COMMUNICATION REQUIREMENTS OF SOFTWARE DISTRIBUTED SHARED MEMORY SYSTEMS

Sumit Roy and Vipin Chaudhary *

Abstract

Software Distributed Shared Memory (DSM) systems have been proposed to provide the convenience of shared memory programming on clusters of workstations. Different research groups advocate specific design decisions in their implementations. In this paper we present an experimental comparison of the communication requirements of three different page based systems software DSM systems, CVM, Quarks and Strings. Programs from the SPLASH-2 benchmark suite and some computational kernels are used to evaluate these systems. This paper shows the effect of different design decisions used in each of the DSMs on the same evaluation environment, a cluster of symmetrical multiprocessors. It is seen that systems that allow multiple application threads, like Strings and CVM, require lower number of messages when compared to Quarks. CVM and Quarks use user level threads for implementing the runtime, and it seen that they cannot effectively exploit multiple processors on each node. The Strings DSM uses kernel level threads and shows the best overall performance in a majority of the applications examined.

1 INTRODUCTION

In recent years it has been found that single processor based computers are not able to solve increasingly large and complex scientific and engineering problems since they are rapidly reaching the limits of possible physical performance. Multiple co-operating processors have instead been used to study *Grand Challenge Problems* like Fuel combustion, Ocean modeling, Image understanding, Rational drug design, etc. A popular design strategy adopted by many vendors of traditional workstations involves using multiple state-of-the art microprocessors to build high performance shared-memory workstations. These symmetrical multi-processors (SMPs) are then connected through high speed networks or switches to form a scalable computing cluster.

Application programs can be converted to run on multiple processors on a node using parallelizing compilers, however using multiple nodes often requires the programmer to write explicit message passing programs. An alternative is to provide a view of logically shared memory over physically distributed memory, known as a Distributed Shared Memory (DSM) or Shared Virtual Memory (SVM). This approach provides an easy programming interface ie. shared memory, as well as the scalability of distributed memory machines. Page based DSMs use the virtual memory mechanism provided by the operating system, to trap accesses to shared areas. Examples include TreadMarks [1], Quarks [2], CVM [3], and Strings [4]. An important aim in DSM design is to minimize the communication requirements

* *Parallel and Distributed Computing Laboratory, Wayne State University, Detroit, MI – 48202*

of the system. Single writer based systems that use an invalidate protocol to enforce consistency can suffer from a phenomenon known as "false sharing". Two processes write to different data items that reside on the same virtual memory page, which causes it to ping-pong between these nodes. This causes excess, needless traffic. Multiple writer implementations with relaxed memory consitency models have been proposed to alleviate this symptom. In this approach, shared memory pages are guaranteed to be consistent at well defined points in the program, typically at synchronization points. An additional optimization to reduce network traffic is to only send the *changed* data from a modified page. These changes are obtained by *diffing* the current page with original copy.

This paper presents our experiences evaluating the communication requirements of three DSMs, CVM, Quarks and Strings, using programs from the SPLASH-2 [5] benchmark suite. CVM and Quarks use user level threads to improve the performance on the systems by overlapping communication and computation. However it is seen that they cannot effectively exploit multiple processors on each node. The Strings DSM uses kernel level threads and shows the best overall performance in a majority of the applications examined.

The rest of the paper is organized as follows. Some of the details of the DSMs used are presented in Section 2. Section 3 describes the experimental environment and the programs used for the evaluation. The results are given in Section 4. Section 5 concludes the paper.

2 SOFTWARE DSM OVERVIEW

The DSM systems that were evaluated are the page based systems CVM, Quarks and Strings.

