impact on the performance of DSM, since the contents of entire pages will be passed in messages to share data.

There are a number of possible organizations of management software for the invalidation protocol [Li89]. Most use replication and migration to offer good caching potential. The machine that has most recently modified a VM page is said to be the *owner* of that page. It must always be the case that either the owner machine has **read-write** (exclusive) access and all other machines have **no-access** to the page, or that the owner has **read-only** (shared) access and any number of other machines also have **read-only** access (i.e., shared and exclusive access to pages must conflict in the standard way). The owner keeps a list of all readers of a page. Accessing operations

to memory cells allowed by the current permissions of a page on a machine are carried out directly by the usual processor mechanism at full memory bandwidth.

When a machine that has **no-access** wants to have **read-only** access to a page (because a page fault occurred on a read access to an invalid page), it sends a message to the *owner*, and the owner changes its own access to **read-only** (if it previously had **read-write** access) and replies with a copy of the current page contents. (The system must include a mechanism to find the owner.) The requesting machine takes the page contents and maps the page into the address space of the faulting process, with **read-only** access, and the process then re-executes the read operation.

When a machine that has either **read-only** access or **no-access** wants to perform a write operation to a page, it must transmit a request to the owner, and the owner will transmit a message to all machines that have **read-only** access to the page telling them to invalidate the page from their memories (i.e., change their access to **no-access**). After each machine invalidates, it replies back to the owner that it has done so. After collecting all of the invalidation replies, the owner invalidates itself and transfers the current page contents and the ownership of the page to the requesting machine. The requesting machine accepts and maps in the page contents with **read-write** access and re-executes the write operation. For an example of page-access transitions, see Figure 2.1.

Many systems such as Dash [Lenoski92] and Memnet [Delp89] implement basically the standard DSM invalidation scheme, but do it in hardware using very small page sizes (16 or 32 bytes) and special high-bandwidth interconnection networks, for high performance. Hardware implementations are, of course, more specialized and expensive.

Alternatively, an *update* protocol could be used. In a pure update scheme, all machines always have **read-only** access to a page and a page fault occurs on every single write operation. The faulting machine transmits an update message to all other machines sharing that page, and then continues. Additional protocol complexities

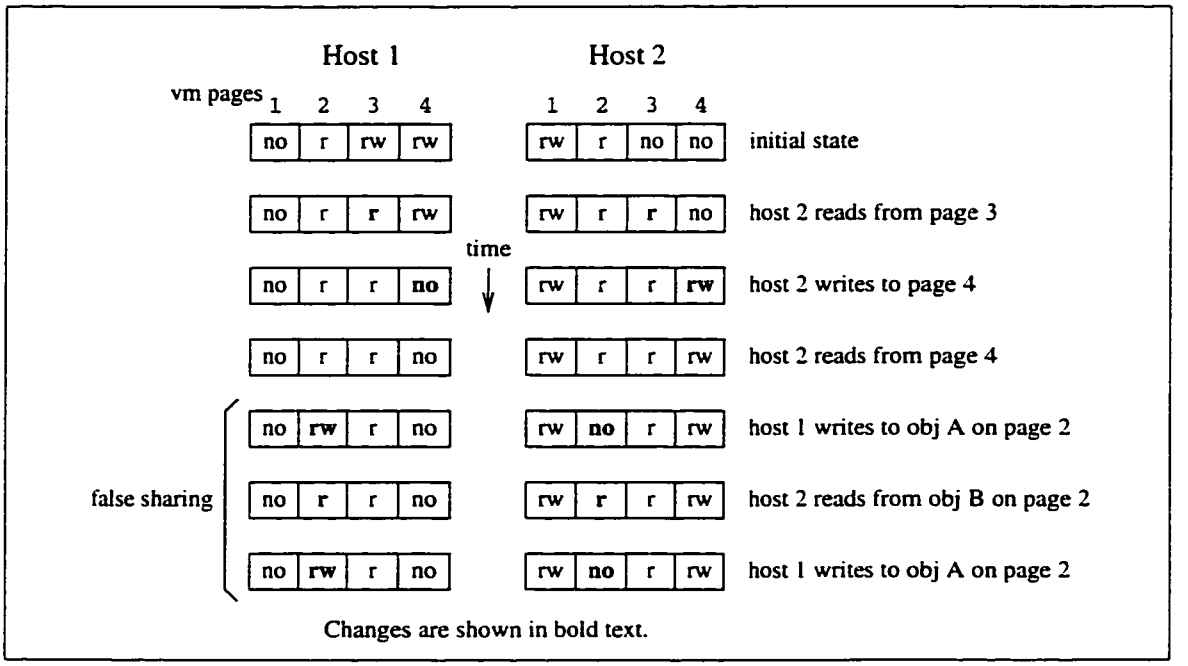


Figure 2.1: DSM VM-Page-Access Transitions

must be implemented to ensure that all updates are consistently ordered among the machines. Possibilities for consistency include using a fixed coordinator, using a *two-phase commit* protocol or sending updates around a *virtual ring* of machines [Tanenbaum95].

This update scheme almost certainly requires special hardware, or the system overhead would be horrendous unless write operations were extremely rare [Wilson93]. One hardware approach would be to have a bus "snooper" that watches for writes to the shared memory pages and generates and sends messages over a high-speed, low-latency network automatically. A special-hardware-based system designed to use a hybrid of the invalidation and update schemes (where choosing between updating and invalidating is made based on accessing statistics) is discussed in [Wilson93]. Some multiprocessors implement a similar scheme as a "write-through cache".

Note that there are many other variations on protocols and architectures for im-

plementing DSM. Some avenues for performance improvement follow [Protić96]. You can use a cost-competitive algorithm that replicates only when the number of accesses to a page exceeds the replication cost, and which simply migrates data otherwise. You can eliminate unnecessary page transfers by keeping a version number and then not transferring data for a reader host who wants access but who coincidentally has the most up-to-date version of the data. An obvious thing to do is to only use the DSM mechanism on pages that actually need to be shared (not code, local variables, or thread-stack space). For keeping track of the owner of a page, the identity of the probable owner can be kept at each site and requests can be forwarded until the real owner is found. (This is often called "Li's Algorithm".) Here, "the performance does not deteriorate as more processors are added to the system, but rather degrades logarithmically as more processors contend for the same page" [Protić96].

Two additional examples are the Mirage and Clouds systems. The Mirage system [Fleisch89] has a dynamically adjustable time-window during which it will inhibit the transferring of ownership of a page, to help prevent thrashing. The Clouds system [Chen91] has a "weak-read" page attribute that allows hosts to read a page with no guarantee about whether someone else will modify the page at the same time, which could lead to inconsistencies (which is perhaps useful for converging algorithms).

2.1.3 Evaluation

These Basic-DSM protocols implement *sequential consistency* (discussed below), which means that existing multiprocessor programs can run correctly in the new environment without modifying any code. This was an objective of the early DSM systems, supporting the "dusty deck" problem. One minor note is that although *Test-And-Set* instructions of multiprocessors can be used for locking in this environment, it may be best to couple this with an actual distributed locking mechanism, since the tight polling loops on the shared memory from the TAS instructions will cause

thrashing if there is contention.

The major computational benefit that software DSM provides is the caching of the partial results of a distributed computation. A process does not have to explicitly request the most recent copy of shared data; it just accesses the memory addresses where the data resides and is guaranteed to receive the most recent data. The major benefit that an invalidation approach has is that pages with very low access contention are shared very efficiently. If a processor were to perform a write operation followed by a "burst" of read and write accesses to a page and then cease to access the page for a while, and if no other processors attempt to access a page during the "burst" of activity, then, after the initial access, accesses to the page will be just as fast as accessing local memory.

The major concern is whether the benefit of the caching outweighs the message-passing overhead caused by the contention in typical parallel applications, since DSM may be expected to perform well with the fine grain of sharing possible with a multiprocessor system. According to an extensive simulation study [Wilson93], the answer is a resounding **no**—the message passing is very bulky indeed. In fact, the requirements far exceed that which is required of the bus of a multiprocessor system. The results do, of course, depend on the page size of the VM hardware: a larger page size will result in a higher probability of access contention to a page, and will require the transport of a greater amount of data to send "fresh" copies of pages to and fro. The update approach was found to require much less data to be passed: an amount similar to that required of a multiprocessor bus. However, the network acts as a processor bus in the update scheme, a job for which it is quite ill-suited.

The other major problem to contend with is communication latency. In a naive implementation, the communication-latency overhead would be absolutely terrible. For every conflicting read or write operation, broadcasts or multiple sequential message passes would have to be performed, and processing could not continue until all of the processes involved became synchronized. Weaker-consistency systems (discussed

below) provide a means of hiding communication latency.

A major flaw of the distributed-shared-memory IPC model is with the unit of granularity of operation consistency—the single memory reference (read or write). This is much too small. It is smaller than what an application is likely to need, so DSM attacks a much more restrictive problem than it needs to. A much more appropriate level of granularity of operation consistency for DSM would be at the level where a *message* would need to be passed in a message-passing scheme.

Also, the unit of data granularity (a VM page, often 4K or 8K in size) may be much larger than necessary, causing more network load than would really be necessary. There is a tradeoff between page size and efficiency. With large pages, it is more efficient to pass large quantities of data around since there is more page data to amortize the network-operation overhead, but it also becomes more likely that different processors will want to contend for the same page at the same time, which will cause large amounts of data to be transferred. With software DSM implementations, there is no choice but to use relatively large page sizes. One pathological situation that can arise is called *false sharing*. Since the variables of a program are allocated in no strategic order in the shared-memory space, it is possible that two unrelated variables can end up being allocated to the same page. Then, if two processes were each to access a different one of the variables heavily, the page would thrash back and forth between the two processors, even though the two processes don't logically conflict at all. This false-sharing pathology can also occur with disjoint accesses to interleaved elements of an matrix on the same VM page.

As an example of a system that tries to use a very fine data granularity, the Blizzard system [Falsafi94] allows for OS support of fine-grain objects using the error-check bits of memory to cause kernel traps, or by inserting fast state checks in software before each shared-memory access. Oddly, however, the software-checking approach seems to give the better results in this system [Falsafi94].

2.2 Memory-Consistency Models

"The memory-consistency model [of a system] defines the legal ordering of memory references issued by a processor, as observed by other processors" [Protić96]. Looser and more relaxed models allow greater performance at the expense of higher programmer involvement in synchronizing shared-data accesses. The stronger consistency models are *strict*, *sequential* and *causal*, *processor*, and *pipelined-RAM* consistency and the more relaxed models are *weak*, *release*, *lazy-release*, and *entry* consistency. The more relaxed models distinguish between ordinary and synchronizing accesses in order to provide better performance, whereas the stronger models inherently synchronize on all accesses.

Strict consistency is what a uni-processor system provides. It is defined by "any read to a memory location x returns the value stored by the most recent write operation to x" [Tanenbaum95]. This is only of theoretical interest in parallel programming, as "most recent" refers to a notion of (effective) absolute global time, which is highly impractical to provide in a distributed environment. Besides, the difference between strict and sequential consistency can only be seen in race conditions anyway, which should not be present in properly written parallel programs in the first place.

Sequential consistency is defined by the condition: "the result of any execution must be the same as if all of the operations of all processors were executed in some [global] sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [Lamport79]. In other words, any valid global interleaving of operations is fine, but all processors must "see" this same global interleaving. It must be as if there were a global memory server that took read and write requests from a queue filled by the processes, and carried them out *in sequence*.

This is a slightly weaker model than strict consistency, but it is all that parallel

programmers expect, since they are taught to guard any potential race conditions with synchronization primitives. This is the (effective) model that general-purpose multiprocessing systems should provide.

Causal [Hutto90], **processor** [Goodman89], and **pipelined-RAM** [Lipton88] consistency are all models that relax the global-interleaving constraint of sequential consistency. Causal consistency requires only that "writes that are potentially causally related must be seen by all processes in the same [global] order" [Tanenbaum95]. (Causality has to do with the "propagation of information:" if a process reads one value and then writes another, the written value is potentially causally related to the read value.) Pipelined-RAM consistency drops the requirement from causal consistency that causally related writes be seen by all processes in the same order. It requires only that all writes performed by a single process be seen by all external processes in the order in which they were issued. Processor consistency adds the constraint that there be a consistent global order of all write operations to a single variable. Some specially constructed programs may be able to make use of these consistency models, but programs generally require sequential-consistency semantics. Therefore, these models are of relatively little importance.

Weak consistency [Dubois86] requires that memory be (sequentially) consistent only on synchronizing accesses, and it can be arbitrarily inconsistent between synchronizing accesses. A synchronizing access must wait for all previous accesses to execute, while ordinary reads and writes must wait for the completion of previous synchronizing accesses. In other words, a synchronization propagates outward all changes made locally and brings in all synchronized changes made remotely. The system serializes the synchronizations.

This type of mechanism removes the unnecessarily and expensively fine operational granularity from DSM systems. A synchronization is performed in the places where a *critical section* is entered or exited in a parallel program (since it guarantees that you are reading the most up-to-date data when you start, and that your writes will be

propagated for everyone else to see when you finish). Since remote processes should not be accessing shared variables while one process is in a critical section. it doesn't matter if inconsistent results are visible to the outside while you are in a critical section.

One thing to note is that this model can work in cooperation with the page migration/replication scheme of classic DSM systems to limit the amount of work that needs to be done on synchronizations. Invalid pages do not need to be synchronized: only pages that you have written to and that other processors in the system have access to need to have information propagated for them (and vice-versa for bringing information in). By making the grain of operational consistency more coarse, it is more practical to use a data-updating approach with DSM pages. rather than strictly an invalidation approach.

Release consistency [Gharachorloo90] extends weak consistency to have two synchronizing accesses, acquire and release, and these are intended to be called when entering and exiting a critical section, respectively. This is an improvement over weak consistency since the mechanism there does not know whether a synchronization is for entering or exiting a critical section, and so must do the work for both.

The ordinary accesses that execute between acquire/release pairs are "protected" and only one process may be in a critical section at a time, and there may be any number of locks that protect the consistency of only specific shared variables. The interleaving of the acquire and release operations must only fulfill the requirements of processor consistency. In effect, the acquire/release pairs make the bursts of accesses inside critical sections into atomic actions. An illustration of page contents for DSM with release consistency is shown in Figure 2.2. Note that VM page 2 in the figure is shared by both objects A and B.

It is also possible to use barriers for memory synchronization. "A *barrier* is a synchronization mechanism that prevents any process from starting phase $n + 1$ of a program until all processes have finished phase n" [Tanenbaum95]. When a process

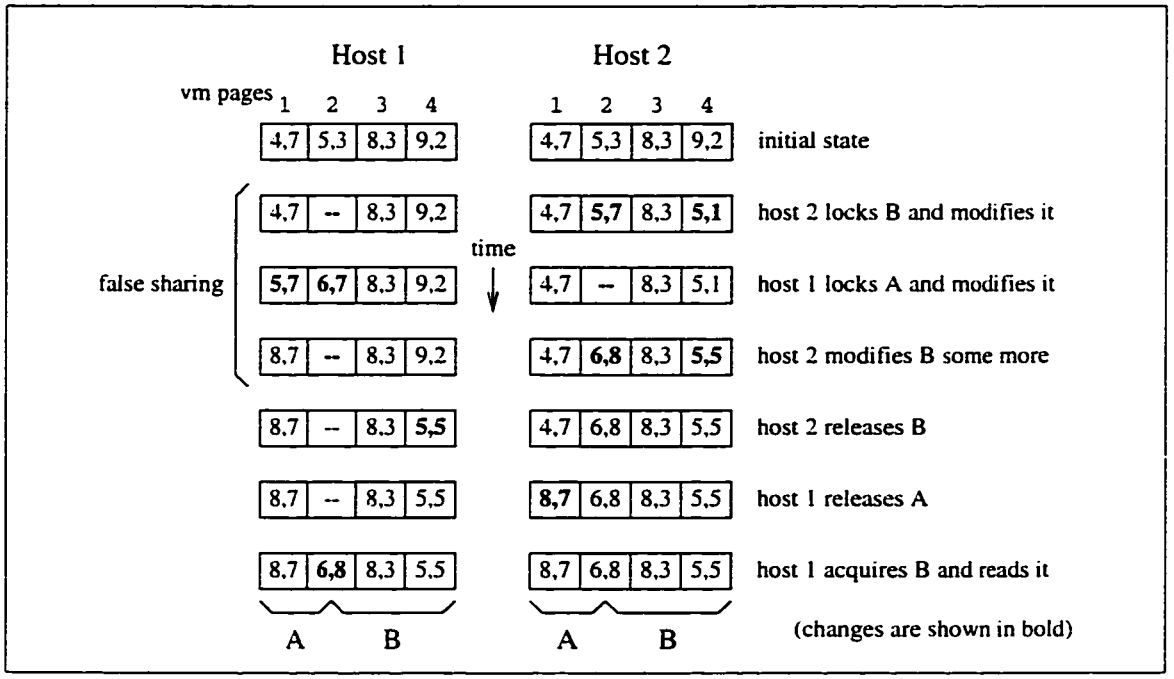


Figure 2.2: DSM with Release Consistency

arrives at a barrier, it should perform a release and when it departs from one, it should perform an acquire.

Lazy release consistency [Keleher92] enhances release consistency in that instead of propagating changes to the shared address space on each release, it waits until the next relevant acquire, and only the modifications associated with the specific lock of the acquire are propagated to the processor doing the acquire. This can potentially save a large amount of work in a program that has a critical section inside a loop. With no outside contention, no data needs to be propagated for the repeated acquire-release pairs.

Entry consistency [Bershad93] also improves release consistency. Each shared data object is associated with a synchronizing variable, such as a lock or a barrier, using language-level notation. (There may be many objects associated with a single synchronization variable.) Then, to access ordinary variables, a program must acquire

a lock on the associated synchronization variable to bring the contents up-to-date. Only a subset of the data needs to propagate at this time, as only a subset of the shared variables will be protected by the synchronizing variable. Locks are released to propagate modifications.

Associating shared variables explicitly with synchronization variables also allows multiple disjoint critical sections to be entered at the same time, increasing parallelism. These locking variables can also be acquired in a non-exclusive mode to allow concurrent read-only access to shared variables. The protocol to control a lock can be an ownership-forwarding invalidation protocol, similar to the classic-DSM mechanism.

2.3 Selected DSM Systems

2.3.1 TreadMarks

Very briefly, the TreadMarks [Amza96] system is a relatively standard lazy-release-consistency DSM system. One special feature about it is that it allows multiple hosts to have write access to the same page at the same time (i.e., it implements a *multiple-writer* control protocol); using the lazy-release-consistency model allows this.

When a process first performs a write to a page, the system makes a twin copy of the page and lets the process proceed to write to the original. Then, when the process performs a release operation to propagate the updates, the system performs a "diff" between the page and the twin and sends along only the changes, in order to cut down on the network traffic. On the receiving end, the changes are merged in with any concurrent changes made by the receiver. An illustration of page contents with TreadMarks with multi-writer lazy-release consistency is shown in Figure 2.3.

The system does not allow user-defined objects but instead generates and manages "diff" records, which are processor-intensive operations. There is also no locking

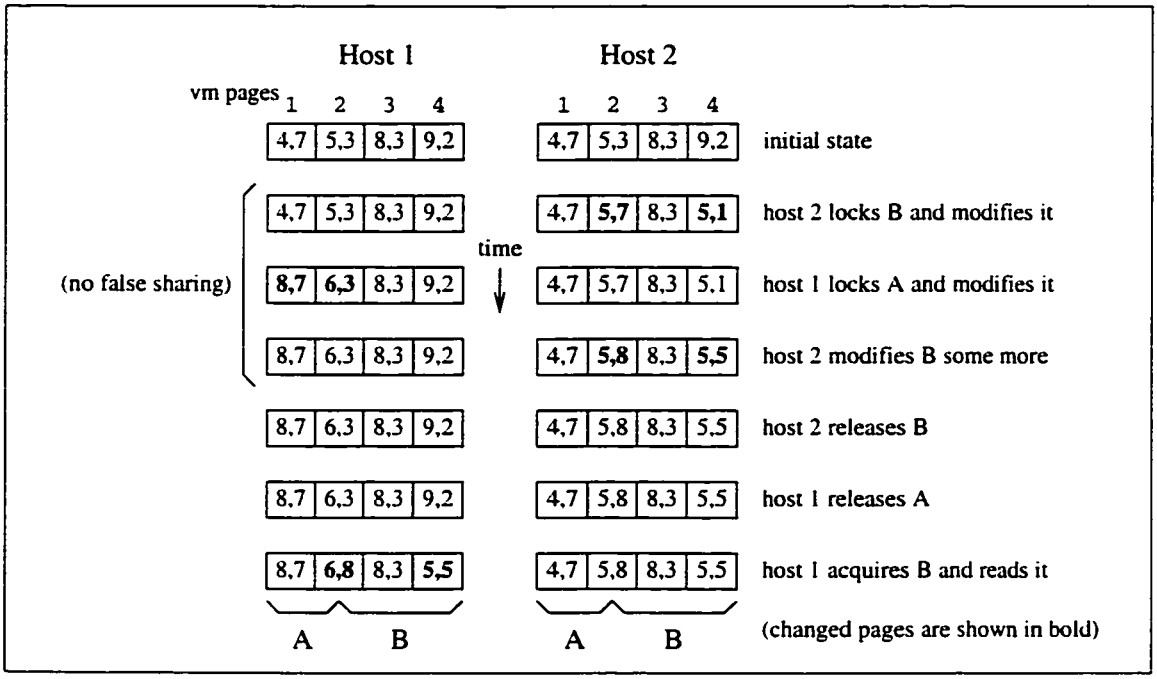


Figure 2.3: TreadMarks Multi-Writer Lazy-Release Consistency

protocol integrated with the data-propagation protocol, so multiple rounds of messages may be needed to modify an arbitrary user object. There is also no distinction between shared and exclusive locks, which limits concurrency.

2.3.2 CVM

CVM [Keleher96a] stands for "Coherent Virtual Machine". CVM is a DSM system that currently supports three protocols: Lazy multi-writer (LMW) which is Lazy Release Consistency that allows multiple simultaneous writers to a page, where the system uses "diffs" to work out the consistency; Lazy single-writer (LSW), where only one writer can modify a page at a given time; and Sequential consistency (SEQ). The latter two protocols have the usual classic-DSM problems, including false sharing and a large message granularity (a page). CVM was strongly influenced by TreadMarks.

CVM has two distinguishing features: it is implemented entirely as a user-level

library for Unix systems (the same as TreadMarks), and CVM is available for free, source code included. It is written in C++ and network communication is provided by protocols built on top of UDP. The virtual-memory functionality is made to work without modifying the Unix kernel by using "mapped file region" or "anonymous memory region" features supported by some versions of Unix. This Unix mechanism provides user processes with functions to change the protections of virtual-memory pages in the special memory regions and callbacks for handling page faults.

Given that CVM is freely available for use by the public, many details are given about how to use it. It appears, at this writing, that CVM is still very much a system under construction, and that one of its main purposes is to provide a platform for testing new DSM consistency protocols, rather than to be a production computation platform. Also, its operation is a bit restricted.

To use CVM, you call cvm_startup() to initialize the system and then you call cvm_alloc(int bytes) for each shared object that you want to allocate. Allocation is static, so you have to allocate everything in your program's initialization phase. After the shared variables are initialized, you call cvm_create_procs() to fork off a number of worker processes that start with the identical memory image to the initial process. These workers are forked off to N different machines, where the number of machines to use is given on the command line and the addresses of the machines is in a configuration file. After that, all workers have regular access to the shared memory, subject to the consistency protocol that is selected on the command line. For synchronization, CVM provides cvm_lock(int id), cvm_unlock(int id), and cvm_barrier(void).

One paper [Keleher96b] compares multi- and single-writer protocols in CVM. In test cases where there is heavy write sharing to pages, the multi-writer protocol ran between 6 and 61% faster, and in cases where there was little write sharing, it performed an average of 3% slower. The multi-writer protocol also requires significantly more memory overhead (72%) than single-writer (10%). The cost of creating a "diff" (in the multi-writer protocol) is substantial, approximately 3/4 the cost of an RPC.

Since the speed of CPUs is increasing at a faster rate than that of memory, "diffing" is likely to become relatively more expensive in the future.

An interesting additional point is that using a $2\mu s$ delay after the last write before being allowed to give up ownership of a page hid the false-sharing problem in most cases, since writes will often be highly concentrated inside a tight loop. A factor of four or five was the performance improvement for some applications.

2.3.3 Munin

Munin [Bennett90] is a DSM system that is based on release consistency with the usual locks and barriers, except that Munin is a slightly different kind of DSM system from the ones discussed so far. It is more fundamentally based on *shared variables* than on shared pages. Using a language-level notation, you must tell which variables of your program are shared, and you can also select a control protocol for each variable from the list: Read-only, Migratory, Write-shared, or Conventional (defined below). The default is Conventional. There used to be several other protocols, but they were found to be only marginally useful.

The compiler puts each shared variable on a separate page, by default, although large variables may occupy several pages. This avoids the false-sharing problem and allows the system to use the standard VM hardware to detect and control accesses to the shared variables.

Read-only variables are never changed after being initialized, so their control protocol is very easy; if a process gets a page fault upon trying to read one, a request is sent to the owner of the page and the content is brought back and mapped in. Migratory shared variables use the standard acquire/release protocol, except that the pages they are on are always migrated and never replicated. This saves the bother of sending updates and/or invalidations. Write-shared variables can have multiple concurrent writers, provided the write operations do not overlap, and the protocol

handling multiple writers is much like the one in TreadMarks. Conventional shared variables are replicated and migrated like classic-DSM pages.

While this system supports user-defined objects, it does not support per-object locking. Instead, the entire memory needs to be synchronized on each lock operation to give sequential consistency, and therefore, the main effect of the user-defined objects is to eliminate false sharing, rather than to limit the amount of data transfer per lock to being only what is necessary.

2.3.4 Midway

The Midway system [Bershad91] is similar to the Munin system, but uses Entry consistency instead of Release consistency. Entry consistency requires that each shared variable be associated with a synchronization variable, either a lock or a barrier. Because of this, whenever a critical section is entered or exited, the system knows exactly which shared variables are affected. Locks can be exclusive or shared (allowing concurrency).

When a lock is acquired, the local machine goes out and gets all of the updates for the shared variables of the lock. Munin implements an owner-based protocol for locks (Li's Algorithm), and since variables are associated with locks, all that needs to be done to bring a processor up to date on its data is that in addition to the lock ownership, the current owner also sends the current contents of all of the associated variables with the message granting ownership of the lock. When a lock is released, no action is required. If there is no contention for a lock, the ownership of the lock will remain with the current host, and if the current host acquires the lock again, no work needs to be done.

To cut down on the amount of shared-variable data sent with a lock grant, only those shared variables whose values have changed since the last time the current host was owner need to be sent. The system keeps track of this using either VM-hardware

or special runtime information to know which variables have been written to and Lamport clocks [Lamport78] to keep track of when and what changes are needed.

The Midway system has a great deal of potential to deliver very good performance compared to other DSM systems because of the way its entry-consistency mechanism works. However, entry consistency requires more information from a programmer than the other systems do, and giving this information makes programming more complex and error-prone than in other DSM systems.

2.4 Shared Data-Object Systems

A different approach to the shared-data problem is a high-level structured-data approach. In this approach, the data items are user-defined and are accessed through language-level mechanisms. This allows the user to directly tune the data and operational granularities for efficient processing. Language-level mechanisms can also make programming more automated and less error-prone and provide more opportunities for runtime optimizations.

2.4.1 Linda

Linda [Carriero86] is a *coordination language* that can be added to existing programming languages. It provides a distributed content-addressable shared memory called the *tuple space* for all interprocess communication. The tuple space is an unordered bag of data to which processes are given *associative access*. "This property, together with the temporal and spatial decoupling inherent in the abstraction, makes it especially easy for use by application programmers" [Bakken93]. The tuple space is managed by the Linda runtime system and the abstraction can be implemented on a number of different architectures, including shared-memory multiprocessors, hypercubes, and networks of workstations, and hide all of the architectural details.

A tuple has one or more fields of the various host-language data types, where the first field is almost always used as a logical name of the tuple. The out primitive operation puts a tuple into the tuple space, for example: out("student", "Craig", 29, TRUE). The in primitive retrieves and deletes a tuple from the tuple space that matches a given pattern of parameter (field) values. Actual and formal parameters are used with the in operation to specify the formal pattern of the required tuple, the values that are required to be in some of the attributes (optional), and which attribute values are to be be assigned to variables. For example, consider in("student", char *name, int age, TRUE). If the specified tuple is not currently in the tuple space, then the process will block until one is deposited. If there is more than one tuple matching the pattern, then one is chosen arbitrarily. A similar read primitive is also available that will not remove the tuple from the tuple space. Tuples are immutable, so there is no operation to modify a tuple in-place. The eval primitive is used for process creation. The process is given arguments, executes to completion, and deposits a tuple into the tuple space when it exits.

Temporal decoupling is achieved because two communicating processes do not need to have overlapping lifetimes; the tuple space exists for the entire run of an application. *Spatial decoupling* is achieved because a process does not need to know the identity of other processes that it communicates with; communication is anonymous. These properties lessen restrictions on user applications and allow for the easier implementation of parallelism.

Linda is used for working with *distributed data structures*, which are "data structure[s] that can be manipulated simultaneously by multiple processors" [Duggan91]. The tuple space is used for storing the distributed data structures, which must be split up into tuples that will require a sufficiently coarse grain of sharing to offset the overhead of distribution. Linda avoids consistency problems by requiring that a tuple must be removed from the tuple space before being modified.

To minimize the amount of searching that must be done to provide associative

accessing, most implementations use buckets and hashing. Tuples are split into disjoint buckets (subspaces) by grouping tuples with an identical structure (number of fields and type of each) and by their first field (name). Often, the name is a constant string, so the bucket that a tuple belongs in is known at compile time. Within buckets, tuples can be hashed on an arbitrary but hopefully well chosen field (other than the first one) within the subspace, for faster accessing.

There are a number of options for a distributed implementation of the tuple space. Some implementations replicate the entire tuple space on all processors, some hash tuples/buckets to specific processors, and some store tuples at the processor that performed the out operation. The full-replication approach requires broadcasts for ins and an equivalent to a two-phase commit protocol for consistent deletion. If outs are stored locally, then a broadcast is required if an in cannot be satisfied locally. Non-replicated distribution requires only point-to-point messages for in and out. "Partial-replication" schemes are also possible [Krishnaswamy91]. In any case, the Linda mechanism is indirect and fairly expensive. Representations of objects such as matrices can also be awkward and inefficient to work with.

2.4.2 Orca

Orca [Bal91] is an object-based programming language and runtime system that supports distributed programming using processes and passive objects. These objects are very similar to the objects of object-oriented programming languages, with private data, methods, and implementation hiding, except that Orca does not support inheritance, and the objects provide the only means of sharing data between processes.

Since the objects are passive, a process enters the object through its interface and executes the code of the called method. Each method is a list of *guard*/block-of-statement pairs, where a guard is a boolean expression. When a method is called, its list of guards is checked for any that evaluate to true and if none of them do, the

calling process blocks until a guard becomes true (by the action of some other thread using the object). If more than one guard is true, then one is arbitrarily chosen and entered. The concept very similar to a monitor.

The runtime system provides the necessary locking to ensure that all of the methods are executed atomically in a globally sequential order, giving the shared data sequential consistency. Object data can either be replicated or single-copy, where the runtime system chooses which is best based initially upon static-analysis information from the code. The single-copy objects are quite straightforward to maintain. When one is invoked, either the object data can be migrated to the local machine or an RPC can be sent to the remote machine to invoke the method there and return the results.

For replicated objects, read-only methods (the compiler notes these) can be performed completely locally. Methods that write are more expensive. If reliable totally ordered broadcasting is available, the caller can simply pass the operation and arguments to all hosts holding a replica for them to perform the operation locally. (This seems to trade off latency for parallelism, which may or may not be a good thing.) For a system that does not have broadcasting, a two-phase protocol can be used, or a sequencer, or the system can decide more favorably to maintain only a single copy of the object. Other optimizations are possible for replicated objects, too.

One interesting feature of the language is that "Orca does not provide pointers for building graphs; instead, graphs are built-in data types, just like matrices and sets. This eliminates the problem ... of passing complex data structures around in a distributed system" [Bal89].

The Orca mechanism does implement user-defined objects and per-object locking. Its replica-management protocol involves sending updates to all hosts for each object-write operation. It depends on the tuned totally-ordered broadcast capabilities of the Amoeba operating system to give good performance.

2.4.3 ABC++

ABC++ [O'Farrell94] is a different system for object-oriented programming in an architecture-independent distributed environment. The "ABC" stands for "Active Base Class", meaning that the objects are "active objects," contrasting with the passive objects of the Orca system. Each concurrent ABC++ object has its own thread of control, and this concurrency is inherited from the ABC root class, so the user has explicit control over which objects are concurrent/distributed.

Active objects are created at runtime by a special call. The user specifies which processor an object should execute on, and the object cannot be migrated thereafter. Since objects are created dynamically, a special pointer type is provided to reference the objects. After creation, methods on the object are invoked in a straightforward client/server remote-procedure-call fashion. However, there is a special extension called *future variables* that allows asynchronous remote execution. These variables are used to receive the return value of a method invocation (by assignment), and the local host will block only in the case that the future variable is referenced before a return value is received by the local host from the object thread. ABC++ also provides facilities for selective method invocation, based on dynamically changing the set of methods which will be accepted.

In addition to active objects, ABC++ also provides a facility called *parametric shared regions* (PSRs) for distributed data sharing. A PSR can contain a C++ object, an array, or a simple type. Since PSRs are user-defined, they introduce no false-sharing problems. These PSRs are accessed using *handles*, which give permissions for read-write or read-only access. Constructors and destructors for the handles control the lock management in the system, so locking is performed automatically by declaration of handle variables and the scoping rules of C++. Handles can also be used as pointers in dynamic distributed data structures in a way that a lock is acquired when a handle (pointer) is copied to a search-control pointer and is usually

released by exiting the block.

Handles can be of Locked or Unlocked type. Unlocked handles are used for locating PSRs and would most likely be found in dynamic data structures, as they are used as static pointers. Locked handles actually lock an object. The assignment of an Unlocked handle to a Locked-handle variable causes the PSR to become locked. Nested locks are also propagated by passing locked handles as parameters. The automation of lock handling removes the likelihood for programming errors from lock handling, although it can also introduce awkward constructions in some cases to control the locking in the desired fashion.

The object-management protocol in ABC++ assigns object statically to hosts, which means that any methods invoked on these objects remotely will involve at least an RPC operation.

2.4.4 Problems with the Object-Oriented Approach

There are problems with using the object-oriented approach in a distributed system [Waldo94]. Hiding the distinction between local and distributed objects is a mistake. There are four separate problems: latency, using memory pointers, partial failure, and concurrency.

The latency problem is obvious; distributed objects will operate with much more latency than local objects, since a distributed algorithm must access remote objects and it is likely that messages will need to be passed. The object-oriented paradigm encourages people to design distributed programs in the wrong way. Basically, programmers are encouraged to write programs for a sequential environment, and then distributed object-management systems are brought in to execute the program, but all of the wrong assumptions about the relative costs of operations and the grain of sharing, etc., give the program sub-optimal performance. Distribution should be designed into programs from the start.

Using memory pointers is a natural way to program, but it leads to complications with distributed objects, and the language needs to use some sort of "abstract pointer" that works between machines. If distributed and local objects are not differentiated, then the overhead of these abstract pointers will need to be endured for every access.

Partial failure is a new problem with distributed systems, and there probably is no good automatic, general way to handle it. Using an object-oriented paradigm, you either program as if partial failure cannot happen, or you program all of your objects to handle partial failures. Concurrency, synchronization, and indeterminacy issues are also introduced in such systems.

2.4.5 Shared Regions

The Shared Regions system [Sandhu95], [Sandhu93] is a software approach for maintaining processor-cache consistency in multiprocessor systems. It explicitly relates synchronization tokens with the data they protect. A region is a user-declared set of memory locations (region). The declaration is performed with the call Region R = bindregion(addr, size), where addr and size define the memory region. A more complex region can be defined using the parameters (addr, size, stepsize, nsteps), to allow an interleave between bands of included memory cells.

After regions are defined, their synchronization and coherence are maintained by explicitly bracketed locking using the primitives readaccess(R), readdone(R), writeaccess(R), and writedone(R). The accessing is straightforward, with any read accessing to a shared region needing to be bracketed by the readaccess/readdone primitives and write accesses being bracketed by writeaccess/writedone. The main issue with this approach is requiring a programmer to correctly bracket each shared-memory access. The ABC++ system above provides a more automatic mechanism for dealing with this issue.

The shared-region approach is similar to the requirements for this thesis. However,

the shared-region system is designed and implemented for processor cache coherence. It also does not address higher-level issues, but only basic object locking and synchronization.

2.5 Message Passing

The alternative to shared-memory interprocess communication is message passing. For a distributed system, the passing of packets of data is a physical means of interprocessor communication; all other mechanisms must be implemented on top of this. For computation, message passing is an explicit and easy-to-understand model for interprocess communication.

There are a few basic message-passing models: *asynchronous*, *synchronous*, and *highly synchronous* [Tanenbaum92]. Asynchronous message passing involves buffering so that the sender can continue executing immediately after the send operation is complete, and so that a receiver does not have to be blocked on a receive primitive when a message arrives. This is the most general form of message passing and offers the most potential for concurrency. Synchronous message passing causes the sender to block until the message that is sent has been received by the destination process. Highly synchronous message passing causes the sender to block until the receiver has picked up the sent message and has returned a reply.

Unix and other operating systems typically provide asynchronous unreliable message passing with UDP (User-Datagram Protocol) and "mostly asynchronous" reliable stream-oriented message passing with TCP (Transmission-Control Protocol). TCP is "mostly asynchronous" because it will block the sender if necessary to prevent the buffers "between" the sender and the receiver from overflowing.

2.5.1 Remote Procedure Call

The *remote procedure call* (RPC) is perhaps the most common scheme for distributed IPC. Its semantics are simple, convenient and familiar, following the semantics of the local procedure (or subroutine) call. Programming languages and environments often hide the complications of executing operations on remote machines and make it look like a local procedure call is taking place. These complications include: argument marshaling and data copying, control transfer, RPC protocol processing, low-level message-passing protocols, and the handling of partial-failure modes.

Use of the RPC mechanism gives rise to specific process-structuring paradigms, the most popular of which is the *Client/Server* paradigm. In this scheme, one process, the *server* contains all of the procedures that are called by external programs, the *clients*. The remote procedures provided by the server are used as interface points to some particular service, such as file storage/access. Constructs known as *stubs* are used to hide the implementation details. These are subroutines that construct and send the call message and wait for the return message on the calling side, and on the remote machine accept the call message and dispatch it to a local procedure, construct and send back a reply message with return values from the receiving side. These operations are often programmed directly with Send(), Receive(), and Reply() primitives [Gentleman81], respectively, provided by highly-synchronous messaging systems. See Figure 2.4 for examples of messages passed for RPCs. Another option, *asynchronous* RPC allows the sender to continue processing after sending the request and to wait for the reply only when it absolutely needs the return value, allowing the client to do other useful work in the meantime [Ananda91] [Schaeffer93].

