Implementing Treadmarks over GM on Myrinet: Challanges, Design Experience, and Performance Evaluation *

Ranjit Noronha and Dhabaleswar K. Panda

Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210
{noronha,panda}@cis.ohio-state.edu

Abstract

Software based DSM systems have traditionally not performed well compared to message passing applications because of the high overhead of communication associated with traditional stack based protcols. Furthermore DSM based packages like Treadmarks use UDP, with mechanisms for retrasmissions and ordered delivery and correspondingly higher overhead. Modern interconnects like Myrinet offer reliable message delivery with very low communication overhead through user level protocols. This paper examines the viability of implementing a thin communication substrate between Treadmarks and GM the rationale being that a layer tuned to the needs of the application would offer better performance and scalabilty as opposed to a generic UDP layer. Tradeoffs for various design alternatives for buffer management, connection setup, advance posting of descriptors and asynchronous messages are discussed. We have implemented the best of these strategies in a layer that is bound to Treadmarks at compile time. Results from micro-benchmarks and applications are presented. These results show that not only does the specialized implementation perform better, it also exhibhits better parallel speedup and scalability. A reduction in total application execution time of upto 6 for a sixteen node system is demonstrated in comparision with the original implementation. The implementation also exhibits superior scaling properties as the application size is increased. This implementation now provides the testbed for the examination of communication strategies for DSM based systems.

Keywords: *Clusters, DSM, User Level Communication, Programmable NIC, Myrinet*

*This research is supported in part by NSF grants #CCR-0204429 and #EIA-9986052

1 Introduction and Motivation

Network of Workstations (NOWs) are becoming increasingly popular for providing cost-effective and affordable high-performance computing for a wide range of applications. Such environments consist of clusters of workstations connected by Local Area Networks (LANs). In the past the nodes were connected through Ethernet which offered a peak bandwidth of 100 Mbps and a small message latency of the order of 100 microseconds. Internode communication was therefore an expensive operation and minimizing communication while desigining applications for cluster based applications was one of the main goals. There has been a lot of research conducted in the development of Softare Based Distributed Shared Memory (DSM) Systems [17, 12, 8]. DSM systems such as Treadmarks [13, 5] which provided an easy application development environment as opposed to a Message Passing Interface(MPI) such as the MPI implementation on Myrinet [3, 15] did not catch on primarily because they are communication intensive and as a result exhibited poor performance. Increasingly a variety of interconnects offering low latency and high bandwidth have been developed. Communication interconnects like Gigabit Ethernet [1], Myrinet [10] and Infiniband [2] can deliver a peakbandwidth of 1, 2 and 10 GBps respectively and a latency under 10 microseconds for small messages. These developments have significantly reduced the communication costs. In this paper we revisit the question of whether a DSM based system could be a viable option for high performance application development in the light of advances in current networking technology. The next section delves into why it necesary to design a specialized communication protocol for a DSM based system.

It has been shown that having the operating system in the critical path of communication siginificantly reduces performance due to the latency of context switching and other overhead. This has fuelled the development of user-level communication protocol systems such as AM [11], VMMC [9], FM [14],U-Net [18, 19] , LAPI [16] and BIP [15]. GM [4] is a user-level communication protocol that runs over the popular Myrinet interconnect [10]. It gives reliable in-order delivery of packets with very low latency, and high bandwidth which makes it a suitable candidate for use with a DSM based system. However Treadmarks uses UDP, which forces us to use the sockets implementation over GM. This sockets implementation has been designed to work with any application and may contain inefficiencies and bottlenecks that will affect the performance and scalabilty of a DSM based system. It is therefore an open challenge whether suitable middleware can be deployed for software DSM systems like Treadmarks. To the best of our knowledge, there has not been any work on developing a middleware for supporting TreadMarks on top of GM for NOWs. In this paper we take on the challenge of designing such an efficient middleware for Treadmarks over GM. The rest of the paper is orgainzed as follows. The next section describes the implementation of Treadmarks. Following that is a description of the GM message passing layer. After that we discuss the challenges with designing and implementing a substrate between GM and Treadmarks. Following that we evaluate the performance of our design using various microbenchmarks and applications. Finally we end with conclusions and future directions.

1.1 Overview of Treadmarks

Treadmarks is a popular software DSM system which runs on NOWs without any modifications to the operating system kernel. Furthermore, it does not rely on any particular compiler. Treadmarks relies on user-level memory management techniques to provide coherency among participating processes. UDP is the communication protocol used. Due to the high overhead of UDP communication efforts have been made to reduce the amount of communication. In the next two sections we discuss the coherency protocol and the communication model and primitives used by Treadmarks.

1.1.1 Coherency Protocol

Shared memory accesses in Treadmarks are divided into two groups: ordinary accesses and synchronization accesses. Synchronization accesses are further divided into into acquire and release accesses. Treadmarks implements the *lazy release consistency protocol* (LRC) where propogation modifications are postponed till the time of *acquire* [13]. Page granularity is used in Treadmarks but the multiple writer protocol is also supported to reduce false sharing. Lazy diff creation is used where a dirty page and its twin are not compared till a request is made. This reduces the overhead to some extent.

