Design Issues for a High-Performance Distributed Shared Memory on Symmetrical Multiprocessor Clusters *

Sumit Roy and Vipin Chaudhary

Parallel and Distributed Computing Laboratory, Department of Electrical and Computer Engineering, Wayne State University, Detroit, Michigan 48202

E-mail: (sroy/vchaud)@ece.eng.wayne.edu

Clusters of Symmetrical Multiprocessors (SMPs) have recently become the norm for high-performance economical computing solutions. Multiple nodes in a cluster can be used for parallel programming using a message passing library. An alternate approach is to use a software Distributed Shared Memory (DSM) to provide a view of shared memory to the application programmer. This paper describes *Strings*, a high performance distributed shared memory system designed for such SMP clusters. The distinguishing feature of this system is the use of a fully multi-threaded runtime system, using kernel level threads. *Strings* allows multiple application threads to be run on each node in a cluster. Since most modern UNIX systems can multiplex these threads on kernel level light weight processes, applications written using *Strings* can exploit multiple processors on a SMP machine. This paper describes some of the architectural details of the system and illustrates the performance improvements with benchmark programs from the SPLASH-2 suite, some computational kernels as well as a full fledged application.

It is found that using multiple processes on SMP nodes provides good speedups only for a few of the programs. Multiple application threads can improve the performance in some cases, but other programs show a slowdown. If kernel threads are used additionally, the overall performance improves significantly in all programs tested. Other design decisions also have a beneficial impact, though to a lesser degree.

Keywords: Distributed Shared Memory, Symmetrical Multiprocessors, Multithreading, Performance Evaluation

1. Introduction

Though current microprocessors are getting faster at a very rapid rate, there are still some very large and complex problems that can only be solved by using multiple cooperating processors. These problems include the so-called *Grand Challenge Problems*, such as Fuel combustion, Ocean modeling, Image understanding, and Rational drug design. There has recently been a decline in the number of specialized parallel machines being built to solve such problems. Instead, many vendors of traditional workstations have adopted a design strategy wherein multiple state-of-the-art microprocessors are used to build high performance shared-memory parallel workstations. These symmetrical multiprocessors (SMPs) are then connected through high speed networks or switches to form a scalable computing cluster. Examples of this important class of machines include the SGI Power Challenge Array, the IBM SP2 with multiple POWER3 based nodes, the Convex Exemplar, the DEC AdvantageCluster 5000, the SUN HPC cluster with the SUN Cluster Channel, as well as the Cray/SGI Origin 2000 series.

Using multiple nodes on such SMP clusters requires the programmer to either write explicit mes-

* This research was supported in part by NSF grants MIP-9309489, EIA-9729828, US Army Contract DAEA 32-93D004 and Ford Motor Company grants 96-136R and 96-628R. A preliminary version of this paper appeared in the Proceedings of the High Performance Distributed Computing Conference, 1998.

sage passing programs, using libraries like MPI or PVM; or to rewrite the code using a new language with parallel constructs, eg. HPF and Fortran 90. Message passing programs are cumbersome to write, while parallel languages usually only work well with code that has regular data access patterns. In both cases the programmer has to be intimately familiar with the application program as well as the target architecture to get the best possible performance. The shared memory model on the other hand, is easier to program, since the programmer does not have to worry about the data layout and does not have to explicitly send data from one process to another. Hence, an alternate approach to using these computing clusters 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). Recent research projects with DSMs have shown good performance, for example IVY [1], Mirage [2], Munin [3], TreadMarks [4], Quarks [5], CVM [6], and *Strings* [7]. This model has also been shown to give good results for programs that have irregular data access patterns, which cannot be analyzed at compile time [8], or indirect data accesses that are dependent on the input data-set.

DSMs share data at the relatively large granularity of a virtual memory page and can suffer from a phenomenon known as "false sharing", wherein two processes simultaneously attempt to write to different data items that reside on the same page. If only a single writer is permitted, the page may ping-pong between the nodes. One solution to this problem is to "hold" on to a freshly arrived page for some time before releasing it to another requester [2]. Relaxed memory consistency models that allow multiple concurrent writers have also been proposed to alleviate this symptom [9,10,4,11]. These systems ensure that all nodes see the same data at well defined points in the program, usually when synchronization occurs. Extra effort is required to ensure program correctness in this case.

One technique that has been investigated recently to improve DSM performance is the use of multiple threads of control in the system. Multi-threaded DSMs have been described as third generation systems [12]. Published efforts have been restricted to non-preemptive, user-level thread implementations [5,13]. However, user level threads cannot be scheduled across multiple processors on an SMP. Since SMP clusters are increasingly becoming the norm for High Performance Computing sites, we consider this to be an important problem to be solved. This paper describes *Strings*, a multi-threaded DSM designed for SMP clusters. The distinguishing feature of *Strings* is that it is built using POSIX threads, which can be multiplexed on kernel light-weight processes. The kernel can schedule these lightweight processes across multiple processors for better performance. *Strings* is designed to exploit data parallelism by allowing multiple application threads to share the same address space on a node. Additionally, the protocol handler is multithreaded and is able to use task parallelism at the runtime level. The overhead of interrupt driven network I/O is avoided by using a dedicated communication thread. We show the impact of each of these design choices using some example programs, as well as some benchmark programs from the SPLASH-2 suite [14].

