SAND2008-1241C

Instrumentation and Analysis of MPI Queue Times on the SeaStar High-Performance Network

Ron Brightwell* Kevin Pedretti Kurt Ferreira
Scable System Software Department
Sandia National Laboratories
Albuquerque, New Mexico 81785–1319
{rbbrigh,ktpedre,kbferre}@sandia.gov

***Abstract*—Understanding the communication behavior and network resource usage of parallel applications is critical to achieving high performance and scalability on systems with tens of thousands of network endpoints. The need for better understanding is not only driven by the desire to identify potential performance optimization opportunities for current networks, but is also a necessity for designing next-generation networking hardware. In this paper, we describe our approach to instrumenting the SeaStar interconnect on the Cray XT series of massively parallel processing machines to gather low-level network timing data. This data provides a new perspective on performance evaluation, both in terms of evaluating the resource usage patterns of applications as well as evaluating different implementation strategies in the network protocol stack.**

I. Introduction

This study is a continuation of previous research into the network resource usage characteristics of scientific parallel applications on distributed-memory massively parallel processing machines. Our initial motivation for this work was a result of our collaboration with Cray, Inc. to design a new high-performance network for Sandia's Red Storm machine [1], which is the prototype of what has become the commercially successful Cray XT series of machines. Cray's initial design for the SeaStar — a network interface and seven-port router chip — presented some significant challenges for our anticipated use of the network. In particular, the SeaStar contains a 500 MHz processor and only 384 KB of on-board RAM. Since we anticipated using the processor and memory for message processing activities, it was unclear whether the speed and limited memory capacity would be able to support the demanding communication requirements of our applications.

In our initial study [2], we analyzed network resource usage characteristics important for offloading MPI protocol processing. We instrumented the MPI implementation for the Myrinet [3] high-performance network to gather data that would help characterize the processing and memory requirements of applications. We began by characterizing the behavior of a well-known parallel application benchmark suite. In a subsequent study [4], we used the same approach to analyze the behavior of several of Sandia's important parallel applications. Following that, we extended our instrumentation framework to differentiate between MPI point-to-point and collective communication resource usage [5].

While these studies provided much needed information about applications, many questions were still left unanswered. Our initial results were obtained on Myrinet cluster, so it was unclear whether we would see the similar behavior with a network that is capable of offloading a significant amount of message processing. Relative to some other parallel computing platforms, the Myrinet cluster was unbalanced in terms of the amount of compute power per node relative to the peak injection bandwidth per node. We were anticipating that Red Storm would be a much more balanced system and were unsure what impact this would have on network resource usage. We also were not able to do the level of instrumentation that was needed to fully characterize certain behavior. For example, we were able to easily determine the number of incoming messages, but we were unable to determine when a message had arrived from the underlying network relative to when the application had requested to receive it.

After working together with Cray to develop and deploy the Red Storm system, we can now use this platform for further explorations in this area. By applying a similar instrumentation strategy for the SeaStar and extending it to support more detailed information, we can attempt to answer these questions. This type of data has also been critical for research into designing next-generation high-performance network hardware [6], [7].

In this paper, we describe our approach to gathering low-level network performance data that we can use to characterize the network resource usage of applications on a highly-balanced massively parallel processing machine. In particular, we extend the existing network to place timestamps on incoming messages in order to have a more fine-grained analysis of network and application behavior. The rest of this paper is organized as follows. The next section provides background on the SeaStar network hardware and software stack, while Section III describes our approach to instrumentation. The details of our test platform and the applications from which we gathered performance data are describe in Section IV. Performance results and analysis are provided in Section V.

*Corresponding author. Phone: (505)844-2099 FAX: (505)845-7442.
Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

We summarize the conclusions of this study in Section VI and discuss avenues of future work in this area in Section VII.

II. Background

A. MPI

The demands of a high-performance network for large-scale parallel computing systems are driven by the way in which applications use MPI. While MPI is typically not the only upper layer protocol that makes use of the underlying high-performance network, it is the most important for two reasons. First, MPI performance has the largest influence on the performance and scalability of network-bound applications. Secondly, MPI is typically the only upper layer protocol where the application programmer directly influences the use of the network at both the source and destination. Other upper layer protocols are usually provided by system services where the application is only a client. Despite the presence of other upper layer protocols, it is MPI that exudes the most stress on network resources.