2.1 CVM: Coherent Virtual Machine

CVM [3] is a page-based DSM implemented as a user level library which is linked with shared memory programs. The base system provides a generic coherence protocol, user level lightweight threads and reliable network communication on top of *UDP*. The shared memory system is created by preallocating a large number of pages from the process heap and then managing this space. CVM programs create remote processes using *rsh*. Each process executes the sequential part of code until application threads have to be created. Shared memory pages are created on all nodes at the same time, and at the same address, but with ownership assigned to the master process. All variables initialized by the master process will also be initialized by the remote slave processes. CVM allows multiple application threads to execute within the context of each process, and can potentially overlap computation in one thread with communication in another thread. An extensible interface for adding additional coherence protocols is also provided. It currently supports Lazy Multi Writer Release, Lazy Single Writer Release, and Sequential consistency. In Lazy Release Consistency, notifications about *diffs* are sent only to the next node that acquires a synchronization variable. In case of CVM, the protocol is implemented using invalidates.

2.2 Quarks

Quarks [2] consists of a multithreaded runtime library that is linked with an application program and a central server that is used to coordinate allocation of shared regions, barriers and locks. *Quarks* programs use *rsh* to start remote tasks during the initialization phase. Each task spawns a *dsm_server* thread and registers itself with the main server. All further execution is controlled by the main thread, which creates shared data regions and then spawns remote threads by contacting the respective *dsm_server*. As part of the spawn semantics, information about all shared memory regions and initialized synchronization primitives are passed to the remote node. A shared page is allocated by the master process using *mmap()*. Access to shared pages is monitored using the virtual memory subsystem. The central server is used to allocate pages from a global address space which is mapped to the local address space via a translation table. Quarks implements a Sequential Consistency with a write invalidate protocol, and Release Consistency [6] using a write update protocol. The thread system is based on *Cthreads* and, like in CVM, is a user level implementation. The currently released version of Quarks does not allow multiple application threads on each node. A reliable messaging system is built on top of the *UDP* protocol. Incoming messages cause an interrupt, and the messages are then handled by the *dsm_server* thread.

2.3 Strings

Strings [4] is a fully multithreaded DSM implemented as a user space library. The distinguishing feature of *Strings* is that it incorporates Posix1.c threads multiplexed on kernel light-weight processes for better performance. The kernel can schedule multiple threads across multiple processors on SMP nodes, using these lightweight processes. Thus, *Strings* is designed to exploit data parallelism at the application level and task parallelism at the DSM system level. A separate *communication* thread issues a blocking listen on a message socket. Hence, incoming messages do not have to generate interrupts to be serviced, which reduces disruption to computing threads. The Strings runtime creates short lived threads for handling requests arriving from other nodes. Thus multiple independent requests can be serviced in parallel.

Portable global pointers are implemented across nodes in the DSM program by mapping the shared regions to fixed addresses in the memory space. Strings implements Release Consistency with an update protocol.

3 TEST ENVIRONMENT

All experiments were carried out on a cluster of four SUN UltraEnterprise Servers. One machine is a six processor UltraEnterprise 4000 with 1.5 Gbyte memory. For *Quarks* the *server* process was always run on this machine. The master program for all DSMs was started on this machine. The other machines are four processor UltraEnterprise 3000s, with 0.5 Gbyte memory each. All four machines use 250 MHz UltraSparcII processors, with 4 Mb external cache. The network used to interconnect the machines in the cluster is 100 Mbps FastEthernet with a BayStack FastEthernet Hub.

3.1 Test Programs

The test programs used to do the evaluation consists of programs from the SPLASH-2 benchmark suite [5], matrix multiplication, as well as a kernel for solving Partial Differential Equations using the Successive Over relaxation technique.

3.2 SPLASH-2 Benchmark Programs

The SPLASH-2 Benchmark programs have been written for evaluating the performance of shared address-space multiprocessors and include application kernels as well as full fledged code. The execution model for SPLASH-2 programs follows the Single Program Multiple Data (SPMD) type, and has three phases. In the first phase the main task reads in command-line parameters and the initial data. It sets up a shared memory region and allocates globally shared data from this region. Synchronization variables are also initialized. After this initialization phase, tasks are created to execute the actual *slave()* routine in the computation phase. The master also runs a copy of this routine. In the termination phase the master thread collects the results and generates timing statistics. The SPLASH-2 program suite has been designed to be portable by encapsulating shared memory initialization, synchronization primitives, and task creation within ANL macros [7]. These macros should then be adapted for any shared memory platform.

