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

Ranjit M. Noronha and Dhabaleswar K. Panda

Network-Based Computing Laboratory Computer and Information Science
The Ohio State University
Columbus, OH 43210
{noronha, panda}@cis.ohio-state.edu

January 7, 2003

Technical Report
OSU-CISRC-12/02-TR32

Implementing TreadMarks over GM on Myrinet: Challenges, 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 like TreadMarks have traditionally not performed well compared to message passing applications because of the high overhead of communication associated with traditional stack based protocols like UDP. 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 Myrinet GM, the rationale being that a layer tuned to the needs of the application would offer better performance and scalability as opposed to a generic UDP layer. Trade-offs 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 show that not only does the specialized implementation perform better, it also exhibits better parallel speedup and scalability. A reduction in total application execution time of up to a factor of 6.3 for a 16 node system is demonstrated in comparison with the original implementation. The implementation also exhibits superior scaling properties as the application size is increased.

1 Introduction

Clusters 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. Inter node communication was therefore an expensive operation and minimizing communication while designing applications for cluster based applications was one of the main goals. There has been a lot of research conducted in the development of Software Distributed Shared Memory (SDSM) Systems [17, 12, 7]. SDSM systems such as TreadMarks [13, 4] which provided an easy application development environment as opposed to the Message Passing Interface (MPI) standard did not catch on earlier primarily because the communication intensive nature of SDSM protocols led to poor performance on earlier cluster interconnects. Increasingly a variety of interconnects offering low latency and high bandwidth have been developed for clusters. Examples include Gigabit Ethernet [11], Myrinet [9] and InfiniBand [1] which can deliver a peak bandwidth 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 SDSM based system could be a viable option for high performance application development in the light of advances in current networking technology.

It has been shown that having the operating system in the critical path of communication significantly reduces performance due to the latency of context switching and other overhead. This has fueled the development of user-level communication protocol systems such as AM [10], VMMC [8], FM [14], U-Net [18, 19], LAPI [16] and BIP [15]. GM [3] is a user-level communication protocol that runs over the popular Myrinet interconnect [9]. It gives reliable in-order delivery of packets with very low latency, and high bandwidth. Thus, it is interesting to analyze whether Myrinet interconnect with GM communication layer is suitable for designing the next generation SDSM system. TreadMarks uses UDP, which forces us to use the inefficient sockets implementation over GM. To the best of our knowledge, there has not been any work on de-

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

veloping a middleware for supporting TreadMarks on top of GM for clusters which we attempt in this paper.

The rest of the paper is organized as follows. Section 1.1 describes the implementation of TreadMarks. Section 1.2 describes the GM message passing layer. Section 2 discusses the challenges with designing and implementing a substrate between GM and TreadMarks. Section 3 evaluates the performance of our design using various microbenchmarks and applications. Section 5 presents conclusions and future directions.

1.1 Overview of TreadMarks

TreadMarks [13] is a popular software DSM system which runs without any modifications to the operating system kernel. TreadMarks implements the *lazy release consistency protocol* (LRC) and 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 in TreadMarks to reduce the amount of communication. In the next section we discuss the the communication model and primitives used by TreadMarks.

1.1.1 Communication Model and Primitives

TreadMarks relies on a request-reply type of communication; Request messages are sent out 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 needed. 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. The communication services required by TreadMarks can be divided into three 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 1.

1.2 GM over Myrinet

Myrinet offers low latency and high bandwidth communication. The switch fabric is based on wormhole routed crossbar technology. The links which are full duplex can deliver at the rate of either 1.28+1.28 or 2+2 gigabits per second. The Myrinet NIC is programmable with a LANai processor running at speeds up to 200 MHz and equipped with up to 4MB SRAM.

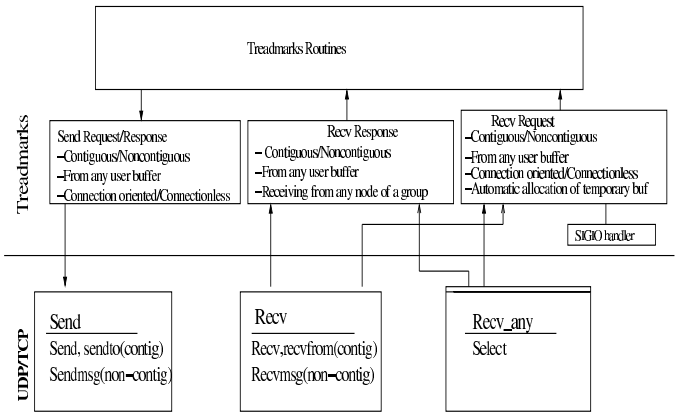

Figure 1. Three major groups of communication services required by TreadMarks and their implementation using UDP/TCP communication primitives

GM is a user-level protocol which runs over the Myrinet network. GM provides reliable, connectionless data delivery services to the user. GM transmits a message from pinned memory to the receiver. 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 [6, 5].