Conceptually an MPI implementation has two message queues — one that contains a list of outstanding receive requests (the posted receive queue) and one that contains a list of messages that have already arrived for which there is no matching receive request (the unexpected or early arrival queue). The posted receive queue must be traversed each time a new message arrives, while the unexpected queue must be traversed each time a receive request is posted. This latter operation — searching the unexpected queue and posting a receive request — must be an atomic operation to insure the pairwise ordering semantics of MPI. To our knowledge, all MPI implementations implement these two queues as linear lists. While other strategies, such as hash tables, are possible, their use is inhibited by the fact that MPI allows for "wildcarding" source and message tag so that a posted receive can match any one of several incoming messages. As such, a hash table approach has the potential to speed up search operations, but prohibitively increases the cost of insertion and deletion operations, particularly with respect to zero-byte ping-pong latency performance. In terms of network processing capability, it is the management of these two queues that largely determines the processing requirements of a network.

B. SeaStar

The SeaStar is an ASIC from Cray, Inc., that combines a network interface and a high-speed seven-port router on a single chip. The SeaStar is connected to an AMD Opteron processor via a HyperTransport link. The current-generation SeaStar is capable of sustaining a peak unidirectional injection bandwidth of more than 2 GB/s and is able to sustain a peak unidirectional link bandwidth of more than 3 GB/s. Each SeaStar contains a 500 MHz PowerPC that can be used to perform protocol processing activities. Each SeaStar also has 384 KB of on-board RAM. For a more detailed discussion of the SeaStar network, see [8].

C. Portals

The lowest-level network programming interface for the SeaStar is the Portals data movement layer [9]. Portals were designed to be protocol building blocks that could be assembled to implement a number of upper layer protocols. In particular, Portals were designed specifically to support a scalable, high-performance implementation of MPI. We briefly describe the relevant Portals structures that are used to implement MPI. See [10] for a more complete discussion.

Portals provides one-sided data movement operations, but unlike other one-sided programming interfaces, the target of a remote operation is not a virtual address. Instead, the ultimate destination of a message is determined at the receiving process by comparing contents of the incoming message header with the contents of Portals structures at the destination.

When an incoming message arrives at the destination, the message header contains a destination Portal number which is used to differentiate between upper layer protocols. MPI allocates a specific Portal for matching incoming receives. Attached to the Portal is a match entry (ME) that contains a set of criteria that must match the incoming header. Messages can be selected based on source node id (nid), source process id (pid), and 64 bits of message tag. The destination can wildcard nid and pid and also has 64 mask bits that can be used to identify a subset of the tag bits to be used for matching.

Attached to each ME is a memory descriptor (MD) that describes the location in memory where the incoming message is to be deposited. There are a number of options associated with each MD that determine how the matching message is consumed. For example, the MD has an option to truncate the incoming message to receive only as much data as the receiver has requested. MDs also have a threshold value that determines whether the MD can be used only once or used multiple times.

An event queue (EQ) can be attached to an MD to record the operations that have occurred on the MD. The EQ is a circular queue of entries that captures the relevant state of the MD at the time the operation completed. Portals has a split event model where an event is generated when an operation is started (START event) and a subsequent event is written when the operation is completed (END event). This allows for recognizing situations where a short message may arrive after a long message but may complete before the long message. It is also used to identify transfers that may have started successfully but encountered a failure at some later point. MDs have to the option to turn off either START or END events or ignore events altogether by not attaching an EQ to the MD.

D. Portals on the SeaStar

Because Portals is the lowest-level network programming interface for the SeaStar, it was important to have a working implementation as soon as possible so that other areas of software development for the Red Storm machine could progress. The initial implementation of Portals for the SeaStar was developed by Cray and was based on the reference implementation from Sandia. In order to get something working quickly,

Cray chose an interrupt-driven strategy. When a message arrives at the SeaStar, it copies the message header into kernel space and interrupts the host processor. The host operating system is then responsible for inspecting the message header, traversing the Portals data structures, and programming the DMA engines on the SeaStar to deliver the contents of the message directly into user space.