The data access patterns of the programs in the SPLASH-2 suite have been characterized in earlier research [8, 9]. FFT performs a one-dimensional Fast Fourier Transform of n complex data points. Three all-to-all interprocessor communication phases are required for a matrix transpose. The data access pattern is hence regular. Two programs for the blocked LU factorization of a dense matrix form part of the suite. The non-contiguous (LU-n) version has a single producer and multiple consumers. It suffers from considerable fragmentation and false sharing. The contiguous version (LU-c) uses an array of blocks to improve spatial locality. The Ocean simulation program with contiguous partitions (OCEAN-c) simulates large scale ocean movements. This version uses a red-black Gauss-Seidel multi-grid equation solver and has a regular nearest-neighbor type access pattern. The last two programs evaluate the forces and potentials occurring over time in a system of water molecules. The first version (WATER-n2) uses a simple data structure, which results in a less efficient algorithm. The second version (WATER-sp) uses a 3-D grid of cells so that a processor that owns a cell only needs to look at neighboring cells to find interacting molecules. Communication arises out of the movement of molecules from one cell to another at every time-step.

3.3 Matrix Multiplication

The matrix multiplication program uses a row-wise block distribution of the resultant matrix. The size of the blocks is a multiple of the page size of the machines used (ie. 8 kbytes), and each application thread computes at complete blocks of contiguous values. This removes the effect of false sharing and makes this application close to ideal for execution on a DSM system.

3.4 Successive Over Relaxation

The successive over relaxation uses a red-black algorithm and was adapted from the CVM sample code. Most of the traffic arises out of nearest neighborhood communication at the borders.

4 EXPERIMENTAL RESULTS

Each SPLASH-2 benchmark was modified so as to conform to the execution model assumed by the respective DSM environment.

4.1 Program Parameters

The programs and the parameters used for this part are shown in Table 1. The program sizes had to be altered in some cases since some of the DSM systems could not support larger datasets. In the case of Quarks, each task corresponds to a separate process, since it does not support multiple application threads per process. CVM and *Strings* used a single process per node with multiple application threads. The tasks distribution for the systems is shown in Table 2.

4.2 Communication Results

Table 3 shows the number of messages sent per computation thread for the three DSMs tested.

Clearly Quarks generates far more messages than Strings or CVM in all programs. This shows that using multiple threads per node effectively reduces the communication requirements. When a page is required by multiple threads on a node, it has to be faulted in only once, and thus generates less network traffic. Figures 1 – 8 show the overall results for each program with the different DSM systems.

The CVM programs indicate that using multiple user level application threads on a node does not provide any performance improvement. The reduced compute time in the 4 task case is compensated for by the increase in the barrier wait time when compared to the 2 task case. For short running programs like FFT and OCEAN-c, the startup costs increase as the number of nodes is increased, and the time spent on paging related activities also goes up. CVM allows out-of-order access to locks, and uses a centralized lock manager. Since the consistency model requires sending write invalidates only to the next lock acquirer, the overall effect is seen as a very small time for locking, when compared to other DSMs. For LU-c and WATER-n2, the relative performance improves as the number of nodes is increased, but the absolute performance is not as good as that of *Strings*.

Quarks shows excessive network collisions in the case of LU-c, as one goes from using 2 nodes to 4 nodes. The performance is so bad that it was not possible to obtain results with 16 tasks for this program. OCEAN-c, WATER-n2, and WATER-sp indicate that there is a large cost associated with lock related traffic. CVM shows much better results for these programs, since multiple requests on the same node preempt requests from other nodes. Quarks on the other hand tries to implement a very fair scheme, maintaining a distributed FIFO queue. The release consistency model also requires that all modified pages are flushed to all nodes before a lock can be released. The barrier costs for Quarks are very high. CVM optimizes barriers by collecting arrivals from all the local threads in a process before passing on the notice to the barrier manager.

