REMOTE SHARED MEMORY OVER SUN FIRE LINK INTERCONNECT

Ahmad Afsahi Ying Qian
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, Canada, K7L 3N6
{ahmad, qiany}@ee.queensu.ca

Abstract

The interconnection networks and the communication system software in clusters of multiprocessors are critical to achieving high performance. Recently, Sun Microsystems has introduced a new system area network, *Sun Fire Link* interconnect, for its Sun Fire cluster systems with some performance results at the MPI level. Sun Fire Link is a memory-based interconnect with layered system software components that implements a mechanism for user-level messaging based on direct-access to remote memory regions of other nodes. This is referred to as *Remote Shared Memory* (RSM). This paper assesses the performance of the interconnect at the RSM level, as well as MPI level. We present the performance of different Remote Shared Memory API primitives, and MPI ping-pong latency over the Sun Fire Link interconnect on a cluster of Sun Fire 6800s. Our results indicate a better performance for the RSM *put* primitive than the *get* primitive for messages of larger than 64 bytes. We also compare the performance of Sun MPI implementation with the RSM level.

Keyword

System Area Networks, Remote Shared Memory, Clusters of Multiprocessors, Performance Evaluation, MPI

1. Introduction

Clusters of *Symmetric Multiprocessors* (SMP) have been regarded as viable scalable architectures to achieve supercomputing performance. There are two main components in such clusters: the SMP node, and the communication subsystem including the interconnect, and the communication system software.

Considerable work has gone into the design of SMP systems, and several vendors such as IBM, Sun, Compaq, SGI, and HP offer small and large scale shared memory systems. Sun Microsystems has recently introduced its Sun Fire systems, supporting two to 106 processors, backed up with its *Sun Fireplane* interconnect [1] used inside the Sun UltraSPARC III Cu. The Sun Fireplane interconnect uses up to four levels of interconnect ASICs to provide better shared-memory performance. All Sun Fire systems use point-to-point signals with a crossbar rather than a data bus.

Essentially, the interconnection network hardware and the communication system software are the keys to the performance of clusters of SMPs. Some interconnect technologies used in high-performance computers include Myrinet [2], Quadrics [3], and Giganet [4]. Each one of these interconnects provides different levels of performance, programmability, and integration with the operating systems. Myrinet provides high bandwidth and low latency, and supports user-level messaging. Quadrics integrates the local virtual memory into a distributed virtual shared memory, and supports remote direct memory access (RDMA). Giganet directly implements the Virtual Interface Architecture (VIA) [5] in Hardware. The InfiniBand Architecture [6] has been recently proposed to support the increasing demand on interprocessor communications as well as storage technologies. Mellanox Technologies [7], among others, has recently introduced its 4X InfiniBand host adapters and switches.

Recently, Sun Microsystems has introduced the *Sun Fire Link* interconnect for its Sun Fire clusters [8]. Sun Fire Link is a memory-based interconnect with layered system software components that implements a mechanism for user-level messaging based on direct access to remote memory regions of other nodes. This is referred to as *Remote Shared Memory* (RSM) [9]. Similar work in the past includes the VMMC memory model [10] on Princeton SHRIMP architecture, reflective memory in DEC Memory Channel [11], SHMEM [12] in Cray T3E, and in software as in ARMCI [13].

The authors in [8] have presented the performance of the Sun Fire Link interconnect at the MPI [14] level on a cluster of 8 Sun Fire 6800s. However, it is very important to discover the performance of interconnect at the closet layer to the hardware; that is, at the RSM level. This paper is the first step in a multi-stage study, which would eventually characterize and explain the performance seen by real applications under different programming paradigms on Sun Fire Link interconnect clusters. Specifically, this paper contributes by presenting the performance of the RSM user-level messaging layer over the Sun Fire Link Interconnect, as well as the overhead of Sun MPI [15] over RSM.

The rest of paper is organized as follows. In Section 2, we provide an introduction to the Sun Fire Link interconnect. Section 3 gives an overview of the Remote Shared Memory API. It then explains how inter-node communication is supported under RSM. In Section 4, Sun MPI implementation over RSM is briefly described. We mention our experimental framework in section 5. Section 6 presents the performance of the interconnect at the RSM and MPI levels. Related work is presented in section 7. Finally, we conclude our paper in section 8.

