ADSMITH: A Structure-Based Heterogeneous Distributed Shared Memory on PVM

Institute of Computer Science
National Tsing Hua University

Adviser: Prof. Chung-Ta King
Student: Wen-Yew Liang

June 1994

Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Chung-Ta King, for great inspiration. Special thanks to all the members of Parallel and Distributed System Laboratory for their support and care. Finally, I would like to thank my family, especially Mother. Without their love and encouragement, I wouldn't accomplish this study.

Contents

List of Tables

List of Figures

Abstract

Message passing is the most often used communication model in distributed-memory systems. With message passing, programmers are forced to manage the data flows explicitly. They have to know where a piece of data is located and when to set up the send/receive pair between two communicating entities. Such a task is tedious and error-prone. On the other hand, with a virtual or physical shared memory, programmers can use shared variables for communication among concurrently executing threads — just as in traditional sequential programs. Programmers need not manage the shared data explicitly. Thus, programs can be developed more easily.

To provide a shared-memory environment on top of distributed-memory systems, the concept of distributed shared memory (DSM) was proposed and soon attracted wide spread attention. Current implementations of DSMs mostly concentrate on the hardware and/or operating system layers. Problems with such implementations are that they are difficult to support heterogeneous systems, expensive in developing, and difficult to port to different architectures.

In this thesis, we propose a new concept of building DSMs, called the *structure-based* approach. In a structure-based DSM, the shared memory consists of many shared objects. Each object in turn consists of many structure units, which contain logically related data. Shared-memory accesses are performed on the structure units. Structure-based DSMs can be implemented at higher layers such as in libraries. The resultant environment is thus very system independent, portable, inexpensive to build, and flexible. Controlling shared data partition and allocation can be very easy. Most importantly, the environment can support heterogeneous systems.

We have implemented a structure-based DSM, called *ADSMITH*. ADSMITH is built on top of PVM at the library layer using traditional C language. In the thesis we will examine different design considerations and study various strategies to developing DSMs. We will show how we arrive at the current design for ADSMITH. Usages of ADSMITH and its implementations will be introduced, and some preliminary performance results will be presented. The results show that even though ADSMITH is implemented on top of PVM, programs running on ADSMITH can achieve a comparable performance as those running directly on PVM.

Chapter 1

Introduction

1.1 Architectures of Parallel Computers

There are two general classes of parallel systems: *distributed-memory* systems and *shared-memory* systems. In a distributed-memory system, each node has a local memory, unshared with other nodes. Examples include multicomputers such as Ncube-2, Paragon, and CM-5, and workstation clusters, such as DEC Alpha Farm, IBM SP-1, and HP Meta. Communication among the nodes in such a system is done by passing messages. Due to this modularity, distributed-memory architecture is easier to construct and has a higher degree of scalability. At the programming level, message passing is the most popular programming style on these systems. With message passing, programmers are forced to manage the data flows explicitly. They have to know where a piece of data is located and when to set up the send/receive connection between two communicating entities. Such a task is tedious and error-prone. This also makes it difficult to translate a sequential program into a parallel one.

In shared-memory systems, nodes have access to a shared physical memory, through which communication among the nodes is done. Unfortunately the shared memory tends to be the system bottleneck and thus these systems usually have a lower degree of scalability. At the programming level, programmers can use shared variables for communication among the concurrently executing threads — just as in traditional sequential programs. With shared memory, programmers need not manage the shared data explicitly. Thus, programs can be developed more easily with shared memory.

From the above discussion, we can see that combining distributed-memory architecture together with shared-memory programming is a right choice for parallel computers. Based on this observation, *distributed shared memory* (DSM) was proposed and has attracted much attention recently [4].

1.2 Approaches to Distributed-Shared Memory

A DSM provides a logically shared memory on top of a network of computers without using a physically shared memory. Previous approaches to DSMs usually partition the shared-memory addressing space into logical pages, which are distributed to the nodes in the system. Through a memory manager, the nodes can access to any page in the shared addressing space, with the physical local memory also serving as the cache of pages. Such a DSM system will be referred to as a *block-based* DSM in this thesis, because the logical shared memory is partitioned into fix-sized blocks.

Block-based DSMs are often taken as an extension of traditional virtual memory systems, and thus are usually implemented at the hardware and/or operating system layers. One advantage of such an implementation is transparency. The fact that the memory system is distributed across different computers is totally hidden from the users. User applications developed previously to run on single computers can now be executed in the DSM system without change, only to see a larger virtual space. Users can also treat the DSM system as a shared-memory multiprocessor, with a much slower communication speed.

However, there are several problems with the block-based DSMs. For block-based DSMs, the size of the blocks (i.e., the page sizes) is a very important design parameter. A larger block size increases the delay in transferring the blocks and the possibility of false sharing. A smaller block size results in frequent transmissions on blocks and thus increases the block transfer overhead. The cost to maintain the block status will also increase. Choosing an appropriate block size is a very difficult task, which depends not only on the system characteristics but also on the applications. So far, there is no systematic way of choosing this parameter.

Implementing block-based DSMs at the hardware and/or operating system layers also has the problem of design complexity. Such an implementation needs to modify existing hardware and operating system, which requires a tremendous effort. The cost to deploy and maintain such systems could be high. Finally, these implementations are usually system dependent. Different computers have different page sizes, byte ordering, and memory management methods. Thus it becomes very difficult when the block-based DSMs are to be implemented on top of heterogeneous systems [12].

Another approach to DSM is *structure-based*. A structure-based DSM partitions the shared memory into small units, each unit contains logically related data. Structure-based DSMs most often are implemented at the language/compiler or library layers.

Structure-based DSMs usually require some modifications and adjustments at the users' end — either to the programming model, language, or style. Thus, transition from sequential computers to such an environment is not so trivial and transparent. Since structure-based DSMs are implemented at a rather high layer in a computer system, performance is of course not as good as that of the blocked-based DSMs. However, structure-based DSMs do offer some unique and important features.

Just because structure-based DSMs are implemented at higher layers in a computer, the resultant environment could be very system independent and flexible. The DSMs can be directly implemented on top of existing generic communication subsystems, such as PVM [7], TCP/IP, or DCE. Not only that system development efforts can be reduced dramatically, but also system porting to different architectures is straightforward. More importantly, the environment can support heterogeneous systems very easily.

Flexibility of structure-based DSMs also facilitates data allocation and access. In structure-based DSMs programmers have a larger control over how data are distributed according to the characteristics of the application. Also, letting programmers determine important parameters, such as the block size, access method, communication mechanism, memory consistency model, etc., can be done easily in structure-based DSMs.

Based on the above discussions, we believe the structure-based approach to DSMs is very promising. In this thesis we will study this approach in more details and introduce an implementation of such a DSM, called *ADSMITH*.

1.3 Overview of ADSMITH

ADSMITH (A DSM In Tsing Hua university) is a structure-based DSM built on top of PVM. The system is implemented at the library layer, and programmers use the system through traditional C language. Note that using traditional language and programming style is one distinct feature of ADSMITH from previous structure-based implementations, e.g., DoPVM (using object-oriented programming) and Linda (using an associative memory.)

ADSMITH inherits all the benefits of structure-based DSMs. With its implementation on top of PVM, ADSMITH supports heterogeneous systems and allows easy porting to different architectures, as long as PVM is supported in the architecture. ADSMITH was developed with flexibility in mind. Many options are supported in the system to allow the users to specify the properties of each declared objects, their access methods, data distribution, memory consistency policies (both strict and relaxed memory access are supported), and communication mechanisms (e.g., message passing or shared variable.) ADSMITH also provides synchronization primitives for shared-memory parallel programming.

As mentioned in the previous section, a major drawback of structure-based DSMs is the large overhead involved. Not every application can be beneficially executed on ADSMITH. Therefore, we must be very careful in choosing the applications to run on our system. Since ADSMITH is built on top of PVM, we believe any application executable on PVM with appreciable performance can be run on our system. Later we will show that adding a layer of DSM on top of PVM in ADSMITH does not compromise the performance. We will compare applications running on our system and on the bare PVM.

1.4 Organization of the Thesis

To design a DSM environment, one must address a number of issues such as low-level system supports, user interface, system architecture, and memory management policies (e.g., data duplication, write policy, coherence protocol, and consistency model.) In this thesis, we will investigate these issues and describe the implementation of ADSMITH with some primary performance results. The rest of the thesis is organized as follows. In Chapter 2, we introduce the design goals of ADSMITH and implementation strategies. In Chapter 3, the user interface of ADSMITH is introduced. Implementation details of this system are described in Chapter 4. In Chapter 5, we show some preliminary performance results of ADSMITH. Finally, we conclude this thesis with future works in Chapter 6.

Chapter 2

Goals and Design Strategies

2.1 Goals

The development goals of ADSMITH are as follows:

1. Provide users a logical shared-memory environment on top of distributed-memory systems.

2. Minimize users' involvement in shared-memory manipulation and allow easy development of parallel applications.

3. Support heterogeneous environments.

4. Make the system highly portable to different systems.

5. Make the system more flexible in adjusting the performance level.

6. Minimize the cost in implementing, deploying, and maintaining the system.

2.2 Design Tradeoffs and Strategies

ADSMITH is a structure-based DSM. We have discussed briefly the pros and cons of structure-based DSMs in Chapter 1. In this section, we will look more closely into the considerations and tradeoffs involved in designing such a system. Based on these discussions we will also show how the design parameters of ADSMITH are chosen.

Layers of DSM Implementations

One of the most important considerations in a DSM implementation is at which layer the DSM is to be built. In general, DSMs can be implemented at several different layers, as shown in Table 1.

Layer	Examples
Hardware	Memnet [15], DASH [19], KSR1 [18]
Operating system	Mach [13], CLOUDS [16], Choices [17]
Language	Fortran-D [11], HPF [10], DoPVM [8], Linda [9]
Library	Munin [20], Agora [14]

Table 1: Different layers of DSM implementations

As mentioned in Section 1.2, DSMs implemented at the hardware or operating system layers have the advantages of transparency and efficiency. However, these implementations are very system dependent, which implies high development complexities and difficulties in porting to different systems and supporting heterogeneous systems. In addition, the efforts involved in maintaining the system and adding new functions will also be high.

For DSMs implemented at the programming language layer, new compilers or preprocessors must be developed. Issues involved in automatic data partition and distribution, parallelism extraction, and communication optimization are still quite open. Related techniques are still immature and the development efforts are enormous. These all increase the difficulties in using and implementing the systems.

As mentioned in Section 1.2, DSM implementations at the library layer suffer from the performance problem. However, the advantages gained from this kind of implementation could outweigh the loss. For example, programmers can use traditional languages augmented with DSM libraries to develop parallel applications. DSMs implemented at the library layer also are very system independent, highly portable, flexible in determining system parameters for performance tuning, and easy to implement and maintain. Most importantly, the DSM system can be built on top of a heterogeneous environment very easy.

Based on the above considerations, we decided to implement ADSMITH at the library layer. This decision then leads to the second important consideration in a DSM implementation: Where and how should the communication subsystem be implemented? The communication subsystem provides necessary communication supports for DSM operations and activities. For ADSMITH, we choose PVM as the communication subsystem [7]. There are several considerations leading to this decision.