2.5.2 Multicasting and Broadcasting

Multicasting is the extension of message passing to send a single message to multiple receivers (one-to-many). Broadcasting is the case of sending a single message to all

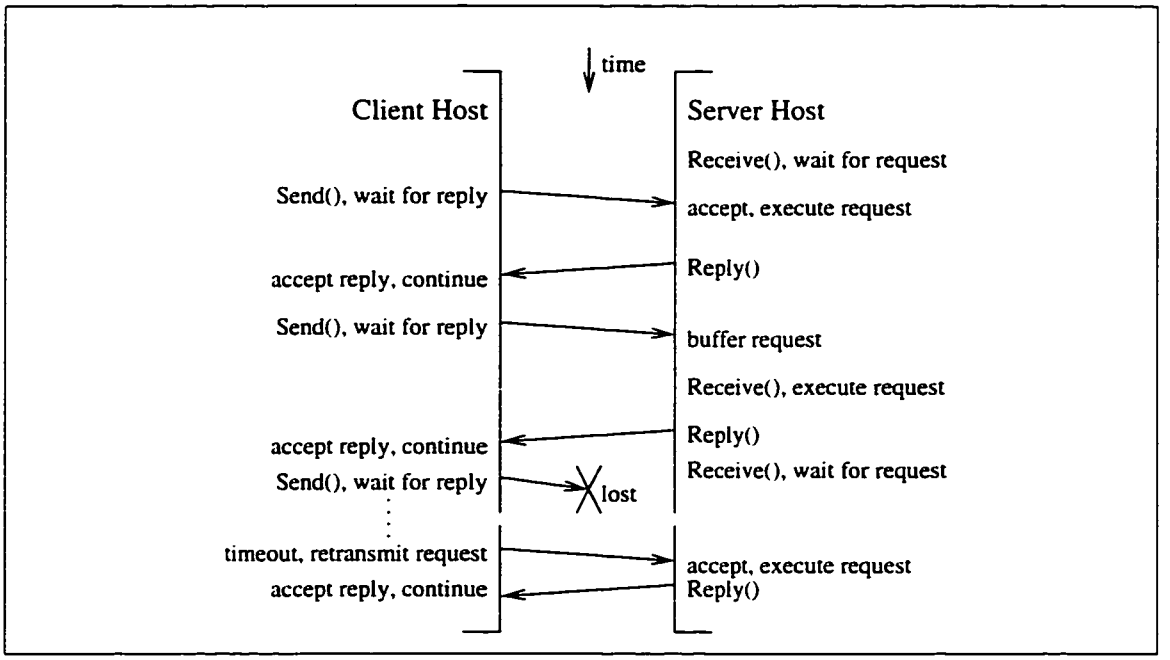


Figure 2.4: RPC Primitives and Messages

machines in a system (one-to-all). One of the main reasons for the introduction of Multicasting was the inherent physical broadcast facility of many types of networks. This means that in the best case, sending to N receivers can be accomplished by sending only one message over the network, rather than N independent messages that may need to be sent synchronously. Broadcasting and multicasting have become an integral element of many distributed algorithms and systems.

One important issue with multicasting and broadcasting is *atomicity*. Multicast (broadcast) message delivery is said to be *atomic* if the semantics of the transfer guarantee that a message will be received by either **all** of the members or **none** of the members of a group. This property is useful to a number of applications but it usually is not guaranteed by the physical message-passing medium, so it must be built on top at greater cost.

Another important issue is the guaranteed ordering of messages as received by all receivers w.r.t. other multicasted (broadcasted) messages. In *consistent-time or-*

dering. all messages are delivered to (seen by) all receivers in the exact same global order. which is not necessarily the order in which they were generated in absolute real time. This case is often also called *total ordering*. This is often (and probably best) implemented using a *sequencer* host that decides the total order of all messages and then multicasts them out. In *causal ordering*. messages are delivered so that they maintain correct causal relationships. Two events (messages) are said to be *causally related* if the occurrence of the first may possibly cause the generation of the second. If a message is received by a process, then all messages subsequently sent from that process are causally related to the one received. The causal relation is a partial order. Two events that are not causally related are said to be *concurrent*. The more restrictive orderings come at a significant cost. The option of providing no guaranteed ordering comes at no cost.

2.6 Conclusions

This chapter examines various existing methods from the literature of providing the shared-data programming paradigm in a shared-nothing distributed computing environment. The original proposal was Distributed Shared Memory (DSM), which allows existing "dusty deck" shared-memory (parallel) programs to run with virtually no modification in a distributed environment. It simulates the semantics of shared memory by using virtual-memory hardware techniques. There are two major problems with this technique. The first problem is that the memory is required to maintain its sequential consistency for each and every memory operation in the distributed system, which is too fine a grain. The second problem is that the smallest object that data coherency is maintained upon is the virtual-memory page, which is usually much too coarse a grain (typically 4–8K), and this can lead to false sharing.

To combat these problems, looser memory-consistency models were proposed. These models require only that memory consistency be brought up to date at certain

times, when processors in an application synchronize, so arbitrary thousands of operations may be applied without explicit synchronization while still maintaining memory consistency. They synchronize by explicit operations on the memory of acquiring and releasing all updates since the most recent synchronization operation. Lazy-release consistency ensures that no wasted effort is applied in bringing updates to processors that do not specifically request updates, and entry consistency allows maintenance of the consistency of only selected memory objects per operation.

Many systems introduce novel features for efficiency, in addition to looser consistency models, such as multi-writer sharing in TreadMarks (with mergeable "diff" records), and per-object coherency protocols in Munin. Munin also carefully allocates variables to separate virtual pages to help avoid false sharing. Programmer annotations help the system operate more efficiently.

An alternative to the virtual-page approach is the object approach. Objects are individually managed user-defined structures that are the unit of sharing in such systems. Different systems address different styles of objects. The Linda system manages objects in the form of tuples in a content-addressable shared tuple space. Interprocess synchronization is implicit in the tuple-space operations.

Other object systems include Orca, ABC++, and Shared Regions. Orca uses an object-based approach of managing passive objects. (It not quite "object-oriented" because of the lack of inheritance.) When a method of a distributed object is invoked, a special mechanism blocks out all other processes from invoking any other methods on that same object until the called method completes. Orca uses a replicated-data protocol for maintaining consistency. ABC++ uses the concept of "active objects" in which objects act like little servers and it implements its system entirely at the user-library level. An RPC environment is used in which objects are managed at fixed hosts, and features "future" variables which allow asynchronous RPC. ABC++ also supports parametric shared regions which can be used to store arrays and dynamic data structures. Shared Regions is a system that provides consistency for raw blocks

of memory in a multiprocessor environment. The coherency is maintained by explicit locking and unlocking.

We also look at message-passing schemes for contrast. RPC is the most basic and popular interprocess-communication paradigm, and provides point-to-point communication between a server and clients, by emulating the operation of an ordinary program-procedure call. Multicasting and broadcasting allow group communication with specific reliability, atomicity, and ordering guarantees. In general, stringent guarantees are very expensive to offer.

Based on an examination of these previous systems, we can determine the attributes a novel DSD system should have. It should provide the benefits of weaker-consistency shared-memory paradigms with the benefits of the object-based approaches. It should be based on user-defined objects to eliminate all possibility of false sharing with a minimum of system processing and to make sure that the size of and the organization of objects fit the application well. It should attempt to use locking integrated with the consistency protocol so that there is no wasted effort, and it should have a simple conceptual basis of organizing distributed data structures, and should be able to represent both matrices and linked structures well. And the system should provide special features for special circumstances and offer many opportunities for programmers to optimize the operation of the system by supplying the system with additional information.

Chapter 3

Basic DSDS System

3.1 Motivation & System Concepts

As discussed in Chapter 1, some applications are better suited to a message-passing environment and others, a shared-data environment. For shared-data applications, many DSD systems have been proposed in the past. Classic page-based DSM systems have fundamental flaws in that operation granularity is very small and data granularity can be too large and isn't likely to fit application semantics. The weak-consistency DSM and shared-object systems also correct these flaws, but the DSDS approach fits in somewhere between these types of systems and offers advantages over both.

We want to provide a system that will allow programs to be written that will access shared data but do not need to use explicit message passing to achieve that. The idea for *Distributed Shared Data Structures* (*DSDS*) is derived from Classic-DSM systems, where DSDS fixes the main problems by using user-defined storage units instead of system-defined pages. DSDS is the realization of the thesis requirements outlined in Chapter 1.

DSDS is based on a very simple system concept, that of shared data "items". A data item is a user-delimited chunk of memory that the user accesses in a highly struc-

tured way, by locking the data item either exclusively or non-exclusively while using it and then releasing the lock when done. This structuring is similar to that required by entry consistency. Hereafter, we will call the data items "objects" for convenience, but the meaning should not be confused with object-oriented-programming objects. Since the objects are well defined and user-delimited, there is no possibility of false sharing between objects and no need to expend system overhead to deal with it.

DSDS also logically fits in somewhere between a shared-memory system and a message-passing system, providing the caching/direct-matrix-accessing/hidden-sophisticated-protocol capabilities and conveniences of DSM and the data-flow-control capabilities of message passing.

Unfortunately, the basic-system data concept and consequently the user-interface design of this system are similar to the Shared Regions design [Sandhu95] for processor cache consistency. The Shared Regions work was discovered only very recently and the work in this thesis is entirely independent of Shared Regions. The similar features of the DSDS system were derived completely independently and follow from previous work with a page-based message-passing implementation [Bruce92]. The good news in this is that the Shared Regions design serves to validate this design. The overlap with the Shared Regions work is limited to the API (application-program interface) only, as Shared Regions is a different implementation for a very different environment and it does not consider the other areas covered in this thesis.

Using highly structured accessing to shared objects allows the mechanism to have a completely user-level software implementation. This ensures portability across a wide range of distributed-system architectures. It also allows the implementation to be very efficient since most DSM systems try to track and figure out what changes to shared memory a program has made while it runs, whereas with this interface, a programmer tells the system directly. The consequence of this is that the operation of the system is made explicit to the programmer, giving the programmer more insight into exactly what will happen during execution.

Insight into system behaviour is good, since in general, programming in a way that ignores the underlying mechanism, as in DSM systems, and pretending that it is something it is not (physically shared memory) is likely to hinder performance. This has been recognized since the early days of virtual-memory systems when ignoring data layout for large structures in virtual memory often led to programs with very poor performance. The DSDS approach is to recognize that the shared data exists on a distributed system and that it must be accessed in certain restricted ways to achieve the best performance.

The design also integrates the locking of objects with the coherency of the data inside the objects, so the two protocols can be integrated and implemented as one. The approach that is taken is to implement an invalidation-based protocol (as the basic protocol) that gives sequential consistency of the data items with respect to their locking (in the basic case).

The basic system introduced in this chapter gives good performance for many operations but it is not as good in other cases. There are many potential routes for performance optimization with such a system, where additional programmer-supplied information can allow the data to flow where it is supposed to when it is supposed to be there. These possibilities are discussed in Chapter 4.

To achieve the requirements, the system needs to provide a set of initialization and locking primitives to the user. Internally, the system needs to allow for the flow of objects in response to the object locking. It therefore needs internal data structures to represent control information for the user-defined objects and it needs an object-management protocol.

The next sections of this chapter describe the user interface to and the prototype implementation of the basic DSDS system; the discussion of design issues and alternatives continues and then at the end of the chapter. The user-interface description begins with the system primitives for managing the objects. The primitives are discussed to give the reader a concrete understanding of the system implementation.

3.2 Primitives

The basic mechanism for DSDS provides the following primitives, expressed in pseudo-C-language prototypes with explanations. Objects are explicitly created and destroyed and must always be accessed through the DSDS system. The primitives are discussed here since they are important to understanding the system as a whole and the remainder of the thesis document. The descriptions also include discussion of design alternatives and discuss efficiency issues important for the later performance enhancements. Lower-level issues (restrictions in the current implementation and details of implementation in a Unix environment) are discussed in Appendix A.

```
int err = DsdsInit( void );
```

This primitive initializes the shared-variable environment, including the shared-variable arena (the memory from which all of the shared variables are allocated). This primitive must be called before any other DSDS primitive, and after the special transport protocol has been initialized. A special transport system is needed to bind host addresses and port numbers to an application, but it is not necessary to discuss it here.

The run-time system is structured so that all processes register with the transport system on startup and the transport system allocates names for all of the processes and then lets them start. When the DsdsInit() call returns, all N user processes have joined the system and the system is ready to operate. The first host to join is assigned the role of "coordinator" in the system, which means that it manages memory allocation of the global shared-memory arena.

```
void DsdsShutdown( void );
```

This primitive shuts down the DSDS environment. No other shared-variable primitive may be called after this one. All operating-system resources held by the shared-variable environment will be reclaimed. The call takes no arguments and returns no result. Control will not be returned to the caller until all other processes in the

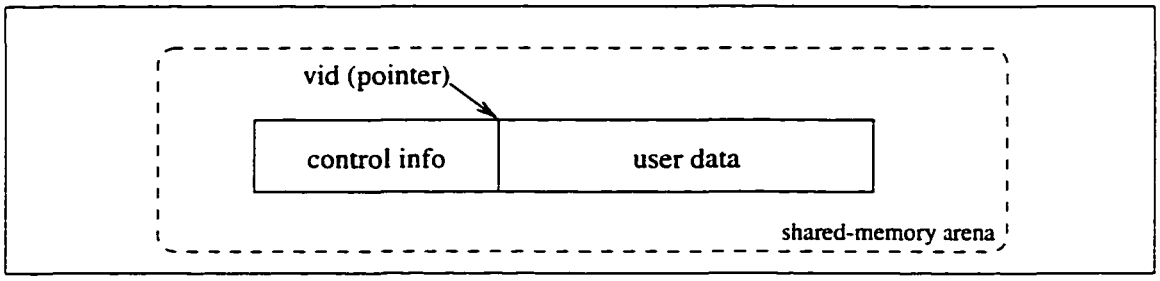


Figure 3.1: Structure of an Allocated Object

DSDS application have called their copy of this primitive also, to avoid race conditions. Adding detection of failed hosts and processes could be done, but this is not in the prototype system.

```
void *vid = SvCreate( int bytes );
```

This primitive creates a shared-data object. Memory is allocated for the object in the shared-variable arena and the user bytes of the object remain *uninitialized*. The bytes argument tells how many user bytes the object is to have. Conceptually, this call is similar to the malloc() call for allocating dynamic memory in C.

The return value for this primitive serves a two-fold purpose. First, it is a pointer to the memory allocated for user storage, and it can be used with all of the usual programming-language techniques for working with dynamic data structures. Second, it is a globally unique identifier for the shared-data object that is used with other calls to identify the object to be operated upon. An illustration of the structure of an allocated object, with control and user storage, is show in Figure 3.1. The returned object id is labelled as "vid", which stands for "shared variable identifier. A NULL pointer is returned if this call fails. The standard C value NULL can be used as a null pointer in dynamic data structures.

The cost of a create operation is some fixed amount of local processing plus at least one allocation of shared memory. The system uses a coordinator machine for memory management. If the creation is performed on a remote machine from the coordinator,

then object creation will involve a round of RPC. (A "round" of message passing includes a request and a corresponding response.) On the coordinator machine, the cost of the memory allocation on the coordinator depends on how fragmented the free memory is; free memory blocks are kept in a linear structure.

For usage, obviously, every object needs to be created. But, object creation is expensive, so a programmer may want to do it sparingly. For instance, if objects are being used basically to transport data from one place to another, instead of creating the objects on demand and then destroying them as soon as they are used, it makes more sense to reuse the existing objects, if practical.

Also, The actual amount of arena space consumed by an object is on the order of 100 bytes larger than the requested size because of system overhead. This overhead should be kept in mind when deciding on the granularity of objects.

```
int err = SvDestroy( void *vid );
```

This primitive destroys an object (deletes it from existence). The arena memory allocated to the object is reclaimed and all control information for the object is deleted. This is a global operation in that the object is removed from all processes of a distributed application, and the coherency semantics of an object removal are the same as for an object update. *Dangling pointers* to the removed object should not be left behind in any user-defined data structures; the system does not clean them up.

This primitive is not actually implemented in the prototype system. It is just a stub and is therefore very efficient. A semantic advantage is that object pointers will never be reused, but, of course, the occupied space will never be freed up. Nevertheless, code should be written to call the primitive in the appropriate places.

```
int err = SvLock( void *vid, int lockType );
```

This primitive locks an object for reading or updating. An object must be locked before its data can be accessed by the user program. When an object is locked, its internal values are guaranteed not to change unexpectedly while the lock is held.

The call takes two arguments. The vid identifies the object to be locked, and the lockType tells whether the object is to be locked for shared (SV_SLOCK) or exclusive (SV_XLOCK) accessing. A shared lock is for read-only access whereas an exclusive lock is for read and write access. Multiple processes in an application can hold a shared lock on an object simultaneously, but only one process may hold an exclusive lock at a time, and while it does, no other processes may hold a shared lock.

A process is allowed to hold "nested" locks on an object. The purpose of this feature is to allow large modular programs that manipulate shared-memory objects to be constructed more easily. For the prototype implementation, a nested lock must be specifically requested by bit-oring the bit value SV_NESTED to the lock type (this is for debugging purposes only, since many programming errors will manifest themselves as unintentionally nesting locks). The prototype system allows only the nesting of locks of the same type, but there is no significant reason that nested locks of mixed types cannot be handled.

If the user data of an object is accessed when the object is not locked, then no consistency/coherency/mutual-exclusion semantics are provided. A correct program should lock data objects whenever they are being accessed.

The most obvious means of synchronizing computation in DSDS is by locking objects. In fact, this is required in order to access coherent object contents. Object locks can be viewed as being very similar to semaphores (or "mutexes"), except that locks can be either shared or exclusive and locking influences the "flow" of objects throughout the DSDS system.

Depending on the current distribution status of the object being locked, SvLock() can have a small constant time or it can require waiting for multiple RPC latencies and even waiting arbitrary periods of time for remote hosts to release objects. Even when an object is owned by the local host, if there are remote readers, they will need to be invalidated before the lock is acquired.

In addition to being used to lock individual objects, the locking primitive can also be used to lock an arbitrary group of objects, in the same way that a semaphore can be used to lock an arbitrary collection of data. A programmer must individually lock every data object he wants to use anyway, but this does not guarantee any consistency semantics for groups of objects, so he can use an auxiliary object to totally order accesses to an abstract structure. Transactions give serializability semantics to database systems, but DSDS does not implement transactions, so using an auxiliary object is a simple and effective way of implementing consistency. However, using an auxiliary object does not allow much concurrency. Depending on the algorithm, there may be other ways of implementing consistency that allow for greater concurrency, such as the technique discussed with linked-structure cursors in Section 5.3.1.

```
int err = SvRelease( void *vid );
```

This primitive releases the lock held on an object obtained by SvLock() and gives remote processes full rights to modify the object in question. (The giving of access rights to remote processes is conditional on there not being another (nested) lock still held by the releasing process.) A return value of zero means the call completed successfully, and a non-zero return value means that the local process does not possess a lock of any type on the object in question.

One thing that a programmer must keep in mind when working with locks is to remember to release the locks. Forgetting to acquire locks leads to unsafe computations and inconsistency, and forgetting to release locks leads to deadlock.

SvRelease() is a local operation, so there is a small fixed processor overhead for the cost of releasing an object. That is, unless there are pending external requests for the object. In that case, a queue of pending requests is processed, which may involve sending out messages to remote hosts.

An object should be released when it is finished being processed. However, it may be advantageous in some cases to hold onto the lock if it is likely that it will be

needed again in the near future. There is a small amount of processing overhead in re-locking a local object.

```
int err = SvRegister( int dirent, void *vid );
```

This primitive is part of the mechanism used for locating important objects when the system starts. Since objects in the system are given arbitrary dynamic names when they are created, a means is needed to give fixed, well known, pre-programmed names to the important objects, so that all processes in the system can locate them easily. This mechanism uses a register/locate model where the process that creates the object registers it with the pre-programmed name, and afterwards, the other processes locate the object using its pre-programmed name.

The names used with this mechanism are integers from the range of zero up to a certain fixed limit, to keep the prototype implementation simple. They are called directory entries.

The *SvRegister()* primitive registers a given object with a directory entry. A return value other than zero means that an error occurred and the object is unregistered. A directory entry may be filled only once.

```
void *vid = SvLookup( int dirent );
```

This primitive looks up an object identifier that was registered with SvRegister(). The directory entry (dirent) to check is given, and if no object is registered with that entry, then a value of NULL is returned. Register/lookup accesses should normally be explicitly synchronized to avoid race conditions, perhaps with a barrier (below).

```
int err = SvBarrier( int barrierId, int totalWaiters );
```

Barriers are used to synchronize computation. Typically, all processes in a computation participate in a barrier, and a barrier is placed between different phases of a computation. The semantics of a barrier are that each process that enters a barrier blocks until all processes have entered the barrier, and they are all unblocked at that point. A barrier is useful where an algorithm requires that all of the objects updated

before a certain point in the computation be consistent before computation continues. A good example of this requirement is between main-loop iterations of many in-place matrix-processing algorithms.

This primitive causes the current process to block until totalWaiters processes (including the local process) of an application call this primitive with the same barrierId and totalWaiters arguments. The barrierId argument is chosen arbitrarily by the user, with the restriction that multiple concurrent barriers must not have the same identifier. Barriers use integer identifiers for simplicity, and this set of identifiers is completely disjoint from the object identifiers and well known register/locate directory entries. The same barrierId can be used again and again for successive synchronizations of an algorithm, with the restriction that a single barrier must be used by the same set of processes every time. The totalWaiters is also an integer, and there must actually be totalWaiters processes in the application that will call SvBarrier() at some time in the future, or the application will deadlock waiting for the processes that never enter the barrier.

The cost of activating a barrier is sending a message to a coordinator host, waiting until all hosts participating send a message, and then waiting for the coordinator to reply. This requires at least one RPC turnaround.

3.3 System Data Structures

Several data structures are used to implement the prototype system. This section describes the shared-memory heap and named objects.

3.3.1 Shared-Memory Heap

Perhaps the most fundamental data structure is the shared-memory heap. This is implemented as a Unix shared-memory segment, or multiple segments if necessary,

and it is used for all of the shared-variable and dynamic management information. A shared-memory segment is necessary for interprocess communication between the master and slave processes, but it is also important for cross-host memory management. This is because shared-memory segments are always mapped into an address space at a user-controllable address. A suitable high address that is well aligned (w.r.t. typical VM page sizes) makes a very good choice.

One implication of using a shared-memory segment in this way is that it complicates using heterogeneous computers in the system. It is possible to implement heterogeneous objects in such an environment if the system is aware of the data types of all fields in a data object and if proper data-type conversions are performed whenever an object is imported or exported. A "canonical form" is likely a good solution here. The only other thing to ensure is that all of the basic data types have the same sizes in bytes. This is straightforward in Unix/C environments, even when mixing 32- and 64-bit machines, provided that 32-bit values are always used (or 64 bits). However, this feature is not implemented in the prototype system, and the prototype system requires a homogeneous environment.

The DSDS system keeps consistent global control over all memory allocations to the shared-object memory, so that all user objects will have the same address at all hosts. It does this by serializing all memory-allocation requests through a single "coordinator" host, which is the host with transport address 1. The transport service that is used maps all hosts to addresses in the range of 1 to N. Serializing all requests through host #1 may be viewed as a bottleneck, but in practice it can also be an optimization. Many scientific shared-memory programs use a *fan-out/fan-in* paradigm where all data is initialized or read in from a file at a single host and then it is dispersed to a number of worker hosts for processing and the data is later brought back together at the initial host to be written to disk or processed sequentially in some way. Thus, the bulk of memory allocations (at initialization time) are carried out at the coordinator, and require only a local subroutine call rather than an RPC.

The DSDS system also maintains complete control over the organization of the heap using a simple custom memory-allocation management mechanism. Actually, two disjoint, independent memory arenas are used, one for the global shared memory, and one used only locally by the master and slave processes for temporary structures such as queues.

3.3.2 Named Objects

Named objects (used with SvRegister() and SvLookup()) are implemented using a simple array (on each host). The array is maintained on the coordinator host of the system and is cached on all other hosts. The arrays on all hosts are initialized with NULL pointers, which is the return value for an unregistered vid. When a host other than the coordinator performs a lookup that results in a NULL pointer, a request is sent to the coordinator (otherwise, the lookup is satisfied locally). The coordinator sends back the requested entry plus a number of successive entries for better performance (in case entries are scanned sequentially). An SvRegister operation on a remote host must communicate with the coordinator. It is expected that most registrations will be executed on the coordinator, in practice, so it will be a local operation.

3.4 Object-Management Protocol

The basic protocol is executed by each process involved in a distributed application. The algorithms of the protocol are explained in this section. An invalidation-based protocol is implemented and it has two major components: acquiring a lock on an object and invalidating remote copies of an object, plus the implementation of barriers and a few minor components.

The process interactions are based on requests, replies, timeouts, and busy signals.

Since the transport service is assumed to be unreliable, timeouts and retransmissions are needed to ensure that a message arrives at its destination at least once. The organization of the algorithms ensures that duplicate messages are handled appropriately. If a request cannot be answered immediately because of some condition that can last for an arbitrary period of time (such as a requested object being locked remotely), then the pending request is recorded by the receiver and a "busy signal" is returned to the sender. The sender then waits for a period of time for the reply before it times out and retransmits. The receiver sends back a reply as soon as it can. Short and long timeout periods, as described here, have been used in various other systems, for example [Peterson96, pp. 318–319]. The value assigned to the long timeout is not critical to performance.

The protocol is also based on a master and slave process executing on each host. The master process executes the user code and executes the requester portions of the protocol interactions when a user calls a system primitive. The slave process is responsible for accepting requests from external masters, processing them, and sending back reply messages. A slave process is used to provide responsiveness to incoming external requests.

3.4.1 Data-Object Control Information

The operation of the protocol requires control information for each shared-data object in the system. The control information for an object is shown in Figure 3.2. These object records and their user data are allocated in the global shared-object arena.

The vid field is the object id. The VIDT is actually a "void *" type, the type of object ids. This field is not essential, but it is convenient to keep a copy of the object id here. The valid field indicates whether the object is currently valid or not. If the object is valid, then the local copy of the user data is up-to-date and can be used. If it is not valid, then the local copy of the user data cannot be used and a fresh copy

```
typedef struct {
    VIDT    vid;
    BOOL    valid;
    int     flags;
    int     lockType;
    int     slocks;
    int     xlocks;
    int     version;
    int     ownerSeq;
    int     readerSeq;
    int     invalSeq;
    int     owner;
    int     lastGrantOwnTo;
    VIDT    prefetchLink;
    void   *memoryPtr;
    int     dataSize;          /*in bytes*/
    int     dataAlloc;         /*in bytes*/
    int     dataInterleave;
    int     dataCount;
    int     readerCount;
    int     readers[ TR_MAX_PEERS ];
    int     holdReaderSeq[ TR_MAX_PEERS ];
    QUEUE   pendingQueue;
    int     magic;
} SV_CORE_REC;
```

Figure 3.2: In-Core Object Record

of the data must be fetched from a remote host before the data can be accessed.

The lockType field indicates the type of lock currently held on the object, either Nolock, Slock (shared), or Xlock (exclusive). The slocks and xlocks are supplemental to the lockType field, and tell the number of each kind of lock currently being held. These fields are necessary for managing nested locks. Since the counts of the nestings are kept rather than some recursive structure, the order of releasing locks can be arbitrarily interleaved and does not necessarily need to be "nested". The prototype only allows locks of one type to be nested, but there is enough information maintained here to allow arbitrary nesting of either kind of lock, providing that the lockType field always contains the most "restrictive" lock type necessary. Note, though, that if a shared lock were to be promoted to an exclusive lock simultaneously on more than one host, then a deadlock would ensue.

The version field gives the version of the current contents. The version number starts at one (1) when an object is created and its contents are uninitialized, and the version number is incremented every time an exclusive lock is acquired (since the holder of the exclusive lock may modify the object). The ownerSeq field is incremented every time ownership of the object changes, and is used as a control in the object-management protocol. The readerSeq field is used similarly, except to control the readership replication of an object, and the invalSeq field is used to control invalidations.

The owner field gives the "best guess" that the local host has of who the current *owner* of the object is. The owner of an object is the only host allowed to perform writes on the user data of an object and it controls the propagation of an object. If the owner field value is the address of the local machine, then the local machine is the owner of the object (in this case, the value is exact rather than a "best guess"). The lastGrantOwnTo field contains the id of the last host that the local host forwarded ownership to, and is needed for anomalous cases in the object-management protocol.

The readerCount field gives the number of *readers* of the object. A "reader" is a

host remote to the owner host that has a *valid* copy of the data object (and hence, is allowed to read the user data of the object). The readers array gives the transport addresses of all of the current *readers* of the object. A fixed-size array was chosen for simplicity, as the prototype transport service places a well known limit on the number of peers of an application. A bitmap may have been a better choice, or a combination using a bitmap (integer) for the first 32 transport addresses and an array (or linked list) for higher-addressed readers. In addition to identifying the readers, the holdReaderSeq field is needed to tell which readerSeq of the object each reader has. These reader-information fields are used only if the local host is the current *owner* of the object.

The memoryPtr, dataSize, dataInterleave, and dataCount fields contain the memory-mapping information for the user data of the object. The dataInterleave and dataCount contain information for more complex mappings of objects as will be discussed in Section 5.2. The memory of a regular object is always allocated in the heap immediately after the control information. This way, the address of the control information can be quickly computed from the pointer to the user data (i.e., the object identifier) using simple pointer arithmetic. The dataAlloc field is included to allow the object size to change dynamically without destroying and recreating the object. In this case, the dataSize could be changed, and the dataAlloc field could be used to determine the maximum size that dataSize can be set to. This feature has not been implemented.

The pendingQueue is an abstract type used to store requests concerning the object that come from remote hosts that cannot be processed immediately by the local host because the object is *busy* when the request is received. The queue implements a bidirectionally linked list of records containing pending request messages, of either Lock or Invalidate types.

There are also a few miscellaneous fields. The flags field contains several flags, which are used for various purposes. The prefetchLink field is used for prefetching, which is described in the Section 4.1. The magic field contains a "magic number"

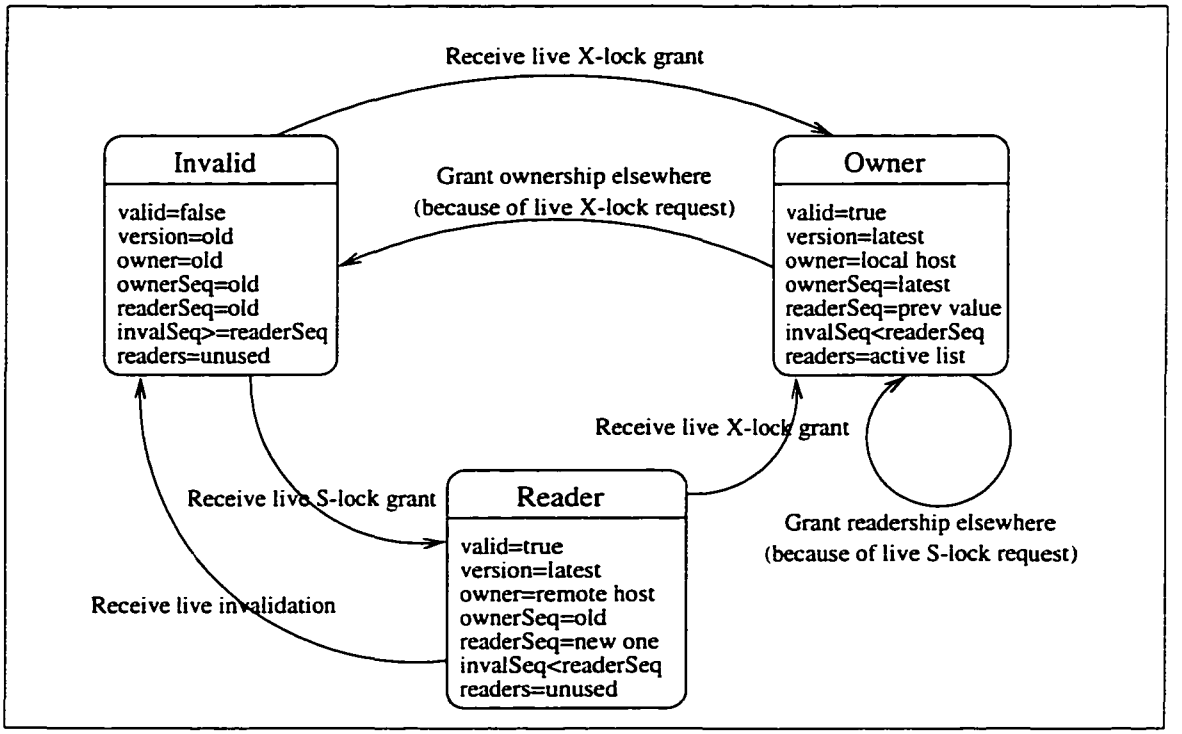


Figure 3.3: Object States on a Single Host

which is used as a first line of defence against a user passing an incorrect object identifier to a system primitive. (A magic number is a specific value stored into a field of a structure that can be used to later on to verify the type of the structure by the presence of the value at the right location).

Figure 3.3 shows an illustration of the states and state transitions that the control information of an object on a single host can go through. The remainder of Section 3.4 further explains the state transitions. The figure qualifies the received messages as "live" to distinguish them from old and duplicated messages. The means for making this distinction are explained later in Section 3.4

3.4.2 Acquiring a Lock

In the simplest case, the local process is the *owner* of the object to be locked, so it has a *valid* copy of the object data and has the right to make all decisions about which hosts are allowed what access to the object. There is only one owner of an object at a time, and the owner can grant itself either a shared or exclusive lock at any time. A shared lock can be granted immediately simply by setting the lockType field of the object record to Slock (and incrementing the slocks count), since no exclusive lock can exist on the object at a remote host at the same time. An exclusive lock can be granted immediately only if there are no remote *readers*. A reader is a process other than the owner that currently has a valid copy of the object data. All readers must be invalidated (procedure explained below) before the exclusive lock can be fully granted and the local process allowed to proceed.

If the local process is not the owner of the object but has a valid copy and is requesting a shared lock, then the shared lock can be also granted immediately without any interaction with other processes.

If the local process is not the owner and it wants an exclusive lock or if the local process does not have a valid copy of the object, then a remote lock must be acquired. The job of acquiring a remote lock on an object is split into two algorithms: one executed by the process that wants to lock an object, and the other by a process that can potentially grant access to a object. The lock-requester algorithm is shown in Figure 3.4.

The requester consults the owner field of its object record for its "best guess" at which host the current owner of the object is, and it sends a Lock-request message to that process. The format of a Lock-request message is shown in Figure 3.5.

All messages have the same header of msgType (special code value), vid (the object the request is for), version (the version of the object that the requester holds or most recently held), highOwnerSeq (the highest owner sequence that the requesting host has

```
REQUESTER: Outline for acquiring a remote lock
1:  Send request for lock to best guess of the current owner;
2:  Get back a reply message or timeout;
    switch( reply message type ) {
    case valid GRANT:
        Extract data and accept the lock–local host now owner or a reader;
        Update owner as sender of grant message if shared lock;
        if (local host requested an Xlock) {
            Accept the information in the Grant message;
            Increment the version number;
            Increment the ownerSeq field;
            Invalidate all of the readers;
        }
        Return from subroutine;
    case valid BUSY:
        Set timeout period to busy-signal timeout value;
        Update best guess of owner as sender of busy signal;
        Go back to step 2;
    case TIMEOUT:
        Go back to step 1;
    }
```

Figure 3.4: Basic Remote-Lock-Request Algorithm

```
typedef struct {
    int     msgType;        // message type
    VIDT    vid;            // object id
    int     version;        // requester's version
    int     highOwnerSeq;   // highest owner encountered
    int     requester;      // address of requester
    int     reqChannel;     // master/slave channel
    int     lockType;       // type of lock requested
} MSG_LOCK;
```

Figure 3.5: Lock-Message Format

held), requester (the transport address of the requesting host), and reqChannel (the channel to send the reply back to on the requesting host, either the master or the slave channel). In the case of a Lock request, the message also includes the lockType field that tells whether a shared or exclusive lock is requested.

After sending out a request message to the presumed owner of the object, the requester waits for a reply to its request, or times out and sends the request again. If the presumed owner of the object is not the actual owner, then the request will be forwarded until it finally does reach the actual owner. If the reply received back is a Busy signal, then the timeout period is adjusted upward and the timeout is repeated without retransmitting the request, since a Busy reply acknowledges that the request was correctly received and will be replied to later.

Alternatively, a Grant message can be received that satisfies the requested lock, and the algorithm completes. A grant message includes the standard header, the owner- and reader-sequence values, the current reader list for an Xlock, the memory-mapping fields of the object record, plus the user data of the object. Grant messages need to be checked for validity before accepting them, because duplicate and old messages are not filtered out at the transport level. The owner-sequence value of the Grant is checked to ensure that the grant came from a host that was more recently the owner of the object. For an Xlock, this alone guarantees that the message is genuine, since the decision to forward ownership to the local host is made once and for all, and if the local host is receiving notification of this for the first time, then ownership could not have propagated any further.

For an Slock, a Grant message is genuine if the ownership sequence is higher than has been received so far, the reader-sequence value is also higher than has been received, and the reader-sequence value has not been invalidated (more later). The Grant is genuine because it is not a duplicate since the local host would have received the reader-sequence value already, and it is still "live" because an invalidation has not been received for it yet. A reader-sequence value is used instead of the version

number of an object to allow the same version of an object to be correctly granted any number of times (this will be useful later).

3.4.3 Handling Invalidations

In acquiring an Xlock, Invalidation messages must be sent out in order to maintain sequential consistency. No other host in the system is allowed to read an object after another host acquires an exclusive lock on it, so all readers must be ordered to invalidate their copies before the exclusive lock is granted. The Invalidation message includes the standard header of the Lock message, plus the reader-sequence value (readerSeq) that is to be invalidated. This value is available from the reader list of an object, which includes the readerSeq values of all current readership grants.

Each host has an invalidation-sequence (invalSeq) value for each object it has. When a host receives an Invalidation request, it checks the readerSeq value against the invalSeq value. If the readerSeq is smaller, then the Invalidation is a retransmission (since the object is already invalidated w.r.t. the readerSeq) and an affirmative acknowledgement is generated. If the object is locked locally, then a Busy signal for the invalidation request is generated and sent back to the owner. Otherwise, the object is currently free and and the invalidation is new, so the object is invalidated (if it is not already) and the invalSeq value is assigned the readerSeq value. This action ensures that, even if a read grant was previously generated that was not received at the reader, the read grant will not be accepted if/when it finally does arrive. An affirmative acknowledgement to the invalidation is generated. For an illustration of invalidation, see Figure 3.6.

