Analysis of Design Considerations for Optimizing Multi-Channel MPI over InfiniBand

Lei Chai, Sayantan Sur, Hyun-Wook Jin and D. K. Panda

Technical Report
OSU-CISRC-11/04-TR61

Analysis of Design Considerations for Optimizing Multi-Channel MPI over InfiniBand *

Lei Chai Sayantan Sur Hyun-Wook Jin Dhabaleswar K. Panda

Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{chail,surs,jinhy,panda}@cse.ohio-state.edu

Abstract

Modern day MPI implementations provide several communication channels for optimizing performance. To obtain the best performance for the most demanding contemporary applications, it becomes critical to manage these communication channels efficiently. Various issues related to overhead for message discovery and thresholds for choosing different channels need to be considered for designing the MPI layer. It is not a trivial task to choose these parameters since application characteristics and demands from the MPI layer vary widely. In this paper we try to address these issues. We propose several different schemes such as static priority and dynamic priority to efficiently implement channel polling. Our results indicate that we can reduce latency by unto 37% and message discovery time upto 45%. Further, we explore several different methodologies to choose appropriate thresholds for different channels.

1 Introduction

Cluster based computing systems are becoming popular for a wide range of scientific applications owing to their cost-effectiveness. These systems are typically built from Symmetric Multi-Processor (SMP) nodes connected with high speed Local Area Networks (LANs) or System Area Networks (SANs). A majority of these scientific applications are written on top of the Message Passing Interface (MPI) [11]. Even though the high performance networks have evolved and have very low latencies, below 5μs, intra-node communication still remains order of magnitudes faster than the network. In order to fully exploit this, MPI applications usually run a set of processes on the same physical node. Therefore, MPI applications usually perform both intra and inter node communications on SMP clusters.

To optimize intra and inter node communication, many MPI implementations such as MVAPICH [2] provide multiple communication channels. These channels may either be for network communication or for intra-node communication. Efficient polling of these communication channels for discovering new messages is often considered to be one of the key design issues in implementing MPI over any network layer. In order to efficiently design an implement these channel interfaces, we need a centralized policy. Since communication patterns as well as need of overlap of communication and computation vary widely over different applications, it becomes hard to design a general purpose policy. We need to carefully consider the overheads and benefits offered by each channel.

In this paper, we try to bring forward important factors that should be considered to efficiently utilize several MPI channels through in-depth measurements and analysis. Our study starts with polling schemes among multiple channels. By our experiments we observe that the intra-node latency can be improved by 37% using our static polling scheme. Further, the adaptive polling scheme can reduce the new message discovery overhead by 45%. Then, we explore methodologies to decide the threshold between multiple channels.

The rest of the paper is organized as follows. Section 2 gives a brief description of the various communication channels used in MVAPICH. Section 3 describes the various schemes we devise for polling different chan-

*This research is supported in part by Department of Energy's grant #DE-FC02-01ER25506, National Science Foundation's grants #CCR-0204429 and #CCR-0311542 and a grant from Mellanox, Inc.

nels. Section 4 describes the methodology followed for choosing efficient thresholds for different channels. In section 5 we detail the previous work done in this direction. Then finally we conclude in section 6 and state our future work directions.

2 Background

MVAPICH [2] is an open-source implementation of MPI over InfiniBand [1]. MVAPICH is based on the Abstract Device Layer of MPICH [6]. In this section we give a brief background of the various communication channels used in MVAPICH.

2.1 Network Channel

InfiniBand Architecture offers both send/receive and RDMA semantics. MVAPICH uses both of them for efficient communication.

2.1.1 RDMA Channel

The MPI eager protocol is mapped to the RDMA channel design. Small and control messages are *eagerly* placed into the receiving MPI's internal buffers. The RDMA Write InfiniBand primitive is used for this channel. Since there is no software involvement at the receiver side, the receiver can only discover new messages based on polling memory contents. The RDMA buffers are maintained in a circular queue, and a pointer is kept to the buffer in which the new message is expected. The sender and the receiver maintain persistent-association of RDMA buffers [9]. After the *head* and the *tail* pointers of the sender point to the same location, the sender cannot send any more RDMA messages. It has to wait for a message from the receiver (with a piggy-backed ACK) or an explicit ACK message.