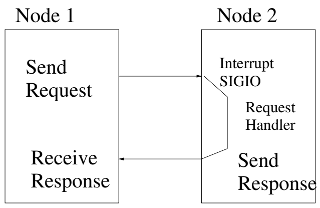

Figure 1. Request-Response Communication Model in Treadmarks

1.1.2 Communication Model and Primitives

Treadmarks relies on a request-reply type of communication. As shown in Figure 1, Request messages (such as a request for page diffs) are sent out by using socket function calls. Upon arrival of a Request message at a node, an interrupt is issued and after the message has been processed by the kernel, the SIGIO signal is raised. The SIGIO signal handler then processes the Request message and sends a Response message if necessary (say the requested diff). It is possible that the Request message may get forwarded to another node to prepare and send the response. Whenever a response is expected, the node which has sent the Request message waits until the Response message is received. Then the received response message is processed (say, by incorporating a page diff). The communication services required by Treadmarks can be divided into four major groups: sending Request messages, sending Response messages, receiving Response messages. These services along with the corresponding UDP/TCP function calls are shown in Figure 2. Select is mainly used for finding out if a request message has been received from any any node. Treadmarks takes advantage of the operating system feature that allows for user level signal to be provided and the fact that completion of receive operations on sockets raises a SIGIO signal. Since Treadmarks was developed on top of UDP/TCP primitives, the communication services required by Treadmarks matches well with the communication services provided by the UDP/TCP services. Examples are ability to send from any user buffer, automatic buffer allocation for temporary receives and ability to receive a message from a group of nodes. As we discuss the GM communication layer in the next section, the mismatch between GM primitives and UDP primitives will become apparent.

1.2 GM over Myrinet

Myrinet offers low latency and high bandwidth communication. The switch fabric is based on worm-hole routed cross-bar technology. The link which are full duplex can deliver at the rate of either 1.28+1.28 or 2+2 gigabits per

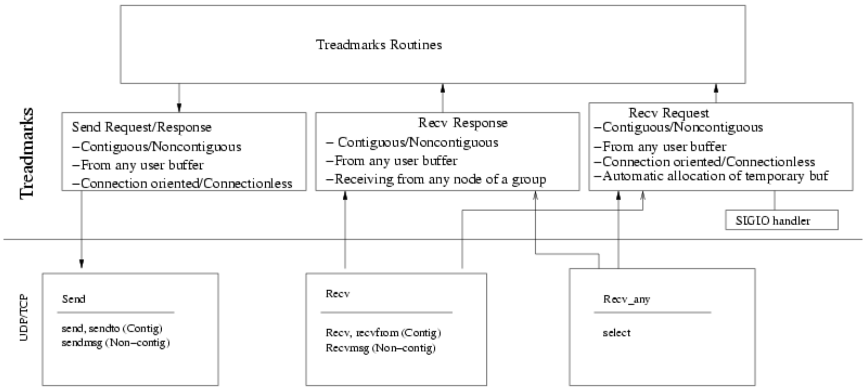

Figure 2. Four major groups of communication services required by Treadmarks and their implementation using UDP/TCP communication primitives

second. The Myrinet NIC is programmable with a LANai processor running at speeds up to 200 MHz and equipped with upto 4MB SRAM. Applications may directly access the NIC bypassing the OS and the associated overhead.

GM is a user-level protocol which runs over the Myrinet network. It has three componenets namely a kernel module, a library and a Machine Control Program(MCP). The kernel module or driver carries out mainly initilizaation functions such as loading the MCP, creating ports and pinning and unpinning memory. It is not in the critical path of communication. A port is a data structure allowing an application to communicate with the Myrinet network through the interface on the host namely the NIC.

GM provides reliable, connectionless data delivery services to the user. The user copies his message into a DMAable or pinned area of memory and passes it to GM. GM then transmits it to the receiver. The receiver then sends an ACK back to the sending NIC which places a callback in the receive queue of the sender. The sender on receiving the callback may reuse the send buffer. GM provides a limited number of *send tokens* to the user. The user is required to provide receive tokens to GM so that it may receive messages. The send and receive tokens together enforce a form of flow control between the application and the NIC.

GM uses the concept of size to control the reception of messages. For a message of length l it may only be received into a buffer of size $\log_2(l+2)$. This makes buffering in GM more complicated. Normally the application should provide at least two buffers for every size expected to allow for communication pipelining. In reality the application would provide a lot of small buffers for the smaller sized messages. The user needs to be careful about providing receive buffers in a bounded amount of time to avoid deadlock of the sending port. GM does not provide any form of asynchronous notification, and the user is instead required to poll for messages. GM also does not offer any scatter/gather operations.

2 Challenges in Designing the Communication Substrate

This section delves into the communication requirements of Treadmarks followed by the disparity in functionality offered by GM. Following that is a discussion of the components involved in bridging this gap and the design alternatives for each of these components. Some of these issues were encountered while implementing Treadmarks over VIA [7, 6].

