Comparing Lustre Performance using Myricom 10G Dual Protocol NICs

Scott Atchley, Brice Goglin

Myricom Inc., Arcadia, CA

{atchley, brice}@myri.com

Abstract

In this paper, we look at how Lustre performs using Myricom's 10 Gb/s NICs. Lustre clients and servers can use these programmable NICs in either native Ethernet mode or in Myrinet mode. Lustre provides metrics for measuring metadata retrieval (small message) performance as well as for bulk read and write operations. We show that with multiple clients Lustre can achieve 95% of line-rate performance using Ethernet (TCP) when reading but only 50% when writing. We show that Lustre achieves a 95% of line-rate with a single client when using the NIC in Myrinet mode. In both cases, Lustre can serve up to 45,000 metadata requests per second using multiple clients.

1. Introduction

As clusters grow larger, cluster users try to solve larger problems or the same problems with finer granularity. Their data sets are growing as well. In [**Bhat04**], the authors discuss a single fusion code that is already generating 1 TB per week and they expect that rate to grow by an order of magnitude within a few years. If several such codes are in use on a large cluster, both the communication and storage requirements for the cluster will be stressed. In response, high-performance interconnects such as Myrinet [**myri**] and Infiniband [**Pfister01**] have been pushing higher bandwidth for communications between the computing nodes, while there has been lots of work to efficiently use these networks for distributed storage. Several systems such as PVFS [**Ligon01**], GPFS [**Schmuck02**] and Lustre [**Schwann03**] now achieve very high performance in a

scalable manner. In the meantime, Ethernet has been making in-roads in HPC as shown by the Top500 list, and 10 Gb/s Ethernet products are reaching the market.

In this paper, we take a look at the convergence of these issues. In section 2, we describe Myricom's new PCI-Express based NICs that can speak native Ethernet as well as Myrinet. We also look at the current software stack for Myrinet. In section 3, we discuss the Lustre distributed file system. In section 4, we describe our implementation of a Lustre Network Driver using MX. In section 5, we compare Lustre performance using Myri-10G NICs in native Ethernet mode and Myrinet mode. Finally, in section 6, we offer our conclusions and perspectives.

2. Myricom 10G

To meet the HPC needs for cluster communication and distributed storage, Myricom has introduced 10 Gb/s network interface cards (NIC) and switches based on the same physical layer (PHY or level 1) as 10 Gb/s Ethernet [**10G**] including CX-4 copper and 10G-BaseR fibre. The NICs support both Myrinet and Ethernet protocols natively. In the past, Myrinet NICs could only provide Ethernet emulation over Myrinet. When used in Myrinet mode, the NICs are connected to a Myrinet switch and, when used in Ethernet mode, they are connected to any vendor's 10 Gb/s Ethernet switch.

2.1 Myrinet Protocols

Myrinet is a packet-communication and switching technology that is an open, standardized protocol and link layer. At the data-link layer, Myrinet is defined by its packet format and flow control. Each packet is has one or more routing bytes, two bytes for packet type, the payload, and a CRC byte. Myrinet has byte-level flow control unlike Ethernet, which uses packet-level flow control. Because Myrinet uses source-routing, switches are simple and NICs are complex.

Due to its open specifications, Myrinet has been widely used for academic research. Its easily reprogrammable hardware led to several major high-speed network software stacks in the past, such as Unet, Active Messages, BIP, and Trapeze. Since the introduction of Myrinet, Myricom has offered three software stacks, GM-1, GM-2 and MX. GM-2 was the primary stack for the PCI NICs and the initial stack for the PCI-X NICs. MX (discussed below) is the successor to GM-2 for the PCI-X NICs and is the only stack offered on the new PCI-E NICs.

2.1.1 Overview of Myrinet Express

Myrinet Express (MX) is a high-performance, low-level, message-passing software interface tailored for Myrinet networks. MX is covered in greater detail in the MX documentation [**MX**], but we provide an overview here.

The MX API mirrors the Message Passing Interface (MPI) API [**mpi**] with some differences. MPI uses a two-sided communication pattern in which senders and receivers participate in the communication. Although MPI-2 provides for one-sided communication as well, the bulk of MPI applications today use the two-sided interface. All communication functions are asynchronous and return request data structures. The programmer can then pass these structures to status checking functions.

All messages in MX contain matching information. This information is used to match incoming sends to receive requests. Both send and receive requests specify matching information, a destination endpoint, and a list of user memory segments and their respective lengths.

Although MX was originally designed for MPI applications, we do our best to satisfy other applications whose needs may be very different. In addition to the MPI style calls, MX also provides a kernel API, efficient support for unexpected messages (a necessity for client/server applications), an active message model [**Eicken92**] (also good for client/server applications), and

matching-aware, race-free completion functions. Thus, MX is efficient for MPI and other APIs and applications such as Sockets-MX and distributed storage. In [**Goglin05**], the compare the efforts to provide a kernel API using the previous Myrinet API, GM-2, as compared to do the same with MX. Unlike with GM, the MX in the kernel is able to reach a higher bandwidth than in userspace.

2.1.2 Unexpected Receives in MX

Since MX is optimized for two-sided, MPI style applications, it assumes that a process' communication pattern is regular or *expected*. Thus, for each sent message, it assumes that a receiver is expecting that message and has pre-posted a receive request to match it.

Even if the application attempts to pre-post receives, it may still be possible to encounter *unexpected* receives (i.e. a send message arriving prior to posting the matching receive) due to clock drift among the hosts. More commonly, however, it is simply due to poor programming practices. In many low-level interfaces, unexpected receives are dropped and the sender has to retransmit later in hopes that a receive has been posted.

MX provides a buffer per endpoint to handle such unexpected messages. In case the receive may be a little late, MX copies the unexpected message into the buffer. When the matching receive is posted, MX copies the data to the correct location and can complete the message immediately. The buffer size is limited and only smaller messages (currently less than 32 KB) can be buffered this way. For larger messages, MX stores the information about the message including the matching information. The buffer size can be adjusted by the application when the endpoint is opened. The threshold for the message size that can be buffered locally is adjustable by the system when the driver is loaded.

