Adsmith: An Object-Based Distributed Shared Memory System for Networks of Workstations *†

Wen-Yew Liang‡ Chun-Ta King§ Feipei Lai¶

Abstract

This paper introduces an object-based distributed shared memory (DSM) system called *Adsmith*. The primary goal of Adsmith is to provide a low-cost, portable, and efficient DSM for networks of workstations (NOW). Adsmith achieves this goal by building on top of PVM, a widely supported communication subsystem, as a user-level library and by incorporating many traffic reduction and latency hiding techniques. Issues involved in the design of Adsmith and our solution strategies will be discussed. Preliminary performance evaluation of Adsmith on a network of Pentium computers will be presented. The results show that programs developed with Adsmith can achieve a performance comparable to that developed with 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 natural programming paradigm. With message passing, however, 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 channel 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 organization with shared-memory programming is thus a more desirable architecture for parallel computers. That is the primary reason why *distributed shared memory* (DSM) systems have attracted so much attention [22].

A DSM presents to applications a logically shared memory on top of a network of processors with physically distributed memories. Many DSM systems have been developed, which followed different design strategies and emphasized different aspects of performance and functionality [22, 24]. In this paper we describe a DSM system called *Adsmith*, which is targeted at networks of workstations (NOW) with low cost, portability, flexibility and efficiency.

Adsmith is implemented entirely as libraries that can be linked to the application programs. It does not require any special hardware supports nor modifications to the systems. Portability of Adsmith is achieved through its use of PVM [9] for interprocess communication. With the wide acceptance of PVM in scientific computing and its support by almost every workstation manufacturer, Adsmith can

*A version of this paper is to appear in *IEICE Trans. Information and Systems.*

†This Research was partly supported by National Science Council, Taiwan, NSC 86-2213-E-002-041 and NSC86-2213-E-007-043.

‡The author is with the Dept. of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan.

§The author is with the Dept. of Computer Science, National Tsing Hua University, HsinChu, Taiwan.

¶The author is with the Dept. of Electrical Engineering and Dept. of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan.

easily be ported to different NOW. Adsmith supports many options that allow knowledgeable programmers to flexibly control the behavior of their application programs. Adsmith also incorporates many optimization techniques that applications with careful design can achieve a performance comparable to that developed directly with PVM.

As far as we know, a DSM system with the above-mentioned features is unique. In this paper, we will discuss the issues involved in developing Adsmith and the optimization techniques taken. Usage of Adsmith will be introduced, and performance of benchmark programs developed with Adsmith will be compared with that using PVM. The results serve as an indicator of the efficiency of Adsmith. Finally, DSMs similar to Adsmith are reviewed briefly, whose functionalities and features are compared with Adsmith.

2 Issues and Design Strategies

This section discusses the issues involved in developing Adsmith and the solution strategies that were taken.

Page-based versus Object-based

Many previous approaches [24] to DSMs partition the global addressing space into fix-sized logical pages. Through a memory manager, a process can access to any page in the shared addressing space. Such *page-based* DSMs are often taken as an extension of traditional virtual memory systems, and are usually implemented at the hardware or operating system layer. One advantage of such an implementation is transparency — the memory system is totally hidden from the users. Unfortunately it is not easy to choose a proper page size that satisfies all applications. Design complexity also makes this approach expensive and difficult to port to different systems.

Object-based DSMs, on the other hand, view the shared memory as consisting of many objects sharable by all processes. They are most often implemented at the language, compiler or library layer. DSMs implemented at these layers usually require some modifications at the users' end — either to the programming model, language, or style. Thus, transition from sequential computers to such environments is not transparent. Performance is also a concern. The implementation must overcome the high communication overhead and inefficiency in existing systems in order to get performance.

However, object-based DSMs do offer some unique and important features. Object-based DSMs are more flexible and system independent than page-based DSMs. By allowing programmers to determine important parameters, such as block size, data distribution, memory consistency model, etc., behavior of applications can be better controlled according to their characteristics. Furthermore object-based DSMs can be directly implemented on top of existing communication and operating systems, such as Unix, TCP/IP, and PVM. System development efforts can thus be reduced dramatically. Also, porting to different architectures is straightforward and the resulting system can be easily modified and improved when newer techniques are available. Thus, Adsmith adopts the object-based approach.

Compiler or Library?

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 [2]. 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, maintaining memory consistency, and optimizing communication. Thus implementing the DSM in the library layer is more feasible, which is also the approach taken in Adsmith.

Reducing the Communication Overhead

Reducing the communication overhead is the most critical design issue for NOW. This is especially difficult for those NOW without special hardware supports. Now, the object-based DSMs attempt to add yet another layer of software on top of existing systems. The communication problem becomes even worse. The problem can be addressed at several levels.

1. Communication subsystem:
   A communication subsystem provides low-level message passing mechanisms to support DSMs. Efficiency of the subsystem impacts directly on the supported DSM. An intuitive design strategy is to bypass the bulky communication software, e.g., TCP/IP, and replace it with a highly optimized subsystem [7, 23]. Unfortunately, since this approach usually requires a higher integration with the lowest-level hardware, portability of the resultant system is a problem.