The following section describes some details of the software system. The evaluation platform and programs for the performance analysis are described in section 3. Experimental results are shown and analyzed in section 4. Section 5 suggests some direction for future work and concludes the paper.

2. System details

The *Strings* distributed shared memory was derived from the publicly available system Quarks [5]. It shares the use of the Release Consistency model with that system, as well as the concept of a `dsm_server` thread. We briefly describe the implementation details and highlight the difference between the two systems.

2.1. Execution model

The *Strings* system consists of a library that is linked with a shared memory parallel program. The program uses calls to the distributed shared memory allocator to create globally shared memory regions. A typical program goes through the initialization shown in Figure 1.

The master process starts up and forks child processes on remote machines using `rsh()`. Each process creates a `dsm_server` thread and a `communication` thread. The forked processes then register their listening ports with the master. The master process then enters the application program proper, and creates shared memory regions. `Application` threads are then created by sending requests to the remote `dsm_servers`. Shared region identifiers and global synchronization primitives are sent as part of the thread create call. The virtual memory subsystem is used to enforce coherent access to the globally shared regions.

The original Quarks system used user level *Cthreads* and allowed only a single application thread. *Strings* allows multiple application threads to be created on a single node. This increases the concurrency level on each node in a SMP cluster.

2.2. Kernel level threads

Threads are light-weight processes that have minimal execution context, and share the global address space of a program. The time to switch from one thread to another is very small when compared to the context switching time required for full-fledged processes. Moreover the implicit shared memory leads to a very simple programming model. Thread implementations are distinguished as being user-level, usually implemented as a library, or as being kernel level in terms of light-weight processes. Kernel level threads are a little more expensive to create, since the kernel is involved in managing them. However, user level threads suffer from some important limitations. Since they are implemented as a user level library, they cannot be scheduled by the kernel. If any thread issues a blocking system call, the kernel considers the process as a whole, and thus all the associated threads, to be blocked. Also, on a multiprocessor system, all user level threads can only run on one processor at a time. User level threads do allow the programmer to exercise very fine control on their scheduling within the process. In contrast, kernel level threads can be scheduled by the operating system across multiple processors. Most modern UNIX implementations provide a light-weight process interface on which these threads are then multiplexed. The thread package used in *Strings* is the standard Posix 1003.1c thread library. Multi-threading has been suggested for improving the performance of scientific code by overlapping communications with computations [15]. Previous work on multi-threaded message passing systems has pointed out that kernel-level implementations show better results than user level threads for a message size greater than 4 K bytes [16]. Since the page size is usually 4 K or 8 K bytes, it suggests that kernel threads should be useful for DSM systems.

2.3. Shared memory implementation

Shared memory in the system is implemented by using the UNIX `mmap()` call to map a file to the bottom of the stack segment. Quarks used anonymous mappings of memory pages to addresses determined by the system, but this works only with statically linked binaries. With dynamically linked programs, it was found that due to the presence of shared libraries `mmap()` would map the same page to different addresses in different processes. While an address translation table can be used to access opaquely shared data, it is not possible to pass pointers to shared memory this way, since they would potentially address different regions in different processes. An alternate approach would be to preallocate a very large number of pages, as done by CVM and TreadMarks, but this associates the same large overhead with every program, regardless of its actual requirements.

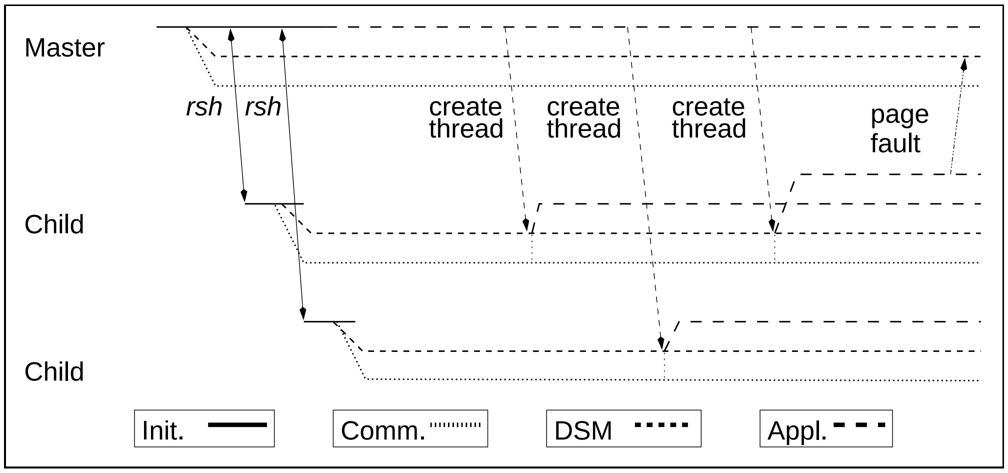

Figure 1. Initialization Phase of a *Strings* Program

Allowing multiple application threads on a node leads to a peculiar problem with the DSM implementation. Once a page has been fetched from a remote node, its contents must be written to the corresponding memory region, so the protection has to be changed to *writable*. At this time no other thread should be able to access this page. User level threads can be scheduled to allow atomic updates to the region. However, suspending all kernel level threads can potentially lead to a deadlock, and would also reduce concurrency. Figure 2 illustrates the approach used in *Strings*. Every page is mapped to two different addresses. It is then possible to write to the 'shadow' address, without changing the protection of the primary memory region.

