A DSM-based portable distributed programming testbed

R.K. Hariram

T.A. Gonsalves

Department of Computer Science and Engineering

Indian Institute of Technology, Madras, India.

E-mail address : tag@iitm.ernet.in

Abstract

For interprocess communication, Shared Memory is a simpler and more transparent programming paradigm than Message Passing. Distributed Shared Memory (DSM) is an abstraction of shared memory on distributed computers having no physically shared memory. This paper presents the design and implementation of a completely user-level DSM system on top of Unix. Our testbed uses only System V shared memory IPC, Berkeley sockets and Unix's signal handling facility. The developed testbed is highly portable because all these facilities are now supported by almost all Unix systems. As the testbed is fully user-level it can be used for testing various DSM algorithms. Currently, it supports two consistency algorithms. We have also implemented distributed semaphores and distributed barriers for synchronization of distributed processes. We evaluate our testbed using a set of applications taken from different fields including applications from the SPLASH (Stanford Parallel Applications for Shared Memory) suite. These experiments show close to linear speedup in many cases.

1 INTRODUCTION

Recent developments in microprocessor technology and experiences in computer networks have increased the interest in distributed computers or loosely-coupled multiprocessors. Distributed computers provide good scalability, reliability, fault-tolerance and cost-performance ratio. However, programming on distributed computers had traditionally been using message passing technique as distributed computers do not share memory physically. Programming with shared memory, which is naturally supported by tightly-coupled multiprocessors, provides a better interface than message passing. It simplifies sharing complex data structures among parallel processes. Hence, recent researches are oriented towards supporting shared memory paradigm on distributed computers. An abstraction of supporting the notion of shared memory on distributed computers is referred

to as Distributed Shared Memory (DSM) [Stumm (1990), Nitzberg (1990)].

One of the approaches in supporting DSM is Shared Virtual Memory [Li (1986), Fleisch (1989), Zhou (1992)], where a virtual address space is shared among all processors in the distributed computers. DSM address space can be structured either into pages of the size of physical memory pages or into segments of user-defined size. Our DSM testbed follows the latter scheme which is called "Paged Segmentation". Other approaches base sharing not on pages but on more software-oriented approaches like tuple spaces [Carriero (1989)] or objects [Ramachandran (1991), Tanenbaum (1992), Carter (1991)].

Basic issues in the design of DSM are granularity, heterogeneity [Zhou (1992)], scalability, recoverability, level of implementation, consistency and synchronization. They are addressed in [Nitzberg (1990)]. Consistency of DSM regulates the access of shared data so as to maintain the correctness of execution. Strict consistency requires that a read should always return the most recent write. Sequential consistency [Lamport (1979)] is slightly weaker but most widely assumed. It requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. Recently, weaker consistencies like Processor Consistency [Bisiani (1990)], Weak Consistency [Dubois (1986)] and Release Consistency [Carter (1991)] have been analyzed. They impose less restriction on data accesses and hence exploit more parallelism. However, they complicate programming. Our testbed implements Sequential Consistency.

Shared Virtual Memory can be supported either at the hardware level [Bisiani (1990)], kernel-level [Fleisch (1989)] or at user-level [Zhou (1992), Li (1986)]. User-level implementations, even though slightly less efficient than their kernel counterparts, make the system highly portable. Also, experimentation can be carried out without rebooting the system and hence such systems are suited for testing multiple protocols. So, we chose user-level for our implementation. Many of the user-level implementations reported so far either use distributed kernels of Unix or require some changes to kernel [Zhou (1992), Li (1986)]. However, we design and implement DSM on Unix completely at the user-level without any changes to kernel. Our testbed uses System V Shared Memory IPC, Unix signal handlers and Berkeley's UDP/IP sockets, which are now available on almost all Unix systems.

We also implement distributed semaphores and barriers at user-level which can be used to synchronize the distributed processes. The programming interface supported is similar to that of System V IPC. This simplifies porting of various parallel applications written for shared memory multiprocessors onto our system.

Our testbed currently runs on SunOS and Linux, a public domain Unix OS. We have conducted experiments on a network of Sun workstations connected by 10 Mb/s Ethernet.

