

A comprehensive distributed shared memory system that is easy to use and program

This article has been downloaded from IOPscience. Please scroll down to see the full text article.

1999 Distrib. Syst. Engng. 6 121

(<http://iopscience.iop.org/0967-1846/6/4/301>)

View [the table of contents for this issue](#), or go to the [journal homepage](#) for more

Download details:

IP Address: 38.107.179.211

The article was downloaded on 20/02/2012 at 21:01

Please note that [terms and conditions apply](#).

A comprehensive distributed shared memory system that is easy to use and program

J Silcock and A Goscinski

School of Computing and Mathematics, Deakin University, Geelong, Victoria 3217, Australia

E-mail: jackie@deakin.edu.au and ang@deakin.edu.au

Received 6 November 1998

Abstract. An analysis of the distributed shared memory (DSM) work carried out by other researchers shows that it has been able to improve the performance of applications, at the expense of ease of programming and use. Many implementations require application programmers to write code to explicitly associate shared variables with synchronization variables or to label the variables according to their access patterns. Programmers are required to explicitly initialize parallel applications and, in particular, to create DSM parallel processes on a number of workstations in the cluster of workstations. The aim of this research has been to improve the ease of programming and use of a DSM system while not compromising its performance. RHODOS' DSM allows programmers to write shared memory code exploiting their sequential programming skills without the need to learn the DSM concepts. The placement of DSM within the operating system allows the DSM environment to be automatically initialized and transparent. The results of running two applications demonstrate that our DSM, despite paying attention to ease of programming and use, achieves high performance.

1. Introduction

Programmers would describe distributed shared memory (DSM) programming as easy if they did not have to learn a new language, learn the semantics of a DSM system, declare shared data in a DSM-driven manner or use unfamiliar synchronization primitives.

A DSM parallel application is made up of two or more processes running on separate workstations which must be initialized on them. In existing DSM systems, programmers are required to decide on the number of workstations that should be used to execute a parallel application and also the physical workstations that should be used. This is a significant load to place upon programmers. Further, this manual approach can lead to load imbalance, which worsens when more parallel applications are executed on a cluster of workstations (COW) and leads to performance degradation.

Since the DSM processes may share memory, sharable objects must be created and associated with parallel processes. Synchronization constructs must be initialized in order to access sharable objects correctly. Barriers, which are used as the coordination mechanism for parallel processes, must also be initialized. These initialization operations are carried out manually by programmers.

An analysis of the existing DSM systems shows that ease of programming and use for application programmers are sacrificed for the sake of performance. In DSM research the major thrust has been to improve the efficiency of the

systems by implementing weaker-consistency models [1].

A new approach is needed. Our aim has been to develop a DSM system that is easy to use and program and which is transparent at user level, while maintaining the same high level of efficiency achieved by other projects. The need for this integrated approach was also raised in [1].

2. DSM in RHODOS

A DSM system has been developed within the RHODOS system. RHODOS is a microkernel and client–server-based distributed operating system [2]. In RHODOS, the system resources are managed by a set of servers such as the Process, Space (Memory), and Interprocess Communication (IPC) Manager. Since shared memory can be viewed as a resource which requires management, the DSM system has been integrated into the Space Manager (figure 1).

By placing the DSM system in the Space Manager of RHODOS we have been able to meet several design requirements. Firstly, programmers are able to use the shared memory as though it were physically shared, hence, the ease of programming requirement is met. Secondly, because the DSM system is in the operating system itself and is able to use the low-level operating system functions the transparency and efficiency requirements can be met.

The DSM system employs release consistency using the *write-update* model [3]. Synchronization of DSM