The owner collects all Invalidation acknowledgments. The readerSeq value that is included in the Invalidation-Acknowledgement message is checked against the value for the reader in the reader list to ensure that the acknowledgment is "live". The readers are successively removed from the reader list as each responds, and after

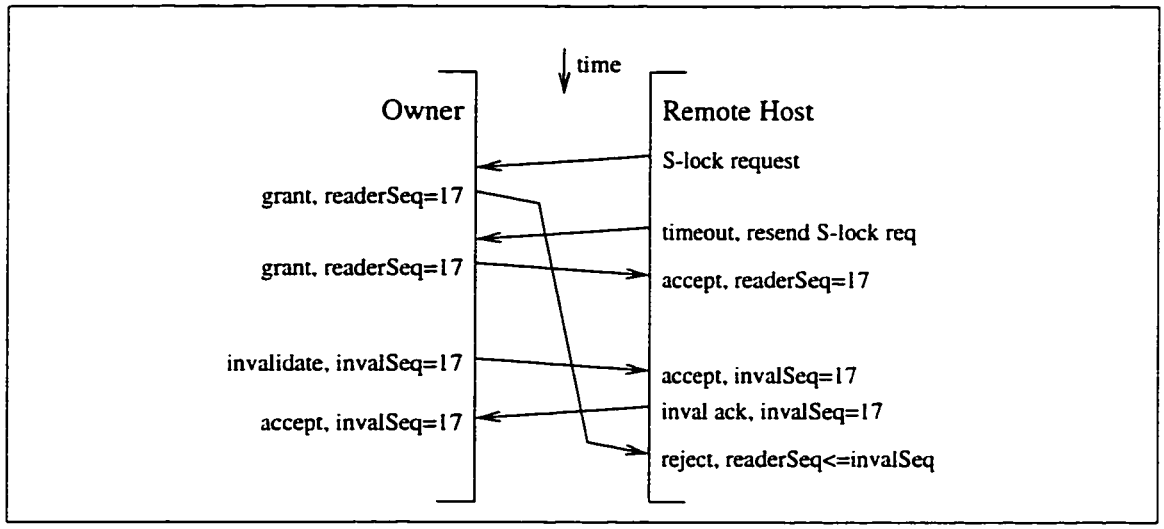


Figure 3.6: Object Invalidation

they have all been removed, the owner is allowed to complete its exclusive-locking operation.

3.4.4 Granting Locks

There are three cases to the lock-granting algorithm. The case of the granter being the owner of the object and receiving an Xlock request, the case of the granter being the owner of the object and receiving an Slock request, and the case of the "granter" not being the owner of the object.

The part of the Granter algorithm for the case that an exclusive lock is requested and the local machine is the owner is shown in Figure 3.7. The action to take in this case is pretty clear. If the object is currently locked locally, in either shared or exclusive mode, then the remote lock cannot be granted immediately; it must wait until after the local host releases its lock. The lock-request message is inserted into the pending queue for the object, which is checked when the SvRelease() primitive is called. The entire lock message is enqueued, as it is fairly small, and at release time,

```
GRANTER: Granter is owner, X-lock request received
    if ( local host has an Xlock or Slock on the object in question ) {
        Send back a busy signal;
        Add the lock request to the pending list of the object;
    } else {
        Invalidate local copy of the object (but retain the user data);
        Update the owner and last-grant fields to point to the requester;
        Remove the requester from the readers list if it is there;
        Send back the reply granting ownership;
        Send out invalidation requests to readers;
            (note: the lock requester will collect the invalidation acks)
    }
```

Figure 3.7: Granter Algorithm: Owner, X-lock Requested.

it is simply "replayed" to the protocol-handling code as if the lock message had been received at that time. This simplifies the implementation.

If, on the other hand, the object is not locked locally, then ownership of the object is free to be granted to a remote host. Since it is normally the case that the lock request will be received because a remote host wishes to acquire exclusive access to the object (in addition to simply wanting to become the owner of the object), the local host invalidates its copy of the object. An alternative would be for the local host to add itself to the reader list, but this would just make it necessary for the new owner to invalidate the copy at the local host later on, at greater cost. This is also a good time to remove the requester from the reader list of the object, if the requester is there.

The local host then sends a Grant message, granting ownership of the object to the requesting host, and sending the up-to-date object data to the requester. One optimization is that if the requester's version of the object is the same as the granter's version, then the requester already has an up-to-date copy of the object data, so no data is sent with the grant message. The requester detects this condition also and retains the previous user-data contents.

```
GRANTER: Granter is owner, S-lock request is received
    if ( requester is already in reader list ) {
        /* this is a repeat request */
        Retransmit Grant message;
    } else {
        if ( local host is holding an Xlock ) {
            Send Busy signal;
            Add the lock request to the pending list of the object;
        } else {
            Increment and record the reader sequence we are sending;
            Add the requester to the reader list;
            Send Grant message;
        }
    }
```

Figure 3.8: Granter Algorithm: Owner, S-lock Requested.

As another implemented invalidation optimization, the local host also sends out invalidation messages to all current readers of the object, telling them to invalidate their copy of the object and acknowledge their invalidation to the new owner. This decreases latency by allowing the Invalidation messages to be sent out while the Grant message is still "in transit." This optimization would not be possible with an RPC transport system. If retransmissions of the invalidation requests are needed, then the new owner will send them out.

The part of the Granter algorithm for the case that a shared lock is requested and the local machine is the owner is shown in Figure 3.8. In this case, a check is made to see whether the requester is already in the reader list. If it is, then the request is taken to be a retransmission and a Grant message is re-sent. One thing to note is that an old read request could be delayed for some obscure reason, and it could arrive now, when the distributed computation has moved beyond its being needed. The algorithm does not specifically detect this, and the algorithm may grant a lock when one is not truly requested, but the algorithm will still operate correctly, grant the unsolicited request, and possibly recover later by invalidating the unsolicited request. The recovery cost

```
GRANTER: Receiver is not owner
    if ( [forwarded] lock request has "seen" a higher ownerSeq of object ) {
        /* ownership of the object has been granted to local host by a */
        /* remote node but the Grant just has not arrived yet */
        Send back a Busy signal;
        Add the lock request to the pending list of the object;
        If the master process is not currently trying to acquire a lock
            on this object, then generate a "bogus" lock request;
    } else {
        if (local host last granted ownership to the requester and an Xlock
                is requested) {
            /* retransmitted current Xlock request */
            Send back a Grant message as if local host were the valid owner;
        } else {
            Forward lock request to local "best guess" of owner;
        }
    }
```

Figure 3.9: Granter Algorithm: Not Owner.

is weighed against the low probability of messages being unexpectedly delayed and correctly delivered later on.

If the reader is not already in the reader list, then a new request is being made. If an exclusive lock is not currently being held locally, then reader access can be granted, so the requester is added to the reader list with a new reader-sequence value, and a Grant message is returned. If an exclusive lock is being held, then the read request cannot be granted so the request is added to the pending-request queue for the object and a Busy signal is returned.

The part of the Granter algorithm for the case that either type of lock is requested but the local machine is not actually the owner of the object in question is shown in Figure 3.9. This case is actually more complicated than the other two, because anomalous situations can arise. Conceptually, all that the needs to be done in this case is to forward the request to the local host's "best guess" of the current owner.

However, lost and reordered messages need to be handled properly. When ownership passes from one host to another, a "forwarding chain" is created, and all lock requests received by non-owner hosts along that forwarding chain are forwarded to the next host until the active owner is ultimately found.

One anomalous case that can arise is that ownership is forwarded to a remote host by the current owner, but the Grant message is lost. When this happens, no host in the system thinks it is the current owner. To allow detection of this situation, the highOwnerSeq field of the Lock-request message is changed every time a lock request is forwarded. It is changed to the ownerSeq value of the forwarding host for the object. Thus, if a Lock request is received that has a highOwnerSeq value that is higher than the ownerSeq of the local host for the object in question, then it is known that the current host must be the rightful owner of the object. This is because a host further along the forwarding chain has forwarded to the local host, and this can only happen if ownership has been granted to the local host and the grant message was lost, and it must be the case that the last host to forward the lock to the local host was the previous owner. Figure 3.10 shows an illustration of this anomaly.

The correct action in this case is to enqueue the request in the pending list and send back a busy signal. However, it is possible that the Master process of the local host is not currently trying to acquire a lock on the object, and if it is not, then a "bogus" lock request needs to be generated and sent to the last host that forwarded the lock request to the local host. This is to ensure that ownership will ultimately reach this host and that the protocol will not stall in this case forever.

Another anomalous case is that the local host has forwarded ownership to another host, and an Xlock request has come in from that other host. This means that it has not received the message and the message might be lost, so the Xlock Grant is retransmitted.

Ownership forwarding is complicated and it may not be completely clear that the mechanism functions correctly in all cases. In addition to the testing of the

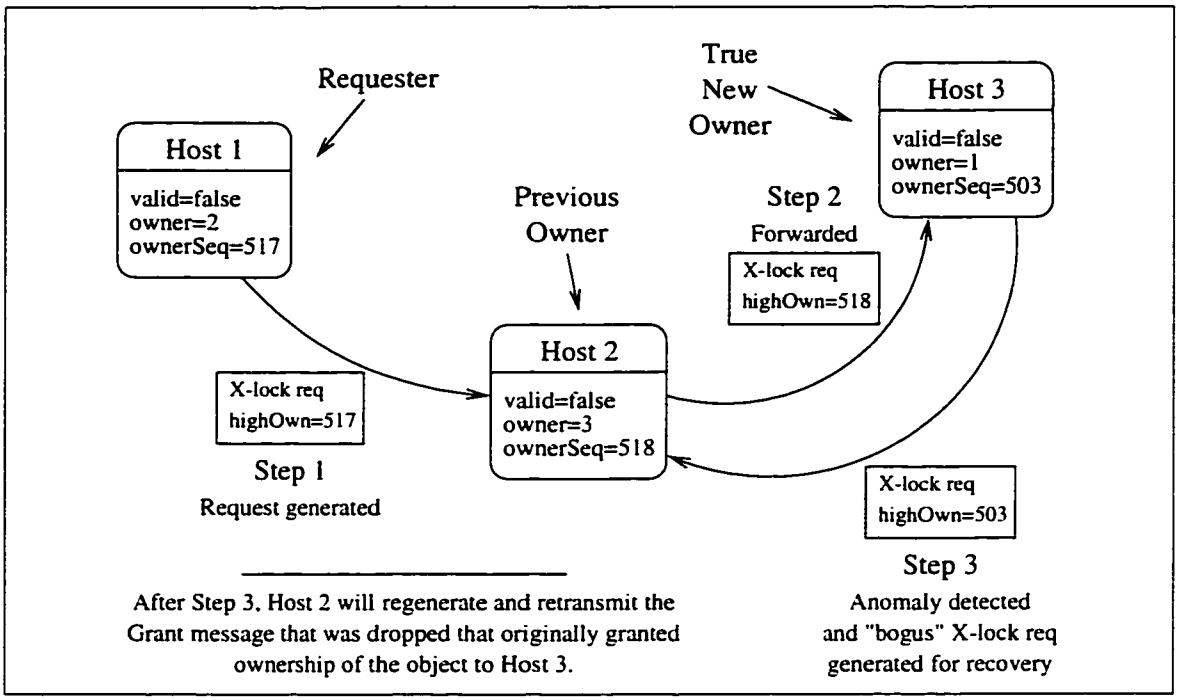


Figure 3.10: Detection of Lost Grant Message

implemented code, we can argue its correctness as follows. An important simplifying aspect of the algorithm is that there is always one host that is the owner of an object. The owner is the host that makes all decisions about granting requests, and the only host that is allowed to hold an X-lock on the object. For purposes of this discussion, ownership changes instantaneously when the grant message is sent, even if the new owner is not aware that it is the new owner. The owner may not "think" it is the owner, but no non-owner will ever "think" it is the owner. Thus, there can never be two hosts in the system that think that they are each the owner; there can only be zero or one host in the system that thinks it is the owner. Ownership cannot change in the lost-grant state (when no host thinks it is the owner because the ownership grant message has not arrived yet from the previous owner). This is because there is no owner to grant ownership anywhere else, except for a retransmission to the true owner in the case of a lost grant. Lock requests cannot be forwarded past the true owner in the lost-grant state because of the control variables, specifically the owner sequence. The case of a lock being forwarded to a host that has a lower owner sequence than some host forwarding the message means that the host the message is forwarded to must be the new owner, although that host has not received a grant message yet. The lock-forwarding chain will also always converge on the true owner, since a host is only allowed to change its "best guess" pointer to a host that was more recently the owner than its previous best guess.

3.4.5 Asynchronous-Protocol Evaluation

The object-management protocol is designed as an asynchronous protocol that is implemented with asynchronous message passing and special control variables for error recovery. The protocol has a very simple basis, that of sending a request and getting back a reply. In this sense, it is similar to RPC; however, RPC is a "highly synchronous" mechanism that always blocks when a process is waiting for a reply. With this protocol, replies are actually decoupled from requests, so that asynchronous

operation, such as forwarding a message without expecting a reply, and prefetching (discussed in Section 4.1) can be provided for free. Requests are idempotent and the system does not need to keep duplicate replies in case of communication failures or explicitly handle duplicate requests. The design also kept the system from becoming more complicated as the system acquired more features.

The sequence variables in the control information of the objects maintain the consistency of the mechanism and correct problems with lost and reordered messages, removing the need for a reliable transport system. The ownership-sequence variable is used to detect lost ownership-grant messages. The reader-sequence and invalidation-sequence variables are used to ignore duplicate readership-grant and invalidation-request messages. Messages are considered "live" only if they contain sequence values that have not been processed already by the local host.

Some of the good things that come out of the approach are that locks can be forwarded directly until the owner is found and invalidation requests can be sent out by the granter when granting ownership to a remote host, and the remote host can collect the invalidations, saving message-passing latency over the case where the new owner would send out invalidation requests after receiving the grant message from the old owner.

3.4.6 Barrier Implementation

The implementation of the barrier mechanism is fairly straightforward. A client/server approach is used in which all hosts participating in the barrier send a request message to a barrier server and then, upon collecting all of the barrier requests, the barrier is triggered and all hosts are notified.

The server is chosen using a hashing function on the barrier id and all hosts are capable of being the server for a barrier. An instance number is also associated with each barrier id, and multiple invocations are controlled by the instance number. A

list of requesters is also kept to detect and handle duplicate requests. When a request arrives that has an old version number, a Wake-up message is returned immediately. Otherwise, the waiting list is checked for the requesting host, and if found and the new instance number is greater than the one used previously, the count for the instance of the barrier is incremented. The current instance number of a barrier is also maintained in the SvBarrier() call on the client side, being incremented every time a barrier completes.

3.5 Discussion of Design Issues

User-Defined Objects

There are a few different choices for the size and semantics of the "objects" managed by the shared-storage system. DSM systems manage pages and can suffer from false-sharing problems. Even multiple-writer control algorithms suffer from unnecessary overhead in processing page faults, copying pages, and computing and re-integrating "diff" records.

Using user-defined objects eliminates false-sharing problems and allows the user to select the correct granularity for the application and the normal operation of the data objects.

Per-Object Locking

Providing a one-to-one relationship between object and object locking allows the lock-management and sharing protocols to be fully integrated. It also provides a conceptually clear interface to the system for the user, in that every object needs to be locked while being used. In many cases, this will be sufficient to handle concurrency-control problems with objects. In other cases, more complicated concurrency control

will be needed, and this will be discussed in later chapters.

On the other hand, it does mean that the user is responsible for properly locking an object every time it is accessed, and managing this properly can be a difficult proposition. However, giving users choice over the object granularity and semantics allows them to choose something sensible for the application.

Object Ids

Using addresses as object ids is convenient. It allows programmers to handle distributed shared data structures in the familiar way that dynamic data structures are managed. Only one name is needed per object, rather than needing to maintain two names if objects are named and accessed separately, or rather than accessing a lock of a separate name that is linked to the object data.

A major programming convenience that distributed-shared-data systems provide over message-passing systems is that users can use and pass data pointers around the system with the pointers being meaningful and usable.

It also allows the system to access object-control information very efficiently at a direct offset in the memory space from the id. This is especially important when up-to-date data is available locally on the machine and pieces need to be accessed independently and repeatedly very quickly, such as with nodes of a graph.

Invalidation Versus Update Protocols

For the basic object-management protocol, an invalidation-based approach was chosen. This type of protocol gives the user a direct indication of what work will be done on each locking-control call, depending on what he knows about the data-sharing patterns of the application. For example, when a programmer codes a SvLock() operation, he knows that the object will be fetched at that point and that if the structure

of the distributed program implies that there will not be a valid copy of the object on the local host at the time that the call is made, that the lock operation he is coding will require remote communication.

The main problem with an invalidation protocol is that it can lead to unnecessary "cache misses" in which expensive operations are needed to retrieve needed data. The main problem with an update protocol is that it can lead to unnecessary message-passing and protocol-processing overhead.

With classic DSM systems, there was no choice but to use an invalidation protocol, since the unit of operational granularity was so small, but more recent DSM systems are able to use an update protocol because the operational granularity is much relaxed. DSDS also has a relaxed consistency model, based on the grain of the locking, so it too is a candidate for using an update protocol, or variations and equivalents thereof.

An invalidation protocol ensures strict sequential update semantics for the shared data items, which makes it easy to understand. An invalidation protocol can also give better performance for data that has strong locality-of-reference patterns. An invalidation protocol can also be modified in a conceptually simple way to become a hybrid of both invalidation and update protocols, which has advantages over either. This subject will be discussed further in Chapter 4.

Ownership Forwarding

There are a number of different ways to control mutual exclusion and in terms of control overhead, the ownership-forwarding scheme is quite a good choice. Distributed and centralized approaches are the two general classes of solutions to distributed mutual exclusion. Distributed solutions involve sending many messages around the system to obtain consensus, and because of this tend to have unacceptably high overheads. In a centralized approach, there is one point of control for making decisions about mutual exclusion, and the issue is whether that point remains static or moves

around. If it remains static, then all remote hosts must interact with the controller to lock the object and obtain the up-to-date contents, and again to post the new version of the data and release the lock. The ownership-forwarding scheme moves object ownership on demand and therefore avoids the post/release phase of the static version. The ownership-forwarding scheme also avoids remote operations altogether when applications exhibit locality of reference.

However, a problem with ownership-forwarding scheme is that a long chain of ownership forwarding can develop for hosts that access the data infrequently w.r.t. the ownership movement. There are steps that can be taken to combat this problem, which include informing infrequent accessors which host is the current owner and providing "shortcuts" for forwarding. A different approach might be to use "conservative" forwarding rather than the "liberal" forwarding that the protocol implies. Application-sharing patterns determine how the ownership forwarding performs. These issues, too, will be discussed in Chapter 4.

Fault Tolerance

The highly structured approach to shared data in this system makes the system a good candidate for implementing a degree of *fault tolerance*. The way that data is passed around with old copies left "lying about" means that there is potential for partial recovery of computation, or, with more system work and constraints, perhaps full recovery. Process checkpointing would be needed for computation recovery, and information about how results flowed between objects may be needed to recover a consistent set of object versions if one host died while holding ownership of many objects. On the other hand, it could be as simple as guaranteeing that there are always at least k copies of the latest version of each object on different processors before continuing from a SvRelease().

While we acknowledge the importance of these issues, we feel they are outside

the scope of this project. If very large pools of machines were intended to be used, fault-tolerance would likely be needed, but our system could still be useful.

Design Mistakes?

The system interface was initially designed to use integers as object ids and an indirect lookup table to resolve these integers into control-block addresses and ultimately into user-data addresses. As noted above, this kind of scheme required the user to work with two different identifiers for controlling each object: the object id and the user-data pointer. The user-data pointer was returned by the SvLock() primitive, and was only valid while the object was locked, and needed to be re-acquired later. There was a large amount of fixed overhead for accessing the data of an object even when the data was available locally and no protocol action was required. It was confusing and tedious to work with two different identifiers, and one particular problem was remembering to use SV_NULL rather than NULL, since zero was a useful object id number and the NULL pointer is always defined to have the value zero.

The advantages of using integer ids were that the object data could change physical addresses with no problems. Also, physical memory could be managed locally without needing to sequence memory allocations through a central coordinator, and each host only had to allocate memory for the objects that it actually used. Object ids can also have meaningful name values, specially selected so that all hosts can just call an object by its name without the application specifically needing to look the identifier up.

Another potential design mistake was deciding to use processes rather than threads to implement the master and slave "threads" on each host. The Unix shared-memory-segment mechanism is standard among different versions of Unix, but threads offer better potential for lower overhead. However, modern threads seem to be migrating into the kernel of operating systems, so the overhead gains may not be as significant

as with fully user-level threads, and user-level threads often have many issues of their own.

It may also be the case that the system guarantees over-strong consistency semantics. That is, when processing groups of data objects, if it is guaranteed that no other process will attempt to access them before some point in a computation, such as a barrier, then it is wasteful to execute the full consistency-maintaining invalidation protocol synchronously. The user has the choice over the grain and semantics of data objects to help avoid such problems, but they may be inherent in some situations. Asynchronous strategies are discussed later on.

Finally, it may have made the design of the protocol much simpler if a reliable-datagram service had been implemented instead of using (essentially) raw UDP. Without reliable transport, there are many anomalous and unexpected situations that can occur in a protocol because the interleaving of executions of special cases can become incredibly complicated. The object-management protocol went through a number of revisions before finishing up with the owner-, reader-, invalidation-sequence-numbering scheme, and all of the protocols were complicated by the unreliable transport service. However, a reliable, in-order datagram service might have resulted in significantly poorer performance.

3.6 Conclusions

This chapter introduces a conceptually straightforward system for managing distributed shared data. The concept is that of a potentially large dynamic distributed structure of user-defined objects that can be accessed by all hosts of an application. This allows the user to directly specify what structure is intended. The system is implemented entirely at the user-process level and is based on individually lockable objects. The necessary primitive operations are provided for allocating/deallocating and locking/releasing the objects. Two types of locks are supported, shared and ex-

clusive, to allow greater concurrency than in systems which provide only exclusive locks, since multiple shared locks can be correctly held concurrently. Barriers are also provided for group synchronization and barriers in general are quite convenient to use for iterative algorithms.

Physical-memory addresses are used for referring to all objects, and the physical address of each object is the same on every machine (although these addresses are allocated dynamically). Thus, objects only need one "name", and this is the same name that is used in programming with conventional pointers. An implication of this is that memory allocation needs to be performed globally, and a coordinator organization is used. The coordinator can allocate memory with very low cost but remote hosts must send RPC requests for global-memory allocations to the coordinator. However, this organization works well with a fan-out/fan-in paradigm of computing, where allocation and initialization is performed at a coordinator anyway. Since the objects are allocated dynamically, a basic name-lookup mechanism is also provided for initializing applications.

The prototype system is built in the Unix environment. Each host provides a multithreaded master/slave environment where the master executes the main program and the slave services protocol requests from remote hosts. This organization provides responsiveness to remote requests. The multithreading is based on Unix processes, kernel semaphores, and shared-memory segments. This may not be the most efficient organization, as kernel-level context switching is relatively expensive, but it is ubiquitous, straightforward, and secure to program. This organization also provides the shared-memory segments that can be mapped into the same address on all hosts, even hosts of different architectures.

The transport system is essentially raw UDP/IP. This mechanism is ubiquitous and very efficient, and UDP fits the requirements well since messages will typically be small and UDP gives the ability to send a message to any host in the world without any prior connection setup. It also creates limitations on the maximum

size of objects without special transport-system extensions and also complicates the control protocol, since UDP is unreliable. The only extension provided is the mapping of IP/port names to numbers from 1 to N for application-programming convenience.

The basic system implements an invalidation protocol that gives sequential consistency at the object level, but which is a loose-consistency model at the memory-byte level, since any number of memory operations may be performed on an object while it is locked, and the effects will only be made visible after the lock is released. The protocol is also dynamic and extensible, and we will extend it for additional functionality in the next chapter.

The protocol is based on ownership forwarding, which has pros and cons, but which is a good general solution to the distributed-mutual-exclusion problem. This paradigm allows any host to become owner of an object and therefore lock it repeatedly with little cost in the absence of accessing contention (only local processing is needed). When there is accessing contention, the locking strategy and explicit user control of the locking means the objects migrate only when one host is actually finished modifying them. The integration of locking and data flow is an efficient choice, since it eliminates the redundant messages that would be needed to control data transfer and locking independently. It also inherently supplies a basic but powerful synchronization mechanism (locking) that is all that will be needed in many cases.

The locking mechanism makes the operation and flow of the objects fairly visible and explicit. Doing the opposite, hiding the actual operation of a system, is in general likely to compromise performance, since a programmer has no idea what is likely to work well and what is not. This system does put a greater responsibility on the programmer than other systems do, but its highly structured environment promotes the writing of programs that have good data-sharing patterns, and gives the programmer control to coax out of the system the minimum number of message-passing operations that is possible.

However, explicit programmer control of a basic low-level locking mechanism is

not necessarily a utopia. There are outstanding issues of additional mechanisms and convenience of programming to be examined more closely in the following chapters.

Chapter 4

Performance Enhancements

In this chapter, we examine ways to enhance the performance of the basic DSDS system. The basic mechanism is fairly straightforward in its conceptual design but its performance may not be ideal for many types of applications, so there are many opportunities for optimization. Unfortunately, many of the optimizations introduced in this chapter are not implemented in the prototype system because of time and scope constraints.

Overall, the basic system works well and can be used effectively for many applications, as is demonstrated in Chapter 6. However, with some applications, performance can be hindered. The performance problems ultimately arise from message-passing latency, the wall-clock-time delay between a message being sent by one process and being received by another. In many cases, because of the request/reply nature of the protocol, the total latency of an operation will be twice the communication latency plus the processing time of the request. If latency were zero, the basic system would behave well without any enhancements.

Therefore, we will look at ways of hiding latency in this chapter. The main primitive that can cause delays is SvLock(), so it and its underlying consistency protocol will be the main areas considered for hiding latency. Many of the optimizations resulted

from experience with coding and observing the test applications in Chapter 6.

In this chapter, we describe direct ways of controlling the object-management protocol with prefetching and non-stalling invalidations. We also describe mechanisms to eagerly propagate and acquire new versions of objects with updating, optimistic locking, and tagged objects. Finally we describe low-level protocol enhancements.

4.1 Prefetching

4.1.1 Motivation

Prefetching is a means of hiding latency by moving data objects to where they will be needed, before they are accessed. In a simple message-passing system where a programmer is left with the task of writing a program with only that lowest-level mechanism, he will likely do whatever he can to make a program run as efficiently as possible. The programmer will often know the general data-flow requirements of the application, whereas a DSD (*Distributed Shared Data*) system could only adapt to this at runtime. The most efficient solution would be to allow the programmer to tell the runtime system about the data flow within the system. for the critical points where the system might be likely to miss out on an important optimization.

A common pattern in distributed computations is for bulky data blocks to move from one host to another or to multiple hosts at certain points in the computation (see Chapter 6 for examples of this pattern). A common subset of this phenomenon is *fan-out/fan-in*, which appears commonly in scientific, matrix-processing applications. There are many other more-complicated standing and one-shot sharing patterns.

The forms of prefetching that are discussed in the rest of this section are direct prefetch, prefetching chains of objects, prefetching initiated by the current owner to send readership or ownership asynchronously, a method of voluntary reader invalidation, and finally a primitive that performs operations locally that normally require

communication but can violate sequential consistency if used incorrectly. All of these prefetching mechanisms are implemented in the prototype system.

4.1.2 Explicit Prefetch

A simple mechanism to provide is explicit prefetching. With this, a programmer specifies which object is to be locked for which type of accessing, and the system internally generates a lock request but does not wait for the lock to be granted. The lock of the object will be granted asynchronously w.r.t. the execution of the user program. The interface is

```
int err = Prefetch( void *vid, int lockType );
```

This call is useful when it is known that an object will be needed in the near future, but there is work that can be done in between the time it is known that the object will be needed and the time that the object is actually needed. When the object is actually needed, a regular SvLock() will acquire it. Future variables in ABC++ [O'Farrell94] are essentially equivalent to explicit prefetching, as both mechanisms are flavors of asynchronous RPC.

If the intermediate work takes at least the time of the remote request/reply cycle, then the latency of acquiring the remote lock will be completely hidden. If not, then the latency will be reduced by the time of the work done. A special "incoming" flag is set in the object-control information to prevent the SvLock() request from generating a lock-request message immediately, for this second case.

If the intermediate work is another remote lock request, then the timing should be approximately correct to hide all latency. If the object with the requested rights is available locally, then the prefetch operation will perform very little work and will send no messages. Also, to prevent an object from being prefetched and then being requested by some other remote host, a "stickiness" flag can be set on the object by

"or-ing" the lockType argument with SV_STICKY. The "stickiness" will be deactivated when the object is explicitly locked.

For a usage example, if a program will be accessing a well known list of currently remote objects, then it makes sense to prefetch one object in advance so that it will be prefetched to the local host in the time of one RPC round. The reduction of total waiting time in this case will depend upon the amount of work performed on each object when it arrives, before moving on to access the next object.

4.1.3 Prefetch Links

Given the organization of the Basic DSDS mechanism, if all data is read in on one host in the system, then it will stay there unless another host requests access to the data. In the case that the "natural size" of an object is small and a remote host is going to be processing a large bulk of data one object at a time, a considerable overhead will be incurred, as the data objects are slowly migrated from the host that they are currently on to the host where they will be processed, on demand, one object at a time. In fact, unless there is enough processing per record to offset the data-migration cost, then we would be better off just to forget about distributing the processing and doing it all on the initial machine. There will be on the order of a 1-ms real-time synchronization (remote-request) penalty for each object accessed by the remote host. This will be a common pattern, since data is commonly read in by or generated by one particular host. Explicit prefetching could be used to transfer these objects, but the object identifiers would need to be known in advance, and a request message would be needed for each object prefetched.

Thus, we would like to have a large set of data objects migrate to another host in one operation. The user should be able to specify to the DSDS system which objects should be migrated together, and upon accessing any member of the migration group, all of the objects in the migration group should be transmitted together to the

requester, in one large network operation.

As a simple implementation of this idea, the following DSDS primitive is proposed:

```
int err = SvSetPrefetchLink( void *vid, void *nextVid );
```

This allows the user to specify an ordered linked list of objects to be migrated together. When another host accesses an object that has its prefetch link set, the requested lock will be granted, along with read access or ownership (whichever was selected) to the object pointed to by the prefetch link, and then the object that the prefetch object points to, and so on down the chain. This continues until an object is encountered that the granting host isn't the owner of or is holding a lock on, or until N objects or B bytes of user data have been sent out in the current "volley". (The term "volley" is used to refer to a group of messages being sent out without pausing in between.) These values are system/application-dependent tuning parameters to prevent over-running the message-transport system. Appropriate values probably need to be determined experimentally and are probably time-varying based on network load. The best solution would be to derive these parameters from current information from the transport system, if available.

In order to set a prefetch link, the local host does not need to have a current lock on the object, but it does need to be the owner of the object at the time or else an error indication is returned. This is a reasonable restriction because we want prefetch links to be modifiable, since object groups may change during the execution of a user program. Setting the prefetch link to NULL is an easy way to disable the prefetch link for an object. The prefetch-link information will propagate with the ownership of the object.

This organization of the mechanism requires very little additional control information (one field of control information per object) but still gives flexible power for specifying prefetch groups. However, there are subtle potential problems that must be handled. If an object-grant message is lost, then when a retransmitted request

for the missing object comes in, an entire chain will be sent out again, unless this situation is handled specifically. With exclusive locks, the granter gives up ownership of every transmitted object in the prefetch chain with the original request, so if it is no longer the owner but a valid X-lock request is received, the local host sends a proper grant reply but does not send any objects from the prefetch chain since they have already been sent. With shared locks, the local host checks if the requesting host is already in the reader list. If so, then the grant is retransmitted, but no objects from the prefetch list are sent, since they will have already been sent.

There is also a subtle problem on the receiving end. To be consistent with the basic protocol (and to ensure that the first object can be used as quickly as possible), the grant of access to each object in the chain will take place as an individual operation represented by an individual message. Each individual message will take a certain amount of time to reach the destination host. So, if the requesting host processes the object that it originally asked for and goes on to requesting the next object in the chain before that next object arrives from the prefetch auto-grant, then it will generate a request for the next object and send it to the granting host. Then, the prefetch grant will finally arrive and the requesting host will continue processing, but the freshly generated request is still live and it will cause the granter to retransmit the object-grant message, wasting bandwidth. If the timing is bad enough, this will happen for every object in the prefetch chain.

To combat this problem, a field needs to be added to the grant message that indicates what the next record is that will be granted along a prefetch chain (or NULL if it is the last record of the chain). This will allow the requesting host to delay sending out a lock request until after the timeout period has elapsed, when it makes a lock request on the next object in the prefetch chain. A bit flag will be required in the object-control information to record that an object is "incoming". The prefetched item should then show up at the requesting host automatically before the timeout, where it can be used without the necessity of sending out a useless lock request. Note

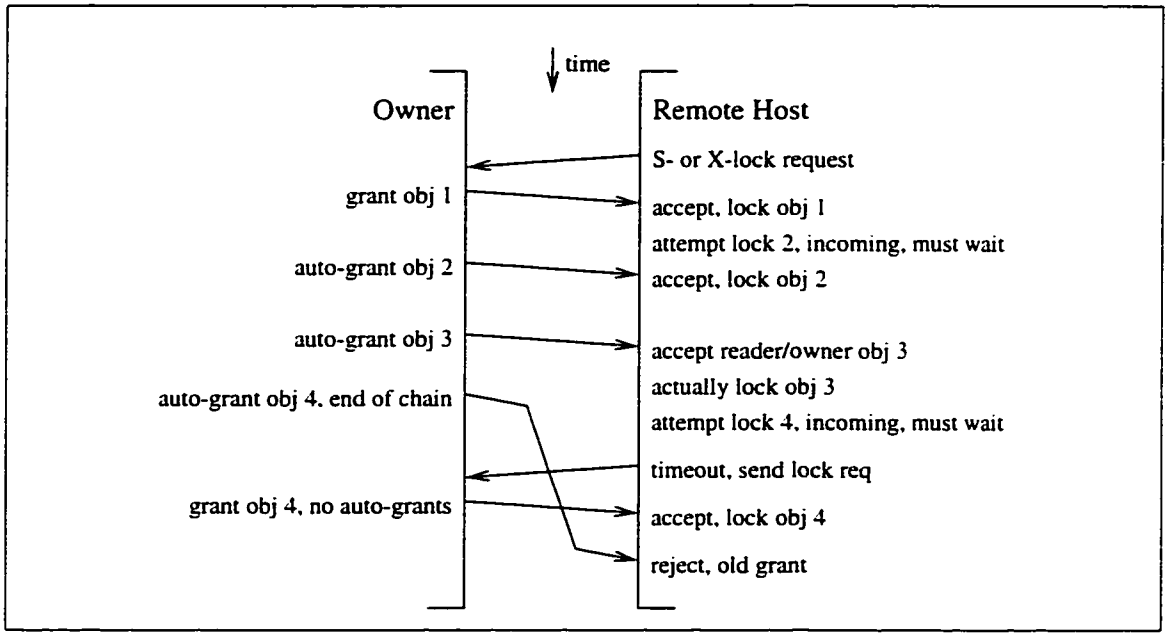


Figure 4.1: Prefetching-Links Example

that this simple solution requires that the data be accessed and prefetched in the same order in order for it to be guaranteed not to produce useless retransmission requests. See Figure 4.1 for an example of various prefetch-link cases.

The benefit of this prefetching method is that the requesting host will request the first object of a prefetch chain and will receive the first object after one round of RPC latency. Then, when the application requests the next object, it may have already arrived in which case the entire request latency is saved, or it will be flagged as "incoming" and the local host will wait to receive it, saving a portion of the remote-request latency.

4.1.4 Asynchronous Grant

The preceding prefetching mechanisms are called by the requester. It is also useful, perhaps even more useful, for the granter to initiate prefetching. This is accomplished

with the primitive

```
int err = SvGrant( void *vid, int lockType, int destHost );
```

The required data migration of an algorithm with regular sharing patterns will usually be spelled out by its structure. For algorithms with a fan-out/fan-in pattern, this primitive is exactly what is needed. In this case, a programmer would grant X-locks (ownership) to the host that will be handling the object during the computation phase, and ownership (or readership, depending on requirements) is transferred back to the controller process during fan-in.

For X-locks, this primitive is equivalent to sending a message, which will be automatically received by the destination host and put into the correct memory area. For S-locks, ownership will be kept by the sender and the new reader will need to be invalidated at some point in the future.

In the ideal case, asynchronous grants will arrive at the destination host before it attempts to access the object. In this case, the request phase of locking will be eliminated, and the latency of the grant phase will be completely hidden. The only cost is the processing overhead in the sender and receiver to handle the messages, but note that these costs are also paid in a normal lock request. (Measurements of the costs of various protocol operations are presented in Section 6.1.)

However, if the receiver requests the object before it arrives, then it will send a useless lock-request message which will generate a useless second grant message. To combat this case, a programmer can "or" the SV_INCOMING flag to the lock type when calling the SvLock() primitive to indicate that the object is likely "incoming". Or, if the asynchronous grants are separated from the locks by a barrier, then it is likely that useless messages will be avoided. This is simply because the asynchronous grants are likely to reach their destinations while the hosts in the system wait for the barrier. Note that there are no synchronization problems with allowing asynchronous grants to be active while the system is awaiting a barrier, since a host only "synchronizes"

with an object when it takes a lock on it, and not necessarily when a readership or ownership grant message arrives at the host. The system is free to change the physical storage host of an object at any time, provided that this does not conflict with locks being held on the object.

This primitive is implemented very simply by synthesizing a "bogus" lock message from the destination host, and then applying it to the protocol. A grant message of the right type is generated and sent off, completely consistently with the rest of the protocol.

4.1.5 Pre-Invalidation

In some cases, a host will lock an object for reading and will not need access to the object again before it is updated by some remote host. In this case, the following primitive can be used:

```
int err = SvInvalidate( void *vid );
```

This will invalidate the object on the local machine and send an invalidation-acknowledgment message to the owner of the object. If the local host is actually the owner, then an error code will be harmlessly returned and the object will remain as it was.

This "prefetching" optimization will save the owner of the object from needing to invalidate the object in the future. The latency of the invalidation-acknowledgment phase will be hidden and the invalidation-request phase will be eliminated. Once again, there is a danger that the owner could decide to invalidate the object before the pre-invalidation-acknowledgment message reaches it.

The implementation of this primitive is similar to the asynchronous-grant primitive. A "bogus" invalidation-request message is synthesized on the local machine and then is applied to the protocol-handling code. The object is invalidated and

the acknowledgment message is sent back to the owner. The owner accepts this invalidation-acknowledgment message and removes the sender from the reader list, regardless of whether it sent a request or not.

4.1.6 Local Invalidation

There is second type of pre-invalidation that requires no protocol messages to be exchanged at all. The primitive is

```
int err = SvLocalInvalidate( void *vid );
```

When executed on a reader of the object, the reader invalidates its copy of the object and does nothing else. When executed on the owner, the reader list is set to empty. This makes short work of object invalidation, but it can only be used in certain circumstances.

Note that all of the previous prefetching optimizations were harmless in that even if they are performed inappropriately, the protocol would still work correctly and guarantee sequential consistency. The only cost is that the inappropriate operations done may need to be un-done. However, with local invalidation, using it inappropriately could lead to violation of sequential consistency.

There is no danger from local invalidation on a reader, since it will not access the local object again until executing corrective protocol operations. It will also still respond correctly to invalidation requests from the owner of the object. However, there is danger with emptying the reader list on the owner, since if that happens but there are still valid readers in the system who can access the old version of the object at some point in the future, then sequential consistency will be violated.

Therefore, this primitive can only be used when it is guaranteed that the owner and all readers of the object will execute the local invalidation at the same, consistent point in the program. A very good example of such a point is a barrier point, situated

between phases or iterations of an algorithm. In fact, it is unlikely that it should be applied anywhere else, and there is strong argument to programmatically tie its operation in with a barrier, to help prevent programming mistakes, although that is not done here for implementation simplicity.