Here is the road map to the rest of this paper. Section 2 describes various design aspects of our DSM testbed and Distributed Semaphores. Section 3 explains the implementation details. In section 4, we present the performance of our testbed. A survey of related work is in section 5. Section 6 concludes this paper with a look to the future.

2 DESIGN

Our testbed consists of three independent functional subsystems: Distributed Shared Memory (DSM), Distributed Semaphores (*DSem*) and Distributed Barriers (*DBar*). All three use a *Naming Subsystem* to manage object names globally. In this section, we present the design of these subsystems. As the design and implementation of *DBar* is very similar to that of *DSem*, we will discuss only *DSem* in detail.

2.1 The Naming subsystem

Naming or binding high level names to communication objects is used by all IPC systems. Our naming scheme follows that of System V IPC. It uses two types of names: keys and handles. A key is a 32-bit integer that the user associates with the object (DSM segment or distributed semaphore). When the user creates or queries an object using a key, a handle is returned. This handle is subsequently used to operate on the object.

The separation of the naming subsystem from the other functional components is a natural one. Creating and deleting an IPC object is considerably less frequent than the functional operations on that IPC object. So, we implement the naming component in a centralized, yet reliable, manner. On one of the sites, a Name Server daemon is run. As there is no dependence between the names of DSM segments and distributed semaphores, our testbed uses different Name Servers, one for each. These daemons can be located on different sites so as to distribute the naming load. The Name Server daemon maintains a name database keeping all keys and handles. In addition, it also records the address of the first creator of the object as it will be useful to locate the server for that object. The DSM and *DSem* subsystems send requests to their respective Name Servers for inserting, querying and deleting the names of their objects.

2.2 The DSM subsystem

2.2.1 DSM algorithms

Consistency of DSM is achieved by a combination or variation of one or more of the following four basic algorithms [Stumm (1990)].

Central Server Algorithm The entire DSM data is held in a single server. Every access to DSM data by a non-server causes a request to be sent to the central server.

Migration Algorithm In this *Single Reader Single Writer* algorithm, DSM is divided into fixed-sized pages. Each page resides on the machine which is currently using it. When a fault occurs on any location within a page, the entire page is transferred from the current holder of the page to the faulting processor. After that, all accesses to that page by the new holder become local, until the page is taken away by a fault

on another processor. This algorithm makes use of locality of accesses to minimize network traffic.

Read Replication Algorithm This *Multiple Readers Single Writer* algorithm allows either multiple readers or a single writer to a page. A read access to a page just replicates the page whereas a write invalidates all other copies leaving only one writable copy.

Full Replication Algorithm This *Multiple Readers Multiple Writers* algorithm allows every processor to read and write into a page. A read is done directly from the local copy whereas a write is sent to a *sequencer* which then broadcasts it to all processors.

No single algorithm performs well under all conditions. Performance depends on the application's read/write characteristics. A queuing network model to analyse these algorithms is presented in [Venkatesulu (1994)].

Our DSM testbed currently implements Read Replication and Migration. The user can choose one of these by passing proper flags to the *dsmat()* call. He can also dynamically change the consistency algorithm by reissuing the *dsmat()* with proper flags. The user can choose Migration if the data is migratory. Data is said to be migratory if a write is going to occur soon after a read. For migratory data, Read Replication will cause read and write faults whereas Migration causes only one fault. Full replication and centralized server algorithms are not attractive as they require every write of the user process to be intercepted which is difficult to achieve and very costly in any user-level implementation.

In Read Replication and Migration algorithms, for every DSM page, one process is termed as owner. In Migration algorithm, owner is a node that has an exclusive read and write access to the page. In Read Replication algorithm, owner is either a reader or the only writer to the page. The owner maintains the *copy set*, the list of readers to that page. A process that wants to write to a page acquires the ownership first and then writes. Thus the owner of a page changes dynamically. Owner of a page can be located in either a centralized or fixed distributed or dynamic manner [Li (1986)]. Our testbed adopts the dynamic approach. In this approach, each process keeps track of the ownership of all pages in a local table. The ownership field in the table points either to itself, if it is the true owner of the page, or to another processor which may be the true owner or in turn point to a sequence of processors through which the true owner can be found.

