

An Analysis of NIC Resource Usage for Offloading MPI

Ron Brightwell Keith D. Underwood
Sandia National Laboratories*
PO Box 5800
Albuquerque, NM 87185-1110
{rbrigh, kdunder}@sandia.gov

Abstract

Modern cluster interconnection networks rely on processing on the network interface to deliver higher bandwidth and lower latency than what could be achieved otherwise. These processors are relatively slow, but they provide adequate capabilities to accelerate some portion of the protocol stack in a cluster computing environment. This offload capability is conceptually appealing, but the standard evaluation of NIC-based protocol implementations relies on simplistic microbenchmarks that create idealized usage scenarios. In this paper, we evaluate characteristics of MPI usage scenarios using application benchmarks to help define the parameter space that protocol offload implementations should target. Specifically, we analyze characteristics that we expect to have an impact on NIC resource allocation and management strategies, including the length of the MPI posted receive and unexpected message queues, the number of entries in these queues that are examined for a typical operation, and the number of unexpected and expected messages.

1. Introduction

A large number of modern cluster computers leverage high performance network interfaces that include a programmable processor. These range from low-end Gigabit Ethernet cards with 88 MHz processors to Myrinet [3] and Quadrics [20] cards with much more computing power. NIC processors have been touted as an opportunity to significantly improve MPI processing (ranging from simple protocol offload to offloading large portions of the MPI stack) for cluster computers. Indeed, eliminating the PCI bus from much of the MPI processing path has significant advantages. Numerous microbenchmarks have been used to

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

illustrate the improvements in bandwidth and latency that can be achieved; thus, aggressive supercomputer designs such as Red Storm[1] have chosen to effectively offload most of the MPI matching semantics by using Portals[5]. It should be expected that with the growth in silicon area available to NICs that even more of the MPI stack will be absorbed; however, NIC-based programmable processors are currently much slower than their host processor counterparts. With the limited power budgets typically available to the NIC, this trend is expected to continue. The impacts of this disparity are not obvious in standard microbenchmarks and the implications can be significant.

One of the problems with standard microbenchmarks is that they only evaluate networks under idealized usage scenarios. Typically, latency is only evaluated with one item in the posted receive queue, and bandwidth is measured only with pre-posted receives. Real applications have multiple items in the MPI posted receive queue and searching the posted receive queue for a match increases latency. Similarly, real applications have unexpected messages that can impact MPI performance in various ways. These effects can be exaggerated when portions of the protocol are offloaded onto a relatively slow NIC processor.

Surprisingly, there have been no studies of the parameter space that real applications create. Designers of NIC offload engines target improved bandwidth and latency in idealized scenarios because full application benchmarks tend to obscure the magnitude of improvements their efforts have yielded. A doubling of bandwidth is much more impressive than the corresponding 5% improvement in application execution time it might produce.

This work seeks to fill a critical gap in the understanding of the way applications use the network. A variety of data was collected from the NAS Parallel Benchmark suite[2] including:

- the length of the MPI posted receive queue
- the length of the MPI unexpected message queue

- the number of entries in each queue that are examined for a typical operation
- the percentage of expected and unexpected messages

This data can have a significant impact on the resources that are needed on a NIC, the way those resources are managed, and the portion of the MPI stack that should be offloaded. We believe this type of data can be used in numerous ways. For example, it can be useful to the networking community to help in designing new network interfaces, developing new protocol processing strategies, and implementing new benchmarks to measure the performance of the network under realistic loads. It may also be important from a performance analysis standpoint, as a significant number of unexpected messages may indicate an opportunity for increasing message passing performance.

The rest of this paper is organized as follows. The next section describes the motivations for this work in more detail. Section 3 discusses how this work relates to other published research. Following that, Section 4 discusses our approach for the measurements taken and Section 5 describes the test system and benchmarks. Measurements from the test system are presented in Section 6 followed by conclusions in Section 7 and future work in Section 8.

2. Motivation

Relative to host processors, network interface processors are slow, but their proximity to the network makes them adequate to accelerate some portion of the protocol stack in a cluster computing environment. The advantages of this form of offloading have been well established. In previous work, we have presented arguments for pushing more protocol information onto the network interface in order to better support upper-level transport protocols, and we have presented an architecture that provides the basic building blocks for constructing protocols for many different higher-level programming interfaces[5].