2. Sun Fire Link interconnect

Sun Fire Link interconnect is used to cluster Sun Fire 6800 and 15K/12K systems [16]. Nodes are connected to the network by a Sun Fire Link-specific I/O subsystem called the *Sun Fire Link assembly*. However, this is not an interface adapter, but a direct connection to the system crossbar. Each Sun Fire Link assembly contains two optical transceiver modules. Sun Fire 6800s can have up to two Sun Fire Link assemblies (4 optical links), where Sun Fire 15K/12K can have up to 8 assemblies (16 optical links). The availability of multiple Sun Fire Link assemblies in a server allows message traffic to be striped across the optical links for higher bandwidth. It will also provide protection against link failures.

The Sun Fire Link network can support up to 254 nodes, but the current Sun Fire switch supports only up to 8 nodes. The network connections for clusters of two to three Sun Fire systems can be point-to-point or through Sun Fire Link switches. For four to eight nodes, the switches are required. Nodes can communicate through a TCP/IP network for cluster administration issues, and exchanging control and status/error information.

The network interface (NI) is connected to the Sun Fireplane interconnect. However, it does not have a DMA engine. In contrast to the Quadrics, and InfiniBand Architecture that use DMA for remote memory operations, Sun Fire Link NI uses programmed I/O. The network interface can initiate interrupts as well as do polling for data transfer operations. It provides uncached read and write accesses to remote memory regions on other nodes. A Remote Shared Memory Application Programming Interface (RSMAPI) offers a set of user-level functions for remote memory operations bypassing the kernel [9].

3. Remote Shared Memory

Remote Shared Memory is a memory-based mechanism, which implements user-level inter-node messaging with direct access to memory that is resident on remote nodes. Table 1 shows some of the RSMAPI functions with their definitions. The complete API can be found in [9]. The RSMAPI can be divided into five categories: interconnect controller operations, cluster topology operations, memory segment operations, barrier operations, and event operations.

Table 1. Remote Shared Memory API (partial).

Interconnect Controller Operations	
rsm_get_controller ( )	get controller handle
rsm_release_controller ( )	release controller handle
Cluster Topology Operations	
rsm_free_interconnect_topology ( )	free interconnect topology
rsm_get_interconnect_topology ( )	get interconnect topology
Memory Segment Operations	
rsm_memseg_export_create ( )	resource allocation function for exporting memory segments
rsm_memseg_export_destroy ( )	resource release function for exporting memory segments
rsm_memseg_export_publish ( )	allow a memory segment to be imported by other nodes
rsm_memseg_export_republish ()	re-allow a memory segment to be imported by other nodes
rsm_memseg_export_unpublish ( )	disallow a memory segment to be imported by other nodes
rsm_memseg_import_connect ( )	create logical connection between import and export sides
rsm_memseg_import_disconnect ( )	break logical connection between import and export sides
rsm_memseg_import_get ( )	read from an imported segment
rsm_memseg_import_put ( )	write to an imported segment
rsm_memseg_import_map ( )	map imported segment
rsm_memseg_import_unmap ( )	unmap imported segment
Barrier Operations	
rsm_memseg_import_close_ barrier ( )	close barrier for imported segment
rsm_memseg_import_destroy_ barrier ( )	destroy barrier for imported segment
rsm_memseg_import_init_barrier ( )	create barrier for imported segment
rsm_memseg_import_open_barrier ( )	open barrier for imported segment
rsm_memseg_import_order_barrier ( )	impose the order of write in one barrier
rsm_memseg_import_set_mode ( )	set mode for barrier scoping
Event Operations	
rsm_intr_signal_post ( )	signal for an event
rsm_intr_signal_wait ( )	wait for an event

Figure 1 shows the general message-passing structure under the Remote Shared Memory model. Communication under the RSM involves two basic steps: 1. *segment setup* and *tear down*; 2. the actual *data transfer* using the direct read and write models.