2.1 Major Issues

Lets look closely at the requirements and characteristics of the services offered by GM and compare them with

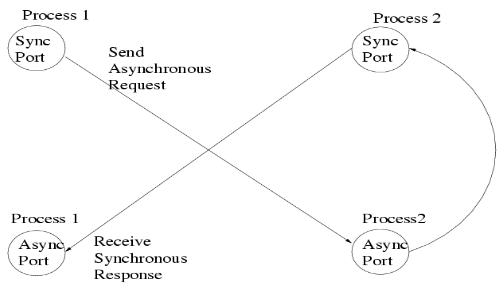

Figure 3. Port setup between 2 processes in Treadmarks

the communication requirements of Treadmarks. The major mismatches are :

- The communication model of Treadmarks on top of UDP as shown in Figure 1, handles Request and Response messages differently. In particular, it can be seen that while Request messages arrive in an asynchronous manner, Response messages are exchanged in a synchronous fashion. GM does not provide any mechanism for handling asynchronous messages such as a notify message or an interrupt. The application in required to continously poll the receive queue for an incoming message. This makes implementing the request / response mechanism much harder when compared to traditional UDP. A way out is to implement a polling thread at the application layer which generates a signal when a request comes in. However this is an expensive solution which is extremely CPU intensive and robs the application of processing resources. Another solution is to modify GM to generate an interrupt on a particular port when a message is received on that port. A minimal bound on the reponse time to an asynchronous application is critical to the overall performance of the application.

- In the UDP protocol, temporary buffers are automatically allocated on the arrival of messages. The user may examine these buffers later on. GM however requires the user to prepost a receive descriptor before the arrival of a message. Failure to do so within a bounded amount of time (30 seconds) fires a timer in the sender, which returns a failure code in the form of a callback for the particular send. This drastically increases application execution time and has to be avoided at all costs. This problem is complicated by the fact that Treadmarks often disables interrupts for consistency reasons which may result in the asynchronous buffers filling up for the interrupt driven implementation. The the sending port is disabled, forcing the sender to reenable the port before attempting another send. This is an expensive operation requiring GM to probe the network. Worse yet the receiving port may deadlock, preventing the application from running any further.

- Memory used for communication in GM has to be locked down in memory before the communication commences. This eliminates the need for copying data between the user buffers an intermediate kernel buffers typically used in traditional network transports. No such memory registration is needed by UDP.

- In traditional UDP, a message of any length may be received into a particular buffer irrespective of the length of the allocated buffer. The only requirement is that the buffer should be large enough to accommodate the incoming message. Treadmarks takes advantage of this by making sure that there is a buffer large enough to receive the biggest mesage which can be sent. GM uses the concept of *size* to decide the buffer into which a message of length l may be received where *size* is the smallest integer less than or equal to $\log_2(l{+}2)$. This complicates matters since now there potentially has to be a buffer for each possible size of message that may be received.

- Treadmarks uses two ports between every process as shown in Figure 3 and discussed in Section 1.1.2; one for sending and receiving requests (asynchronous) and another one for sending and receiving replies (synchronous). GM offers connectionless reliable delivery and a maximum of eight ports which may be used out of which one of them is reserved for the mapper. That gives us only seven ports which may be used. This would entail mapping several connections to a particular port.

2.2 Components of the Substrate

Inorder to alleviate the above mismatch, the following architecture shown in Figure 4 is proposed. The main driving force behind proposing this architecture has been threefold: 1) not making any changes to the coherency protocol of Treadmarks or its communication model, 2) minimal changes to the GM layer, and 3) minimal modifications to the communication primitives of Treadmarks so that they can be interfaced with the new substrate.

The substrate can be broadly divided into four main components as shown in Figure 4; connection setup and management, receive buffer preposting and allocation, buffer

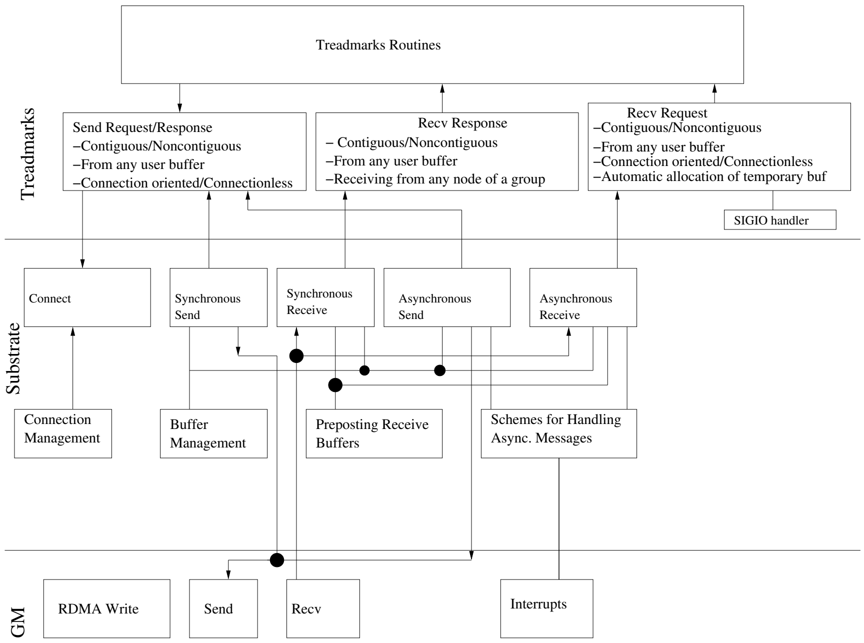

Figure 4. Components of the communication substrate and their relation to GM services and Treadmark requirements

management, and finally asynchronous message management techniques. These components bridge the gap between the requirements of Treadmarks and the inequities in the services provided by GM and result in five new primitives namely Connect, Synchronous send, Synchronous Receive, Asynchronous Send and Asynchronous receive. The design alternatives behind each of the four components and their cost-performance tradeoffs are discussed in detail in the following subsections.