First, ADSMITH will be very portable if it is implemented on top of PVM. This is because PVM has been ported to many systems and is being widely used. Second, most heterogeneity processing operations, such as data format conversion, can be handed over to PVM, which uses XDR [21] to convert data during message packing and processing [6].

Third, since network details and process management problems are handled by PVM, ADSMITH can be independent of the underlying systems as much as possible. As a result, its implementation is very easy and the development cost and time are greatly reduced. This also simplifies system maintainance and extension efforts.

Memory Hierarchy

DSMs usually maintain a two-level memory hierarchy: local cache and remote shared memory. A run-time system is needed to help manage the shared data which are distributed across different nodes. Upon accessing a shared datum non-existing in the local host, the run-time system should fetch the datum from the remote host and store it into the local cache. For library-layer DSMs, such as ADSMITH, the run-time system can be conveniently implemented as daemon processes. In ADSMITH one daemon is spawned for each host. The daemon is responsible for shared-data manipulations on the local host.

There are several ways of implementing the communications between the daemon and local processes. Some implementations, such as DoPVM [8], use UNIX pipe and shared memory to reduce the overhead in communication. On the other hand, the daemons of ADSMITH communicate with local application processes only through PVM so as to be independent of low-level systems as much as possible. To overcome the performance loss due to a higher level implementation of interprocess communication[1], ADSMITH maintains a cache in each application process's addressing space.

Thus, we reconstruct the shared-memory system into two layers. The upper layer is called the *Process Cache Layer* (PCL), which is inside the application process's addressing space. The lower layer is called the *Logical Shared Memory Layer* (LSML), which is formed by all daemons. In this way, each memory access from a process has only one entry point and one exit point above LSML and data in this layer are strictly consistent.

Data Granularity

Data granularity is the unit size of the data during internal data allocation and external transmission between physically distributed memories. In block-based DSMs, the granularity is a system-defined parameter, which is difficult to change. In structure-based DSMs, users have the freedom of specifying the structure unit and the associated data granularity. Another important benefit is that users have a larger control

[1] In this thesis, *interprocess communication* refers to the communication between processes on the same host, while *network communication* means the communication between different hosts.

over how shared data are distributed — based on the application behavior, data access pattern, and machine characteristics. This capability can help to reduce load imbalance and communications among the hosts. Note that it is also possible to use compilers or run-time systems to perform data partition and allocation automatically in structure-based DSMs.

Data Mapping Directory

A critical function in a DSM is keeping track of the status of shared data. Usually such information is stored in a directory. The directory can be centralized or distributed. For DSMs with a centralized directory, the directory is queried for every data access which is not found in the local host. As a result, the centralized directory becomes a system bottleneck. In ADSMITH, we use distributed directories.

In a distributed implementation each daemon will hold a directory, in which information such as the characteristics of the shared object, its owner, the associated valid bit, and data status is recorded. Since such information is stored locally, accesses can be very fast. One problem with this approach is that changing the owner of a shared object needs a broadcast to inform all daemons, which may cause a large amount of data transmission. Thus, we fix the owner of each shared object in ADSMITH. A more efficient directory management scheme will be proposed as a future work in Section 6.2.

Flexibility

As mentioned previously, since ADSMITH is implemented at a very high layer, the resultant environment is very flexible. We will study this feature in more detail here. First, ADSMITH is very flexible in specifying the access characteristics of shared objects. Different objects may require different allocation methods. It will be inadequate to use the same data allocation and access method for shared objects with different characteristics. For example, if we require that every shared object be stored in one host only, then the access latency will be long for those data that are read frequently but written infrequently. In this case, the shared object should be duplicated and stored in the hosts which need to read the object. On the other hand, if we require that all shared objects be duplicated to where they are accessed, then again a heavy network traffic will be induced for those data that are written frequently.

The access characteristics can be specified in ADSMITH as MULTICOPY/SINGLE-COPY, CACHE/NOCACHE, MOSTREAD/MOSTWRITE, and LOCAL. These specifications will be treated by the system as directives to select suitable access or allocation methods. Using appropriate allocation and access schemes for shared objects of different characteristics, the system performance can be greatly improved. For a detail description of the specifications, please refer to Section 3.2.

Second, ADSMITH is flexible in choosing the programming models. Not all applications are suitable for the shared-memory model. For example, when there is only one datum to be transferred from one process to the other, it will be simpler to use send/receive primitives than to use DSM write/read with extra synchronization. ADSMITH allows users to adopt message-passing and/or shared-memory primitives in the same program, according to their needs (see Section 3.6.) Finally, ADSMITH is also flexible in mixing memory models to achieve the maximal performance, as is discussed next.

Memory Model

Strict consistency can relieve the users from the stale data problem [1]. However, the model requires a longer access time and causes shared data to be accessed sequentially. This conservative policy leads to inefficiency in the resultant systems, especially when there is no need to access the newest data. For example, since programmers can be sure that there is no competing access in a critical section, write accesses need not be performed with respect to any other processes immediately. Thus the write operations can be pipelined to overlap their transmission times within the critical section, or they can be delayed to the exit point and then packed into one message to reduce the message counts.

Relaxed consistency policies take advantages of chances similar to those described above to improve the system performance. Future discussions of memory consistencies can be found in Sections 3.3 and 3.5. ADSMITH supports both strict and relaxed consistency models. Users can select to use only the strict consistency model for all memory accesses if they do not want to consider the consistency problem. Alternatively, they can insert primitives for relaxed consistencies at appropriate locations in their programs for improved performance (see Section 3.5.)

Write Policy and Coherence Protocol

Two general write policies for memory hierarchy are write-through and write-back. Write-through will write the data into both the cache and the shared memory, while write-back will delay the write to memory until certain conditions are met, such as when the data block is to be replaced in the cache.

Since our current implementation uses pipeline write for relaxed consistency models, write-through is adopted. All written data are updated in cache and local daemon first[2] and then saved into owner daemon of the shared object.

There are two widely used methods for data coherence: write-invalidate and write-update. Write-update will update all the copies of the written data, while write-invalidate will only invalidate the copies. The advantage of write-update is that the copy holder can retrieve the updated data directly from its local cache. However, write-update may cause a large amount of network traffic. Thus, this policy is suitable

[2] SINGLECOPY data will be written to their owners only; MULTICOPY data with NOCACHE will be written to the local and the owner daemons.

for systems with high speed data transmission capabilities or for those with a fast broadcast network, such as a bus.

Since our system is developed upon PVM, the underlying system may include any kind of networks — with Ethernet being the most popular one. In other words the network latency may be long and broadcast might not be supported. Thus, write-invalidate is adopted in ADSMITH. Write-invalidate only requires a small message to invalidate all copies in the system. This in turn shortens the write latency and avoids heavy network traffic.

2.3 Organization of ADSMITH

ADSMITH are completely built on top of PVM, no other system dependent facilities are used. The interface includes a header file and a user-level library. A daemon will be spawned for each host to support run-time shared-data manipulations. Internal manipulations of shared data are totally transparent to users. The basic organization of ADSMITH is shown in Figure 1.

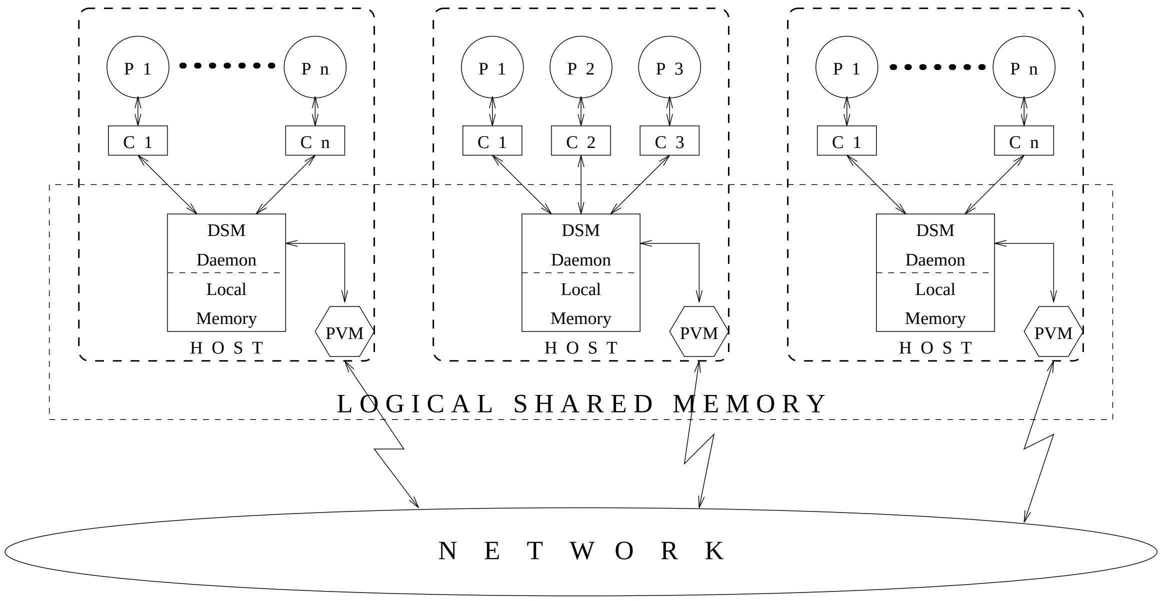

Figure 1: ADSMITH system architecture

The architecture can be divided into two layers: *Logical Shared Memory Layer* (LSML) and *Process Caches Layer* (PCL). PCL is located in the addressing space of each application process. Recently used data will be held in the cache to reduce the access latency. LSML is supported in the daemons. Each daemon will interact with the enrolled processes on its local host to provide shared-memory services. During the service, the daemon also has to interact with daemons in other hosts to manage the shared objects, which are distributed in the memories of the participating hosts. The daemons communicate with local application processes and other daemons through PVM.

Accesses to shared objects will be performed in these two layers in order; that is, shared data will be searched in PCL first, and then in LSML on cache misses. Within LSML, status information of shared objects is kept in the data mapping directories, one in each daemon. From the consistency's point of view, all accesses to LSML can be seen as accessing to a physically shared memory, i.e., strict consistency is applied in this layer. Details of the implementation will be described in Chapter 4. Usage of the system will be introduced in the next chapter first.

Chapter 3

User Interface

ADSMITH presents to the users as a user level library, which is implemented in traditional C language with PVM as its communication platform. In this chapter, we will introduce main functions provided in the system. To demonstrate the usage, an example of cord-pulling game will be used throughout the chapter. Complete listing of the program is in Appendix A (Figures 13 and 14.)

3.1 System Initialization and Termination

Priori to any shared object manipulations, processes must enroll into the ADSMITH system first. Current implementation only supports shared memory functionalities between processes with the same ancestor. Thus, the whole application must start with one root process, which invokes the **adsm_startup()** procedure to start up the whole ADSMITH system (see line 13 in Figure 13.)

Processes in the application can spawn child processes by pvm_spawn(). The parent process should also issue an **adsm_parentinit()** call to initialize the children if the child processes also want to enroll into ADSMITH. Another way to spawn child processes is to use the single **adsm_spawn()** function (see line 23 in Figure 13.) All spawned child processes must also use **adsm_childinit()** or **adsm_enroll()** to enroll into ADSMITH (see line 13 in Figure 14.)

Once a process has finished all its shared-data manipulations and wants to quit from ADSMITH, it must use **adsm_exit()** to inform the system (see line 47 in Figure 13 and line 52 in Figure 14.) ADSMITH will terminate only when all of the participating processes have executed adsm_exit().