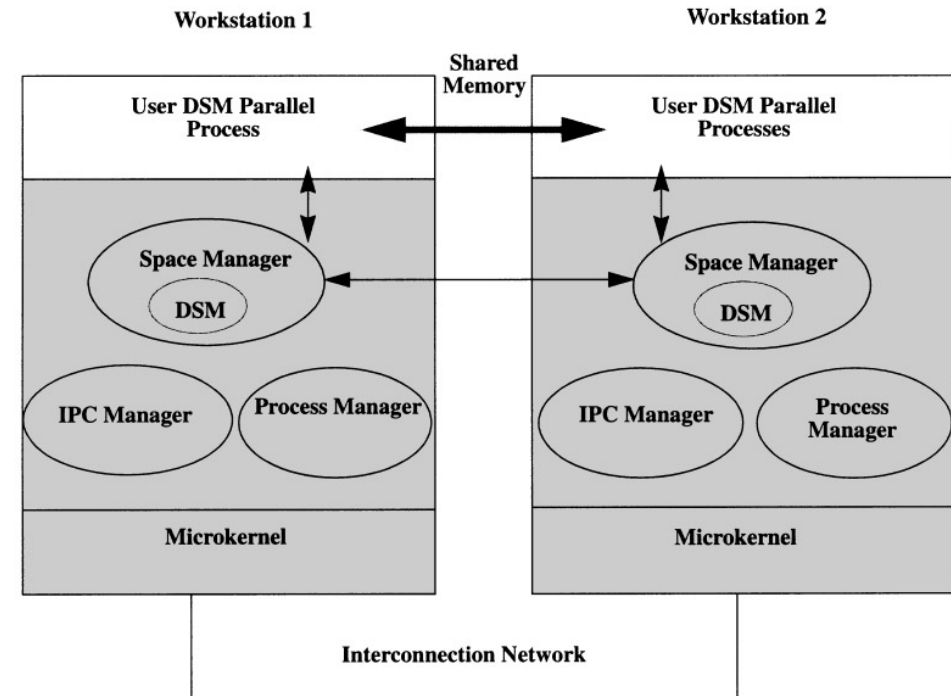


Figure 1. DSM system integrated into RHODOS.

processes sharing memory takes the form of a semaphore-type synchronization for mutual exclusion. Barriers are used to coordinate executing processes.

3. Programming aspects of RHODOS DSM applications

Here, we present the semantics of initialization in RHODOS' DSM to show that the primitives used by programmers evoke and perform operations which are completely invisible to them, and that programmers are not forced to go beyond their knowledge of concurrent programming.

Figures 2 and 3 show the codes of the Travelling Salesman problem (TSP) and Jacobi iteration, respectively, showing how RHODOS' DSM supports programmers by allowing them to initialize parallel processes and shared data, and program using the same shared memory code that they would use when programming for a physically shared memory system.

The *start_dsm()* or *dsm_parstart()* functions return the address of the block of DSM memory allocated during the calls. These primitives allow programmers to invoke process initialization of both parent and child processes. Programmers indicate the preferred number of processes to execute the application. The actual number allocated by the operating system may differ from this number. Programmers are required to declare the semaphore (semaphore-based mutual exclusion was selected as the synchronization method) and barrier (used as the coordination mechanism) variables. Furthermore, the code shows the simple primitives required for the initialization of the parallel processes.

Figures 2 and 3 show that programmers do not have to use any knowledge of DSM. They can solely concentrate on the application; the only requirement is the placement of the

initialization primitives and barriers which practically results from parallelization of the application.

4. Automatic initialization of the RHODOS DSM system

Parallel execution of a program means that there is a sequential process, which at one stage of execution forms a set of parallel processes (children of the parent process) [4]. These processes must be initialized on selected workstations. The processes may share memory which means that sharable objects must be created and associated with parallel processes. Furthermore, semaphores and barriers, in RHODOS, must also be initialized. Here we show that these operations are performed automatically.

When an application using RHODOS' DSM starts to execute, the parent process initializes the DSM system with a single primitive (figures 2 and 3) which initially requests the memory server to allocate a block of memory for the globally shared memory. When the block has been created the memory server attaches it to the parent process.