2.2.1 Connection Management

As mentioned earlier, under the communication model of Treadmarks, the arrival of a request message at the receiver cannot be predicted. A request message also requires an interrupt to be generated. This interrupt need not be generated for a response message and the corresponding overhead can be avoided. Therefore we use one port for sending and receiving request and another port for sending and receiving the response. Furthermore to allow for an interface with the connectionless services of GM we multiplex multiple connections over the same two ports. The descriptors now returned by Connect are nothing more than the GM ID of the destination node. This elegant solution allows for much better scalability since we now only need two ports, no matter how many processes are present. This also not only helps us reduce the overhead by generating an interrupt only when a request comes in, but also helps us to seperate the buffering strategies required for the two different types of services.

2.2.2 Pre-posting of Receive Buffers

As discussed earlier in Section 2 we need to prepost receive buffers to guarantee that a message will be received and to avoid the possibilty of deadlock. GM differentiates messages based on size as discussed in Section 2 which allows us to optimize the number and size of receive buffers. We have to guarantee that for (n-1) processes there should be at least o*(n-1) buffers available at any given time if o outstanding messages are allowed at any point in time. However if we examine the situation more closely, we see that most asynchronous request are small typically of the order of eight bytes. We could then prepost a large number of buffers of size four (corresponding to eight bytes) to allow for GM communication pipelining. The minimum required would be n if only one outstanding message per process is required. The larger request messages are only required for sending asynchronous responses to a barrier upto size fifteen. Given the characteristics of the barrier (one message per process at the root node) we only need to provide (n-1) buffers (for n processes) for each of the larger sizes five to fifteen. This requires an allocation of 64K bytes for each of the (n-1) prcesses. For the synchronous case we may receive a response only after sending a request, which means we need to allocate a single buffer for each of the sizes (for a single outstanding request allowed per process). We use sizes from four to fifteen which give us a buffer requirement of approximately 64K. Totally the requrement is 72K bytes. For a system with 256 nodes the size is 16K x 256 + 64 K =

16 MB (approx) which is reasonable for most high-end systems today. For systems for which the allocation is too high, the sizes thirteen and above can eliminated and in its place a *rendezvous* protocol can be implemented, where a user first sends a message to the receiver to pin down a memory area larger than 8K before actually sending the message. This brings the total requirement down to 6MB for a 256 node cluster which is very reasonable. This however increases the communication overhead.

2.2.3 Buffer Management

GM requires that send and receive buffers be in registered memory regions. Registered memory regions contain memory memory pages which are pinned down in physical memory. Therefore, the size of registered memory is limited by the size of physical memory and the OS requirements on each node and can affect the performance of running applications.

Outgoing Request and Response messages are constructed by Treadmarks. By calling the send function in GM multiple times it is possible to send messages from non-contiguous buffers. In order to avoid extra data copies, Treadmarks is modified such that outgoing messages are constructed in registered memory regions. We did not follow this approach. Instead we used a pool of send buffers in the registered memory for outgoing messages and copied outgoing messages into these buffers before sending them. This also allows for pipeling synchronous sends in the case where a sender receives multiple request in a short span of time.

The incoming messages can be received into a buffer in registered memory before being processed. Since a pointer to incoming requests is passed to Treadmarks routines, request messages can be processed without any extra data copies. The Response messages are required to be copied from the buffer in which they have been received to TreadMarks data structures before being processed. This approach does require an extra data copy on the receive side but does not require any changes in Treadmarks. We used this approach. Alternatively, Treadmarks can be modified such that received Response messages can be processed without making any extra copies.

2.2.4 Handling Asynchronous Messages

Treadmarks makes use of the SIGIO signal handler to deal with asynchronous requests. GM does not provide a signal handler which we can use. To deal with lack of functionality we had four options; namely a timer which wakes up a thread intermittently to check for asynchronous messages, modifying GM to produce interrupts and finally a polling thread based approach. A detailed analyses of these approaches can be found in [7, 6]. The conclusion from these studies was that the interrupt based approach works best which was also the method adopted in this case.

3 Performance Evaluation

This section evaluates the performance of the proposed implementation. Since this is a system level study, the characteristics of the testbed affect the results and a brief description of the system is given. Following that we evaluate the susbstrate against UDP in terms of bandwidth and latency. Next a suite of micro-benchmarks for barriers, locks, diffs and pages is used to evaluate the proposed substrate and UDP. A suite of applications is then used to compare the performance of the substrate with that of UDP. The applications are used to evaluate the implementations both in terms of application size and system size.