The implementation of this primitive is straightforward, as explained by its function. It gives a very simple means of achieving a "clean slate" of no readers for objects that are used to disseminate information between steps of a computation, where all of the otherwise-necessary invalidation would be time-consuming.

4.1.7 Prefetching in General

Prefetching is the weapon that DSDS uses to make processing more efficient. However, in some cases, prefetching can actually make applications run less efficiently. A few cases were pointed out above where useless requests would be generated and processed, but it is also possible that in some applications, different phases require different amounts of CPU utilization and the whole algorithm runs faster if in certain critical phases *all* CPU time is devoted to user computations rather than to prefetching activities. There is also extra overhead of context switching to handle incoming auto-prefetched objects which can detract from CPU time available for processing.

Where prefetching is appropriate, there is the question of which prefetching mechanism to use. SvPrefetch() is used by the requesting host sometime before requests are made, where it is ideal if it is about one RPC latency time, and only a few objects should be prefetched at a time. SvSetPrefetchLink() should be used if there is a static set of objects that should all be migrated together, and this saves on the network traffic of individual prefetch requests. And SvGrant() should be used by the granter in "pipeline" (e.g., fan-in/fan-out) computations to save the next stage of the pipeline from enduring the latency of explicitly requesting an object.

4.2 Non-Stalling Invalidations

Another method of hiding latency is to delay the point at which synchronous replies are collected in the management protocol. A caution is that this can change the consistency semantics of the protocol.

Whenever there are readers of an object in the system and the owner of the object wants to acquire an exclusive lock (X-lock) on the object, it must invalidate all of the readers to ensure the coherency of the object data in the system. In the basic system, this invalidation is performed when an object is being locked for exclusive access by sending out invalidation requests to all of the readers of the object and then waiting to collect all of the invalidation-acknowledgment messages from the readers (with retransmissions for lost invalidation-request messages).

A performance enhancement to the basic mechanism is to wait to collect the invalidation-acknowledgments at some time later than when acquiring the X-lock. We will call this waiting-for-invalidations situation "stalling," since the lock-requesting host cannot proceed any further until the operation is finished. We also call it "stalling" to distinguish this form of waiting from other forms of waiting for synchronization. There are many potential times to stall, but only four of them are useful for our purposes.

4.2.1 Stall on Lock

"Stall on lock" is what the basic mechanism as described in Chapter 3 does. It turns out that this mechanism is more conservative than it needs to be, and fails to take advantage of potential parallelism.

The data-consistency semantic that this mechanism gives is *sequential consistency*. That is, data objects are accessed (locked) in the order that they would be if there was a single-ported global memory store (a server that serializes all shared-object

S-lock and X-lock requests and releases).

It is easy to see that there is a total order to all modifications of an object: this is guaranteed by the token-passing object-ownership protocol (assuming that consistency-breaking mechanisms such as local invalidation are used correctly). Only one host can officially own an object at any given time (regardless of the intricacies of the management protocol, since the decision of which host is the real current owner of an object is made atomically), and since only an object owner can hold an exclusive lock, all X-locks necessarily conflict.

It is also clear that shared and exclusive locks will always conflict properly (i.e., no other lock will exist in the system while an exclusive lock is being held on an object). The object-management protocol guarantees that there are no reader hosts for an object in the system that are not registered in the reader list (at the owner). When an X-lock is being requested, the owner sends out invalidation messages to all readers and waits for all of them to invalidate their reader status and acknowledge the (potentially new) owner, guaranteeing that there are no readers in the system when the X-lock processing continues on the (potentially new) owner host. Since a reader cannot acquire an S-lock locally when it is invalid, it will have to request to become a reader again with the owner, and the owner will not grant reader status again until after the X-lock has been released.

Therefore, shared and exclusive locks always conflict, and there can be at most one exclusive lock in the system at any time, so all *critical* lock requests are totally ordered and sequential consistency is guaranteed. The term "critical" here refers to the fact that there isn't an imposed total order on the concurrent acquisition of S-locks on multiple reader hosts, but this is unimportant since the object data will be the same at all reader hosts. Another minor thing to note is that the object owner can also be considered a "reader".

There are two more subtle semantic issues to consider in discussing the other invalidation-stalling schemes: the flow of object data and the flow of access control.

Since we know that the stall-on-lock mechanism is sequentially consistent (giving correct data flow) and guarantees no overlapping conflicting locks (which is the best control flow), we want to compare the properties of the other schemes to this one to determine if they are strictly correct and if not, then how usable they are.

4.2.2 Stall on Next Operation

In the "stall on next operation" scheme, the owner does not stall waiting for invalidations upon acquiring an exclusive lock on an object, but rather, stalls for this purpose upon attempting the very next DSDS object-access operation. The advantage of this scheme is that the user program is not stalled for the latency of the invalidation cycle; it can continue to execute useful user code, most likely processing the object that it just acquired, in parallel with the invalidation cycle. The disadvantage is that conflicting locks on objects can overlap in global time. For example, host #1 could acquire and use an X-lock on object A and host #2 could acquire and use an S-lock on object A even after host #1 has X-locked the object, since the invalidation request might not have reached host #2 yet. However, it turns out that this overlapping of conflicting locks does not violate sequential consistency.

There are no problems with overlapping locks since in the time between acquiring an X-lock and collecting the invalidation acknowledgments, no internal information is "leaked" from the owner host to the outside world. And since no DSDS operations can be performed inside the interval (by definition), all data objects accessed in the user processing code must have already been locked before acquiring the X-lock on the current data object (i.e., no external data can leak in within the interval). Therefore, this scheme is indistinguishable from and semantically equivalent to the basic mechanism; the outside world cannot see or influence the internal computation on the owner host, and vice-versa.

The performance of this scheme has been examined and compared to the basic

mechanism, using a simple illustrative test. The test is called the "random walker" test and consists of a graph of randomly-connected objects with a random amount of work to do at each node (summing from 1 to 2^r where r is a random number), and N hosts wandering around the graph and doing the work at each stop. A certain percentage of the time (w), an X-lock is acquired, and the rest of the time, an S-lock is acquired. With $N = 4$, $r = [9, 14]$, $w = 15\%$, and 300,000 total walking steps, the computation time is in the range of [127, 136] seconds (a range is shown here since different hosts give different execution times, especially the coordinator host) for the stall-on-lock case. The stall-on-next-operation case gives a range of [110, 119] seconds, a distinguishable improvement, in the range of 13%. The improvement depends mostly on the amount of work done between the lock and the next operation (whether it is enough to hide the latency), and the invalidation-latency period compared to the total computation time.

Please note that performance testing was performed here mostly for illustration purposes, and this is not necessarily a realistic test. The main performance testing of the system is in Chapter 6. The random-walker program was written early on to test the correctness of the object-management protocol, and it gives a simple but good stress test that can be configured to provide extreme circumstances.

4.2.3 Stall On Release

Another option is to stall on releasing the locked object. This gives a weaker type of consistency which we will call "isolated-object sequential consistency." In this scheme, any read of an object will return the value that was most recently written to it, but there are no guarantees made about consistency between objects. Each object is considered to be an independent memory w.r.t. sequential consistency. This consistency model is likely as generally useful as sequential consistency when applications are designed around the shared objects with the consistency in mind. This option

is selected for an object by setting the SV_ISOSEQ bit flag using the SvSetFlags() primitive.

4.2.4 Asynchronous Invalidation

As was pointed out in Section 3.5, the consistency guaranteed by the locking mechanism may be stronger than is really necessary for some applications. For example, applications that have phases or iterations to computations or which use concurrency control at a higher level than object locks may guarantee that an object will not be accessed by any other process until after a barrier or a release of some higher-level concurrency block. In this case, it does not make sense to stall for invalidations on acquiring each lock (or on the following operation), but rather, it makes sense to stall for invalidation acknowledgments only at a point immediately before other hosts might access the object in question.

This option is activated by "or-ing" the lockType argument of SvLock() with SV_ASYNC when locking the object. For convenience, the flag SV_ASYNC could be attached to an object with SvSetFlags(vid, flags) so that it does not need to be specified on every lock. At a later time, when the invalidations do need to be collected, the user executes the SvSync() primitive. Entering a barrier will cause asynchronous invalidations to complete also.

The implementation of this option is quite simple. When an object is locked, invalidation requests are sent out in the usual way, but the owner does not wait for invalidation acknowledgments. Instead, it adds the object to a list of outstanding asynchronous invalidations and returns to the user program. When invalidation acknowledgments arrive, the readers are removed from the reader list of the object and when all readers acknowledge, the object is removed from the list. The list is used to retransmit invalidation requests to remaining readers after synchronization is requested, and to repeat until all readers respond.

Like local invalidation, improper use of this facility will violate sequential consistency. If invalidations were synchronized at completely arbitrary times, this mechanism would give the semantics of *processor consistency*, by the definition of processor consistency.

4.3 Update Protocol

4.3.1 Motivation

Updating remote copies of an object automatically is another means of hiding latency, since data objects are distributed before they are accessed, so that the request phase of the write-invalidate protocol is not needed. The main issue is in the cost of delivering useless updates, that is, updates that will not be accessed before the next update is generated.

There are two cases in which an update protocol is likely to give optimal performance. They are when there is potentially a large number of data objects that will be accessed in an essentially random pattern, and when there is a fixed sharing pattern between one "producer" and any number of "consumers". In both cases, the objective is to produce a high "hit" rate for reading the data objects.

This is achieved in the first case by making the new data available to all hosts that are likely to access the data, before the time that they actually access it. Normally, an object would be invalidated at all remote hosts on a write operation and it would remain so. Thus, all readers would be required to obtain a remote read lock, taking one round of RPC communication. If an update arrives before the lock operation is attempted, then the request phase is eliminated and the latency of the grant phase is hidden. The second case is similar.

4.3.2 Application Interface

To select that an update protocol be used with an object, the user must set a control flag with the call SvSetFlags(vid, SV_UPDATE). (A minor note is that this will have an effect only if the local host is the owner of the object.) The flag can be cleared in a similar way using the SvClearFlags() primitive. For more temporary-only modification of the protocol, the SV_UPDATE flag can be added to the lock type field of the SvLock() primitive.

4.3.3 Implementation

Offering both an update protocol and sequential object consistency is a somewhat complicated issue. One cannot simply disregard invalidation and fire out updates instead, as this would cause consistency problems. The most naive implementation of an update protocol that upholds sequential consistency is to emulate what the user can achieve with the SvGrant() prefetching primitive. Thus, the locking and invalidation works as before, except that upon locking, the reader list is saved in a temporary location, and on SvRelease(), the equivalent of an SvGrant() is executed to grant read access to each host in the saved reader list.

A more sophisticated approach is to use stall-on-next-operation invalidations to limit the number of messages normally sent out. In this case, upon acquiring an exclusive lock, no invalidation requests are sent out. Instead, work is performed at either the corresponding SvRelease() or the next memory operation performed. If the next memory operation is the SvRelease(), then instead of an invalidation, a grant message is sent to all of the readers of the object, with a flag indicating that the grant is doubling as both a grant and an invalidation. Then, the readers accept the invalidation/grant and return an acknowledgment for the invalidated version. The owner stalls until all of the acknowledgments have been collected.

In the case that some memory operation other than the corresponding SvRelease()

is the next one called after the X-lock is acquired, then the remote readers must be invalidated at that point. Invalidations must be sent out and acknowledgments must be collected before the owner can begin to execute the next memory operation. Then, after the SvRelease(), the updates can be sent out without collecting any acknowledgments. This situation is not an issue when isolated-object sequential consistency is selected for an object (stall on release).

Unfortunately, the more sophisticated approach has a longer critical path than the naive approach, since the more sophisticated approach needs to collect acknowledgments after the release operation and the naive approach does not. The sophisticated approach is likely to be better when the object is locked and released with no other object operations in between and when the object is held for a relatively short period of time w.r.t. the round-trip RPC time, since the sophisticated approach passes fewer messages. Note that the naive approach can be used with either stall-on-lock or stall-on-next-operation invalidation.

If asynchronous invalidations are being used, then the sophisticated approach can be used without blocking to collect acknowledgments until the SvSync() or barrier is performed. Also in this case, no exception needs to be made if the next operation after the lock is not the corresponding release, since sequential consistency is not guaranteed until synchronization. A list of unacknowledged invalidation/updates can be kept, as with pure asynchronous invalidations. An alteration on asynchronous updates would be to never explicitly synchronize the updates and allow the system to work in a best-effort manner. Some other mechanism would be needed to give useful memory semantics in this case.

4.3.4 Broadcasting and Multicasting

Broadcasting and multicasting are commonly used techniques in distributed systems and distributed object/memory systems. Reliable totally ordered broadcasting and

multicasting are expensive system services, so we shall consider only best-effort unreliable broadcasting and multicasting. Fortunately, the system mechanisms already assume unreliable transport, so no real adaptation is needed, and the system mechanisms can already recover from message-reliability and message-ordering problems.

The update protocol discussed above sends out grant(/invalidate) messages to readers one at a time, but this can also be done at one time with one message to all hosts or subsets of hosts using broadcasting and multicasting. This can give a substantial saving over sending individual messages both in terms operating-system overhead and network usage. It also saves on overall latency, since a message must be delayed until the previous message has been completely transmitted over the physical network, for most network types.

A minor issue with granting readership is the way that reader-sequence numbers are currently used. Currently, each read grant to each host will be made with a unique readership sequence, but this is not actually a necessity. The only requirement is that independent grants to the same host must have unique, increasing reader-sequence numbers, so the same physical message can be used to grant ownership to an arbitrary number of hosts with no material changes to the protocol.

Greater mechanism issues are involved in deciding which subsets of hosts to broadcast/multicast to, as these facilities are hardware-dependent and may have restrictions. Typically, broadcasting facilities are limited to only the local physical network, and multicasting, if available, requires the establishment of process groups prior to sending any messages. Ways to deal with these restrictions are to use thresholds to decide whether to broadcast to all hosts in the system or to use point-to-point messages, and to cache process groups for repeated multicasting. Multicasting can also be implemented using broadcasting of messages that include a user-supplied list of target hosts. The waste in this case is the processing overhead of the hosts not in the list to receive the message and check the list, plus operating-system overhead.

Broadcasting and multicasting can be useful in other places too, specifically, with

invalidations (independent of using an update protocol), barriers, and user grants. Invalidation requests can be broadcast or multicast, costing one message, but the acknowledgments still need to be collected with point-to-point messages. Barriers would be the opposite in that they would be entered using point-to-point request messages and triggered using a broadcasted message (thereby decreasing the latency for most hosts). User grants would be sent with a single message. A means for the user to specify broadcast/multicast to a list of hosts would need to be provided, such as using special values for the destHost field and an optional argument which is a list of host ids.

Take special note that DSDS does not require broadcasting as a fundamental part of the protocol, but it can benefit from it nevertheless.

4.3.5 Tuning

There are two parameters that should be tuned when using an update protocol: the members of the reader list, and whether an update protocol should be used at all. Related to the latter parameter is whether a setting of using an invalidation protocol should be changed to using an update protocol. A flag SV_AUTO can be set with SvSetFlags() to specify that the system should dynamically choose which protocol is better for an object.

To avoid wasted-effort overhead, control is kept over which other processes are to receive unsolicited updates. If a process receives too many updates in a row that it does not attempt to read, then it will voluntarily and automatically execute the equivalent of an SvInvalidate() primitive to remove itself from the reader list of the object in question. This trimming can keep the wasted effort in the system from becoming severe, and the necessary decision-making variables are all available locally.

To decide whether an object should be using the update or the invalidation protocol, the variables used can be: how many sequential versions are requested at a

reader/all readers, and how many readers are heavy readers and how many are just occasional readers. This information is available at the owner, since all explicit read requests go through it. When a threshold is exceeded, the object becomes an updated object. When the reader list is voluntarily invalidated to zero readers, the object should revert back to being a write-invalidated object. Conceptually, if more information were maintained, the object could be updated for some hosts, and invalidated for others. This option is probably of limited utility, however.

4.4 Optimistic Locking

4.4.1 Motivation

Optimistic locking is another means of hiding latency. The locking protocol described so far is *conservative*, in that it only proceeds with operations when it is definitely known that proceeding will produce the correct results. The opposite of the term "conservative" is "optimistic," which means that a protocol will assume that the response it will receive from a counterpart will have a certain value and the protocol will continue processing as if the message had been received, and when the response is actually received, a full recovery will be performed if the value of the response is other than what was assumed.

The main benefit of optimism is a potential statistical speedup of the operation of locking protocols, since there are fewer critical-path delays. The costs of optimistic locking is greater complexity of protocols and usage, and the necessity of providing for recovery when the optimism fails.

Our intention is to provide a simple but usable mechanism for optimistic locking. This simplicity includes requiring *forward recovery* from user programs in the case that optimistic operations fail. There are two types of recovery in the presence of failures: forward and backward recovery. *Backward recovery* is usually provided by

the system and steps back to an earlier correct state of the computation and restarts from there. Providing this as an automatic service involves complexity and costs hidden from the user. Forward recovery is usually left to the user to deal with. The system returns an error indication, and the user code is responsible for taking the computation forward to a consistent state and continuing processing from there.

A paper related to this subject [Hermannsson94] provides a similar mechanism to what we propose to provide, although they rely on a hardware implementation to make the mechanism feasible. They also go into detail in comparing their system to the entry- and lazy-release-consistency mechanisms w.r.t. the performance and number of messages passed for sample problems. Ultimately, the performance of optimistic locking is likely to depend greatly on the algorithm being implemented.

4.4.2 Application Interface

The main place for optimism to be utilized in this system is in acquiring locks on objects. The general sequential-consistency guarantees of the objects and protocols limit the applicability of optimism, but the option of providing asynchronous updating with no synchronization, as mentioned in Section 4.3.3 is a place for optimism to be applied. The specific optimistic assumption is that the copy of the object that is "on hand" is the most up-to-date version of the object. With synchronous updating, this would be guaranteed w.r.t. sequential consistency (if the local version were valid), but with asynchronous updating, this is not guaranteed. (With asynchronous updating, the local copy of the object is *always* "valid" on all hosts that are actively using an object.)

Thus, for an object to be usable for optimistic locking, it must have its SV_ASYNC and SV_UPDATE flags set (using the SvSetFlags() primitive).

Actually, there is a restricted case where optimistic locking can also be applied without asynchronous updating of objects, that is if the local host is a reader of an

object that it can become the owner of the object with no opposition. However, this is a sub-case of the more general case discussed below, so this case does not need to be covered separately.

To acquire an optimistic lock, the SvLock() primitive is used in the standard way, except that the bit flag SV_OPTIMISTIC is "or-ed" to the lockType argument. Computation can then continue using the object as if it were locked in the appropriate way, up to a point where the lock is to be released or that it needs to be known for certain that the lock is valid. In either case, a new primitive needs to be executed:

```
int err = SvConfirm( void *vid );
```

This primitive verifies that the optimistically obtained lock is truly valid, in a sequentially consistent way. If so, then the primitive returns a value of zero indicating that all is well. If not, then the primitive returns an error code (SV_ERR_FAILED), the lock on the object is thrown away, and the user is expected to recover the local variables and semantics of his program to a consistent state to continue from. Cascading rollbacks potentially involving remote processes also need to be recovered by the user, so a wise implementor will likely limit use of these optimistic primitives to situations in which these cascades of rollbacks cannot occur.

Modified shared objects may need to be recovered too. The object that was locked speculatively and failed will be recovered automatically by the system. Other objects that were involved in the speculative execution will need to be recovered by other means. User objects that were X-locked normally can either have their contents restored by the user and be released like normal, or the user can lock them with a special bit flag of SV_BACKUP included in the lockType value. This instructs the system to make a backup copy of the data in the object in case the lock needs to be undone and the previous data needs to be restored. The recovery is performed using the primitive

```
int err = SvAbort( void *vid );
```

A regular SvRelease() operation on an object that had a backup copy made will discard the backup data. The SvAbort() operation also releases the holding of the lock without committing the changes, so an SvRelease() does not need to be called after an SvAbort().

Optimistically locked objects (other than the one that failed and was recovered automatically by the system) can be recovered in the same way, either using the SV_BACKUP flag on locking or by being user-restored, but the SvAbort() primitive always needs to be called for all optimistically locked objects that need to be discarded, except that this is optional for the lock whose SvConfirm() has failed, since object recovery will be applied upon detecting the failure. User-restored objects need to have their data restored before the call to SvAbort().

In the normal case that the SvConfirm() primitive succeeds, processing can continue with the user being confident that the lock succeeded, and an SvRelease() is called in the normal way to release the lock. It is an error to call SvRelease() without first calling SvConfirm(). The call could be made internally by the system, but this requirement is made explicit to help reduce programming errors.

4.4.3 Implementation

The implementation for the case that the local host is the owner of the object is straightforward—the object is locked in the usual way. There are three states to an optimistically locked object: Pending, Confirmed, and Denied. The object is marked as Confirmed. If the object is operating synchronously, then locking continues in the normal way, invalidating the readers if necessary. If the object operates asynchronously, then the readers do not need to be invalidated upon acquiring an X-lock.

If the local host is not the owner, then if the object is valid, the optimistic lock is returned and the system starts to confirm the validity of the lock. It does this by requesting ownership (and X-lock) of the object (regardless of whether a shared or ex-

clusive lock has been requested). The ownership must be obtained when asynchronous updates are being used since this is an efficient way to guarantee that sequential consistency is maintained. This is because the local host needs to guarantee that the version of the object that it possesses is the latest version of the object anywhere in the system. Even if a message were sent out to confirm with the owner that this is so, the owner would need to be prevented from allowing any more X-locks until after the read operation has completed, to maintain consistency. This would require more messages to be exchanged than to simply acquire ownership of the object.

An X-lock request is sent out to the owner at the time of returning an optimistic lock to the user. This request may include a special flag of SV_CONFIRM requesting that no user data be sent along with the grant, to limit the message size. If this flag is set in the lock request, then the owner sends back a grant message with no user data if the version field of the request is the same as the version of the object on the owner (i.e., the requester has the latest version of the object). Otherwise, the granter sends back the user data with the grant message, insuring that the requester will ultimately become owner of a consistent version of the object.

The X-lock request will be received by the owner of the object and ownership will be transferred to the local host while the user program continues to process the current contents of the object. When the ownership grant is received by the local host, sufficient information is available to confirm the optimistic lock simply by comparing the version of the ownership grant with the version of the object data that was optimistically granted. If they are the same, then the optimistic lock is valid; if they differ (i.e., if the optimistic version is out of date), then the optimistic lock is invalid and the state of the object is set to Denied. In this case, the user data accompanying the grant message (it will be included in this case) is stored in memory in the same way as if the user had locked the object with a SV_BACKUP flag, facilitating the automatic recovery. The data cannot be copied into the live user-data memory for the object since the user program may still be modifying that memory.

When the user executes the SvConfirm() primitive, the validity of the lock will be passed on to the user and any recovery will take place as appropriate. If the lock was confirmed, then everything is fine. And if the lock is still pending, then the mechanism stalls until the status of the optimistic lock is determined, retransmitting X-lock requests as necessary.

If the lock is denied, then the user data was transmitted with the ownership grant and the local host is the new owner, so the situation is set up to run efficiently when, in all likelihood, the lock attempt is repeated.

One note is that if synchronous invalidations are being used, then the true validity of the object can be determined simply by whether it is locally valid or not. If it is not, then it is extremely likely that the version of the object data available locally is out of date, so an optimistic lock need not be granted and a regular lock should be sought.

However, if synchronous invalidations are being used and the local host has valid readership of the object, then an optimistic X-lock can be granted and can be confirmed or denied using the same procedure as above.

Because of the costs and complexity involved in granting optimistic locks, they should be used only in certain situations, most particularly where enough computation is performed to hide the latency of verifying the optimistic locks. Otherwise, optimistic locking may be more expensive than regular locking, since regular locking does not need to verify valid objects. The optimal amount of computation to perform during this time would be that which takes the same amount of real time as acquiring a remote X-lock.

A typical use of optimistic locking might be the case in which a host is the reader of an object and wants to modify it. It acquires an optimistic lock locally and then operates on the object while ownership of the object is sought asynchronously. If the optimistic lock fails, then the speculative work is thrown out and the object data is

automatically recovered by the system. If it succeeds, then the latency of acquiring the X-lock from the time of the SvLock() call to the time of calling SvConfirm() is hidden.

4.5 Tagged Objects

4.5.1 Concept

Tagged objects are another means of hiding latency, by distributing values before they are accessed, and by avoiding global synchronization, at a cost of more memory usage. They were inspired by a paper on Versioned Objects [Feeley92]. A versioned object is an object that the user accesses by its version number. Each version is immediately distributed to all readers as soon as it is generated. where it is stored until needed. It is useful for producer-consumer data sharing in iteration-oriented distributed-data computations (typically matrix problems) and allows certain classes of programs to continue processing in a "critical path" data-dependency fashion without requiring expensive global synchronization (i.e., barriers).

An important semantic property of versioned objects is that individual versions of objects are *immutable*, so a version can be distributed with no need to maintain its coherency; it will always have only one value. A new version is created by obtaining a write lock on the object, updating its contents, and releasing it. A limitation of the Versioned Objects system is that it requires that exclusive locks (updates) be sequenced by the structure of the user program. When a user program attempts to lock a version of an object for reading, if the version is available locally, the lock will succeed immediately. If not, then the host will block until the version does become available. An eager-update protocol is used to distribute new versions to previous readers as soon as new versions become available, on the assumption that the readers will likely need them.

Tagged Objects is a slightly more general mechanism that is designed to be used with the DSDS system. It is based upon accessing data objects using user-defined integer tag values. In fact, the tagged objects can be thought of as a generalization of the DSDS mechanism, with regular objects working with only one (system-reserved) tag value.

4.5.2 Application Interface

The main primitive for using tagged objects is

```
void *ptr = SvLockTagged( void *vid, int lockType, int tag );
```

This primitive works the same as the familiar SvLock(), except that a tag version is also given as an argument. When this primitive is used to acquire an exclusive lock, the local host becomes owner of that tag of the object and can update the contents. The change affects the specified tag only, and all other tags of the object retain their old values. There can be an arbitrary number of tags of an object in circulation. If a tag is X-locked for the first time, then the tag is created at that time, and the initial data contents of the newly created tag are set to that of the most recent version of the most recently created tag anywhere in the system. Accessing the object data is done in the usual way.

When SvLockTagged() is used to acquire a shared lock on a tag, the object data for that particular tag is fetched and made available. If a tag is locked that does not currently exist in the system, then the requester will block until one is manufactured (X-locked and released).

The way this primitive is intended to be used is in a producer-consumer fashion. The producer X-locks an object using a particular (new) tag value, such as an iteration number (or any event counter), and then the consumer S-locks the object requesting a certain tag value, to retrieve the corresponding object data. There are

no race conditions since the consumer will block until the tag that it asks for becomes available. This scheme functions very similarly to message passing, but with a shared-memory paradigm and more freedom of data accessing.

An issue that the Versioned Objects paper does not address at all is version (tag) removal. Once a tag is consumed by its final consumer and is no longer needed for the computation, it should be discarded. However, the problem is that the system will not know whether or not a tag is going to be accessed later on in the program (halting problem), so it must therefore retain the tag data just in case. It is therefore necessary for the user to explicitly or implicitly tell the system when the program will no longer be needing an object, since otherwise an unlimited amount of "garbage" would accumulate, taxing system storage for no good reason. Two primitives are proposed to deal with this problem:

```
int err = SvPurgeTagged( void *vid, int tag );
```

```
int err = SvSetConsumeLimit( void *vid, int limit );
```

The SvPurgeTagged() primitive explicitly purges a tag of an object from the system. If the tag is ever used again, it will be as if the tag were being created for the first time. Care must be taken to avoid race conditions. The SvSetConsumeLimit() primitive is a more convenient implicit purging mechanism. When set to a certain value L, the object will be automatically purged after being consumed (S-locked and released) by L different hosts in the system. This latter mechanism will be sufficient for the most common producer-consumer sharing patterns of one (or more) producer and L consumers. One note is that SvSetConsumeLimit() applies to all tags of an object, so generally it only needs to be set once (when the object is created), although the object-global value can be changed at any time in case the sharing pattern changes. If it is changed, the change affects only tags created subsequently, not existing tags.

These mechanisms are somewhat prone to programmer errors since if the programmer forgets to use one of these mechanisms, the old tags will accumulate. Perhaps

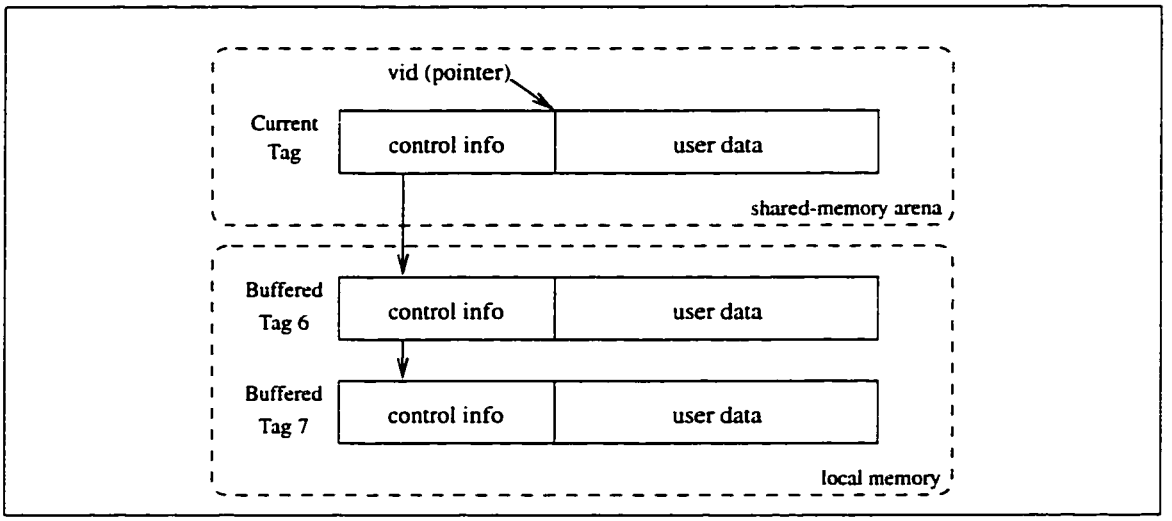


Figure 4.2: Storage Structure on a Host of an Object with Tags

some more automatic mechanism should be devised. Or perhaps disallowing the object from being used until a consume limit has been set for it (with a special value of zero meaning that the object will be purged explicitly) will suffice for this problem.

4.5.3 Implementation

This mechanism is implemented with extensions to the existing object-management protocol. A field tag is added to the object-control record that records the tag value for the current object. The main object (pointed to by the object id, a pointer into the global shared-memory arena) stores the control information and the user data for the tag of an object currently active at this host. There is also a linked list of subordinate tag objects descending off of the main object. The subordinate tag objects are stored in the local memory of the current host and the object data is stored contiguously immediately after the tag (even for "mapped" objects, which are introduced in Section 5.2.2). See Figure 4.2 for an illustration.

The subordinate tags are tags of the object that are not currently in use but which must be retained in case they are needed. The subordinate tags operate in some ways

as independent objects, with versions, ownership, readership, and reader lists, and in other ways they act as alternate incarnations of the main object.

When an S-lock (readership) of a tagged object is received by a host, the tag record is added to the subordinate tag list if the main object is active, or in the place of the main object if the main object is not active at the time of receipt. Then, when a tag is S-locked locally, the main object and then the tag list are checked for a matching tag. If a matching tag is found, the tag is copied into the place of the main object, and the previously main object is either discarded (if it is a reader version) or it is placed into the tag list (if it is the owner version). Read-only tags can be discarded at any time since they can be re-fetched from the tag owner when needed. Note that tags only need to be buffered if they are received *before* the consumer is ready to consume them. In all other cases, there are no buffering/data-copying costs.

If the S-lock cannot be satisfied locally, then a lock request is sent out in the normal way to the owner of the most recently modified tag (the equivalent of the owner of a regular object). That host is responsible for maintaining a list of the owners of all tags of an object that are active in the system. In the normal case that there is a static producer (owner) of tags, this list is obviated since that host will be the owner of all currently active versions. If the requested tag value does not exist (yet), a busy signal is returned to the requester.

To acquire an X-lock, the local host must become owner of the main object and owner of the tag. This means that it must acquire ownership of the tag from the tag owner and ownership of the main object from the owner of that, to give the tag to be modified the status of being the most recently modified object in the system. In the normal case that there is a static producer of the object, no messages need to be exchanged to fulfill this condition. If the tag value does not already exist, then it is created anew using the data contents of the (now acquired) most recently modified tag of the object. The X-lock can then be granted to the user program for tag-content modification.

Normally, when an X-lock is released, a synchronous update protocol is used to immediately propagate the new tag contents to all readers of the tag (or main object if the tag is being created). Thus, in the normal case, the new tag value will be sent to all expected consumers as soon as possible. If the reader is already waiting for the tag, then the user continues processing as soon as the tag arrives. If the producer is running ahead of the consumer, then the tag will be buffered in the subordinate tag list of the consumer (reader) until it is locked, at which time the tag will be acquired from its local memory. If an update protocol is undesirable for whatever reason, then it can be cleared using SvClearFlags(vid, SV_UPDATE).

For explicit purging, a tag can be purged from all readers simply by invalidating the readers and then it can be deleted from the owner. (A host must become the owner before explicitly purging an object.) For consumption-limit purging, when a reader releases its S-lock on the object, it internally performs the equivalent of an SvInvalidate() which purges the tag from the reader and communicates back to the owner to subtract one from the consumed limit. When the consumed limit reaches zero, the tag is purged in the same way as for explicit purging.

This mechanism is a bit complicated, but it allows producer-consumer programs to be implemented in a fairly direct fashion with a minimum of wasted effort and without global synchronization, since waiting for specific tags allows a data-dependency-driven style of synchronization to be used.

4.6 Protocol Issues

4.6.1 Exploiting Semantics

The semantics of locking and releasing objects and of the flow of ownership and readership are straightforward enough that in certain cases, they can be exploited to simplify programming. Essentially, data "stays put" even if it is not currently

locked, unless some external host performs operations on it. If the structure of the program guarantees that there will be no contention for an object during a certain period of execution, then the data of the object can be safely accessed without the data actually being locked.

A good example of this is data initialization. When an object is created, it is created so that the local host is the owner and the data is allocated in memory but is uninitialized. Therefore, the user program could initialize the object data without actually locking the object. This is very convenient with matrices that are composed of multiple objects (see Section 5.2.2). And it is likely also a safe operation, since object creation and initialization usually needs to be protected using a barrier or equivalent.

A primitive that allows the user to take advantage of object semantics is

```
void *ptr = SvGrab( void *vid, int lockType );
```

This primitive is the equivalent of an SvLock() followed immediately by an SvRelease(). Although this is conceptually a useless operation, it will fetch the required data to the local machine to be accessed without a lock. This primitive has the major advantage that it is only a single operation and not a lock/release pair in which the release phase may be accidentally left out. Although, this does introduce a danger that a correct program could be unwittingly changed into an incorrect program by forgetting about data that is being accessed outside of locking. For this reason, the best use of this facility is for data that is static during the run of a program, but which needs to be centrally initialized and then be disseminated to all hosts, for example, a list of the object ids that will be used for computations. If due care is taken, this primitive can also be used with objects that are protected with a barrier or equivalent.

4.6.2 Light-Weight Locking

The amount of CPU time spent on locking an object that is available in local memory is actually a significant issue for applications that lock objects very frequently. In fact, for some access-intensive matrix problems that use one object to represent every row, one-half of the total computation time of the program can be spent on acquiring and releasing local locks, for the basic system implementation.

The main reason for this is that the basic implementation uses Unix kernel facilities for acquiring and releasing locks for synchronization between the master and slave processes that implement the protocol. Kernel calls cause a large amount of overhead. A minor additional reason is that the code was written to be very general and the SvLock() and SvRelease() functions handle all possible cases. Thus, a conceptually simple, general implementation produces an unacceptable overhead in some circumstances. (Such situations must be watched for, particularly in low-level "systems" code such as this.)

To combat these problems, a different implementation has been created that uses a light-weight "test-and-set" style of locking which requires only a few processor instructions instead of a kernel call, and the locking is implemented as a macro that uses the light-weight locking and checks to see if the object is available locally for the required access. If it is, then the lock field of the object is set (thereby locking the object in the protocol) and the macro returns. If the object is not available locally, then the complete, heavy-weight function is called to acquire the lock remotely. The result is the elimination of the great bulk of the overhead spent on locking local objects.

4.6.3 Limiting Data Transfer

The primary issue for optimization of a distributed program is the number of messages sent and the timing of these messages on the critical path of computation. A

secondary issue which also deserves some discussion is the sizes of the messages sent. The CPU/real-time cost of passing a message in a distributed system is a fixed overhead for every message sent plus a small additional overhead for each byte of message data sent. If the number of bytes is large, this can become a significant cost.

In some cases, it is desirable to initialize an object at a host other than the host at which it was created. For this purpose, and more generally to provide efficient support for write-only locks, a bit flag of SV_NODATA can be included in the lock type of an SvLock() call to obtain a lock without any user data being included with the grant message. For larger objects, this cuts a large amount of unnecessary data transfer.

Another desirable facility would be support for arbitrarily large objects. Large objects can make working with large sections of matrices much more convenient than cutting logical partitions into many much smaller physical objects. The cause of this system limitation is the 64K maximum UDP packet size. What is needed is a better transport service, one which can handle arbitrarily large messages, but which can operate as efficiently as UDP for small messages. A better transport service is not provided, but using one would improve the system. One issue requiring care when dealing with large messages is flow control.

An issue associated with large messages is that sometimes only a small part of an object needs to be modified, and therefore, only the changed portions of the object theoretically need to be propagated to other hosts which already hold the correct data for the unmodified portions of an object. A primitive SvDirty() could be used when only small pieces of an object are modified, to identify the ranges of the object which are modified, using start-pointer and modified-size arguments. A small list of these ranges can be maintained with the object-control information that associates modifications with different versions of the object. Since lock-request messages include the version number that the requester last held, the granter can check the modification list to see which modifications need to be sent or whether the

whole object needs to be sent.

If implicit modification tracking would be more convenient, a flag could be set on the object indicating that "diffs" should be taken. Then, when an object is X-locked, a temporary copy is made of its contents and when the object is released, the new version is compared to the saved version to identify the "bands" of modifications. These modification records would be maintained as with the explicit mechanism. Note that we would not want the system to take "diffs" by default, since objects are very often modified in their entirety, and taking a "diff" is an expensive operation that would be completely wasted in this case. One should note that multi-writer DSM systems, such as TreadMarks [Amza96], *always* pay the computational cost of using the "diff-record" approach to track changes to VM pages.

At the other end of the spectrum from large messages, it would also be useful to pack multiple small protocol messages into a single UDP packet. For features such as prefetch links, this would allow the objects to be granted with much less operating-system and network overhead. This could be useful with asynchronous consistency mechanisms in that messages destined for the same host could be accumulated before they are sent out. And as a user-controlled mechanism, the user could explicitly say that messages resulting from certain operations (such as SvLock() and SvGrant() should be accumulated in a similar way, by adding the bit flag SV_GROUP to the lock-type fields and then executing an explicit function to flush all grouped messages.

4.6.4 Object-Owner-Locating Cost

There is a fairly simple way to impose a highly probable upper bound on the time it takes to track down the owner of an object when pursuing a remote lock. Let us imagine a DSDS system with dozens of hosts in it, and say that we want to impose a limit of five misses for any of the hosts attempting to locate an object.