The `mprotect()` call is used to control access to the shared memory region. When a thread faults while accessing a page, a page handler is invoked to fetch the contents from the owning node. *Strings* currently supports sequential consistency using an invalidate protocol, as well as release consistency using an update protocol [10,5]. When a thread tries to write to a page, a twin copy of the page is created. At releases time, ie. when a lock is unlocked, or a barrier is entered, the difference or *diff* between the current contents of the page and its twin is sent to other threads that share the page. The release consistency model implemented in Quarks has been improved by aggregating multiple diffs to decrease the number of messages sent.

2.4. Polled network I/O

Early generation DSM systems used interrupt driven I/O to obtain pages, locks etc. from remote nodes. This can cause considerable disruption at the remote end, and previous research tried to overcome this by aggregating messages, reducing communication by combining synchronization with data, and other such techniques [17]. *Strings* uses a dedicated `communication` thread, which monitors the network port, thus eliminating the overhead of an interrupt call. Incoming message queues are maintained for each active thread at a node, and message arrival is announced using condition variables. This prevents wasting CPU cycles with busy waits. A reliable messaging system is implemented on top of UDP.

2.5. Concurrent server

The original Quarks `dsm_server` thread was an iterative server that handled one incoming request at a time. It was found that under certain conditions, lock requests could give rise to a deadlock between two communicating processes. *Strings* solves this by creating separate threads to handle each incoming request for pages, lock acquires and barrier arrivals. Relatively fine grain locking of internal data structures is used to maintain a high level of concurrency while guaranteeing correctness when handling multiple concurrent requests.

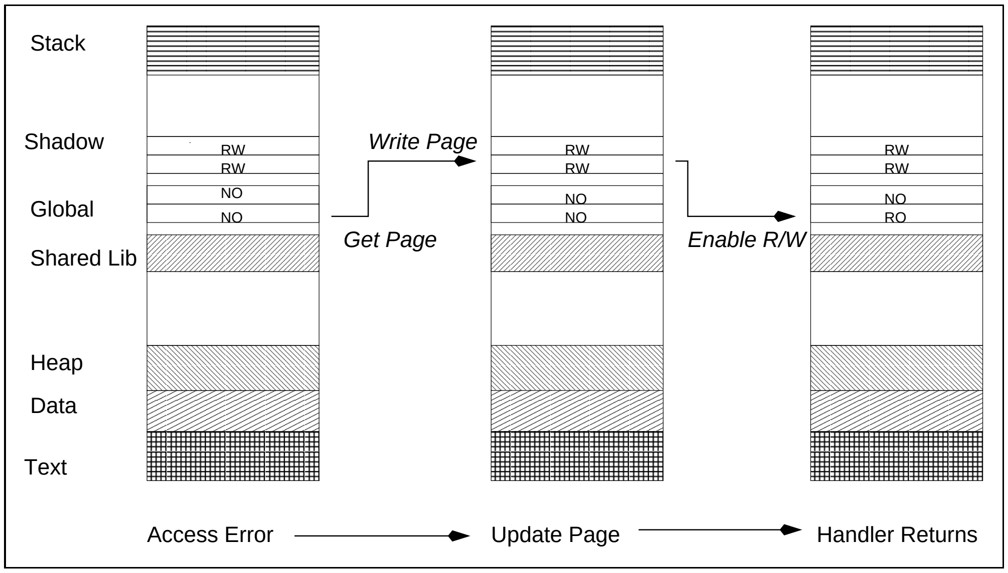

Figure 2. Thread safe memory update for *Strings*

2.6. Synchronization primitives

Quarks provides barriers and locks as shared memory primitives. *Strings* also implements condition variables for flag based synchronization. Barriers are managed by the master process. Barrier arrivals are first collected locally, and are then sent to the barrier manager. Dirty pages are also purged at this time, as per Release Consistency semantics [10].

Lock ownership is migratory with distributed queues. For multiple application threads, only one lock request is sent to the current owner, the subsequent ones are queued locally, as are incoming requests. Requests on the same node pre-empt request from remote node. While this does not guarantee fairness or progress, this optimization works very well for data parallel programs. A similar optimization was employed in CVM [18]. Release Consistency operations are deferred if the lock transfer is within the local node.

3. Performance analysis

We evaluated the performance of *Strings* using programs from the SPLASH-2 benchmark suite [14]. These programs have been written for evaluating the performance of shared address-space multiprocessors and include application kernels as well as full fledged code. Additionally we show results for matrix multiplication, a program from the field of medical computing, as well as a kernel for solving partial differential equations by the successive over-relaxation technique and the classical traveling salesman problem.