3.1 Experimental Testbed and Setup

The experiments were conducted on a cluster of sixteen PCs interconnected with Myrinet and Fast Ethernet (100 Mbps) to connect it to the outside world. Each PC is equipped with four 700 MHz Pentium III processors with 1 GB of SDRAM and a 66 MHz/64-bit PCI bus. Linux 2.4.18 which has a SMP kernel is the operating system on all these machines. The Myrinet network runs at a speed of 2.0 Gbps and is connected by a low-latency, cut-through, crossbar switch through fiber links to LANAI 9 network cards. The experimental setup consists of Treadmarks running over the Myrinet implementation of Sockets (Sockets version 1.1) from Myricom referred to as UDP/GM and Treadmarks implemented over the proposed substrate referred to as FAST/GM in the rest of the paper. Latency and bandwidth tests were run for GM alone, UDP/GM and FAST/GM. For GM it was observed that the latency was 8.99 us for a message of 1 byte and the raw bandwidth was 235 Mbps for a message size of 32678. For FAST/GM the latency was 9.04 us and the bandwidth was 225 Mbps. For UDP/GM the latency was 11 us. However bandwidth could not be measured accurately because of the unreliable nature of UDP which was reflected in UDP/GM.

3.2 Microbenchmark level Evaluation

The microbenchmark-level evaluation was carried out by using the four microbenchmarks included in the Treadmarks distribution. These microbenchmarks are: *Barrier*, *Lock*, *Diff* and *Page*. These microbenchmarks measure the time required for performing basic Treadmarks operations. The Barrier microbenchmark reflect the time for performing a barrier across a set of nodes. In the Lock microbenchmark, the cost of acquiring a lock is measured. There are two versions of this microbenchmark: direct and indirect. The direct case reflects the situation where the lock being

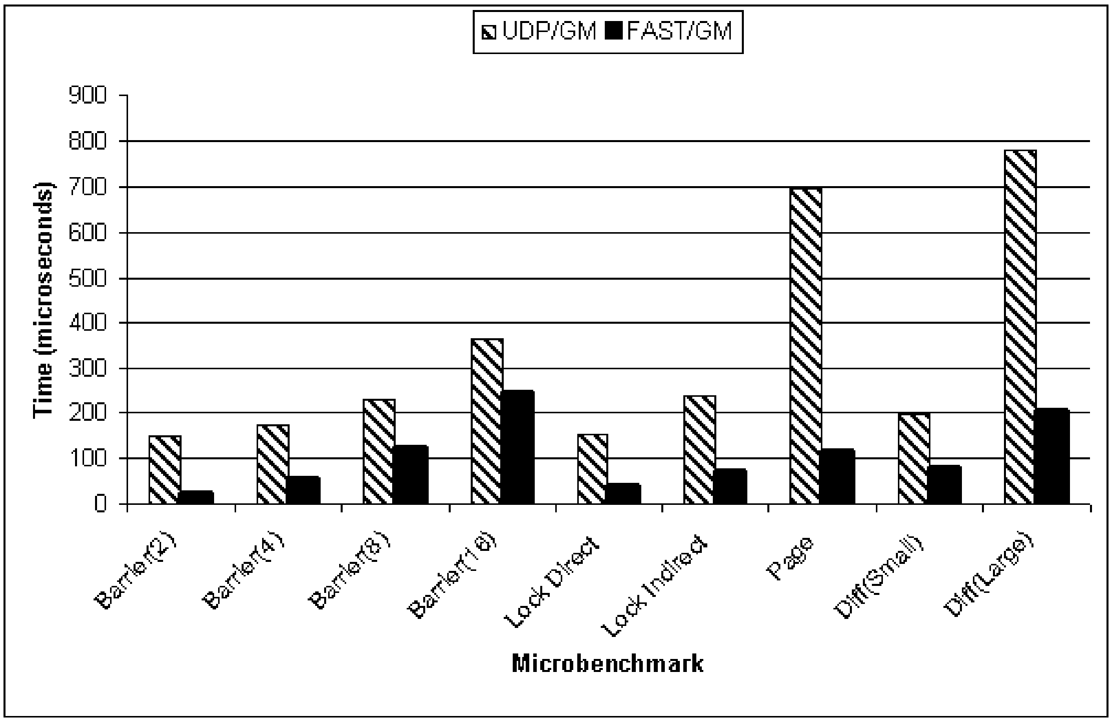

Figure 5. Performance results of four microbenchmarks(Barrier, Lock, Page and Diff). Different cases of Barrier, Lock and Diff are shown. Barrier (x) indicates the time to achieve a barrier on x nodes. For each of the microbenchmarks and their individual cases, the four bars (left to right) reflect the time on 2 different commuincation subsystems UDP/GM and FAST/GM

acquired has already been acquired and released by its managers node. The indirect case reflects the situation where the lock being acquired has been acquired and released by a third node.

The Page and Diff microbenchmarks are used to evaluate the performance of Treadmarks when shared memory is accessed and diffs are obtained and applied to a page. In the Page microbenchmark, a shared memory region consisting of mulitple memory pages is first created (by using Tmk_malloc) and then distributed among participating processes (by using Tmk_distribute) by process 0. After process 0 reads one word from each page, process 1 reads the same word from each page.

The Diff microbenchmark has two cases: small and large. In the first case, one word from each page is read by one process while the same words have been written into by another process earlier. The second case is similar to the first case with the difference being that all words of the shared memory region are accessed by the writer and reader processes.