Each object in the system can have a "locator" host for it, determined by a simple

hashing function. If it would not be too much of a bottleneck, it would be convenient if the locator host for all objects were the coordinator host, since it will always have object records initialized for all objects, although this is an easily overcome concern.

If we wanted a constant bound of (about) two accesses per attempt, we could simply send a message to the locator machine every time that ownership of the object moves, informing the locator machine of the new owner. Then, every time that any host in the system wants to access the object, it sends the lock-request message to the home machine for the object, and the home machine forwards the request in the usual way. Whether this approach in itself would be best or not depends entirely upon the behaviour of applications. Objects that move frequently and which are accessed by many hosts would benefit, but objects that either move infrequently or which are accessed only by a very small group of hosts would likely work more efficiently under the request-forwarding scheme.

As a compromise, we can use both approaches, with a tuning parameter to select more of one and less of the other as appropriate. The tuning parameter (H) controls how often the locator machine is informed and how often to consult it.

Given the nature of the object-management protocol, messages may be lost and an object may move while a host is seeking it, and each of these problems can can lead to arbitrarily long searches for the location of an object. (Please note that only the loss of a specific type of message will cause problems with the mechanism below.) But, if specific messages are not lost and the object does not move while it is being sought, then we can guarantee that at most H steps will be required to locate the object. The value of H determines the tradeoff between the cost of updating the locator host and the cost of locating the object.

Each lock-request message already has a "hop-count" field. When an intermediate host receives a lock request for an object that it does not own, it increments the hop count and then forwards the lock request. All we have to add is a test so that if the number of hops so far is equal to $\frac{H-1}{2}$ then that intermediate host forwards the

request to the locator machine for the object rather than its best guess of the current owner.

To ensure that the locator machine is proportionately up to date, we add an array of $\frac{H-1}{2}$ integer entries to the object-grant message (and the in-core object record). This array will contain host numbers: the $\frac{H-1}{2}$ unique hosts that were previously owner of the object. Each time that ownership is forwarded, the host that is forwarding the object checks to see whether the new owner is already in the previous-owner list. If it is, then no special action is required and ownership is forwarded as usual.

If the new owner is not in the previous-owner list, then a short message is sent to the locator machine informing it of who the new owner is. Currently, this can be accomplished by sending a "Busy" message to the locator machine. The reason to update the locator machine from the granting host is so that the location-update and grant messages will arrive at their destinations at approximately the same time, keeping the information reasonably "in sync."

After sending the location-update message, the granting host picks some host to remove from the list of previous owners (using LRU or whatever), adds the new host, and then grants ownership to the new host. This owner-history array acts as a cache, to minimize location-update messages to the home machine in case the object just bounces around the same few hosts.

Given the assumptions that no locator-update messages are lost and that the object will not move while it is being sought, the above results in a maximum of H hops to locate an object. A request will be sent to the locator after $\frac{H-1}{2}$ hops, and the forward to the locator is one additional hop. After this, the locator forwards the request and the owner will be located within $\frac{H-1}{2}$ hops, including the forward from the locator, because the set of possible hosts has cardinality $\frac{H-1}{2}$. Thus, adding up the hops ($\frac{H-1}{2}+1+\frac{H-1}{2}$), we have demonstrated that at most H hops will be required to locate the object. The value of H needs to be set according to characteristics of the object movement and the desired efficiency in locating an object. If the object

frequently moves among a small set of hosts, then we need to set H larger than the size of that set.

4.6.5 Reader Lock Granting

It is possible and advantageous to allow a reader of an object to grant shared locks to remote hosts. If a host wishes to acquire a read lock, it will send a request through the forwarding chain to the owner, as only the owner currently can grant read access. However, it may be the case that the request is forwarded to a host that is currently a reader of the object before the request reaches the owner. The reader has available all of the information needed to grant a read lock (the up-to-date object data, version, etc.), so it would be an optimization for it to do so.

The only problem is that the owner needs to be consistently notified of the new reader so that the new reader can be invalidated when the need arises. Alternatively, the reader could keep track of readers it has granted access to and invalidate them when it receives an invalidation request from the owner. This would create a hierarchical system of invalidation.

4.6.6 Deadlocks

Very briefly, the DSDS system makes no attempt to detect or recover from deadlock situations, and they can indeed happen unless the user program is careful to avoid them. The use of "sticky" objects with prefetching can also lead to deadlocks (although a maximum holding time could be used there). Tanenbaum [Tanenbaum95] indicates that deadlock prevention and detection can be applied in distributed systems, but prevention is not applicable at the level of DSDS—it would need to be applied in user-level code. Also, deadlock detection is complicated and expensive, and thus, ignoring the problem "is as good and as popular in distributed systems as it is in single-processor systems."

However, if a deadlock-detection mechanism were to be added to the system, it could wait for a long delay of, say, 30 seconds for trying to lock an object, and then a host could attempt to construct a consistent global "waits-for" graph to look for a deadlock. If there is one, then the user operation, for example, SvLock(), would return a failure.

4.7 Conclusions

This chapter proposes lower-level performance enhancements for the basic DSDS system that are based in most cases on hiding latency in the management protocol. Many techniques were proposed, including prefetching, non-stalling invalidations, optimistic locking, "tagged objects," and many protocol enhancements. All of the prefetching methods are implemented in the prototype system, as are stall-on-lock and stall-on-next-operation invalidation, light-weight locking, nested locking, and exploiting semantics. Optimistic locking, tagged objects, other methods of non-stalling invalidation, an update protocol, broadcasting, and some low-level protocol extensions are not implemented in the prototype.

A variety of techniques is required to achieve good performance across a wide variety of applications. Although many techniques are introduced in this chapter, they are largely orthogonal to each other and the original system design was sufficiently clean and flexible that adding them presents no fundamental problems.

Prefetching has been a very popular technique in many systems because it has the potential to eliminate data-request latency. The logic is fairly simple. If a programmer can arrange for the data his program needs to arrive shortly before it is actually needed, then he saves the time of fetching it from a remote source. However, there are issues to be careful of. Prefetching too much could cause network/buffer-congestion problems, and prefetching items that will not actually be required can cause the previous problems plus wasted overhead. The programmer typically needs to decide

when prefetching will be useful.

Five methods of prefetching are examined in this chapter. Prefetch links are statically set up in a data structure to fetch a list of objects whenever the first object is fetched. This mechanism provides a great opportunity for improving object-migration performance, since one request is sent and numerous additional requests can be avoided. The internal mechanism can also be tuned to give maximum throughput without overloading the network or the receiver. There are limitations to this mechanism, and care must be taken to either re-link the prefetch chain or to cut it if and when the semantics of a group of objects changes.

The explicit prefetch provides a smaller potential gain than prefetch links, since there is one request for each object granted; however, it may be more generally applicable, since one can be inserted at any point in the program where some time-consuming operation is performed before the object is needed. There is no real wasted effort to this mechanism (with the right implementation guard).

Asynchronous granting is applicable where object flow is known, and it allows a programmer to code in essentially a message-passing fashion where it is appropriate and helpful. It is equivalent to a Send(). Pre-invalidation is useful for temporary readers which just need to quickly read the object. It avoids the invalidation phase of the protocol. Local invalidation is similar to pre-invalidation, but it saves message passing altogether. It is useful for barrier loops.

Non-stalling invalidations also provide a means of avoiding synchronization delays. Waiting to collect all invalidation acknowledgments when acquiring an X-lock on an object is called stalling. Stalling on a lock operation was the semantic of the basic system, but it can be safely extended to stalling on the next operation, since no action visible to the outside world can occur in the interim. If there is significant computation in the time between acquiring an X-lock and the next operation, then the invalidation latency is hidden. It is also possible to stall on the next release, but this only guarantees a weaker per-object consistency. Or, the invalidations can

be made completely asynchronous so that they are only collected at specific times, such as during a global synchronization of a computation. There is no loss if strict consistency on the objects involved is not needed in the interim.

Stress testing of the basic system was performed using the random-walker algorithm discussed in Section 4.2.2. It is a simple test, but it does test multiple aspects of the locking protocol and can be configured to provide extreme circumstances including a high degree of access contention, a high percentage of dropped messages, and a very short time-out period. The random walker helped to confirm that the object-management protocol works correctly and the testing provided a basis for devising system performance enhancements. The main performance testing of the system with realistic applications is undertaken in Chapter 6.

The basic protocol can also be extended to provide an update service on selected objects. This means that when an object is updated, an auto-grant is sent to the previous readers. If the stall-on-next-operation invalidation semantics are in use and the next operation is the release of the X-lock, which is actually fairly likely, the update message can double as an invalidation request. Broadcasting and/or multicasting can be used, if they are available, for auto-granting with no protocol difficulties. Because of the nature of the protocol, the broadcasting facility does not require any special ordering, atomicity, or reliability properties in order to work correctly. Objects can be selected to use either an update or an invalidation protocol and this property can be changed dynamically.

Optimistic locking is another mechanism intended to hide latency. It requires that an update protocol be in use with asynchronous invalidation. Optimistic locking allows locks to be acquired without the local host being sure that it has the most up-to-date version of the data. This is confirmed asynchronously, while user processing continues. This mechanism avoids synchronous delays. Tagged objects can also eliminate unnecessary operations in certain application areas. Specifically, in iterative algorithms where successively refined versions of an object (such as a sub-matrix of

Step	Host A	Host B	Other Hosts
initial state	reader	owner	some readers
1	Lock request		
2		Lock grant	
3			Sequential invalidations
4	Use of object		
5			Some read request
6			Some ownership request
7	Object release		

Table 4.1: Conservative Object-Locking Sequence

a matrix) are computed, all of the usual invalidation and iteration-synchronization mechanisms can be avoided.

These methods for hiding latency can be described as modifications of the sequence of events in the conservative locking model. Table 4.1 presents a typical sequence for conservative locking. Conservative locking requires steps 1, 2, 3, 4, and 7, in this order. The various forms of prefetching allow step 2 and potentially step 3 to occur before step 1. The two forms of pre-invalidation allow step 3 to occur before step 1 or not at all. Non-stalling invalidations allow reordering of events as follows. Stall on Lock is the same as conservative locking. Stall on Next Operation allows step 4 to occur before step 3. Stall on Release allows steps 4 and 7 to occur before step 3. And Asynchronous Invalidation allows step 3 to occur at any time after step 2. Optimistic locking allows step 4 to occur before step 1 and requires a confirmation step before step 7. Tagged objects allow step 2 to occur before step 1 and typically eliminate steps 3, 5, and 6, although the main benefit of eliminating global synchronization is not captured in the model of Table 4.1.

Many of these methods of hiding latency can also be applied to other distributed-

shared-data systems. Prefetching mechanisms can be applied to object-based systems using explicit methods on objects for transport and to DSM-based systems using special no-operation memory-referencing processor instructions [Mowry94]. Invalidation stalling and trading consistency for performance apply to all systems. Optimistic locking and versioned objects apply mainly to object-based systems, since the semantics revolve around specific objects. Update/broadcast is applicable to all systems, and most use it.

Many lower-level protocol mechanisms are introduced also, for more efficient operation. It is possible to exploit the semantics of the system to avoid the bother of locking objects when it is not necessary. The system can also provide light-weight locking, in which the weight of the Unix-kernel semaphore/process mechanism is avoided in favor of a processor mutex operation and a few integer operations, for the case that the object is available locally. Approaches to handling object-owner-locating costs and of allowing reader lock granting are also introduced.

Chapter 5

High-Level Mechanisms and Usage

5.1 Motivation

The previous chapters introduced a number of concepts and primitive mechanisms that the DSDS system provides, but it is not necessarily obvious how those mechanisms should be used to write real programs, and some mechanisms can be abstracted to make distributed programming more convenient. This chapter will describe high-level mechanisms that extend the basic DSDS mechanism. Unlike the previous chapter, in which improving performance was the goal, the goal here is to improve programmer convenience. Because programmer convenience is less easily determined than performance, the arguments in favor of these mechanisms are necessarily informal.

This chapter also demonstrates that the primitives introduced in Chapter 3 can be used to create higher-level mechanisms. In this sense, this chapter defends the contents of Chapter 3. The primitives from Chapter 3 can also be used to create other higher-level mechanisms that are not considered in this chapter.

This chapter proposes mechanisms and approaches for matrix processing and linked-structure processing, and proposes higher-level programming interfaces. Pro-

grammers also need to know how to make the best use of the DSDS system, so this chapter also discusses how to use DSDS most effectively and the programming equivalences between other systems and DSDS.

5.2 Matrix Processing

Matrix processing is a natural use for a distributed-shared-data system and a common application of parallel and distributed systems in general. The most common type of matrix encountered is a 2-dimensional one. However, issues of specifying matrices concisely and accessing and sharing them conveniently and efficiently need to be explored further. For convenience, we want to program in a style that is very much like sequential matrix processing, and for efficiency, we want the approach to give the best performance possible for a network-of-workstations environment.

In this thesis, matrix multiplication is frequently used as an example of matrix processing because it is a very simple, well understood matrix algorithm.

5.2.1 Data Organization

Given that the runtime system is intended to manage independent data objects, it may be obvious to split the data of an application up into these independent data objects. For a matrix, this usually means dividing the data up into "tiles," where each tile is a sub-matrix of the overall matrix, storing the data for the cells starting at a certain set of matrix subscripts. For example, if you had one big matrix and four processors that independently processed one segment of the matrix, you could split the matrix into quadrants and make each processor responsible for processing one of the tiles (quadrants). See Figure 5.1 for examples of dividing up a matrix. Discussions of real applications and good ways to divide matrices into tiles are presented in Chapter 6.

An important issue with distributed-matrix algorithms in general is what sizes and

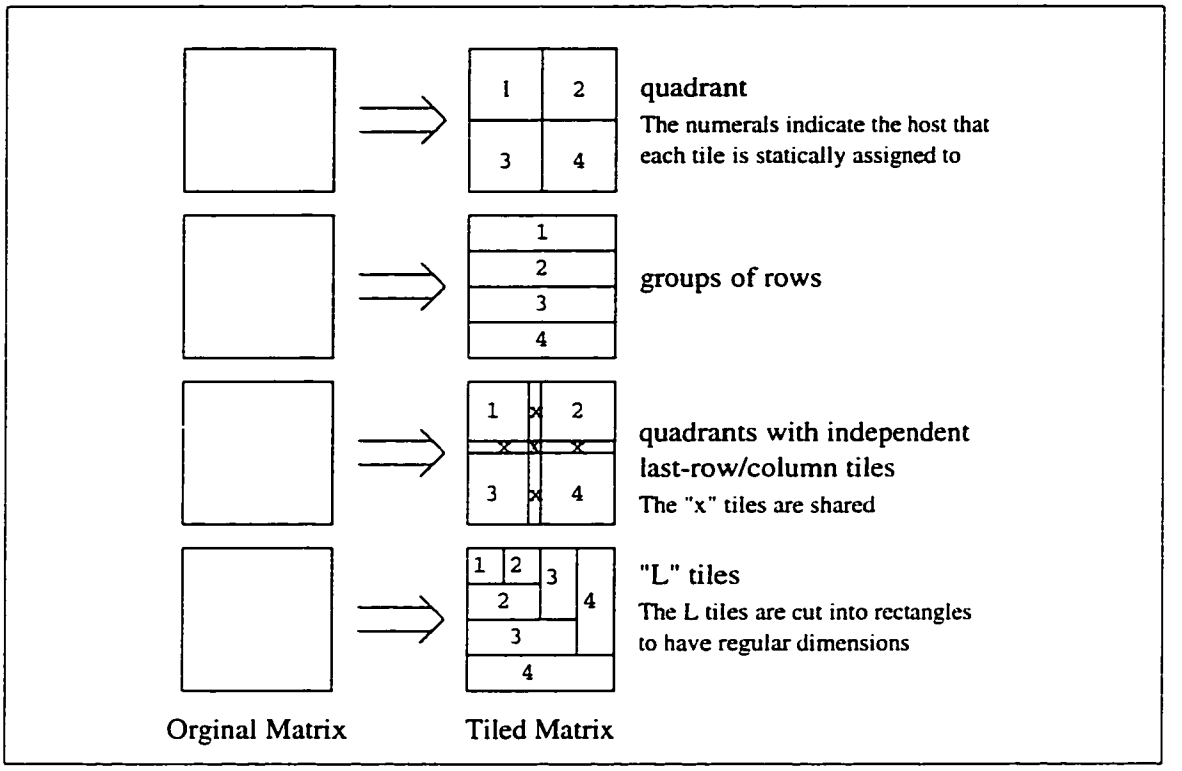


Figure 5.1: Examples of Dividing a Matrix into Tiles

shapes of tiles to use, although in many cases, given the number of hosts to be used, it may be fairly obvious how to divide up the matrix. However, the prototype DSDS system does not allow individual data objects to be larger than 64K bytes (minus control information), so multiple tiles will likely be needed to "cover" the ranges of the distribution segments. Grain size is a general-case issue, since there will be a non-zero amount of overhead to store each object. It makes sense to allocate multiple matrix elements inside a single shared object. (In fact, storing one element per object would be extremely wasteful if matrix elements are no bigger than the control information for each object.) It also makes sense to make the objects inside an object spatially adjacent to one another. Since most matrix algorithms scan through matrix elements sequentially, this will give good caching effects.

When the components that a matrix naturally divides into are divided into even smaller DSDS objects, the issue arises of object locking versus larger-structure locking. Arbitrary subsets of objects that form the larger structure are allowed to be locked by the application. In this situation, two general approaches for dealing with these objects make sense: either locking the larger structure by locking one object at a time, or locking only the objects that are needed by the inner loops of a matrix application when they are needed. It is often the case that there is no access contention between most of the larger components of a distributed matrix in distributed-matrix applications (this can be seen in the examples in Chapter 6). This means that it is often the case that there are no deadlock issues in locking the objects of the larger structure in any arbitrary order. However, deadlock prevention can be implemented in user code, for example by always locking objects in the same order.

In general, possibly the best way to divide up a matrix is row-wise, unless this fundamentally contradicts the way that the matrix is going to be used. This gives good caching effects, since processor cache lines are organized for accessing sequential words of memory, and compilers tend to be good at performing optimizations for tight loops of sequential matrix accessing. This also makes the mapping very simple

to understand and to program for.

However, in many algorithms, there will be accessing contention between processors for (some) common matrix elements, so this needs to be taken into account. Using smaller tiles will reduce accessing contention. A special kind of accessing contention is a fixed-sharing pattern, in which case dividing the matrix into irregular tiles which have slightly different sharing/location attributes may be best.

5.2.2 Possible Approaches

There are many different possible approaches for the distributed implementation of matrices. There is a naive-object approach, a tile-oriented approach, a matrix-cursor approach, and a mapped-matrix approach. For many matrix problems, the mapped-matrix approach is the best of those we have considered, as demonstrated in Chapter 6, but the other approaches will be discussed for contrast.

Naive Approach

A problem with using independent data objects comes into play when we need to perform the computations. Sequential code is written to directly access elements of matrices, but the distributed-case elements are dispersed among many different machines in tiles. The naive way to "pave over" the implementation details is to write functions that read and write individual cells of the matrices. These functions take the subscripts as arguments, calculate and locate the object to lock, lock it, perform the operation, release the object, and return the results. This type of interface may be the most straightforward means of providing an object-oriented or Linda-type system, to hide the implementation details or to fit the distribution model.

Unfortunately, regardless of the convenience of use, the naive indirect-access approach performs *very* inefficiently with the DSDS system, and very likely with any other distributed system. A simple test was performed multiplying two 1024×1024

matrices on four processors using the naive-accessing approach, which performed approximately 114 times slower than the corresponding sequential implementation. The processing overhead of converting the matrix indices to tile vids and then accessing each element of the shared objects through the DSDS interface seriously limits the performance. In the sequential version, the data-accessing overhead is little more than a memory reference. The caching of shared-object locks and the liberal use of macros in the source code would probably improve the distributed performance substantially, but a different approach to the problem is required to make DSDS outperform the sequential version. The DSDS organization includes a significant amount of overhead for handling locks compared to a single memory reference.

Tile-Oriented Approach

A different approach, which involves substantially changing the loop-control code, is the tile-oriented approach. In this approach, the matrix is tiled as set forth above, with tiles containing arbitrary numbers of rows and columns. The program has two levels of looping, tiles and cells. For the example of matrix-multiply, the program loops through the tiles of all matrices needed in a computation in a specified order. This is the outer loop. Then, for each set of tiles required to compute results, the tiles are locked and the cells of the tiles are scanned in tight loops to produce the results for the tiles. This is the inner loop. An illustration of processing for matrix multiply is shown in Figure 5.2. Note that the tiling of input matrices is not vital here, but input matrices will be too large to fit into a single object in many cases: in real applications, the input matrices may be the output matrices of some other matrix algorithm that needs to tile them a certain way; and in the case of matrix multiply, only parts of the input matrices are needed at each host, so tiling them limits data transfer.

This approach produces reasonable performance results (compared to the distributed results in Chapter 6), as objects are used completely when they are locked

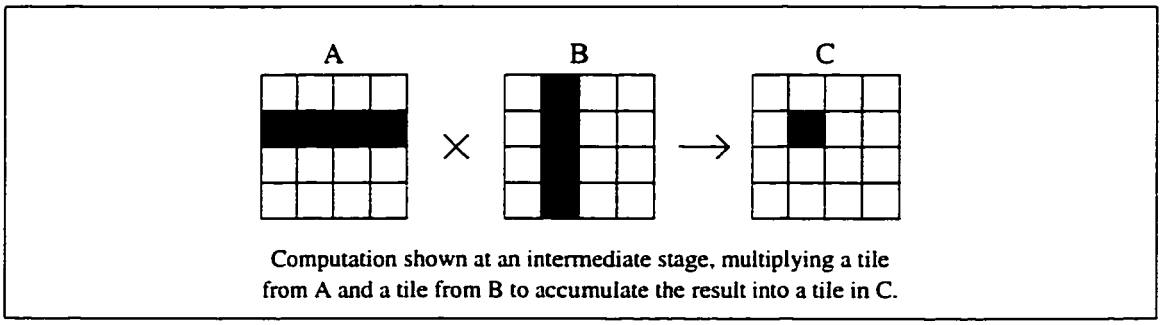


Figure 5.2: Tile-Oriented Approach to Matrix Multiply

and there is a sufficient amount of computation to amortize the cost of each lock. However, in our experience with writing test applications, we found this approach to be both difficult to understand and to program.

Matrix Cursors

The Matrix-Cursors approach is a compromise between the naive and tile-oriented approaches. It uses the concept of a "cursor" that the user positions into a matrix to access elements through, and then he moves the cursor one position at a time in the direction of the next element to be accessed. This requires a set of library calls to maintain the cursor position and to perform object-locking management underneath. Locks are cached and only change when the cursor moves off the edge of one tile and onto another. The user is isolated from managing objects and locks, as instead he only needs to deal with matrix subscripts, but our experience with this approach is that it is not much more convenient than directly locking objects, since the cursors must be repositioned for every element.

This approach basically provides the functionality of individually locking each matrix element, but uses macros to scan elements and through them caches locks so that objects are locked and released only occasionally. As expected, the performance of this approach is somewhere between the efficiency of the two approaches, slower

than the tile-oriented approach because of the overhead of the extra operations, the relatively expensive indirect-memory-accessing costs, and the lack of optimization of tight loops because of the complexity of the macro instructions.

Mapped Matrices

Using mapped matrices is probably the best of the approaches we have considered for all matrix algorithms. This approach uses Mapped Objects, which are described in detail later in this section. Basically, they are a minor extension to the system to allow the user-data portion of a DSDS object to be mapped onto an arbitrary submatrix of a programming-language matrix allocated in contiguous memory in the global shared-memory arena. The binding of the mapped object to a specific submatrix is performed when the mapped object is created. Locking a mapped object brings the corresponding matrix-tile data in local DSDS memory up-to-date (w.r.t. the consistency protocol) and locks it according to the usual semantics.

Using mapped matrices is mostly as convenient as the naive approach and it is, from our experience in writing test applications, the most efficient of the approaches we have considered. The reason that it is only "mostly" as convenient is that locking and accessing are decoupled, so objects need to be locked independently and potentially at awkward times w.r.t. accessing data elements. For example, depending on the application, if a matrix were divided into tiles of 256×16 cells each, then a new element would need to be locked every time some iteration-control variables reached values evenly divisible by 256 and some others reached values evenly divisible by 16. On the other hand, the matrix elements can be accessed directly using standard programming-language constructs, so this can be expressed simply and be well optimized by compilers.

However, locking and accessing can often be more conveniently linked, by performing locking on a row-wise basis, which often conveniently matches each iteration of an outer loop that sweeps a matrix. On the other hand, in applications where there

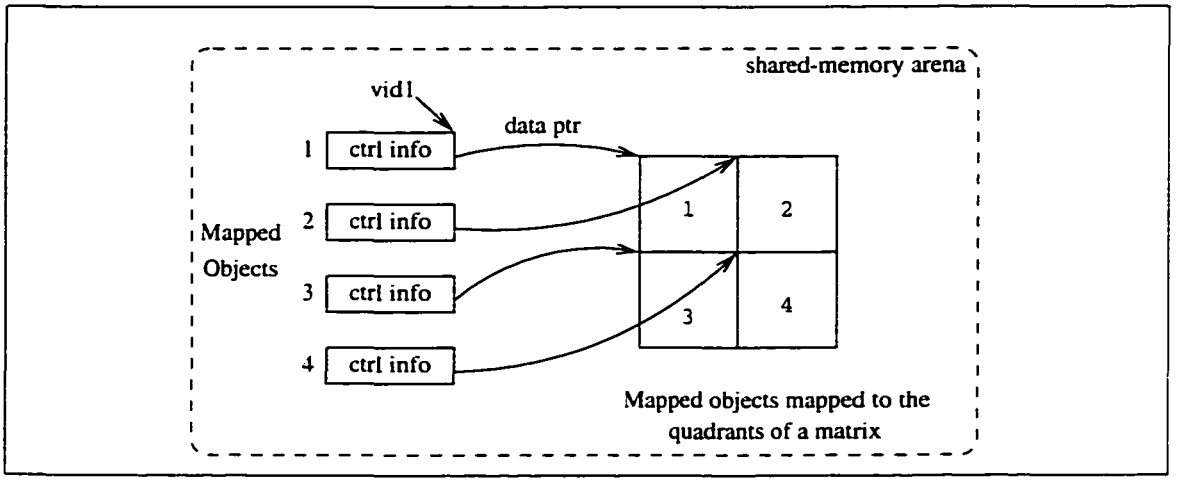


Figure 5.3: Structure of Mapped Objects

is no accessing contention for objects that are assigned to be processed by only one host, locking and computation can be carried out in different phases. This approach decouples a potentially awkward interleaving of tile locking and element accessing, so tiles can be arbitrarily shaped and still be convenient to work with. Mapped objects can also, of course, be locked arbitrarily like regular objects.

Mapped-Object Primitives

Mapped objects are DSDS objects whose user-data component is mapped to a submatrix of a contiguous matrix stored in the global shared-memory arena. An illustration of this mapping is shown in Figure 5.3. Some minor extensions to the basic DSDS systems are required to implement mapped objects. The system primitives that are used to set up mapped objects are described below.

```
void *vid = SvCreateMapped( void *ptr, int bandsize, int ileave, int count );
```

This primitive creates a mapped object. A mapped object operates identically to a regular shared-data object, except that the memory that the user data is to be stored into is allocated with SvMalloc() (below) before the mapped object is created.

and the mapped object is made to point to the pre-allocated memory. The main reason that this type of object is useful is that programming languages and compiler optimizers work best when matrix data is operated upon and stored in a certain way: in row-major contiguous memory.

This type of object can be mapped to sub-matrices of a 2-dimensional matrix. The mapping is specified using four arguments, including the pointer to the first byte to be mapped (ptr), the number of successive bytes to be included in the first "band" of the mapping (bandsize), the total interleave (ileave) between bands to be mapped, and the number of bands (count) in the mapping. Each of the four submatrices shown in Figure 5.3 have multiple interleaved bands of included matrix elements.

For example, if a two-dimensional 128 × 128 character matrix were allocated in shared memory and an object were to be mapped to the top right-hand quadrant, the parameters would be: ptr=address of element at subscripts [0][64], bytes=64, interleave=128, count=64. (This must follow the mapping of matrices in the target language, in this case C.) To map to simple linear memory, the interleave value is ignored and can be passed as zero (0) and the count argument must be one (1).

Conceptually, there is no system limitation against mapping to tiles in 3D or higher-dimensional matrices or even arbitrary structures, but this call only has the parameters for two dimensions, since this is considered to be the most useful. For higher-dimensional matrices, additional pairs of interleave and count arguments are required for each dimension. For arbitrary mapped structures, which probably are not very useful in practice, a list of groups of parameters for N-dimensional matrix mappings would be needed.

The system does not prevent the user from mapping overlapping regions with multiple mapped objects, but the system provides no guarantees about the consistency of the overlapping region. There is a possibility for the system to automatically handle the proper locking of overlapping mapped objects (after being informed by the user of which other objects the current one overlaps with), but this is not implemented in

the prototype system. Overlapping locking would probably be most useful if used in a hierarchical fashion, to address data-granularity issues.

The vid returned by this function serves only a single purpose, to identify the object. Unlike the SvCreate() primitive of Chapter 3, this vid cannot be used to access the data, although in practice, the user will have a more convenient way of accessing the user data anyway, such as with matrix subscripts. These objects are destroyed in the same way as regular objects.

```
void *ptr = SvMalloc( int bytes );
```

This primitive allocates contiguous memory for a matrix in the global shared-memory arena, to be used with mapped objects. The bytes argument tells how many bytes to allocate and the call returns the pointer to the first byte allocated, or a NULL value if the call fails (from running out of memory). Unlike the SvCreate() primitive, there are no consistency or coherency semantics implied for the allocated memory itself.

```
void SvMemFree( void * memptr );
```

This primitive frees memory allocated with SvMalloc(). It is analogous to the C-library function free().

5.2.3 Matrix Creation

The process of creating a mapped matrix involves a number of steps that could be tedious to code manually but which fortunately can be performed automatically. The parameters needed to set up a matrix are shown in the following function prototype:

```
void *matHdr = MatCreate( int nrows, int ncols, int trows, int tcols,
                          int elemSize, int flags, int wellKnownMatId );
```

The parameters include the total number of rows and columns in the matrix (for a 2-D matrix), the tile dimensions (assuming regular tiling), the byte size of a data

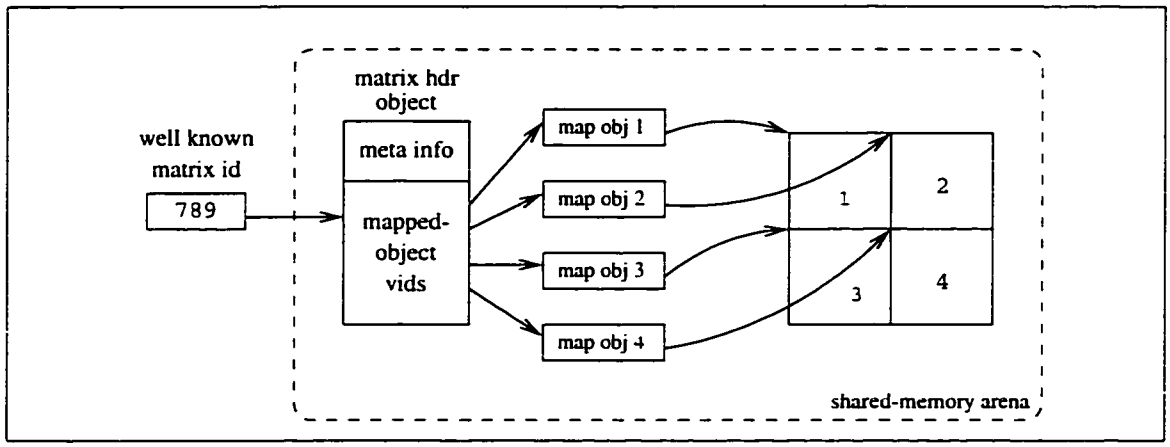


Figure 5.4: Structure of an Automatically Mapped Matrix

element, and flags that can be used for control parameters of the created objects. The final parameter gives a unique identifier to the matrix that allows remote hosts to locate it. The function allocates the global memory required for the matrix, creates and maps all of the mapped objects for locking the tiles, and create a "header" object that includes all of the relevant metadata for the matrix plus a list of all of the mapped-object ids to handle tile-naming translations. The resulting structures are illustrated in Figure 5.4.

Since objects are allocated dynamically in the DSDS system, a means is needed to tell all hosts the names of the objects, and to give them convenient names. The name registry provided with the system (the SvRegister()/SvLookup() mechanism from Chapter 3) can be used to solve this problem, but it is not intended for such volumes of names as would be required for a tile list. Instead, a simple user-level solution is to create and use the header object for the matrix that contains a matrix of the vids (object ids) of the tiles of the different matrices. The header object also contains other metadata for the matrix, such as the dimensions. This object is registered with the global name registry using the wellKnownMatId, and all remote hosts read it after the initialization phase of the program. With the information in the header object, the tile id for any element in the matrix can be easily calculated.

The above primitive would be executed by one host (the coordinator for the matrix) and must be guarded against being executed by multiple hosts, and another primitive is needed to look up the header object for the matrix. Or, with additional mechanism (an internal barrier), all hosts could execute this primitive, only one would perform the actual initialization work (usually host #1), and the other hosts would wait, and then all hosts would receive back the header id.

One thing to note is that a more complex mapping notation can be developed. There is no requirement that the tiling be regular. Irregular tiling can be useful for applications in which, for example, the main bodies of segments are used only by one host, but the border rows and columns are shared with other hosts. Also, programming-language extensions could be provided to declare distributed shared matrices statically.

5.2.4 Matrix Accessing

There are two levels to accessing mapped matrices: the element level and the tile level. The element-level read and write operations are handled by the compiler, but there needs to be additional mechanism to lock and release the tile objects during computation. The most direct way to do this is to require the user to make a primitive call every time his program is about to access a new tile. This call would be of the form

```
int err = MatTileLock( void *matHdr, int row, int col, int accessMode );
```

A corresponding Release primitive would be needed also.

The row and column arguments are subscripts for the matrix (rather than of the tile, for convenience), and the function would compute the tile indices and obtain the tile object id from the translation table in the matrix header object. Then, the usual lock/release operation would be performed on the tile and the matrix elements would be ready for accessing in their correct matrix positions.

In a programming-language/compiler environment, there may be enough information available to automatically insert the tile-locking calls for simple loops. This would require special declarations of the matrix tiling and index-loop analysis.

5.3 Linked-Structure Processing

5.3.1 Link Cursors

The basic design of DSDS makes it straightforward to implement linked dynamic data structures. What is needed for this purpose are "pointers" that efficiently identify data objects in the system and which can be dereferenced to access the data inside the objects. The basic primitives of the DSDS system provide this functionality directly and the system hides the protocol that manages the necessary data migration.

Another important issue in a distributed environment, beyond access to the shared data, is concurrency control. DSDS provides a conceptually simple mechanism for concurrency control in user-linked structures: per-object locking.

With the DSDS object-locking primitives, DSDS also gives the user the responsibility of making sure they are used properly and without error. This may be a tedious, low-level coding responsibility. To help alleviate this tediousness, the "cursor" mechanism can be applied here. The idea is to scan around inside the linked structure using the cursor as the current pointer, and let the system manage lock acquire/releases automatically as the cursor moves. This mechanism is similar to "iterators" that are used with abstract data types but this mechanism adds the object locking. Sample code using a cursor is shown in Figure 5.5. A more customizable language such as C++ would make this feature much "cleaner" to provide, as shown in Figure 5.6.

A cursor is a data type that has an internal structure and operations (methods) for creation/initialization, repositioning, flushing, and destroying. When a cursor is

```
CURSOR cursor;
USEROBJ *p;

CursorInit( cursor, p, SV_XLOCK, /*chain*/1, SV_NOFWD );
CursorPos( cursor, headPtr );
while (p->nextPtr != NULL) {
    p->value += 1;
    CursorPos( cursor, p->nextVid );
}
CursorFinish( cursor );
```

Figure 5.5: Link Cursor example in C

```
CURSOR<USEROBJ> p (SV_XLOCK); /*chain=1, SV_NOFWD default*/

p = headPtr;
while (p->nextPtr != NULL) {
    p->value += 1;
    p = p->nextVid;
}
```

Figure 5.6: Link Cursor example in C++

"flushed", it remains active (in scope in C++) but releases any lock it may be holding. A convenient way to flush a cursor is to assign a NULL object-pointer value to it. This is convenient because some dynamic data structures are naturally terminated by a NULL pointer. The C example contains about as much "baggage" as the direct DSDS implementation, since the cursor operations need to be called explicitly and since two "pointers" (the cursor and the data pointer) need to be maintained, but the C++ version is identical to the sequential version, except for the cursor declaration, since the cursor operations can be implied.

The implementation details are straightforward enough that they do not need to be discussed in detail, except for the locking semantics. The mechanism locks the object that is currently being pointed to and releases that lock when the pointer advances to a new object. Auto-invalidation and auto-forwarding features can also be added to cursors, as shown in Figures 5.5 and 5.6.

5.3.2 Concurrency Control

Concurrency control is required when working with user-defined distributed linked structures. This is, of course, not required for sequential algorithms, but it is for parallel algorithms, so the object-lock-management code required for DSDS is similar to that required in parallel systems.

The DSDS mechanism provides a simple concurrency-control mechanism, object locking, but the question is whether it is enough for processing linked dynamic structures. It is enough to guarantee single-object consistency, but we need to guarantee the consistency of the entire linked structure.

One method to achieve this is to use an auxiliary object as a semaphore to serialize accesses to the entire data structure. This is simple and effective, but it disallows actual concurrency in accessing the data structure. A slightly better mechanism would be to use the shared/exclusive locking of DSDS objects to lock the auxiliary in

the mode of the access to the linked structure. This would allow shared accesses to be performed concurrently. This may be good enough for many purposes, but exclusive accesses would block out all other accesses, even if these accesses do not truly conflict.

A fairly straightforward concurrency-control mechanism useful for many linked dynamic structures would be merely to prevent other, conflicting processes from "passing" the local process while traversing a linked structure, but while allowing non-conflicting remote processes to pass and allowing conflicting processes to access the objects that are "behind" the local process or which branch off into an independent traversal path.

The beauty of this concurrency-control method is that it integrates very naturally with the link-cursors mechanism. The original design of that mechanism stated that, when a cursor moves from one node of a structure to the next, the lock on the old node is released and then the lock on the new node is acquired. Instead, we simply alter this mechanism to acquire the new lock before releasing the old one. This will prevent other conflicting processes from "passing" the local process in its traversal of the structure. Because one lock is always maintained, no process with a conflicting lock can "catch up" to the local process, since the remote process will be delayed until the local process releases its furthest-behind lock. This method of moving cursor locks is required because releasing the previous lock and acquiring a new one is not an atomic operation in the DSDS system.

However, one can imagine unusual cases in which this may cause problems, such as two processes scanning a list in opposite directions where only single-object consistency is required. When the two met, they would deadlock. The way around this situation is to make the "always hold one lock" rule optional. This is provided with the "chain" argument. To be more general, a number is given rather than a boolean. The number gives the minimum number of locks to hold at all times (except when beginning from an empty cursor). (One plus this number is the maximum.) A special value, SV_INF means infinity, to hold locks all the way back to the root. The default

is one, which gives no-pass concurrency. A straightforward way to avoid deadlocks completely is to always scan a structure in the same direction.