However, there are several fundamental issues that can impact the amount of protocol processing that is appropriate for offload. Most of these issues are related to the amount of resources that are needed from the network interface and how those resources are managed. For example, the Quadrics Tports interface and the Myricom MX interface[19] are all designed to support some sort of tagged message selection processing by the network interface. These interfaces have chosen to enable offloading of a portion of the MPI matching semantics because the extra processing resources required for tag matching on the network interface can provide a significant performance increase compared to performing tag matching using the host processor — provided there are sufficient NIC processing

cycles available. The Portals interface that we have proposed allows for offloading message selection capability, but also provides a much richer set of semantics that require processing as well as memory resources.

Currently, the amount of NIC resources that can be used for protocol processing are scarce relative to resources on the host. We believe that as networks increase in performance and the bottleneck between the host processor and the network processor is eliminated (or significantly reduced), a richer set of network processing semantics will be required to deliver raw network performance to applications. However, this ability to deliver performance will depend on having the appropriate allocation and management strategies for network resources.

For example, the Portals interface supports an implementation of MPI where the posted receive queue and unexpected queue exist in NIC memory. When a message comes into the NIC, the posted receive queue is traversed and the message is deposited directly into user memory. If the receive was pre-posted, the message is delivered directly into the appropriate user-supplied buffer. If the message was unexpected, it is delivered into a user-level buffer provided by the MPI library. We believe this approach can offer a significant performance improvement compared to MPI implementations where these queues are managed on the host by the MPI user-level library. However, the limited resource environment of most network interfaces prohibits us from traversing an extremely long posted receive queue, either because this prevents the NIC processor from other duties or because NIC memory is not large enough to hold an arbitrarily long queue. In this case, the appropriate strategy for managing these NIC resources is largely dependent on the behavior of the MPI application. This paper analyzes application benchmarks in an attempt to provide much needed data to answer questions about how NIC resources should be managed and how much processing capability might be needed on the NIC.

3. Related Work

There is a significant amount of work in the area of parallel application performance analysis. However, we know of no work that collects, analyzes, or uses MPI unexpected messages or MPI queue information as a basis to characterize performance and scalability. Most performance analysis tools for MPI use the MPI profiling interface to gather message tracing and timing information. Since unexpected messages are not exposed in the MPI programming interface, this information is not available to the profiling layer. Many of the issues with unexpected messages that we are analyzing in this paper have motivated work on a portable interface for exposing low-level MPI implementation details, such as unexpected messages, to application develop-

ers and performance tool developers. This interface, called PERUSE [11], is currently being explored by a number of organizations in the MPI research community. This work emphasizes the need to be able to capture low-level MPI performance information to assist in characterizing application message passing requirements. We have also tried to motivate this need with respect to unexpected messages in previous work [4, 6].

In addition to performance analysis, there is also a significant amount of work that characterizes the message passing behavior of applications and application benchmarks in an attempt to understand or predict how well they will scale. Example of this type of analysis can be found in [30] and [31]. As with performance tools, this analysis does not consider the impact or effect of unexpected messages or queue lengths, largely because this information is not easily attainable.

A variety of research efforts have studied the use of NIC processors to accelerate protocol processing[24, 25, 26] or have added hardware to support protocol processing[9, 18, 22, 23]. In general, these efforts have demonstrated improvements in achievable latency and bandwidth. However, with particularly slow embedded processors, it is already clear that insufficient processing power is available to fully support the network bandwidth[25]. Other protocol processing efforts outside of cluster computing have used promising new network processors such as the Intel IXP1200 and IXP2800[16, 12, 27] that have significantly more processing power, but are still resource constrained.

Some efforts have studied protocols that offload a portion of the MPI matching semantics. For example, in [17] portions of a Portals[5] stack were offloaded. Similarly, the Quadrics network[20] offloads the MPI matching stack onto the network interface, and others have explored this approach for Myrinet [29]. A new software stack from Myricom for Myrinet, called MX [19], appears to do this as well. A continuing challenge for such implementations is the relatively slow speed of the NIC processor. When the posted receive queue is short, such as with a ping-pong benchmark, dramatic latency reductions are seen; however, longer posted receive queues could lead to significant performance degradation.