From the application's perspective, unexpected message handling is transparent. The receiving process does not know if the incoming send has already been stored in the unexpected buffer. If it is in the buffer, the receive can complete immediately. If the message is large and only the message information but not payload is in the buffer, then MX will notify the sender to begin immediate transmission.

If unexpected receives are unavoidable and the programmer does not wish to pre-post generic receives, MX provides a callback handler that the programmer can use to immediately handle the request without copying it into the MX unexpected buffer. Because this function halts progress in the MX library, it is essential that it executes as quickly as possible and does not block (e.g. allocate memory, attempt to acquire a semaphore, etc.).

Between the efficient handling of unexpected and the active message model provided by the callback handler, MX provides support for non-MPI applications such as client/server applications like Lustre. We discuss how this impacts Lustre in section 4.2.

2.1.3 MX Kernel Library

MX was designed as a userspace library. It has been extended to support various types of memory addressing by one of the authors. The kernel API is nearly identical to the userspace API. The kernel API, however, does require a different memory address structure and support for pinning various types of memory (i.e. pages, kernel virtual addresses, and user virtual addresses.) Unlike the userspace library, errors always return rather than exit and the unexpected buffer size if fixed since memory in the kernel may be a limited resource.

2.2 Ethernet

When used in Ethernet mode, the NIC appears to the OS as a traditional Ethernet NIC. Although it does not operate like a TCP Offload Engine (TOE), however, it does utilize the on-

board processor to handle some tasks on behalf of the CPU. These include Ethernet checksum offload, TCP Segmentation Offload (TSO and also known as Large Send Offload or LSO), zero-copy sends (sendfile), automatic accommodation of VLAN MTUs, PCI-Express relaxed ordering, and New Driver API (NAPI) that enables interrupts per device rather than per packet.

3. Lustre

3.1 Overview

Lustre is a high-performance, distributed file system developed by Cluster File Systems. The goal for Lustre is to provide a scalable file system suitable for large-scale clusters (10,000+ nodes). Lustre borrows some ideas from previous distributed file systems such as the Network File System (NFS) and the Andrew File System (AFS), but diverges in areas to increase performance and/or scalability.

Like IBM's General Parallel File System (GPFS) [**GPFS**], Lustre uses object-based storage instead of volume-based storage to allow storage of data large than any single disk or disk array. By default, data is striped over multiple servers in a RAID-0 fashion. GPFS also supports mirroring data (RAID-1) and using parity checksums (RAID-5) for improved fault-tolerance, which Lustre intends to implement but does not yet support. Like GPFS and the Parallel Virtual File System (PVFS), Lustre separates metadata handling from data storage.

Unlike GPFS, which runs only on TCP/IP, Lustre can use multiple, heterogeneous interconnects at the same time. A Lustre installation may span multiple networks that use different interconnects and thus have different performance levels. Lustre uses hosts with connections to the different networks as routers to move messages from one network to the other. In order to prevent traffic from a faster network from overwhelming a router trying to move that data onto a slower network, Lustre requires flow control between peers. GPFS relies on TCP flow control

which means that all interconnects must provide native Ethernet or at least Ethernet emulation or encapsulation.

While GPFS may only get high performance through the Ethernet or IP emulation of high-speed networks that are available in clusters, Lustre may use the native network API which is generally much faster and does not have a high CPU utilization since it uses advanced features such as OS-bypass, zero-copy and/or RDMA.

3.2 Lustre Networking

Originally, Lustre used the Portals API from Sandia National Lab. Portals [**Portals**] was designed as a high-performance MPI implementation for very large clusters (1000s to 10000s of nodes). Portals provides for both two-sided and one-sided operations. It provides both a *matching put* and a *matching get* that do not use explicit memory addresses, but matching information. If the underlying network supports RDMA, Portals can take advantage of it.

For communication, Lustre used the Portals one-sided API with PUT, ACK, GET and REPLY operations. The PUT may optionally be followed by an ACK, but a GET is also followed by a REPLY that returns the requested data. These are all asynchronous calls.

For each supported interconnect, Portals has a Network Abstraction Layer (NAL) that implements these functions. Although the implementations varied according to the capabilities of the interconnect, each implemented the actual PUT and GET operations with multiple steps on the network. Interestingly, Portals uses a two-sided send/receive interface when calling the NALs.

Starting with Lustre 1.4.6, CFS has forked the Portals code and now calls it Lustre Networking (LNET). The interconnect implementations are now called Lustre Network Drivers (LNDs). The major difference from the driver point of view is that there is only one send and one receive call

from LNET. Previously in Portals, there were separate send and receive calls depending on whether the payload was mapped into virtual kernel memory or simply a list of pages.

4. MXLND

The MX Lustre Networking Driver (MXLND) uses the Myrinet Express kernel API to implement the LNET calls. Building a MX LND is necessary to allow Lustre to migrate to Myri-10G NICs since GM will not be ported to these NICs. Since MX is compatible with both existing Myri-2G PCI-X NICs as well as the new Myri-10G PCI-E NICs, Lustre using MXLND can run on thousands of installed clusters in addition to new clusters. Lustre can also be used in Ethernet mode on 10Gb/s Ethernet without a MX port and we look at the performance of Lustre using Ethernet and MX in section 5.

Since Lustre actually uses a two-sided implementation to implement a one-sided API, MX is ideally suited to build a LND. Also, MX has a very low connection overhead (128 bytes per peer) and no need for per peer buffers, which allows it to scale to very large clusters.

4.1 Data Structures

For each peer, MXLND itself maintains information including its Lustre network ID (NID), MX endpoint address, peer state, available send credits and accumulated peer credits for flow control, separate queues for sends requiring credits and those that do not, as well as counters for tracking queued send messages and in-flight (posted) messages.