3.2 Shared Object Declaration

In a structure-based DSM like ADSMITH, the share space consists of many shared objects, each object can contain many units of logically related data which have the same structure. For example, suppose we want to share the following array among processes in ADSMITH.

```
struct s_unit {
  char a[3];
  int  b[4];
  float c;
} x[100];
```

Then the shared space in ADSMITH will have an object, which contains to the array `x[100]`. The object in turn consists of 100 units of the structure `s_unit`. Each unit of the structure `s_unit` will be referred to as a *structure unit*.

Shared objects must be declared using **adsm_decl()** before they are used (see lines 16 and 17 in Figure 13.) After the shared object is declared, an object handler is returned for later manipulation. One restriction in ADSMITH is that every enrolled process must make the declaration locally in order to get access to a shared object. Declarations for the same shared object must have the same set of parameters (except the LOCAL attribute that will be mentioned later.) Otherwise the behaviors of the shared object will become nondeterminant. If the specified shared object is known to exist already, one can use **adsm_find()** to obtain the handler (see lines 16 and 17 in Figure 14.) The advantage of using adsm_find() is that it only requires the object name, other information of the object can be obtained from the existing one.

adsm_decl() requires four parameters (see lines 16 and 17 in Figure 13.) The first parameter is an object name, which is given by the user to identify the object. The second parameter is a data structure descriptor, used to specify the structure unit. Current implementation of data structure descriptors only supports basic type specifications, including char, short, ushort (unsigned short), int, uint (unsigned int), long, ulong (unsigned long), float, double, etc. The format of data structure descriptors is as follows.

```
" type length (type length)* "
```

For example, "char 3 int 4 float 1" means that the data structure consists of a character arrary of length 3, an integer array of length 4, and a floating point. This is used for both heterogeneity and automatic internal partition described previously.

The third parameter specifies the number of structure units in the shared object. The fourth parameter, called *attributes*, provides users a way of specifying the characteristics of the shared object. The reason of specifying this parameter has been discussed in Section 2.2. In the current implementation, users are responsible for setting these attributes according to the characteristics of individual shared objects. Each object can take the following attributes.

- SINGLECOPY/MULTICOPY indicates whether the system will keep only one instance of the object in the entire system or allow the object to be duplicated on different hosts at the logical shared memory layer.

- CACHE/NOCACHE will enable/disable the caching of the object in the process cache layer (PCL).

- MOSTREAD/MOSTWRITE indicate that most accesses to the object are read/write operations. In ADSMITH implementation, the object is stored in the same way as in MULTICOPY/SINGLECOPY.
- LOCAL will try to make the local daemon as the owner of the object.
- ZERO will make the shared object to be initialized to zero at the time when it is declared.

3.3 Shared Object Access

Current implementation does not support memory attachment as in UNIX shared memory. Thus, all accesses to shared objects must be done through explicit read/write operations. To give users the flexibility in accessing their shared memory, ADSMITH provides two kinds of read/write primitives. One is based on a strict memory consistency model and consists of two primitives, **adsm_scread()** and **adsm_scwrite()** (see line 20 in Figure 13 and lines 21 and 23 in Figure 14.) The other is based on a relaxed memory consistency model and consists of **adsm_rcread()** and **adsm_rcwrite()** (see line 31 in Figure 13 and lines 33 and 42 in Figure 14.)

These primitives all require four parameters: the shared object handler, the address of a buffer space to hold the retrieved data or the data to be written out, the index of the structure unit to start accessing, and the number of structure units to be accessed. Their main differences are described below:

- adsm_scread() will try to retrieve the most up-to-date values of the specified structure units.
- adsm_scwrite() causes the values to be saved into PCL, local daemon, and owner daemon, and all copies of the object in other daemons to be invalidated. The call will be blocked until the above operations have finished and an acknowledgment message has been returned. See Section 4.5 for more details.

- adsm_rcread() will read from the shared object without checking whether the values are the most up-to-date.
- adsm_rcwrite() will write the values to the shared object. The call returns immediately without waiting for the write operation to complete. Using this call, the process can continue its execution when a write request is being serviced. Note that, although adsm_rcwrite() does not wait for the write operation to complete, subsequent read operations for the same object in the same process are guaranteed to obtain the updated values.

adsm_scread() and adsm_scwrite() are usually used in accessing shared memory following a strict memory consistency model. On the other hand, adsm_rcread() and adsm_rcwrite() will have to be coupled with special operations such as release and acquire to ensure the consistency of the shared memory. Please see Section 3.5 for the usages.

3.4 Synchronization

In shared memory programming, users often need to perform synchronization on shared memory accesses. ADSMITH provides users with several synchronization primitives:

- **adsm_p()** and **adsm_v()** implements traditional semaphores for synchronization (see lines 20, 24, 32, 35, and 43 in Figure 14.) The only parameter of the primitives is a semaphore, which can be any of the existing shared object handlers. In ADSMITH, each shared object is associated with an integer semaphore, which can only be accessed by adsm_p() and adsm_v() primitives. Note that the value of the shared object is different from the semaphore value associated with the object.

- **adsm_lock()** and **adsm_unlock()** are used primarily in mutual exclusion control such as in critical section. The parameter specifies a shared object, which is treated as a *lock tag*. Note that in ADSMITH a shared object can only be used as either a semaphore or a lock tag. The locking sequence must obey the following rule: One cannot unlock the tag before it is locked, and, after the tag is locked, it must be the locking process to do the unlock.

- **adsm_barrier()** is used for barrier synchronization (see lines 26 and 38 in Figure 13 and lines 27 and 49 in Figure 14.) It is the same as pvm_barrier() except that it must be used when a relaxed memory consistency model is used in shared memory access.

3.5 Memory Consistency Models

As mentioned above, ADSMITH supports two kinds of memory consistency models. Using the strict consistency model, users do not have to worry about memory consistency — they can always get the most up-to-date values of shared objects. The drawback is that users cannot take advantages of application and architecture characteristics and allow inconsistency in the shared memory so as to improve the system performance.

In ADSMITH, the primitives adsm_scread() and adsm_scwrite() are used to support the strict memory consistency model. In addition, when adsm_scwrite() is used, the following approaches can also be used to satisfy such a model no matter what kind of read operation is used.

1. Specify the shared object as SINGLECOPY. In this case the shared object will not be duplicated. All accesses to that shared object will be sequentially performed on its owner. In this way a strict consistency is obtained implicitly. This approach is suitable for shared objects with frequent write accesses.

2. Specify the shared object as NOCACHE. In this case the object will not be cached in PCL. Since a strict consistency is maintained in logical shared memory (LSML) (see Section 4.7), disabling caching will cause shared data to be accessed through LSML. As a result, a strict consistency is achieved. This approach allows shared objects to be duplicated in different hosts so as to reduce the network communication overhead.

The relaxed memory consistency model in ADSMITH is provided for knowledgable users who wish to access shared memory more efficiently by controlling memory consistency explicitly. Several relaxed memory models have been proposed, such as the weak consistency model (WC), the release consistency model (RC), and their variations [1, 3]. Current implementation of ADSMITH supports a variant of the RC model. Shared memory accesses are classified as ordinary accesses (e.g., read and write) and special accesses (e.g., acquire and release.) In fact, the relaxed memory access of the current implementation is slightly stricter than the actual RC model. It is between WC and RC now. But we recommend the users to take it as a RC model when writing their programs; thus, the programs can execute correctly on the current and the future versions of the system.

Current ADSMITH does not support explicit acquire and release primitives. Instead, these operations are embedded inside the synchronization primitives. The acquire operation is executed at the end of adsm_p(), adsm_lock(), and adsm_barrier() calls, while the release operation is executed at the beginning of adsm_v(), adsm_unlock(), and adsm_barrier() calls. It follows that when using ADSMITH, users must use the P or lock primitives to obtain the most up-to-date values of shared objects. Similarly, users can use the V or unlock operations to inform other processes that a new value for a shared object has been generated.

3.6 Accommodating Shared Memory with Message Passing

To give users more flexibility in developing their programs, all PVM primitives can be invoked together with any ADSMITH calls. In this way, programmers can mix both message passing and shared memory programming together to take the advantages of both paradigms. However, there are two restrictions:

1. The PVM message tag values from 0 to 99 are preserved by ADSMITH for internal communication. Thus, users cannot use any of these values as the message tag when they call PVM message passing routines.

2. Users must keep in mind that when adsm_rcwrite() and adsm_rcread() are used, a relaxed memory consistency model is being followed. In this case, they cannot use any other methods to do synchronization except those synchronization primitives provided by ADSMITH.

Since ADSMITH is a structure-based DSM, additional functions are provided by the system to pack structured data according to the specified data structure descriptors. Two of the major functions are **adsm_dspk()** and **adsm_dsupk()**. They require three parameters, including the data structure descriptor string, a pointer to the data to be packed/unpacked, and the number of structure units to pack/unpack. Other functions of ADSMITH can be found in Appendix B.

Chapter 4

Implementation of ADSMITH

ADSMITH is wholly built on top of PVM. Process management, synchronization, and shared-object manipulations are done through the underlying PVM system. In this chapter, we will describe how ADSMITH is implemented.

4.1 Message Ordering

Message ordering is important in our implementation. For example, to ensure that a message M from site A has arrived at site B, site A can send a message $M_{arrived}$ to site B (see Figure 2.) If messages from the same source are delivered in the same order as they are sent, then the receiving of $M_{arrived}$ by site B in the above case will imply that M has arrived. PVM guarantees in-order message delivery. This property is used widely in ADSMITH, such as in declaration, acquire operations, etc.

4.2 Initialization

To start up ADSMITH, adsm_startup() should be invoked first. This will initiate one daemon on each host in the PVM configuration. One of the initiated daemon will serve as the master daemon, which is responsible for maintaining data allocation and process status information.

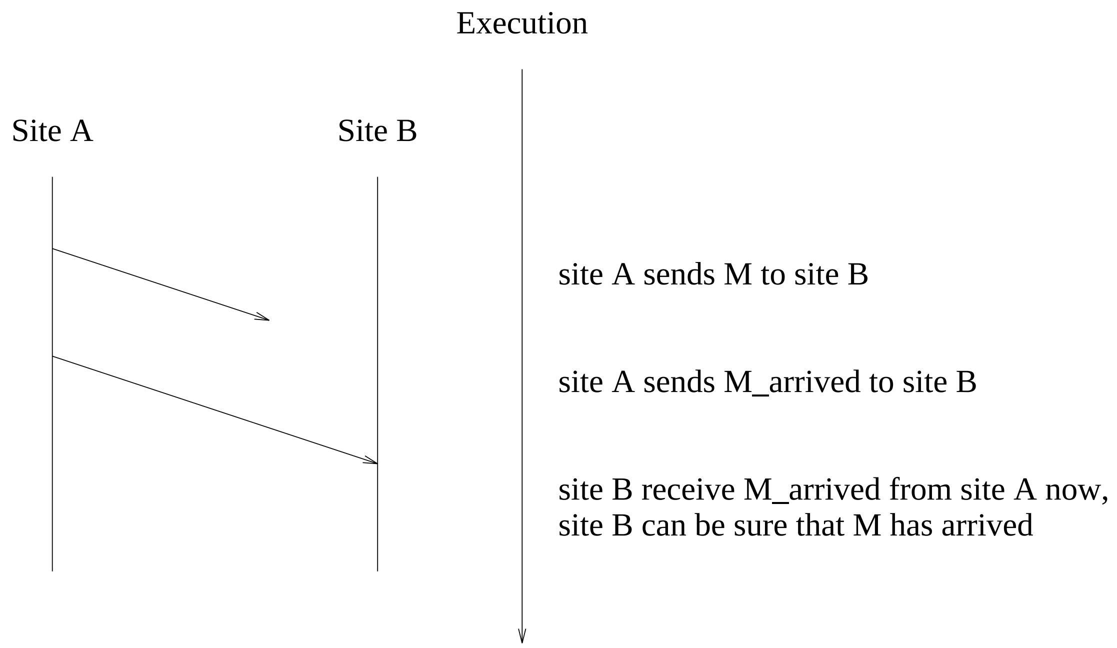

Figure 2: An example of message ordering

Applications using ADSMITH must start from one root process. All other processes in the application will be descended of this root process. A process can fork a child process by calling the pvm_spawn() primitive. After the child is spawned, the parent should call adsm_parentinit() to send initialization information to the child. On the other hand, the spawned child must call adsm_childinit() or adsm_enroll() to receive such information and do some initialization work.