3.1. SPLASH-2 programs

The data access patterns of the programs in the SPLASH-2 suite have been characterized in earlier research [19,11]. FFT performs a transform of n complex data points and requires three all-to-all interprocessor communication phases for a matrix transpose. The data access is regular. LU-c and LU-n perform factorization of a dense matrix. The non-contiguous version has a single producer and multiple consumers. It suffers from considerable fragmentation and false sharing. The contiguous version uses an array of blocks to improve spatial locality. RADIX performs an integer radix sort and suffers from a high-degree of false sharing at page granularity during a permutation phase. RAYTRACE renders a complex scene using an optimized ray tracing method. It uses a shared task queue to allocate jobs to different threads. Since the overhead of this approach is very high in a DSM system, the code was modified, to maintain a local as well as global queue per thread. Tasks were initially drained from the local queue, and then from the shared queue. VOLREND renders three-dimensional volume data. It has a multiple producers with multiple consumers data sharing pattern, with both fragmentation and false sharing. WATER-sp evaluates the forces and potentials occurring over time in a system of water molecules. A 3-D grid of cells is used 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. WATER-n2 solves the same problem as WATER-sp, though with a less efficient algorithm that uses a simpler data-structure.

3.2. Image deblurring

The application tested is a parallel algorithm for deblurring of images obtained from Magnetic Resonance Imaging. Images generated by MRI may suffer a loss of clarity due to inhomogeneities in the magnetic field. One of the techniques for removing this blurring artifact is the demodulation of the data for each pixel of the image using the value of the magnetic field near that point in space. This method consists of acquiring a local field map, finding the best fit to a linear map and using it to deblur the image distortions due to local frequency variations. This is a very computation intensive operation and has previously been parallelized using a message passing approach [20]. The shared memory implementation uses a work-pile model, where each thread deblurs the input image around a particular frequency points and then updates the relevant portions to the final image. Since these portions can overlap, each thread does the update under the protection of a global lock.

3.3. Matrix multiplication

The matrix multiplication program (MATMULT) computes the product of two dense, square matrices. The resultant matrix is partitioned using a block-wise distribution. The size of the blocks can be set to a multiple of the page size of the machine. Since each application thread computes a contiguous block of values, this eliminates the problem of false sharing.

3.4. Successive Over Relaxation

The successive over relaxation program (SOR) uses a red-black algorithm and was adapted from the *CVM* sample code. In every iteration, each point in a grid is set to the average of its four neighbors. Most of the traffic arises out of nearest neighborhood communication at the borders of a rectangular grid.

3.5. Traveling Salesman Problem

The Traveling Salesman Problem (TSP) was also adapted from the *CVM* sample code. The program solves the classic traveling salesman problem using a branch-and-bound algorithm.

3.6. Evaluation Environment

Our experiments were carried out so as to show how various changes in the system impact performance. The runs were carried out on a cluster of four SUN UltraEnterprise Servers, connected using a 155 Mbs ForeRunnerLE 155 ATM switch. The first machine is a 6 processor UltraEnterprise 4000 with 1.5 Gbyte memory. The *master* process was always run on this machine. The three other machines are 4 processor UltraEnterprise 3000s, with 0.5 Gbyte memory each. All the machines use 250 MHz UltraSparcII processors with 4 Mbyte cache.

The program parameters and the memory requirements for the sequential version are shown in Table 1. It can be seen in each case that the memory requirements do not exceed the capacity of any one node.

3.7. Runtime Versions

The *Strings* runtime was modified to demonstrate the incremental effect of different design de-

Program	Parameters	Size
FFT	1048576 points	54 Mbyte
LU-c	2048 × 2048, block size 128	37 Mbyte
LU-n	1024 × 512, block size 32	13 Mbyte
RADIX	1048576 integers	7.5 Mbyte
RAYTRACE	balls	9 Mbyte
VOLREND	head, 4 views	27 Mbyte
WATER-n2	4096 molecules, 3 steps	8.3 Mbyte
WATER-sp	4096 molecules, 3 steps	37 Mbyte
MATMULT	1024 × 1024 doubles, 16 blocks	29 Mbyte
MRI	PHANTOM image, 14 frequency points	15 Mbyte
SOR	2002 × 1002 doubles, 100 iterations	13 Mbyte
TSP	19b	5 Mbyte

Table 1
Program Parameters and Memory Requirements

cisions. The following runs were carried out:

S *Single Application Thread:* sixteen processes, four per machine, with a single application thread per process. User level threads are used throughout. The `dsm_server` thread handles one request at a time, and the network I/O is interrupt driven. This approximates typical existing DSMs that do not support multiple application threads like TreadMarks, which has been studied on ATM networked DECstation-5000/240s [4].

M *Multiple Application Threads:* four processes, one per machine with four application threads per process. This case is similar to other DSMs that allow multiple application threads but are restricted to using user level threads, eg. *CVM* results were presented on a cluster of SMP DEC Alpha machines [18]. This was approximated by setting the thread scheduler to only allow process level contention for the threads. These were then constrained to run on a single processor per node.

K *Kernel Threads:* This version allows the use of kernel level threads that can be scheduled across multiple processors on an SMP node.

C *Concurrent Server:* The `dsm_server` thread now creates explicit handler threads so that multiple requests can be handled in parallel.

P *Polled I/O:* A `communication_thread` waits on message arrivals and notifies the other threads on node. The overhead of generating a signal and switching to the user level signal handler are thus avoided.