Other efforts have considered absorbing other portions of the MPI stack. A particularly common option is to offload a portion of the MPI collective operations[15, 7, 28, 8]. Even with relative simple collective operations, limited CPU performance can be constraining.

The logical extension of previous work will be embedded network processors that absorb greater portions of MPI. However, none of the efforts thus far have presented studies with respect to the requirements of real applications for such implementations. For these efforts to be successful outside of the research laboratory, it will be necessary for

them to address real application usage scenarios. This paper provides some of the data that will be needed to properly design such MPI offload engines.

4. Approach

The MPICH [14] implementation has an abstract device interface (ADI) [13] that provides a network transport layer with the functions necessary to implement MPI semantics. In particular, the posted receive queue and unexpected message queue are linked lists that are managed within the ADI code. These linked lists are not usually manipulated by the underlying transport layer directly. The ADI abstracts the implementation of these queues away from the transport layer.

For example, the ADI provides a function call, `MPID_Msg_arrived()`, for the transport layer to use to signify the arrival of a new message. This function traverses the posted receive queue to see if there is already a matching receive posted. If there is, it removes the entry from the queue and proceeds. If not, it enqueues information about the new message in the unexpected queue. In order to keep track of the average number of times the posted receive queue is searched, we increment a counter each time `MPID_Msg_arrived()` is called. Inside this function, we increment another counter each time a queue entry is inspected.

The unexpected queue must be searched each time an MPI receive is posted. The MPICH ADI function, `MPID_Search_unexpected_queue_and_post()`, searches through the unexpected queue looking for a matching message. If no match is found, the receive is added to the posted receive queue. If a match is found, the unexpected message is dequeued and the receive is processed. This function actually calls another ADI function that searches the unexpected queue. We simply increment a counter each time this function is called, and increment another counter each time an unexpected queue entry is inspected.

We also used this function to keep track of the number of unexpected and expected messages. We traced the ADI code to find all of the places where this function is called, and incremented a counter for each type of message. In order to have more detail about short versus long messages, we traced the code further down into the device-specific transport layer (`ch_gm`) and inserted counters there.

We also instrumented the queue management utility functions in the MPICH ADI to keep track of maximum queue length. Each time an entry is enqueued, we increment a length counter associated with the queue. Likewise, this counter is decremented each time an entry is dequeued. Each time a new entry is enqueued to the posted receive or unexpected message queue, we inspect the length counter

to maintain its maximum value.

In order to allow applications to access these counters and maximum values, they were implemented as global variables. This approach allows them to be initialized without an explicit function call and allows them to be exported to the application easily. The MPICH ADI is not multi-threaded, so the global values are only manipulated by a single thread of execution.

We used the MPI profiling interface to collect the data and write it out to a file. We defined our own `MPI_Finalize()` routine that records the values and then gathers them to rank 0, which simply opens a text file and writes them out for each rank. This way, we do not need to modify applications at all. We simply re-link the code with the profiling code and the instrumented MPI library.

The overhead of instrumenting MPICH this way is negligible. The additional computation and logic needed for this instrumentation is insignificant, especially for unexpected messages, which are already in the low performance path. For a posted message, the computation and logic operations are performed after the message has been received, so the additional computation does not impact the transfer of the data.

In order to reduce variability, we ran each test four times and report the average over all of the runs. Each run was also made on the same set of compute nodes for each of the different processor counts.

5. Platform and Benchmarks

All of our tests were run on the Vplant machine at Sandia National Laboratories. Vplant is a large Linux cluster with approximately 320 compute nodes composed of Intel Pentium-3 and Pentium-4 processors. All of our experiments were run on nodes containing dual Pentium-4 Xeon processors running at 2.0 GHz. Each node has 1 GB of main memory and a Myrinet-2000 [3] network interface. The nodes are connected in a Clos topology. Vplant was running a Linux 2.4.18 kernel, GM version 1.6.4, and MPICH/GM version 1.2.4..8. All of our runs used only one process per node.