The implementation of this mechanism is quite straightforward. All that we need is a circular queue of object identifiers of the length of the maximum number of previous locks to hold. We also need two index pointers, to the front id and to the tail id. Provisions can also be made for most options, such as dynamically changing the number of previous locks to hold on-the-fly. The semantics would be that when you call it to increase the number of previous locks, no new locks are acquired at that time, but they will be acquired when the cursor is moved. When the number of locks is decreased, the excess locks are released immediately.

5.4 Higher-Level Interface

Since the DSDS mechanism is fairly low-level and general-purpose, many abstractions can be built on top of it to make it more convenient to use. In this section, we examine more ways of escaping the tedious lock/release operations using higher-level abstractions.

5.4.1 Block-Structured Languages

The recursive organization and scoping rules of block-structured languages provide a familiar basis for automatic structuring and control of shared-data accessing. An example of a very simple structuring mechanism is to use block structuring to wrap shared-variable accesses, as in Figure 5.7.

This replaces SvLock()/SvRelease() pairs with a block structure. The shared variables named are locked in the mode specified. This makes it more difficult for a programmer to forget to release an object, since forgetting the closing curly-brace will be interpreted as a compile-time syntax error. As a small additional advantage,

```
with (vid1:X, vid2:S, vid3:X) {
    vid1->data = 3;
    ...
}
```

Figure 5.7: Block-Structured Shared Variable Access

```
dsds int x, y;
dsds usertype z;

int func( void )
{
    int i;
    dsds read x, y;
    dsds update z;

    i = x;
    z.f1 = 67;
    ...
}
```

Figure 5.8: Possible Language-Level DSDS Types

if multiple objects are named for locking within the block, then their lock requests can be grouped if they are being sent to the same place.

The simplicity of this mechanism comes at a cost. This block-structured methodology imposes the same ordering restrictions on locks and releases that the Ada programming language [ANSI83] imposes on accepts and releases in its rendezvous mechanism: they must be fully nested.

If we assume more ability to shape our own language constructs, we could produce a mechanism as shown in Figure 5.8. At the top are sample global declarations of shared objects, and these can be brought into scope and locked by the additional local declarations inside functions (or inside blocks inside functions). The compiler would take care of managing and distributing the object ids associated with the symbolic names "x", "y", and "z". The flexibility of the mechanism that allows multiple locks

per object for a single process makes this abstraction operate correctly in a nested-function-call environment.

5.4.2 Data-Structure Models

In our experience, most of the suitable applications for DSD systems use some massive aggregation that is accessed a small piece at a time. Matrix-processing abstractions are the most useful for this. Abstractions for common data structures would be useful in this context also.

Of all the theoretically possible data structures that could be used in programming computers, there are only a small number that are commonly used in practice. Included in this set are matrices (which we have already addressed), queues, hash tables, and B-trees. Queues are often used in the small scale and hash tables and B-trees are more typically used for maintaining a large database of information inside (or outside) of a program. (Binary trees are of theoretical interest, but they do not seem to be used all that often in practice; however, tree structures may be considered simplifications of B-trees for our purposes.) The advantage of distributing these data structures is obvious: concurrent accessing and parallel processing. But, of course, there are complications.

Distributed Linked-List Queues

Linked-list queues are perhaps the most often used linked data structure for organizing data inside a program. Because of the pointer/object nature of the DSDS system, implementing basic distributed linked-list queues is straightforward. A programmer needs only to build procedures to maintain the links of the list properly for the exact type of queue and the accessing operations that are being implemented. Figure 5.9 shows a classical head/tail linked list.

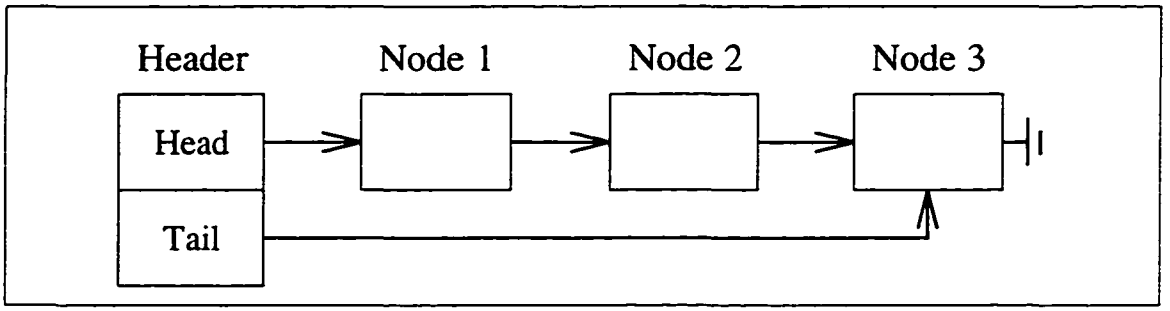


Figure 5.9: Classic Linked-List Structure

Unfortunately, however, the most direct and obvious translation of sequential-program linked lists to become distributed linked lists encounters some problems, namely, with concurrency control and efficiency. Concurrency control is important, of course, since the purpose of building distributed queues is to allow items to be inserted and removed correctly by multiple hosts.

While queue operations such as "count" and "is-empty" may be performed with read-only access, the insert and delete operations modify at least one existing object of the structure. B-trees and hash tables are natural for a shared-data approach, because of their relatively static structure, but queues are inherently much more dynamic.

Fortunately, the primitive operations of queues, insert at tail and remove from head, are relatively "brief" and involve acting upon only a few nodes of the linked list: the header and the head node for remove or the header and the tail node for insert (plus the node being inserted). These nodes can be simply X-locked and modified. For removal from the list, the critical path is locking, modifying, and releasing the header, since the old head node will be effectively removed from the list at the time of release. For insertion, perhaps the best approach is to X-lock the node to be inserted (which it likely will be already), initialize its next pointer to NULL, and release it; prefetch the tail node; lock and modify the header; lock, modify, and release the tail node; and release the header. The critical period is the total time that the header is locked. Alternatively, for a simpler implementation of removal, the header can be

X-locked for the duration of the removal and act as a lock for the structure.

For more complicated operations which require searching the linked list, concurrency control is more important. For the simple linked-list structure that includes only a head pointer in the header node, there is a straightforward solution to the concurrency-control problem. (This structure is sufficient for a priority queue, although more complex structures are usually used in practice.) As a result of the structure of the linked list, the insert/remove functions merely need to maintain a "guard" X-lock behind the current list node when accessing the list, since only the previous list node (or the header) can point to the current node. The link-cursor mechanism can be used to make this programming more convenient.

Prefetch linking could also be used to improve performance, setting the prefetch links according to the links in the list when changing the list links of a node, and by setting the prefetch link of the object to NULL when the object is removed from the list. This would be useful if many nodes are typically scanned before the sought-after one is found.

Process suspension on the list-empty condition is an issue that is not really addressed by any DSDS primitive. A timed-polling mechanism with an optional explicit wakeup signal worked well in our experiments (see Section 6.6.2). The wakeup signal would make the operation efficient and the timed polling would guarantee the wakeup signal is delivered even with an unreliable transport system.

There is an efficiency issue of exactly what should be enqueued into the linked list. For a list where a portion of the list is scanned every time using X-locks, the objects will be migrated to each host while it scans, even if it does not need the objects. If many objects typically need to be scanned or if the objects are large, this will involve a large amount of object-migration overhead.

Another possibility would be to use a "pointer queue", where there is a linked-list queue that works as described above, except that it is a queue of small matrices of

object-id/priority pairs (for simple priority queues). This involves less object migration during the search phase.

A special sub-class of queue is a single-producer/single-consumer queue. There are a few ways of implementing this. It could be implemented as a general queue, although this fails to take advantage of a regular sharing pattern in which one host writes and one host reads. The header will ping-pong back and forth between hosts. Another possibility is to use a circular buffer of a fixed size of pre-identified objects. The producer and consumer locally track the next object id that they will access in the queue, in a circular fashion. To control concurrency and enable full/empty suspension, the producer locks the next element in the queue before releasing the previous one, and keeps this object locked until data is put into it. This way, when the producer and consumer meet, the one "coming in from behind" is blocked until a new object (list item or free spot) becomes available.

Another possibility would be to extend the DSDS mechanism to allow user specification of *call-back* functions to be invoked when an object arrives asynchronously in the DSDS system (from an SvGrant()). If parameters queueId and seqNum were added as options to the grant message and passed up, the user could manage his own producer/consumer queues.

But, possibly the best and most obvious mechanism to use in the DSDS system for producer/consumer queues is the Tagged-Objects mechanism (Section 4.5), which supports producer/consumer queues fairly directly. However, in general, the shared-memory model does not support queues very efficiently in a distributed environment. This is because they inherently do not operate according to a locality-of-reference property, so objects are often referenced when they really are not needed, and extremely little processing is done per object to maintain the list. This functionality could probably be better implemented using a client/server RPC mechanism, where the server host maintains the queue entirely internally and all clients make Insert and Remove requests to the server.

Distributed Hash Tables

Hash tables are used for storing and looking up large numbers of objects very efficiently according to a "key value". The key value is mapped to a "bucket" in the hash table, and the object is stored in this bucket. There are various methods for handling "collisions" of mappings of different objects to the same bucket. The overall operation of a hash table may or may not exhibit the property of locality of reference, but hash tables can be relatively static in the sense they are often used in applications where the number of lookup operations significantly exceeds the number of insert/delete operations.

The implementation is very straightforward, as the hash table is really just an array. It can be implemented using the array (matrix) mechanism for buckets or for bucket pointers. A tile size should be chosen for larger arrays that gives a good tradeoff between accessing contention and storage overhead. In this case, a "good" size can be computed. For example, if hash-bucket entries were 16 bytes in size and each DSDS object has 128 bytes of control information, then if 32 buckets were grouped to each object, the portion of wasted storage would be 20 per cent. These values can be adjusted derive an acceptable storage overhead for an application, keeping in mind smaller groupings of buckets will experience less accessing contention.

A single bucket could be a linked list of user objects with the bucket as the header. Accessing contention for these queues is a much lesser issue than with general queues, since a well parameterized hash table will have an average of less than one item per bucket. Rehashing could also be used to handle collisions (instead of chaining using linked lists), in which case the buckets would only be pointers to user objects being inserted into the table.

Concurrency control is quite simple, as a host just locks the object containing the bucket it wants when it wants it, modifies the bucket, and releases the object. The rest of the hash table is free to be accessed independently. Additionally, accesses are

read-only for lookup and read-write for update, so concurrent reads can be performed on the same bucket at the same time even if the bucket became over-populated. If updates are frequent, then a performance improvement beyond simply relying on the basic DSDS mechanisms would be to broadcast updated objects since buckets are modified and looked up randomly. Broadcasting increases the likelihood that a bucket will be available for reading on a host when it randomly requests it.

One thing to note in the grouping of multiple hash buckets per DSDS object is that we are again facing an object-versus-structure locking issue, but rather than there being multiple objects per structure as there were in Section 5.2.1, there are multiple logical components per object. With the hash table, because of the randomizing effect of the hashing function, the accessing contention has a statistical relationship to group size of components mapped to a single object. In other applications, there will be different implications.

Distributed B-Trees

B-trees are used to store and access data according to "key" values, similar to hash tables. The difference is that B-trees store their values in order in a tree structure that grows in a well behaved way regardless of scale. The typical operations are insert, lookup, and remove. In many situations, the remove operation is needed only infrequently, so records are marked as 'deleted' rather than being removed from the structure. This is fortunate, since delete is a complicated operation. Insert is also a complicated operation. B-trees are widely used in database systems and can be used for 'databases' of information inside application programs. We will not discuss the full structure and algorithms for classical B-trees here (see [Cormen90]).

The purpose of a distributed B-tree is to make its 'database' of information available to all hosts in a distributed application. Like hash tables, B-trees are also a good application area for shared-data systems, since the data is relatively static. Unlike hash tables, the required concurrency control is very complicated.

The objects being looked up and the interior nodes of the B-tree should be represented with objects, so the B-tree is a giant linked structure of DSDS objects. As with linked lists, each B-tree will need one control object (header) to store the configuration meta-information and to identify the head node of the structure, since it can change.

For concurrency control, we can use the user-linked-structure-cursor mechanism described earlier. For performing a simple lookup, all that needs to be done is to scan down the tree in the usual sequential way using a link cursor with a shared-lock chain two objects long. This ensures that no remote hosts modifying the tree "pass" the local host. For an insert, processing is complicated by the splitting of index nodes that become over-filled. Potentially, they can split all the way up the tree back to the root, so one implementation would be to hold X-locks on all nodes that are traversed on the way to a leaf node to guarantee that the required access will be available to all parent nodes in case of cascading split operations. However, this effectively serializes all updates and disallows all lookups that start while an update is in progress. There is an easy way to correct this problem, however, since the X-lock traversal can safely release any parent(s) of a node that has room to add a new key. Using the linked-structure cursor mechanism, this can be accomplished by adjusting the chain-length parameter on the fly.

5.4.3 Transactions

The DSDS primitives could also be used to implement a transaction-processing system, although we will not attempt to give a detailed description of how to do that here. Basically, the DSDS mechanism provides the concept of objects and locking, and what is needed is a control algorithm that groups lockings and releases into transactions, monitors the concurrency of the transactions, and aborts and recovers objects that were involved in a transaction that fails, and handles cascading rollbacks.

5.5 Techniques for Using DSDS

In this section, we provide suggestions for how an effective implementation using DSDS may be produced for various circumstances. In the first subsection, we describe general principles for choosing DSDS objects and in later sections we describe how to adapt programs written for other shared-data systems to DSDS.

5.5.1 Mapping Data Structures to DSDS

The DSDS system is a system for managing user-defined data objects, but what should application objects *be*? There are many choices and tradeoffs.

For applications that have data items which have distinct boundaries and distinct units of what should be considered and processed as an independent entity, using a single object to represent that is a very logical and natural choice.

However, one must be careful not to use data objects that are either too large or too small to be managed efficiently by the DSDS system. One must keep in mind that the transfer of an object from one host to another in a network of workstations incurs a fixed cost in time on the order of 1 ms, and this can have an impact on the usable granularity of objects in the system. One must also keep in mind that large objects require extra time to propagate between hosts on the network, which can slow down the overall application if a large amount of time is spent unnecessarily transmitting data to and fro. This is particularly wasteful if only small pieces of large data objects are modified but the entire data object needs to propagate through the network.

If only a small piece of an object is modified, the DSDS system does not make any optimizations to propagate only the modified portion. This would require the system to keep track of which portions of an object have been modified, which actually is not a large extension to the system, but the user would need to tell the system which parts of an object have been modified with each release operation, since there is no

hardware mechanism available to determine this automatically.

So, instead, it would be better to divide a naturally large object up into smaller pieces so that fewer system resources will be required to propagate the changes. However, if a large object basically needs to be sent as a whole from one host to another and this is needed relatively infrequently (a "one-shot deal"), then it is better to leave it as a single large object so that less system overhead will be required than to send and process separate messages.

There is another tradeoff to keep in mind when dividing objects up: the memory overhead of the control information that is used to maintain each object. In addition to the memory required for the user data of the object, on the order of 100 bytes of control memory is also required. So, obviously, there would be a very large overhead if one were to use separate objects for a large number of four-byte integers of user data. In this case, the user would want to group together multiple data items into a single DSDS object in order to amortize the control-information overhead.

More generally, data granularity is an open research question. When you group independent data items together, you can start to get false-sharing problems, as in classic-DSM systems. There is a tradeoff between grouping and splitting up natural objects that needs to be considered with the semantics and accessing patterns of the user application in order to come to a good decision. It may also be a complicating factor that different organizations would work better in different phases of a computation. DSDS provides no mechanism to support this, so the cost of brute-force copying of data from one form to another needs to be taken into consideration.

DSDS is organized to very naturally support arbitrary dynamic data structures. Nodes of these structures are often small in size which fits well with DSDS (as long as they are not *too* small). The object identifiers were intentionally designed to be memory pointers to allow data-accessing operations to use the usual programming-language pointer operations for dynamic data structures.

5.5.2 Classic DSM

There are two major differences between DSDS and DSM: the organization and accessing methods of user data. DSDS works with user-defined objects and requires that object accessing be performed using explicit locking primitives. DSM systems allow direct processor access to cached shared memory and allow user objects to be mapped to the shared memory in arbitrary ways. These fundamental differences make mapping programs from DSM to DSDS difficult. The primary advantage of DSM is that mapping programs from parallel code to DSM is easy and may require no code modification at all. The disadvantage, of course, is that the mapping may perform poorly.

To map from DSM to DSDS, the major difference between the two systems must be papered over. User objects need to be identified and declared to be DSDS objects, and accessing of the objects must be controlled. Automatic means of performing this mapping would involve a language-level translation mechanism that identifies shared declarations, determines a good DSDS-object representation, and inserts code to control object access. This is clearly not a trivial issue, and is beyond this scope of this thesis. So, practical mapping would need to be performed manually, using techniques and approaches outlined in this thesis to make good choices.

5.5.3 Munin

Munin provides independently controlled user-annotated data objects and a loose consistency protocol. The annotation for an object tells the system what memory-coherency model should be used for the object. Since shared objects are explicitly identified and the likely operational semantics are given, there is more information available to help automate the mapping process from Munin programs to DSDS programs. DSDS essentially provides a superset of the Munin annotations, achieved by composing the various DSDS mechanisms and techniques. The possible Munin

Data-Object Type	Coherence Mechanism	DSDS Equivalent
Private	None	Private storage
Write-once	Replication	Replication, pre-S-lock granting
Write-many	Delayed update	User-controlled X-lock release
Results	Delayed update	User-controlled X-lock release
Synchronization	Distributed locks	Integrated locks
Migratory	Migration	X-locks, pre-X-lock grant
Producer-consumer	Eager object movement	Pre-granting, prefetching
Read-mostly	Broadcast	Multi-pre-grant (broadcast)
General read-write	Ownership	General protocol

Table 5.1: DSDS Equivalences to Munin Coherence Annotations

user annotations [Bennett90] and the DSDS equivalents are shown in Table 5.1.

The object declarations and annotations of Munin make object mapping much more automatable: however, the problem still remains of data accessing. Munin uses page-fault mechanisms to detect accesses to shared objects and to propel the coherence-control protocol, whereas DSDS still requires the use of explicit primitives.

5.5.4 Linda

Linda provides a high-level highly structured explicit interface to distributed shared data. Because of this, DSDS can be used to program a system that emulates Linda, and this is what would be needed to map Linda programs to DSDS. This would involve providing facilities to create, destroy, index, and retrieve tuples from the tuple space.

An obvious organization would be to store each tuple as a DSDS object, which would allow the system to use the DSDS concurrency-control and data-migration mechanisms directly. This makes the function of in, out, and read operations straight-

forward, involving creating (or reusing), X-locking, or S-locking objects, respectively. The other main requirement is an indexing mechanism for searching for tuples. A hashing scheme based upon computing a hash value from the first field of the tuple data could be used, which is standard Linda practice. A global array split into appropriately sized tiles could be used as the hash table. To perform a tuple search, if the key field is given, the value is hashed and a DSDS object is fetched and returned. If the key field is not given, then the entire tuple space must be searched, unless a more complicated data structure is devised.

Linda provides a convenient abstract interface for distributed programming. However, it may be necessary to program Linda in a special way in order to make a program run efficiently. If Linda programs are written without regard for Linda's internal operation, then programs may not work very well. This is a sad fact of life for many such abstract systems.

Linda programs can also try to indirectly (wastefully) emulate straightforward things such as matrices. With matrices, elements (or tiles) are very simply named and mapped, so using a general-purpose mechanism such as a distributed hash table to locate them is overkill. The DSDS methods for processing matrices will likely give far superior performance because of their directness and their user-controllability.

5.5.5 Orca

Henri Bal's Orca language implements distributed active objects. Each object logically has its own thread of control and the object accepts method invocations, executes the method atomically (by locking other conflicting invocations out while executing the given one). Method invocations are totally ordered and Bal uses a replication/update protocol with a sequencer host.

Of course, the DSDS system is not really "Object-Oriented", but this is not an extremely important issue, since "Objects" are mostly a convenient "illusion" for

providing concurrency-controlled access to user-defined chunks of data, which *is* the purpose of the DSDS system.

To emulate the object-oriented type of functionality with DSDS in C, we need a subroutine for each method and the first argument should be the vid of the specific object. Inside the method, the object would first be locked with the required accessing mode, its internal information would be accessed and processed appropriately, and then the object would be released and the subroutine would return. This is quite straightforward and may be fairly automatable. On the other hand, Bal also provides more complicated features such as method-invocation guards, which would require more effort to emulate. But, the message here is that the systems operate fairly analogously, so mapping from one to the other is conceptually straightforward.

5.5.6 Shared Regions

The Shared Regions system and the basic DSDS system (of Chapter 3) share a very similar programming model and object-locking primitives. Thus, there is a fairly direct mapping of the basic objects and operations from Shared Regions to DSDS, for the Shared Regions primitives that exist. Beyond this, the two systems are very different, designed for very different environments.

5.6 Conclusions

This chapter introduces mechanisms to make programming DSDS applications easier, and it explains how to make the best use of the lower-level primitives.

Matrix processing is a common application for a DSD (Distributed Shared Data) system. Data in the matrix needs to be divided up into many segments (tiles) that have differing sharing patterns. These tiles can be operated upon independently to maximize concurrency. The matrix data needs to be organized in this way since this

is the only practical way to process it in a network-of-workstations environment, and in most cases applications can easily work with data organized in this form. Segments of the matrix are generally allocated to the control of different individual processors. DSM systems always split matrices into pages. Pages are unlikely to match the "natural" or optimal tiling of a matrix for an algorithm, and false-sharing problems can exist between independent tiles.

There are many different possible approaches to providing access to these tiles and to the matrix elements. The ideal would be a way that completely abstracts out the tile handling while simultaneously providing optimally efficient access to the matrix elements. Achieving this ideal is difficult. A naive approach abstracts out tile handling while providing terrible performance, while some other approaches can be difficult to use and still provide sub-optimal performance. Probably the best approach considered for the DSDS environment is to use mapped matrices. In this approach, the matrix is accessed in the usual way, which allows easy programming and full compiler optimization. The down side is that tiles need to be locked and released at the correct times. However, there are many cases in which tile boundaries match the loops of the algorithm conveniently. Creating and naming mapped matrices is a tedious process with the low-level primitives, but fortunately, it is quite automatable.

Facilities are also provided for mapping the storage of one object onto memory in an interleaved fashion in the shared-memory arena, for example representing a two-dimensional matrix. This allows independent sharing semantics for different pieces of the matrix while eliminating false-sharing thrashing or "special" system overhead (e.g., which would be required for generating and merging "diff" records).

Another general class of problems involves processing linked dynamic data structures. The DSDS system provides a direct representation of these structures in a distributed environment and convenient pointers for direct programming. However, data coherency and concurrency control are issues in a distributed environment. The objects must be locked and released as they are accessed, and link cursors are proposed

as a mechanism to make this more convenient. The user moves the cursor around in a linked structure and as the cursor moves, it automatically acquires and releases locks on objects. Options can be set to provide automatic prefetching operations for these objects.

Concurrency control can be provided by using straightforward mutex locking for structures, or by using a "no-pass" method. The "no-pass" method involves acquiring and releasing locks in a special way that does not allow remote processes using conflicting access modes to traverse past the current position of the local process. This method integrates very naturally with the link-cursor mechanism.

The nested nature of block-structured languages can also be used to make lock handling more convenient. A direct approach is to define a special control structure, inside of which named objects are locked in a specified way. This removes the problem of forgetting to release a lock. This can also be provided at a function level for importing and locking an object from global declarations during the execution of a function.

Implementation details for abstract data structures for linked-list queues, hashing tables, and B-trees are also proposed. A high-level approach to provide data storage and retrieval for user objects is to implement the operations of the three common storage structures mentioned above in a library environment and provide an interface to user applications.

This chapter also discusses in detail how to make the best use of the DSDS system and primitives. How to map from other DSD systems to DSDS is also covered, both to inform application programmers and to demonstrate that the DSDS system provides or can be made to provide the functionality of the other DSD systems.

Chapter 6

Test Applications

This chapter applies the system and concepts from the previous chapters to example application programs. The purpose is to illustrate the use of the system, to demonstrate that it performs well, and to allow discussion of the techniques needed for good performance. The problem classes of various matrix algorithms and game-tree searching are examined and performance results are obtained and discussed.

6.1 Test Environment

The test environment consists of eight RS/6000 workstations connected by both a 10-Mbps Ethernet and a 100-Mbps FDDI network. Each RS/6000 has 64 MB of RAM and a PowerPC-601 processor running at 66 MHz, and each workstation runs AIX 4.1.3, a version of Unix. Two different compilers are available in the test environment, gcc from the Free Software Foundation and xlC from IBM. All programs are compiled using -O3 optimization for the best results. The DSDS system also runs with its special tracing mode disabled to minimize overhead. (This feature was built in for protocol debugging.) Testing is performed in a lightly loaded environment.

The performance results are measured mostly in average elapsed real execution

time (in seconds). These figures are compared against tuned sequential-execution versions rather than just the one-processor distributed versions, to give a fair comparison. One must note that tuned sequential implementations perform very well, especially compared to distributed implementations in a network-of-workstations environment, since processors and compilers are optimized for sequential implementations. This provides strong competition for distributed algorithms.

For most matrix applications, the problem size considered is a matrix of 1024 × 1024 double-precision floating-point elements. This size is large enough to produce stable results for runs, and the size is constrained by the amount of available memory for the application without using virtual memory.

The wall-clock execution times of various network and DSDS operations were also measured. These DSDS results are summarized in Table 6.1 with some simpler sequential operations for comparison. The compiler is very effective at optimizing loops in the sequential examples. Analysis of the assembler code revealed that the compiler unrolled the loops to process four or eight elements in a series and it interleaved the register usage to avoid data dependencies that would conflict with the pipelining of the processor execution. For the DSDS primitives, the object used was always 1024 (user) bytes in size. The SvCreate() operation on a remote host requires an RPC with the coordinator as does the SvLock()-remote operation. The SvLock() with a forwarded request requires one RPC and one asynchronous send by an intermediate host. The SvGrant() operation requires one asynchronous send, and the SvBarrier() operation requires an RPC and potentially some waiting, but a barrier was executed immediately before the one timed to insure that the hosts were well synchronized beforehand.

The network results for RPC operations of various sizes on FDDI and Ethernet are summarized in Table 6.2. The Ethernet driver appears to be better tuned for latency than the FDDI driver, as evidenced by the four-byte UDP timings. The TCP times do not include connection time and the connection is "primed" by executing

Operation	Time
integer sum 1 to 1000	42 μs
floating-point sum 1 to 1000	67 μs
sum 1000-element floating-point array	25 μs
copy 1000-element integer array	45 μs
semaphore-lock kernel call	18 μs
SvCreate(size=1024) on coordinator	158 μs
SvCreate() on remote	2.900 μs
SvLock(X) local heavy	142 μs
SvLock(X) local light	28 μs
SvLock(X) remote	3.200 μs
SvLock(X) remote w/one req forwarded	4.500 μs
SvGrant(X) to remote	690 μs
SvBarrier() with 4 hosts	4.200 μs

Table 6.1: DSDS-Primitive and Sequential-Operation Times

Protocol	Request Bytes	Reply Bytes	FDDI	Ethernet
UDP	4	4	2.03 ms	1.48 ms
UDP	4	1,024	2.30 ms	2.47 ms
UDP	4	65,500	14.50 ms	69.67 ms
TCP	4	4	2.05 ms	2.48 ms
TCP	4	1,024	2.39 ms	2.71 ms
TCP	4	65,500	14.76 ms	66.17 ms
TCP	4	1,000,000	165.30 ms	969.33 ms

Table 6.2: Network RPC Times

several large RPC requests before the ones measured, and large internal socket buffers are used.

6.2 Matrix Multiply

Although *Matrix Multiply* (MM) may not be considered a "real-world" problem, it has been widely used as an important example in research on distributed shared data. The reason that MM may not be considered a real-world problem is that its results are not extremely useful on their own, so it is often integrated into a larger program that computes other things and in that environment, MM can often be integrated into other more complex computations to absorb the $O(n^3)$ cost of the usual algorithm for MM.

MM is also an odd choice for demonstrating the effectiveness of a DSD (Distributed Shared Data) system because it is easily observed that the algorithm has no inherent data-accessing contention. The algorithm is easy to distribute among P processors using just about any distributed-computation paradigm and the speedup should be nearly linear, minus system overheads. The obvious way to organize the concurrency is to make each processor compute the results for an exclusive region of the result matrix.

However, the important thing that MM does provide is a means of comparing the overheads of DSD systems, hardware platforms, and computation paradigms by measuring how far from linear the resulting speedups are. The distributed computation should involve the fan-out and fan-in of the matrix data, to demonstrate the overheads involved in these operations, and whether the communication balances well against the computation.

6.2.1 Sequential Implementation

The sequential implementation is very straightforward and uses three different $N \times N$ matrices, $A \times B \rightarrow C$. The A and B matrices are initialized to static values according to a simple function based on matrix indices, and C is (effectively) initialized to all zeroes. The classic $O(n^3)$ nested-loop algorithm is, in a template form:

```
for(i=1,N) for(j=1,N) for(k=1,N) C[i][j] += A[i][k] * B[k][j];
```

There are some important optimizations that can be made to tune the algorithm for implementation. An accumulator variable can be used instead of referencing C[i][j] in the inner loop (since a compiler may or may not be sophisticated enough to optimize this case); the C array should not be initialized since the accumulator variable will be initialized in its place; and the contents of the B matrix should be transposed so that it, too, will be accessed in a row-major fashion (B[j][k]), since processor-cache systems are optimized for row-major matrix accessing. In general, we made a serious attempt to tune the sequential versions of algorithms to give the best competition possible for the distributed versions. This tuning involved organizing loops so that they could be well optimized by compilers and in some cases looking at the generated assembler code for alternate codings.

The performance of the sequential implementation is split into three phases: matrix initialize, multiply, and "print". Instead of actually printing the results, which would generate large volumes of extraneous I/O, a simple checksum of the matrix is generated instead, simulating only the single-pass reading behaviour of printing. Results are obtained for the naive, accumulator, transpose, and the transpose+accumulator versions with 1024×1024 matrices of doubles compiled with gcc. Results are also obtained for the transpose and transpose+accumulator versions with the xlC compiler plus with 512×512 matrices transposed with xlC. All timings are in seconds, and the results are shown in Table 6.3.

The results show that the optimizations suggested above produce significant im-

Phase	Naive	Accum	Transp	Tr-Acc	Tr-xlC	Tr-Ac-xlC	Tr-xlC-512
init	3.55	2.03	2.02	2.03	1.62	1.62	0.40
mult	1522.58	1527.69	227.25	177.87	130.87	131.00	16.43
print	0.14	0.14	0.14	0.14	0.11	0.11	0.03
TOTAL	1526.28	1529.87	229.42	180.04	132.68	132.73	16.87

Table 6.3: Sequential Matrix-Multiply Performance

provements, and that the xlC compiler produces "faster" code than the gcc compiler, at least for tight-loop numerical computations. Transposing the B matrix, produces the greatest improvement of all, demonstrating that the caching effects of memory-access patterns play a major role in the performance of modern processors. The xlC compiler and transpose version of the algorithm (the best) will be used for the distributed implementation.

6.2.2 Distributed Implementation

We use mapped arrays for the distributed version. This is probably the best approach for MM. This means that we must split the arrays into "tiles" for sharing and processing. For example, if we had four processors, we could split the A, B, and C matrices each into quadrants and make each processor responsible for generating one of the tiles (quadrants) of the C matrix, implying that it needs to read certain tiles of the A and B matrices. The size and shape of tiles is a general issue.

The matrices are split into tiles of 256×16 elements each (or of slightly different sizes for tests with odd numbers of hosts to make the tiling divide evenly). Using tiles of this size of double floating-point values makes each tile 32K in size, which is a good size for efficiency when transferring over a network, since the size is large enough to amortize the overhead of a request-response protocol.

The algorithm is coded using a fan-out/fan-in paradigm: the coordinator host (host #1) first initializes the matrices, then the workers (and coordinator) lock and process their data, and finally the coordinator locks all result objects to checksum the result. Also, to limit network waste, the C-matrix tiles are acquired with the SV_NODATA option, since they are uninitialized. The contents of the B matrix are transposed in all distributed-MM tests, since the sequential non-transposed results are so poor. The workers lock, process, and release objects in different internal phases (as mentioned above), so the actual processing loops are identical to the sequential version (except that different ranges of rows and columns are accessed by different hosts).

There are two different ways to split up the matrix data for this algorithm: using quadrant-oriented or row-wise approaches. The quadrant-oriented approach involves splitting the matrix into as similar a number of columns and rows as possible. This yields a minimal total volume of data to be transferred to the hosts for the A and B matrices, compared to the row-wise tiling, which splits the matrices into ranges of rows. The row-wise approach is more expensive because it needs to transfer the entire contents of the B matrix to all hosts involved in the computation. An illustration of the tiling and data transfer for the matrices is shown in Figure 6.1. However, if broadcasting is available, then the row-wise approach is better because the entire B matrix only needs to be transmitted on the network once. The row-wise approach also has slightly better processor-caching potential, since all bytes needing to be processed are physically contiguous in memory. Unfortunately, broadcasting is not available in our system, so the row-wise implementation will need to transmit the B matrix point-to-point to each worker host.

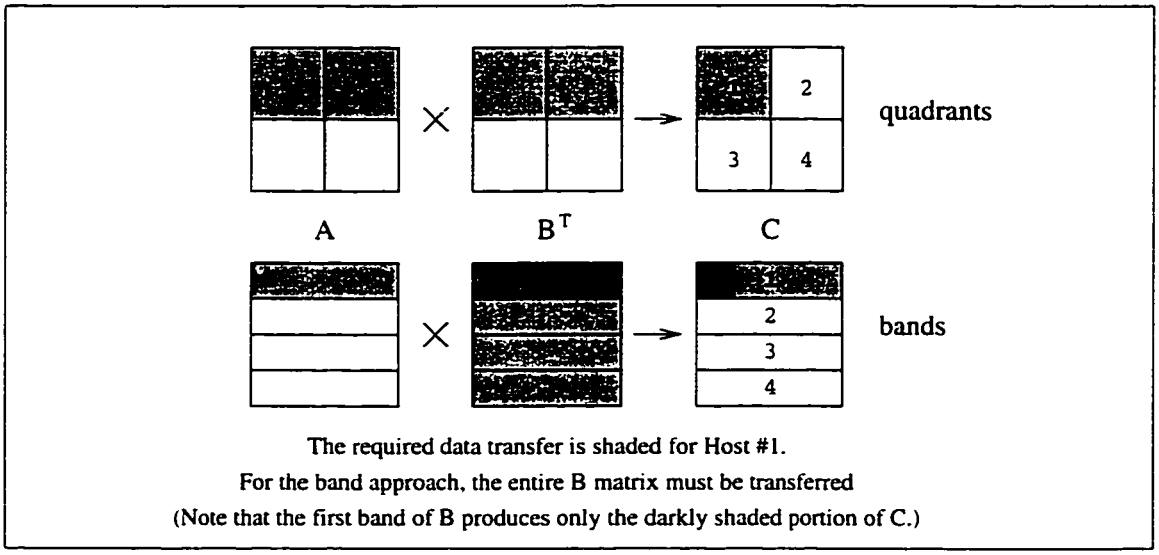


Figure 6.1: Splitting and Transfer of Matrices for Matrix Multiply

6.2.3 Distributed Performance

The results for the basic testing of the MM application with 1024 × 1024 matrices are show in Table 6.4. The timings are split into four phases: initialization (done at the coordinator), locking (fan-out), multiplying, and printing (fan-in). The "SEQ" column shows the sequential-implementation times, and the other columns show the distributed-version times for the number of hosts indicated. The "Q" notation means the Quadrant version and the "B" notation means the Band (row-oriented) version. The "-E" notation means Ethernet and the absence of "-E" means FDDI. The "Spdup-S" and "Spdup-1" rows show the speedup factors versus the sequential version and the one-host distributed version, respectively.

The results show that the fan-out times are monotonically increasing with the number of hosts involved, which is expected, and that the fan-in times are roughly equal, which is also expected, since a constant total volume of data is being transmitted back to the coordinator (the worker-owned tiles of the C matrix). Also as expected, the Ethernet fan-out and fan-in times are much greater than the FDDI

PHASE	SEQ	1	2B	3B	4Q	4B	4Q-E
init	1.62	3.34	3.34	3.34	3.46	3.44	3.47
lock	0.00	0.06	5.39	6.80	6.69	8.46	31.56
mult	130.87	167.28	83.79	52.46	42.03	41.79	42.43
print	0.11	0.12	1.78	2.80	2.49	2.46	7.59
TOTAL	132.68	169.21	92.71	64.42	54.47	55.02	83.73
Spdup-S	1.00	0.78	1.43	2.06	2.44	2.41	1.58
Spdup-1	1.28	1.00	1.83	2.63	3.10	3.07	2.02

Table 6.4: Basic Distributed Matrix-Multiply Performance

times, since the application is most bound by the volume of data that needs to be transferred rather than the amount of synchronization, and FDDI transfers data ten times faster, even though the two network types have similar effective latencies. The fan-out times of the row-wise Band versus Quadrant versions are also as expected.

The results also show that the multiply-phase times include a fixed CPU-processing overhead of about 30% over the sequential version, even for the one-host version, which requires no remote operations. Beyond this issue, the speedup of the multiply phase is very close to linear, as expected. It turns out that this 30% overhead is caused by an implementation issue. In the sequential version, the array accessing is performed with the language construct A[i][j], for a statically allocated array, whereas the distributed version uses the construct of (*A)[i][j], since the array must be dynamically allocated (to be in the shared-memory arena). Apparently, the xlC compiler is not able to optimize the indirect-access loops as well as the direct loops, even though there is no conceptual reason why it should not be able to. The solution is to access the distributed matrix using an overlaying vector with the construct aVec[j] (declared in C as "double *aVec;"). In this approach, every time a new row is to be used, the vector is set to point to the first element of that row, and then each column j of

PHASE	4Q-512	4Q-512-E	1V	2V	4BV	8QV
init	0.88	0.84	1.71	1.72	2.00	1.73
lock	1.54	7.50	0.06	5.15	10.53	41.08
mult	5.21	5.23	130.21	67.15	33.83	17.02
print	0.63	1.92	0.12	1.65	2.79	2.83
TOTAL	7.93	15.46	132.56	75.65	49.17	62.64
Spdup-S	2.13	1.09	1.00	1.75	2.69	2.05
Spdup-1	2.68	1.37	1.28	-	-	-

Table 6.5: Option Distributed Matrix-Multiply Performance

the row can be accessed with aVec[j]. This requires a little more programming effort. although it meshes with row-oriented tiling, and the compiler is able to fully optimize the accesses.

The results for additional testing of different options of the MM application are show in Table 6.5. The row and column are annotations are similar to those in Table 6.4, except that "-512" shows results for a 512 × 512 matrix, and the "V" indicates the use of Vector accessing to the matrix instead of indirect accessing. The 512^2-matrix version is much faster, as expected, since this is an $O(n^3)$ algorithm. although the Ethernet version fails to achieve much speedup because of fan-out overhead (which is actually greater than computation time). The Vector versions show initialization and multiply times that correspond to the sequential version, since the compiler is apparently able to make the same optimization for the matrix-accessing loops.