While Cray to continued to develop their interrupt-driven implementation, Sandia worked on an implementation that would do all of the Portals message processing work using the SeaStar's on-board PowerPC processor and avoid any host processor involvement in message reception. Due to the limited processing capability and available memory on the SeaStar, it was unknown whether this strategy would have the ability to support the demands of tightly-coupled parallel applications using tens of thousands of endpoints. It was this concern over the limited capabilities of the SeaStar that motivated our initial work into characterizing the network resource usage characteristics of MPI applications.

Sandia's implementation of Portals using the SeaStar's processor has since been integrated into Cray's production software environment so that it is possible to switch between using the interrupt-driven implementation, called Generic Portals (GP), or the offloaded version, called Accelerated Portals (AP), when a parallel job starts. Micro-benchmark results show that AP has some significant advantages over GP. In particular, AP has a 1-2μs latency advantage, a steeper bandwidth curve, and much less host processor overhead [8]. However, no formal study comparing application performance between the two approaches has ever been conducted. Informal testing at Sandia has shown that some applications can see a 10-15% improvement in performance, but, for most applications, the performance differential is not significant. For the applications in this study, the difference between AP and GP is not significant.

III. Approach

We are interested in the following measurements related to the MPI queues:

- Average number of items that were inspected each time the queue was searched.
- Maximum number of items that were inspected.
- Maximum number of items in the queue at any one time.

For the posted receive queue, the average items searched gives us an idea of how many items must be looked at before a matching receive is found. Ideally, each incoming message would match the first entry in the list. The maximum number of items that were searched places an upper bound on the number of entries that an unexpected message had to traverse. The maximum length of the posted receive queue provides an upper bound on the number of outstanding receive requests that the application has at any given time. This data point helps us characterize the amount of memory that is needed to hold these requests.

Similarly, for the unexpected queue, the average number of items that were inspected gives some insight into the resources that need to be given to handling unexpected messages. As the number of items in the unexpected message queue grows, the longer it takes to search this queue before posting a receive request. The maximum number of items inspected and the maximum length of the unexpected queue give us an idea of the resources that are needed to search this queue.

One of the limitations of our previous work on MPI queue analysis is that we had no way of knowing the arrival times of messages. We could easily count the number of expected and unexpected messages, but we had no way to determine the arrival time of a message relative to the time that the application posted a receive request. We therefore extended our instrumentation environment with message timestamps to collect this type of data. We are now able to determine how long a message sat in the posted receive queue before being matched or how long a message sat in the unexpected queue before it was requested by the application.

We also added instrumentation to the MPI library to count the number of times that Portals failed to post a receive due to subsequent incoming messages. Since the process of searching the unexpected queue and posting a receive must be atomic, Portals provides a function that will conditionally insert an ME into match list provided there are no pending events on an event queue. The MPI library is responsible for processing all of the events in the unexpected event queue. If no match is found, then it calls the conditional insert function. If this conditional insert fails, it means that a message has come into the unexpected message queue and MPI must first see if this is the matching message. We are interested in measuring how often this fails to help determine whether the effectiveness of this approach to getting atomicity.

A. Timestamps

Unlike our previous approach to instrumenting MPI to gather data on queue processing and message arrival data, all of the necessary information is not at the user-level inside of MPI. In the MPI implementation for Myrinet, the MPI library is responsible for managing both the posted receive and unexpected message queues. However, for the Portals MPI implementation, the posted receive queue is completely managed by Portals, so we needed to extend our instrumentation into the Portals implementation and provide a mechanism for MPI to retrieve this information. Here we describe the instrumentation that was added to Portals and the mechanisms that the MPI implementation uses to collect and record the data.

The instrumentation for the posted receive queue is straightforward. We keep a running total of the number of times the match entry list on the MPI receive Portals was traversed. Each time we traverse the list, we keep a count of the number of items that were inspected. We add this value to a total count in order to calculate the average, and we also compare it to a maximum value.