For this initial analysis, we chose the NAS Parallel Benchmarks (NPB) version 2.4 [10] class B. These benchmarks are a collection of MPI applications that are distilled from real computational fluid dynamics applications. Except for EP, they all exhibit particular message passing and computation patterns that stress different parts of the system. We omitted the EP benchmark from our study, since it does virtually no message passing. These benchmarks have been well-studied, but we have seen no data that characterizes their performance or behavior with respect to unexpected messages or MPI queues.

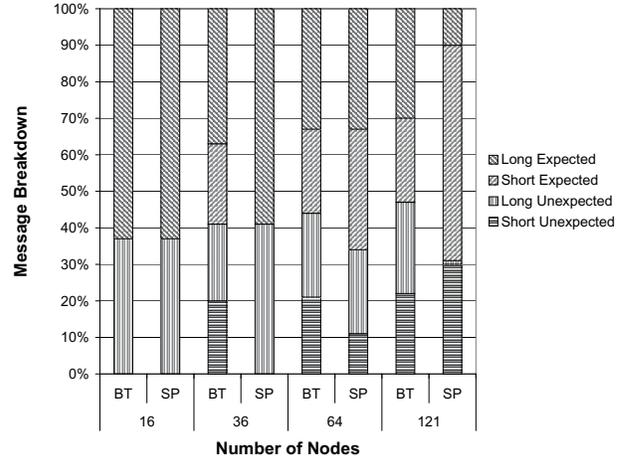


Figure 1. Message Breakdown for BT and SP

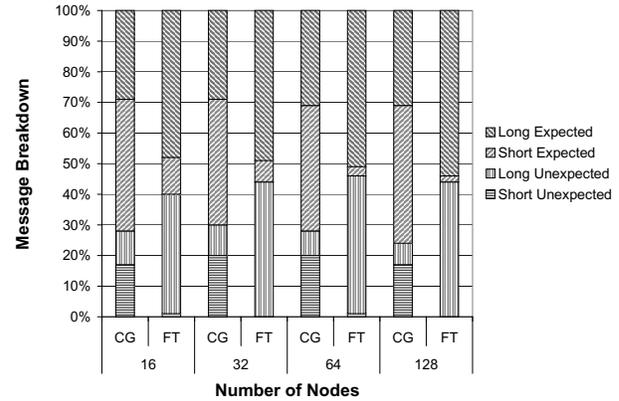


Figure 2. Message Breakdown for CG and FT

6. Results

6.1. Unexpected Messages

Unexpected messages can cause a significant amount of performance degradation for MPI. They are considered to be the “slow path”. However, Figures 1, 2, and 3 clearly indicate that a significant portion of the messages received by the NPB benchmarks are unexpected. Of all of the benchmarks, only MG has a relatively small number of unexpected messages.

The trends in Figures 1, 2, and 3 do not show any particular increase in the percentage of unexpected messages as the number of nodes is scaled. The ratio between short and long messages, however, does increase since the benchmarks use a fixed problem size. In the end, the overriding message is

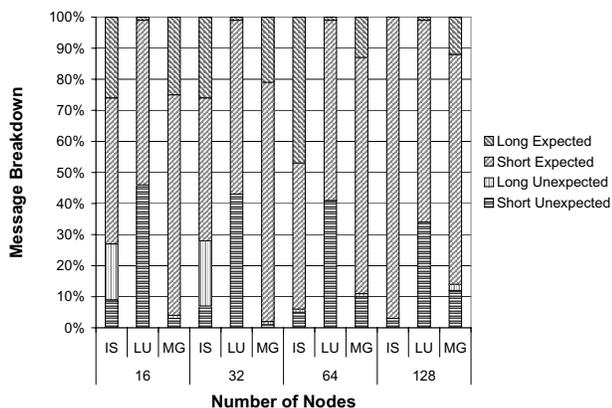


Figure 3. Message Breakdown for IS, LU and MG

that NICs that hope to offload a significant portion of MPI must deal with unexpected messages on a regular basis.