2.1.2 Send/Receive Channel

In this channel the messages are sent over InfiniBand send/receive primitives. The receiver pre-posts a number of buffers at the initialization time. For transferring a message, the sender first copies the application buffer into the registered (pinned) buffer and then issues an InfiniBand send operation. Upon arrival of the message at the receivers end, a completion entry is generated for the receive (which was pre-posted earlier). The receiver has to poll the completion queue to detect the new arrival. It is to be noted that the same completion queue can be shared among all connections.

2.2 Shared Memory Channel

This channel involves each MPI process on the same node attaching itself to a shared memory region. This shared memory region can then be used amongst the local processes to exchange messages and other control information. This shared memory based design has been used in MPICH-GM[12] and other MPI implementations such as MVAPICH. The sending process copies the message along with other information required for message matching to the shared memory area. The receiving process can then match the tags of the posted receives and accordingly copy over the correct message to its own buffer. We note that this approach involves minimal setup overhead for every message exchange. However, there are at least two copies involved in the message exchange. This approach might tie down the CPU with memory copy time. This method is feasible for shorter messages, where the send, receive and shared buffer are probably present in the cache. But as the size of the message grows, the performance deteriorates. Such vigorous copy-in and copy-out also destroys the cache contents for the end MPI application.

2.3 Kernel Module Channel

This channel has been designed and implemented in a previous work [7]. This approach takes help from the operating system kernel to copy messages directly from one user process to another. The sender or the receiver process posts the message request descriptor in a message queue indicating its virtual address, tags etc. This memory is then mapped into the kernel address space. When the other process (either sender or receiver) arrives at the message exchange point, the message descriptors are matched at the kernel level and the kernel performs a direct copy from the sender buffer to the receiver application buffer. Thus this approach involves only one copy. This approach reduces the number of copies. However, there are other overheads, which include time to trap into the kernel and locking of other kernel data structures. In addition, one CPU is still required to perform the copy operation. We note that these overheads are relatively less as compared to typical message transfer latencies of medium and large messages.

3 Channel Polling

3.1 Channel Polling Overheads

Each channel has different polling overhead and mechanism to deliver messages to the MPI layer. An efficient polling scheme should minimize the overhead associated with discovery of new messages.

3.1.1 Network Channel Overhead

MVAPICH [2] uses two methods to communicate over the InfiniBand network. For short and control messages, it uses the Eager protocol. This protocol uses RDMA. Since there is no software involvement at the receiver side, the only way to check for incoming messages is by polling memory locations. The polling overhead involved in polling memory locations is negligible. Polling this RDMA channel is equivalent to polling n bytes, where n is the number of processes on different physical nodes. Another network communication channel used is the InfiniBand sends and receives. These generate message completions at the receiver side. The receiver polls the completion queue and can look at new incoming messages. The overhead associated with polling the completion queue is constant and does not vary across the number of processes. However, the overhead to poll an empty completion queue, it takes around 0.3μs.

3.1.2 Shared Memory Channel Overhead

The shared memory channel uses a FIFO queue for each shared memory connection. In addition, the channel maintains a counter which indicates whether a new message is available for this connection. So, polling the shared memory channel is equivalent to polling n bytes, where n is the number of processes on the same physical node.

We note that in the industry, mostly 2 to 16 way SMPs are used. Hence, the polling overhead for this shared memory channel is not significant.