4.1. DSM space creation

The shared object used for DSM is a space, the logical memory unit in RHODOS. Spaces are made up of one or more pages. Hence, the unit of granularity used in RHODOS DSM is a page. The resource model of a DSM process contains a DSM space as well as stack, data and text spaces [5]. The DSM spaces on all workstations are identical in size and position.

```

typedef struct{
    .....
    }GlobalMemory;          /*Define structure GlobalMemory*/
char child[20];             /*Name of child code*/
SNAME sp_name;             /*Declare space sname variable*/
int num_procs;
int max_num_procs;        /*Maximum number of processes*/
int memory_size;
int numsems, numbarriers; /*Synchronisation variables*/
SNAME barrier[numbarriers], sem[numsems];
GlobalMemory *glob = NULL; /*Initialise GlobalMemory*/
memory_size = sizeof(GlobalMemory);
memory_address = start_dsm() or dsm_parstart() /*Initialisation primitive for parent or
                                                child process*/

glob = (GlobalMemory *)memory_address;

dsm_barrier(barrier[0])    /*Barrier after initialisation*/
while(){
    wait(sem[0])           /*Wait for access to priority queue*/
    if (empty queue)
        signal(sem[0])    /*Signal exit the critical region*/
        exit()
        Add to the queue until the head
        tour looks promising
        Path = Tour from head of queue
        delete head tour
        signal(sem[0])     /*Signal exit the critical region*/
    }
curr_shortest = try all cities not in Path recursively to find the shortest tour length
wait(sem[1])             /*Wait to access curr_shortest*/
    if (length < Shortest_length)
        curr_shortest = length
signal(sem[1])           /*Signal exit the critical region*/
dsm_barrier(barrier[0])  /*Barrier to synchronise exit */

```

Figure 2. Pseudocode for the TSP.

4.2. DSM parallel process initialization

The parent process initializes the DSM system with a single primitive, shown in figure 4.

This is followed by the instantiation of the child processes. These processes can be instantiated explicitly by the user on each workstation. However, we considered this to be an irksome burden for programmers. Thus, these processes are initialized automatically by the operating system, using the code shown in figure 5, that concurrently creates them directly on the selected workstations from a single executable image on disk (*remote_process_create()*) [6].

The decision regarding the workstations on which to create the child processes is important for relieving programmers of the operating system oriented activities, the speed of execution of the application and the functioning of the whole operating system. While the user indicates the maximum number of processes to execute an application, the operating system makes the final decision based on the current system load. The Global Scheduler in RHODOS collects the system load information from all workstations. The Parent's Execution Manager contacts the Global Scheduler which, based on system load information, returns remote workstation names to the Execution Manager, that creates a single child process from an executable image on disk on each of these remote workstations.

4.3. Shared data initialization

The values for the variables in the shared data space have to be initialized before computation can commence. In RHODOS DSM each child process initializes the shared variables in the DSM space independently. This initialization is carried out by the code in the child process which either reads the data from a file or assigns values in the code.

4.4. DSM system initialization semantics

Figure 6 shows the sequence of messages surrounding the automatic initialization of the DSM system to the point where the semaphores and barriers are initialized. Depicted on the source and remote workstations are the parent process and the child process to be created on the remote workstation, and the Execution Manager, Global Scheduler and Space Manager. The DSM space itself is shown attached to the user processes. When a parallel application using DSM starts to execute the parent process initializes the DSM system. Shared data initialization and DSM child process creation are performed in the following steps:

The parent process executes a *start_dsm()* library call (figure 4) which requests (*message 1*) the Space Manager to allocate a memory block for the DSM space.

- The Space Manager creates the space using the *space.create()* call, which creates a space at a specified base address.

```

SNAME psn;                /*process sname */
char  child[20];          /*name of child code*/
SNAME sp_name;            /*declare space sname variable*/
int num_procs;
int max_num_procs;        /*maximum number of processes*/
int memory_size;
int numsems, numbarriers; /*synchronisation variables*/
SNAME barrier[numbarriers], sem[numsems];

memory_size = sizeof(GlobalMemory);

memory_address = start_dsm() or dsm_parstart() /*initialisation primitive for parent or child
                                                process*/

glob = (GlobalMemory *)memory_address;
dsm_barrier(barrier[0])    /*Barrier after initialisation*/
for current = 0 to ITERATIONS{
  for i = 1 to DOWN        /*For matrix size DOWN by ACROSS*/
    for j = 1 to ACROSS
      scratch[i][j] = (glob->grid[i-1][j] + glob->grid[i+1][j] + glob->grid[i][j-1] +
                       glob->grid[i][j+1]) / 4
    dsm_barrier(barrier[0]) /*Barrier before scratch[][] written to grid[][] */
    for i = 1 to DOWN
      for j = 1 to ACROSS
        glob->grid[i][j] = scratch[i][j]
    dsm_barrier(barrier[0]) /*Barrier before next iteration */
}
dsm_barrier(barrier[0])    /*Barrier to synchronise exit*/

```

Figure 3. Basic pseudocode for the Jacobi algorithm.

```

GlobalMemory *glob = NULL; /*Declare a pointer to DSM global memory*/
main(){
  SNAME psn;                /*process sname */
  char  child[20];

  /*DSM variables*/
  SNAME sp_name;            /* declare space sname variable*/
  int num_procs;
  int max_num_procs;        /*maximum number of processes*/
  int memory_size;

  /*synchronisation variables*/
  int numsems, numbarriers;
  SNAME barrier[numbarriers], sem[numsems];

  memory_size = sizeof(GlobalMemory);

  glob = (GlobalMemory *) start_dsm(&sp_name, consistency_model, memory_size,
  child, max_num_procs, sem, barrier, numsems, numbarriers, &num_procs);
  .....
  .....