2.2.2 OS requirements

In order to implement Shared Virtual Memory on top of an existing operating system, we need the ability to (a) install page fault handlers for DSM, (b) change the access protection of individual pages, and (c) send messages from one processor to another.

We designed our testbed for systems that meet all these requirements at user-level. In Unix, the signal SIGSEGV (Segmentation Violation) is issued to a process whenever it tries to access an unmapped memory location or when it attempts to write to a location mapped only for read. The signal handler for SIGSEGV is given the address of fault as a parameter.* Such a signal handler acts as the page fault handler in our system.

Even though Unix does not allow a user-level process to change its page access protections, this could be done in two indirect ways. One method is to let the process issue memory map, unmap and protect calls to the virtual page that corresponds to the DSM page. The other way is to let the process issue System V Shared Memory IPC calls to a segment that corresponds to the DSM page. We have chosen the latter approach as System V IPC is widely used. In this indirect approach, every DSM page is mapped to one local shared memory segment of the same size. If a process has to invalidate its access to a DSM page, it issues *shmdt()* system call to the shared memory segment corresponding to that page. Similarly to change access privilege from/to read-write to/from read-only, it issues *shmdt()* followed by *shmat()* with the required permission flags set.

In order that the DSM processes communicate with each other for servicing page faults, we implement our own communication layer using Berkeley's UDP/IP sockets. This layer adds reliability to unreliable UDP/IP socket communication and provides request-response mode of communication with forwarding capability.

2.2.3 DSM architecture

Figure 1 shows the organization of DSM implementation on a site. Each dotted box denotes a process. Descriptions within a dotted box denote modules. On every site, users can create any number of processes that belong to either the same distributed application or different applications. The user program is linked to a library which provides our DSM subsystem. This user process is hereafter referred to as the *DSM process*. The DSM library contains the following major modules:

Page Fault Handler To handle local faults by sending requests to the the page owner.

Segment Interface Handler To handle user's call to create, delete, attach or detach DSM segments by communicating with the Name Server and Segment Manager.

Memory Controller To degrade or invalidate the process's access privilege to a page, on a remote fault.

Page Server and Segment Manager are the two server daemons that run on each node using DSM. The Page Server maintains the copy sets for pages that are owned by local DSM processes. Any fault requests from remote processes are serviced by the page server

*Not all Unix systems provide this address to the signal handler.

Figure 1 Organization of the DSM subsystem

with or without the help of local owner process. We have designed the functions of the Page Server and its co-ordination with the DSM process such that the Page Server takes up time-consuming jobs with little or no intervention of the DSM process. Having a Page Server daemon gives us the following advantages.

- After a user process has finished its job and exits, the pages owned by it need not be flushed out to any other site, as all remote requests are directed only to Page Server and it can still service them.
- Some of the remote faults can be serviced by the page server without disturbing the DSM process. For example, if a read fault occurs at a remote DSM process and the local owner process is a reader of that page, the page server replies with a copy of the local page and updates the copy set maintained by it.
- Some remote faults can be serviced by the page server with less disturbance to the DSM process. For example, if the local DSM process is owner of a page with write permission and a remote read fault request arrives to the Page Server, the DSM process is just requested to reduce its access permission to that page to read-only and the remaining task of sending entire page is carried out by the Page Server.

The page ownership information is stored in a locally shared segment shared by DSM

processes and Page Server. So, a DSM process can communicate with page owners directly without the help of the local Page Server thereby avoiding an otherwise context switch.

The Segment Manager maintains the status of all segments that are created by local DSM processes. Even though pages and their ownership may move from site to site, the segment management always remains with the Segment Manager on the site on which the segment was created.

2.3 The *DSem* subsystem

Distributed semaphores can be implemented either on top of DSM or on top of messages independent of DSM. In the first approach, the data structures for distributed semaphores are stored in DSM space. Consistency of DSM does not itself ensure atomicity as a page may be pre-empted by a remote fault at any time during operations on distributed semaphores residing in that page. Atomicity can be ensured in two ways:

1. By an option to wire-in DSM pages to a process until they are released. DSM should delay servicing of remote faults until the local process releases such pages.
2. By using a lock in DSM space, the processes can use atomic test-and-set instruction on that lock before entering into operations on distributed semaphore. This approach will cause large network traffic due to thrashing of the DSM page containing the lock, when many processes test the lock simultaneously.