MXLND uses a common data structure for both send (TX) and receive (RX) messages. This structure includes an expiration time, message state (e.g. completed, idle, etc.), the peer's NID and a pointer to the peer's data structure, message type (discussed below), message size, matching information (cookie), and pointers to the LNET message(s) that LNET passed into MXLND.

The message structure also contains a pointer to an unmapped kernel page that is used to hold the entire message if it is smaller than a page (typically 4 KB or 8 KB depending on CPU). The structure then maps a MXLND small message header onto that page. The message structure has a MX kernel segment descriptor that points to this page for small messages. For large messages (up to 256 pages as of this time), the message structure also has a pointer to an array of MX kernel segment descriptors to map kernel virtual memory or unmapped pages.

MXLND has global data structure that includes its own MX endpoint, a list of peers, lists of all TX and RX structures (for cleanup), lists for idle (unused) TX and RX descriptors, next cookie, and a list of pending (posted send and receive) messages.

4.2 Unexpected Message Handling

In the MX discussion, we discuss the preference to avoid unexpected messages. Lustre is not like a typical MPI application and every PUT and GET is unexpected. To handle this, most LNDs pre-allocate and pre-post receives. The number of pre-posted receives must be large enough to allow a connect message from each potential, yet unknown peer as well as to allow posting receives for the number of flow control credits for each known peer.

In MX, matching of receives means traversing a list. To avoid traversing a list with a few thousand generic receives to look for a matching RDMA, we chose to only post expected receives and to use the MX unexpected callback handler to get an unused RX descriptor and post a matching receive in the callback. This will keep the expected receive list short and fast as well as avoid copying the unexpected message into and out of the unexpected buffer.

4.3 Message Types

As mentioned earlier, the two primary Lustre Networking functions are send and receive. MXLND needs to service these calls without blocking. Lustre will send and receive four mes-

sage types, PUT, ACK, GET and REPLY. In order to implement these four messages, MXLND uses the following ten message types.

4.3.1 Eager Messages

We have mentioned that MXLND will send small messages using a single page. These messages use an eager protocol in MX. In MXLND, we call these messages EAGER as well and they may be used for small LNET requests.

4.3.2 Connection Setup

As mentioned previously, Lustre requires flow control. To avoid deadlock and memory corruption, MXLND must assure that all peers are using the same values for the amount of credits and small message size.

To do this, MXLND provides connection request and connection acknowledgement messages. Before a new peer or an established peer with an unready connection state can send a LNET message, MXLND will perform a connection request/ack handshake. Once completed, both peers are ready to send as shown in figure 1.

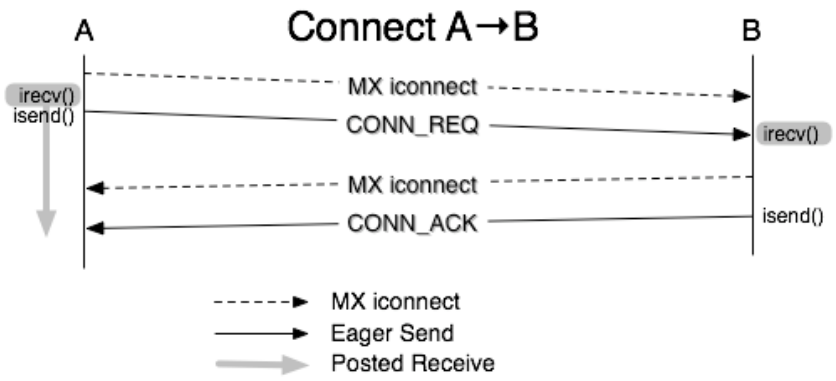

Figure 1: Connect handshake

Prior to writing MXLND, MX only had a synchronous connect() function. In order to avoid having a separate thread manage connection requests, we added an asynchronous connect, mx_iconnect.

4.3.3 PUT

MXLND has to handle two cases when handling a PUT. If the message header and payload will fit on a page, the initiator will send the entire message using an eager message and the target does not send anything back.

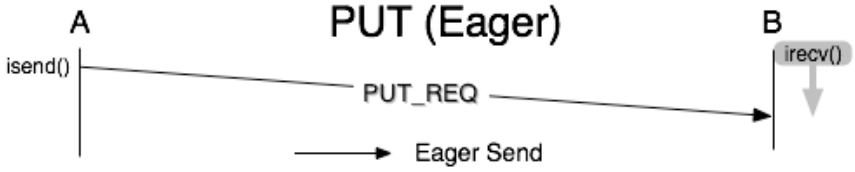

Figure 2: Small message (eager) PUT

If the message is larger than a page, then MXLND will need to use a PUT request, PUT ack, and a PUT RDMA. The initiator first posts a receive for the PUT ack, then sends a PUT request with the Lustre header information to the target. The target, maps the memory segments and posts a receive for the RDMA, and then sends the PUT ack. The initiator then maps the memory segments and sends the RDMA message.

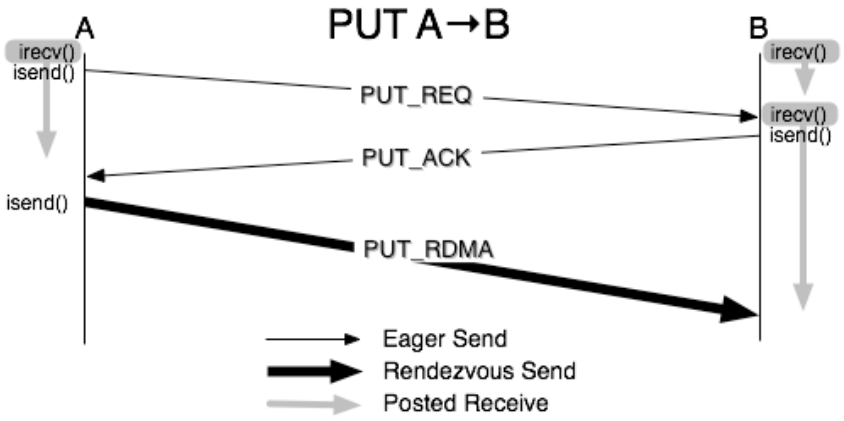

Figure 3: Large message PUT with RDMA

If the target cannot map the memory, it returns a NAK message instead of a PUT ack.