In essence, an application process running as the "export" side, as shown in Figure 1, should first create an RSM export segment from its local address space, and then publish it to make it available for processes on the other nodes. One or more remote processes as the "import" side will create an RSM import segment with a virtual connection between the import and export segments. This is called the setup phase. After the connection is established, the process at the "import" side can communicate with the process at the "export" side by writing into and reading from the shared memory. This is called the data transfer phase. When data are successfully transferred, the last step is to tear down the connection. The "import" side disconnects the connection and the "export" side unpublishes the segments, and destroys the memory handle.

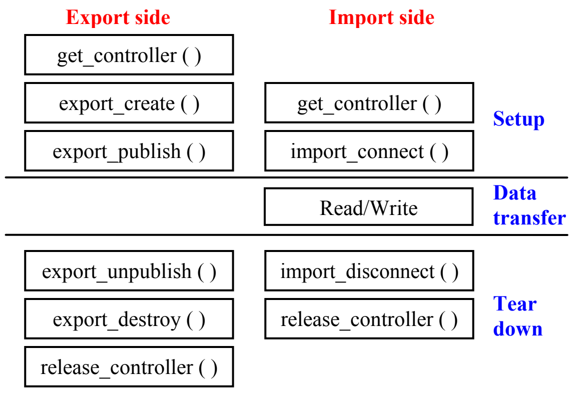


Figure 1. Setup, data transfer, and tear down phases under the RSM communication.

Figure 2 illustrates the main steps for the data transfer phase. The "import" side can use the RSM *put*/*get* primitives, or use *mapping* technique to read or write data. *Put* writes to (*get* reads from) the exported memory segment through the connection. The mapping method maps the exported segment into the imported address space and then uses the CPU store/load memory operations for the data transfer. This could be through the use of *memcpy* operation. However, *memcpy* is not guaranteed to use the CPU Block Store/Load instructions. Thus, some library routines should be used for this purpose. The *barrier* operations ensure the data transfers are successfully completed before they return. The *order* function is optional and can impose the order of multiple writes in one barrier. The *signal* operation is used to inform the "export" side that the "import" side has written something onto the exported segment.

4. Sun MPI over RSM

Sun MPI has been implemented on top of RSM for internode communication [15]. Because the segment setup and tear down have large overhead (as seen in Section 6.1) connections remain established during the application runtime. Messages are sent in one of two fashions: short messages (less than 3912 bytes) and long messages. Short messages are sent *eagerly*. They are fit into multiple postboxes, 64 bytes each. Buffers, barriers, and signal operations are not used due to their high overhead. Note that even for messages less than 64 bytes, a full 64-byte message is written into the postboxes. This is because data transfers less than 64 bytes invoke a kernel interrupt on the remote node, which incurs more delay.

Long messages are sent in 1024-byte buffers under the control of multiple postboxes. A long message is sent eagerly if it is less than 256 Kbytes. Otherwise, *rendezvous* protocol is used. Barriers are opened for each stripe to make sure the writes are successfully done. Connections are half-duplex. Once the message is transferred from the staging buffers (exported segments) to the application user space, the receiver sends an *Ack* message to the sender, on its own connection, so that the buffers could be reclaimed.

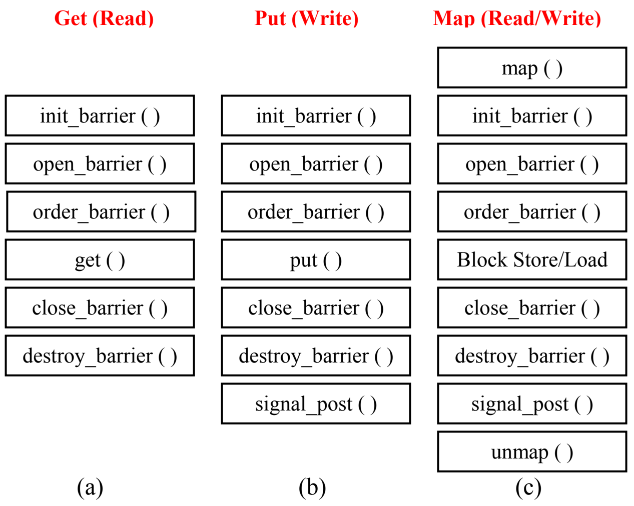


Figure 2. Steps in the data transfer phase: (a) get, (b) put, (c) map.