From the *Strings* performance it can be seen that, though the overall performance is good, in case of OCEAN-c and WATER-sp, the lock time forms a large component of the total execution time. *Strings* uses the same algorithm for lock management as Quarks, hence suffers from similar problems. Release consistency requires that updates are flushed to all nodes that share pages, at lock release time. The advantage of using multiple kernel threads is seen in compute bound programs like MATMULT, LU-c and WATER-n2. The barrier costs are noticeably higher when compared to CVM. The current version sends one message per arriving thread to the barrier manager. This part of *Strings* should be optimized further.

5 CONCLUSION

Based on the experiments so far, the performance of the DSMs studied depends to some degree on the characteristics of the application program. However using multiple threads per node reduces the number of messages generated per task and leads to performance improvments. The choice of consistency model also affects the number of messages generated. However the total amount of data that is transmitted is roughly the same. This suggests that in case of Strings, a larger number of small, acknowledgment messages are being sent. Hence future work should address the actual implementation of the communication subsystem in CVM and Strings.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaeonopoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, pp. 18–28, February 1996.

[2] D. Khandekar, *Quarks: Portable Distributed Shared Memory on Unix*. Computer Systems Laboratory, University of Utah, beta ed., 1995.

[3] P. Keleher, *CVM: The Coherent Virtual Machine*. University of Maryland, CVM Version 2.0 ed., July 1997.

[4] S. Roy and V. Chaudhary, "*Strings:* A High-Performance Distributed Shared Memory for Symmetrical Multiprocessor Clusters," in *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, (Chicago, IL), pp. 90 –97, July 1998.

[5] S. C. Woo, M. Ohara, E. Torri, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the International Symposium on Computer Architecture*, pp. 24–36, June 1995.

[6] J. Bennett, J. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," in *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming*, (New York), pp. 168–176, ACM, ACM Press, 1990.

[7] R. Hempel, H. C. Hoppe, U. Keller, and W. Krotz, *PARMACS V6.0 Specification. Interface Description*. Pallas GmbH, November 1993.

[8] D. Jiang, H. Shan, and J. P. Singh, "Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors," in *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming*, (Las Vegas), pp. 217 – 229, ACM, 1997.

[9] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood, "Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation," in *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming*, (Las Vegas), pp. 193 – 205, June 1997.

Program	Parameters
FFT	1048576 complex doubles
LU-c	512 $\times$ 512 doubles, block size 16
LU-n	512 $\times$ 512 doubles, block size 32
OCEAN-c	258 $\times$ 258 grid
WATER-n2	1728 molecules, 3 steps
WATER-sp	1000 molecules, 3 steps
MATMULT	1024 $\times$ 1024 doubles, 16 blocks
SOR	2002 $\times$ 1002 grid, 100 iterations

Table 1: Program Parameters for DSM Comparison

Total Tasks	Nodes Used		Processes per Node		Tasks per Process	
	Quarks	CVM Strings	Quarks	CVM Strings	Quarks	CVM Strings
1	1	1	1	1	1	1
2	2	2	1	1	1	1
4	4	4	1	1	1	1
8	4	4	2	1	1	2
16	4	4	4	1	1	4

Table 2: Task Distribution for DSM Comparison

Program	CVM	Quarks	Strings
FFT	1290	2419	1894
LU-c	135	-	485
LU-n	385	2873	407
OCEAN-c	1955	15475	6676
WATER-n2	2253	38438	10032
WATER-sp	905	7568	1998
MATMULT	290	1307	645
SOR	247	7236	934

Table 3: Messages Generated for each DSM (16 tasks)