3.1.3 Kernel Module Channel Overhead

The kernel module channel [7] copies messages directly from the sender buffer to the receiver buffer. This approach can avoid multiple copies to and from a shared memory region. Also, it avoids the pinning cost at one side. To reduce the pinning cost at one side, the copy is performed in either the send or the receive. Polling of the kernel module channel is expensive as it requires a context-switch to the kernel-space. The kernel module can provide two mechanisms to poll for incoming messages:

- Busy polling of the kernel module in the blocking MPI send, receive or wait functions. In this case, we poll the kernel module channel explicitly when a message is expected to arrive from that channel.
- The kernel module can provide some signalling bit to indicate the arrival of new messages to the MPI layer. Although it can reduce the number of context switches, still we need to trap into the kernel to match MPI headers. In the worst case, if some unexpected message arrives in the kernel, the MPI layer still needs to poll that message because the signal bit does not have information about the MPI header.

In order to avoid multiple context-switches and needlessly introducing overhead polling the kernel module, we place the polling of the kernel module outside the main MPI progress engine. So, if any messages are not expected from the kernel module channel, then that channel is not polled at all. All unexpected messages arriving through the kernel module channel are kept queued by the kernel module. The messages are copied when the receiver posts the matching receive.

3.2 Channel Polling Schemes

As described in section 3.1 there are different costs associated with polling of each channel. In this section we design different polling schemes to reduce the overhead associated with polling network and shared memory channels and enable faster message discovery. As we have described in section 3.1, the polling of the kernel module is placed outside the main progress engine. Since, the kernel module is not polled if no messages are not expected from it, we exclude the kernel module from the study of these polling schemes. Broadly speaking, the channel polling schemes can be classified as static and dynamic depending on whether the scheme itself changes over time or not.

3.2.1 Static Channel Polling Schemes

Static polling schemes are decided at the start of the MPI application. This scheme can assign different priorities (or weights) to different channels. The intuitive idea behind such schemes is that some channels may be used more or faster than the others. Many such schemes are possible based on various considerations. Some of them can be listed as:

- **Latency Based Priority Scheme:** In this scheme the channel with the lowest latency is given the highest priority. Hence, in this scheme messages from the lowest latency channel are discovered before messages from other channels.

- **Overhead Based Priority Scheme:** In this scheme the channels which have the least overhead of polling are given the highest priority. This scheme tries to minimize the overall overhead associated with the MPI progress engine.

In our paper, we consider both the schemes. In section 4.2, we notice that the shared memory channel has the lowest latency. Also, in section 3.1 we notice that the overhead of polling this channel is the least. In the new hybrid polling scheme, we give most priority to the shared memory channel. In addition, since the overhead of polling this channel is the least, we always poll this channel with at least as much weight as the other channels.

3.2.2 Dynamic Channel Polling Schemes

Dynamic polling schemes can change over the course of the execution of the MPI application. There are various factors to be considered while designing such a dynamic scheme:

- **Update rate:** This factor determines how often the priority ratios are calculated. A very high update rate would imply increased overheads for short messages, whereas a low update rate would miss smaller bursts of messages from other channels.

- **Message history:** This factor determines the number of messages recorded for computing the new priority ratio. The more messages are considered, the slower the priority ratio will change. This might miss smaller bursts of messages, whereas when lower number of messages are considered a lot of fluctuation may occur even with small bursts of messages from a channel.

In section 3.3.2, we design a MPI micro benchmark to evaluate the average time taken to discover a message. We then vary the update rate and the message history to observe the variation of the discovery time.

3.3 Performance evaluation of Polling Schemes

We conducted experiments on an 8 node cluster with the following configuration:

Super Micro SUPER X5DL8-GG nodes with dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, 2 GB memory, PCI-X 64-bit 133 MHz bus. The Linux kernel version used was 2.4.22smp from kernel.org. The Mellanox InfiniBand stack [10] was used. The version of VAPI was 3.2 and firmware version 3.2.

3.3.1 Evaluation of Static Polling Scheme

One crucial factor to determine for the static polling scheme is *"how much priority should be given to the shared memory channel?"* Obviously, if we give more priority to shared memory channel, then the shared memory latency will reduce. But at the same time the latency of messages coming over the network will also increase. To find out the optimal priority ratio, we conducted the standard ping-pong latency test with different priority ratios. Figure 2 shows variation of ping-pong latency with various priority ratios.