We extended the Portals API in four different ways. First, we added a timestamp field to the event queue entry so that timestamps could be recorded with each event. We also added three new function calls. The first new call provides a

mechanism for getting the current time from the Portals layer. The second new call allows us to retrieve the time when the last match entry as added was added into a match list. Finally, we added a call that provides a current snapshot of the queue instrumentation data.

Adding timestamps to the GP implementation was relatively straightforward. We decided to use the processor timestamp counter on the Opteron, since the kernel does all message processing and is responsible for writing the event into user-space. Using the processor's timestamp counter also allowed us to implement the Portals network time function without a system call, which is much more costly and could add significant perturbation.

Adding timestamps to the AP implementation was slightly more challenging. We used the timestamp counter on the PowerPC for timestamping events. In order to provide this time to the application, we mapped an area of SeaStar memory into user space and had the SeaStar firmware update the mapped memory location with the current time on every iteration through its main loop. An idle pass through the main loop takes about 200 ns.

Retrieving the timestamp of the last entry that was added to a match list was also straightforward. In order to record how long a receive request had been sitting in the posted receive queue, we needed to know the time when it had been inserted in the match list. Our first attempt at this was to record the creation time of the ME inside the Portals implementation and then provide a function call to return this time for a specific ME. After a successful atomic search and update, we immediately call this function to get the ME creation timestamp. Unfortunately, we did not account for a race condition where the ME could be inserted and then immediately matched and deleted before the application could call back into the library to get the creation time. So rather than return the creation time of a specific ME, the Portals implementation just keeps track of the time the last ME was created.

IV. Test Environment

In this section we provide an overview of the hardware and software environment of our test system and briefly describe the applications from which performance data was collected.

A. Platform

The machine used in our experiments is a Cray XT3/4 development system with 80 compute nodes. Each compute node contains a 2.4 GHz dual-core AMD Opteron processor and 2 GB of RAM. The software environment is based on Cray software release 2.0.35 running the Catamount lightweight kernel operating system. We provide performance results running in both single-node (SN) mode, where only one of the processing cores on a node is used, and virtual-node (VN) mode, where both cores are active. It should be noted that Catamount does not support shared-memory communication, so messages between the processes on the same node in VN mode use Portals. The Cray MPI implementation sends short messages, those less than 128 KiB, eagerly, and uses a rendezvous protocol for larger messages.

B. Applications

We have collected results from several applications that are an important part of Sandia's modeling and simulation workload, including CTH, SAGE, and ITS. We also include results from HPCCG, which is a mini-application designed to mimic the message passing and computation behavior of several much larger and complex applications. We mad three runs of each application for each configuration (SN/VN mode, AP/GP mode) for each processor count from two to 160.

CTH is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia. CTH has models for multi-phase, elastic viscoplastic, porous and explosive materials. Three-dimensional rectangular meshes; two-dimensional rectangular, and cylindrical meshes; and one-dimensional rectilinear, cylindrical, and spherical meshes are available. It uses second-order accurate numerical methods to reduce dispersion and dissipation and to produce accurate, efficient results. CTH is used for studying armor/anti-armor interactions, warhead design, high explosive initiation physics, and weapons safety issues.

SAGE, SAIC's Adaptive Grid Eulerian hydrocode, is a multi-dimensional, multi-material, Eulerian hydrodynamics code with adaptive mesh refinement that uses second-order accurate numerical techniques [11]. It represents a large class of production applications at Los Alamos National Laboratory. It is a large-scale parallel code written in Fortran 90 and uses MPI for inter-processor communications. It routinely runs on thousands of processors for months at a time.

The Integrated Tiger Series (ITS) code, is a radiation transport Monte Carlo code [12], [13]. ITS permits Monte Carlo solution of linear time-independent coupled electron/photon transport radiation transport problems, with or without the presence of macroscopic electric and magnetic fields of arbitrary spatial dependence. Physical rigor is provided by employing accurate cross sections, sampling distributions, and physical models for describing the production and transport of the electron/photon cascade from 1.0 GeV down to 1.0 keV. The sample problem set performs the Starsat MITS test with CAD flow and geometry and ACIS simulation mode.