2.1 Major Issues

Lets look closely at the requirements and characteristics of the services offered by GM and compare them with the communication requirements of TreadMarks. The major mismatches are :

- The communication model of TreadMarks on top of UDP, receives Request messages asynchronously and Response messages synchronously. GM does not provide any asynchronous notification mechanism. 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 is required to continuously 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 response time to an asynchronous message 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 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 before the communication commences. This eliminates the need for copying data between the user buffers and 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 as long as it is big enough to hold the message. 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 message 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 discussed in Section 1.1.1; 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 mismatches, the architecture shown in Figure 2 is proposed. The main driving force behind proposing this architecture has been three-fold: 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 2; connection setup and management, receive buffer preposting and allocation, buffer management, and finally asynchronous message management techniques. The design alternatives behind each of the four components and their cost-performance trade-offs are discussed in detail in the following subsections.

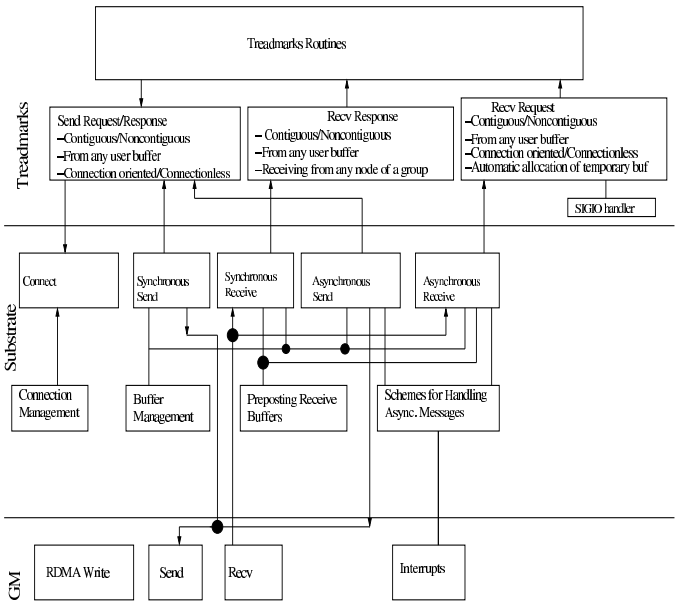

Figure 2. Components of the communication substrate and their relation to GM services and TreadMark requirements

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 separate 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 possibility 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. Since most asynchronous request are small typically of the order of eight bytes, we could 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-1) if only one outstanding message per process is required. The larger request messages are only required for sending asynchronous responses to a barrier. Given the characteristics of the barrier (one message per process at the root node) we only need to provide (n-1) buffers for each of the larger sizes five (corresponding to a maximum length of 24 bytes) to 15 (corresponding to a maximum length of 32K, which is the largest message TreadMarks could potentially send). This requires an allocation of 64K bytes for each of the (n-1) processes. 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 requirement is 64K*(n-1)+64K after combining the requirements of the synchronous and asynchronous cases. For a system with 256 nodes our systems memory requirement is 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 be 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 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 registered memory for outgoing messages and copied outgoing messages into these buffers before sending them. This also allows for pipelining 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 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 the lack of this functionality, we investigated three options; 1) a timer which wakes up a thread intermittently to check for asynchronous messages, 2) modifying GM to produce interrupts and 3) finally a polling thread based approach. A detailed analysis of these approaches can be found in [6, 5]. The conclusion from these studies was that the interrupt based approach works best which was also the method adopted in this case. The NIC firmware was modified to generate an interrupt whenever a message was received on the asynchronous port.

3 Performance Evaluation

This section evaluates the performance of the proposed implementation. We first describe the testbed used to run the experiments. Following that a suite of microbenchmarks 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. 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 running 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 Million bytes per second (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 reflects 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 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