When a process enters an MPI call, the *Progress Engine* in Sun MPI might act on a variety of messages belonging to past or future MPI calls [15]. For instance, the environment variable "MPI_POLLALL" can be set to '1' or '0'. In the *general polling* (MPI_POLLALL = 1), Sun MPI polls for all incoming messages to drain system buffers. In the *directed polling* (MPI_POLLALL = 0), the implementation searches only for the specified connection.

5. Experimental Framework

We test the performance of the Sun Fire Link interconnect on a cluster of 4 Sun Fire 6800 nodes at the High Performance Computing Virtual Laboratory (HPCVL) at the Queen's University. HPCVL participated in a beta program with Sun Microsystems to test the Sun Fire Link hardware/software before its official release in Nov. 2002. HPCVL is still using the original Sun Fire Link hardware. We are not aware of the differences, if any.

The Sun Fire 6800 is a mid-size symmetric multiprocessor with up to 24 UltraSPARC III Cu processors. Each Sun Fire 6800 node at HPCVL has 24 900 MHz UltraSPARC III processors with 8 MB E-cache and 24 GB RAM. The software environment includes Sun Solaris 9, Sun HPC Cluster Tools 5.0, and Forte Developer 6, update 2. We had exclusive access to the cluster during our experimentation, and we bypassed the Sun Grid Engine in our tests. Note that for our experiments, we only used two of the Sun Fire 6800s in the cluster under the directed polling. Our timing measurement was done using the high resolution timer available in Solaris. In the following, we present our framework experimenting with the Sun Fire Link cluster.

5.1. Remote Shared Memory API

The RSMAPI is the closest layer to the Sun Fire Link. We test the performance of RSMAPI primitives, as shown in Table 1, with varying parameters over the Sun Fire Link interconnect.

5.2. Latency

We would like to see the overhead of Sun MPI over the RSM. Latency is defined as the time it takes for a message to travel from the sender process address space to the receiver process address space. The *latency* test is the ping-pong test where the sender sends a message and the receiver upon receiving the message, immediately replies with the same message size. This is repeated sufficient number of times to eliminate the transient conditions of the network. Then, the average round-trip time divided by two is reported as the one-way latency. This test is repeated for messages of increasing sizes. We test using matching pairs of blocking sends and receives under different MPI send modes; that is, the *standard* mode, the *synchronous* mode, the *buffered* mode, and the *ready* mode. We also report the latency under different message buffers.

6. Experimental Results

6.1. Remote Shared Memory API

Table 2 shows the execution times for different RSMAPI primitives. Some API calls are affected by the memory segment size (shown here with 16 Kbytes memory segment size), while others are not affected at all. Note that the API primitives with the asterisk sign are normally used only once for each connection. The minimum memory segment size is 8 Kbytes in the current implementation of RSM. Figure 3 shows the percentage execution times for the "export" and "import" sides with a typical 16 KB memory segment, and data size. It is clear that the *connect* and *disconnect* calls take more than 80% of the execution time at the import side. However, this should happen only once for each connection. The time for *open barrier, close barrier*, and the *signal* API primitives are not small compared to the time to *put* small message sizes. This is why barrier is not used for small message sizes, and data transfer is done only through postboxes.

Figure 4 shows the time for several RSMAPI functions at the "export" side affected by memory segment size. The *export_destroy* primitive is the least affected one. The results imply that applications are better off creating one large memory segment for multiple connections instead of creating multiple small memory segments.

Figure 5 compares the performance of the RSM *put* and *get* functions under different message sizes. It is clear that *put* has a much better performance than *get* for message sizes more than 64 bytes. This implies that applications and middleware packages/libraries should use remote writes rather than remote reads. For the same reason, Sun MPI [15] uses push protocols rather than pull over Sun Fire Link. Also, it is better to have as many *put* operations within a barrier due to the barrier overhead. The performance of *get* is better for messages up to 64 bytes for the same reason stated earlier in section 4.

Table 2. Execution times of different RSMAPI calls.