```

Figure 4. DSM initialization code for the parent process.

- The *start_dsm()* call sends *message 2* to the Execution Manager requesting parallel initialization.
- The Execution Manager requests (*message 3*) the number and location of idle or lightly loaded workstations from the Global Scheduler. The Global Scheduler returns the addresses of these workstations to the Execution Manager.
- The Execution Manager requests (*message 4*) the Execution Managers on specified workstations to create a single child process on each of them. The child processes are created from the executable image on disk.
- The parent process' Execution Manager is informed (*message 5*) when the creation of the child processes is completed.
- The Execution Manager sends *message 6* to the parent process, containing the number and ids of processes successfully created.
- The child processes each execute a *dsm_parstart()* call (figure 5) that causes the process to block waiting for *message 7* from the parent process with the name of the parent process' DSM space.
- The child process passes (*message 8*) control to the Space Manager which sends *message 9* to the parent process' Space Manager requesting the DSM space's details.
- The Space Manager sends (*message 10*) the base address of the space and size for the DSM space and the number of participating processes.
- *Message 11* from the Space Manager passes control back to the child process. When the space information is received the DSM space is created and attached to the child process.

```

GlobalMemory *glob = NULL; /*Declare a pointer to DSM global memory*/
main(){
/* DSM variables*/
SNAME sp_name; /* declare space name variable*/
/*synchronisation variables*/
SNAME barrier[2];
SNAME sem[2];
int num_procs;

uint32_t numsems = 1, numbarriers = 2;

glob =(GlobalMemory *) dsm_parstart(&slaveNum, &sp_name, &num_procs, num-
sems, numbarriers, sem, barrier);

```

Figure 5. DSM initialization code for the child process.

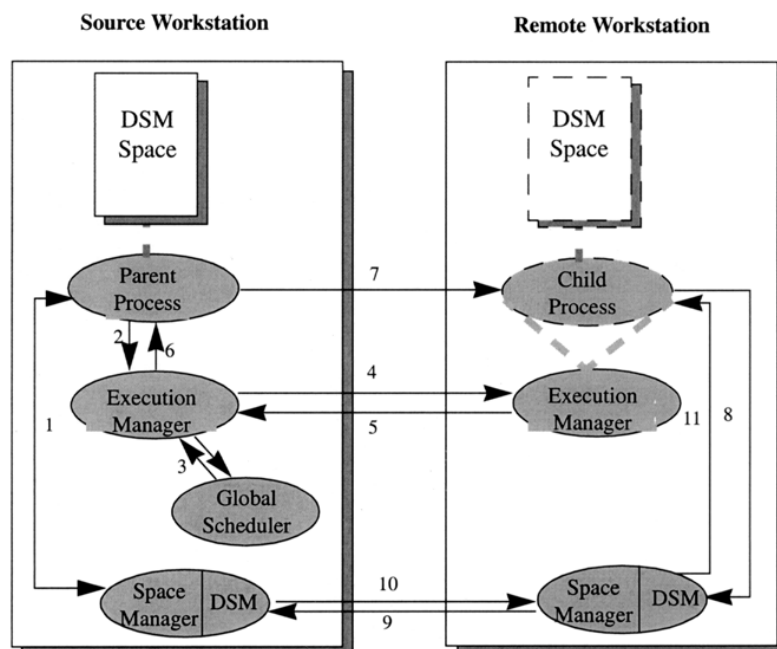


Figure 6. Semantics of automatic initialization on RHODOS.

The followed-up semaphore and barrier initialization is carried out by library calls made from the *start_dsm()* and *dsm_parstart()* functions. The semaphores and barriers are declared as an array of names in the parent and child processes as shown in figures 4 and 5. Programmers pass the number of semaphores and barriers required to the *start_dsm()* and *dsm_parstart()* functions and the *sem* and *barrier* variables are populated during the function calls.