   If portability is a major concern, then it is a good design choice to use existing, widely accepted communication subsystems, while leaving the efficiency issue to other components of the system. Adsmith takes this approach and is built on top of PVM. PVM enables a collection of heterogeneous computers to be used as a coherent concurrent computing engine. It has been ported to many systems and accepted by scientific computing community for application development. Through PVM, Adsmith can support both message passing and shared memory programming, and leave process management, message buffering, and other message passing functionalities [4] to PVM.

2. Data distribution:
   Communication overhead can be reduced dramatically if data are allocated to the distributed memories properly. In general, programmers have the best knowledge of the application. Thus they must be allowed to specify how shared data are distributed, according to the application behavior and machine characteristics. Adsmith provides this flexibility and determines the home location of a shared object through the help of the run-time system, the programmer, and the parallelizing compiler.

   A data object, once allocated, may or may not change its home location. In the former case, data may change its home node dynamically or be redistributed from time to time according to the run-time status and application requirements. Conversely, the latter approach eliminates the extra communication overhead caused in changing the home node. This thus is more suitable for operations such as reduction. Adsmith allows the home node of a shared object to be fixed or movable.

3. Consistency model and coherence policy:
   A weaker memory consistency model can reduce shared memory accesses or defer them for later aggregation, whereby reducing network traffic and hide communication overhead [16]. Adsmith implements the RC_{sc} release consistency model [10], in which competing accesses are sequentially consistent. Programmers or compilers, however, are responsible for writing properly-labeled programs [10] by utilizing the provided primitives.

   For shared memory systems using caching, data coherence policies are crucial to the correctness of application execution and they affect the communication overhead indirectly [16]. Unfortunately, different applications have different data access characteristics and must be supported by different policies to get the optimal performance. Again with its software implementation, Adsmith supports both write-update and write-invalidate policies. Users can also choose from write-through or write-back policies. To implement the coherence protocol, Adsmith employs the *acknowledge-free coherence* *scheme* [19] to remove all acknowledge messages that are originally required in invalidate or update operations. In this way, messages involved in the coherence protocol can be greatly reduced.

4. Shared object accesses:

Communication overhead of a DSM system can be reduced by various degrees with optimized accesses to shared objects. Adsmith employs the following optimization techniques.

- *Access style:* Ideally, users of a DSM need not distinguish between local and remote data accesses and issue calls to different objects differently. Unfortunately, an implementation to achieve this goal is very involved and may not be efficient. Adsmith adopts a load/store access style for shared accesses. Let the time between an acquire access and its immediate following release access in a process be called an *interval*. Before an object is first read during an interval, users need to perform a load operation called *refresh* to load the up-to-date value from the DSM system into the local buffer. At the end of the interval, if the object was modified, a store operation called *flush* must be performed to write the data back to the DSM system. Other accesses to the object during the interval can be performed locally through the buffer. In this way most references in an interval are local, which in turn reduces communication overhead.

- *Computation with communication overlapping:* To allow computation to overlap with communication, Adsmith supports *prefetching*, which allows a process to issue load requests far before the requested object is actually used. Then, when the object is referenced, its data is already available locally. Similarly, Adsmith provides *nonblocking stores*, which allow a process to resume its computation immediately after sending a write request. The acknowledge message will be handled later. To achieve further overlapping, Adsmith also supports a nonblocking version of the acquire operation, through which computation can proceed before these accesses are really performed.

- *Bulk transfer:* By aggregating data which are to be sent to the same host together into one message, we may reduce the number of messages transmitted. Adsmith provides primitives for bulk transfer.

- *Multiple writers:* Multiple-writer protocols have been used to avoid the false-sharing problem in page-based DSMs [12]. Adsmith does not have the false-sharing problem, because shared objects are not allocated on a page basis. However, Adsmith does support an *object-based multiple-writer protocol* for eliminating synchronizations such as critical sections. The protocol allows several processes to write to different parts of a shared object at the same time. If no two processes write to the same part, then there is no need for a critical section to protect the object. This also helps to reduce the communication overhead.

- *Specialized accesses:* Several access patterns are very common in applications. By providing specialized primitives to handle these accesses, it is possible to further reduce communication overhead. Adsmith provides primitives to combine synchronization operations with shared object accesses. Such accesses are called *atomic accesses* and they are useful for situations where a single shared object is accessed within a critical section. Atomic accesses can be further specialized to support barrier synchronizations. Such accesses are called *collect accesses* in Adsmith and are used for operations such as data accumulation. Adsmith also supports *active accesses*. In active accesses a short code segment, such as an expression, is transmitted to the host owning the requested data, where the code is executed atomically. In this way, there is no need to transfer the data back and forth between the host and the local node.