Because both the ways of ensuring atomicity are not satisfactory, we implement *DSem* on messages. In this approach [Tanenbaum (1987), Yuan (1992)], the data structures for the semaphores are kept in private address space and atomicity is ensured by queuing the requests. As in DSM, semaphores can also be replicated, migrated or centralized. We observe that semaphore operations exhibit poor locality compared to operations on DSM. So, we take a simple way of distributing semaphores in a fixed manner.

2.3.1 *DSem* architecture

The user program is linked to a library that provides our *DSem* subsystem. The *DSem* module accepts requests to create, delete and operate on a distributed semaphore. It handles create and delete requests by sending request messages to the *DSem* Name Server. To service requests to execute *up* or *down* on a semaphore, it locates the Semaphore Server owning that semaphore using a table and then communicates with that Semaphore Server.

On every site there is a Semaphore Server daemon. It owns all semaphores that are created by local processes. The data structures of the distributed semaphores owned by this server are stored in its private address space. Since semaphores do not migrate, their distribution among different sites is determined by the pattern of semaphore creation by the user processes.

2.4 Programming model

The programming semantics supported by our DSM system are similar to those of System V shared memory IPC [Bach (1986)]. It provides *dsmget()*, *dsmat()*, *dsmdt()* and *dsmctl()* calls to create, attach, detach and remove a segment respectively. In addition to the above four IPC calls, as our testbed is at the user-level it also provides routines *InitDsm()* and *ExitDsm()* for initializing and finishing up operations. Similar set of calls are supported in case of *DSem* and *DBar* subsystems.

3 IMPLEMENTATION

3.1 Table handling

The modules of the DSM subsystem linked to the user program maintain the status of segments used by the user program in a table called *SegTable*, shown in Figure 2.(a). Each entry in *SegTable* contains information about one segment, consisting of one or more pages. The table is hashed by the segment id. Any page within a segment is identified by the tuple <segmentId, offset>, where the *offset* is in number of pages within the segment.

Each valid entry of *SegTable* has segment identifier, segment size, flags for type of attaching, address of segment manager, a pointer to a list of attached addresses and a pointer to a page table. Each entry in the page table contains the id of the System V shared memory segment corresponding to that page and the permission with which this System V segment is currently attached to the process.

The segment table of the Page Server contains entries with fields segment id, size and a pointer to page entries. These page entries contain more information including the copy set and a pointer to the queue of processes waiting for the page. The segment table of the Segment Manager is very simple and contains only segment id, segment status and the list of processes attached to that segment.

As both the Page Server and DSM processes need the ownership information of the DSM pages, it is kept in a locally shared space, *shTable* (Figure 2).(b). This table has two parts: a pointer array and an ownership table. Each entry in the pointer array corresponds to one DSM segment and points to the ownership table entries for that segment. Each entry in the ownership table corresponds to one DSM page. All the entries of one segment are linked together. All the free entries are linked into a free list. Each entry contains a lock to control access to that entry, a pointer to the list of processes waiting for that lock, access permission of the page owner, address of the owner and links to the next page entry. Using *shTable*, given <segmentId, offset>, the owner of the page can be found.

Figure 2 Structure of Tables

3.2 Page fault handling

Local faults

When a page fault occurs the Page Fault Handler module of the DSM process checks whether the faulting address lies in attached DSM address space. If so, it maps the address to the $\langle \text{segmentId}, \text{offset} \rangle$ tuple that identifies the DSM page in which the fault address lies. The fault handler then finds the address of the owner of the page from the shared ownership table and sends a request to the Page Server of the owner's node. In case of a Read Fault, it receives the page as reply and then returns. In case of a Write Fault, it receives the page and the copy set. It then sends invalidation messages to all processes listed in the copy set, receives acknowledgements, updates the shared ownership table by setting the ownership of the page to itself and then returns. Figure 3 illustrates handling of Read Fault and Write Fault.