6.2. Queue Lengths

One of the greatest limitations of most modern NIC hardware is the extremely limited amount of memory on the card. As such, the maximum length of the posted receive queue and the unexpected message queue have significant implications for the feasibility of message offload. Figure 4 indicates that, for small numbers of processors, the maximum length of the posted receive queue is well within the limits of the memory that a modern NIC would support; however, the number of posted receives appears to grow linearly with the number of processors for both the IS and FT benchmarks. This has the potential to be a major limiting factor in the scalability of the offload of the MPI matching semantics. Although several of the benchmarks show flat scaling of the number of posted receives and a very short typical posted receive queue, an offload implementation must work and perform reasonably for all codes.

Similarly, Figure 5 indicates that FT, IS, and LU benchmarks all have a large maximum length for the unexpected message queue. As with the posted receive queue, the maximum length for this queue grows with the number of processors. Current implementations of unexpected messages typically use a threshold between “short”, eagerly sent messages that are buffered at the receiver if they are unexpected and “long”, rendezvous messages that are not buffered at the receiver. The current default threshold for GM on Myrinet, for example, is 16 KB (minus a small amount of header). At 10,000 nodes with a 16 KB threshold, a linear increase

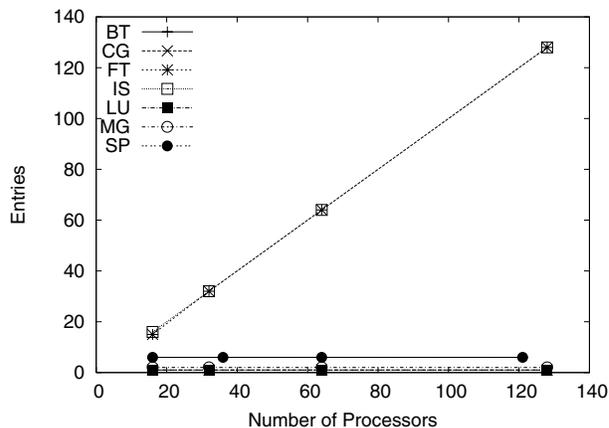


Figure 4. Maximum Length of the Posted Receive Queue

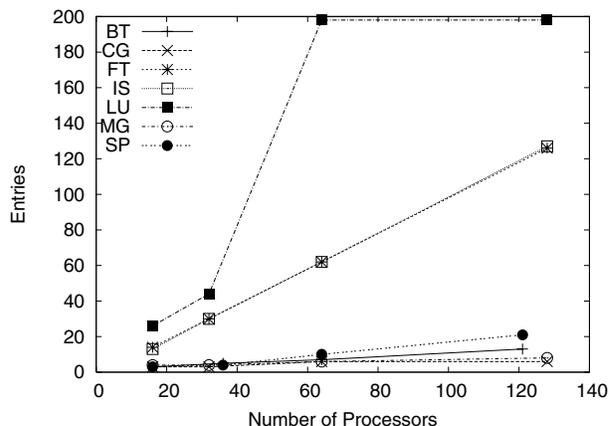


Figure 5. Maximum Length of the Unexpected Message Queue

in the maximum size of the unexpected queue would imply that 160 MB of memory was needed for buffering. In the worst case, a NIC that offloads unexpected message handling would need a mechanism to buffer this much data somewhere.

The current trends in networking are a second aspect to consider with respect to unexpected message offloading. At the high end, network bandwidth is improving faster than network latency. Since the optimal threshold increases with the bandwidth delay product, this implies that the threshold between short and long messages will increase. This will increase the amount of buffering an unexpected message offload will need to do to obtain maximum performance.