B *Summing Barrier:* Instead of each application thread sending an arrival message to the barrier manager, the arrivals are first collected locally, and then a single message is sent.

Release Consistency model was used in each case.

4. Experimental Results and Analysis

The overall speedup results are shown for each version on the runtime in Figure 3. The time measured in this case excludes the initialization times of the application programs.

To better analyze the performance behavior due the various design choices, the overall execution times are split up into components in Figure 4. The results are normalized to the total execution time of the sequential case. The data is separated into the time for:

Page Fault: the total time spent in the page-fault handler.

Lock: the time spent to acquire a lock.

Barrier Wait: the time spent waiting on the bar-

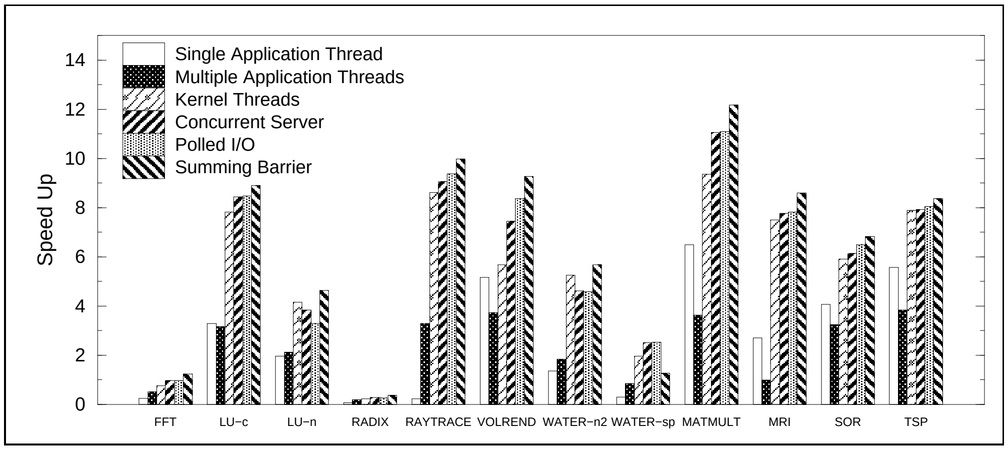

Figure 3. Speed-Up Results for Benchmark programs

rier after completing Release Consistency related protocol actions.

Compute: this includes the compute time, as well as some miscellaneous components like the startup time.

All times are wall clock times and thus include time spent in the operating system.

4.1. Single Application Thread

If multiple processes are used on a node, only programs with very little false sharing and communication are seen to provide speedups. These include VOLREND, MATMULT, and TSP. On the other hand, many programs have a significant slowdown, including FFT, RADIX, RAYTRACE, and WATER-sp. From Figure 4, it can be seen that a significant part of the execution time is spent on page faults and synchronization. Since each process on an SMP node executes in its own address space, the pages have to be faulted in separately. This leads to an increase in network contention as well as disruption at the server side.

4.2. Multiple Application Threads

The use of multiple application threads significantly affects the number of page faults incurred per thread, as seen in Figure 5. Since the threads on a single SMP node share the same memory space, a page that is required by multiple threads has to be faulted in only once. The very high improvement for MRI is due to the use of a workpile model. Essentially, the first thread does all the work, and incurs all the faults. This can be verified by looking at the high compute time for this program. From Figure 6, the time per page fault does not decrease as significantly as the total number of faults. This is a result of using user level threads, which are sequentialized on a single processor. When a user level thread has a page fault, the thread library will schedule another runnable thread. Once the page arrives, the faulted thread will be allowed to run only after the current thread is blocked in a system call, or its timeslice expires. This can also be seen in the increased compute time in programs like LU-c, LU-n, RAYTRACE, MATMULT, and TSP.

4.3. Kernel Threads

When the application threads are executed on top of kernel threads, the operating system can schedule them across multiple processors in an SMP. This is clearly seen in the reduction in overall execution time. Figure 7 shows that the time spent on each barrier decreases substantially when using kernel threads, since the application threads are no longer serialized on a single processor. This effect is particularly visible in programs with high computational components like LU-c, LU-n, RAYTRACE, WATER-n2, MATMULT, MRI, and TSP.