4.3 Declaration of Shared Objects

The declaring process P of a shared object will send a request to its local daemon L. When L receives the request, it will check to see if the object has already been declared. If so, it will send an acknowledgment to P immediately[1]; otherwise, the request will be forwarded to the master daemon M.

To prevent re-declaration, M will again check to see if the object has already been

[1]All send and broadcast operations mentioned in this thesis are nonblocking.

declared. If the object has not been declared, M will select an owner for the object and broadcast the new object's declaration information to all other daemons, which then recorded the information in their shared object directories. On the other hand, if the object is checked to be declared already, then the broadcast message for notifying the declaration must be on its way to L. In this case M will send a DECLARED message to L[2]. In either case, L will inform the local process P after the notification message has arrived.

By default, objects will be circularly allocated to daemons as their data owners. In the future, we will modify the allocation strategy to base on more flexible criteria, such as object size or free memory status of the hosts. Programmer can also specify that the object be allocated in the local daemon if they know that the shared object will be referenced by the local process most of the time. If several processes want to declare a shared object as local simultaneously, the object will be allocated to the local daemon of the process which reports the declaration to the master daemon first.

The data structure descriptor specified in adsm_decl() will be parsed and translated to an internal form to save memory space and for faster manipulation. The internal form will then be used to interpret the raw shared data and transfer the data between heterogeneous systems with necessary data conversions. Note that it is the programmers' responsibility to specify the data structure correctly, i.e., to match the corresponding data structure declared in the C program. A handle is returned from adsm_decl() and is used to reference the shared object in the following accesses.

4.4 Read Operation

For each read access with the target object specified as MULTICOPY with CACHE, the data will be first searched in the process cache layer (PCL), which is in the addressing space of the application process P. Note that the data may only include

[2] Note that this uses the property of in-order message delivery mentioned above.

part of the shared object, i.e., a few consecutive structure units, according to the specification in the read/write primitives (see Section 3.3.) If the object does not exist in the local cache, a request for a copy of the whole object will be sent to the logic shared memory layer (LSML). When the copy is received from LSML, it will be cached, and the specified part of data will be returned to P. On the other hand, if the object is specified as MULTICOPY with NOCACHE, then the data will be directly requested from LSML without searching in PCL.

In LSML, when the local daemon L of the requesting process receive a read request, it will record the request information first for later data maintainance. If the object exists on L, it will be sent to P through its PCL. Otherwise, another request will be sent to the object's owner daemon O through the network. O will also record the request information, and then send one copy of the whole object to L, which in turn caches and transfers the object to P.

If the object is specified as SINGLECOPY (which also implies NOCACHE), the data will be directly requested from O. O will record the request information and send the specified part of data in the object back to P.

As described in Section 3.3, ADSMITH provides two read functions: adsm_rcread() and adsm_scread(). The former will immediately read the shared object no matter if it contains the up-to-date data or not. Thus it can be used in a more relaxed memory consistency model. The function adsm_scread() will fetch the newest data at the time, but it is slower. The transition diagram of the read operation is shown in Figure 3.

4.5 Write Operation

The write policy and coherence protocol used in ADSMITH are write-through and write-invalidate. When a write operation is being performed, the data will be first written into the process's cache and then sent to the logical shared memory layer

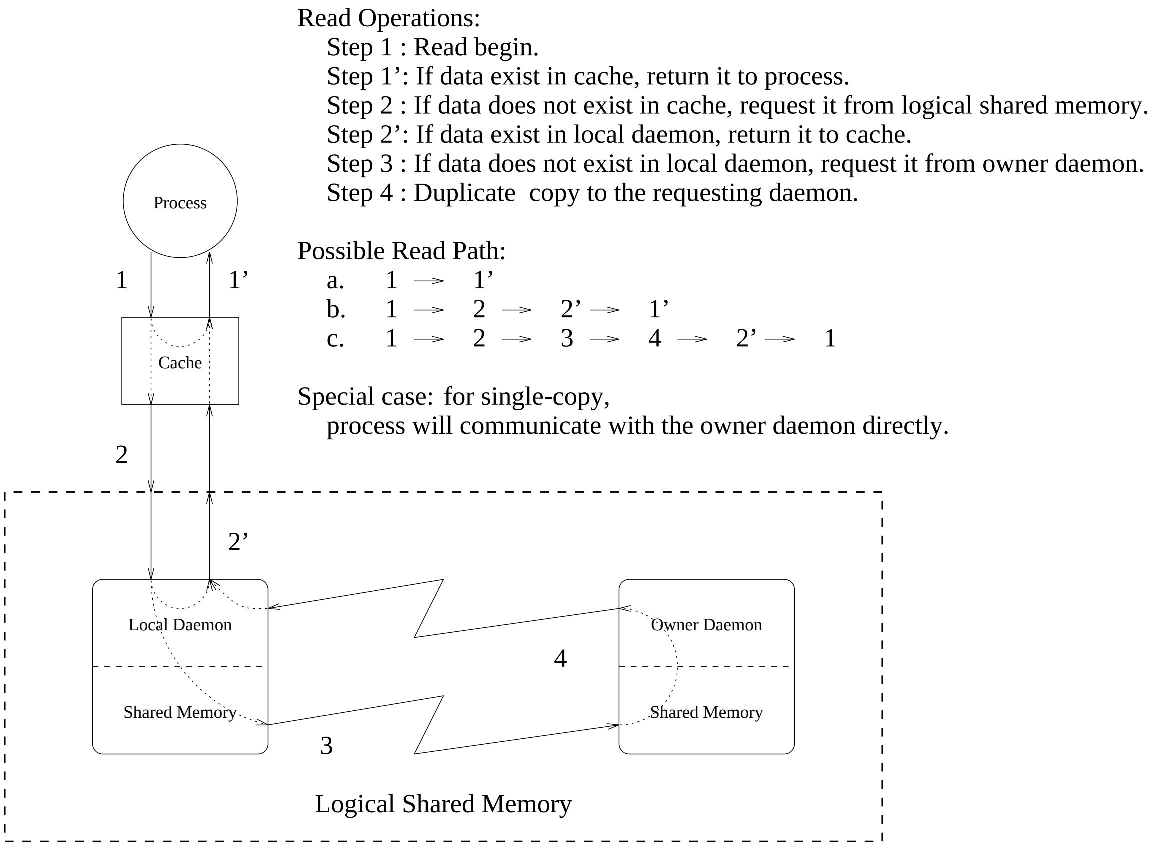

Figure 3: The transition diagram of the read operation

(LSML). Note again that the data may only include part of the shared object.

In LSML, the local daemon will receive the updated data first. It will then save the data and forward it to the owner daemon O. O will update the object and send invalidate messages to all other daemons which have a copy of the object. The owner daemon will wait for all acknowledgments from these daemons. After that, it sends an acknowledgment back to the writing process to inform the completion of the write operation. Note that when O is waiting for the acknowledgments from all other daemons with a copy of the object, it will not be blocked. Other events or requests still can be processed by O, except the read/write requests for the same object currently being written. Such read/write requests will be queued and then be processed when the current write for the object is finished.

A daemon which receives an invalidate message from the owner daemon will first send an invalidate message using non-blocking sends to all local processes that have the specified copy in their caches. The daemon then sends an acknowledgment back to the owner daemon. The information about which daemons or processes to be invalidated is obtained from the request information recorded by daemons in previous read operations. The transition diagram of the write operation is shown in Figure 4.

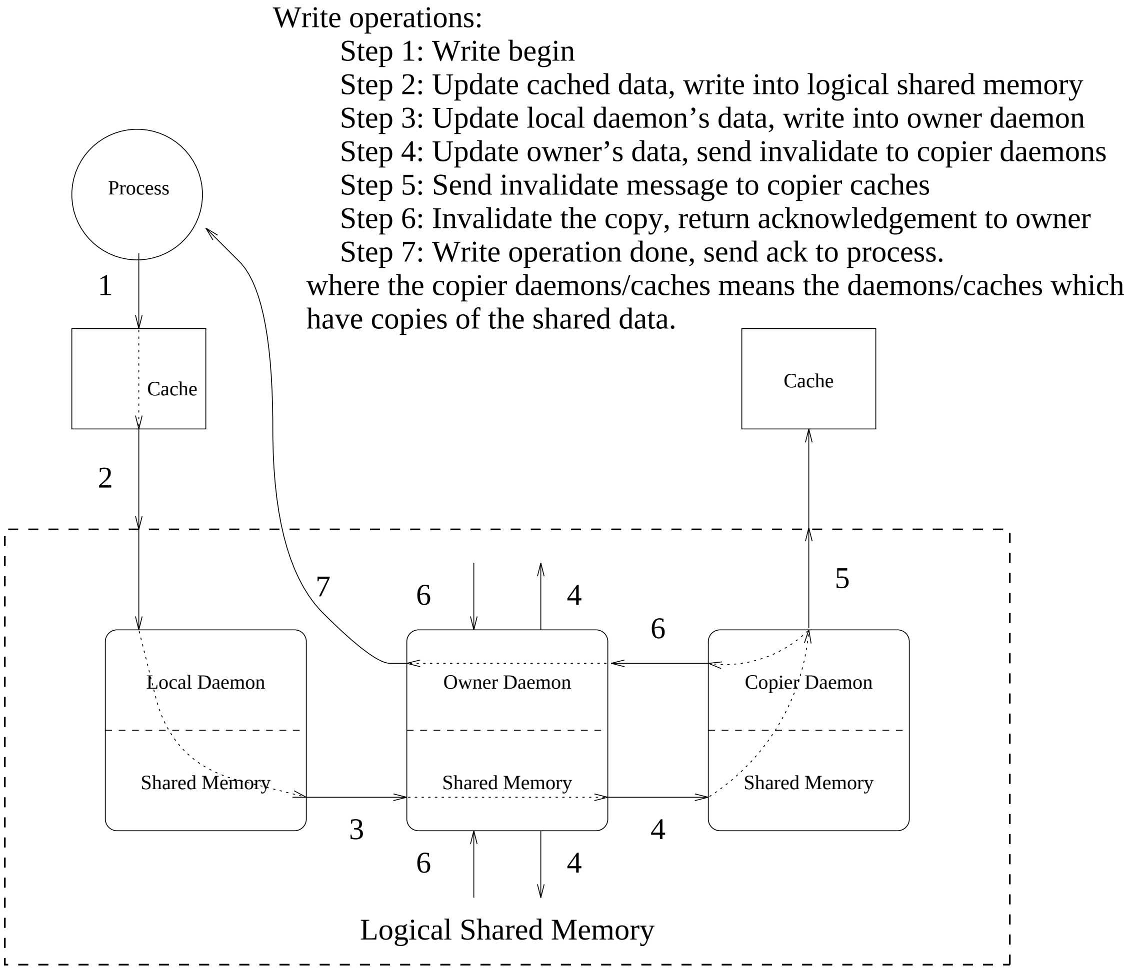

Figure 4: The transition diagram of the write operation

Note that when a write is "finished", copies of the written object in other daemons will have been invalidated, but copies in caches of the application processes may not have been invalidated yet. This will influence our implementation for relaxed memory consistency model in Section 4.7.