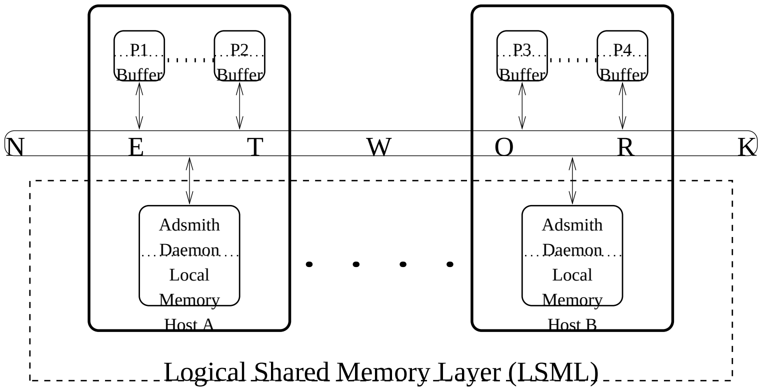


Figure 1: System architecture of Adsmith

3 Programming with Adsmith

Adsmith is implemented as a user-level library in C++ with PVM as the communication subsystem. 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. This section gives an overview of the functions provided in Adsmith and its programming conventions. Details of Adsmith implementation can be found in [18].

3.1 Organization of Adsmith

The basic organization of Adsmith is shown in Fig. 1. There are two logical layers: *logical shared memory layer* (LSML) and *process buffer layer* (PBL). LSML is supported by daemons, one for each host. 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 is implemented inside each application process, which is linked as library routines to the application program. Shared data are buffered in PBL and are loaded from and stored to LSML when necessary. Note that there is no limitation on the buffer size and the whole local memory can be used as buffers. Status information of shared objects are distributed among the data mapping directories in each daemon and the referencing processes.

3.2 System and Process Control

With Adsmith, there is no need for the programmers to initialize or terminate the run-time system explicitly. All these work are done implicitly by the Adsmith library. Adsmith provides its own spawning function `adsm_spawn` so that the shared memory functionality can be properly enabled between the parent and the child processes. A tree-like parent-child relationship is maintained among processes. The parent process will not terminate until all its children have terminated. Adsmith also provides the functions to explicitly wait for specific child processes to terminate.

3.3 Shared Object Allocation

In Adsmith shared objects must be allocated before they are used. The home node where the shared object is to be allocated can be specified with the allocation primitives. Currently Adsmith supports the following options, or *properties*, to affect the access behavior of a shared object: *AdsmDataLocal*, *AdsmDataCache*, *AdsmDataUpdate*, *AdsmDataMWriter*, and *AdsmDataMovable*. AdsmDataLocal will cause the shared object to be allocated on the local host. AdsmDataCache specifies that the shared data will be cached in application processes and managed through the coherence protocol. AdsmDataUpdate indicates that write-update coherence protocol is to be used for the declared object. By default, write-invalidate will be used. AdsmDataMWriter tells the system that the object-based multiple-writer protocol will be used on the declared shared object. AdsmDataMovable allows the home node to be changed dynamically. Combinations of these options can be set by ORing them together.

After a shared object has been allocated, an address will be returned, which points to the buffer space of the object. Every shared object has a buffer space in the process which has allocated the object. Data in a buffer will be up-to-date only when the acquire access is performed, according to the definition of release consistency. Without caching *every* load request needs to be forwarded to the home node to fetch the data. Caching avoids unnecessary communication if the up-to-date data exist in the buffer.

3.4 Shared Object Accesses

In release consistency, memory operations are classified as ordinary accesses, synchronization accesses, and non-synchronization accesses. Adsmith provides the `adsm_refresh` primitive for ordinary loads and `adsm_flush` for ordinary stores. Under the load/store-style memory accesses, refresh is invoked only once before the first load access following an acquire, and flush is invoked only once after the last store access preceding a release.

Adsmith supports three types of synchronization operations: semaphore, mutex and barrier. Among the synchronization functions, semaphore wait, mutex lock and barrier are acquire accesses, and semaphore signal, mutex unlock and barrier are release accesses. Under the RC model, non-synchronization accesses are competing accesses that are not for synchronization purposes. Adsmith has two basic non-synchronization primitives: `adsm_refresh_now` and `adsm_flush_now`. A store through `adsm_flush_now` will be seen *immediately* by all the following loads through `adsm_refresh_now`, even when they are invoked by different processes.

3.5 Advanced Functions

Adsmith supports advanced functions for reducing communication overhead. Data prefetching is specified in Adsmith with the primitive `adsm_prefresh`. It can be inserted far before the actual load access is called. Adsmith supports bulk transfer by providing two sets of delimiter primitives, which are used to tag the beginning and the end of the corresponding group of functions or accesses for bulk transfer. Currently allocation functions, prefetch, refresh, and flush accesses can be aggregated.

Atomic accesses are supported in Adsmith with the primitives: `adsm_atomic_begin` and `adsm _atomic_end`. The former can be viewed as a combination of acquire and refresh, while the latter as a combination of flush and release. Let the program segment between the two functions be called an *atomic section*. Atomic accesses can be further classified into two types: AdsmAtomicWrite and AdsmAtomicRead. AdsmAtomicWrite indicates that both read and write accesses are included in the atomic section, while AdsmAtomicRead indicates that only read accesses exist in the section. A writer can enter the atomic section only when there is no reader nor writer inside. A reader can enter the atomic section only when there is no writer inside.

Collect accesses are supported in Adsmith with functions: `adsm_collect_begin` and `adsm_collect _begin`. These functions specialize atomic accesses to barrier synchronizations for further reduction in messages. Finally active accesses are supported with the function `adsm_atomic`, in which an expression string is given as a parameter. The shared object will be accessed and manipulated atomically by its owner host according to the expression.