When a process writes to a read-only page, it generates an Upgrade Fault, sends an upgrade request to the Page Server. If the upgrading is successful, the reply contains copy set alone. If the upgrading fails, the reply contains copy set and the page. As in case of Write Fault, it sends invalidation messages to all readers in the copy set. Support of upgrading avoids transfer of a page when the reader already has a valid copy. In order to check whether the copy of the page held by the reader is valid, a version number is used for each page. Whenever an upgrade or write fault occurs, the version number is incremented. On a read fault, the version number is also received along with the page.

Remote faults

When Page Server receives a Read Fault or Write Fault or Upgrade Fault request for a DSM page $\langle \text{segmentId}, \text{offset} \rangle$ it checks the entry in the ownership table to see whether the owner of the page is a local client. If not, it forwards the request to the owner listed in the ownership table. If the owner is local, the Page Server maps $\langle \text{segmentId}, \text{offset} \rangle$ to

Figure 3 Page Fault Handling

an address at which the Page Server has attached the System V shared memory segment corresponding to that DSM page. In case of a Read Fault, it sends degradation message to the local owner process and all other local processes that have write access to that page, adds the requestor to the copy set of that page and replies with the page. In case of a Write Fault, it sends invalidation message to all local writers and readers using that page. Only the writers reply to this invalidation message. After receiving reply from all local writers, the Page Server replies to the remote faulter with the page and copy set. In case of an Upgrade Fault, it invalidates local clients as in Write Fault servicing. It compares the version number of the page held by the requestor to the current version number. If both are same, it replies with copy set alone. If both are different, it replies with copy set and the page.

When the Page Server receives an invalidation request, it checks whether the DSM process is alive and if so, it forwards the request to that process and sends an acknowledgement to the requestor. When a DSM process receives an invalidation request or degradation request for a page, its Memory Controller module detaches or reattaches the System V segment corresponding to that page with new permissions.

3.3 Communication substrate

The implementation of DSM, *DSem* and Naming requires a reliable communication protocol. They typically operate in request-response mode. For example, on a page fault, the fault handler sends a request to the probable owner of the page which either replies with the page or forwards the request to another host. The most suitable protocol for these remote operations is therefore a request-response protocol, with forwarding capability.

While Sun's RPC package supports request-response mode it does not provide forwarding and would reduce portability. Hence we selected the more widely available Berkeley sockets. TCP/IP sockets are unsuited as the pattern of communication between DSM processes varies dynamically according to the access to the DSM data and a process can communicate with multiple processes. So, it is expensive to keep multiple connections

alive or to open and close a single connection to suit the pattern of communication. Hence, we opted for the UDP/IP connectionless protocol. Unfortunately, UDP/IP does not provide reliable communication. Hence, our protocol implements timeout and retransmit mechanism to ensure reliability.

The communication layer provides *SendReqRecvReply()*, *SendReply()*, *ForwardReq()* and *SendAck()* calls to the upper layer. *SendAck()* is used by a replier to prevent the requestor from retransmitting if replying may take a long or arbitrary time. The upper layer should provide functions *ServiceReq()* and *ServiceDupReq()* which are called by the communication layer on receiving a new or a duplicate forwarded request respectively.

4 PERFORMANCE EVALUATION

We now examine the performance of our DSM testbed. First we describe a series of experiments for evaluating the costs of important operations. Besides providing insight into our implementation, these can be used as inputs to analytical or simulation models of DSM. Next we discuss the performance on our DSM of several parallel programs that are commonly used to benchmark shared memory multi-processors. Experiments were performed on machines interconnected by a 10 Mb/s Ethernet. We used *gcc* compiler. During experiments, the facility was devoted solely to us.

4.1 Costs of faults

On Sun 3/60+, the average end-to-end delays in handling a read fault (without forwarding the request), a write fault (without invalidations) and an upgrade fault (with reply containing no page) are 36.6 ms, 35.6 ms and 20.6 ms respectively. The cost of sending and receiving a 30 byte fault request packet by UDP is 3.3 ms and of an 8 KB fault reply packet is 18.3 ms. Thus, our implementation adds an overhead of 15 ms, 14 ms and 14 ms for read, write and upgrade faults respectively.