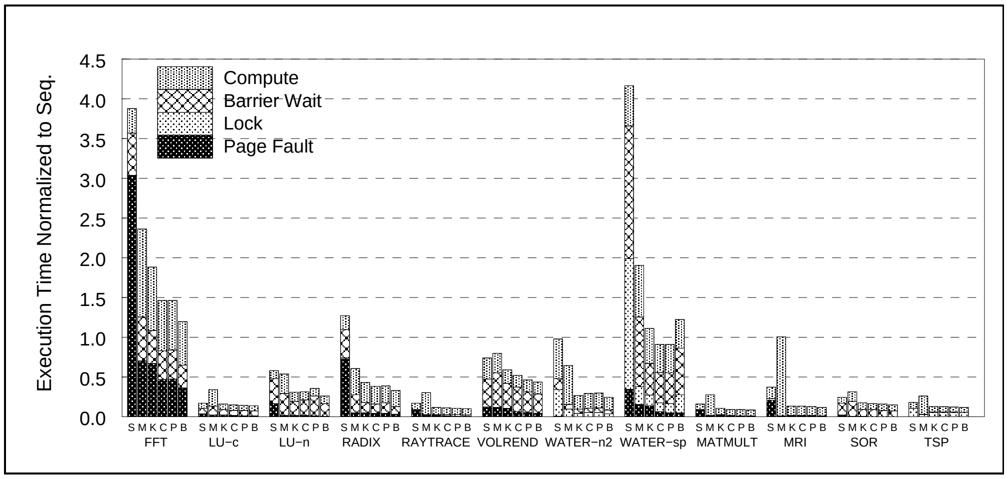

Figure 4. Execution time break-down for Benchmark programs

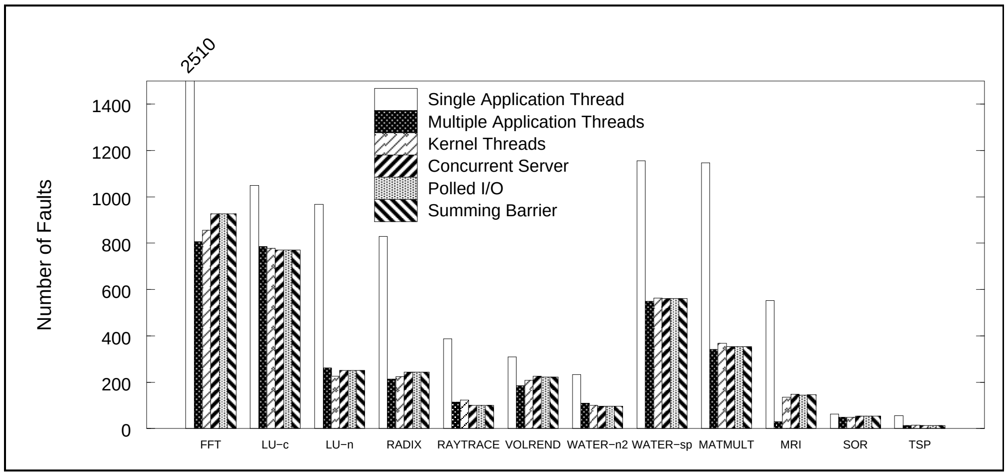

Figure 5. Average number of Page Faults

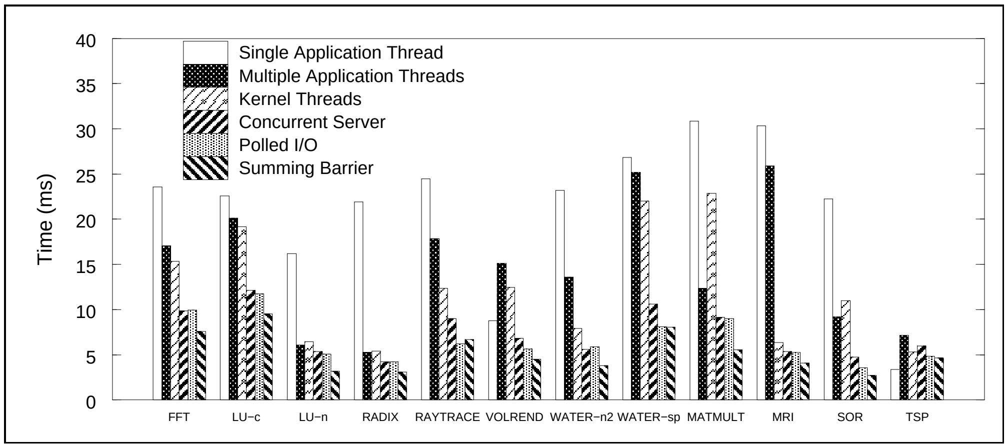

Figure 6. Average time taken for a Page Fault

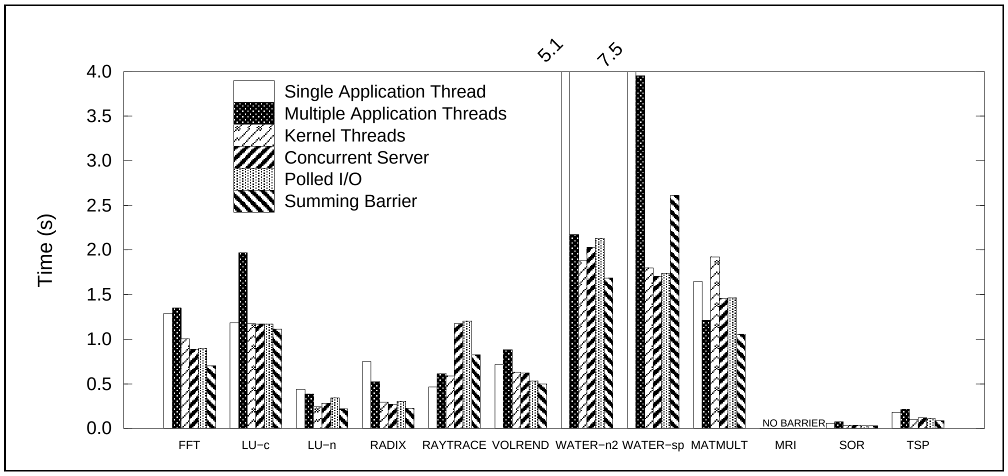

Figure 7. Average time spent per Barrier Call

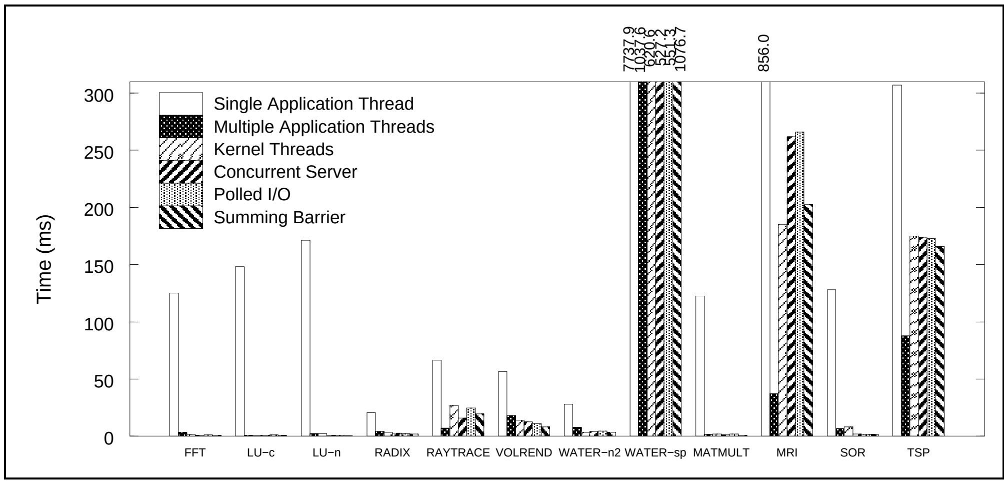

Figure 8. Average time required to acquire a Lock

4.4. Concurrent Server

Programs like WATER-n2 and WATER-sp have a high overhead due to locking of shared regions. This is effectively reduced by using the concurrent server, which allows requests for different locks to be served in parallel. The decrease is close to 50 % in case of WATER-sp, as seen from Figure 8. However in may of the programs tested, the overhead due to lock base synchronization is very low.

4.5. Polled I/O

The behavior of signal driven I/O compared to polled I/O can be explained by referring to Table 2. The overhead of signal generation becomes apparent as soon as the message size drops below 4 k bytes. For larger average packet sizes, as seen in FFT, LU-n, and RADIX, the signal driven I/O version performs as well if not better than polled I/O. The polled I/O provides visible benefits when the number of messages is small, and the packet size is moderate, as in RAYTRACE and VOLREND.

4.6. Summing Barrier

From Figure 7, collecting all the barrier arrivals locally reduces the time per barrier, by a factor of 10 – 20 % in most cases. This provides an additional source of performance improvement in the programs.

5. Related Work

When compared to message passing programs, additional sources of overhead for traditional soft-

Program	Number of Messages	Average Size (byte)	Latency (ms) Polled	Latency (ms) Interrupt
FFT	7894.13	4164.32	67.56	77.82
LU-c	5419.66	3336.62	44.64	208.08
LU-n	6045.73	4957.07	14.07	45.97
RADIX	2655.20	4225.83	20.33	34.67
RAYTRACE	784.66	2703.32	17.11	17.61
VOLREND	1736.40	2320.78	29.89	35.70
WATER-NSQUARED	97006.53	165.32	475.45	499.03
WATER-SPATIAL	4625.16	3694.65	414.78	380.93
MATMULT	2582.33	3311.59	22.61	34.26
MRI	1135.06	3448.16	3.00	3.12
SOR	1537.91	1480.63	16.09	17.38
TSP	588.33	947.28	7.45	9.36

Table 2