Export side			Time (µs)
get_interconnect_topology ( )		*	12.65
get_controller ( )		*	841.00
free_interconnect_topology ( )		*	0.61
export_create ( )	16KB	*	103.61
export_publish ( )	16KB	*	119.36
export_unpublish ( )	16KB	*	73.48
export_destroy ( )	16KB	*	16.73
release_controller ( )		*	3.63

Import side			Time (µs)
import_connect ( )		*	173.45
import_map ( )		*	13.56
import_init_barrier ( )			0.33
import_set_mode ( )			0.38
import_open_barrier ( )			9.93
import_order_barrier ( )			16.80
import_put ( )	16KB		27.73
import_get ( )	16KB		373.01
import_close_barrier ( )			7.13
import_destroy_barrier ( )			0.14
signal_post ( )			23.78
import_unmap ( )		*	21.40
import_disconnect ( )		*	486.31

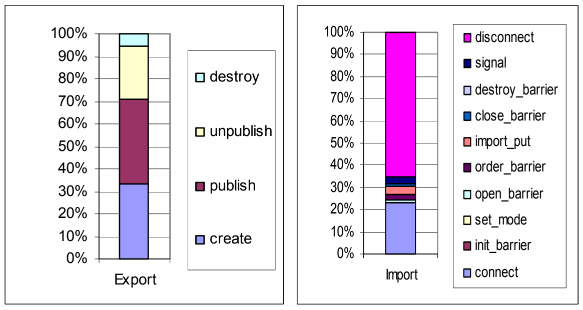


Figure 3. Percentage executions time for the export and import side (16 KB segment, and data sizes).

6.2. Latency

Figure 6 shows the latency for off-node communication, where the endpoints are on different SMP nodes. The latency for 1-byte message is 5 microseconds for *Standard*, *Ready, Synchronous,* and *Diff buf* modes, and 6 microseconds for the *Buffered* mode. This remains almost fixed for up to 64 bytes. Figure 6 also shows that the Sun MPI uses the short message protocol for messages up to 3912 bytes and then switches to the long message protocol.

Note that our MPI measurements have been done under the directed polling. Shorter message latency (3.7 microseconds) has been reported in [8] for zero byte message under general polling. The performance for short messages is better than the results for MPI over Quadrics reported in [3], and comparable to the results for MPI over InfiniBand in [17].

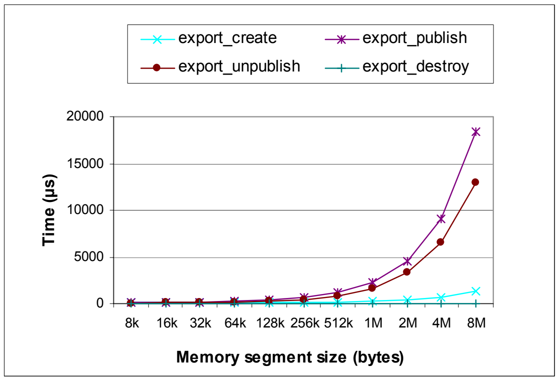
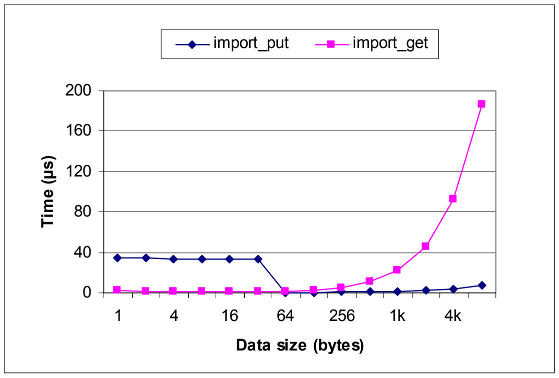


Figure 4. Execution times for several RSMAPI calls.

Figure 7 compares the performance of RSM *put* with the standard MPI ping-pong latency. Note that we have assumed the same execution time for *put* with 1 to 64 bytes.