Our times of 35–37 ms/fault compare favourably to the 40 ms/8KB transfer time using SunRPC on a 10 Mb/s Ethernet reported by Van Renesse et al. [Renesse (1988)]. Ramachandran et al. [Ramachandran (1991)] report 43.4 ms/fault in their DSM system, about 16–20% poorer than ours.

The read and write fault costs in the case of Linux, with an i386 node generating the faults and an i486 servicing them, are 26.0 ms and 26.3 ms respectively. The time between sending a 30 byte request packet and receiving 4KB reply packet is 20.2 ms. So, the overheads for read and write fault are 5.8 ms and 6.1 ms respectively.

Figure 4 Performance of Matrix Multiplication

4.2 Performance of applications

We have currently implemented three applications with different shared access behaviours and synchronization requirements on our testbed. The input data size is chosen not to be too large so as to test the performance of our testbed for medium granularity also. All measurements were on Sun 3/50 machines.

4.2.1 Matrix multiplication

Matrix multiplication is inherently parallel with minimal sharing. The result matrix is divided into a number of sets of adjacent rows equal to the number of processes p , and assigned to the processes. A master process initializes the two input matrices, creates one process on each remote machine and then computes its share of the result matrix. A barrier is used to synchronize the termination of all processes. The input matrices are read-shared and the result matrix is write-shared. During computation there will not be any communication between worker processes except for any page containing rows assigned to different processes. This *false sharing* becomes negligible for large matrices.

Figure 4 illustrates the results. For matrices of 250×250 , near linear speedup was observed for upto 4 processors (Figure 4.(a)). As the number of processors increases, the computation assigned to each processor becomes small. Also, the read faults for the input matrices increase (Figure 4.(b)) as each processor needs one of the input matrices fully. So, for $p > 4$ the speedup rises slowly, levels off and then starts to fall for $p > 10$.

4.2.2 Linear equation solver

Linear Equation Solver implements a parallel version of Jacobi's[Bertsekas (1989)] iterative algorithm to solve a set of n linear equations, $Ax = b$, where A is $n \times n$ matrix and

Figure 5 Performance of Linear Equation Solver

x and b are $n \times 1$ vectors. Starting with some initial vector $x(0)$, a new vector $x(t+1)$ is computed in the $(t+1)^{th}$ iteration using,

$$x_i(t+1) = \frac{1}{a_{ii}} \{b_i - \sum_{j \neq i} a_{ij} x_j(t)\}$$

This algorithm is parallelized by dividing the result vector x into a number of sets of adjacent elements equal to the number of processes, p , and assigning the task of evaluating that portions to the processes. Every iteration has two phases: computation phase and update phase. In the computation phase, each process calculates its portion of x into a temporary array. In the update phase, every process copies the new values from the temporary array into the globally shared array x . Processes synchronize at the end of each phase using barriers.

We coded A and b into our program as the number of DSM pages available in our testbed is limited by the number of available Sys V segments. As A and b are only read shared, this does not affect the overall performance of the application appreciably. x is stored in DSM space. The entire vector x is read by all processes in the computation phase and portions of it are written by each process in the update phase.

Graphs in Figure 5 show the speedup and page faults for solving a system of 6000 equations using 5 iterations. We observe linear speedup upto $p = 6$. For $p > 6$, speedup is only marginally below linear.

4.2.3 Water

Our third test program, *Water*, is taken from the SPLASH [Singh (1992)] suite of realistic benchmarks for shared memory multiprocessing systems. *Water* is an N-body molecular dynamics program that evaluates forces and potentials in a system of water molecules

Figure 6 Performance of *Water*

in the liquid state. The computation is performed over a user-specified number of time-steps. Every time-step involves setting up and solving the Newtonian equations of motion for water molecules in a cube with periodic boundary conditions. The total potential is computed as the sum of intra- and intermolecular potentials.