Communication characteristics (per node)

ware DSM systems have been identified to include separation of data and synchronization, overhead in detecting memory faults, and absence of aggregation [21]. Researchers have attempted to use compiler assisted analysis of the program to reduce these overheads. Prefetching of pages has been suggested by a number of groups for improving the performance of TreadMarks, by saving the overhead of a memory fault [17,8]. This technique sacrifices the transparency of a page oriented DSM, but can be incorporated in parallelizing compilers. In *Strings*, a faulting thread does not block the execution of other application threads on the same process, hence the benefit of prefetching is not expected to be very large. Asynchronous data fetching was also identified to be a source of performance improvement [22]. In our system, the dedicated `dsm_server` and communication thread together hide the consistency related actions from the application threads.

SoftFLASH [23] is a similar project, but uses kernel modifications to implement a SVM on an SMP cluster. In contrast, our implementation is completely in user space and thus more portable. Some other research has studied the effect of clustering in SMPs using simulations [19]. We have shown results from runs on an actual network of SMPs. HLRC-SMP is another DSM for SMP clusters [24]. The consistency model used is a modified version of invalidate base lazy release consistency. They do not use a threaded system since they claim that it leads to more page invalidations in some irregular applications. *Strings* uses an update based protocol, and it is not clear whether the same results can be applied. Cashmere2L exploits features found in the DEC MemoryChannel network interface to implement a DSM on a cluster of Alpha SMPs [25]. Our system is more general and provides good performance even with commodity networks. We have observed similar speed-up results with a hub based FastEthernet network [26]. Brazos [12] is another DSM system designed to run on multiprocessor cluster, but only under Windows NT. The *Strings* runtime on the other hand is very portable and has currently been tested on Solaris 2.6, Linux 2.1.x, and AIX 4.1.5.