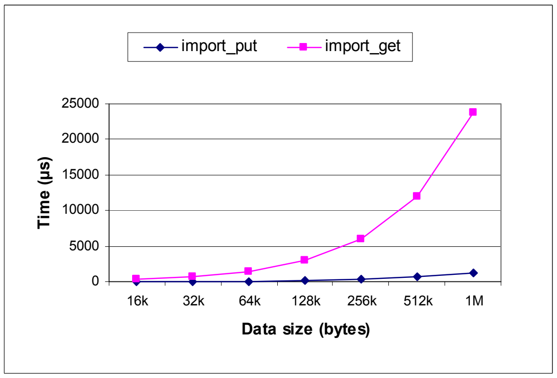
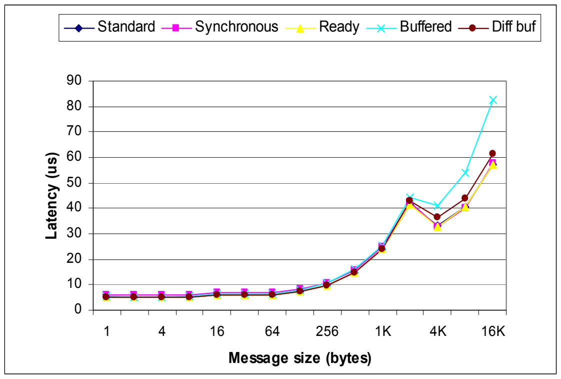


Figure 5. Comparison of the RSM put and get under different message sizes.

7. Related Work

Research groups in academia and industry have been studying the performance of the clusters and the interconnection networks. The performance of Quadrics interconnection networks has been studied in [3]. The authors in [3] have shown the performance of QsNet under different communication patterns. Numerous research studies have been done on the Myrinet [2]. In [17], the authors have assessed the performance of their MPI implementation on top of the Mellanox's InfiniHost adapters and InfiniScale switches. Sun Microsystems has introduced the Sun Fire Link interconnect, and the Sun MPI implementation [15]. They have presented some performance results on a cluster of 8 Sun Fire 6800s. While Quadrics QsNET, and InfiniBand Architecture [6] use DMA engines to perform a remote operation, Sun Fire Link uses programmed I/O instead.



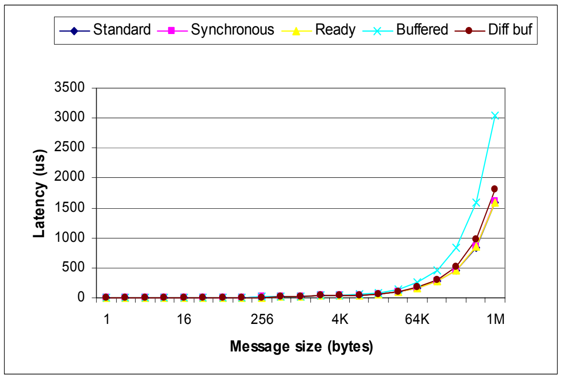


Figure 6. Small and large message size MPI latencies.

There are different semantics supported by different networks. Cray T3E uses shared-memory concept [12], where it provides a globally accessible, physically distributed memory system to provide implicit communications. VMMC [10] provides protected, direct communication between the sender's and receiver's virtual address spaces. The receiving process exports areas of its address space to allow the sending process to transfer data. The data are transferred from the sender's local virtual memory to a previously imported receive buffer. There is no explicit receive operation in VMMC. The reflective-memory model, supported by DEC Memory Channel [11], is a sort of hybrid between explicit send-receive and implicit shared-memory models by providing a write-only memory "window" in another process address space. All data written to the window go into the address space of the destination process. ARMCI [13] is a software architecture for supporting remote memory operations on clusters.

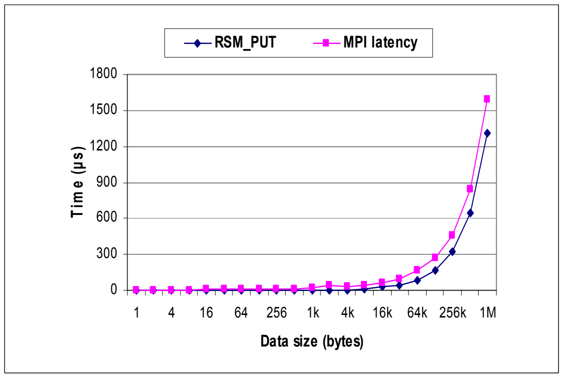


Figure 7. RSM put and MPI latency comparison.