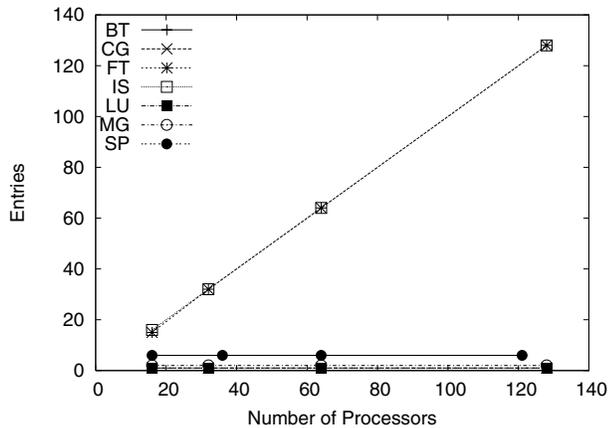


Figure 6. Maximum Search Length of the Posted Receive Queue

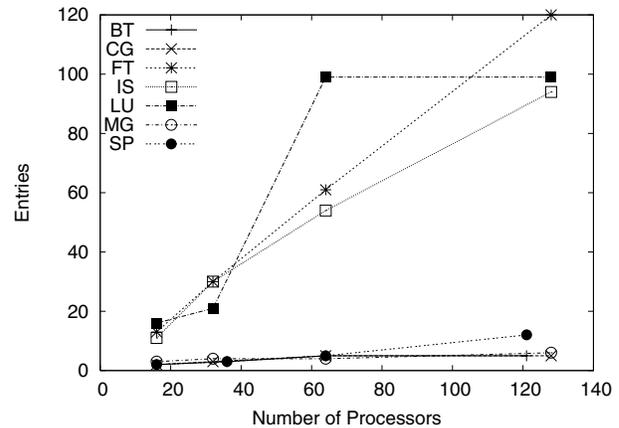


Figure 7. Maximum Search Length of the Unexpected Message Queue

6.3. Search Length

The search length of a queue is the number of queue entries that are traversed in a given search. While long queues have implications for the amount of memory required for a NIC offload implementation, the portion of those queues that are searched has a significant impact on the processing power needed on the NIC. It also affects the real latency seen by applications.

Maximum Search Length. The maximum search length is the largest number of queue entries that are traversed in any of the queue searches. Although the maximum search length can vary between application runs, the maximum search length over several application runs places an upper bound on the number of queue entries that are likely to be traversed. This frees an MPI offload designer to optimize to this likely upper bound.

Unfortunately, Figure 6 indicates that the maximum search length of the posted receive queue is equivalent to the length of the posted receive queue. This would seemingly prohibit, for example, an offload of the MPI matching semantics that chose to buffer a small portion of the posted receive queue in NIC memory and then walk the remainder of the queue by accessing host memory from the NIC.

When comparing this to Figure 8, it is also clear that there is a large difference between the average case and the worst case. This will inevitably introduce variability into the execution time of a time step (the time between synchronization points). This type of variability is the key symptom of the “rogue OS effect”[21], which leads to significantly longer applications execution times. This type of variability will also prove to be one of the major limiting factors in scaling from 10,000 to 100,000 nodes.

Figure 7 indicates that applications are slightly less likely to traverse the entire unexpected message queue. They do, however, still traverse a significant portion of the unexpected message queue. A comparison between Figure 7 and Figure 9 indicates a dramatic difference in the maximum and average portions of the unexpected message queue that are traversed. Thus, the unexpected message queue can be an even larger contributor to the variability in application execution time than the posted receive queue.

Average Search Length. The average search length is the average number of queue entries that are inspected when the queue is searched. The average search length is important since it can significantly affect the average latency that a process sees. If the average search length is long and the average message is short, average search length can have a detrimental effect on bandwidth. Both long searches of the posted receive queue and long searches of the unexpected message queue can have detrimental effects on the real latency an application experiences. The former slows down message reception and the latter slows down message posting. Because NIC processors are typically an order of magnitude slower than host processors, this effect (relative to the latency of the physical network layer) is exaggerated when MPI offload implementations are performed.

Figure 8 indicates that the average number of entries inspected for IS and FT grow almost linearly with the number of processors. This is not surprising since the maximum queue length grows linearly in the number of processors and the percentage of unexpected messages remains constant. Unexpected messages are guaranteed to traverse the entire posted receive queue. This data explains results in [29], which indicate that the performance advantage of MPI offload decreases as the number of processors increases. In-