One thing to note is that the fan-out times increase drastically for the 1024^2-Vector times as the number of hosts increases from one to eight. However, it turns out that this is a programming issue rather than inherent. The issue is that there was no synchronization block between the fan-out and multiplication phases in the

PHASE	1V	2BV-br	4QV-br	4BV-br	8QV-br
init	1.71	1.82	1.90	1.80	1.67
lock	0.06	5.23	4.20	5.19	7.26
mult	130.21	65.37	32.86	32.52	16.50
print	0.12	1.72	2.43	2.49	2.94
TOTAL	132.56	75.22	41.85	42.69	28.96
Spdup-S	1.00	1.76	3.17	3.11	4.58

Table 6.6: Start-Barrier Distributed Matrix-Multiply Performance

original testing. It was not there simply because it was not necessary, but the absence caused an unexpected and undesirable effect. Since all of the matrix data is available at the coordinator after initialization, the fan-out locking phase executes very quickly there, because all of the objects are local. Thus, the coordinator entered the multiply phase while all of the worker hosts were in the locking phase, sending requests to the coordinator for the tiles they needed. The coordinator was unable to respond efficiently, since, even though there was a thread free to service the object requests, (at least) half of the CPU time was allocated to the multiply master thread, and significant operating-system overhead was incurred from constantly switching between the two threads.

To solve this problem, a barrier was placed between the fan-out and multiply phases, and the performance was drastically improved, as shown in the "8QV-br" column of Table 6.6 versus the "8QV" column of Table 6.5, by nearly six times. The other cases improved significantly also.

Overall, a maximum speedup of 4.6 times is achieved. With the vector version providing similar compiler optimization, the actual computation speedups are mostly linear, and the overhead is in the network transmission of data. Broadcasting would provide for even better speedups.

To help determine the accuracy and stability of measurements obtained, a small test was carried out. Three different arrangements of MM were run several times and the results were compared. All tests used eight hosts since this maximum number would introduce the most instability, and were run with the Quadrant+Vector configurations. For a test with a pre-multiply barrier with the FDDI network, an average time of 28.504 and standard deviation of 0.136 was obtained. For a test with a pre-multiply barrier with the Ethernet network, an average time of 81.315 and standard deviation of 0.627 was obtained. The Ethernet was lightly loaded with other users at the time. For a test without the pre-multiply barrier with the FDDI network, an average time of 62.581 and standard deviation of 0.364 was obtained. These results indicate that the measurements obtained in experiments in the test environment are highly stable, and therefore, no extensive analysis to obtain confidence intervals and statistical significance is required for the tests in this chapter. In all cases, three standard deviations from the average represents a maximum of only a 2.31 per cent difference. These results do not prove that other applications are equally stable, but the results in this chapter were also more informally observed to be generally stable in the course of conducting all experiments, which provides additional confirmation for stability.

6.3 All-Pairs Shortest Path

All-Pairs Shortest Path is an $O(n^3)$ graph algorithm that, given an adjacency matrix of costs of moving from one vertex to another, will compute the shortest paths between all pairs of vertices.

6.3.1 Sequential Implementation

The classic sequential implementation is shown in Figure 6.2. The sequential algo-

```
for (k=0; k<VERTICES; k++)
    for (i=0; i<VERTICES; i++)
        for (j=0; j<VERTICES; j++) {
            newcost = cost[i][k] + cost[k][j];
            if (newcost < cost[i][j]) {
                cost[i][j] = newcost;
                next[i][j] = k;
            }
        }
```

Figure 6.2: Sequential All-Pairs Shortest-Path Algorithm

rithm takes the path from vertex i to vertex j and compares it to the path from i to k to j, for all k, i, and j. If the path that goes through k is shorter, then the path from i to j is updated to go through k. After trying all k's, the shortest paths are guaranteed to be found. The sequential implementation is not much more complicated than the code in the figure.

6.3.2 Distributed Implementation

The clearly best way to divide the matrix for distributed processing is in row-wise bands. Rows give the most efficient processor-cache performance and rows also give a programmatically convenient place to insert tile-locking operations. There is a small issue with how the row bands should be organized. Constraints of the problem and the 64K-object-size constraint of the system make obvious a good solution, which is to make each row an individual tile (object). The problem constraint is that, if you examine the sequential algorithm, for each iteration k of the problem, the contents of row k are needed at each host. Using per-row tiles conceptually simplifies the data-sharing pattern. Figure 6.3 illustrates the assignment of the bands and the distribution of row k.

Another possibility is to use a separate object for sharing row k and copy the data to this object on each iteration. Making this a tagged object would probably

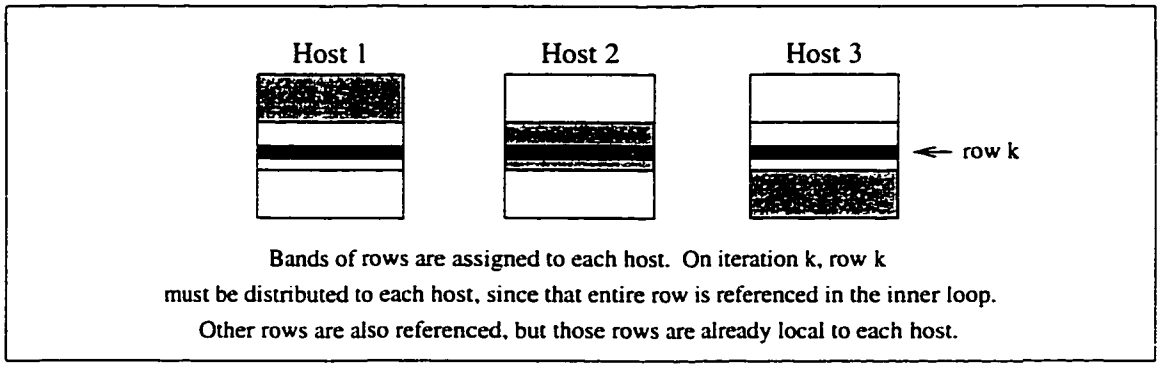


Figure 6.3: All-Pairs Shortest Path Row Assignment

be the best solution, although our implementation of tagging is inferior to the Versioned Objects in this case, since the owner can change. However, the owner changes infrequently, so there would be minimal overhead with changing ownership of the most-recently updated tag in the system. In any case, the tagged-object facility is not implemented, so it cannot be tested.

The implementation uses the fan-out/fan-in paradigm and the matrix is initialized at the coordinator to random values. For variation, instead of using a mapped array, each row of the matrix is allocated as a regular, independent object of the DSDS system. The ranges of rows are split up into N bands for N hosts, and after fan-out, all of the rows remain statically assigned to hosts, except for row k which is read by all hosts on each iteration.

A barrier is used to coordinate iterations and keep hosts synchronized, and locking is also necessary to avoid race conditions. For efficiency, and because of the locking constraint, objects remain locked during execution. After row k is computed at the host that contains it, the lock on that row is released and all other hosts are free to read it. A barrier is executed at the bottom of each iteration to ensure that all other hosts have read row k before the local host re-locks it to continue computation.

As an option to improve performance, prefetching is implemented for the fan-out, row-k distribution, and fan-in operations. For fan-out, the rows are SvGrant()ed to

PHASE	SEQ	1	2	4	4-PF	8	8-PF
init	2.73	3.19	3.13	3.25	5.18	3.23	5.54
lock	0.00	0.05	3.97	2.42	0.03	2.72	0.03
solve	419.92	406.77	222.65	136.31	122.97	95.41	82.63
print	0.20	0.31	4.02	5.88	3.99	6.79	4.57
TOTAL	419.92	410.32	233.77	142.32	132.21	108.22	92.83
Spdup-S	1.00	1.02	1.80	2.95	3.18	3.88	4.52
Spdup-1	0.98	1.00	1.76	2.88	3.10	3.79	4.42

Table 6.7: All-Pairs Shortest Path Distributed Performance

their worker host (calculated by a simple function) as soon as they are generated. Since there is a non-trivial amount of computation between rows (to generate the random numbers), the system is less likely to run into flow-control problems. For row-k dissemination, it is SvGrant()ed point-to-point to each other worker host in the system. The hosts SvInvalidate() their copies as soon as they are done with them. For the fan-in at the end of the algorithm, the rows at each host are linked together for prefetching and when the coordinator accesses them, it receives many at a time. This organization prevents the coordinator from being overwhelmed as it might be if all hosts simply SvGrant()ed their tiles to the coordinator. The prefetch-link approach was experimentally found to be the best in this case.

6.3.3 Distributed Performance

The sequential and distributed performance is shown in Table 6.7. The timings are split into the initialization, locking, solving, and "printing" phases (again, only a summation of the matrix elements is taken). The "SEQ" column shows the sequential-version performance and the results for differing numbers of hosts are indicated by numbers between 1 and 8. The "PF" notation indicates the prefetching version of

the application. The results mostly follow expectations. One oddity is that the one-processor version actually runs faster than the sequential version. It is suspected that this is because of compiler optimizations. The distributed version works with row tiles and therefore, its computations are placed in loops that access vectors, whereas the sequential version uses more complicated nested loops that use different loop-index values in different subscripts. It is suspected that the compiler is unable to optimize the more complicated references, similar to the problems experienced with matrix multiply.

The prefetching has the intended effect of speeding up the computation by removing synchronous delays. The fan-out speedup is only moderate. The results show the it virtually eliminates the locking time by transferring the cost to the initialization phase. The fan-in speedup is more significant, as the rows are requested from one host at a time by one host at a time (the coordinator). There is more round-trip delay here compared to the beginning where $N - 1$ hosts request rows and are answered per round trip. Linking the rows returns multiple rows per round-trip. Most of the speedup comes from removing critical-path time from the main processing-iteration loop.

The maximum speedup obtained is 4.5 times with eight hosts. A greater proportion of speedup is achieved with four hosts, and greater still with two. The main cost in the computation is the synchronization between iterations and the transfer for row k. A larger number of hosts means more critical-path synchronization delays, even with prefetching.

6.4 Successive Overrelaxation

Successive Overrelaxation (SOR) is a familiar test application in distributed systems. The classic algorithm is defined by the following equation for a single element of a matrix:

$$M'[r,c] := M[r,c] + \omega \times (av - M[r,c])$$

where M'[r,c] is the new value (for use in the next iteration) of the matrix element M[r,c], ω is the "relaxation factor", and av is the average value of the four cells above, below, and to the sides of the current matrix element. A relaxation factor is used to over-exaggerate the change in the current cell to make the process converge faster. The algorithm terminates when no element has changed by a certain tolerance value during the current iteration from the previous iteration.

The outside edges of the matrix are initialized to predefined values and these values are "pinned" in that they are not modified by the computation. The interior cells are initialized to arbitrary values, often zero or pseudo-random values. The algorithm can be thought of as a "heat-flow" simulation with constant amounts of heat being applied to the different outside walls and the interior temperatures of the rectangle settling to a smooth gradient of average values of the applied heats.

6.4.1 Sequential Implementation

The sequential and distributed implementations use a variation of the algorithm based on viewing the array as a checkerboard pattern of red and black cells and computing values for all red cells and then all black cells. Using this approach means that the values in the cells used in the computation are from the previous iteration. This simplifies the distributed version of the same algorithm. Other methods for computing SOR exist but were not explored. This method is commonly used for distributed systems.

The algorithm executes in $O(n^2)$ time, for each of m iterations. The exact number of iterations m depends on the relaxation factor ω and the tolerance value to stop at. For testing, values of ω and the tolerance are chosen to cause about 150 iterations of computation to be performed. This type of algorithm can often require multiple hundreds of iterations for convergence. The matrix size tested is 1024×1024.

6.4.2 Distributed Implementation

The matrix for the distributed version is probably best split into consecutive bands of rows assigned to each host. Because of the operation of the algorithm, the red and black (all) elements of adjacent rows of neighbouring hosts need to be read at the local host on each iteration. This is a fixed sharing pattern. The matrix could also be split into arbitrary rectangular tiles, but this would create the more complicated situation of sharing columns of the tiles. Columns are more expensive to deal with because their cells do not occupy contiguous memory. Mapped objects are used for variation, but under the circumstances, this causes little difference from using per-row shared objects.

Iteration control is implemented using a barrier in the usual way. However, only one barrier for every red/black iteration pair is needed, since the old red and black values are all that are needed, and hence, the latest version of the red values are not needed on the black iteration.

The computation within an iteration is implemented fairly naively. There are two loops, one which scans rows and one which scans columns, as in the sequential implementation. The only change needed is inserting code in the row loop to lock the three rows of the matrix that will be needed to process the current row and to release them at the end of the loop. Three rows need to be locked, since the current row reads element values from the previous and next rows of the matrix. Therefore, the current row is locked exclusively and the adjacent rows are shared-locked. This naive organization results in a great deal of locking as the program runs, which degrades the performance achieved.

After a certain tolerance of change is achieved, the algorithm must stop, and therefore, some mechanism is needed to facilitate this. Given the shared-data primitives, this is quite straightforward to implement using an object that all hosts lock, cast their votes into, and release. However, if a simple "aggregate-AND" mechanism is

used (where all hosts must vote TRUE to stop), then the accumulator variable must be reset to TRUE after each iteration. This may require the use of another barrier, so there can be no race conditions between resetting the accumulator variable, voting, and reading vote results.

A counting mechanism is used for voting instead of a boolean mechanism to eliminate the problems mentioned above. The counting mechanism is implemented as a simple integer inside a shared object. At the end of an iteration, any hosts wishing to object to continuing (because their changes are under the tolerance) lock, increment, and release the counter object before entering the iteration barrier. After the iteration barrier, all hosts read the counter object and if the value has incremented by N from the previous iteration (they save the value from the last time they read it), then computation stops; otherwise, they all compute a new iteration. This organization removes any need for an additional barrier, since the base value is effectively "reset" when the counter is read, and this organization also avoids unnecessary protocol operations, since the object is only X-locked when any host wishes to stop, which is typically in the last few iterations of a computation that often lasts for hundreds of iterations. When there are no changes to the counter variable, no operations are needed and all hosts repeatedly refer to their cached-valid copies of the counter.

6.4.3 Distributed Performance

The performance results for SOR are shown in Table 6.8. As usual, the results are split into "init" (initialize matrix), "solve" (perform SOR iterations), and "print" (sum the final matrix) phases, and results are shown for various versions of the application. The "SEQ" column shows the performance of the sequential implementation, and the numbered columns show the results for the distributed versions on that number of hosts. The "HL" and "LL" notations mean "heavy-weight locking" and "light-weight locking", respectively, and "PF" means prefetching with light-weight locking.

PHASE	SEQ	1-HL	1-LL	2-LL	4-LL	4-PF	8-LL	8-PF
init	1.31	1.41	1.42	1.42	1.44	2.73	1.46	2.99
solve	41.78	102.63	48.28	30.89	16.82	13.35	11.80	8.26
print	0.11	0.18	0.11	2.61	3.86	3.86	4.49	4.47
TOTAL	43.19	104.24	49.82	34.92	22.15	19.95	17.79	15.77
Spdup-S	1.00	0.41	0.87	1.23	1.95	2.16	2.43	2.74
Spdup-1	1.15	0.48	1.00	1.43	2.25	2.50	2.80	3.15

Table 6.8: Successive Overrelaxation Distributed Performance

The terms heavy- and light-weight locking refer to the type of locking performed on the shared memory at the local host to process the SvLock() and SvRelease() primitives. Heavy-weight locking uses a Unix semaphore and therefore requires a kernel call to acquire and release shared-memory locks. Light-weight locking uses special "test-and-set" (or equivalent) instructions of the processor and is executed in-line. Light-weight locking also uses code that is designed to efficiently check if the object is available locally and lock it quickly if it is. As can be seen in the "solve" times for the "1-HL" and "1-LL" versions of the application, the type of locking makes a large difference in this application, because there is a large volume of locking of objects that are almost always available locally. The solve phase runs over twice as fast, although not quite as fast as the sequential version, which is expected since the sequential version performs no locking at all. A disadvantage of light-weight locking is that it uses spin-locking, so if it is found that an object is not available locally, then heavy-weight locking is used to make the kernel block the process.

As can also be seen in the results, the prefetching makes only a small improvement. This is because it is only applied in one of three places in which it might be applied: the fan-out phase. It might also have been applied in the fan-in phase and to distribute overlapping rows to neighbours during the computation, but that was not done in

order to keep the implementation "naive". The maximum speedup that is attained is 2.74.

6.5 Gaussian Elimination

Gaussian Elimination is a classic numerical algorithm for solving systems of simultaneous linear equations. Given the A matrix of coefficients of variables and the b vector of constants, the algorithm solves for vector x of the equation $Ax = b$.

The classic $O(n^3)$ algorithm uses two distinct phases to solve the equations. In the forward-elimination phase, A is converted into an upper-triangular matrix by subtracting multiples of rows from other rows, and in the backward-substitution phase, solutions for known variables are substituted into the upper-triangular matrix to obtain more solutions to propagate.

6.5.1 Sequential Implementation

The sequential implementation implements the classic algorithm with the extension of "partial pivoting" during the forward-elimination phase to provide more numerically stable results. There are three nested loops for the forward elimination, the outer one that scans the diagonal, a middle one that scans all rows beneath the diagonal, and an innermost one that scans all columns of the row below the diagonal, making this phase $O(n^3)$. It is on the "diagonal" loop (or "iteration" loop) that the partial pivoting is performed. On each iteration, the row with the largest absolute value in the column of the diagonal is searched for and then swapped with the row of the current diagonal. This ensures fewer loss-of-precision errors during the floating-point computation. The backward-substitution phase is only $O(n^2)$.

Also in the sequential implementation, the A matrix and b vector are augmented, with the b vector as the rightmost column of A, for storage and processing convenience,

and, in fact, the b vector is actually multiple vectors (or a matrix), augmented to multiple columns of A. This allows multiple sets of simultaneous equations to be solved that use the same coefficient matrix, more efficiently than computing them separately. The test size chosen is a 1024×1024 A matrix augmented with 16 b vectors.

6.5.2 Distributed Implementation

The matrix is split into rows as it is with other matrix algorithms, with each row stored in an independent object. However, because of the structure of the algorithm, the assignment of the rows to hosts that process them is done in an interleaved fashion rather than in contiguous bands of rows. This is done to divide the workload more evenly among the hosts, since the algorithm processes triangular matrices; more work will be required for rows at one end of the matrix than at the other. Interleaving rows is a very simple and convenient way of evenly splitting up the work, and it imposes no prohibitive dependency problems (as it would with the SOR application).

The forward-elimination phase implements partial pivoting in the same way as the sequential version, except that a parallel search is performed to find the row with the largest diagonal element. Each host searches for the largest pivot value in its own rows and makes the value and row id available in an object. The coordinator reads these objects, selects the largest one, and swaps the rows. A barrier is used to coordinate this activity, and the high-value objects are normally locked and released only when they have valid data.

After the pivoting, the elimination is performed on all rows below the diagonal. All hosts S-lock the row of the diagonal and then process all of their own rows below the diagonal. Overall, the forward-elimination phase requires a great deal of synchronization between hosts. An additional barrier is used to coordinate iterations of the elimination phase.

PHASE	SEQ	1-HL	1-LL	4	4-PF	8	8-PF
init	2.02	2.05	1.94	1.92	3.72	2.04	4.17
elim	61.65	145.10	65.49	82.14	69.55	85.22	70.25
back	11.66	4.66	4.35	23.04	7.83	19.39	11.07
print	0.01	0.12	0.03	0.03	0.03	0.10	0.10
TOTAL	75.35	151.95	71.82	107.15	81.19	106.81	85.67

Table 6.9: Gaussian-Elimination Distributed Performance

The backward-substitution phase is implemented using "solution objects," which are used to contain the solution of each variable of the system of equations (or the multiple solutions if multiple b vectors are used). At the start of the back-substitution phase, all solution objects are locked. As each host performs its processing, it locks the solution objects that it needs to compute a new solution, and then it updates the new-solution object and releases it for other hosts to use. Although this organization makes the progress of the computation restricted only by data dependencies, unfortunately, the data dependencies of this problem inherently serialize the computation.

6.5.3 Distributed Performance

The results of the Gaussian Elimination application are show in Table 6.9. The headings of the table have the usual meanings. The results show that using light-weight locking instead of heavy-weight again makes a significant improvement in this application, again because local data items are locked many times without migrating. The results also show that the prefetching versions perform significantly better than the non-prefetching versions. Extensive prefetching operations are applied to this application to assist in all places where object flow is required. However, overall, the distributed version of this application fails to achieve a speedup over the sequential version. This is because the application is inherently ill-suited for distribution because

of the extraordinary amount of synchronization required by its computation. One anomaly is that the 1-LL version runs faster than the sequential version, but this is caused by a different order of accessing solutions (where differing orders were more "natural" to program for the different versions).

6.6 Game-Tree Searching: Othello

Othello is a board game that is played on an 8×8 grid of squares with pieces that are black on one side and white on the other. It is a zero-sum, two-player game where the objective is to capture enemy pieces to possess the greatest number of pieces at the end of the game. Each player plays either black or white pieces, and on each turn, a player lays down one piece so that it captures at least one enemy piece. When a played piece is placed next to one or more enemy pieces that are in a straight or diagonal line which has a "friendly" piece at the other end, then all of the enemy pieces between the friendly pieces are "flipped" to be the capturing player's color. Enemy pieces are captured in all straight and diagonal lines that apply, on each move.

Othello is an example of a game-tree searching application. These applications subdivide the problem of selecting the best move to searching a tree that represents all of the moves and counter-moves of the players. Each level of the tree is called a "ply". Scoring of the moves is done in some mechanical way, and the top-level move that gives the best overall score for moves and counter-moves is selected and performed by the computer.

The main computational problem with this type of algorithm is that the search time to examine greater numbers of plies, and therefore produce better moves, grows exponentially with the number of plies to look ahead. Thankfully, Othello is less expensive than chess, the *de facto* standard computerized tree-searching program, which has an average branch factor of around 35 moves per board configuration. Othello has been observed to have a **maximum** branch factor of around 18. But,

```
int CutoffSearch( Player, Depth, CutoffValue )
{
    if (Depth == 0) {
        return( 0 );
    } else {
        BestScore = -infinity;
        Generate the move list for Player as per current board setup;
        for (each move in the move list) {
            Make the current move and get MoveScore;
            SubTreeCutoffValue = MoveScore - BestScore;
            Score = MoveScore - CutoffSearch (enemy of Player,
                Depth-1, SubTreeCutoffValue);
            Unmake the current move;
            if (Score > BestScore) BestScore = Score;
            if (BestScore >= CutoffValue) exit the for loop;
        }
        return( BestScore );
    }
}
```

Figure 6.4: Sequential Othello Tree-Search Algorithm

in practice, this just means that computers are made to look more plies ahead with Othello than with Chess, and so they take a similar amount of time to make a move.

To combat the problem of exponential growth of the search tree, tree-pruning methods are applied to the problem. It is well known in game-tree-searching practice that large sections of the search tree do not have to be examined at all because it can be proven that they will not be included in the best-move sequence for the tree. Another remedy to the exponential-growth problem is parallelization.

6.6.1 Sequential Implementation

The pseudocode for the sequential algorithm for searching the game tree is shown in Figure 6.4. This is a mini-max algorithm and it implements tree pruning ("cutoffs"). Essentially, the computer searches for the move that generates the highest score for

the current player, taking into account the score for the enemy player's best counter-move to the first move. The score for the enemy's best counter-move takes into account the current player's best counter-move to that move, and so on recursively, down to the depth of the number of plies to look ahead.

The basic mechanics of the algorithm are implemented straightforwardly. The score computed for a move is simply the number of enemy pieces captured plus a four-point bonus for moving into a corner. The playing board is an 8×8 matrix of characters (X's and O's are used in place of white and black circles). The board is stored in a global variable for convenience. The operation of generating the move list scans all empty squares on the board and computes how many enemy pieces would be flipped if the player moved there by scanning all eight straight and diagonal directions that radiate from that square. After generation, the move list is randomized to prevent the moves from being too predictable, and it is then sorted by the numbers of pieces captured (while remaining randomized within each capture-count group).

The cutoffs operate according to a straightforward principle. If player X makes a move and then player O makes a very strong counter-move that results in X's move being not as good as some other move it has already scanned, then there is no need to scan the rest of the list of O's counter-moves, because the move from X and all descendent moves can be discarded immediately, with no additional scanning. The effect of cutoffs tends to make the tree "left-side heavy", in that more moves are scanned on the left sides of subtrees than on the right sides (since the right sides are often cut off by the above rule). The order that moves are scanned also plays a role in performance, since if the best moves are scanned first, then the more inferior moves will be pruned more quickly and a smaller total number of moves will need to be examined. This is the reason that the move list is sorted at every level: to attempt to approximate a best-move-first scanning order. Tree pruning improves sequential performance enormously.

6.6.2 Distributed Implementation

Game-tree searches are difficult to parallelize because of the complexities of the same tree pruning that makes the sequential versions more efficient. An approach to parallelizing a tree search while realizing benefits from tree pruning is to "emulate" the operation of the sequential algorithm at a coarse grain (at high levels of the search tree), but to evaluate multiple subtrees in parallel at a fine grain (at the lower levels of the search tree).

This approach has been implemented here using DSDS objects that maintain the necessary structure of the search tree for accumulating search results and doling out parcels of work using a "job-jar" approach. Each job includes a board configuration, a list of moves to be carried out at that level, and the parameters passed into the CutoffSearch() function show above. A global "evaluation horizon" is used to tell what should be done with jobs. If a job is retrieved that is at a level above the evaluation horizon, then a sub-job is generated that would result from applying the first move in the move list, and sub-job and the original job (minus the generated move) are placed back in the job jar. If the job is at or below the evaluation horizon, then the job is evaluated directly by the machine that acquired it using the sequential searching algorithm, and the evaluation result is obtained.

Each job in the job jar has a priority value which ensures that jobs are retrieved in left $\rightarrow$ right tree order to simulate the progress of the sequential scanning algorithm and to prevent the list of outstanding jobs from exploding exponentially. This order is important because it allows cutoff values to propagate and be used in the same way as the sequential algorithm.

Each job also includes a pointer to a "result record". Each result record accumulates the best score found so far for the associated node of the tree and maintains the number of moves still outstanding and a pointer to its parent result record. As jobs complete, their results are checked against the result record, and if the job has

a better move, that move is recorded. The number of outstanding moves is decremented by the number of moves evaluated in the job, and when that number reaches zero, the scanning is complete for that node of the tree. (Note that result records may accumulate their results from multiple (potentially concurrently executed) jobs.) When scanning is complete, the result record merges its results into the parent result record's, and if that completes, the results percolate back up, potentially to the root of the tree. When all of the outstanding moves in the root result record have been completed, then the scanning is finished.

The implementation also includes priority queues which are implemented by using bi-directional linked lists. The job jar and free queues for completed jobs and result records are implemented as such. The basic DSDS object/locking mechanisms are sufficient for the entire implementation above, except that there is no automatic means for waiting for an arbitrary condition (such as a queue being non-empty). Thus, if a host attempts to acquire a job when the job jar is empty, it will wait for a fixed period of time and then try again. However, to make this more efficient, when a host adds a job to the job jar, it sends a (unreliable) wake-up message to a waiting host that will cause it to stop its waiting if the message is received.

6.6.3 Distributed Performance

The Distributed Othello performance results are shown in Table 6.10. Each test case is identified by the number of Workers (hosts) that executed the distributed version (or "SEQ", identifying the sequential version), the Depth of the search tree, and the evaluation Horizon (the height from the leaves of the tree at which to directly compute solutions). The measured values of the total elapsed Time in seconds and the total number of Scans (moves actually checked) are shown. Finally, the derived values of the number of scans per second, the improvement factor in the number of scans per second over the sequential version, and the real-time speedup are shown.

Workers	Depth	Horizon	Time	Scans	Scan/Sec	Scanup	Spdup
SEQ	9	-	55.84	1,314,203	23,535	1.00	1.00
1	9	5	61.12	1,314,362	21,505	0.91	0.91
2	9	5	66.37	1,904,781	28,699	1.22	0.84
4	9	5	88.09	2,353,460	26,717	1.14	0.63
4	9	6	39.19	2,807,000	71,625	3.04	1.42
4	9	7	39.67	3,145,912	79,302	3.37	1.41
4	9	8	29.05	1,761,581	60,640	2.58	1.92
8	9	5	159.48	2,336,153	14,649	0.62	0.35
8	9	6	27.76	3,121,028	112,429	4.77	2.01
8	9	7	28.71	3,804,331	132,509	5.63	1.94
8	9	8	20.09	2,234,642	111,232	4.73	2.78
8	9	9	57.00	1,314,203	23,056	0.98	0.98
SEQ	10	-	374.33	9,163,780	24,480	1.00	1.00
8	10	6	173.91	22,998,017	132,241	5.40	2.15
8	10	8	133.48	22,895,656	171,529	7.01	2.80
8	10	9	212.92	16,940,289	79,562	3.25	1.76

Table 6.10: Distributed Othello Performance

A relatively simple test case was used to gather the results. A reachable and apparently typical board configuration with 20 pieces on the board was chosen at random (from letting the computer play itself to that point with a nine-ply lookahead). Then, the board configuration was saved and used as a starting point for all of the above tests. The test case has the desirable qualities of requiring about one million moves to be scanned on level nine, which can be scanned in a reasonable time, and play is "opening up" into the mid-game.

The performance figures show some mixed results. When the solve-horizon is set too low (at five or below), the distributed performance suffers because there are too few moves actually scanned to amortize the control overhead. Performance improves as the horizon rises, and it peaks near the top of the tree. The number of moves scanned also increases at higher horizon levels and as more worker hosts are added. This was expected to a degree because of the slowed-down propagation of optimal cutoff values, but the result figures go beyond what was expected and we are still not completely sure why. The scanning rate also increases as the horizon rises. Performance goes back to near-sequential when the horizon reaches the tree depth, since there is only one job to be solved, and only one host acquires it.

Overall, the maximum scanning-rate speedup factor was 7.01 and the maximum elapsed-time speedup was 2.80. And overall, the DSDS mechanism provides an effective means of implementing this distributed application.

6.7 Performance Comparison to Other Systems

This section compares DSDS to other systems. Comparisons will be based partly on experience gained from implementation and tuning of test applications.

6.7.1 Matrix Multiply with Message Passing

The implementation of matrix multiply with message passing is studied by Rees and Black [Rees91]. The environment used is up to five DEC MicroVAX IIs interconnected by Ethernet using Shoshin with synchronous message passing and TCP/IP. Although the RS/6000 and FDDI environment used for testing DSDS is considerably more powerful, the two environments have a similar 'balance' between CPU performance and network performance. Their results indicate that the algorithm must be chosen carefully in order to achieve speedup at all and only coarse-grained parallelism works efficiently. The best strategy is to reduce the number of independent messages sent and send the largest volume of data with each message to amortize the "flat-rate" message cost.

They study many different versions of matrix multiply, including ones which improve the core algorithm over the classic $O(n^3)$ one. Their results are summarized in Table 6.11. For the relevant cases, their speedups are greater than with DSDS. They do not mention whether they transpose the B matrix, which would make a large difference in their favor, since matrices can be accessed more efficiently in a row-wise fashion and this would slow down their computations relative to their communication costs. They also are using tuned message-passing algorithms. Their super-linear speedups are also curious.

6.7.2 CVM

CVM [Keleher96a] is a DSM system based on multi-writer lazy-release consistency. CVM actually allows three memory-consistency protocols. Sequential which makes its behaviour identical to Classic DSM, Single-Writer Lazy-Release which can lead to false sharing between overlapping objects, and Multi-Writer Lazy-Release which eliminates false sharing but which incurs more CPU overhead in generating and merging "diff" records.

Rank	Algorithm title	Time	Hosts	Speedup
1	Winograd's method, distributed	82	4	5.66
2	Distributed control, inner-product workers, multiple vectors per message (quadrant method)	102	4	4.55
3	Hybrid Winograd-Strassen algorithm, distributed	107	5	4.34
4	Distributed control, inner-product workers, multiple vectors per message (row-bands method)	128	4	3.63
5	Distributed control, outer-product workers, multiple vectors per message	-	3	-
6	Hybrid Winograd-Strassen algorithm, sequential	318	1	1.46
7	Winograd's method, sequential	384	1	1.21
8	Classic sequential inner product	464	1	1.00
9	Distributed control, inner-product workers	1543	3	0.30
10	Centralized control, inner-product workers	5067	4	0.09

Table 6.11: Rees and Black Message-Passing Matrix-Multiply Results, n=256

Algorithm	Hosts	Consistency	Time	Speedup
Mandelbrot	1	SW-LR	29.2	1.00
Mandelbrot	2	SW-LR	17.2	1.70
Mandelbrot	3	SW-LR	13.0	2.25
Mandelbrot	4	SW-LR	10.4	2.80
Mandelbrot	5	SW-LR	8.72	3.35
Life	1	SW-LR	13.3	1.00
Life	2	SW-LR	9.20	1.45
Life	3	SEQ	6.80	1.96
Life	4	SEQ	5.77	2.31
Life	5	SW-LR	8.00	1.66
Threshold	1	SW-LR	1.38	1.00
Threshold	2	SW-LR	5.04	0.27
Threshold	3	SW-LR	4.75	0.29
Threshold	4	SW-LR	4.03	0.34

Table 6.12: Ward's CVM Performance Results

The performance of CVM is studied by Ward [Ward97] in the same RS/6000 environment that DSDS was tested in. A subset of the Cowichan II [Wilson95] problem set was studied, including Mandelbrot-set generation, the game of life, and image-intensity histogram thresholding. These algorithms are all different from ours but there are some comparable features. The CVM results are summarized in Table 6.12. The times are all in seconds and the "Consistency" column shows the consistency model that produced the best result in each case; "SW-LR" means Single-Writer Lazy-Release and "SEQ" means Sequential. It is interesting to note that the Multi-Writer Lazy-Release consistency was never the best, since it has additional CPU overhead and since for all of the algorithms, care was taken to align bands of rows to

VM pages. Also, all applications use 1024 × 1024 matrices.

The Mandlebrot application is of a similar type to matrix multiply, in that the elements of the result matrix are all calculated independently and bands of rows can be statically assigned to hosts. There are also many differences between the two algorithms, including that only an array of booleans is required for Mandlebrot. Special care was taken to balance the load since elements near the origin of the set can require much more computation than those on the periphery. The speedup achieved is similar to that achieved with DSDS for matrix multiply; however, it must be noted that the Mandlebrot application essentially requires no initial matrix distribution, and the speedups are all relative to the one-host execution of the distributed version, including all CVM overheads.

The Life application is of the same type as SOR, in that bands of rows are statically assigned to hosts and the first and last row of each band is shared with other hosts per barrier-controlled iteration of the main algorithm. The performance is roughly similar also. Again, Life uses only an array of booleans, and speedup is relative to the one-host CVM version. An absolute-time comparison is meaningless since very different work is performed per innermost loop.

Threshold is an example of an application that performs very inefficiently with CVM. The nature of the algorithm is such that a large amount of data distribution occurs relative to the amount of computation per element.

6.7.3 Orca

Orca [Bal91] by Bal implements user-defined passive objects with monitor-like methods in an environment with object replication and remote invocation. The test environment consists of ten MC68020 CPUs connected by a 10 Mbit/sec Ethernet. All computations are performed using integers since the CPUs lack hardware-floating-point support. Three different run-time systems were used, but we will look at the

Algorithm	1-Time	1-Speedup	2-Speedup	4-Speedup	8-Speedup
Matrix Multiply	780.1	1.00	1.99	3.90	7.32
A-P Shortest Path	400.0	1.00	1.95	3.51	5.46
SOR	3071.4	1.00	1.99	3.90	7.31
Game-Tree Search	2069.5	1.00	1.68	3.36	5.56

Table 6.13: Orca Performance Results

Amoeba-RPC results, which also uses point-to-point messages, although with a specially tuned transport system that can handle arbitrarily large messages.

The performance results are shown in Table 6.13, for 1, 2, 4, and 8 hosts with 250×250 matrices for matrix problems. The table includes data for the test problems most similar to those included in this thesis. The speedup numbers are very good; however, all speedup numbers are relative to the one-host execution time of the distributed version written in Orca, rather than a tuned sequential version written in a language with an optimizing compiler. Bal admits that matrix programs could run 3 to 5 times faster in C than Orca. Plus, no one-host optimizations are performed, so all Orca-object operations are performed with RPCs to the local host. This has the effect of making the sequential version unusually slow, which makes the multi-host speedups much better than they would otherwise be.

For the matrix-multiply, a dedicated coordinator process is used that contains the A and B matrices and sends them out on request of worker processes. Apparently, this same structure is used for the 1-host version, so one complete distribution of the A and B matrices is included in the "sequential" comparison version. This makes one wonder exactly what is being measured. The all-pairs shortest-path algorithm is implemented without global barriers using a data-dependency method and a second matrix of the k rows, which allows it to have more parallelism than the version implemented for DSDS. APSP also transports the *entire* matrix to a worker process

for the 1-host comparison version, and unlike with matrix multiply, the contents of the matrix are distributed effectively only once regardless of the number of worker hosts. The tree searching uses a method that propagates cutoff values better.

Also, if DSDS could use arbitrarily large messages, many matrix algorithms would be much more efficient, since entire quadrants (or whatever) of matrices could be migrated in a single large OS/network operation.

6.8 Conclusions

6.8.1 Summary

This chapter has presented distributed implementations of several algorithms and discusses design decisions and tuning methods for the implementations. With the exception of Gaussian Elimination, speedup was achieved for each, sometimes quite good speedup.

Matrix multiply is a fairly simple problem with no actual conflicting memory access (for a system free of false sharing), but it is illustrative nevertheless. Studying it revealed many characteristics of distributed and sequential matrix processing. The sequential version revealed differences between row-wise and column-wise accessing and differences among the compilers tested. Accessing matrices in a row-wise fashion can make processing over ten times faster. The loop optimizations provided by different compilers also have a significant impact on performance.

The distributed version assigns different tiles of the result matrix to the different hosts to compute. Breaking the matrix into "quadrants" and into "bands" was found to give roughly equal performance. Using quadrants requires less data to be transferred and using bands gives slightly better caching effects. FDDI performed significantly better than Ethernet, as expected. A problem was discovered with accessing array elements indirectly, caused by the failure of the compiler to recognize

a valid loop optimization. The problem was resolved by accessing segments of the matrix as vectors. Also, a dynamic-behaviour problem was discovered when allowing the coordinator host to execute the multiplication while remote hosts are still acquiring the matrix data. This overloads the coordinator host by splitting processing between multiplying and protocol handling and slows down the whole computation significantly. With this problem corrected by inserting a barrier, a maximum speedup of 4.6 times was achieved over the best sequential implementation. This figure includes the fan-out and fan-in overheads. The timing results were also found to be very stable with repeated runs of the application.

All-pairs shortest path is an algorithm for finding all shortest paths of a graph given an adjacency matrix. The distributed implementation uses one regular (non-mapped) object per row of the matrix and bands of rows are assigned to be computed by the different hosts. It is an iterative algorithm using a barrier for global synchronization. The sharing pattern is such that on each iteration i, row i must be read by all hosts. Prefetching is implemented in this distributed algorithm to help with the fan-out/fan-in phases and to assist with distributing row i on each iteration. The prefetching improves performance significantly, by 14 per cent with eight hosts. Overall, a maximum speedup of 4.5 times over the sequential implementation was achieved.