ADSMITH also provides two write functions: adsm_rcwrite() and adsm_scwrite(). The former will make consecutive writes pipelined till a release operation is encountered (see Figure 5.) Then write acknowledgments are collected. On the other hand, adsm_scwrite() will block the process on the write operation until it is finished.

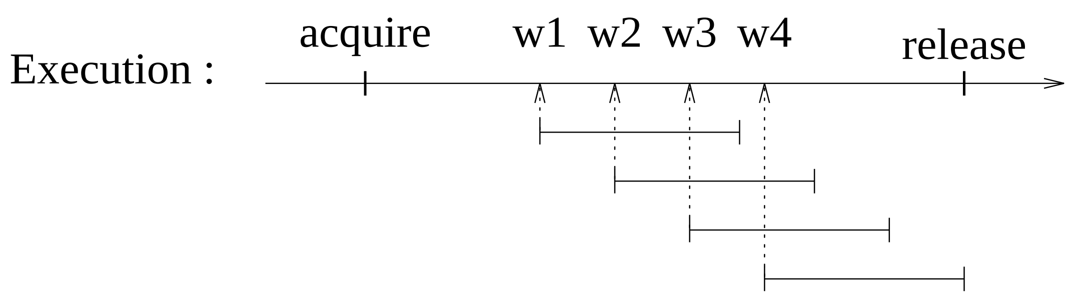

Figure 5: Timing diagram of a pipeline write

4.6 Synchronization

P/V operations are the first kind of synchronization functions provided by ADSIMTH. Each pair of P/V operations works on a semaphore, which can be any shared object (see section 3.4.) The semaphore is guarded by the shared object's owner daemon, which acts as a coordinator. When a process invokes the P operation, a p-request message will be sent to the owner. If the semaphore value is greater than zero, the owner will immediately send a grant message to the requesting process and decrement the semaphore value by one; otherwise, the p-request will be recorded in the p-queue and the requesting process will be blocked.

On receiving a v-request message sent by a V operation, the owner will check the corresponding p-queue. If there exist p-requests, the owner will remove one from the queue and send a grant message to wake up the corresponding blocked process; otherwise, the semaphore will be incremented by one. Since each daemon serves one request at a time — not matter what kind of requests — mutual exclusion in P/V operations is thus guaranteed. The algorithms are outlined in Figure 6:

```
S is initialized to 1;

P(S):                       V(S):

  if (S > 0)                  if (p-queue == NULL)
    S--;                        S++;
  else                        else
    block in p-queue;             wake up a process in p-queue.
```

Figure 6: The algorithms for P/V operations

The second kind of synchronization functions provided in ADSMITH is lock/unlock. Lock/unlock are similar to P/V operations except that the invoking sequence must obey the rule described in section 3.4. A lock request from adsm_lock() will be recorded in the shared object's lock-queue[3] by the owner. A grant message will be immediately sent back to the requesting process when the lock-queue is empty. When the owner receives an unlock request, it will first check to see if the requesting process is the same as the one issued the lock (by checking the information recorded at the head of the lock-queue.) If so, the record will be removed from the lock-queue and the next requester will be granted the lock.

The third kind of synchronization functions is barrier synchronization. It is the same as the barrier for PVM process group, except that adsm_barrier() will be used within ADSMITH's relaxed consistency memory accesses.

4.7 Relaxed Memory Consistency

ADSMITH supports relaxed memory consistency models at the process cache layer (PCL). The implementation includes two major components: ordinary accesses and special accesses. Ordinary accesses include read/write accesses, in which adsm_rcread() and adsm_rcwrite() are mainly used. The implementations for ordinary accesses have

[3]Lock-queue and p-queue are the same queue.

been described in previous sections. Special accesses include acquire and release to support relaxed consistency models. In current implementation release and acquire are performed within synchronization operations. Thus they are executed sequentially because of the mutual exclusive property of these operations, which make ADSMITH sequentially consistent between special accesses.

Before a release operation can start, one must ensure that all previous write accesses have finished [1]. To do so, a process must record all its adsm_rcwrite() activities. When a release command is encountered, the process will then wait for all its previous writes to finish. Note that the term "finished" means that the acknowledgment message for write from the owner of the written object has been received by the process. This implies that all invalidations for the write have been performed on the logical shared memory layer. However, invalidations to PCL might not have completed yet (see Section 4.5.) The latter will need help from the acquire operation discussed next.

For acquire access, the process which invokes the operation will try to gather the most up-to-date data which are released by a previous release operation (no matter which process performed the release.) Only when the acquire operation is finished can subsequent ordinary accesses be issued [1]. In ADSMITH the acquire operation is done by first checking that all invalidate messages for previous release operations have arrived. Any subsequent ordinary accesses will check the invalidation information and, if the accessed objects are invalidated, will be forced to go to the owners of the objects to fetch the most up-to-date values. The relation between and required operations in release and acquire operations are shown in Figure 7.

To make sure that all invalidations for out-of-date data have arrived at the local cache, the process which invokes the acquire operation will send a checkpoint request to the local daemon L. Since invalidate messages have already been sent from L to the process at this time, which is guaranteed by the previous release, the receiving of the checkpoint reply message from L will imply that all previous invalidate messages

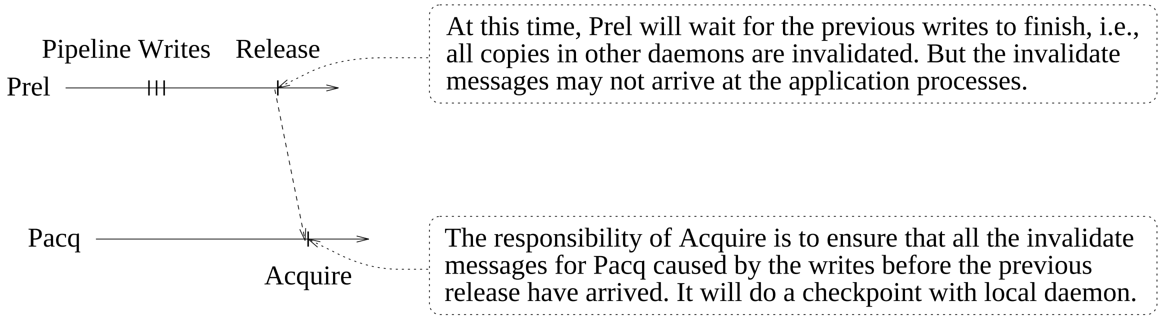

Figure 7: Relation between and operations in release and acquire

have arrived at the process. This again uses the message ordering property described previously in Section 4.1. Furthermore, cache invalidation in ADSMITH does not occur immediately after an invalidate message arrives. Instead, ADSMITH delays the invalidation processing until the moment when the shared object is referenced, e.g., declarations, reads, writes, and synchronization, etc.

To clarify the points, consider a critical section problem as an example. We will demonstrate how two processes using the P/V operations as synchronization primitives cooperate to satisfy the release consistency model. The code is listed in Figure 8.

```
             Pi                             Pj
             .                              .
          Pi(S)              --------> Pj(S)
     [critical section]    .       [critical section]
          Vi(S) ---------           Vj(S)
             .                              .
             .                              .
```

Figure 8: An example code of a critical section problem

Suppose process P_i enters the critical section first, i.e., P_i obtains $P_i(S)$ earlier. Thus process P_j is blocked on $P_j(S)$. When P_i is going to exit the critical section by issuing $V_i(S)$, it will wait until all the write accesses between $P_i(S)$ and $V_i(S)$

finished — a release operation. When P_j enters the critical section, it will first make a checkpoint verification with its local daemon L_j to make sure that all the invalidate messages from the write accesses between $P_i(S)$ and $V_i(S)$ have arrived. The transition diagram of this example is shown in Figure 9.

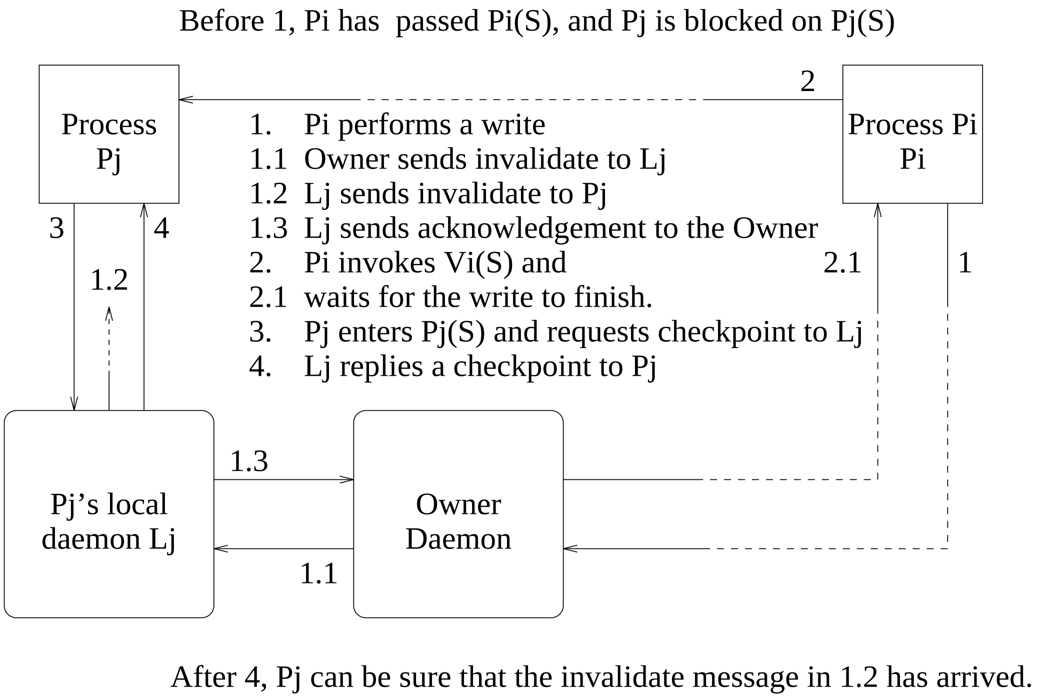

Figure 9: Transition diagram for the critical section example

Since all the invalidate messages for the cache of P_j have been sent out from L_j before $V_i(S)$ is completed, the checkpoint verification will guarantee that these messages have arrived at P_j and will be performed on the cache before any of the shared objects are referenced. This thus accomplish the acquire operation.