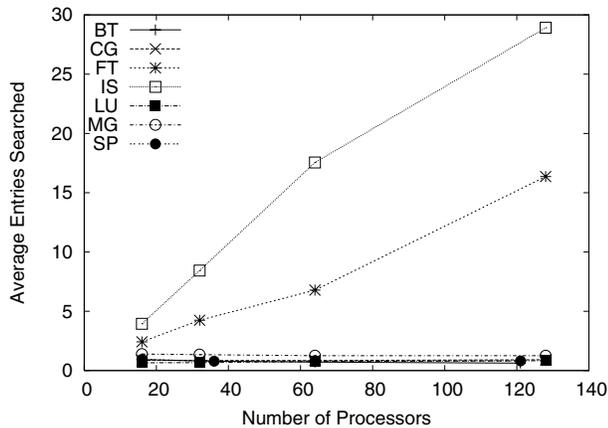


Figure 8. Average Search Length of the Posted Receive Queue

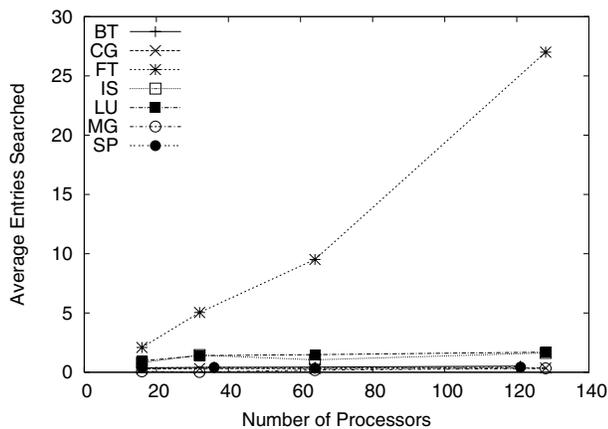


Figure 9. Average Search Length of the Unexpected Message Queue

deed, as the number of processors increases, the offload implementation loses to the processor implementation.

The average search length for the unexpected queue is relatively short for all applications except FT. For FT, the search length grows linearly with the number of processors. In general, this is good news for NIC designs that seek to offload more of the MPI layer; however, NIC offload engines must still be aware of the subset of codes that must search significant portions unexpected message queue.

7. Conclusions

The results presented clearly indicate that the usage of MPI resources varies dramatically across applications in the NPB suite. For example, at 128 processors, the average

number of messages traversed in the posted receive queue ranges from a low of approximately 1 to a high of 30. There is also significant variability in the parameters within one application — although the average number of posted receive queue elements traversed is 30, the max is over 100 (presumably, the minimum is 1). Most notable, however, is the linear growth in both the maximum search length and the average search length of these queues as the number of processes increases.

The implications of this data for NIC-based MPI offload are dramatic. For example, applications with long posted receive queues will require NICs to provide much more computing power than they currently have or will require a change in the matching structures typically used by MPI. Many consider such applications to be rare, but they represent a significant component of the NPB suite. Similarly, unexpected messages (and the time they consume when posting an `MPI_Recv`) are clearly a factor that must be dealt with for some applications. Finally, the results in this paper illustrate the need for better microbenchmarks. These microbenchmarks must provide clear insight into the improvements that network optimizations achieve while testing the network under a usage model that occurs in applications.

8. Future Work

There are three logical successors to this analysis. The first step is to extend this analysis to consider real applications. Obtaining, compiling, and running multi-language applications with over 500,000 lines of code is a daunting effort that is only justified in the light of this analysis. The second step is to design readily accessible benchmarks that test network parameters under more realistic workloads. For example, latency should be tested with items in the posted receive queue and bandwidth should be tested with unexpected messages. The final step is to study the relative performance (versus microbenchmarks) of modern cluster interconnects under real usage models.

References

- [1] R. Alverson. Red Storm. In *Invited Talk, Hot Interconnects 10*, August 2003.
- [2] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA, Dec. 1995.
- [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*, 15(1):29–36, Feb. 1995.
- [4] R. Brightwell. Ready-mode receive: An optimized receive function for MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 9th European PVM/MPI Users' Group Meeting*, September 2002.