The main data structure is a large array of structures, VAR. Each structure holds all the data for one molecule, about 600 bytes. For intra-molecular computation, parallelism is maintained by assigning every processor a set of molecules that are located next to each other in the VAR array. There is no communication in the intra-molecular computations, except in the small number of accumulations to a global sum every time-step. For intermolecular computations, a processor computes the interactions for every molecule in its partition with the $n/2$ molecules following it in the VAR array. Then a processor communicates with exactly half the total number of processors.

Distributed Barriers are used to maintain dependencies before and after computing intermolecular interactions. Distributed semaphores are used to accumulate the private sum of each process into the shared sum once every time-step and also to make updates to the force locations of all atoms to be mutually exclusive.

The problem size used in our measurements is 288 molecules. Process creation, segment creation and semaphore creation are omitted in time measurements as we simulate only 2 time-steps. The speedup obtained is shown in Figure 6.(a). As the communication involved in intermolecular computations is much larger than that in other applications, the number of faults is substantially higher as seen from Figure 6.(b). Therefore, the speedup is not as linear as in the other two applications. However, reasonable speedup is observed for upto 4 processors.

5 RELATED WORK

A number of software DSM systems have been developed in the recent past. Ivy [Li (1986)] was the first page-based DSM system developed on an Apollo ring using memory map calls to attach and detach pages. Ivy-II is a user-level implementation but requires addition of some system calls to the kernel. It implements consistency using *Single Writer Multiple Readers* protocol. It supports page-based sharing whereas our testbed supports paged-segmentation which is more flexible.

Mirage [Fleisch (1989)] is a kernel-level paged-segmentation DSM system on Locus operating system. It locks a page with a certain processor for a certain time to avoid page thrashing due to false sharing. Mermaid [Zhou (1992)] is a heterogeneous page-based DSM system developed for Sun and Firefly workstations. It is also at user-level, but requires addition of a new system call to SunOS so as to set the page access permissions from user-level.

Ramachandran *et al.* [1991] have implemented an object based DSM system on Clouds, a distributed operating system, and also on Unix. Their Unix implementation uses Sun's Network Interface Tap for communications and Unix file system to store DSM segments whereas our testbed uses Berkeley sockets and System V shared memory IPC for respective needs. Their design involves more context switches in servicing page faults as each request and reply has to go through a special process meant for communication. In our design we use asynchronous sockets to handle asynchronous events. Also, we minimise context switches by using a locally shared table between DSM user processes and Page Server. Hence, our system provides 16-20% better performance.

Munin [Carter (1991)] is an object-based software DSM system. It supports multiple coherency protocols including Release Consistency. The user annotates the shared data with the type of its expected access pattern and these annotations are then used by the run-time system to choose a consistency protocol best suited to that access pattern. A.S.Tanenbaum *et al.* [Renesse (1988)] implement distributed shared objects using Full Replication over a reliable broadcast layer. They have also designed a new parallel programming language, called Orca [Bal (1992)] which supports parallel processes and distributed shared objects.

6 CONCLUSIONS

We have designed and implemented a distributed programming testbed for a network of Unix machines based on the shared memory communications paradigm. In addition to a distributed shared memory interface similar to the Unix System V shared memory, our testbed provides distributed semaphores and barriers. The implementation is entirely at the user-level and is hence portable. The testbed currently runs on SunOS and Linux.

The testbed could be used for evaluating various DSM algorithms. Currently it supports two consistency algorithms, Read Replication and Migration. It also supports Upgrading for write faults onto a read-only page.

The fault handling times of our user-level implementation (35 ms/8 KB page) compare favourably with other implementations. Several realistic applications exhibit near linear speedups for upto 8 processors. However in an application with substantial interprocess communication, the observed speedup is considerably less than linear.

In our testbed currently the user specifies whether Read Replication or Migration algorithm is to be used for a segment. The future work will be to design an automatic switching by the DSM subsystem between these algorithms depending on the shared access behaviour of the application.