4.5. Initialization of process data

At this stage each of the DSM processes, parent and child processes, initializes its global and local variables. This is achieved by each process assigning values to the shared variables independently either by reading data from a file or by assigning values entered by programmers so that the DSM spaces attached to parent and child processes are all consistent. The DSM processes synchronize at the end of this initialization phase, using *dsm_barrier()*, before starting execution of the application.

5. Performance of test applications

The objective of this section is to demonstrate that the proposed DSM system provides high performance, using the results of our tests which measure the speed-up of two applications. The RHODOS DSM system runs on Sun 3/50 workstations, connected by a 10 Mbps Ethernet. The granularity of the shared memory is an 8 K page. The experiments were carried out using from one to eight workstations.

5.1. The TSP

In the computation of TSP (figure 2) the major bottleneck is access to the priority queue. Processes must wait for the semaphore before accessing the queue itself. This semaphore is not released until new tours have been put back onto the queue.

In order to make the results of the performance study of TSP within the RHODOS DSM environment and Munin and TreadMarks environments comparable we also used an 18-city tour. The speed-up results are shown in figure 7. The speed-up for eight workstations is 6.85.

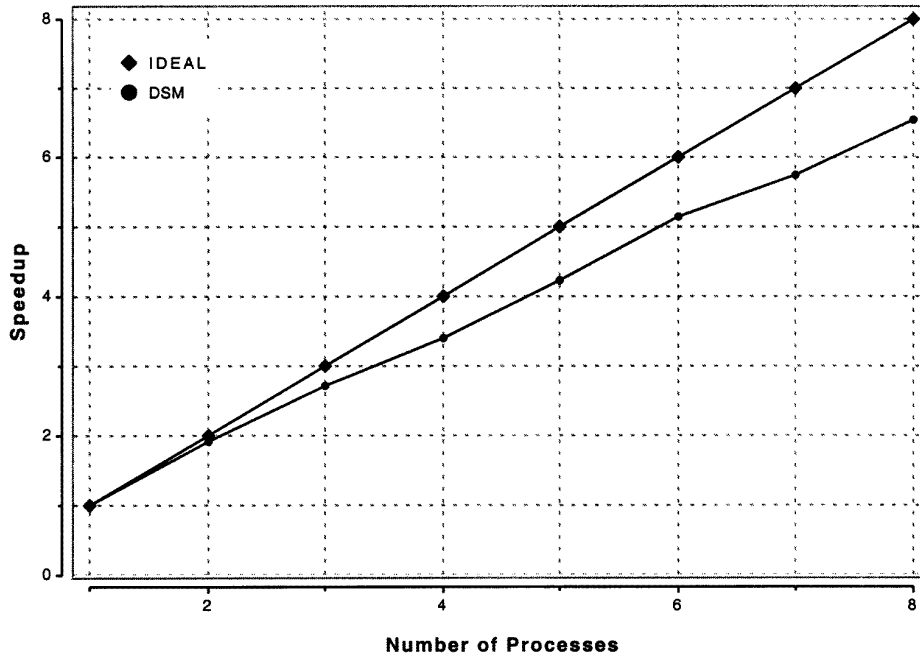


Figure 7. Speed-up for TSP using RHODOS DSM for an 18-city tour.

5.2. The Jacobi algorithm

The Jacobi algorithm (figure 3) is based on a form of successive over-relaxation. Synchronization is supported exclusively through barriers which may make the network become a bottleneck with a large number of processes, since all processes will reach the barrier at roughly the same time and will be distributing their updates simultaneously. This would explain the drop off in performance as the number of processes increases. As this implementation of DSM requires a large amount of memory to store copies of pages while they are being updated and the available memory on our workstations was limited, a matrix size of 60×1024 was used. Other implementations used matrices of 2048×2048 . The speed-up results are shown in figure 8. The speed-up for eight workstations is 5.2.

6. Related work

Here we report on the work of other researchers on DSM systems paying attention to whether they are easy to program and use. The DSM systems we discuss are Munin, Midway and TreadMarks, selected because they use similar consistency models to RHODOS DSM. Furthermore, when we carried out our performance tests on RHODOS DSM [7] we used the application code written by the researchers who developed Munin: a version of TSP used in [8] was obtained from the ftp site at Rice University and TreadMarks: a version of Jacobi was obtained from Keleher [9]. We were able to see the input required by programmers when running these systems and compared it with the input required by programmers using RHODOS DSM.