Table 1: Configuration of the machines in the experimental environment

Component	Type
CPU	Pentium-100
RAM	32M
Network	10 Mbps Ethernet
OS	Linux 2.0.27
PVM	Version 3.3.11
Adsmith	Version 1.8.0

3.6 Other Features

Pointers to shared memory are supported in Adsmith, but with some restrictions. Programmers are required to translate the local address of a shared object to a globally recognizable address through the function `adsm_gid` before that address is passed to other processes. The processes which obtain the pointer then translate the global address back to the local address through the function `adsm_attach`. This restriction is needed because the address of a shared object in one process may not be the same as that in another process.

Adsmith allows programmers to use message passing functions in PVM together with its shared memory functions. Thus, for example, for accesses with known patterns, message passing can be used directly to minimize the number of messages, but for unknown patterns, shared memory accesses are used [8]. However, there are two restrictions when using PVM messaging. First, processes must be spawned using `adsm_spawn` if they are Adsmith programs. Second, seven message tags are reserved by Adsmith, which are from MAXINT down to MAXINT−6, where MAXINT is the largest integer defined in the system. These tags cannot be used in PVM message functions.

4 Performance Evaluation

This section evaluates the performance of benchmark programs written in Adsmith. The results are compared with the same programs developed using PVM. Due to equipment limitation, our experiments could only be conducted on four Pentium machines, whose configurations are listed in Table 1. In the experiments, the time and messages induced during system startup and shared object allocation were not counted for fair comparison. Both Adsmith and PVM applications used the PVM options PvmRouteDirect and PvmDataInPlace [4].

4.1 Performance of Basic Primitives

In this set of tests, one process was used to access 100 shared integer objects distributed over four machines. No properties were set for the objects. The averaged time to access these 100 objects are listed in Table 2.

In the table, N represents the number of hosts, i.e. four in our experimental environment. The ordinary store operation "flush" uses bulk transfer implicitly. The expressions $1+1$ or $N+N$ in some entries mean that the reception of reply messages can be deferred until the time when the object is referenced or until the release time of the store accesses. The corresponding time for these entries is shown in the form x/y, where x is the time without waiting for the reply message and y includes the time for the reply message. Since the shared objects are distributed on N machines, the total number

Table 2: Measured time of Adsmith primitives

Primitive	Ave. Time (msec)	Ave. Message Count
allocation	1.091/1.597	$1+1$
bulk allocation	0.100/0.180	$\frac{N+N}{100}$
flush now	0.785/1.257	$1+1$
(bulk) flush	0.002/0.047	$\frac{N+N}{100}$
refresh now	1.249	2
refresh	1.260	2
bulk refresh	0.051	$\frac{2N}{100}$
prefresh	0.765	1
bulk prefresh	0.032	$\frac{N}{100}$
atomic begin	1.263	2
atomic end	0.764/1.267	$1+1$
atomic expr	1.397	2
collect begin	1.266	2
collect end	0.759	2
mutex lock	1.252	2
mutex unlock	0.506	1
semaphore wait	1.295	2
semaphore signal	0.522	1
barrier	1.300	2
spawn	3.679	

of messages for a bulk transfer is N. The primitive "atomic expr" measures the time in running the primitive `adsm_atomic` using the expression "[int] @+=1" [18], which increments the specified object by one.

Note that the measured time of the ordinary accesses also represents the penalty of misses in the local buffer. To compare, we measured the hit time of these accesses, which is about 0.001 milliseconds. The large gap between the two indicates that bulk transfer is really useful. With bulk transfer the average transmission time is greatly reduced and the difference between the hit time and the miss time is shortened. Comparing the entries with x/y, we can see that deferring the reply messages can achieve a better overlapping between computation and communication and thus reduce the total execution time. In the table the time of "refresh now" and that of "refresh" are close, because they basically perform the same set of operations. However, in practice "refresh" may not be actually performed because the most up-to-date data may have already been in the cache. Finally, comparing the use of atomic accesses and that of ordinary accesses plus synchronization accesses, we can see that the former can save about half of the execution time. Similar effects can be found for the collect accesses.

4.2 Adsmith versus PVM

This section compares the performance of Adsmith and PVM based on two benchmark programs: TSP and Jacobi. TSP solves the Traveling Salesman Problem on 24000 cities connected in a complete graph. Jacobi is a simple Successive Over-Relaxation program on a 1500×1500 matrix with 100 iterations. Both algorithms have a time complexity of $O(N^2)$. Results of the experiments are shown in Fig. 2.