REFERENCES

- Bach, M.J. (1986) *The Design of the Unix Operating System*. Prentice-Hall of India, New Delhi.
- Bal, H.E., Kaashoek, M.F. and Tanenbaum, A.S. (1992) "ORCA: A Language for Parallel Programming of Distributed Systems". *IEEE Trans. Soft. Engg.*, 18(3):190-205.
- Bertsekas, D.P. and Tsitsiklis, J.N. (1989) *Parallel and Distributed Computation - Numerical Methods*. Prentice-Hall Int'l Editions.
- Bisiani, R. and Ravishankar, M. (1990) "PLUS: A Distributed Shared Memory System". In 17th *Annual Int'l Symp. Comp. Arch.*.
- Carriero, N. and Gelernter, D. (1989) "How to Write Parallel Programs: A Guide to the Perplexed". *ACM Computing Surveys*, 21(3):323-357.
- Carter, J.B., Bennet, J.K., and Zwaenepoel, W. (1991) "Implementation and Performance of Munin". In *Proc. of 13th ACM Symp. Op. Sys. Principles*, pp. 152-164.
- Dubois, M., Scheurich, C. and Briggs, F.A. (1986) "Memory Access Buffering in Multiprocessors". In 13th *Annual Int'l Symp. Comp. Arch.*, pp. 434-442.
- Fleisch, B.D. (1989) *Distributed Shared Memory in a Loosely Coupled Environment*. PhD thesis, University of California, Los Angeles.
- Lamport, L. (1979) "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs". *IEEE Trans. Comp.*, C-28(9):690-691.
- Li, K. (1986) *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Department of Computer Science, YALE University.
- Nitzberg, B. and Lo, V. (1990) "Distributed Shared Memory: A Survey of Issues and Algorithms". *IEEE Comp.*, 24(8):52-60.
- Ramachandran, U. and Khalidi, M.Y.A. (1991) "An Implementation of Distributed Shared Memory". *Software-Practice and Experience*, 21(5):443-464.

- Renesse, R.V., Staveren, H.V. and Tanenbaum, A.S. (1988) "Performance of the World's Fastest Distributed Operating System". *Operating Systems Review*, 22(4):25-34.
- Singh, J.P., Weber, W.D. and Gupta, A. (1992) "SPLASH: Stanford Parallel Applications for Shared Memory". *Comp. Arch. News*, 20(1):5-44.
- Stumm, M. and Zhou, S. (1990) "Algorithms Implementing Distributed Shared Memory". *IEEE Comp.*, 23(5):54-64.
- Tanenbaum, A.S. (1987) *Operating Systems: Design and Implementation*. Prentice-Hall of India, New Delhi.
- Tanenbaum, A.S., Kaashoek, M.F. and Bal, H.E. (1992) "Parallel Programming using Shared Objects and Broadcasting". *IEEE Comp.*, pp. 10-19.
- Venkatesulu, D. and Gonsalves, T.A. (1994) "A Queuing Network Model of Distributed Shared Memory". In *Proc. of NETWORKS '94 - Int'l Conf. Computer Networks, Architecture and Applications*, Madras, India.
- Yuan et al, S.M. (1992) "Design and Implementation of a Distributed Semaphore Facility". In *Third Workshop on Future Trends on Distributed Comp. Sys.*, pp. 180-184.
- Zhou, S., Stumm, M., Li, K. and Wortman, D. (1992) "Heterogeneous Distributed Shared Memory". *IEEE Trans. Parallel and Distributed Sys.*, 3(5):540-554.

R.K. Hariram received a B.E. (Computer Science) from Government College of Engineering, Tirunelveli, Tamil Nadu, India in 1992. He was the University top rank holder in B.E. (Computer Science). He is currently an M.S. research scholar in the Department of Computer Science and Engineering, I.I.T., Madras. His research interests include distributed systems, parallel processing and computer networks.

Timothy A. Gonsalves received a B.Tech.(Electronics) from I.I.T., Madras in 1976, an M.S. (EE) from Rice University, Houston, USA in 1978, and a Ph.D.(EE) from Stanford University, USA in 1986.

From 1986 to 1988 he was an Assistant Professor in the Computer Science Department, Worcester Polytechnic Institute, USA. In January 1989, he joined the Department of Computer Science and Engineering, I.I.T., Madras where he is currently an Associate Professor. His research interests include distributed computer systems, computer networks, and performance evaluation. He is active in technology/product development and consultancy for a number of industries. Dr.Gonsalves is a member of CSI, ACM and IEEE.