4.8 Cache Implementation

Process cache layer (PCL) is located in the addressing space of each application process. It is initialized when the process enrolls into the system. The cache line number and the degree of set associativity can be defined by the programmer at the system startup time. These parameters are then fixed during the execution. On the other hand, the cache size is not fixed. It depends on the total number of the shared objects that are cached. Current implementation caches a whole shared object in a cache line. This may be modified to reduce the cache space in the future.

Each shared object will have a global identifier after it is declared. When a shared datum is read by a process, the whole shared object will be copied from the logical shared memory layer and cached in PCL (unless the object is declared as NOCACHE or SINGLECOPY.) The global identifier of the shared object will be used as the tag to determine the cache set number (by id mod set_number.) Then the object will be put in the first available cache line in the set.

Shared objects will be invalidated when the process receives an invalidate message or when they are replaced. In the latter case, a cache replacement algorithm should be used. ADSMITH now supports only the simple FIFO algorithm. It may be modified to use LRU in the future. The effects of using cache to the system performance will be discussed in Section 5.1.

4.9 Data Conversion

The format of data structure descriptors has been introduced in Section 3.2. When a shared object is declared, its data structure descriptor will be parsed and stored using an internal form, which records the types, lengths, and the offsets of each element in the data structure. Several problems may occur when DSMs are to be supported on a heterogeneous system [12]. In our implementation, shared data will be interpreted and disassembled into its basic data type segments according to the data structure descriptor and the architecture that the system is currently executing. These data segments will then be converted into XDR if necessary and transmitted by the PVM pack and send primitives. When the data segments are received at the receiving process, they will be unpacked by the PVM unpack primitives if necessary and assembled to form a structured data. Thus, the key to heterogeneity in ADSMITH is to interpret the data allocated to a particular architecture. The major problem is data alignment in a data structure. For example, on SUN4 machines, the structure of “int 1 char 1 float 1” will be arranged as in Figure 10.

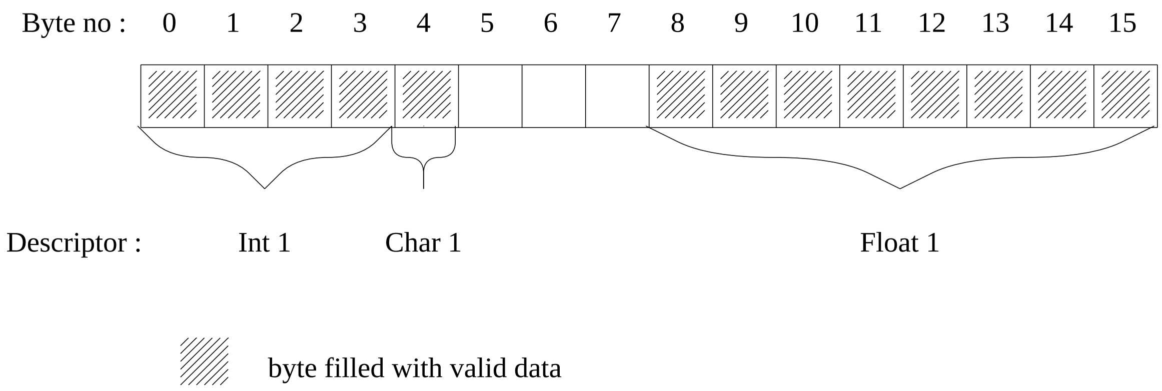

Figure 10: An example of alignment in the structure “int 1 char 1 float 1” on SUN4

Because the alignment number on SUN4 is 4 bytes, the starting address of the floating point should be at byte 8 but not byte 5. Thus, the total number of bytes needed in this data structure is 16 bytes. Different architectures may have different alignment numbers, the system must know the number for a particular architecture to interpret the data correctly. This problem can be solved by defining the value of ALIGN in the source code before the system is compiled. Since we can indicate the beginning point for each element in the data structure, problems such as byte ordering and data conversion (by XDR) can be solved by the underlying PVM system. Other problems mentioned in [12], such as that caused by data across two pages, will not happen in our implementation, because data will be allocated and transferred according to their structures instead of a fix-sized page. As a result, there is no “cross-page” problem.

Chapter 5

Performance Evaluation

Several experiments were conducted to evaluate the performance of ADSMITH. Our evaluations are confined by the following two restrictions. First, since current implementation does not support automatic data partition, we distributed the shared objects manually. Second, we performed the experiments on a network of workstations. Although we had tried our best to dedicate the workstations to our experiments, the system load was still affected by other unrelated processes in the systems. Therefore, we can only take the average from different runs of the experimental programs.

5.1 Comparison of Shared Object Specifications

A shared object can be specified as SINGLECOPY, MULTICOPY with NOCACHE, or MULTICOPY with CACHE. In this section, we will study the effects of different specifications. Matrix multiplication was used in our experiment. Two matrices A and B are partitioned by row and column respectively (see Figure 11.) Each row or column will form a shared object.

To compute the target matrix C, the distributed processes will compete for the task of computing the next un-computed row of C. The algorithm is based on a dynamic load balance method, called the *Chunk Self Scheduling* [25]. To study the effects of different specifications, we deliberately set the structure unit to be each

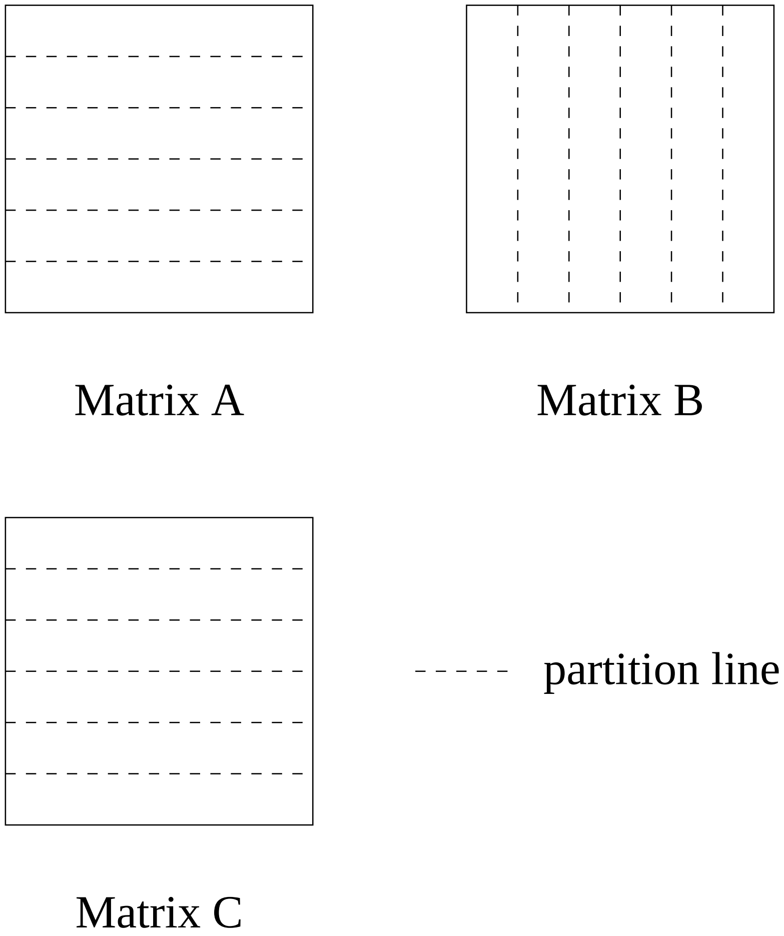

Figure 11: Data partition in the matrix multiplication example

individual array element instead of the whole row or column. In other words, to obtain $C_{i,j}$, each $A_{i,k}$ and $B_{k,j}$ will be obtained by calling read primitives separately.

For SINGLECOPY, all elements referenced are directly requested from the owner daemon. For MULTICOPY with NOCACHE, the referenced elements will be requested from the logical shared memory layer. For MULTICOPY with CACHE, the whole row or column will be requested and cached in the process cache layer first before any of the elements in the row or column can be referenced.

In this experiment, we collected the total execution time of matrix multiplication using different sizes of square matrices on eight DEC Alpha workstations. The results are shown in Figure 12.

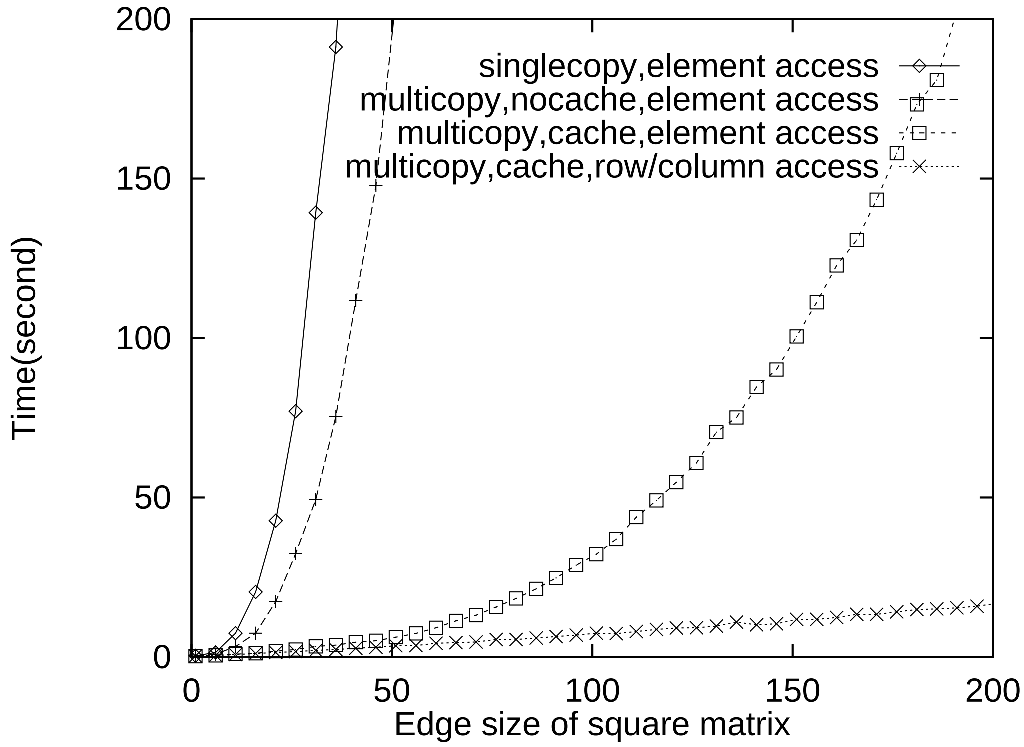

Figure 12: Execution time of matrix multiplication with different specifications

The upper three curves shown in this figure are for SINGLECOPY, MULTICOPY with CACHE, MULTICOPY with NOCACHE respectively. For comparison purpose, we also show the execution time of matrix multiplication using the whole row or column as the structure unit. As expected, this scheme is the fastest, because it uses larger messages to reduce the communication overhead.

From the plot, we can find that using cache has greatly improved the performance. Note that the problem size is proportional to the square of the value on x-axis. For SINGLECOPY, since elements were all directly requested from the object owners, the execution time is the longest due to network communication. For MULTICOPY with NOCACHE, most of the data references will be done in local host through interprocess communication. Thus the difference between SINGLECOPY and MULTICOPY with NOCACHE is largely due to the difference in network communications and interprocess communications.