4.3.4 GET

Whether the message is large or small, the same steps are involved although small messages still copy the data onto a single page while large message map the memory and perform RDMA. The initiator first posts a receive for the incoming data and then sends a GET request to the target. The target parses the header and then either sends the data or a NAK if the target could not fulfill the request.

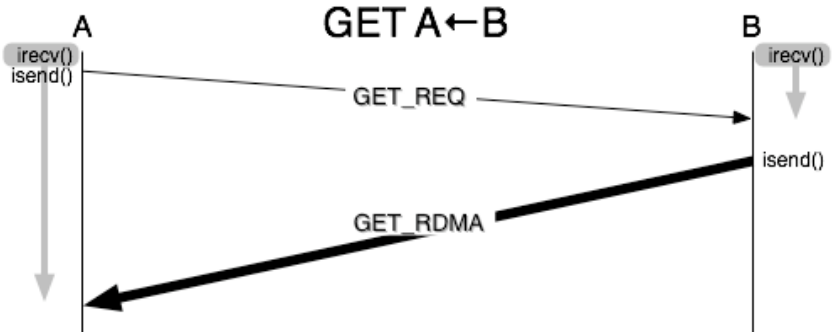

Figure 4: Large message GET with RDMA

MXLND does not actually use the LNET REPLY message type unless the message passes through a router. In that case, a REPLY is handled just like a PUT.

4.3.5 NOOP

If a pair of peers is not communicating in a balanced manner, one will run out of credits waiting for the other to send a message that may never come. To avoid starvation, when a peer notices that it has received credits from a peer that exceeds a high-water mark, it will use a NOOP message to return the credits to the peer.

5. Performance

Lustre provides tests for three types of operations: metadata, bulk read, and bulk write. For the metadata test, the client sends a small message request to the server and the server replies with a small message. Small messages are 4 KB or less. Because read and write (i.e. PUT and GET) are implemented differently, we measure both. The bulk read and write tests involve a small message request from the client to the server followed by either a PUT (for a read request) or GET (for a write request). The PUT or GET operation transfers 256 kernel pages of 4 KB each.

Most LNDs incorporate flow control between peers to prevent fast peers from overwhelming routers that connect to peers on slower networks. Both LNDs are set by default to allow eight messages in-flight between any two peers. The tests allows us to use varying numbers of threads to generate requests, but there are never more than eight requests outstanding at any one time.

The test machines have dual, dual-core 2.4GHz Opterons, with a nVidia chipset and 8 GB of RAM. We used Myri-10G CX-4 cards for all tests. To test native Ethernet, we connected the Myri-10G cards to a Fujitsu XG700 Ethernet switch and, to test MX, we connected the cards to a Myrinet 128 port Myri-10G switch. Fujitsu claims 450 us latency and a non-blocking design although we do not test these claims. We estimate the Myrinet switch latency at ~350 ns and cards on the same switch line card are connected via a 16-port, non-blocking crossbar.

We used a beta version of Lustre 1.4.6 on Red Hat 3.4.5 (2.6.9-34 kernel) with the Lustre zero-copy TCP patch. We found that using the patch increased both read and write performance from 370 to over 600 MB/s. Lustre was compiled with debugging turned on, so performance may actually be a little better in a production environment.

We used a beta release of MX-1.2 which has new functions necessary for Lustre including the asynchronous connect, mx_iconnect(), as well as the mx_test_any() and mx_wait_any() completion functions. These test and wait functions are matching-aware, race-free completion functions. MX-1.2 supports all possible vectorial communications (user and kernel) but some types are not recommended because they are inefficient. We used an improved vectorial communication support designed for the special needs of file systems, i.e. sets of entire physical pages incoming from the page cache of the operating system. As with the zero-copy Ethernet patch in Lustre, this enhancement doubles the read and write performance from about 550 to 1170 MB/s.

Lastly, we used the 0.9.0 Ethernet driver and we modified the sysctl.conf file as recommended in the driver's documentation. We also turned on TSO and NAPI is on by default.

5.1 Baseline Performance

To establish the maximum throughput for Ethernet, we ran NetPerf between two nodes directly connected to each other (i.e. back-to-back or switchless). We measured TCP STREAM throughput of 9.156 Gb/s or 1,144.5 MB/s. Since Lustre reports MB/s, we will use that unit of measure for the rest of the paper. We also ran Intel MPI Benchmark (IMB) to look at the PingPong latency and we measured 13.7 us.

In general, Ethernet switches have not been optimized for HPC and we suspected that the switch might limit the performance of the cards. We performed the first Lustre test back-to-back (B2B) to determine what is the maximum performance we can attain using the Myri-10G cards.

We then reran the tests using the switch. The switch does not make much difference for metadata (less than a 2% loss), but it lowers read performance by 12% from 617 to 544 MB/s and it lowers write performance by 10% from 601 to 538 MB/s.

To establish a baseline for MXLND, we ran the Intel MPI Benchmark, a standard HPC micro-benchmark that exercises various MPI calls. The two most frequently cited are PingPong, which measures one-way latency and throughput, and SendRecv, which measures bi-directional throughput. With PingPong, we measured latency of 2.9 us, which is slow for MX, and throughput of 1,149 MB/s. The higher than expected latency is due to additional debugging code we are using with Lustre. With a non-debugging kernel and MX-10G, we measured latency of 2.5 us. We also measured SendRecv throughput at 2,265.8 MB/s, which is slightly more than 90% of line rate. Again, with a non-debugging kernel and MX-10G, we see throughput of more than 2,400 MB/s.

Using the Myri-10G switch, we did not see any impact on metadata, read, or write performance. With and without the switch, read and write performances are within 5% of line rate.