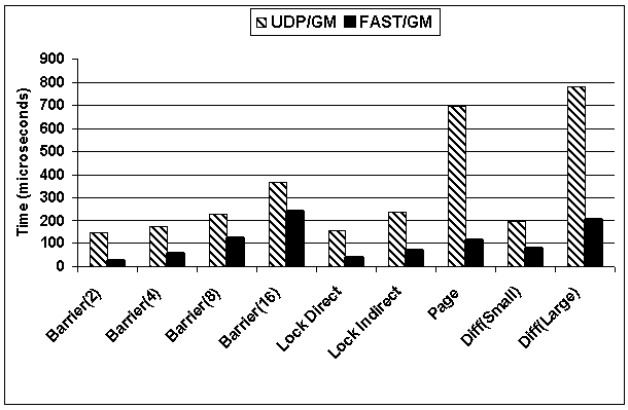

Figure 3. Performance results of four microbenchmarks (Barrier, Lock, Page and Diff). Different cases of Barrier, Lock and Diff are shown. Barrier (x) indicates the time to achieve a barrier on x nodes. For each of the microbenchmarks and their individual cases, the four bars (left to right) reflect the time on 2 different communication subsystems UDP/GM and FAST/GM

is accessed and diffs are obtained and applied to a page. In the Page microbenchmark, a shared memory region consisting of multiple 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 3 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 [5] 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 4, 8 and 16 nodes. Figure 4 illustrates the results in terms of execution time for the four applications described above. In all cases our implementation FAST/GM outperforms the UDP/GM, often by a significant amount. In the case of Jacobi, running on 16 nodes, a factor of improvement of 2 in execution time is obtained when UDP/GM is replaced by FAST/GM. Jacobi exhibits a high computation to communication ratio, which explains why the factor of improvement is only 2. Inspite of that Jacobi scales better with FAST/GM. The speedup increased from 3.66 to 22 when going from 4 to 16 nodes as opposed to UDP/GM which increased from 3.18 to 10.19. Again for SOR on 16 nodes FAST/GM shows a factor of improvement of 6 in execution time over UDP/GM. The speedup when going from 4 nodes to 16 nodes increased from 2.96 to 7.41 for FAST/GM, while for UDP/GM it only increased from 1.08 to 1.21. The significantly higher cost of obtaining a lock in the case of UDP/GM from Figure 3 and the fact that SOR uses locks extensively for global synchronization explains why there is a slowdown for SOR over UDP/GM. For 3D FFT and TSP the factor of improvement in execution time was 6.3 and 2.8, respectively. For TSP from 4 to 16 nodes the speedup increased from 3.11 to 5.48 for FAST/GM while for UDP/GM it decreased from 2.79

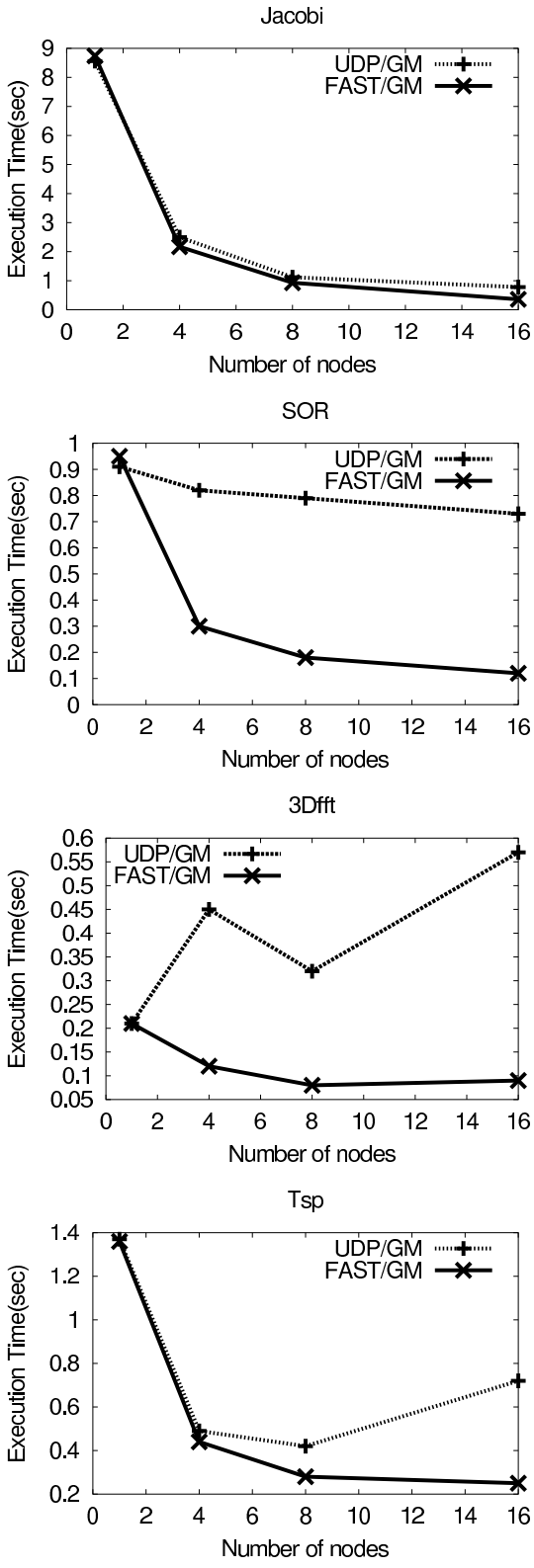

Figure 4. Execution times for four applications on two different communication subsystems as the number of nodes increases from 4 to 16