Number of Hosts	Pipelined Write		Sequential Write		Single Copy
	Before V	After V	Before V	After V	Write
1	1.154	10.923	7.469	7.475	5.871
2	1.849	18.824	14.963	14.969	7.684
3	2.252	26.790	17.182	17.191	7.955
4	1.577	50.381	18.906	18.916	8.310
5	2.306	61.984	18.973	18.981	7.769
6	1.561	96.213	19.106	19.114	8.708
7	1.489	98.952	21.351	21.439	10.396
8	1.709	100.646	20.904	20.912	9.346

Table 2: Elapsed times for pipelined and sequential writes

5.2 Comparison of Pipelined and Sequential Writes

To compare pipelined write (using adsm_rcwrite()) and sequential write (using adsm_ scwrite()), we conducted the following experiment. Process P_w declared 1000 objects as MULTICOPY with CACHE on one to eight DEC Alpha workstations. For each host in the system, a process P_i is spawned to read all the objects. After the read operations, P_w begins to write. Any write from P_w will invalidate the copies on the daemons and P_is' caches. There is a P operation performed before all writes and a V operation after. Two time segments were recorded. The first segment is the interval from the first write (i.e., after the P operation) to the end of the last write (i.e., before the V operation.) The second segment starts from the first write until the time when all writes are guaranteed to finish (i.e., after the V operation.) For comparison, the time for sequentially writing 1000 SINGLECOPY objects was also measured. The experimental results are listed in Table 2.

For SINGLECOPY objects, the writing times are shorter, because there is no need for invalidating procedures. The large gaps between the timings for one and two hosts are caused by the network communications which are necessary for configurations with more than one host. Since all writes are pipelined before the V operations in the pipelined write, the first time segment is much shorter than that of the sequential write. This time segment is also independent of the number of hosts, because the process only has to send out the write requests without having to wait for writes to finish. On the contrary, the time spent in waiting for all writes to finish (i.e., the second time segment minus the first time segment) in pipelined write is longer. This seems to be caused by PVM implementation. As processes send write messages using the pipelined write method, many messages may enter in the PVM system in a short period. In this case, message retrieval from the PVM system becomes slow, and it will take longer for those writes to finish. This problem may be reduced when the pipelined write method used in relaxed memory consistency model is replaced by the delayed write method. That is, the write operations (performed by adsm_rcwrite()) will be delayed to the release point and then packed to send the write requests.

5.3 Distributed Ray Tracing

We have implemented a parallel program for ray tracing using both message passing and distributed shared memory approaches. Ray tracing is one of the most popular methods for rendering high quality computer graphics images [23]. One ray will be shut for each pixel from a view point. When a ray hits an object, the intersection point between ray and the object surface will be calculated and the color and intensity at that point will be evaluated from three basic lights (including ambient light, diffuse light, and specular light) plus the lights of reflection and transmission at that point [22]. Evaluations for the reflection and transmission rays are recursive calls of the ray tracing procedure. Because most of the execution time is spent in ray-surface intersection calculation (which occupies almost 90sequential ray tracing is very slow [24]. Since rays are independent of each other, we can easily parallelize the computation by distributing the whole job to several computers.

In this experiment, we used 59 objects in a 3D space (a room) for rendering.

Number of Processor	Execution Time (second)		Machine involved
	PVM version	ADSMITH version	
1	452	475	DA*1
2	231	237	DA*2
3	163	159	DA*3
4	130	129	DA*4
5	103	102	DA*5
6	86	89	DA*6
7	84	84	DA*7
8	79	75	DA*8
9	77	74	DA*8 and S10*1
12	68	64	DA*8,S10*1, and S2*3
DA=DEC Alpha, S10=SUN Sparc-10, S2=SUN Sparc-2			

Table 3: Execution time of Ray Tracing for the PVM and ADSMITH versions

One to eight DEC-Alpha and some SUN4 workstations were used. The chunk self-scheduling method was again used to dynamically balance the load. The target image had a size of 300×300 pixels and was divided into 30 chunks, each chunk consists of 10 rows. A process was spawned for each host as a ray tracer. For the message passing version, when a process has completed a chunk, it will send a request for the next un-computed chunk to the master process. The master then sends the chunk specification to the requesting process. For the version using ADSMITH, when a process has completed a chunk, it will retrieve a free chunk from the shared memory directly. The results are listed in Table 3.

From the experiment results, we can find that the execution time of these two implementations are close. Sometimes the execution time for ADSMITH version is shorter than that of PVM version. This is because we can specify the shared object with suitable attributes. For example, the partition table is specified as MULTICOPY with CACHE, since there is no write to this table. The job-index used for self-scheduling is set as SINGLECOPY, since it is often written. These flexible-settings overcame the DSM overhead. Thus, from this application, we can conclude that using

ADSMITH with suitable attributes (or other flexibilities supported in the system) will not compromise the performance. In addition, the programs are easier to understand and to write. From the lowest two rows of the table, we can find that the speedup is not so significant. This is because when the program ran on several DEC and SUN workstations, the execution time is dominated by the SUN workstations, which have worse performance than that of DEC Alpha workstations. In fact, this problem can be partially solved by adjusting the chunk size used in the chunk self-scheduling.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Message passing is the most popular programming paradigm in distributed systems. However programming with message passing is tedious and error-prone, especially when the communication pattern becomes complicated. Instead, programs written using shared variables are easier to develop. Thus DSM was proposed to provide shared memory capabilities in distributed-memory environments.

In this thesis we introduced one approach to DSM and described how such a system, called ADSMITH, is built. We implemented ADSMITH as a structure-based DSM at the library layer on top of PVM. PVM allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. Since PVM has been ported to many systems, our ADSMITH is thus highly portable and can support heterogeneous systems easily. Major characteristics of ADSMITH are as follows.

1. It is wholly built on top of PVM.

2. It is implemented as a user-level library in traditional C language.

3. It supports heterogeneous environments.

4. It has a two-level memory hierarch: process cache layer and logical shared memory layer.
5. Structure-based accesses are used.
6. Information about shared objects is kept in all daemons, and ownership of shared objects is fixed.
7. Shared data can be allocated to hosts (i.e., as their owners) by the system or through user specifications.
8. Data can be duplicated/cached in different parts of the system if necessary.
9. Write-through and write-invalidate are used.
10. Two kinds of memory consistency models are supports: strict consistency and relaxed consistency.

From a programmer's point of view, programs written in ADSMITH are more readable than those written in PVM. The execution time, on the other hand, does not suffer. Experimental results have also shown that the use of caches in processes' addressing space will greatly reduce the access time of the system. In the release consistency memory model, pipelined write takes longer than sequential write in our evaluation. This is due to the implementation problem of the underlying PVM system.

The main problem with our system is the large overhead in the system software. However, ADSMITH does have the potential to be a powerful tool for parallel program development. To make it more practical and efficient, the system can be improved along the directions outlined next.

6.2 Future Work

We list below several possible directions for improving ADSMITH, though some of them may conflict with our design goals and strategies.

1. System performance: We can adopt faster algorithms for data computation or management, reduce the data space used in the system, and minimize the overhead involved in operations.

2. Prefetch: Current implementation caches all the data of a shared object in the local daemon, and perhaps the cache of the application processes, when any part of the shared object is read. We hope to prefetch related data as early as possible so as to hide the communication latency. For example, suppose that objects A, B, and C are related. They may be distributed to different hosts. As any of these objects, say B, is read by a process P, A and C will be fetched to P before they are really needed in P. In doing prefetch, we can further reduce the message count by packing A and C into one message, if they are allocated to the same host. Several methods can be used to define the "related" relationship between shared objects.

    - Simple method: Let objects that are declared consecutively be related. Of cause, this simple scheme may cause unnecessary prefetchs.
    - User-defined method: Since users have the best knowledge of what data are related, we can provide a primitive for users to group related objects. This method is easy but requires users to specify the group explicitly.
    - Automatic method: System can examine data access status and find related groups of data. So far, it is not clear how this can be done efficiently.

3. Automatic data partition: Current implementation allocates the whole shared object in a daemon. If a large shared object can be partitioned can be and stored different partitions into different daemons, then accesses to the shared object can be done concurrently. Especially, when several processes want to write to different parts of the shared object, these write operations can be done at the same time, i.e., write sharing.

4. Complex data structure descriptor: Current data structure descriptors only allow users to define a structure consisting of basic types. For more complex data structures which include sub-structures, users must specify explicitly the detail of the sub-structures in the descriptor. If the system can support functions such as data structure descriptor macros or aliases (similar to typedef in the C language), specifying data structures will be easier. Alternatively, a preprocessor may be provided to help parsing and translating the (complex) structures defined in the C language to a descriptor string [12].

5. Distributing the shared object directory: We now keep the whole shared object directory on every daemon. This will require a large amount of memory space if there are many shared objects declared. One solution is to keep the whole information in only a few manager daemons. When one directory entry is to be referenced and it does not exist in the local daemon, a request for the entry or a group of entries is sent to the nearest manager daemon. The returned entry will then be stored locally until it is replaced when the pre-allocated memory space is full.

6. Transparent shared-memory access: Current implementation requires users to access the shared objects through system functions, such as adsm_rcread(), adsm_scread(), adsm_rcwrite(), and adsm_scwrite(). Users also have to allocate a small buffer to hold the accessed data. There are several ways to make shared-memory accesses more transparent. The first approach is to use pointers to manipulate the shared objects. For example, shared objects can be updated using an acquire command, which in turn returns a pointer to the object in the local cache for later manipulations. To inform other hosts of the most up-to-date values of the shared object, users have to issue a release command explicitly.

   The second approach is to use the operator overloading capability of object-oriented programming languages. We can redefine the assignment and other operators for shared object accesses so as to perform the read/write operations implicitly. Using OOP, complex shared data structures can be specified and data conversions can be done easily [8]. The third approach is to use preprocessor or compiler techniques to translate all the shared data references to system primitives for shared data accesses.

7. Release consistency model implementation: ADSMITH currently implements a stricter memory consistency model than the release model. Also, it implements the acquire and release operations implicitly within synchronization operations. As a result the benefit gained from such an implementation is limited. To take the full advantages of the release model, we may implement explicit acquire and release primitives in ADSMITH in the future. Moreover, lazy release consistency model may also be implemented to further reduce the message count and the amount of data in messages [2].

8. Thread facility: The major difference between general process and light weight process (thread) is that the latter has the same addressing space shared between threads. Since DSMs have provided a shared memory environment between distributed processes, it can be very easy to provide a thread facility in this system [12].

Bibliography

[1] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessor," In *Proceedings of the 17th Annual International Symposium on Computer Architecture,* pp. 15–26, May 1990.

[2] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," In *Proceedings of the 19th Annual International Symposium on Computer Architecture,* pp. 13–21, May 1992.

[3] S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel, "Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology," In *Proceedings of the 20th Annual International Symposium on Computer Architecture,* pp. 144–155, 1993.

[4] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer,* Vol. 24, No. 8, pp. 52–60, Aug 1991.

[5] A. Mohindra, "Issues in the Design of Distributed Shared Memory Systems," *Ph.D. Thesis,* Geogia Institute of Technology, 1993.

[6] A. Geist, et al., "PVM 3.0 User's Guide and Reference Manual," Oak Ridge National Laboratory, 1993.