As it turns out, Lustre does not seem to be very latency sensitive. Both the TCP and MX LNDs reach 50,000 metadata transfers per second, which works out to 20 us per metadata (round-trip latency). Given MX's lower latency than Ethernet, if Lustre is latency-sensitive, then we would expect much higher metadata performance for MXLND.

5.2 Single Client/Single Sever

5.2.1 Metadata

First, we ran the metadata test to compare the TCP and MX Lustre Network Drivers. These and the remaining tests all use either the Ethernet switch or the Myrinet switch. This test has the

client request metadata about a file from the server. The client sends a small (less than 4 KB) message to the server and the server replies with another small message. We vary the number of threads (i.e. number of concurrent requests) from 1 to 8, 12 to 32 in steps of 4, and 64. In a production system, however, Lustre will not attempt to use more than the flow control in-flight limit, which is 8 for both of the LNDs.

We see that for 20 threads or less, MXLND outperforms TCP. From 24 threads to 64 threads, however, TCP performs substantially better than MXLND. Since both LNDs limit in-flight messages to 8 and yet we see performance increases up to 20 for MXLND and 24 threads for TCP, we believe that Lustre may be pipelining requests so the LNDs can immediately begin handling the next request as soon as the complete the old request.

In figure 8, we also see that MXLND is using about 20% more CPU than TCP. Initially, we implemented the completion thread in MXLND to block until a send or receive request completes. Using that model, CPU usage was much lower (about 15% for read and 5% for write). For these tests, we added a combination polling/waiting scheme for request completions, which is known to be a good compromise between CPU utilization and latency. We modified MXLND to poll for 10,000 loops (or about 100 us) before blocking. This increased metadata performance by 18% when we used 8 threads and it did not affect read or write.

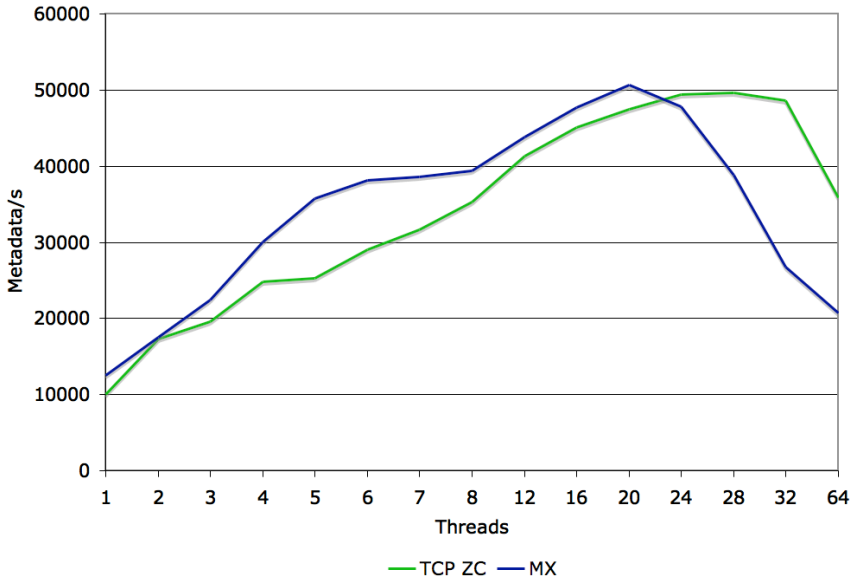

Figure 5: Single client metadata performance

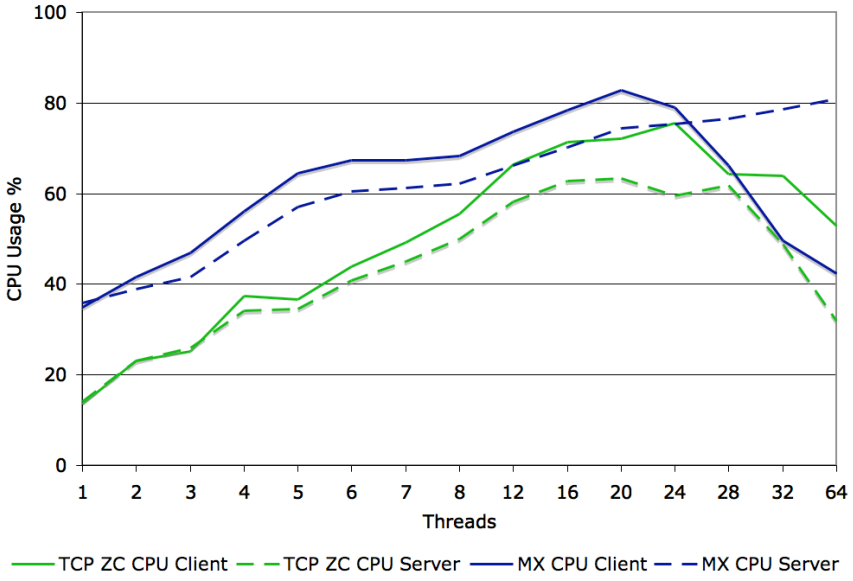

Figure 6: Single client metadata CPU usage

5.2.2 Single Client Read and Write

When we measured the bulk data movement using read and write, we see a clear benefit to using MXLND. A single client can obtain more than twice the throughput for both read and write. Also, both LNDs reach their respective maximums with just 2-4 threads.

CPU usage is completely symmetric between read and write for both LNDs. The TCP server usage is lower than MXLND when reading and the TCP client usage is lower than MXLND when writing. This corresponds to the peer that is benefiting from the TCP zero-copy patch. The remaining tests exhibit similar behavior and the CPU usages remain at these levels.