Successive overrelaxation is an algorithm that computes the stable state of a matrix where the value of each cell is the average of the four cells that share a side with it. The distributed implementation uses mapped objects and a variant of the algorithm based on processing the array in a "checkerboard" pattern. The matrix is assigned to hosts in bands of rows, and the rows on the borders of the bands are shared between processors. An issue exposed by this algorithm is the impact of the weight of the locking operations, since three locks are acquired and released for every row of every iteration processed. The kernel-based heavy-weight locking added an overhead twice that of using direct, light-weight locking. Prefetching was also applied to the problem

and gave a 12 per cent improvement with eight hosts. Overall, a maximum speedup of 3.2 times over the sequential implementation was achieved.

Gaussian elimination is an algorithm for solving systems of simultaneous equations. Many variations are tried, and prefetching results in significant improvements in performance, but the distributed version does not achieve speedup over the sequential implementation. The algorithm simply requires too much synchronization and sharing to be effectively implemented in a network-of-workstations environment, especially since partial pivoting is implemented. There are several synchronizations required per row of processing.

Othello is a zero-sum board game. The distributed implementation builds a distributed tree structure representing the traversal of the search, down to a certain horizon. This tree is used to allocate work to processes in a left-most-node-first order and to accumulate the results of the search. The game tree can be scanned in a sub-optimal order with the distributed algorithm implemented, because of its dynamic activity. The key issue is the propagation of "cutoff" information, which propagates more slowly when multiple hosts are active in different sub-trees simultaneously. The result is that significantly more moves can be scanned in the distributed version than with the sequential version. Therefore, even though the distributed algorithm achieves a move-scanning speedup of 7.0 times with eight hosts in the best case tested, many more moves were scanned than in the sequential version, resulting in an overall elapsed-time speedup of only 2.8 times. Better tuning of the algorithm itself may increase performance in this application.

For all algorithms, care is needed in designing the distribution, but DSDS provides a good set of facilities for implementing distributed schemes. Other features that have been described but not implemented would increase performance of some algorithms presented and allow other types of algorithms to be handled effectively as well.

6.8.2 Distributed Algorithms

In this section, we consider some more-general issues related to distributed algorithms and networks of workstations than can be illustrated by the algorithms of this chapter. The costs of operations in a distributed system of workstations are fairly well known. The shared-object paradigm papers over these costs, but they must be paid nevertheless. If an algorithm exhibits too high a degree of synchronization or too large a volume of required message passing, then it is going to perform poorly. Networks of workstations also suffer from large system overheads for passing individual messages, both in terms of latency and OS processing.

The applications examined with DSDS were mostly of the $O(n^3)$ type. SOR is actually $O(n^2)$, however it requires a large number of iterations for a solution. Othello is an $O(B^n)$ problem, which is even more demanding. The reason that the complexities of the problems are important is that they limit what can and what cannot be implemented effectively on a distributed system of standard workstations.

One needs to examine the degree of message passing and synchronization of a distributed algorithm and compare that to the amount of computation in order to gain a first approximation of whether distribution will be successful. For example, many matrix algorithms require that their $O(n^2)$ matrix data be fanned out and later potentially fanned in during the course of their computation. So, if an $O(n^2)$ matrix computation were implemented, then it would perform very poorly, because the cost of the distribution would very likely outweigh the speedup from solving sub-parts of the matrix in parallel.

There would be a similar problem with $O(n \log n)$ algorithms that operate on $O(n)$ data. If all of the data needs to be distributed and then collected, then the network time to transmit a single item of data needs to be shorter than the time needed to perform the processing on a single data item. If a single data item takes only a dozen processor instructions times the factor $\log n$, then performance will

likely be poor, except for very large n, since the network/operating-system time to transfer the data will be orders of magnitude larger than the time to execute a dozen processor instructions. If a larger amount of computation is performed per data item, then naturally the break-even point becomes more achievable with a network of workstations.

Similar arguments apply to the degree of required synchronization for an algorithm, keeping in mind that synchronization operations, especially global ones (barriers) are very expensive in high-latency networks of workstations.

A very rough estimate can be constructed for our system of RS/6000s based on the execution times for some of the matrix algorithms. They all performed operations that require only a dozen or a couple of dozen processor instructions on their innermost loops and used 8-byte floating-point numbers. Taking all-pairs shortest path as an example, the sequential version took 419.92 seconds to perform 1024^3 executions of its inner loop, which is 0.39 μsec per inner loop. The distributed version without prefetching on four hosts takes 142.32 seconds to execute, which means that there are 37.34 seconds too much time over its ideal speedup of 4.0 times. This extra time is used up by various overheads, almost all data-transfer and synchronization. Dividing by the data size, this means that it took 17.8 μsec per data item (array element) to distribute and synchronize the data. (The algorithm "distributes" 2×1024^2 data items, 1024^2 initially and then one row of 1024 for each of the 1024 main loops.) Thus, discounting the synchronization, element network-distribution time is 45.6 times greater than the inner-loop time, but there are 1024 times as many inner-loop executions as data-item distributions. Thus, the very rough estimate is that the "complexity" of the algorithm needs to be 46 times as much as the "complexity" of the data distribution. As stated above, this informal analysis does not include synchronization, the processing quanta are a dozen or so processor instructions, and the data items are 8-byte doubles.

A similar analysis can be applied to matrix multiply and SOR. Gaussian elim-

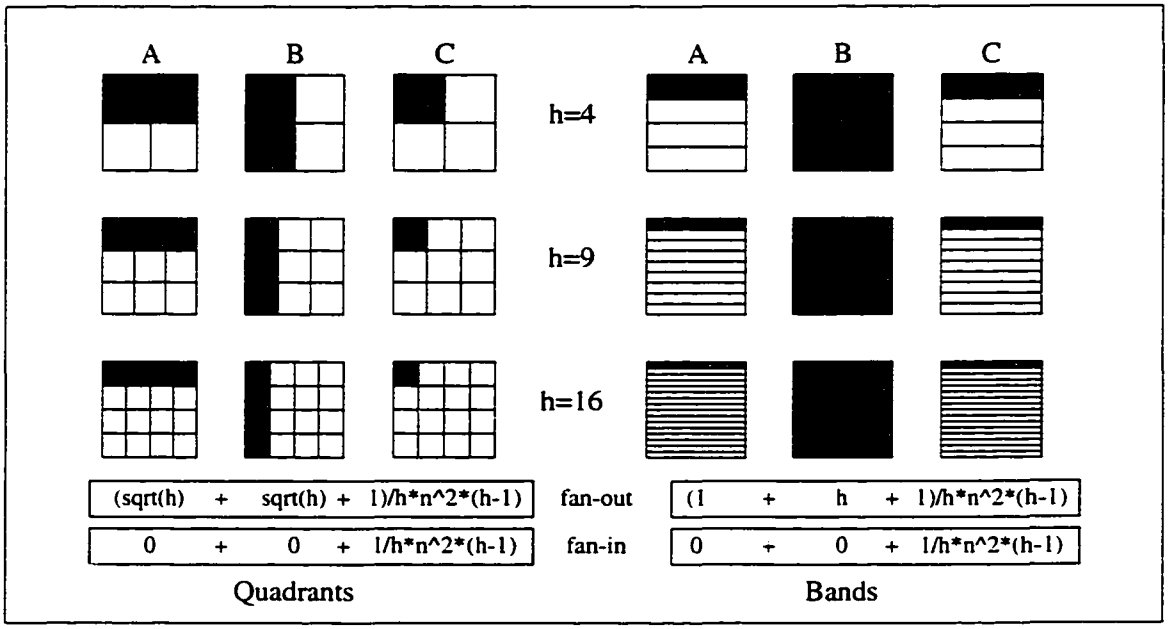


Figure 6.5: Examples of Data Distribution for Matrix Multiply

ination and Othello are too complicated for a simple model. MM has an $O(n^2)$ data-distribution and an $O(n^3)$ computation cost, like all-pairs shortest path. However, the distribution constants for MM are different. For quadrant-oriented MM, the fan-out phase data volume in matrix elements is $\frac{(2\sqrt{h}+1)(h-1)}{h}n^2$, where h is the number of hosts and is a perfect square. Figure 6.5 shows examples of the data distribution for a host, where each matrix is size n^2 and the distribution must be done for $h-1$ hosts, since the data does not need to be distributed to the coordinator host. The data volume for the fan-in phase is $\frac{h-1}{h}n^2$, since the all of matrix needs to be fanned in except for the portion already present on the coordinator. For bands, the fan-out cost is $\frac{(h+2)(h-1)}{h}n^2$ matrix elements and the fan-in phase cost is $\frac{h-1}{h}n^2$. A quadrant-oriented approach for h not a perfect square would have distribution costs between the perfect-square quadrant approach and the band approach.

For the four-host-quadrant version with starting barrier, the fan-out cost is 3.93-million matrix elements, the fan-in cost of 0.786-million, and the inner-loop count

is 1.07-billion executions. It executes this in 41.85 seconds. Compared to the ideal speedup of 4.0, this distributed execution achieves only a 3.17 speedup, which means that it is 8.68 seconds longer than ideal. Almost all this time is presumed to be taken by data distribution. Thus, the element-distribution time is 1.85 μs per data element and the effective inner-loop time is 0.01 μs. The element-distribution time is 185 times greater than the inner-loop time.

For SOR, $\frac{(h-1)}{h}n^2$ matrix elements need to be fanned out, $(2h-2)n$ elements need to be redistributed per iteration, and $\frac{(h-1)}{h}n^2$ elements need to be fanned in. For eight-host execution without prefetching, 0.918-million elements need to be fanned out, 2.23-million exchanged during 156 iterations, and 0.918-million fanned in. The inner loop is executed 164-million times. The execution runs 12.39 seconds longer than the ideal time of 5.40 seconds. Assuming that the additional time is consumed by communication and synchronization, the per-element distribution time is 3.05 μs and the inner-loop time is effectively 0.03 μs, 102 times faster.

The examples in this chapter also provide some indication of techniques that may be generally useful, particularly if they were useful in more than one of the examples. These include: access matrix data in the inner loops in a row-wise fashion, split matrices into individual rows or groups of rows, use prefetching at critical points, use the NO_DATA option to transfer uninitialized objects, modify code to assist loop optimization in the compiler, use the xlC compiler because it is better at loop optimizations than the gcc compiler, place a barrier between the end of a fan-out and the beginning of the main processing, assign most data statically to hosts (after distribution) for the duration of the main computation, and pay less attention to optimizing the less-frequently-executed portions of programs (such as the Othello queue mechanism). Table 6.14 summarizes the application of these techniques and some others.

TECHNIQUE	MM	shortest path	SOR	Gauss	Othello
row-wise accessing	X	X	X	X	
split into rows	X	X	X	X	
pre-grant for fan-out		X	X	X	
prefetch-links for fan-in		X		X	
pre-grant item before barrier		X		X	
pre-invalidate read-once objs		X		X	
light-lock mostly local objs			X		
keep most items locked		X			
NO_DATA for uninitialized objs	X				
coax loop optimizations	X			X	
use xlC	X	X	X	X	X
barrier between init. main	X	X	X	X	X
mostly static data	X	X	X		
not optimize infrequent code					X

Table 6.14: Optimizations Applied to Applications

6.8.3 Future Work

The examples in this chapter have demonstrated the value of DSDS, but further examples would be useful. If time were available, the next algorithm that might be examined is sorting. Although sorting does have an $O(n \log n)$ processing versus $O(n^2)$ distribution problem, the tables might be turned if the distribution problem could be transformed into a parallel network-file-reading solution.

A graph algorithm to attempt might be Graph Isomorphism. This problem is very computationally intensive and there are many available algorithms to choose from, so it may be well suited for parallelization. We have not considered it in any detail, however.

Chapter 7

Conclusion

7.1 Research Summary

7.1.1 Basic DSDS System

DSDS introduces a conceptually straightforward system for managing distributed shared data. The concept is that of an arbitrary dynamic distributed data structure of user-defined objects that can be accessed by all hosts of an application. The system is implemented entirely at the user-process level and is based on individually lockable objects. The necessary primitive operations are provided for allocating/deallocating and locking/releasing the objects. Shared and exclusive locks are supported to allow the most reasonable concurrency control, and barriers are also provided for convenient global synchronization.

Direct memory pointers are used for all distributed objects, which simplifies programming and makes accessing object data direct and efficient. These pointers are also easy to link into dynamic data structures. Memory allocation is performed globally, but this is usually not a performance problem because applications tend to have centralized initializations.

A prototype of the system is implemented in the Unix environment in C. The basic system uses an invalidation protocol based on ownership forwarding. Object locking and data replication/migration are integrated in the protocol for simplicity-of-concept and efficiency. In many cases, the inherent locking is all that is required for concurrency control in distributed applications.

The object-management protocol provides sequential consistency for the objects, and is fully asynchronous to provide maximal responsiveness. Two threads are used per host to allow requests from external hosts to be processed as soon as possible, and the internal locking between the two threads is designed not to hold locks for long periods of time. The protocol is designed to be robust and extensible and to work with an unreliable transport system.

The transport system used is a thin layer on top of UDP/IP. This system is responsive, simple, asynchronous, and tuned for small messages. It can also send messages to any host in a distributed application efficiently and matches the connectionless semantics of the object-management protocol.

7.1.2 Performance Enhancements

There are several potential performance enhancements that can be applied to the basic system, and many were suggested from testing the system. Prefetching is a very popular technique in many systems because it has the potential to eliminate data-request latency. Five methods of prefetching are provided by the DSDS system: single-object explicit prefetching; prefetch links, which fetch a list of objects when the first one is requested; asynchronous granting, which allow a programmer to send an object as if it were a message where appropriate; pre-invalidation, which avoids an invalidation round; and local invalidation, which saves all messages in certain situations.

Non-stalling invalidations, optimistic locking and tagged objects are other mech-

anisms to hide latency. Stalling (waiting for invalidation acknowledgments) can be performed at different times, which allows a tradeoff between efficiency and strictness of consistency. Optimistic locking allows locks to be acquired without the local host being sure that it has the most up-to-date version of the data. This is confirmed asynchronously, while user processing continues. Tagged objects can also eliminate costly global-synchronization operations (barriers) in certain application areas, including iterative matrix algorithms.

Several protocol-level optimizations are introduced also. An update protocol on selected objects can be provided easily by sending asynchronous grants automatically when objects are released. Broadcasting capabilities in hardware can be used to make this action more efficient, without requiring global ordering, atomicity, or even reliability in the broadcast service. Methods for light-weight locking, exploiting semantics, and handling object-owner-locating costs are also introduced.

7.1.3 High-Level Mechanisms and Usage

High-level mechanisms are introduced to make programming more convenient. Matrix processing is a common application for a DSD system. Data in the matrix is divided up into tiles that have different sharing patterns or that are statically assigned to different hosts for processing. Different approaches to abstracting out the low-level mechanisms are considered for matrix processing, but the best overall general approach of those considered is to use mapped objects. Mapped objects allow the elements of a matrix to be accessed directly, allowing full memory speed and full compiler optimization with no false sharing, but the tiles need to be locked explicitly. However, there are frequently occurring cases in which the explicit locking can be inserted conveniently. Array creation and tile-name management are also abstracted.

Another general class of problems involves processing linked dynamic data structures. The DSDS system provides a direct representation of these structures in a dis-

tributed environment and convenient pointers for direct programming. A link-cursor mechanism is proposed to make accessing more convenient by handling locking and unlocking internally with optional automatic prefetching. The mechanism can also be used to provide a method of "no-pass" concurrency control.

Language-level and library-level abstractions are proposed to make programming more convenient. Methods of using the nested nature of block-structured languages to make lock handling more convenient are also proposed. As well, libraries for managing the common object-organization structures of linked-list queues, hash tables, and B-trees are proposed and the design issues are discussed.

Techniques for mapping data structures to DSDS are discussed. Techniques for mapping from other systems to DSDS are also discussed, to demonstrate that the DSDS system provides or can be made to provide the functionality of the other DSD systems.

7.1.4 Test Applications

Distributed implementations of several algorithms are discussed. The discussion includes design tradeoffs that must be considered and methods for tuning implementations. Performance measurements for Matrix multiply, All-pairs shortest path, Successive overrelaxation, Gaussian elimination, and Othello algorithms are obtained and discussed. With the exception of Gaussian elimination, speedup was achieved for each, sometimes quite good speedup.

For all algorithms, care is needed in designing the distribution, but DSDS provides a good set of facilities for implementing distributed schemes. Other features that have been described but not implemented would increase performance of some algorithms presented and allow other types of algorithms to be handled effectively as well.

Comparisons to other DSD systems and more-general issues related to distributed algorithms and networks of workstations are also discussed.

7.2 Contributions to Computer Science

The requirements for a DSD system were set forth in Chapter 1. The properties of DSDS were summarized in the preceding section. This section describes key contributions that correlate to the major requirements in Chapter 1.

There are several primary contributions made by this thesis, all related to the straightforward and efficient implementation of programs using data sharing in a network-of-workstations environment. The system concept proposes a straightforward approach to managing a potentially large dynamic distributed structure of user-defined objects that can be accessed by all hosts of an application. Integrated shared and exclusive locking of independent objects allows full concurrency. Mapped objects and direct-memory pointers are used to make accessing object data and metadata efficient and convenient.

Prefetching allows object-access latency to be hidden by making object data available at the local host shortly before it is needed. Several methods are proposed and implemented for the system. Optimistic-locking and tagged-object methods and protocol implementations are also proposed to hide latency.

An array-processing methodology of splitting arrays into potentially irregular tiles that have different sharing patterns is also proposed. This method makes use of mapped objects to make array-element accessing very efficient, even in the memory-object based environment of DSDS. A link-cursor mechanism with built-in concurrency control is also proposed. The thesis also contains a performance study of several distributed applications, demonstrating the effectiveness of DSDS.

7.3 Future Work

Possible future work includes the implementation of the mechanisms described in this thesis that are not yet implemented in the prototype system. These mechanisms

include optimistic locking, tagged objects, and several protocol-level mechanisms. A careful study of the behaviour of these mechanisms should be pursued to determine their strengths and areas of best application.

Issues of nested objects and data granularity should also be studied. Large objects can have a natural hierarchy where it makes the most sense to access the whole object in some instances and small parts of it in others. A tradeoff exists between granularity and performance/system overhead. Small objects require more overhead to process but are less likely to incur conflicting simultaneous access.

Object-migration policies should be studied in more depth and automatic tuning mechanisms should be developed to allow the protocol and object migration to operate as efficiently as possible with as little user involvement as possible.

Additional programming-language extensions should be devised, implemented, and studied to take complete control of object-lock management. The system should retain as much efficiency as possible relative to user-controlled lock management.

The use of DSDS in environments with high-bandwidth, low-latency communication should be studied. The network-of-workstations environment is not tuned toward providing the most effective means of implementing distributed-shared-data systems. The savings in message-passing and protocol-processing overheads and latency hiding that the DSDS system can provide compared to other systems can be leveraged in environments with greater parallel-processing and shared-data capabilities.

Fault-tolerance should also be investigated. The nature of the system makes it possible to distribute multiple copies of the latest version of each object on multiple hosts to recover from failed hosts, but complicated issues may arise in recovering a consistent set of objects and a consistent state of computation.

Finally, at present only one person has written applications for the DSDS system, so the usability of the system by programmers in general is unknown. It would be an interesting but a very demanding project to determine how difficult it is to write

programs for DSDS versus other distributed/parallel environments, and how efficient the programs produced are in comparison to those produced by an expert DSDS programmer.

7.4 Conclusion

This thesis proposes the DSDS approach and system to implementing distributed-shared-data algorithms. The concept is based on the sound principle of combining user-defined memory objects with user-controlled locking and optimizations. A prototype system is implemented that provides an invalidation-based shared-object consistency protocol. Many mechanisms are proposed to enhance the performance of the basic system, including prefetching, which is implemented. Many higher-level mechanisms are also proposed to simplify development of applications for the system. Performance figures are obtained for several test applications, demonstrating the good performance of the system in a variety of application areas.

Appendix A

Prototype-Implementation Details

A.1 Design Restrictions

The prototype implementation of DSDS has a number of limitations. A very visible restriction is on the size of user objects, as only objects up to about 64K in size may be used. This is because objects must fit into a single transport-layer datagram in order to simplify the object-management protocol. If the transport layer did not restrict datagram sizes to 64K, then user objects could be arbitrarily large. The consequence of the restriction is that otherwise large user objects will need to be split into smaller chunks in order to be used with the prototype system.

The DsdsInit() call needs to be augmented with three integers, svArenaSize, mgmtArenaSize, and globalDirEnts. The svArenaSize argument gives the memory size in bytes that should be reserved for holding global shared variables and their control information. The mgmtArenaSize argument gives the memory size to reserve for local-management control information. These should be the maximum size expected for the run, and, unfortunately, these values may be rather difficult to predict. In a production implementation of the DSDS system, these arguments would not be necessary, but in a prototype environment, knowing the maximum arena size in advance

simplifies the coding. This total amount of shared virtual memory will be mapped into the local address space for sharing. Note that this argument is allowed to be very large since intelligent operating systems initialize virtual memory tables for mapping but do not actually allocate real memory until the virtual memory is accessed. The globalDirEnts gives the number of entries ("names") to reserve for use with the SvRegister() and SvLookup() calls. Requiring the user to specify this at the start makes it possible for the system to reserve a simple array for the lookup table.

In the prototype system, objects are designed to be maintained on a "per-application" basis. Separate applications that start executing at different times are not allowed to share objects (although this is only a limitation of the start-up and transport mechanisms and is not a restriction of the DSDS system itself). The model of concurrency in applications is multiple independent processes that communicate via the shared-variable mechanism and perhaps other interprocess-communication techniques. A single application consists of an arbitrary number of user processes ("peers"), up to some fixed limit. These processes may run different programs that are designed to share objects using the DSDS interface. The prototype system provides no means for dynamically creating new user processes for an application during execution.

Normally, each process will be run on a different host in a distributed system of workstations, although multiple processes may run on a single host as if they were on separate hosts if neither requires full CPU utilization. There is no special sharing advantage to running "peers" on the same machine, as physically-shared-memory techniques are not used in this case, although this could be implemented without fundamental changes. The number of processes in an application is specified at the time that the application is started. All processes of an application share objects equally (i.e., there is no security of any kind).

To start up the system, a manager program is executed that sets up and starts listening to a special, well known network port for connections from user processes.

This program is given as an argument the number of processes that will be participating in the application, and it waits until that number of user processes have connected to it, at which point it assigns all machines an abstract, simple address from 1 to N and returns the internet-address/port/simple-id mapping information for all application processes to each process. After this, the distributed application starts running on the fixed number of application processes (hosts), and each process possesses the required information to communicate with each other process.

The protocol is a write-invalidation protocol in which most process interactions are based on the request/response paradigm. It is designed to be implemented with two concurrent counterparts (master and slave threads) for each host on top of an unreliable connectionless transport protocol.

A.2 Unix-Implementation Implications

Each application processes has two threads, a master thread that executes the user code, and a slave thread that handles incoming requests from other processes in the system. A slave thread is needed to ensure timely responses to the incoming requests.

The master and slave 'threads' of each application 'process' are actually implemented using separate Unix processes. The slave process is started using fork() after the master process has initialized. The "shm" group of Unix system calls [Curry88, pp. 143–147] allocate a single shared-memory segment to share between the two 'threads'. A Unix semaphore operated with the "sem" group of Unix system calls [Curry88, pp. 140–143] is used to control access to the shared-memory segment.

A threads package could have been used for this, and perhaps should have been used, since there would be less system overhead in switching between the master and slave processes (threads) and for synchronizing between them. Thread packages were not very common or standard when the system was first implemented.

For simplicity, the shared memory is allocated when the system starts and before the fork() call is made. Because of the limitations of some Unix systems, multiple segments of some maximum-limited size may need to be allocated rather than one large segment. For example, SunOS imposes a limitation of 1 MB for shm segments. This may pose problems with large allocations, although more modern systems have much larger limits.

Only a single semaphore is used to control access to the shared-memory segment. The protocol was originally designed to run using a single thread, and it still can if given the correct compile-time configuration. The single semaphore is used to serialize the execution of the master and slave processes when they are "inside" the DSDS code: only one is allowed to be active at a time. This is for simplicity. At all points where the execution of a thread is suspended waiting to receive a message from a peer (for a reply, for example), the semaphore-lock is released and is re-acquired after the required message is received. The code is organized so that the shared data structures will be consistent whenever the semaphore-lock is released. Running the protocol with a single thread will eliminate all context-switching overheads but makes the responsiveness of the protocol to asynchronous requests arbitrary. If the user code performs a large amount of synchronization, then it may be better to use only one thread.

The transport system uses UDP/IP sockets, with a very thin additional layer of custom software. The transport service is therefore unreliable, and messages can arrive out of order. UDP was chosen since it is ubiquitous among Unix workstations and since TCP/IP would not be practical because a fully-connected graph of connections would be necessary between an arbitrary number of hosts. Thus, the implemented object-management protocol must be able to successfully deal with lost, duplicated, and reordered messages.

Bibliography

[Amza96] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. "TreadMarks: shared memory computing on networks of workstations." *IEEE Computer*, Volume 29, Number 2, pp. 18–28, February 1996.

[Ananda91] A.L. Ananda, B.H. Tay, E.K. Koh. "ASTRA - an asynchronous remote procedure call facility." *Proc. 11th Int'l Conf. on Distributed Computing Systems*, IEEE, pp. 172–179, 1991.

[Anderson95] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. "A case for networks of workstations: NOW." *IEEE Micro*, February 1995. Web Page: "http://now.cs.berkeley.edu/Case/case.ps".

[ANSI83] American National Standards Institute. *Military Standard Ada Programming Language*. ANSI/MIL-STD-1815A-1983, 1983.

[Bakken93] David E. Bakken, Richard D. Schlichting. "Supporting Fault-Tolerant Parallel Programming in Linda." *IEEE Trans. on Parallel and Distributed Systems*, Volume 6, Number 3, pp. 287–302, March 1995.

[Bal89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. "Programming languages for distributed systems." *ACM Computing Surveys*, Volume 21, Number 3, September 1989.

[Bal91] Henri E. Bal. *Programming Distributed Systems*. Hemel Hempstead, England: Prentice Hall Int'l, 1991.

[Bennett90] J.K. Bennett, J.K. Carter, W. Zwaenepoel. "Munin: Distributed shared memory based on type-specific memory coherence." *Proc. Second ACM Symp. on Principles and Practice of Parallel Programming*, ACM, pp. 168–176, 1990.

[Bershad91] B.N. Bershad and M.J. Zekauskas. *Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors.* CMU Report CMU-CS-91-170, Sept. 1991.

[Bershad93] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. "The Midway distributed shared memory system." *Proc. IEEE COMPCON*, IEEE, pp. 528–537, 1993.

[Bruce92] Craig S. Bruce. *Design and Implementation of an Instructional Operating System*. Master's Thesis, Technical Report TR92-071, Faculty of Computer Science, University of New Brunswick, Canada, August 1992.

[Burns93] Alan Burns and Geoff Davies. *Concurrent Programming*, Addison-Wesley Publishers Ltd., 1993.

[Carriero86] N. Carriero, D. Gelernter, and J. Leichter. "Distributed data structures in Linda." *Proc. ACM Symp. on Principles of Programming Languages*, ACM, 1986.

[Chen91] R. C. Chen and P. Dasgupta. "Implementing Consistency Control Mechanisms in the Clouds Distributed Operating System." Proc. of the 11th Int'l Conf. on Distributed Computing Systems, May 1991.

[Cormen90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, 1990.

[Curry88] David A. Curry. *Using C on the UNIX System*. Sebastopol, California: O'Reilly & Associates, Inc., 1998.

[Delp89] G.S. Delp. *The Architecture and Implementation of MemNet: An Experiment on High-Speed Memory Mapped Network Interface*. Ph.D. Thesis, University of Delaware, 1988.

[Dubois86] M. Dubois, C. Scheurich, and F.A. Briggs. "Memory access buffering in multiprocessors." *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 434–442, 1986.

[Duggan91] Dominic Duggan, Neil Coburn. *CS354 Software Systems Course Notes*. University of Waterloo, Canada, 1991.

[Falsafi94] Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ioannis Schoinas, Mark D. Hill, James R. Larus, Anne Rogers and David A. Wood. *Application-Specific Protocols for User-Level Shared Memory*. Technical Report CS-TR-94-1239, Computer Sciences Department, University of Wisconsin, Madison, 1994.

[Feeley92] Michael J. Feeley and Henry M. Levy. "Distributed shared memory with versioned objects." *Proc. of the Seventh Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92)*, pp. 247–262, October 1992.

[Fleisch89] B. D. Fleisch and G. J. Popek. "Mirage: A Coherent Distributed Shared Memory Design". *Proc. of the 12th ACM Symp. on Operating Systems Principles*, pp. 221–223, December 1989.

[Gentleman81] W. Morven Gentleman. "Message passing between sequential processes: the reply primitive and the administrator concept." *Software–Practice and Experience*, Volume 11, pp. 435–466, 1981.

[Geist94] Al Geist, Adam Begulin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. Cambridge, Massachusetts: MIT Press, 1994.

[Gharachorloo90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. "Memory consistency and event ordering in scalable shared-memory multiprocessors." *Proc. 17th Ann. Int'l Symp. on Computer Architecture*, ACM, pp. 15–26, 1990.

[Goodman89] J.R. Goodman. "Cache consistency and sequential consistency." Technical Report 61, IEEE Scalable Coherent Interface Working Group, IEEE, 1989.

[Hermannsson94] Gudjon Hermannsson and Larry Wittie. "Optimistic synchronization in distributed shared memory." *Proc. 14th Int'l Conf. on Distributed Computing Systems*, IEEE, June 1994.

[Hutto90] P.W. Hutto and M. Ahamad. "Slow memory: weakening consistency to enhance concurrency in distributed shared memories." *Proc. 10th Int'l Conf. on Distributed Computing Systems*, IEEE, pp. 302–311, 1990.

[Keleher92] P. Keleher, A.L. Cox, and W. Zwaenepoel. "Lazy release consistency." *Proc. 19th Int'l Symp. on Computer Architecture*, ACM, pp. 13–21, 1992.

[Keleher96a] Pete Keleher. *CVM: The coherent virtual machine*. Technical Report, University of Maryland, November 1996. Web Page: "http://www.cs.umd.edu:80/projects/cvm/".

[Keleher96b] Peter J. Keleher. "The relative importance of concurrent writers in weak consistency models." *Proc. 16th Int'l Conf. on Distributed Computing Systems*, IEEE, pp. 91–98, 1996.

[Krishnaswamy91] V. Krishnaswamy. *A Language Based Architecture for Parallel Computing*. Ph.D. Thesis, Yale University, 1991.

[Lamport78] L. Lamport. "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM*, Volume 21, pp. 558–564, July 1978.

[Lamport79] Leslie Lamport. "How to make a multiprocessor computer that correctly executes multiprocess programs." *IEEE Transactions on Computers*, Volume C-28, pp. 690–691, Sept. 1979.

[Lenoski92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. "The Stanford Dash multiprocessor." *IEEE Computer*, Volume 25, Number 3, pp. 63–79, March 1992.

[Li89] Kai Li, Paul Hudak. "Memory coherence in shared virtual memory systems." *ACM Transactions on Computer Systems*, Volume 7, Number 4, pp. 321–359, November 1989.

[Lipton88] R.J. Lipton and J.S. Sandberg. *Pram: A Scalable Shared Memory*. Technical Report CS-TR-180-88, Princeton University, September 1988.

[MacLennan87] Bruce J. MacLennan. *Principles of Programming Languages (Second Edition)*. New York: CBS College Publishing, 1987.

[Maekawa87] Mamoru Maekawa, Arthur E. Oldehoeft, Rodney R. Oldehoeft. *Operating Systems: Advanced Concepts*. Menlo Park, California: The Benjamin/Cummings Publishing Company, Inc., 1987.

[Mowry94] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. Ph.D. Thesis, Stanford University, Computer Systems Laboratory, March 1994. Web Page: "http://www.eecg.toronto.edu/~tcm/thesis/thesis_tech.html".

[O'Farrell94] William G. O'Farrell and H.L. Operowsky. *ABC++ User's Guide*. IBM Canada, 1994.

[Parsons97] E. W. Parsons, M. Brorsson, and K. C. Sevcik. "Predicting the performance of distributed virtual shared-memory applications." *IBM Systems Journal*, Volume 36, Number 4, p. 527, 1997. Web Page: "http://www.almaden.ibm.com/journal/sj/364/parsons.html".

[Peterson96] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. San Francisco: Morgan Kaufmann Publishers, Inc., 1996.

[Protić96] Jelica Protić, Milo Tomašević, and Veljko Milutinović. "Distributed shared memory: Concepts and systems." *IEEE Parallel & Distributed Technology: Systems & Applications*, Volume 4, Number 2, pp. 63–79, Summer 1996.

[Rees91] Stephen A. Rees and James P. Black. "An experimental investigation of distributed matrix multiplication techniques." *Software—Practice and Experience*, Volume 21, Number 10, pp. 1041–1063, October 1991.

[Saltzer84] Jerome H. Saltzer, David P. Reed, and David D. Clark. "End-to-end arguments in system design." *ACM Transactions on Computer Systems*, Volume 2, Number 4, pp. 277–288, November 1984.

[Sandhu93] Harjinder S. Sandhu, Benjamin Gamse, and Songnian Zhou. "The shared regions approach to software cache coherence on multiprocessors." *Proc. 4th ACM Symp. on Principles and Practices of Parallel Programming*, ACM, May 1993.

[Sandhu95] Harjinder S. Sandhu. "Algorithms for dynamic software cache coherence." *Journal of Parallel and Distributed Computing*, Volume 29, Number 2, 1995.

[Schaeffer93] Jonathan Schaeffer, Duane Szafron, Greg Lobe, Ian Parsons. "The Enterprise model for developing distributed applications." *IEEE Parallel & Distributed Technology: Systems & Applications*, Volume 1, Number 3, pp. 85–96, August 1993.

[Tanenbaum92] Andrew S. Tanenbaum. *Modern Operating Systems*. Englewood Cliffs, N.J.: Prentice Hall, Inc., 1992.

[Tanenbaum95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Englewood Cliffs, N.J.: Prentice Hall, Inc., 1995.

[Thekkath91] Chandramohan A. Thekkath, Henry M. Levy. *Limits to Low-Latency RPC*. Technical Report 91-06-01, University of Washington, U.S.A. 1991.

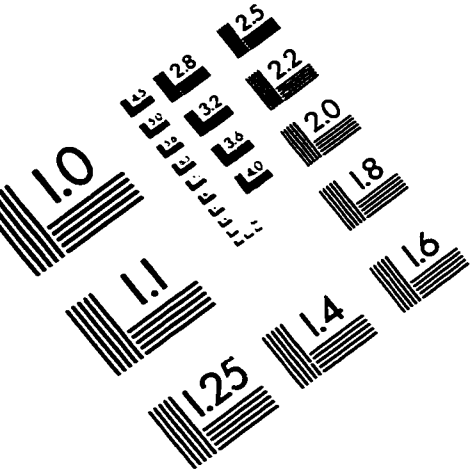
[Waldo94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. *A Note on Distributed Computing*. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., November 1994.

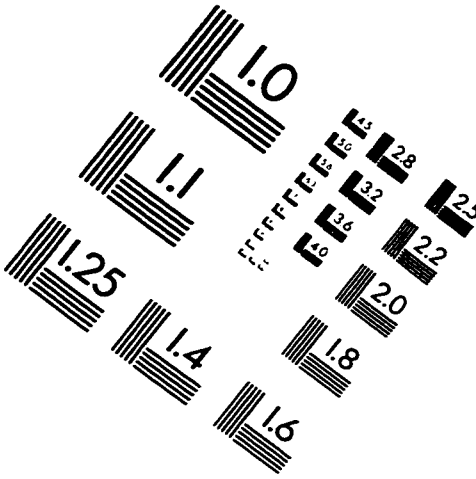
[Ward97] Paul A.S. Ward. *A Comparison of Distributed Shared Memory and Message Passing*. CS 756 Student Research Paper, University of Waterloo, April 1997.

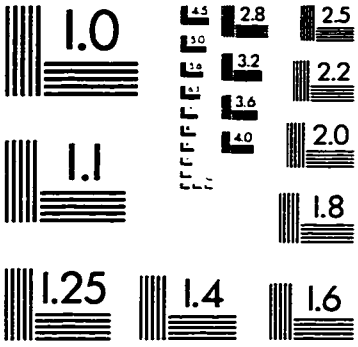
[Wilson93] Andrew W. Wilson Jr., Richard P. LaRowe Jr., and Marc J. Teller. "Hardware assist for distributed shared memory." *The 13th International Conference on Distributed Computing Systems*. IEEE, pp. 246–255, May 1993.

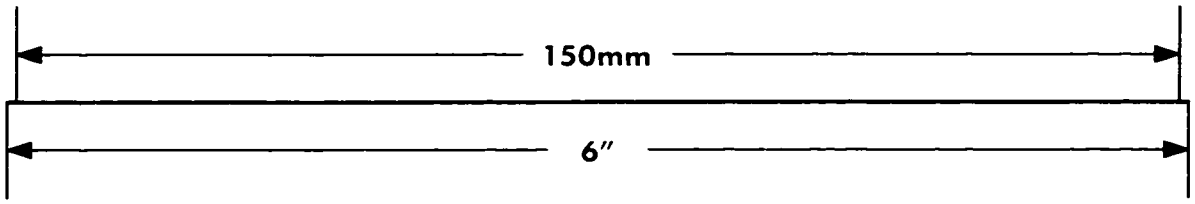
[Wilson95] Gregory V. Wilson and R. Bruce Irvin. *Assessing and Comparing the Usability of Parallel Programming Systems*. Technical Report, University of Toronto, 1995.

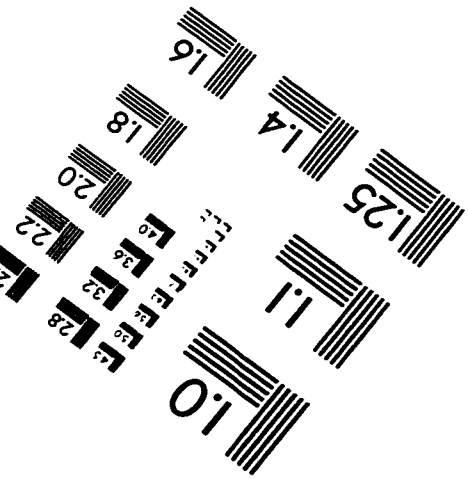
IMAGE EVALUATION
TEST TARGET (QA-3)

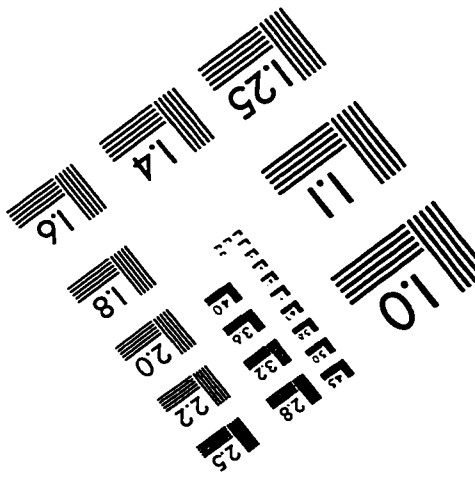












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved