Adsmith: An Efficient Object-Based Distributed Shared Memory System on PVM

Wen-Yew Liang
Dept. Computer Science
and Information Engineering
National Taiwan University
Taipei, Taiwan

Chun-Ta King
Dept. Computer Science
National Tsing Hua University
Hsinchu, Taiwan

Feipei Lai
Dept. Electrical Engineering
& Dept. Computer Science
and Information Engineering
National Taiwan University
Taipei, Taiwan

Abstract

In this paper, we describe an object-based distributed shared memory called Adsmith. In an object-based DSM, the shared memory consists of many shared objects, through which the shared memory is accessed. Adsmith is built on top of PVM at the library layer using C++. PVM is used as the communication subsystem because it is a de facto standard and encapsulates many system related details. Several mechanisms are used to improve the performance of Adsmith, such as release memory consistency, load/store-like memory accesses, nonblocking accesses, and atomic operations, etc. Performance results show that even though Adsmith is implemented on top of PVM, programs running on Adsmith can achieve a performance comparable with those running directly on PVM.

1 Introduction

For ease of construction and high scalability, many high performance parallel computers today are built as distributed memory systems. In such systems, message passing is the most general programming paradigm. 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. Shared-memory programming, on the other hand, relieves programmers from managing shared data explicitly. Thus programs can be developed more easily. Combining distributed-memory architecture with shared-memory programming is thus a right choice for parallel computers. Based on this observation, *distributed shared memory* (DSM) was proposed and has attracted much attention [8].

A DSM provides a logically shared memory on top of a network of computers with a physically distributed memory. Previous approaches to DSMs usually partition the shared-memory addressing space into logical fix-sized 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. Such a DSM system is referred to as a *block-based* DSM in this paper. 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 memory system is totally hidden from the users. However, block-based DSMs have the problem of choosing the right size for the blocks. The block size depends not only on the system characteristics but also on the applications. Another problem is the design complexity. Implementation at the hardware and/or operating system layers needs to modify existing systems, which requires a tremendous effort. Also, these implementations are usually system dependent. Thus it is very difficult to implement block-based DSMs on top of heterogeneous systems.

Another approach to DSM is *object-based*, which partitions the shared memory according to logical data structures. Object-based DSMs are most often implemented at the language/compiler or library layers. DSMs implemented at this layer usually require some modifications at the users' end — either to the programming model, language, or style. Thus, transition from sequential computers to such an environment is not so seamless and transparent. Since object-based DSMs are implemented at a rather high layer in a computer system, performance is not as good as that of blocked-based DSMs. However, object-based DSMs do offer some unique and important features.

An implementation at higher layers makes object-based DSMs very flexible and system independent. Programmers have a larger control over how data are distributed according to the characteristics of the application. Also, programmers can determine important parameters, such as the block size, access method, communication mechanism, memory consistency model, etc., easily in object-based DSMs. The DSMs can be directly implemented on top of existing generic communication subsystems, such as TCP/IP, PVM, or DCE. Not only that system development efforts can be reduced dramatically, but also porting to different architectures is straightforward. Furthermore, the resulting system can be easily modified and improved when newer techniques are available.

Object-based DSMs can be implemented at the language/compiler layer or the library layer. For the former implementation, new compilers or preprocessors must be developed. Techniques involved in automatic data partition and distribution, parallelism extraction, and communication optimization are still immature and the development efforts are enormous. For library-layer implementations, we only have to support primitives which may be useful for programming and compiler, such as those for distributing data, for utilizing efficient memory consistency model, and for performance tuning. Thus, implementing the DSM in library layer is more feasible.

Another issue in implementing object-based DSMs is choosing a suitable communication subsystem. An ideal communication subsystem must be general enough and have well-defined communication interface, in which system details are encapsulated. PVM is one of the best choices. PVM stands for Parallel Virtual Machine [10]. It enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource [3]. PVM provides process management, message buffering, and other useful message passing utilities. PVM has been ported to many systems and is a de facto standard in high performance computing. A DSM built on top of PVM can support both message passing and shared memory programming. Thus for accesses with known patterns, message passing can be directly used to minimize the number of messages, but for unknown patterns, shared memory accesses can be used [1].

We have designed and implemented an efficient object-based DSM called *Adsmith*. It is implemented at the library layer on top of PVM. Programmers use the system through C++. Although Adsmith can be easily ported to different architectures through PVM, it currently does not support heterogeneous environments for efficiency reasons. Adsmith supports many options that allow the users to specify the properties of each declared object, data access methods, data distribution, memory consistency policies, and communication mechanisms. Although Adsmith is built on top of PVM, applications with careful design can still achieve good performance on Adsmith. Later we will compare applications running on Adsmith and on the bare PVM.

In this paper, we will investigate the issues involved in the implementation of Adsmith and describe its user interface. The rest of the paper is organized as follows. In Section 2, we introduce the design strategies of Adsmith. In Section 3, the user interface and the programming style of Adsmith are described. In Section 4, we show some preliminary performance results of Adsmith. We conclude this paper in Section 5.

2 Implementation Strategies

Since Adsmith is totally independent of the underlying operating system, its performance may suffer. Reducing the number of messages and the communication latency is very important. Several methods are used in Adsmith to solve this problem, including the use of the Release Consistency memory model[1] [4], load/store-like data accesses (Section 2.2), nonblocking accesses (Section 2.4 and 3.4), and atomic accesses (Section 3.7.) We will describe these techniques in more detail below.

2.1 Communication Subsystem

Generally, active messages [2] are used in DSMs to eliminate the need for message buffering and to reduce the access latency. Unfortunately, Adsmith cannot use active messages, because PVM does not provide needed supports. Note that software active messages usually need the help of system dependent functions. Since the communication details are encapsulated by PVM, using any system dependent functions directly in Adsmith could be very dangerous. Beside, PVM uses nonblocking sends. Thus, message buffers in user space always exist. As a result, Adsmith does not employ active messages but use nonblocking sends instead. One advantage of nonblocking sends is to overlap computations and communications, which is important for high performance parallel computing.

2.2 Data Granularity

Data granularity is the unit size of the data during internal data allocation and external transmission. In object-based DSMs, users have the freedom of specifying the data granularity. As a result, the false-sharing problem can be avoided.

A problem with object-based DSMs is that the shared objects tend to be small. Since each reference to a shared object may cause a read or write request, many messages may be generated. Since memory references exhibit locality, we solve the above problem with a *load/store-like memory access style*. A load operation is performed only for the first read access, and a store operation for the last write access. Other accesses for the shared object in the program segment can be performed locally through a cached copy. This is similar to the data access methods in load/store architecture. Load/store operations in Adsmith will be described in Section 3.

2.3 Data Distribution

Under object-based DSM, users have a larger control over how shared data are distributed — based on the application behavior and machine characteristics. In our implementation, the home location of a shared object is randomly selected by default. The programmer or parallelizing compiler can also determine how the shared objects are distributed.

Allowing the home nodes to be moved from time to time is not practical in distributed environments, because many messages will be produced due to the change of home nodes. Adsmith fixes home nodes to simplify the implementation. The problem of such a scheme is that the home node may not be the one which accesses to the object the most. Programmers can help to solve this problem by setting the home node to the host that has the most references to the object. If the process with the most references changes at different execution phases, the programmer/compiler can also force the home to be changed. The ability to

[1] Currently, RC_{sc} is implemented.

manually change the home nodes is now under development in Adsmith.

2.4 Write Policy and Coherence Protocol

Two general write policies are write-through and write-back. In Adsmith, both write policies are supported. Since load/store-like memory accesses are used, the last write accesses are always delimited by the programmer. Thus write accesses need not actually be performed to the home node until the last write access is encountered. The last write access can use a write-through policy, while others a write-back policy.

Pipeline write means that several write requests can be outstanding at the same time. Pipeline write will have no benefit if the write function is invoked for every write access, because this will produce a large amount of messages [7]. Since most accesses in Adsmith are done locally, pipeline write can be used to overlap the communication with the computation.

There are two general 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. Write-update needs to include the content of the modified data in its coherence message. Since Adsmith is an object-based DSM, most objects are small. A write-update and a write-invalidate message will have similar communication costs. To allow maximum flexibility, Adsmith supports both.

2.5 Memory Model

Release consistency (RC) is implemented in Adsmith. Shared accesses in RC are classified as competing accesses (special accesses) and noncompeting accesses (ordinary accesses). Competing accesses mean that two or more accesses may refer to the same shared memory location at the same time and at least one is a write access. Special accesses are further categorized as synchronization accesses and nonsynchronization accesses. Nonsynchronization accesses are competing accesses which are not used for synchronization purposes. Synchronization accesses are further divided into acquire accesses and release accesses. Adsmith provides all these access operations. Programmers are responsible for writing properly-labeled programs by utilizing these operations [4].

2.6 Architecture of Adsmith

Adsmith is completely built on top of PVM. A daemon will be spawned for each host to support runtime shared object handling. Internal manipulations of shared objects are totally transparent to the users. The basic organization of Adsmith is shown in Figure 1.

The architecture can be divided into two layers: *Logical Shared Memory Layer* (LSML) and *Process Buffer Layer* (PBL). LSML is supported by the daemons. Each daemon will interact with the application processes to provide shared-memory services. Shared objects are distributed to the memories of the participating hosts. PBL exists in each application process. Shared data are buffered in PBL and refreshed from and flushed to LSML when necessary. Note that there is no limitation on the buffer size. The whole local memory can be used as buffers. Status information of shared objects are distributed in the data mapping directories on each daemon and the referencing processes.

3 Programming on Adsmith

Adsmith is implemented as a user level library in C++ with PVM as its communication platform. It can be viewed as adding a DSM layer on top of PVM. Both the PVM message-passing library and the Adsmith shared-memory library are accessible at the same time. In this section, we introduce the main functions provided in Adsmith.

3.1 System and Process Control

The programmers need not do any initialization or termination explicitly. All these works are automatically accomplished by the object initialization facilities provided by C++. The library contains a system object, which is responsible for system initialization and termination.

For process creation, although PVM has provided the function pvm_spawn(), we require that child processes be created through adsm_spawn() from Adsmith. This is because some system information will be transmitted during the process creation time.

3.2 Shared Object Allocation and Deallocation

Shared objects can only be allocated at run time. Two forms of the allocation function are supported:

void * **adsm_malloc**(char **identifier*, int *size*,
int *hint* = *AdsmDataDefault*);
void * **adsm_malloc**(char **identifier*, int *size*,
void **init*, int *hint* = *AdsmDataDefault*);

In the declaration, *size* is the size of the shared object and *identifier* is the string name used to refer to the shared object. All shared objects must be allocated before they are used. The parameter, *init*, in the second form is used to set the initial value for that shared object.

Several options can be set through the *hint* parameter to affect the access behaviors of a shared object. Currently, the value of *hint* may be *AdsmDataCache*, *AdsmDataLocal* and *AdsmDataUpdate*. AdsmDataCache means that the shared data will be cached in application processes and managed through the coherent protocol. AdsmDataLocal means that the shared object will be allocated on the local host. This can be used when most accesses of the object are performed by the local processes. AdsmDataUpdate means that write-update will be used as the coherence protocol for the declared object. By default, write-invalidate is used if AdsmDataCache is selected. All these values can be set simultaneously by the *or* operation in C++.

After the shared object is done referencing, the buffer space can be freed by the following function:

adsm_free(void * *ptr*);

Freed objects can be reused by reallocating them again.

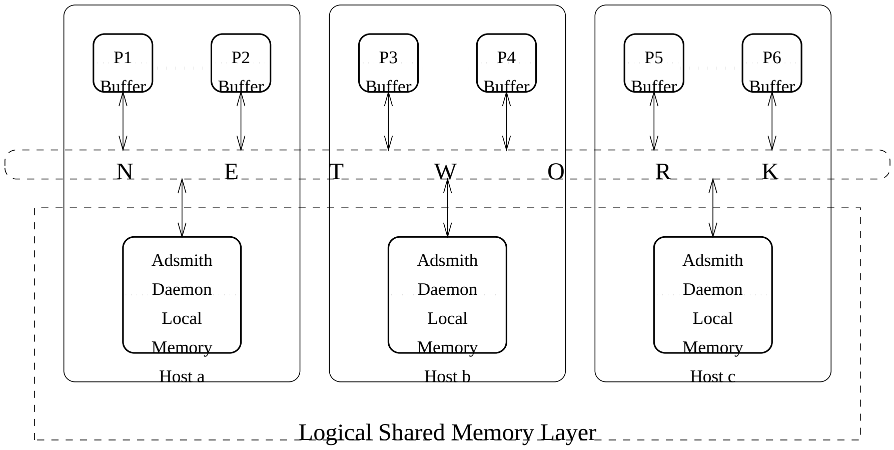

Figure 1: Adsmith system architecture

3.3 Shared Array Declaration

Arrays are often used in scientific computations and the distribution method will significantly affect the execution efficiency. Adsmith allows the programmer to specify the distribution method of an array. The array allocation function has two forms:

> void **adsm_malloc_array**(char * *identifier*,
> int *elmt_size*, int *num*, void * *array*,
> int *hint* = *AdsmDataDefault*);
> void **adsm_malloc_array**(char * *identifier*,
> int *elmt_size*, int *num*, void * *array*,
> int * *dist*, int *hint* = *AdsmDataDefault*);

We explain this function by the following example. Assume that we want to declare a two dimensional array of integers according to a certain distribution method. The code will look like this:

```
int *C[N][N]; // pointers to allocated elements
int distC[N][N]; /* distribution array, which
contains the home node for each element */
adsm_malloc_array ( "arrayC",sizeof(int),
      N*N,C,distC); // allocate
··· refer to each element by *C[i][j] ···
```

The identifier "arrayC" is the name of the whole array. Each element can be referenced by "arrayC[i]", where *i* means the *i*-th element. The home node where each element is distributed to is specified in distC[N][N]. After allocation, pointers to the N*N shared objects are stored in array C. Further references to the shared array can then be made through the returned pointers, i.e., *C[i][j]. Usually, the distribution is determined by the parallelizing compiler or the programmer.

3.4 Ordinary Accesses

After a shared object has been allocated, an address will be returned, which points to the buffer space of the shared object. As described previously, Adsmith uses a load/store-like memory access style. Thus most shared object accesses are done on the local buffer. Actual accesses to the shared memory must be performed through the following two operations. They refresh the buffers from and flush their contents to the shared memory when necessary.[2]

> **adsm_refresh**(void * *ptr*);
> **adsm_flush**(void * *ptr*);

Since no hardware or operating system related facilities are used in Adsmith, data should be manually refreshed (loaded) from LSML by the programmer before they are accessed. Similarly if data are modified, they should be flushed (stored) back to LSML after they are referenced. Under RC, adsm_refresh() is for ordinary loads, and adsm_flush() is for ordinary stores. The value refreshed is guaranteed to be as up-to-date as that at the time of the last acquire (see the next section).

For more efficient data accesses, Adsmith also supports nonblocking load, i.e., data prefetching, through the following function.

> **adsm_prefresh**(void * *ptr*);

The function is very similar to adsm_refresh() and the programmer can insert the prefetch function before the first load access as far as possible. The sequence of shared data accesses in Adsmith is depicted as follows:

> Acquire → Prefresh → Refresh →
> Local Accesses → Flush → Release

where Refresh and Flush are ordinary accesses discussed above. Nonblocking load and store will be performed between prefresh-refresh and flush-release pairs respectively. The code segment below is a typical example to perform computations on a shared object within a critical section:

[2] If AdsmDataCache is specified, **adsm_refresh**() may be performed locally without the need of any communication.

```
typeA *A = (typeA*) adsm_malloc(
                        "A", sizeof(typeA));
AdsmMutex mutex("mutex name");
// AdsmMutex is a synchronization class
mutex.lock();
adsm_prefresh(A);
··· prologue of computation ···
adsm_refresh(A);
··· computations with local access on A ···
adsm_flush(A);
··· epilogue of computation ···
mutex.unlock();
```

Note that adsm_refresh() is still required for the first load access to ensure that the requested data has arrived.

3.5 Synchronization Accesses

Ordinary accesses require that the programmer use enough synchronizations to ensure the correctness. Adsmith provides three classes of synchronization operations: counting semaphore, mutex and barrier. The public methods are listed as follows:

```
AdsmSemaphore::wait();
AdsmSemaphore::signal();
AdsmMutex::lock();
AdsmMutex::unlock();
AdsmBarrier::barrier( int count );
```

Among the synchronization functions, semaphore wait, mutex lock and barrier are acquire accesses, and semaphore signal, mutex unlock and barrier are release accesses. An acquire is needed in order to gain the access right to a set of data, and a release is used to grant the access right. The RC model guarantees that ordinary accesses after an acquire will obtain the most up-to-date data available at the time of the acquire.

3.6 Nonsynchronization Accesses

Adsmith has two nonsynchronization functions:

```
adsm_refresh_now( void * ptr );
adsm_flush_now( void * ptr );
```

These two accesses will be performed without waiting for previous ordinary accesses. That is, a write through adsm_flush_now() will be seen immediately by all the following loads through adsm_refresh_now(), even when they are invoked by other processes.

3.7 Atomic Accesses

Consider accessing a shared object in a critical section. The number of message required may be at most seven, including two for acquire, two for refresh, two for flush, and one for release. It will be expensive when there is only one object in the critical section.

The problem can be solved by allocating the synchronization arbitrator to the home node of the shared object and combining these two operations. During an acquire, the requested data can be piggy-backed on the lock grant message. After the computations, the modified data can also be sent back with the release message. In this way, the required messages will be reduced to four (two for acquire and refresh, and two for flush and release) at most. Since most of the shared objects are small, carrying the data contents directly in the acquire/release messages should not affect the performance.

Adsmith provides *atomic accesses* to support this kind of accesses. Two functions are supported:

```
adsm_atomic_begin( void *ptr,
    int type = AdsmAtomicWrite );
adsm_atomic_end( void *ptr );
```

The function adsm_atomic_begin() can be viewed as a combination of acquire and refresh, while the function adsm_atomic_end() as a combination of flush and release. Note that in Adsmith these operations are categorized as nonsynchronization accesses. It can not be used as synchronization accesses, because coherence of other shared objects are not maintained here. Here is an example of atomic accesses modified from that in Section 3.4.

```
typeA *A =(typeA*) adsm_malloc("A",
                        sizeof(typeA));
adsm_atomic_begin(A);
··· computation with local access on A ···
adsm_atomic_end(A);
```

Let the program segment between adsm_atomic_begin() and adsm_atomic_end() be called *atomic section*. Two types of atomic operations can be specified in the *type* parameter in adsm_atomic_begin(): AdsmAtomicWrite and AdsmAtomicRead. The former is to indicate that both read and write accesses are included in the atomic section; and the latter is to indicate that only read accesses exist in the section. Adsmith implements *single-writer/multiple-readers* protocol. For a writer, adsm_atomic_begin() can be performed only when there are no readers nor writers in the atomic section. For a reader, adsm_atomic_begin() can be performed only when there is no writer in the atomic section. For fairness purpose, Adsmith implements the *writer first* protocol. That is, when a writer is waiting to enter the atomic section, readers which come after the writer will be blocked until the writer has finished its atomic section. Of course, readers before the writer can proceed until they all exit their atomic sections.

3.8 Pointer

Pointers in shared memory are supported in Adsmith, but the usage is not so straightforward. This is because the address of a shared object in one process may not be the same as that in the other process. Thus, the programmers are required to translate the local address of a shared object to a globally recognizable address before the address is passed to other processes through the shared memory. Functions for pointer manipulations are as follows:

```
int adsm_gid( void *ptr );
void * adsm_attach( int gid );
```

The function adsm_gid() translates the local address of a shared object into its global address, which is represented by an integer. The function adsm_attach(), on the other head, translates a global address back to the local address for the requesting process.

For example, if the programmer wants to pass the address of shared object T from process *P1* to process *P2* through the shared object (pointer) S, the following two code segments can be used.

```
// process P1 sets the pointer
AdsmBarrier Bpointer("barrier for this code");
sometype *T = (sometype*) adsm_malloc(
                  "target data", sizeof(sometype));
int *S = (int*)adsm_malloc( "pointer to T",
                                  sizeof(int));
*S = adsm_gid(T); // get global address of T
adsm_flush_now(S); // flush immediately
Bpointer.barrier(2); // done
// process P2 gets the pointer
AdsmBarrier Bpointer("barrier for this code");
int *S = (int*)adsm_malloc( "point to T",
                                  sizeof(int));
Bpointer.barrier(2); // wait until P1 is done
adsm_refresh_now(S); // get the pointer value
// attach the pointer into local address space
sometype *T = (sometype*)adsm_attach(*S);
```

4 Performance Evaluation

In this section we study the performance of Adsmith through an application program that solves the Traveling Salesman Problem. We will compare the performance of the application programs developed in Adsmith and in PVM. The PVM version was written following the master-slave programming model. We ported it onto Adsmith using the SPMD (Single Program Multiple Data) model with the algorithm unchanged. There are two major communication parts in the program, which are used to compute a global maximum (minimum) from local maximums (minimums). The related code segments are listed below.

PVM version:

Master Program

```
int slaves[SLAVE_NUM]; // slave tids
// get local maxs and compute the global max
float max=0.0;
for (int i=0; i<SLAVE_NUM; i++) {
      pvm_recv(-1,SOME_TAG);
      pvm_upkfloat(&local_max,1,1);
      if (local_max>max) max=local_max;
}
// broadcast the global max
pvm_initsend(PvmDataDefault);
pvm_pkfloat(&max,1,1);
pvm_mcast(slaves,SLAVE_NUM,SOME_TAG);
```

Slave Program

```
int master; // master tid
// compute local max
float local_max= ···
// send local max to master
pvm_initsend(PvmDataDefault);
pvm_pkfloat(&local_max,1,1);
pvm_send(master,SOME_TAG);
// get global max from master
float max;
pvm_recv(master,SOME_TAG);
pvm_upkfloat(&max,1,1);
```

Adsmith version:

```
// compute local max
float local_max= ···
float *max=(float*)adsm_malloc(
      "max",sizeof(float));
// compute max thru atomic operation
adsm_atomic_begin(max);
if (local_max>*max) *max=local_max;
adsm_atomic_end(max);
// wait for all processes done
AdsmBarrier Btsp("tsp");
Btsp.barrier(PROC_NUM);
// get the global max
adsm_refresh(max);
```

Atomic accesses are used in the Adsmith version, because there is only one shared object in the critical section. From the code, we can roughly compute the ratio of the number of messages required by the PVM version to that by the Adsmith version, which is about 1:4. One reason for the larger number of messages in the Adsmith version is because we did not write the Adsmith version from scratch, but only translate it from the PVM version directly.

Two Sparc 2 workstations were used in the experiment. The performance results are shown in Table 1. Surprisingly, we find that as the problem size increases, the execution time of the Adsmith version becomes closer to that of the PVM version. It is even shorter when the number of cities is greater than 10,000. One explanation is that the communications are overlapped between processes in Adsmith. Besides, the implementation overheads are lower in Adsmith.

5 Conclusion

In this paper we have introduced an object-based approach to DSM designs and described how such a system, called Adsmith, is built. Major features of Adsmith are listed below:

1. It is an object-based DSM implemented as a user-level library in C++ on top of PVM.
2. It has a two-level memory hierarchy: process buffer layer and logical shared memory layer.
3. Information about shared objects is distributed in both daemons and application processes. Programmers are allowed to specify the distribution of shared objects/arrays.

Cities	PVM version	Adsmith version	Speedup
6000	1629.07	1745.91	-7.17%
7000	2188.47	2298.07	-5.01%
8000	3055.45	3113.7	-1.91%
9000	3565.23	3594.6	-0.82%
10000	4381.78	4293.12	2.02%
20000	17474.32	16103.1	7.84%
30000	38822.39	35408.4	8.79%
40000	37133.62	31138.7	16.14%

Table 1: Execution times for the TSP program

4. The home nodes of shared objects are fixed. Programmers can determine whether the shared object is to be cached or not.

5. It employs a load/store-like data access style and the Release Consistency model is fully supported.

6. It provides atomic accesses to minimize the number of messages.

7. Nonblocking store (pipeline write) and nonblocking load (prefetch) are used to overlap communications and computations.

8. Both write-through and write-back are supported. Different shared objects can have different coherence protocols.

Since we built our system on top of PVM, the implementation considerations were quite different from others. For example, using PVM prevents us from adopting the active message mechanism, and a higher communication overhead is involved. Thus, our major task is to reduce the number of messages. Many features listed above help to achieve this goal. In addition, many flexibilities are provided to help performance tuning, especially for the parallelizing compilers. For example, shared objects can be set to use cache or not, to use write-update or write-invalidate, etc. Home nodes can be assigned by the programmer or the compiler, so can the distribution of shared arrays. Also prefetching is supported to hide the load access latency. Preliminary experimental results show that Adsmith is efficient and can achieve very good performance.

References

[1] Tzi-cker Chiueh and Manish Verma, "A Compiler-Directed Distributed Shared Memory System," *9th ACM International Conference on Supercomputing*, 1995.

[2] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

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

[4] 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.

[5] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory," In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

[6] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas and Willy Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," In *Proceedings of the 1994 Winter Usenix Conference*, pp. 115-113, Jan 1994.

[7] Wen-Yew Liang, "ADSMITH: A Structure-based Heterogeneous Distributed Shared Memory on PVM," *Master Thesis*, National Tsing Hua University, Taiwan, June 1994.

[8] 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.

[9] Steven K. Reinhardt, James R. Larus, and David A. Wood, "Tempest and Typhoon: User-Level Shared Memory," In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, April 1994.

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