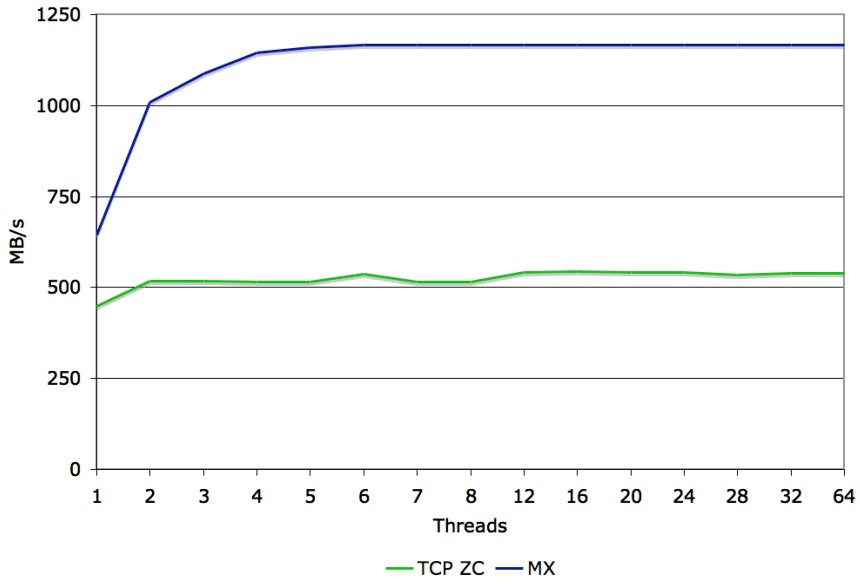

Figure 7: Single client read performance

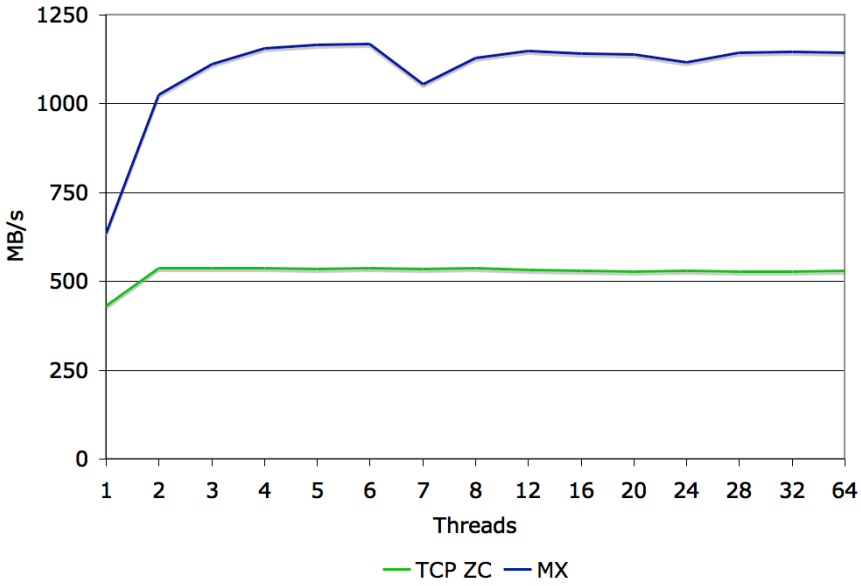

Figure 8: Single client write performance

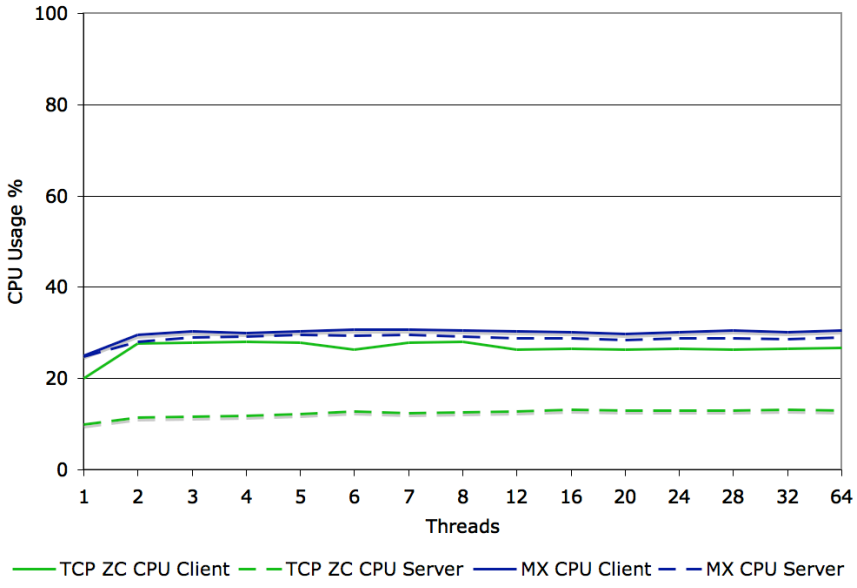

Figure 9: Single client read CPU usage

5.3 Multiple Client/Single Sever

To determine how well a single server handles multiple clients, we ran the tests as in the previous section while adding a second and third client. These tests show the aggregate amount of metadata and MB/s handled by the server. In all cases, the clients share the results evenly with only slight differences. These tests reflect the fact that most Lustre systems will have more clients than servers.

5.3.1 Metadata

When adding more clients, MXLND metadata performance fluctuates. Figure 12 shows the metadata performance for three clients. The performance using two clients falls almost evenly between the single client and three client numbers. Aggregate server performance increases with fewer threads per client. Each client, however, is getting slightly less than it does as a single client.

After 12 threads per client both LNDs start losing performance. MXLND's decline, however, is steeper than TCP's decline. It does seem to level off at ~22K metadata/s. Since the flow control limits each client to 8 messages in-flight, the drop-off might not occur in a production system.