The HPCCG [14] mini-application is a simple conjugate gradient solver that represents an important workload for Sandia. It is commonly used to characterize the performance of new hardware platforms that are under evaluation. The majority of its runtime is spent in a sparse matrix-vector multiply kernel. We ran HPCCG in both strong scaling (fixed problem size) and weak scaling (scaled problem size) modes.

V. Results

The amount of data that we have collected is too large to cover in great detail in this study. Therefore, we limit our analysis to only the data which our previous approach using Myrinet did not support — namely low-level queue timing data. In general, the data that we gathered relative to the

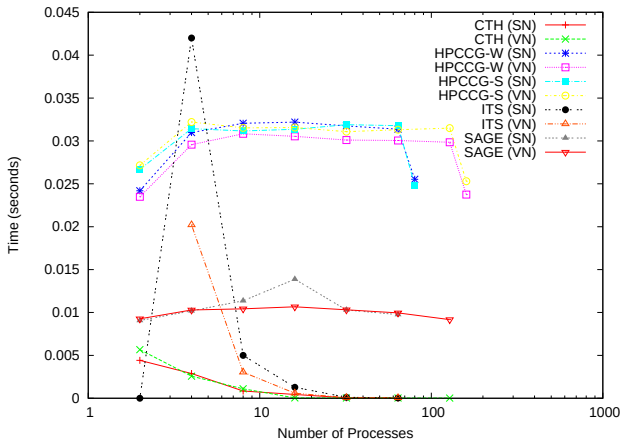

Fig. 1. Short Unexpected Average Queue Time

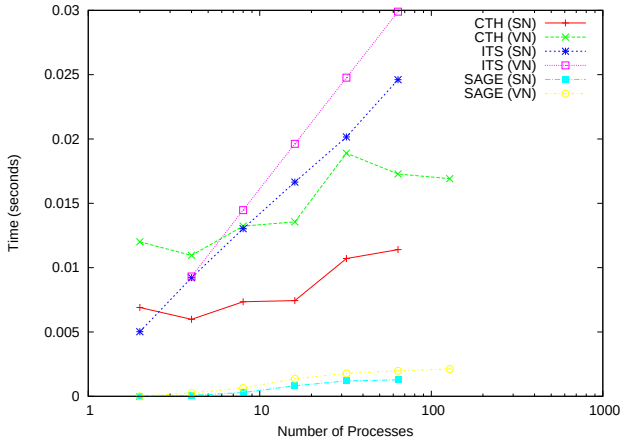

Fig. 2. Long Expected Average Queue Time

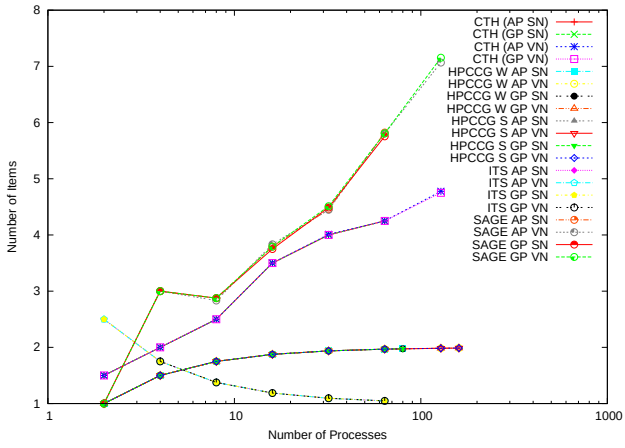

Fig. 3. Posted Queue Max Length

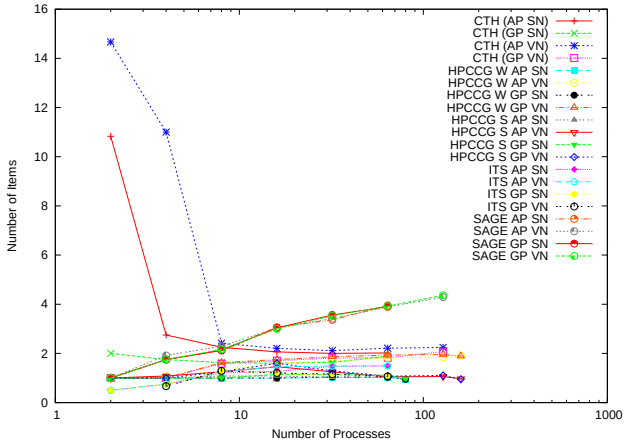

Fig. 4. Unexpected Queue Max Length

percentage of expected and unexpected messages is consistent with our previous study and does not appear to be impacted by the balance of the system.