Figure 5 presents performance results of four microbenchmarks and their different cases. In all cases FAST/GM outperforms UDP/GM. For the barrier operation FAST/GM outperforms UDP/GM by a factor of 1.5. For the indirect lock operation replacing UDP/GM with FAST/GM gives a factor of improvement of 3.85. For the direct lock case the factor improvement is 3.36. For the Page microbenchmark the factor of improvement is 6.0 while on the case of Diff the improvement is 3.80.

3.3 Application Level Evaluation

This section starts with a description of the applications used in the evaluation and then discusses the results. The section following that looks at the effect of system size on the performance of the applications. Finally the last section looks at the effect of application size on performance. It should be noted that no attempt was made to improve the performance by modifying the applications. These are the same applications from the original Treadmarks distribution.

3.3.1 Characteristics of Applications

We used four applications from Treadmarks distribution for evaluating our implementation. These applications are: Sor (red-black successive over-relaxation on a grid), TSP (traveling salesman problem), Jacobi and 3D FFT. By default Sor performs 10 iterations on a grid of 2000 x 1000 single-precision floating point numbers. TSP solves the traveling salesman problem for an 18-city tour. Jacobi uses a 1022 x 1022 grid of real numbers. By default, FFT works on a 32 x 32 x 32 array. In [6] a detailed study of the characteristics of the applications has been done. That study showed that Jacobi exclusively uses barriers for synchronization. On the other hand, Sor uses locks for synchronization more than any other application. TSP and 3D FFT mostly use locks and barriers respectively for synchronization. Jacobi has a high computation to communication ratio while 3D FFT exchanges a large volume of messages per unit time. The average size of exchanged messages in ascending order belongs to TSP, Jacobi, SOR, and FFT. The data exchange rates in ascending order belong to Jacobi, TSP, SOR and FFT.

3.3.2 Effect of System Size

To observe the effect of system size on application performance, the four applications were run on four, eight and sixteen nodes. Figure 6 illustrates the results in terms of execution speed for the four applications described above. In all cases our implementation FAST/GM outperforms the UDP/GM, often by a significant amount. For example in the case of Jacobi, the speedup is 6 for FAST/GM when going from four nodes to sixteen nodes while it is only 3.21 in the case of UDP/GM. Jacobi exhibhits a high computation to communication ratio, which explains its good scalability. Again for Sor the speedup for 16 nodes over 4 nodes is 2.5 for FAST/GM while it is only 1.12 for GM/UDP. 3D FFT and Tsp experienced a speedup of the order of 1.33 and 1.76 repectively while UDP/GM showed a slowdown. 3D FFT has a high communication to computation ratio which reduces overlapping and pipelining of communicaton and computation accounting for the lower speedups obtained. The significantly higher cost of obtaining a lock in the case of UDP/GM from Figure 5 and the fact that Sor uses locks extensively for global synchronization explains why there is a slowdown for SOR over UDP/GM. In the next section we will discuss the effect of application size on performance.

3.4 Effect of Application Size

The application execution times on a 16 node system for different problem sizes are shown in Figure 7. The problem sizes (A x B) used for Sor were 2000 x 1000, 2000 x 2000, 2000 x 3000, and 2000 x 4000. The X-axis reflects the changes in B. For TSP different problem sizes for 16, 17 and 18-city tours were used. For Jacobi problem sizes (A x B) used were 500 x 500, 1000 x 1000, 1500 x 1500 and 2000 x 2000 were used where B is the variable along the X-axis. Problem sizes used for 3D FFT (A x A x A) were 8 x 8