Munin [10] is an object-based DSM which has multiple consistency protocols and runs on top of the V operating system. Munin and RHODOS show a speed-up for TSP of approximately 6.8 and 6.85, respectively. The initialization

stage of an application using Munin requires programmers to define the number of workstations to be used. The names of the workstations are read from a file containing a list of workstations which has been created by programmers. Having identified the workstations on which the application will run, programmers must create both a thread and initialize the shared data on each of these workstations, and create the synchronization barriers [10].

TreadMarks is a DSM system implemented on top of Unix. Programmers are required to have substantial input into the initialization of DSM processes by either entering the number of workstations and their names as command-line arguments or creating a workstation list file which contains a list of the workstations that comprise the COW and may be used for parallel applications [11]. Tests were carried out on TSP and Jacobi; both achieved speed-ups of approximately 7. Our speed-up for TSP is 6.85 and that for Jacobi is 5.2. However, it is difficult to compare these results with those for our DSM system because the COW used for the TreadMarks tests was considerably faster than the one used for our tests. Their COW consisted of eight DECstation-5000/240, each with a Fore ATM interface and connected to a Fore ATM switch with an aggregate throughput of 1.2 Gbps [12].

In existing DSM systems ease of programming is largely ignored with programmers being expected to go to extraordinary lengths in order to extract the best execution performance. Midway [13] uses entry consistency and requires programmers not only to label shared variables but also to associate these variables with synchronization constructs. Munin [10] requires programmers to use different consistency protocols for variables according to their memory access patterns. These approaches clearly require programmers to gain additional skills and to have an insight into the implementation of the DSM system and a deeper than usual insight into the data access patterns

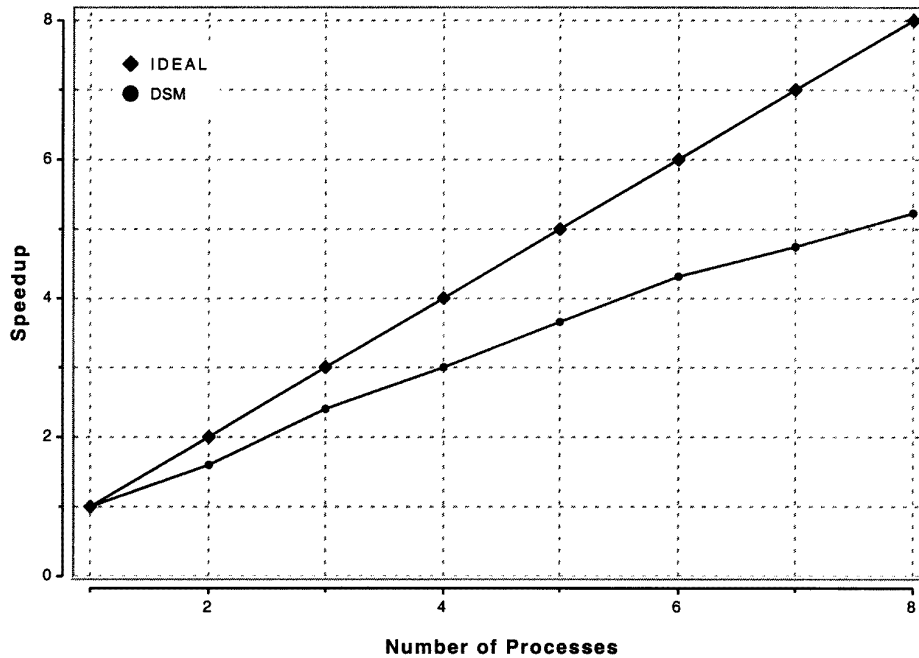


Figure 8. Speed-up for Jacobi on a matrix of 60×1024 elements.

of the application they are running in order to use the DSM successfully. The only known transparent approach to the sequential consistency problem able to improve the execution performance of the DSM-based application is presented in [14]. This approach does not require programmer's annotations to associate shared data objects with synchronization operations. Furthermore, consistency scopes are not defined by programmers. However, programmers must still be involved in the coding of the initialization of processes, data, synchronization primitives and virtual computer definitions.