6. Conclusions

Though the performance of each implementation can be seen to depend on the data sharing and communication pattern of the application program, some general trends can be observed. It is found that using multiple processes on SMP nodes provides good speedups only in programs that have very little data sharing and communication. In all other cases, the number of page faults is very high, and causes excess communication. Multiple application threads can improve the performance in some cases, by reducing the number of page faults. This is very effective when there is a large degree of sharing across the threads in a node. However, the use of user level threads causes an increase in computation time and response time, since all the threads compete for CPU time on a single processor. If kernel threads are used additionally, the overall performance improves significantly in all the programs tested. Using a dedicated `communication` thread to poll for incoming messages is a preferred alternative to signal driven I/O. The concurrent `dsm_server` approach reduces the latencies for page-faults by allowing multiple requests to be handled concurrently. Finally, using a hierarchical summing barrier improves the barrier wait times in most of the programs.

Overall, using kernel threads is very promising, especially for regular programs with little false sharing. Additional work needs to be done to identify the sources of overhead in the barrier implementation, since this dominates the execution time in the cases where the overall results are not that good. Our current work is to improve the performance of the release consistency protocol.

Acknowledgments

We would like to thank John Carter and Dilip Khandekar for putting the Quarks source code in the public domain. This allowed us to concentrate our efforts on developing a *multithreaded* DSM. We thank the anonymous reviewers, whose helpful comments shaped an earlier version of this paper. We also thank Padmanabhan Menon for porting the MRI code to *Strings*.

References

[1] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems*, vol. 7, pp. 321–359, November 1989.

[2] B. D. Fleisch and G. Popek, "Mirage: A Coherent Distributed Shared Memory Design," in *Proceedings of the ACM Symposium on Operating System Principles*, (New York), pp. 211–223, ACM, 1989.

[3] 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.

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

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

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

[7] 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.

[8] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel, "Compiler and Software Distributed Shared Memory Support for Irregular Application," in *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming*, 1997.

[9] B. N. Bershad and M. J. Zekauskas, "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors," Tech. Rep. CMU-CS-91-170, Carnegie Mellon University, Pittsburgh, PA 15213, September 1991.

[10] J. B. Carter, "Design of the Munin Distributed Shared Memory System," *Journal of Parallel and Distributed Computing*, 1995.

[11] 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.

[12] E. Speight and J. K. Bennett, "Brazos: A Third Generation DSM System," in *Proceedings of the First USENIX Windows NT Workshop*, August 1997.

[13] K. Thitikamol and P. Keleher, "Multi-threading and Remote Latency in Software DSMs," in *Proceedings of the 17th International Conference on Distributed Com-*

puting Systems, 1997.

[14] 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.

[15] E. W. Felten and D. McNamee, "Improving the Performance of Message-Passing Applications by Multithreading," in *Proceedings of the Scalable High Performance Computing Conference*, pp. 84–89, April 1992.

[16] S.-Y. Park, J. Lee, and S. Hariri, "A Multithreaded Message-Passing System for High Performance Distributed Computing Applications," in *Proceedings of the IEEE 18th International Conference on Distributed Systems*, 1998.

[17] R. Mirchandaney, S. Hiranandani, and A. Sethi, "Improving the Performance of DSM Systems via Compiler Involvement," in *Proceedings of Supercomputing 1994*, 1994.

[18] P. Keleher and C.-W. Tseng, "Enhancing Software DSM for Compiler-Parallelized Applications," in *Proceedings of International Parallel Processing Symposium*, August 1997.

[19] 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.

[20] P. Menon, V. Chaudhary, and J. G. Pipe, "Parallel Algorithms for deblurring MR images," in *Proceedings of ISCA 13th International Conference on Computers and Their Applications*, March 1998.

[21] A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel, "Evaluating the Performance of Software Distributed Shared Memory as a Target for Parallelizing Compilers," in *Proceedings of International Parallel Processing Symposium*, April 1997.

[22] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System," in *ASPLOS-VII Proceedings*, vol. 24, (Cambridge, Massachusetts), pp. 186–197, ACM, October 1996.

[23] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy, "SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory," in *ASPLOS-VII Proceedings*, vol. 24, (Cambridge, Massachusetts), pp. 210–220, ACM, October 1996.

[24] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh, "Home-based Shared Virtual Memory Across SMP Nodes," in *Proceedings of the Fourth International Symposium on High Performance Computer Architecture*, 1998.

[25] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott, "CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network," in *Proceedings of the ACM Symposium on Operating System Principles*, (Saint Manlo, France), October 1997.

[26] S. Roy and V. Chaudhary, "Evaluation of Cluster Interconnects for a Distributed Shared Memory," in *Proceedings of the 1999 IEEE Intl. Performance, Computing, and Communications Conference*, pp. 1 – 7, February 1999.