We can observe from these figures that if we give the shared memory channel a priority ratio of $50:1$, then we can get a reasonable balance between improvement of shared memory latency and not hurt the network latency. Our experiments indicate that we can achieve upto 37% improvement in intra-node latency using the static polling scheme scheme with $1000:1$.

3.3.2 Evaluation of Dynamic Polling Scheme

In order to evaluate the dynamic polling schemes we need to devise a new MPI micro benchmark that appropriately captures the message discovery time at the MPI layer. There are three processes in the benchmark. Two processes are on the same node, whereas one process is on a separate node. This process sends messages over the network, whereas the process on the same node sends messages exclusively through shared memory channel. On the receipt of each message the *"root"* process replies with an ACK. The process sending the *"burst"* number of messages to the root is alternately selected between the network peer and the shared memory peer. This test captures the message discovery time by the root process before it can send an ACK to the peer process. As the rate of updation of the priority ratios is changed, there are tradeoffs between reducing the message discovery time of the shared memory messages and network messages. Figure 1 illustrates this micro benchmark where we are trying to measure time T.

Figure 3 shows the performance results of this micro benchmark with message burst sizes of 100 and 200 with varying update rates.

We observe that with the increase of update rate, the message discovery time actually decreases. How-

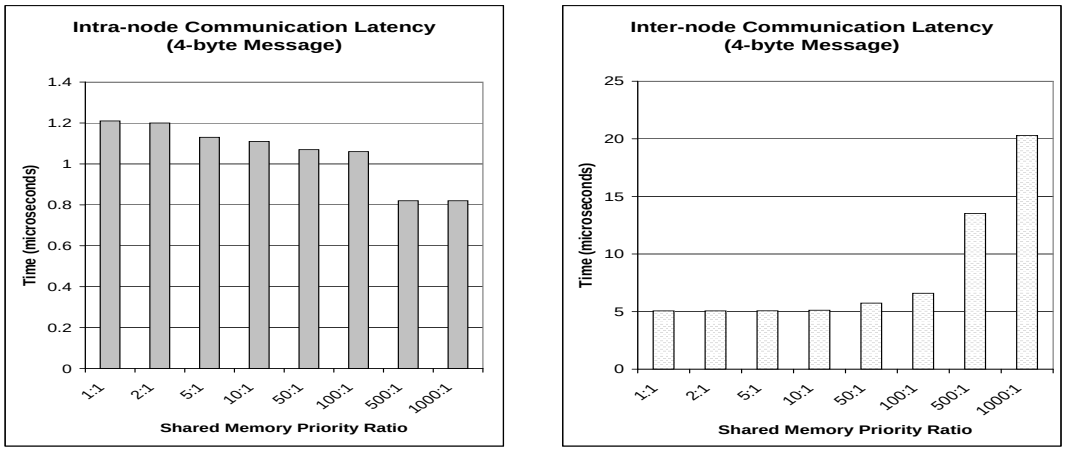


Figure 2. Latency of Static Polling Scheme

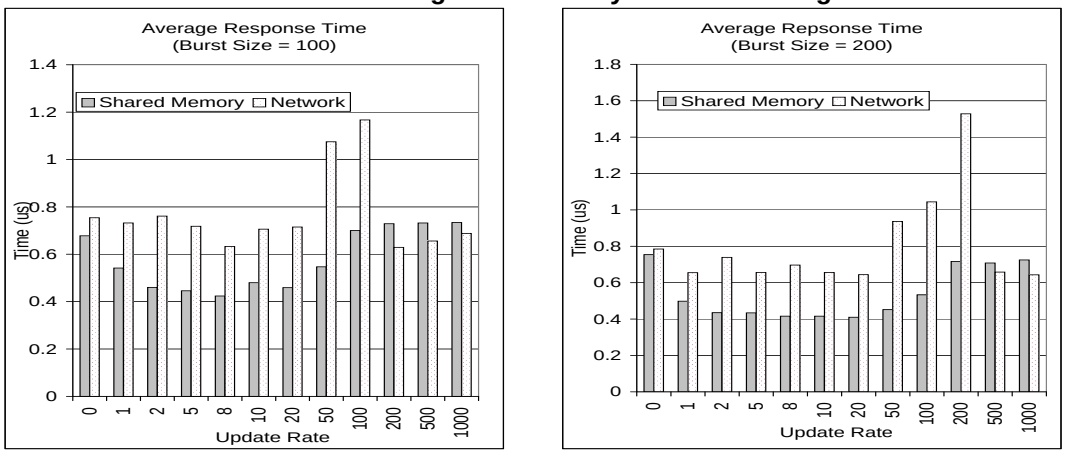


Figure 3. Message Discovery Time of Dynamic Polling Scheme

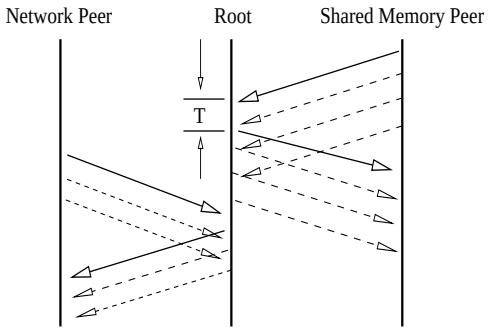


Figure 1. Message Discovery Micro benchmark

ever, when the update rate becomes higher, the overhead causes the discovery time to rise. We also observe that when the burst size is equal to the update rate, the discovery time increases significantly due to continuous wrong predictions. We conclude from these figures that an update rate of 8 or 10 is enough not to introduce too much overhead and also sustain fairly small burst of messages. Our experiments indicate that we can achieve upto 45% improvement rate of message discovery time with burst size of 200.

4 Channel Thresholds