8. Conclusion and Future Research

Clusters of multiprocessors have been regarded as a viable platform to provide supercomputer performance. However, the interconnection network, the network interface, and the supporting communication software are the deciding factors in the performance of such clusters. Thus, it is very important to assess the performance of high-performance interconnects at the user and MPI levels.

In this paper, we attempt to measure the performance of the recently introduced Sun Fire Link interconnect. Sun Fire Link is a memory-based interconnect, where the Sun MPI library uses the Remote Shared Memory model for its user-level internode messaging protocol. Our performance results include the execution times for different RSMAPI primitives, and the MPI ping-pong latency. The Sun MPI implementation achieves 5 microseconds for off-node latency. Our RSM results indicate that *put* has a better performance than *get* on this interconnect, as in other memory-based interconnects. We also compare the performance of Sun MPI with the RSM level.

As for future research, we would like to do an extended analysis of the interconnect in terms of its latency, bandwidth, parameters of LogP model, collective communications, and different permutation patterns. We are also interested in finding the performance of real applications on Sun Fire Link interconnect clusters under different parallel programming paradigms.

Acknowledgements

Special thanks go to E. Loh of Sun Microsystems for providing the latest RSM code for the Sun Fire Link. We would also like to thank K. Edgecombe, and H. Schmider of High Performance Computing Virtual Laboratory (http://www.hpcvl.org) at the Queen's University, and G. Braida of Sun Microsystems for their kind help in accessing the Sun Fire cluster with its Sun Fire Link interconnect. This work was supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Queen's University. The second author was also supported by Ontario Graduate Scholarship for Science and Technology (OGSST).

References

[1] A. Charlesworth, The Sun Fireplane Interconnect, *IEEE Micro, 22(1)*, 2002, 36-45.
[2] N. J. Boden, D. Cohen, R. E. Kulawik, C. L. Seitz, J. N. Seizovic, & W-K Su, Myrinet: A Gigabit-per-Second Local Area Network, *IEEE Micro, 15(1),* 1995, 26-36.
[3] F. Petrini, S. Coll, E. Frachtenberg, & A. Hoisie, Performance Evaluation of the Quadrics Interconnection Network, *Journal of Cluster Computing, 6(2),* 2003, 125-142.
[4] W. Vogels, D. Follett, J. Hsieh, D. Lifka, & D. Stern, Tree-Saturation Control in the AC3 Velocity Cluster, *Proc. Hot Interconnect 8*, 2000.
[5] V. Dunning, G. Regnier, G. McAlpine, D. Cameron, B, Shubert, F. Berry, A. Merritt, E. Gronke, & C. Dodd, The Virtual Interface Architecture, *IEEE Micro, 18(2),* 1998, 66-76.
[6] InfiniBand Architecture (http://www.infinibandta.org).
[7] Mellanox Technologies (http://www.mellanox.com).
[8] S. J. Sistare, & C. J. Jackson, "Ultra-High Performance Communication with MPI and the Sun Fire Link Interconnect, *Proc. Supercomputing Conference*, 2002.
[9] Remote Shared Memory API for Sun Cluster Systems (http://docs.sun.com).
[10] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, & K. Li, VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication, *Proc. Hot Interconnects V*, 1997.
[11] R. Gillett, MEMORY CHANNEL Network for PCI: An Optimized Cluster Interconnect, *IEEE Micro, 16(2),* 1996, 12-18.
[12] A. Barriuso, and A. Knies, *SHMEM user's guide*, (Cray Research Inc., SN-2516, 1994).
[13] J. Nieplocha, V. Tipparaju, A. Saify, & D. Panda, Protocols and Strategies for Optimizing Performance of Remote Memory Operations on Clusters, *Proc. International Parallel and Distributed Processing Symposium*, 2002.
[14] Message Passing Interface Forum: MPI, A Message Passing Interface Standard, Version 1.2, 1997.
[15] Sun HPC ClusterTools 5 Documentation (http://docs.sun.com).
[16] Sun Fire Link System Overview (http://docs.sun.com).
[17] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, & D. K. Panda, High Performance RDMA-Based MPI Implementation over InfiniBand, *Proc. 17th ACM International Conference on Supercomputing*, 2003, 295-304.