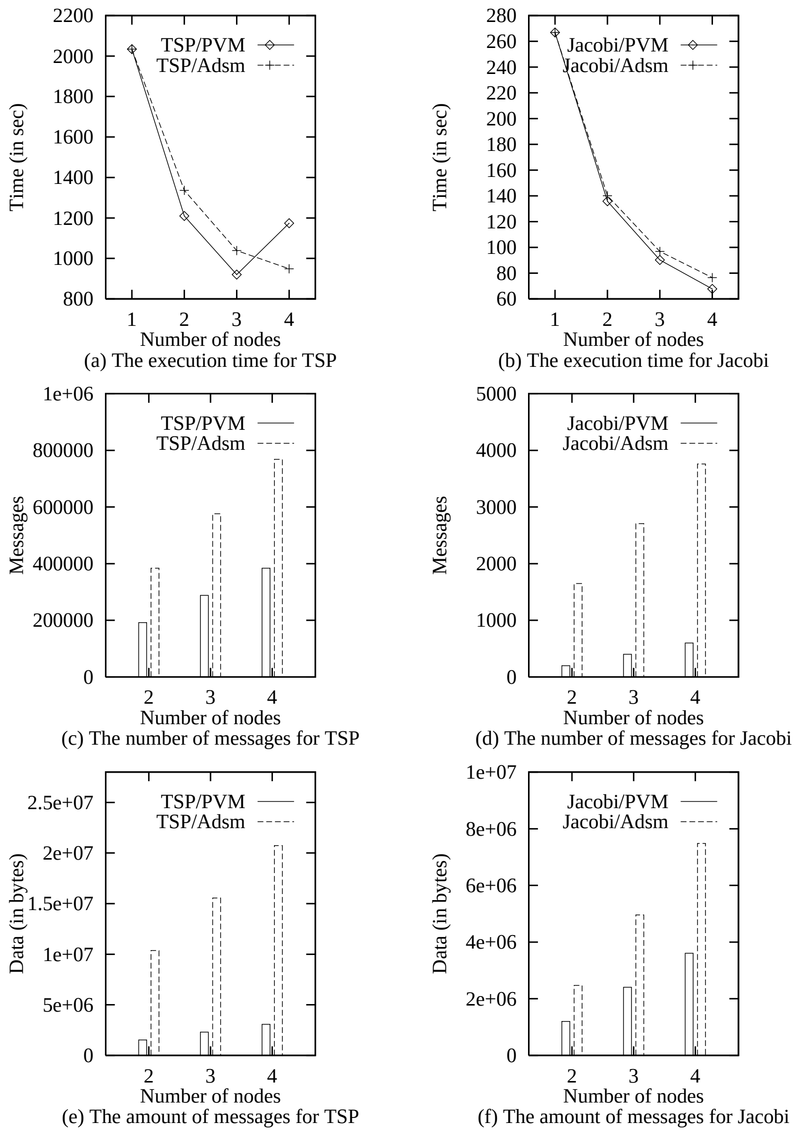


Figure 2: Comparison of benchmark programs written in Adsmith and PVM

TSP

Communication in TSP occurs in two places, one to find the maximum value and the other to find the minimum. In the PVM version, a master process is responsible for receiving the local maximum and minimum from every slave process to calculate the global maximum and minimum. In the Adsmith version, a maximum/minimum record is shared among all the processes for finding the maximum and minimum. The shared record is referenced through collect accesses.

Fig. 2(a) shows that the execution time of the Adsmith version (TSP/Adsm) is within 13% of that of the PVM version (TSP/PVM). This implies that the overhead introduced by Adsmith is not very significant and that the performance of programs developed with Adsmith is comparable with that using PVM. Note that the execution time of TSP/PVM increases when the number of nodes is four. This is because the master process becomes a bottleneck during the reduction phase for calculating the maximum and minimum.

From Fig. 2(c), we can see that the number of messages transferred in TSP/Adsm is twice as many as that of TSP/PVM. Furthermore, the amount of data transferred in TSP/Adsm is six times more than that of TSP/PVM, according to Fig. 2(e). The reason is that, although the shared record is small, the extra system messages associated with the collect accesses are relatively large. The problem can be easily solved by adopting denser data structures in the implementation.

Comparing timing statistics with message statistics, we can find that the execution time does not increase in proportion to the number of messages and the amount of data transferred. This is a good evidence that Adsmith implementation is able to hide the communication latency by overlapping communication with computation.

Jacobi

For Jacobi, communication only occurs when two neighboring processes need to exchange their boundary data. In the PVM version (Jacobi/PVM), each process receives two rows of data from each of its neighbors at the beginning of each iteration. After calculating the new values, it sends two rows of boundary data to each of its neighboring processes. In the Adsmith version (Jacobi/Adsm), each process reads the boundary rows owned by each of its neighbors at the beginning of each iteration. After updating local elements, it stores its new boundary data to the shared memory. Two barrier synchronizations are used in each iteration.