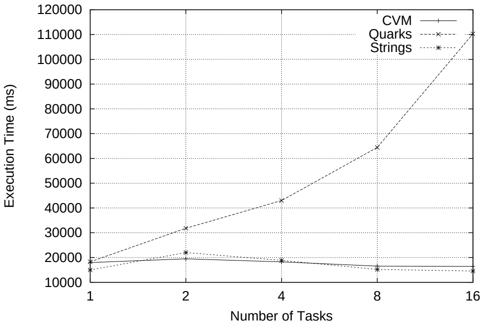

Figure 1: FFT

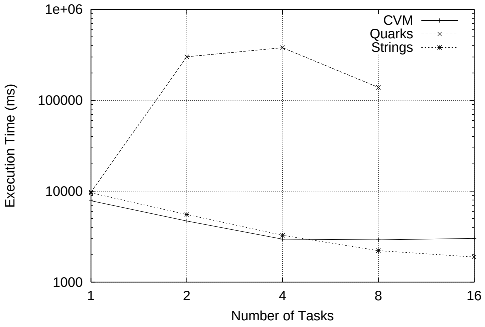

Figure 2: LU-c

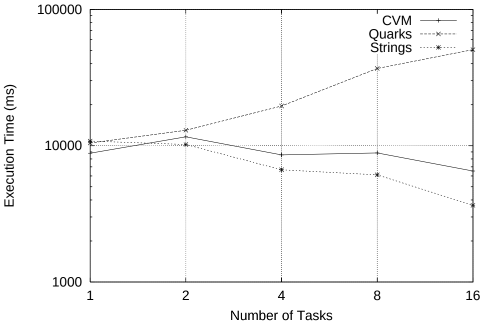

Figure 3: LU-n

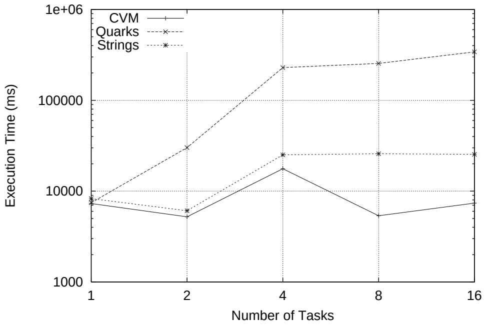

Figure 4: OCEAN-c

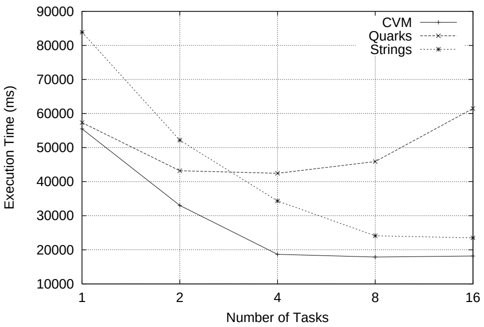

Figure 5: WATER-n2

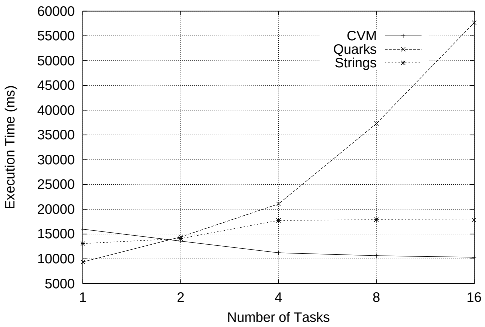

Figure 6: WATER-sp

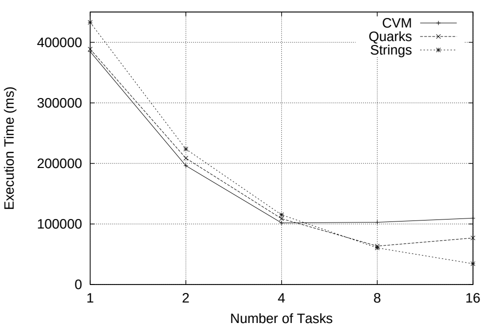

Figure 7: MATMULT

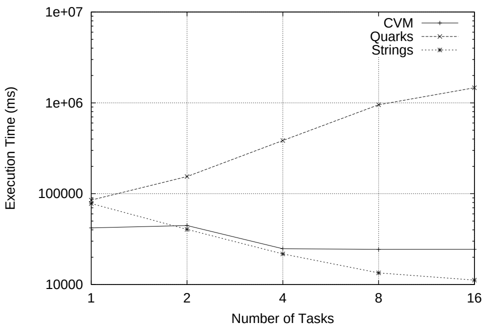

Figure 8: SOR