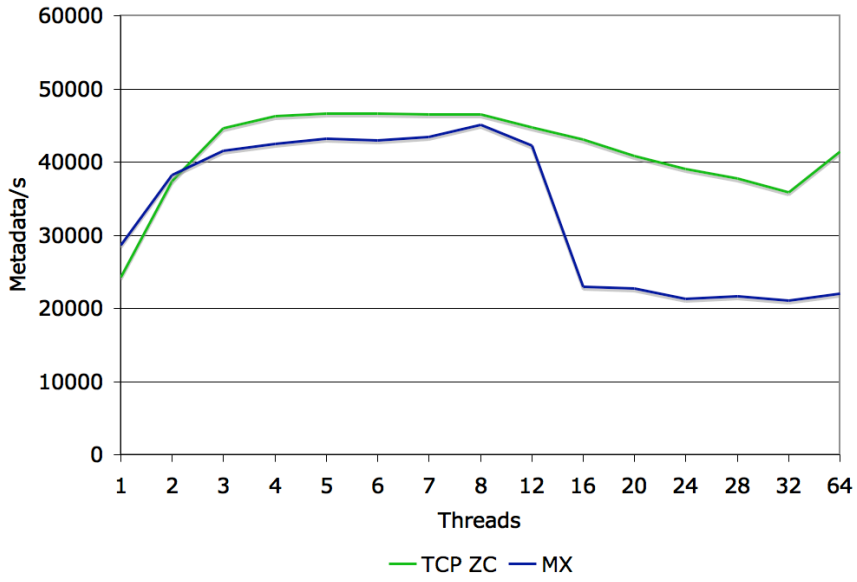

Figure 10: Three client metadata performance

5.3.2 Read

Using two clients increases aggregate TCP performance to 1,000 MB/s and using three clients matches MXLND's performance of ~1170 MB/s.

Since MXLND can nearly saturate the link using a single client, we simply checked to see if the multiple clients would share fairly and they do.

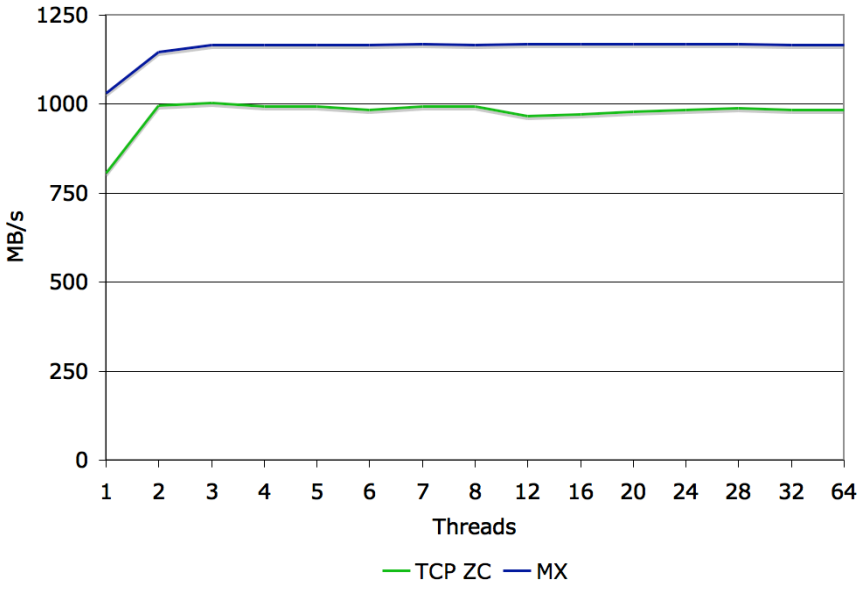

Figure 11: Two client read performance

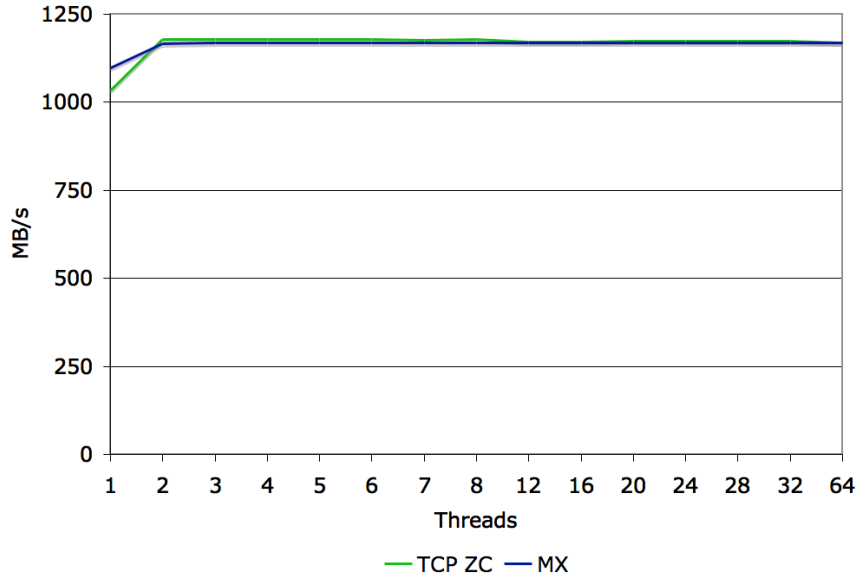

Figure 12: Three client read performance

5.3.3 Write

Unlike reads, however, write performance did not increase with the number of clients for TCP and it stays fairly constant at ~600 MB/s. In discussions with Cluster File Systems' Eric Barton, he indicated that the Lustre zero-copy TCP patch eliminates the memory copy on one side of the

transaction only. Read performance scales up because the clients incur the copy penalty. Writing, however, requires a copy on the server side and this limits performance regardless of how many clients write.