A. Short Unexpected Average Queue Time

Figure 1 shows the average time that a message sat in the unexpected short message queue for the GP implementation. In general, there was little difference between GP and AP. We would hope that the queue time for a short unexpected message is very short, since short unexpected messages tend to consume buffer space and require memory copies. A short queue time means that the application just missed being able to post a matching receive. We can see that the queue time is generally the same between SN and VN modes for most applications, except in a few cases with ITS and SAGE where the SN times are significantly longer than the VN times.

B. Long Expected Average Queue Time

Figure 2 shows the average time that a long expected message sat in the MPI posted receive queue. These numbers are for the GP implementation, as again the AP implementation showed little difference. HPCCG did not send any messages larger than 128 KiB. There are clearly some trends visible in this graph. In all cases, there's a consistent difference between SN and VN modes, with the queue times for VN mode consistently greater. The time for ITS scales with the number of processes in the job, while the others tend to flatten out. The data for SAGE seems to indicate that it is able to more quickly retire long posted receive messages – perhaps without any synchronization messages.

C. Maximum Queue Length

Figure 3 shows the maximum length of the posted receive queue. We can see that the behavior this queue shows little dependence on whether we are running in GP or AP mode or SN or VN mode. Interestingly, the maximum length of the posted receive queue for ITS is on a downward trend, whereas the others are not. The maximum length for SAGE seems to scale with the number of processes in the job, so it will be interesting to see if this trend continues to larger process counts.