to 1.79. For 3D FFT from 4 to 16 nodes the speedup decreased from 11.5 to 5.75, while for UDP/GM a slowdown was observed at 16 nodes. 3D FFT has a high communication to computation ratio which reduces overlapping and pipelining of communication and computation accounting for the lower speedups obtained. 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 listed in Table 1 are shown in Figure 5. Here FAST-1/GM and UDP-1/GM refer to the execution time for 1-process while FAST-16/GM and UDP-16/GM refer to the execution time on 16 nodes.

Application	Size 4	Size 3	Size 2	Size 1
Jacobi(ZxZ)	2000	1500	1000	500
SOR(2000xZ)	4000	3000	2000	1000
TSP(cities)	18	17	16	
FFT(ZxZxZ)	64	32	16	8

Table 1. Application sizes

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 improvement is 2.84. Also in all cases the graphs show an increasing separation between the curves as the application size increases. This is particularly prominent in the case of 3D FFT. For Jacobi which has the highest computation to communication ratio there is a deviation between the curves 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 demonstrate the benefits of the proposed sub-system.

4 Related Work

A comparison between the performance of PastSet Software DSM system using TCP/IP and VIA is discussed in [7]. In this, it is shown that by replacing TCP/IP by the MVIA implementation of VIA improves the performance of a few microbenchmarks. The authors indicate 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 SDSM systems are discussed in [12]. 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

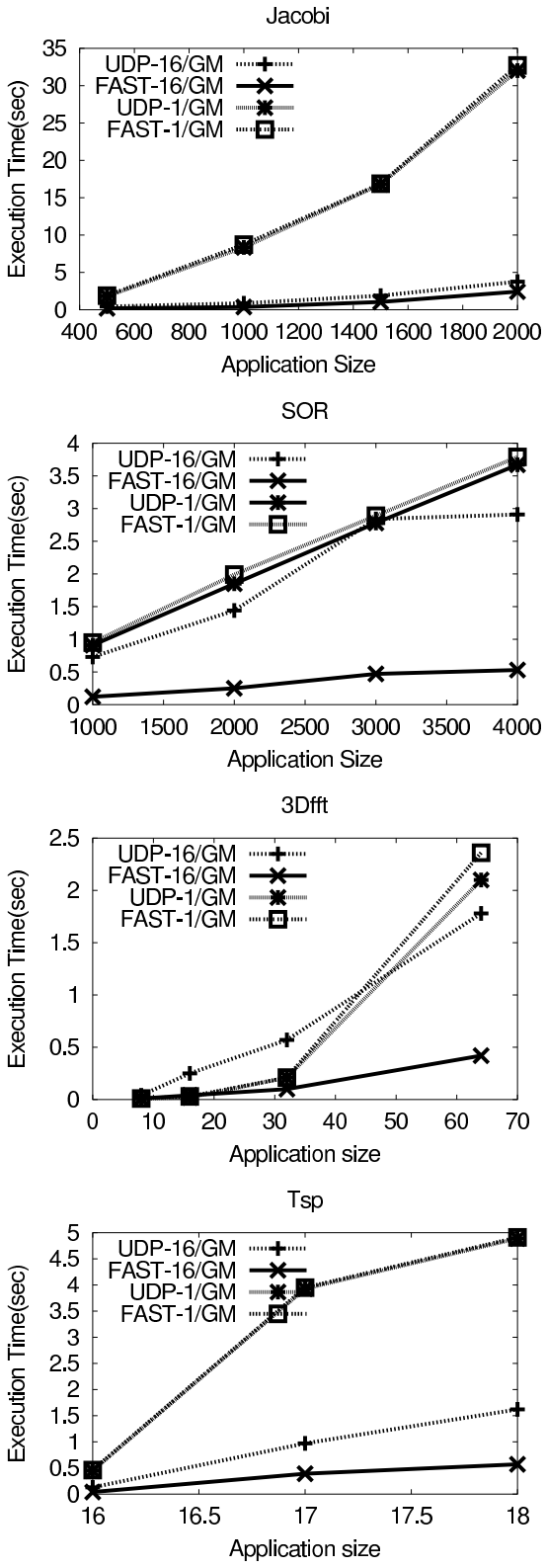

Figure 5. Execution times for four applications for different problem size sets. See the text for details of the size representation