[7] V.S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience,* Vol. 2, No. 4, Dec. 1990.

[8] C.L. Hartley and V.S. Sunderam, "Concurrent Programming with Shared Objects in Networked Environments," In *Proceedings of 7th Intl. Parallel Processing Symposium,* pp. 471–478, Los Angeles, April 1993.

[9] M. Arrango, D. Berndt, N. Carriero,D. Gelernter, and D. Gilmore, "Adventures with Network Linda," *Supercomputing Review,* Vol. 3, No. 10, Oct. 1990.

[10] "High Performance Fortran Language Specification," Rice University, Houston Texas, May 1993.

[11] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.W. Tseng, and M.Y. Wu. "Fortran D language specification," Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, Dec. 1990.

[12] S. Zhu, M. Stumm, K. Li, and D. Wortman, "Heterogeneous Distributed Shared Memory," *IEEE Transaction on Parallel and Distributed System,* Vol. 3, No. 5, pp. 540–554, September 1992.

[13] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew, "Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 31–39, 1987.

[14] R. Bisiani and A. Forin, "Multilingual parallel programming of heterogeneous machines," *IEEE Transactions on Computers,* Vol. 37, No. 8, pp. 930–945, August 1988.

[15] G.S. Delp, A.S. Sethi, and D.J. Farber, "An Analysis of Memnet: An experiment in high-speed shared-memory local networking," *Computer Communication Review,* Vol. 18, pp 165–174, Stanford, California, August 1988. ACM SIGCOMM

[16] P. Dasgupta, R. LeBlanc, M. Ahamad, and U. Ramachandran, "The CLOUD distributed operating system," *IEEE Computer,* April 1991.

[17] A. Sane, K. MacGregor, and R.H. Campbell, "Distributed virtual memory consistency protocols: Design and performance," *IEEE Workshop on Experimental Distributed Systems,* pp. 91–96, Oct 1990.

[18] Kendall Square Research, "KSR1 Principles of Operations," 1991.

[19] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-bashed cache coherence protocol for shared memory multiprocessors," *Proceedings. The 17th Annual International Symposium on Computer Architecture,* pp. 148–159, 1990.

[20] J.B. Carter, J.K. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," *The 13th ACM Symposium on Operating Systems Principles,* Oct 1991.

[21] Sun Microsystems, "XDR: external Data Representation Standard," *Internet request for Comments RFC 1057,* June 1988.

[22] T. Whitted, "An Illumination Model for Shaded," *Communications of the ACM,* Vol. 23, No. 6, pp. 343–349, June 1980.

[23] J. Foley, A.V. Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice,* Addison-Wesley, 1990.

[24] R. Kuchkuda, "An Introduction to Ray Tracing," *Theoretical Foundations of Computer Graphics and CAD,* pp. 1039–1060, R. A. Earnshaw, ed., Springer-Verlag, 1988.

[25] J. Liu and V.A. Saletore, "Self-Scheduling on Distributed-Memory Machines," *ACM Suptercomputing,* 814–823, Nov 1993.

Appendix A

An Example of ADSMITH

To illustrate the usage of ADSMITH, a cord-pulling game was written. The source programs are shown in Figures 13 and 14. Two racer processes will be spawned from the monitor process. They simulate two teams, each of which holds one side of the cord and tries to pull the cord to its side. The center position of the cord is initially 0. When the center position is moved right to position 5, the right side wins. When the center position is moved left to position -5, the left side wins. In the duration of the race, whenever a side has pulled a unit, it will take a rest for a random time.

```
#include <stdio.h>
#include "pvm3.h"
#include "adsm.h"

main() {
  int     value; /* buffer of cord center value */
  adsmobj obj_cc, /* cord center, the shared object */
          obj_role; /* for self-decide role */
  int     role,NTHU=0,NCTU=1;
  int     mytid=pvm_mytid(); /* can be removed */

  /* system startup */
  adsm_startup(0,0); /* use default cache line num and set associate */

  /* declare the shared object */
  obj_cc=adsm_decl("obj_cc","int 1",1,ADSM_SINGLECOPY|ADSM_ZERO);
  obj_role=adsm_decl("obj_role","int 1",1,ADSM_DEFAULT|ADSM_ZERO);

  /* initialization */
  adsm_scwrite(obj_role,(char*)&NTHU,0,0);

  /* spawn child process */
  adsm_spawn("racer",NULL,0,"",2,NULL);

  /* race start barrier */
  adsm_barrier("Race_Start",3);

  /* start to race: monitor */
  while (1) {
    usleep(100000);
    adsm_rcread(obj_cc,(char*)&value,0,0); /* read center value */
    printf("current center position = %d\n",value);
    if (value<-5||value>5)
      break;
  }

  /* race end barrier */
  adsm_barrier("Race_End",3);

  /* find out the winner */
  if (value>5)
    printf("NTHU win!\n");
  else
    printf("NCTU win!\n");

  /* exit from ADSMITH and PVM */
  adsm_exit();
  pvm_exit();
}
```

Figure 13: Listing of monitor.c

```c
#include <stdio.h>
#include "pvm3.h"
#include "adsm.h"

main() {
  int      value; /* mainly for buffer of cord center value */
  adsmobj obj_cc, /* cord center, the shared object */
          obj_role; /* for self-decide role */
  int      role,NTHU=0,NCTU=1;
  int      mytid=pvm_mytid(); /* can be removed */

  /* register into ADSMITH */
  adsm_enroll();

  /* find the declared object */
  obj_cc=adsm_find("obj_cc");
  obj_role=adsm_find("obj_role");

  /* decide who am i */
  adsm_p(obj_role);
  adsm_scread(obj_role,(char*)&role,0,0);
  if (role==NTHU)
    adsm_scwrite(obj_role,(char*)&NCTU,0,0);
  adsm_v(obj_role);

  /* barrier synchronization */
  adsm_barrier("Race_Start",3);

  /* start to race: child try to pull to 10  */
  srandom(time(NULL)/7+1);
  while (1) {
    adsm_p(obj_cc); /* enter critical section */
    adsm_rcread(obj_cc,(char*)&value,0,0); /* read center value */
    if (value<-5||value>5) { /* done */
      adsm_v(obj_cc); /* exit critical section */
      break;
    }
    if (role==NTHU)
      value++; /* NTHU pull right */
     else
      value--; /* NCTU pull left */
    adsm_rcwrite(obj_cc,(char*)&value,0,0); /* write value back */
    adsm_v(obj_cc); /* exit critical section */
    /* take a rest : sleep longer if tend to win */
    usleep(random()%100000+100000/(role==NTHU?7-value:abs(-7-value)));
  }

  /* race end barrier */
  adsm_barrier("Race_End",3);

  /* exit from ADSMITH and PVM */
  adsm_exit();
  pvm_exit();
}
```

Figure 14: Listing of racer.c

Appendix B

Primary functions of ADSMITH

B.1 Primary Functions

System Initialization

- **adsm_startup**($<$ *lines* $>$, $<$ *deg* $>$)
  Starts up ADSMITH. (For the root process only.) Sets cache line number and the degree of set associativity. Returns the task tid of the calling process.

- **adsm_parentinit**($<$ *tids* $>$, $<$ *num* $>$)
  Initializes the specified child processes which also wants to enroll into ADSMITH.

- **adsm_spawn**($<$ *exec* $>$, $<$ *argv* $>$, $<$ *flags* $>$, $<$ *where* $>$, $<$ *count* $>$, $<$ *tids* $>$)
  Spawns child processes by pvm_spawn() and then invokes adsm_parentinit() for all child processes. Returns the number of successfully spawned processes.

- **adsm_childinit**($<$ *parentid* $>$) and **adsm_enroll**()
  Enrolls into ADSMITH. Returns the calling process's tid.

Shared Object Declaration

- **adsm_decl**($<$ *name* $>$, $<$ *dscptr* $>$, $<$ *unitnum* $>$, $<$ *attrs* $>$)
  Delares a shared object. Specifies the data structure descriptor, the number of structure units, and the attributes. Returns a object handler.

- **adsm_find**($<$ *name* $>$)
  Finds the shared object. Returns the object handler.

Shared Object Access

- **adsm_rcwrite**($< obj >$, $< buf >$, $< beginunit >$, $< unitnum >$)
  Writes the specified number of structure units into the shared object from a buffer nonblockingly. (For relaxed memory consistency model.)

- **adsm_scwrite**($< obj >$, $< buf >$, $< beginunit >$, $< unitnum >$)
  Writes data blockingly. (For strict memory consistency model.)

- **adsm_rcread**($< obj >$, $< buf >$, $< beginunit >$, $< unitnum >$)
  Reads the specified number of structure units from the shared object into a buffer no matter if it is the most up-to-date. (For relaxed memory consistency model.)

- **adsm_scread**($< obj >$, $< buf >$, $< beginunit >$, $< unitnum >$)
  Reads the most up-to-date data. (For strict memory consistency model.)

Synchronization

- **adsm_p**($< obj >$)
  Performs a P operation with respect to the semaphore (associated within the shared-object.) Embeds an acquire operation after it.

- **adsm_v**($< obj >$)
  Performs a V operation with respect to the semaphore. Embeds a release operation before it.

- **adsm_lock**($< obj >$)
  Performs a lock operation with respect to the lock tag (associated within the shared-object.) Embeds an acquire operation after it.

- **adsm_unlock**($< obj >$)
  Performs a unlock operation with respect to the lock tag. Embeds a release operation after it.

- **adsm_barrier**($< group >$,$< count >$)
  Performs pvm_barrier(). Embeds an acquire and a release operation before and after it respectively.

System Termination

- **adsm_exit**()
  Informs the system the exit of the calling process.

Information Retrieval

- **adsm_dumptable**($< obj >$)
  Dumps table entry and data contains of the shared-object on all directories of each daemons.

- **adsm_debug**($< type >$, $< set >$)
  Sets some debugging flags.

B.2 Miscellaneous Functions

Structure Data Packing/Unpacking for PVM Send/Receive Primitives

- **adsm_ds2ids**($< dscptr >$)
  Transforms the data structure descriptor string into the internal form.

- **adsm_ids2ds**($< idscptr >$)
  Transforms the data structure descriptor internal form to the string form.

- **adsm_idspk**($< idscptr >$, $< data >$, $< unitnum >$)
  Packs the specified number of structure units data into the PVM sending buffer according to the data structure descriptor internal form.

- **adsm_idsupk**($< idscptr >$, $< buf >$, $< unitnum >$)
  Unpacks from the PVM receiving buffer the specified number of structure units data into the buffer according to the data structure descriptor internal form.

- **adsm_dspk**($< dscptr >$, $< data >$, $< unitnum >$)
  Packs the specified number of structure units data into the PVM sending buffer according to the data structure descriptor string.

- **adsm_dsupk**($< dscptr >$, $< buf >$, $< unitnum >$)
  Unpacks from the PVM receiving buffer the specified number of structure units data into the buffer according to the data structure descriptor internal form.

Error Message

- **adsm_seterr**($< onoff >$)
  Turns on/off the automatic error message displaying of the calling process.

- **adsm_perror**($< headmsg >$)
  Forces to print the nearest error message with a message header.

- **adsm_setmaxerr**($<$ *num* $>$)
  Sets maximal number of error messages that can be automatically displayed.

- **adsm_clearerr**()
  Reset error message counter to 0.