The maximum length of the unexpected queue is shown

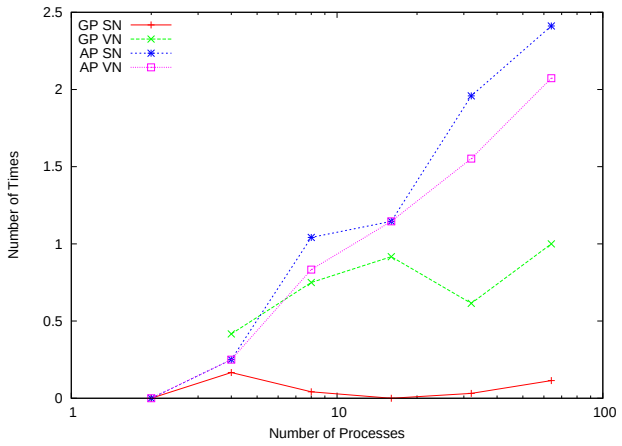

Fig. 5. Post failures - ITS

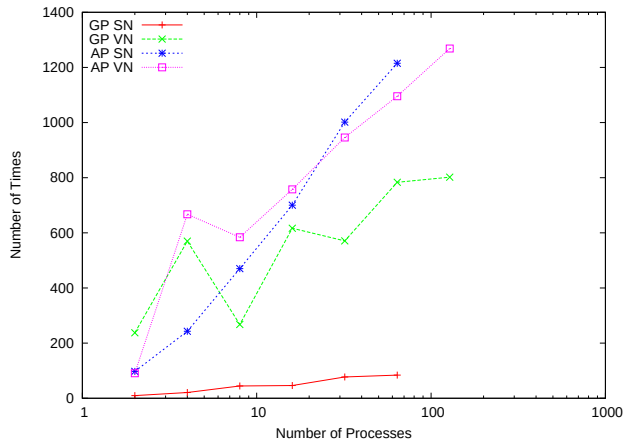

Fig. 7. Post failures - SAGE

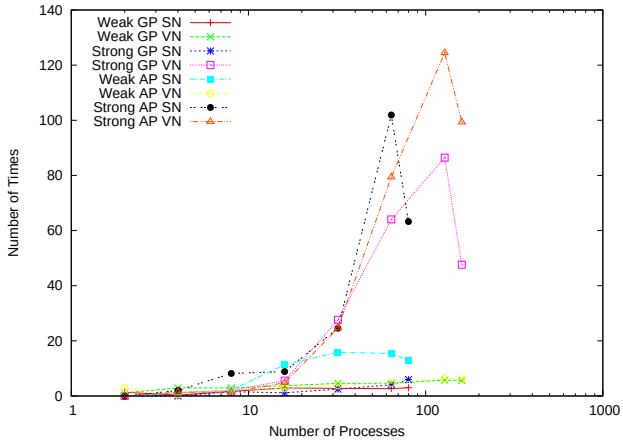

Fig. 6. Post failures - HPCCG

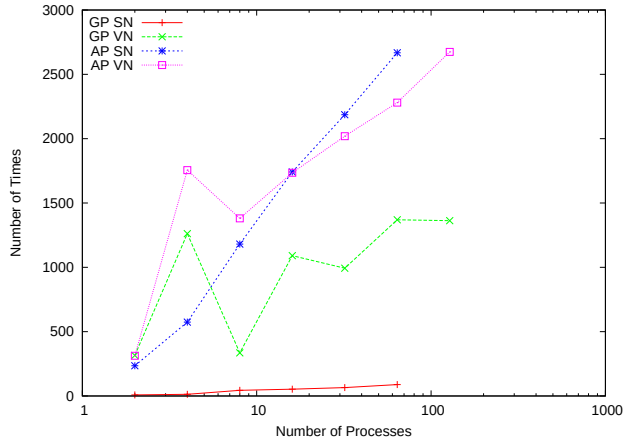

Fig. 8. Post failures - CTH

in Figure 4. Again, there does not seem to be a significant difference between the various modes for these applications, and the queue for SAGE appears to scale with the number of processes in the job. Overall, these results for maximum queue length agree with our previous results on Myrinet.

D. Post Failures

One of the more interesting and unexpected data points that we discovered as a result of our instrumentation concerns the number of times that MPI failed to successfully post a receive because the atomic search-and-post operation failed. Figure 5 shows the number of times that ITS failed to post a receive due to an incoming message. ITS sends and receives a relatively small number of messages compared to the other applications, so this small number of failures is not too surprising.

Figure 6 shows the number of post failures for HPCCG. Here we see a significantly larger number of failures, with the worst case being AP VN mode. Overall the number of failures is still fairly reasonable.

Continuing on, we see the same data for SAGE in Figure 7. Again, we see that the worst case is for AP VN mode. At 64 processes, the number of failures between AP and GP in SN mode is nearly a factor 12. We also notice that trend for AP and VN mode is increasing.

Finally, we see the same data for CTH in Figure 8. The curves in this graph are nearly identical to those for SAGE, except that the scale is much higher. This trend is disturbing, and much more investigation is needed to determine why AP and VN have so many more failures compared to GP SN mode. This is perhaps one of the reasons that the micro-benchmark performance increase seen with AP over GP is not reflected in actual application performance.

VI. Conclusions

In this paper, we have described our approach to instrumenting the Portals implementation for the Cray SeaStar in order to gather low-level MPI message and queue information. While our approach was similar to previous work that was done for the Myrinet network, the SeaStar network provided a more balanced system on which to collect performance data and provided a chance to compare and contrast our previous results with a network that is able to offload a significant part of the

message processing stack. In general, however, our results for the SeaStar are consistent with our previous results. Neither the balance of the machine nor the offloading ability of the network seemed to have an impact on the message passing behavior of the applications that we studied.

Overall, there were few differences in the queue statistics between the Generic and Accelerated implementations of Portals. The queue behavior seems to be inherent in the application and immune to whether message processing is done by the host processor or offloaded to a network processor. This is generally true for single-node versus virtual-node mode as well.

The most interesting difference between the various modes that we have discovered is the difference in the number of posted receive queue failures. A few of the applications we tested showed a significant difference in the number of failures when using VN and AP modes – as much as a factor of twelve in some cases. Clearly this needs further investigation, and we hope to augment our existing instrumentation to provide further details about this discrepancy.