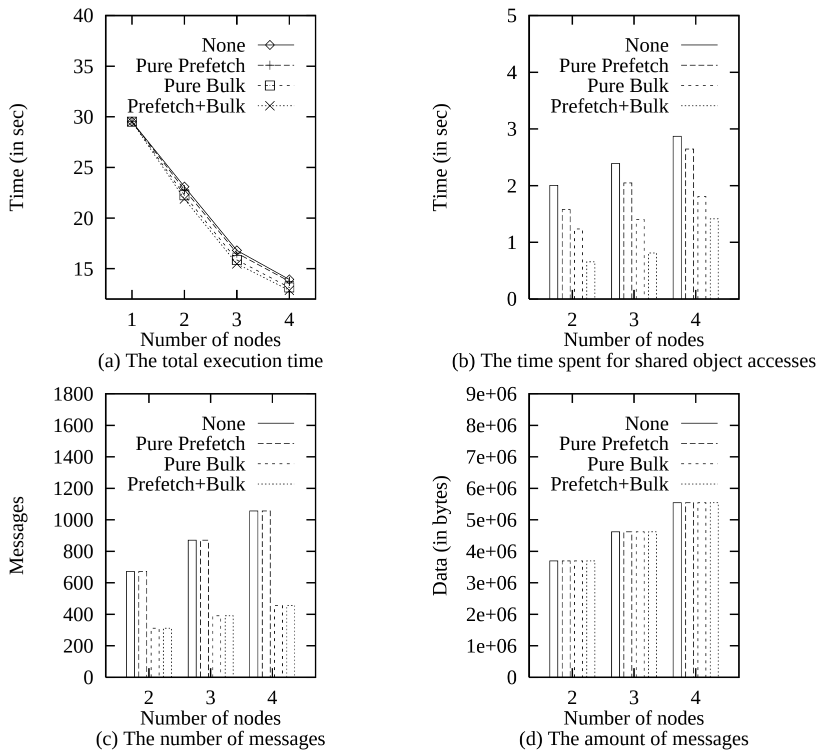
We can see from Fig. 2(b) that the execution time of Jacobi/Adsm is also within 13% of that of Jacobi/PVM. However, the message statistics are quite different from those of TSP. The number of messages transferred in Jacobi/Adsm is from six times to eight times more than that in Jacobi/PVM; while the amount of data transferred in Jacobi/Adsm is only twice the amount of that in Jacobi/PVM (see Fig. 2(d) and (f)). This is because half of the messages are generated from the two barrier synchronizations in Jacobi/Adsm, and the shared objects are large relative to the extra data associated with the request messages. Again, the efforts by Adsmith in hiding the communication latency result in a comparable execution time between Jacobi/Adsm and Jacobi/PVM, despite a much larger amount of communication.

4.3 Effects of Bulk Transfer and Prefetch

In this subsection, we evaluate Adsmith's implementation of bulk transfer and prefetching. The evaluation is based on a test program which performed matrix multiplication on two matrices of size 480×480. Each matrix was partitioned into 96 square sub-matrices, each was declared as a shared object. There were totally 96×3 shared objects declared and distributed to the four nodes.

In the test program, the bulk transfer optimization aggregated the accesses to four objects and sent all requests in one message. Prefetching was issued one iteration ahead of the iteration where the object or the bulk of objects was accessed. Four cases were compared: *none* using no optimization, *pure prefetch* using prefetch only, *pure bulk* using bulk transfer only, and *bulk+prefetch* using both bulk transfer and prefetch. The results are shown in Fig. 3.

Fig. 3(a) and (b) show the total execution time and the time spent in accessing shared data, respectively. We can find that bulk transfer improves the performance (or reduces the time spent in shared data accesses) more than prefetch. The reason why the total execution time does not improve much is because caching is used in the program. As a result communications (caused by read misses) occur only at first-time read accesses, which constitute a small portion of the total execution time. Note that prefetch targets at more aggressive overlapping between computation with communication, but it does not reduce the messages transferred (see Fig. 3(c) and (d)). Since the cost of transmitting a message is very high in NOW, bulk transfer, which focuses primarily on reducing the number of messages (see Fig. 3(c)), is thus more effective. The access time in Fig. 3(b) increases as the number of nodes increases



(a) The total execution time (b) The time spent for shared object accesses (c) The number of messages (d) The amount of messages

Figure 3: The effects of prefetching and bulk transfer

because shared data are distributed among more number of nodes and thus more messages are generated. Note also that the amount of data transferred with bulk transfer is smaller than that without bulk transfer, because messages are combined and thus system messages associated with the requests are reduced. Finally, the table shows that the two optimizations can be applied together constructively to obtain further performance improvements.

4.4 Effects of Object-based Multiple-Writer Protocol

In this subsection, we evaluate the effectiveness of using the object-based multiple-writer protocol. In this experiment, TSP working on 24000 cities was used. A vector recording local maximum and minimum was shared among the processes. Each process put its own local maximum and minimum into the corresponding slot in the vector. After all the slots in the vector had been filled, the processes read the whole vector and calculated the global maximum and minimum themselves. A barrier synchronization was used to ensure that the vector was ready to be read.

Two versions of TSP were compared. The first version, called CS, used critical sections to protect the shared vector. The second version, called OMW, specified the shared vector with the property 'AdsmDataMWriter' and used the object-based multiple-writer protocol to access the vector. The performance results are shown in Fig. 4.

From the table, we can see that the number of messages transferred is greatly reduced by removing the critical sections in OMW. In addition the amount of data transferred is also decreased, because only the modified portions of the shared vector are transmitted. Although the object-based multiple-writer protocol requires extra overhead to calculate and maintain the modified portions, the total execution time is actually reduced due to the reduced number of messages for synchronization. This shows that the protocol is indeed helpful for improving the performance.