- [5] R. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April 2002.
- [6] R. Brightwell and K. Underwood. Evaluation of an eager protocol optimization for MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 10th European PVM/MPI Users' Group Meeting*, September 2003.
- [7] D. Buntinas and D. K. Panda. NIC-based reduction in Myrinet clusters: Is it beneficial? In *Proceedings of the SAN-02 Workshop (in conjunction with HPCA)*, February 2002.
- [8] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-based barrier over Myrinet/GM. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.
- [9] Y. Coody, J. S. Ong, and M. J. Feeley. Using embedded network processors to implement global memory management in a workstation cluster. In *Proceedings of The Eighth IEEE International Symposium on High Performance Distributed Computing*, Redondo Beach, California, USA, Aug. 1999.
- [10] R. F. V. der Wijngaart. NAS parallel benchmarks version 2.4. Technical report.
- [11] R. Dimitrov, A. Skjellum, T. Jones, B. de Supinski, R. Brightwell, C. Janssen, and M. Nochumson. PERUSE: An MPI performance revealing extensions interface. Presented at the Sixth IBM System Scientific Computing User Group, August 2002.
- [12] M. Gries, C. Kulkarni, C. Sauer, and K. Keutzer. Exploring trade-offs in performance and programmability of processing element topologies for network processors. In *Workshop on Network Processors (in conjunction with HPCA)*, February 2003.
- [13] W. Gropp and E. Lusk. *MPICH ADI Implementation Reference Manual*. Mathematics and Computer Science Division, Argonne National Laboratory, October 1994.
- [14] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
- [15] R. Gupta, P. Balaji, D. K. Panda, and J. Nieplocha. Efficient collective operations using remote memory operations on VIA-based clusters. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003.
- [16] C. Isert and K. Schwan. ACDS: Adapting Computational Data Streams for High Performance. In *14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, April 2000.
- [17] A. B. Maccabe, W. Zhu, J. Otto, and R. Riesen. Experience in offloading protocol processing to a programmable NIC. In *IEEE International Conference on Cluster Computing*, September 2002.
- [18] T. Mummert, C. Kosak, P. Steenkiste, and A. Fisher. Fine grain parallel communication on general purpose LANs. In *Proceedings of 1996 International Conference on Supercomputing (ICS96)*, pages 341–349, Philadelphia, PA, USA, May 1996.
- [19] Myricom, Inc. Myrinet Express (MX): A high performance, low-level, message-passing interface for Myrinet. <http://www.myri.com/scs/MX/doc/mx.pdf>, July 2003.
- [20] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/February 2002.
- [21] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Identifying and eliminating the performance variability on the ASCI Q machine. In *Proceedings of the 2003 Conference on High Performance Networking and Computing*, November 2003.
- [22] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *International Conference on Computer Architecture*, pages 260–267, Chicago, Illinois, USA, Apr. 1994.
- [23] M.-C. Roşu, K. Schwan, and R. Fujimoto. Supporting parallel applications on clusters of workstations: The intelligent network interface approach. In *Proceeding of the 6th International Symposium on High Performance Distributed Computing (HPDC 97)*, 1997.
- [24] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *Proceedings of the 2001 Conference on Supercomputing*, Nov. 2001.
- [25] P. Shivam, P. Wyckoff, and D. Panda. Can user-level protocols take advantage of multi-CPU NICs? In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2002.
- [26] S. Sumimoto, H. Tezuka, A. Hori, H. Harada, T. Takahashi, and Y. Ishikawa. The design and evaluation of high performance communication using a Gigabit Ethernet. In *International Conference on Supercomputing*, pages 260–267, Rhodes, Greece, June 1999.
- [27] R. Thomas, B. Mark, T. Johnson, and J. Croall. High-speed legitimacy-based DDoS packet filtering with network processors: A case study and implementation on the Intel IXP1200. In *Workshop on Network Processors (in conjunction with HPCA)*, February 2003.
- [28] V. Tipparaju, J. Nieplocha, and D. K. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003.
- [29] B. Tourancheau and R. Westrelin. Support for MPI at the network interface level. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 8th European PVM/MPI Users' Group Meeting*, September 2001.
- [30] J. S. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *16th International Parallel and Distributed Processing Symposium (IPDPS'02)*, pages 27–29, April 2002.
- [31] F. Wong, R. Martin, R. Arpaci-Dusseau, and D. E. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *Proceedings of the SC99 Conference on High Performance Networking and Computing*, November 1999.