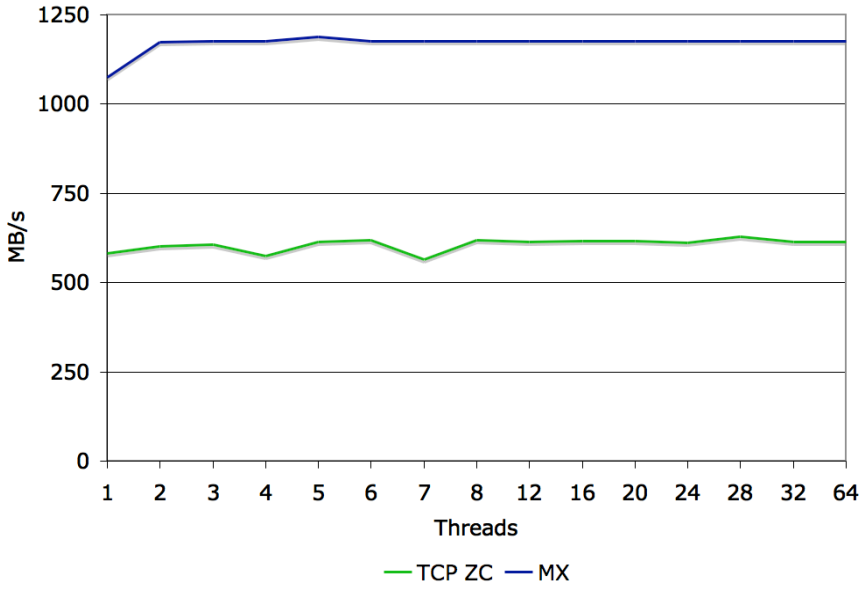

Figure 13: Three client write performance

6. Conclusions and Perspectives

Overall, both the TCP and our MX LNDs achieve high-levels of performance. We have shown that both can nearly saturate the link when reading and MXLND can saturate it when writing. Both can serve metadata at 50K messages per second. These results use the Lustre lctl tool and we intend to benchmark a mounted file system as well. We anticipate that benchmarking the mounted system will be much harder since, as usual for distributed file systems, lots of variables are involved, such as parallel access, caching, physical access to the disk, etc.

We need to refine the use of polling to achieve a trade-off between performance and CPU usage or allow system administrators to select the policy they want per node (e.g. blocking for clients and polling for servers).

Also, unlike most LNDs, which spawn a worker (or scheduler) thread for each CPU, MXLND only uses one completion thread. Based on short tests, we saw better performance using one thread versus using one thread per processor (or core). This may be due to lock contention or cache effects. We need to investigate this further. Given that a single thread easily saturates the link for read and write, it may not be necessary to spawn multiple threads unless doing so provides some other benefit (e.g. higher metadata performance or lower overall CPU usage).

We need to investigate the metadata performance declines for higher levels of threads. In discussions with CFS, they mention that Lustre does not use more threads than the message queue depth. If so, then this is not an issue since performance does not decline until 16 threads and the message queue depth is 8 for these tests.

We intend to run similar tests using Infiniband NICs to compare MXLND to other LNDs using other high-speed networks and will include those results in the revised paper.

Finally, as Ethernet grows in HPC and as 10 Gb/s Ethernet becomes more widely available, more organizations will be looking to run Lustre and other distributed file systems on TCP. While the zero-copy patch allows servers to saturate the link when multiple clients are reading, the server's need to copy the incoming data in memory will limit writing to roughly 50% of link rate on a 10 Gb/s Ethernet network. Another consideration when using Ethernet will be finding switches that allow full use of the fabric. In our tests in this paper, we showed that the switch can reduce single client read and write performance by ~10%. Prior to running these tests, we tried using a different switch from another company and measured a 20% decrease in performance. Lastly, for large clusters, finding high-density (high port count) 10 Gb/s Ethernet switches might limit Ethernet's use in this application.

7. Acknowledgements

We would like to thank Eric Barton at Cluster File Systems for his technical advice and support. We also wish to thank Brett Ellis for setting up the test clusters, patching and compiling numerous kernels, and providing invaluable support throughout this process.

8. References

Bhat04 Viraj Bhat et al, *High Performance Threaded Data Streaming for Large Scale Simulations,* in the proceedings of 5th IEEE/ACM International Workshop on Grid Computing, Pittsburgh, PA, USA, Nov 2004.

myri Nanette J. Boden and Danny Cohen and Robert E. Felderman and Alan E. Kulawik and Charles L. Seitz and Jakov N. Seizovic and Wen-King Su, *Myrinet: A Gigabit-per-Second Local Area Network*, IEEE Micro, Vol. 15, Number 1, pp. 29-36, 1995.

Pfister01 Gregory F. Pfister, *Aspects of the InfiniBand Architecture*, Proceedings of the 2001 IEEE International Conference on Cluster Computing, pp. 369-371, Monterey, CA, USA, Oct. 2001.

Ligon01 Walt Ligon, *Next Generation Parallel Virtual File System*, Proceedings of the 2001 IEEE International Conference on Cluster Computing, Newport Beach, CA, USA, Oct. 2001.

Schmuck02 Frank Schmuck and Roger Haskin, *GPFS: A Shared-Disk File System for Large Computing Clusters*, Proceedings of the Conference on File and Storage Technologies (FAST'02), USENIX, pp. 231-244, Monterey, CA, USA, Jan. 2002.

Schwann03 Philip Schwann, *Lustre: BBuilding a File System for 1,000 Node Clusters*, Proceedings of 2003 Linux Symposium, pp. 401-408, Ottawa, CA, USA, July 2003.

10G *Myri-10G Overview*, http://www.myri.com/Myri-10G/overview/

MX Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet, July 1, 2005, Myricom Inc., http://www.myri.com/scs/MX/doc/mx.pdf

mpi Message Passing Interface Forum, http://www.mpi-forum.org/

Eicken92 von Eicken, T. and D. E. Culler and S. C. Goldstein and K. E. Schauser, *Active Messages: a Mechanism for Integrated Communication and Computation*, Proceedings of the 19th Int'l Symposium on Computer Architecture, Gold Coast, Australia, May 1992.

Goglin05 Brice Goglin, Olivier Glück and Pascale Vicat-Blanc Primet, *An Efficient Network API for in-Kernel Applications in Clusters*, IEEE International Conference on Cluster Computing, Boston, Massachusetts, USA, September 27-30, 2005.

GPFS IBM General Parallel File System, http://www-03.ibm.com/servers/eserver/clusters/software/gpfs.html

Portals Ron Brightwell, Arthur B. Maccabe, Rolf Riesen and Trammell Hudson, *The Portals 3.3 Message Passing API*, May 16, 2003, http://www.cs.sandia.gov/~ktpedre/portals/docs/portals3.pdf