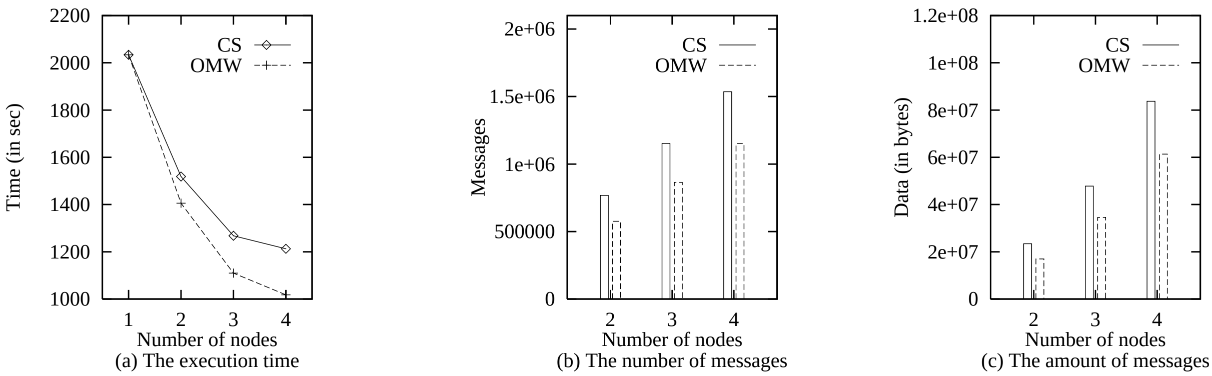


Figure 4: The effects of using object-based multiple writer protocol

5 Related Work

Adsmith emerges as a highly portable DSM system. It differs from most other DSM systems in that it is completely system independent by building atop the widely supported, de facto message passing standard — PVM, without language or compiler assistance. Other DSM systems, on the other hand, usually require either system software supports for memory management, networking, and interrupts [17, 5, 30, 12, 11], or hardware supports, e.g., special network interface for efficient communications [14, 25] or cache coherence [15, 26].

Among the existing DSM systems, Munin [5, 6] shares the most common features with Adsmith: both support release consistency (RC) model and both manipulate shared data with objects. They also allow users to specify the properties of each shared object. For example, the shared object can be replicated elsewhere, fixed in a particular site or movable, can be invalidated or updated when modified, and can be simultaneously modified by multiple writers, etc. In addition, they both provide such advanced features as combining synchronization operations with ordinary accesses and prefetching.

However, it is not straightforward for Munin to port to other systems, because Munin requires a linker to create the shared memory segments and operating system supports to handle page faults and manipulate page tables. In reducing consistency related communication, Munin provides an *update timeout mechanism* [6] to invalidate replicas of a cached variable that has not been accessed recently. The aim is to reduce the number of messages with unnecessary updates or stale updates. TreadMarks [12], a descendant of Munin, supports a variation of the RC model, called the *lazy release consistency* (LRC) model [13]. It postpones the propagation of modifications to the time of the acquire such that the messages for maintaining consistency can be minimized. However, it requires more memory space to keep related information and incurs extra protocol overhead.

Adsmith instead uses the *acknowledge-free coherence scheme* [19] to remove all acknowledge messages that are originally required in invalidate or update operations. This greatly reduces the messages involved in the coherence protocol. The scheme only requires constant space and can be implemented easily. Adsmith provides other features to optimize message transmissions, which are not available in Munin. For example, Adsmith supports bulk transfers of load accesses in addition to message aggregation for write accesses.

The *function-shipping* idea proposed in Locust [8] is similar to active accesses in Adsmith. As active accesses are limited to C expressions, function-shipping supports arbitrarily complex, user-defined operations. However, function-shipping has to be processed first by the compiler, while active accesses can be directly utilized by users through the Adsmith library.

Some DSM systems achieve portability through programming language extensions [1, 3, 8, 20]. However, most of them require compilers to translate the source code into programs that call lower-level message passing primitives. Instead, Adsmith chooses to separate shared memory functionalities from compiler optimizations. The rationale is that the share memory system should concentrate on run-time optimizations and support compilers with useful primitives. On the other hand, compilers should dedicate their efforts on static optimizations and place the shared-memory primitives appropriately to facilitate run-time optimizations. One major advantage of such a separation is that irregular computations can be parallelized at run-time easily. Note that access patterns of irregular computations cannot be analyzed accurately at compile time, and run-time information is required in order to parallelize the code. For irregular computations, compilers and the shared-memory system need to coordinate closely in order to achieve the best performance.

6 Conclusions

In this paper we have introduced an object-based DSM called Adsmith. Issues involved in its design and our solutions are described. For portability and cost considerations, we have implemented Adsmith completely in software as a user-level library on top of PVM. Efficiency of Adsmith is achieved by incorporating many traffic reduction and latency hiding techniques that can be invoked by users according to application characteristics. Several such techniques, e.g., bulk transfer, prefetching, object-based multiple writer, active accesses, are discussed in the paper and have been shown to be effective in improving program performance. Preliminary experimental results also show that programs developed with Adsmith can run as efficient as those with PVM.

Adsmith is an on-going project and is now being extended at several directions. Large applications, including both scientific and non-scientific programs, are now under development to further study the performance of Adsmith. Scalability of Adsmith will be investigated when larger experiment environment is available. Also, we are combining Adsmith with other portable thread libraries such as pthread [21] and LPVM [29]. With thread supports, communication can be further overlapped with computation and parallel I/O can be supported more efficiently. Another extension direction is to provide a class library or a compiler [28] to permit programmers almost transparent manipulation of shared objects. In this way, Adsmith is more user-friendly and easy to use.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *IEEE Computer*, vol. 19, no. 8, pp. 26–34, Aug 1986.