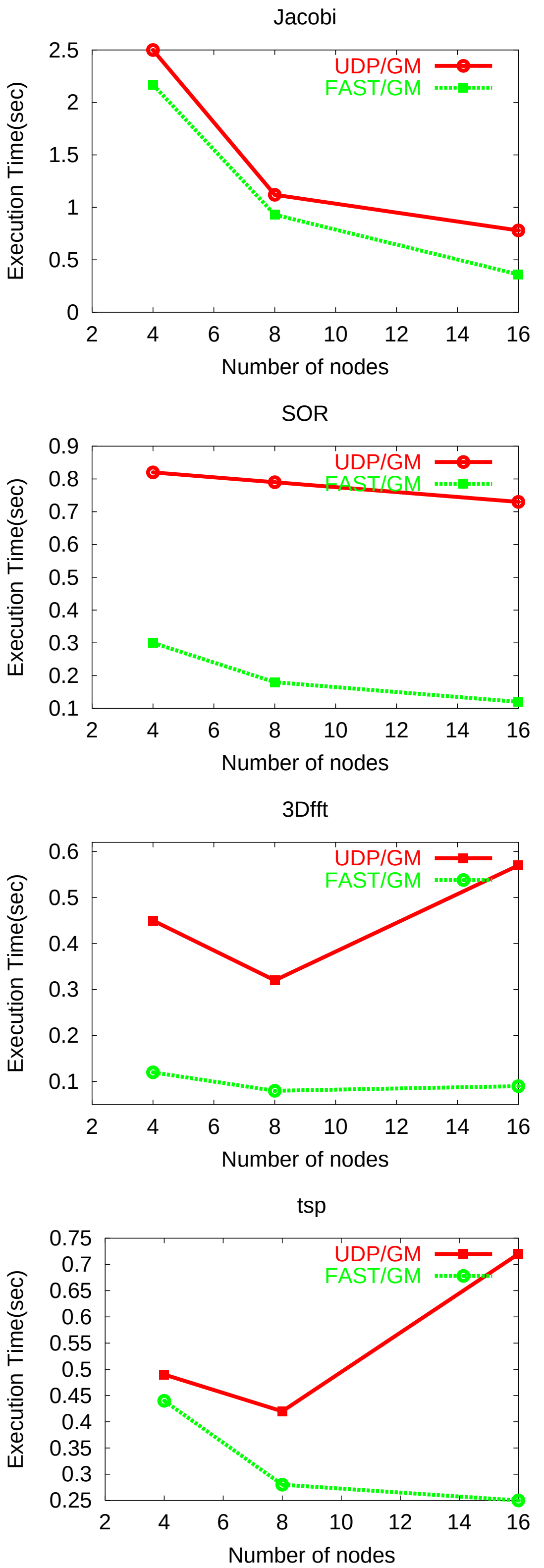

Figure 6. Execution times for four applications on two different communication subsystems as the number of nodes increases from 4 to 16

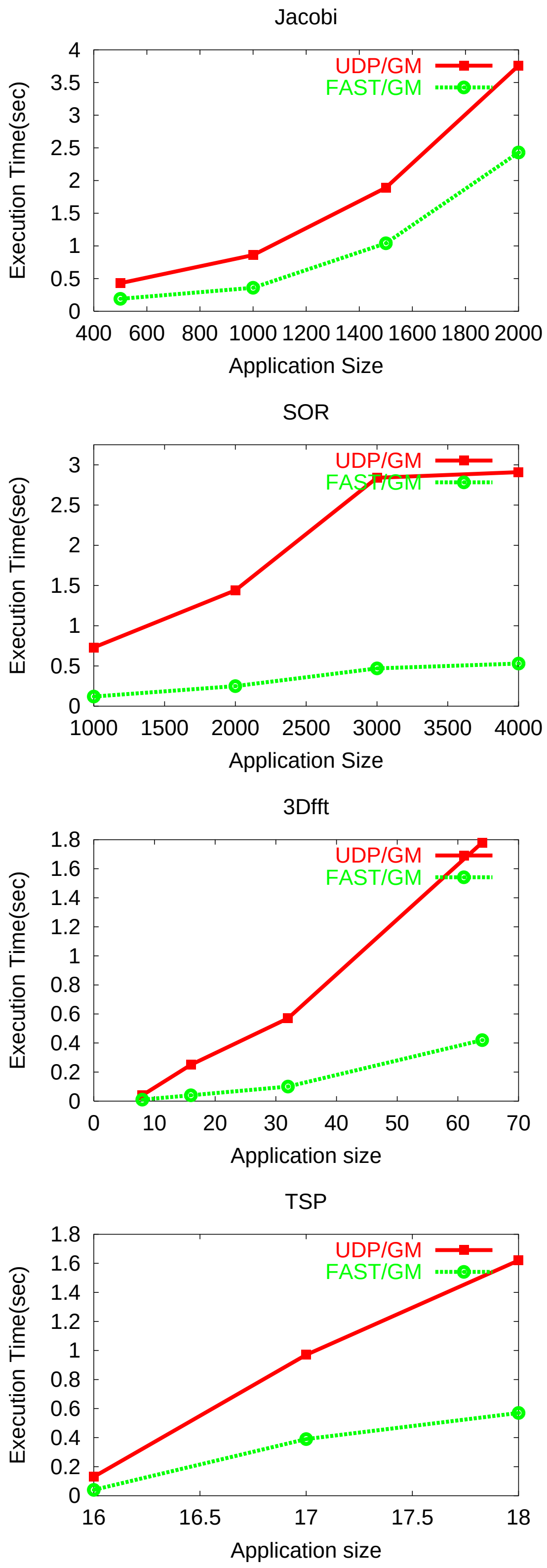

Figure 7. Execution times for four applications for different problem size sets. See the text for details of the size representation

x 8, 16 x 16 x 16, 32 x 32 x 32 and 64 x 64 x 64 where A is the variable along the X-axis. When UDP/GM is replaced with FAST/GM, 3D FFT shows a factor of improvement of 4.34. For Jacobi the factor of improvement is 1.54. For Sor an improvement of 5.5 is obtained. Finally for Tsp the obtained speedup is 2.84. Also in all cases the graphs show an increasing seperation between the curves as the application size increases. This is particularly prominent in the case of 3D FFT. For the application which has the highest computation to communication ratio the deviation between the curves is apparent indicating that any application will benefit from an efficient communication sub-system. These trends together with the observed parallel speedup in Section 3.3.2 clearly show the scalability of the proposed subsystem.