VII. Future Work

We intend to continue to use the instrumentation infrastructure described in this paper to gather data on more applications. The applications we selected for this study represent a significant part of Sandia's workload, but there are several others that we would like to analyze as well. We also expect to get dedicated time on the production Red Storm machine, which currently has more than 13 thousand nodes, to continue to explore the impact of scale of low-level network behavior. We also plan to explore some methods that will allow for better instrumentation. Rather than just keeping running averages and maximums of some values, we would like to be able to keep more complete information to better understand the distribution of data points and try to correlate them with specific parts of the application. We would also like to be able to separate out data for collective communication from point-to-point communication as we did in previous studies using Myrinet. Unfortunately, this will likely require us to make some changes to both the MPI implementation and our current instrumentation framework so that collective communication operations can be identified correctly inside the Portals layer. We are also interested in exploring how protocols within the MPI implementation impact this type of performance data. The Cray MPI implementation used in this study is one of several that are available for Portals. There are other implementations that employ different strategies and protocols to achieve the same semantics. For example, past studies have shown that sending all messages eagerly can potentially lead to an increase in application performance, and we would like to explore how this change impacts MPI queue data. We are also exploring the possibility of adding network timestamp data to InfiniBand so that we can gather similar types of information for our applications on large-scale commodity-based cluster machines.

References

[1] W. J. Camp and J. L. Tomkins, "Thor's hammer: The first version of the Red Storm MPP architecture," in *In Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD, November 2002.

[2] R. Brightwell and K. D. Underwood, "An analysis of NIC resource usage for offloading MPI," in *Proceedings of the 2004 Workshop on Communication Architecture for Clusters*, Santa Fe, NM, April 2004.

[3] N. J. Boden, D. Cohen, R. E. F. A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, Feb. 1995.

[4] R. Brightwell, S. Goudy, and K. D. Underwood, "A preliminary analysis of the MPI queue charateristics of several applications," in *Proceedings of the 2005 International Conference on Parallel Processing*, June 2005.

[5] R. Brightwell, S. Goudy, A. Rodrigues, and K. Underwood, "Implications of application usage characteristics for collective communication offload," *International Journal of High-Performance Computing and Networking - Special Issue: Design and Performance Evaluation of Group Communication in Parallel and Distributed Systems*, vol. 4, no. 3/4, 2006.

[6] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell, "A hardware acceleration unit for MPI queue processing," in *19th International Parallel and Distributed Processing Symposium (IPDPS'05)*, April 2005.

[7] K. D. Underwood, A. Rodrigues, and K. S. Hemmert, "Accelerating list management for MPI," in *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, September 2005.

[8] R. Brightwell, T. Hudson, K. T. Pedretti, and K. D. Underwood, "SeaStar interconnect: Balanced bandwidth for scalable performance," *IEEE Micro*, vol. 26, no. 3, May/June 2006.

[9] R. Brightwell, T. Hudson, K. Pedretti, R. Riesen, and K. Underwood, "Implementation and performance of Portals 3.3 on the Cray XT3," in *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, September 2005.

[10] R. Brightwell, A. B. Maccabe, and R. Riesen, "Design, implementation, and performance of MPI on Portals 3.0," *International Journal of High Performance Computing Applications*, vol. 17, no. 1, pp. 7–20, Spring 2003.

[11] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings, "Predictive performance and scalability modeling of a large-scale application," in *Proceedings of the ACM/IEEE International Conference on High-Performance Computing and Networking (SC'01)*, November 2001.

[12] B. C. Franke *et al.*, "ITS version 5.0: The Integrated TIGER Series of coupled electron/photon Monte Carlo transport codes with CAD geometry revision 1," Sandia National Laboratories, Tech. Rep. SAND2004-5172, September 2005.

[13] M. Rajan *et al.*, "Performance analysis, modeling and enhancement of Sandia's Integrated TIGER Series (ITS) coupled electron/photon Monte Carlo transport code," in *Proceedings of the Los Alamos Computer Science Institute Symposium*, October 2005.

[14] M. Heroux, "HPCCG MicroApp," July 2007, http://www.cs.sandia.gov/~maherou/HPCCG-0.3.tar.gz.