[2] D.F. Bacon, S.L. Graham, and O.J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, Dec 1994.

[3] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, "Orca: a language for parallel programming of distributed systems," IEEE Trans. Software Engineering, vol. 18, no. 3, pp. 190–205, Mar 1992.

[4] A. Beguelin, J. J. Dongarra, G.A. Geist, R. Manchek, and V.S. Sunderam, "A Users' Guide to PVM Parallel Virtual Machine," Oak Ridge National Laboratory, ORNL/TM-12187, September, 1994

[5] J.B. Carter, J.K. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," Proc. *ACM Symp. Operating Systems Principles*, pp. 152–164, Oct 1991.

[6] J.B. Carter, J.K. Bennett, and W. Zwaenepoel, "Techniques for reducing consistency-related information in distributed shared memory systems," *ACM Trans. Computer Systems*, vol. 13, no. 3, pp. 205–244, Aug 1995.

[7] T.C. Chiueh and C. Venkatramani, "Design, implementation, and evaluation of a software-based real-time ethernet protocol," Proc. *ACM SIGCOMM '95*, pp. 27–37, 1995.

[8] T.C. Chiueh and M. Verma, "A compiler-directed distributed shared memory system," Proc. *9th ACM Int'l Conf. Supercomputing*, pp. 77–86, 1995.

[9] G.A. Geist and V.S. Sunderam, "Network Based Concurrent Computing on the PVM System," *Journal of Concurrency: Practice and Experience*, vol. 4, no. 4, pp. 293–311, June 1992.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessor," Proc. *17th Ann. Int'l Symp. Computer Architecture*, pp. 15–26, May 1990.

[11] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory," In *Proceedings of the 15th Symp. Operating Systems Principles,* December 1995.

[12] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Treadmarks: distributed shared memory on standard workstations and operating systems," Proc. *94' Winter USENIX Conf.*, pp. 115–131, Jan 1994.

[13] P. Keleher, A.L. Cox, and W. Zwaenepoel, "Lazy consistency for software distributed shared memory," Proc. *19th Ann. Int'l Symp. Computer Architecture*, pp. 13–21, May 1992.

[14] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The Standford Flash multiprocessor," Proc. *21th Ann. Int'l Symp. Computer Architecture*, pp. 302–313, Apr 1994.

[15] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Standford DASH multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, Mar 1992.

[16] D.E. Lenoski and W.-D. Weber, *Scalable Shared-Memory Multiprocessing*, Morgan Kaufmann Pub., 1995.

[17] K. Li, "IVY: a shared virtual memory system for parallel computing," Proc. *Int'l Conf. Parallel Processing*, pp. 94–101, 1988.

[18] W.Y. Liang, C.T. King, and F. Lai, "Adsmith User Interface," Technical Report, ADSMTR-961128, Architecture Laboratory, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, Nov 1996. (*http://archi1.ee.ntu.edu.tw/∼wyliang/ad-smith*)

[19] W.Y. Liang, C.T. King, and F. Lai, "The acknowledge-free coherence scheme for release consistency memory model," Technical Report, Architecture Laboratory, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, Jun 1997.

[20] D.B. Loveman, "High performance fortran," *IEEE Parallel and Distributed Technology: System and Applications*, pp. 25–42, Feb 1993.

[21] F. Mueller, "A library implementation of POSIX threads under UNIX," Proc. *93' Winter USENIX Conf.*, pp. 25–41, Jan 1993.

[22] 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.

[23] S. Pakin, V. Karamcheti, and A. Chien, "Fast Messages: Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors," *IEEE Parallel and Distributed Technology*, 1997 (To appear).

[24] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: concepts and systems," *IEEE Parallel and Distributed Technology*, pp. 63–79, Summer 1996.

[25] S. Reinhardt, J. Larus, and D. Wood, "Tempest and Typhoon: user-level shared memory," Proc. *21th Ann. Int'l Symp. Computer Architecture*, pp. 325–336, Apr 1994.

[26] J. Rothnie, "KSR-1 memory system," Technical Report, Kendall Square Research, Cambridge, MA, 1991.

[27] D. Scales and M. Lam, "Design and evaluation of a shared object system for distributed memory machine," Proc. *1st USENIX Symp. Operating Systems Design and Implementation*, pp. 101–114, 1994.

[28] R.J. Shang, W.Y. Liang, J.C. Lin, and F. Lai, "A compiler supporting distributed shared memory," submitted to 1997 Workshop on Distributed Technologies and Applications, May 1997.

[29] H. Zhou and A. Geist, "LPVM: a step towards multithread PVM," *Journal of Parallel and Distributed Computing*, submitted special issue on Multithreading for Multiprocessors, July 1995

[30] S. Zhou, M. Stumm, and T. McInerney, "Extending distributed shared memory to heterogeneous environments," Proc. *Int'l Conf. Distributed Computing Systems*, pp. 30–37, 1990.