While it is difficult to quantify ease of programming and use, these systems clearly require programmers to have significant input into and intimate knowledge of the implementation of the DSM system and knowledge of the network on which these systems work as well as the load on individual workstations.

7. Conclusions

In this paper we have demonstrated that it is possible to develop a comprehensive DSM system which allows programmers to use knowledge of sequential programming where only barrier-based synchronization of parallel processes is needed and is easy to use, and which still provides high performance, comparable to or even better than other DSM systems. Operating-system-based DSM makes operations transparent and nearly completely reduces the involvement of programmers beyond classical activities needed to deal with shared memory.

The automatic initialization of the RHODOS DSM processes provides users with a convenient environment and extends the operating system by giving it control over the load on the individual workstations in the COW. Programmers using RHODOS DSM are relieved from the operating-system-oriented activities to locate and schedule processes

to the least loaded or idle workstations in the COW and to balance the load of the system. Furthermore, they are able to initialize the DSM system and parallel processes through the use of a single, simple library call. Being able to use DSM with a single library call makes RHODOS DSM transparent, easy to program and easy to use. Moreover, the performance of RHODOS DSM, as shown in this paper, is good.

Thus, we can conclude that DSM, when implemented within a distributed operating system, is one of the most promising approaches to parallelism management and guarantees huge performance improvements with the minimum of involvement of programmers.

Acknowledgments

The authors wish to express their gratitude to the anonymous referees and the editors for their constructive comments and suggestions. This work was partly supported by the Small ARC grant 0504003157.

References

- [1] Iftode L and Singh J 1999 Shared virtual memory: progress and challenges *Proc. IEEE* **87** 498–507
- [2] De Paoli D, Goscinski A, Hobbs M and Joyce P 1996 Performance comparison of process migration with remote process creation and execution in RHODOS *Int. Conf. on Distributed Computer Systems (ICDCS-96) (Hong Kong)* (Piscataway, NJ: IEEE) pp 554–61
- [3] Silcock J and Goscinski A 1997 Update-based distributed shared memory integrated into RHODOS' memory management *The 3rd Int. Conf. on Algorithms and Architecture for Parallel Processing (ICA3PP'97) (Melbourne)* (Piscataway, NJ: IEEE) pp 239–52
- [4] Goscinski A 1997 Parallel processing on clusters of workstations *IEEE Singapore Conf. on Networks (SICON '97) (Singapore)* (Piscataway, NJ: IEEE) pp 427–54

- [5] De Paoli D, Hobbs M and Goscinski A 1995 The RHODOS microkernel, kernel servers and their cooperation *IEEE 1st Int. Conf. on Algorithms and Architectures for Parallel Processing (ICA3PP'95) (Brisbane)* (Piscataway, NJ: IEEE) pp 345–54
- [6] Hobbs M and Goscinski A 1997 Supporting parallel processing on the RHODOS cluster of workstations *1997 Int. Symp. on Parallel Architectures, Algorithms and Networks (Taipei)* pp 261–7
- [7] Silcock J and Goscinski A 1997 Performance studies of distributed shared memory embedded in the RHODOS operating system *The 4th Australasian Conf. on Parallel and Real-Time Systems (PART '97) (Newcastle)* pp 356–67
- [8] Amza C, Cox A, Dwarkadas S, Keleher P, Lu H, Rajamony R, Yu W and Zwaenepoel W 1996 Treadmarks: shared memory computing on networks of workstations *IEEE Comput.* **29** 18–28
- [9] Keleher P 1996 Private communication
- [10] Carter J 1993 Efficient distributed shared memory based on multi-protocol release consistency *PhD Dissertation* Rice University
- [11] ParallelTools, LLC 1994 *Concurrent Programming with TreadMarks*
- [12] Keleher P, Cox A, Dwarkadas S and Zwaenepoel W 1994 TreadMarks: distributed shared memory on standard workstations and operating systems *The 1994 Winter Usenix Conf.* pp 115–31
- [13] Bershad B, Zekauskas M and Sawdon W 1993 The midway distributed shared memory system *The IEEE COMPCON Conf.* (Piscataway, NJ: IEEE) pp 528–37
- [14] Sun C, Huang Z, Lei W and Sattar A 1998 Toward transparent selective sequential consistency in distributed shared memory systems *The Int. Conf. of Distributed Computing Systems (ICDCS'98) (Amsterdam)* pp 572–81