4 Related Work

This section discusses other research with a relationship to this work. A comparision between the performance of PastSet Software DSM system using TCP/IP and VIA is discussed in [8]. In this it is shown that by replacing TCP/IP by the MVIA implementation of VIA improves the performance of a few microbenchmarks. The author indicates that due to problems with the MVIA implementation they haven't been successful in designing and implementing the complete system. A few issues involved in taking advantage of low-latency high-bandwidth communication layers in Software DSM Systems are discussed in [8]. The communication system used in this work is Fast Messages (FM) on Myrinet. In this work, a new mechanism called MultiView for providing small-size pages is proposed for avoiding false sharing, reducing the size of messages, and preventing excessive buffer copying. Methods for customizing network drivers for providing efficient support for asynchronous messages is discussed for the Windows-NT operating system. [6] discusses an implementation of Treadmarks over VIA.

5 Conclusions and Future Work

This paper discussed the challenges involved in implementing Treadmarks over a low-level substrate like GM. Different tradeoffs and design issues were discussed and the best set of features were choosen for implementation. Different micro-benchmarks were run to validate the objective of the implementation; namely that Software DSM based Systems can perform better by reduction in communication overhead. Siginificant speedup was observed for the micro-benchmarks. The implementation was then evaluated in terms of both system and application size for four different applications in the Treadmarks suite. Parallel speedup was obtained in all cases when system size was increased for our implementation up to a maximum of 6 when going from a four node to a sixteen node run while the maximum speedup obtained for the original UDP implementation was only 3.21. For two of the applications TSP and 3D FFT there was a slowdown for the original UDP over Myrinet implementation as system size was increased. As application size increased our implementation showed improved performance upto 4.34 over the UDP over Myrinet implementation. Also the rate of increase of execution time in our implementation is much smaller than that of UDP over Myrinet. All this clearly exhibhits the scalability of our implementation.

There are many directions for future work. With this implementation it will now be possible to experiment and tune different parameters in the communication layer such buffer distribution to obtain the best performance for an application. Futhermore profiling across a number of different applications will allow us to determine the most common message sizes. This will enable us to further optimize the critical path for the most common message lengths. To improve peformance even futher we can remove send callback processing from the critical path of communication. This can be done by having a daemon wake up and process the send callbacks when our supply of send tokens falls by half. This will also help to improve scalability. We are studying techniques for scaling a DSM system to a cluster having 256 and higher number of nodes. This would require further optimization to communication. One technique would be to push as much functionality as possible down to the NIC. This would help us process messages in the network at the NIC level, eliminating accesses across the PCI bus, reducing contention and freeing up resources on the host, further improving performance. Locks and barriers are primitives that are widely used for synchronization and are amenable to a NIC based implementation. They exchange a large number of messages both large and small. Pushing them down to the NIC will dramatically improve the performance of the application. We are also exploring techniques for NIC based implementations of diff processing. Finally Infiniband connected networks offer very high bandwidth of the order of 10 Gbps and low latency of 8 us. This introduces a whole new dimension for optimizations given the resource rich nature of the network. We will be exploring the design issues for implementing DSM over the Infiniband architecture.

References

[1] Gigabit ethernet: From 100 to 1000 mbps.

[2] Infiniband trade association. *www.infinibandta.com*.

[3] Mpich-gm software. *www.myrinet.com/scs*.

[4] The GM API. http://www.myri.com/scs/.

[5] C. Amza, A. Cox, et al. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer, 29(2):18-28*, feb 1996.

[6] M. Banikazemi, J. Liu, . K. Panda, and P. Sadayappan. Implementing treadmarks over via on myrinet and gigabit ethernet: Challenges design experience and performance evaluation. *Int'l Conference on Parallel Processing*, sep 2001.

[7] M. Banikazemi, D. K. Panda, and P. Sadayappan. Implementing TreadMarks on Virtual Interface Architecture (VIA): Design Issues and Alternatives. In *Ninth Workshop on Scalable Shared Memory Multiprocessors, held in conjunction with ISCA*, June 2000.

[8] J. Bjoerndalen, O. J. Anshus, B. Vinter, and T. Larsen. Comparing the performance of the pastset distributed memory system using tcp/ip and m-via. *The Second International Workshop on Software Distributed Shared Memory*, 1995.

[9] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 96–107, April 1991.

[10] N. J. Boden, D. Cohen, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[11] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware-Software Approach*. Morgan Kaufmann, March 1998.

[12] A. Itzkovitz, A. Schuster, and Y. Talmor. Harnassing the power of fast low-latency networks for software dsms. *The First Workshop on Software Distributed Shared Memory*, 1999.

[13] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, Jan. 1994.

[14] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.

[15] L. Prylli and B. Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. *In Proceedings of the International Parallel Processing Symposium Workshop on Personal Computer Based Networks of Workstations*, 1998. http://lhpca.univ-lyon1.fr/.

[16] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and experience with lapi - a new high performance communication library for the ibm rs/6000 sp. *International Parallel Processing Symposium*, March 1998.

[17] R. Stets, S. Dwarkadas, L. Kontothanassis, U. Rencuzogullari, and M. L. Scott. The Effect of Network Total Order, Broadcast, and Remote-Write Capability on Network-Based Shared Memory Computing, 2000. accepted for presentation.

[18] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, 1995.

[19] M. Welsh, A. Basu, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proceedings of Hot Interconnects V*, Aug. 1997.