As described in section 2 each communication channel has different performance characteristics. Some channels have low latency and some channels have high bandwidth. In addition, some channels do not require the involvement of the host CPU. In this section, we take a look at selecting appropriate thresholds for efficient message passing.

4.1 Threshold Decision Methodology

In this section we discuss different approaches to choosing appropriate thresholds for choosing the best possible channel for communication.

4.1.1 Microbenchmark Based Decision

In general, it is very difficult to decide the threshold of communication channel for all applications. However, it is widely accepted that such decisions are based on latency and bandwidth measurements. Therefore we can look at MPI microbenchmarks to see the basic performance of each channel. We evaluate based on the standard ping-pong latency and bandwidth tests [8].

4.1.2 CPU Utilization Based Decision

In this approach we calculate the CPU utilization for message passing. Although some channels might provide higher latency but they may effectively overlap computation and communication. This might be beneficial for applications that are efficiently programmed to overlap them. However, this is very application specific. For applications which mostly use blocking operations, simply providing the lowest latency channel would be enough.

4.2 Evaluation of Threshold

In this section we run the above mentioned decision approaches on the cluster described in section 3.3. We use the standard ping-pong latency and bandwidth to evaluate the threshold points for the three channels.

Figure 5 shows the experimental results of the latency and bandwidth tests.

We find that for messages less than 4KB size, it is beneficial to use the shared memory based approach. This is because the shared memory channel avoids context-switch cost and maximizes cache effect. For messages greater than that, it is useful to have the kernel module based approach. This is mainly because the number of copies has been reduced to one.

Figure 4 shows the CPU utilization taken by each channel. We note that the CPU utilized by the network is close to 1% whereas the other copy based schemes it is higher since the CPU is tied down with the copy operation. Therefore, if the MPI application tries to maximize the overlap between computation and communication, the network channel can be better than others even for intra-node communication. We are analyzing the high CPU utilization of the kernel module. We will include that analysis in our technical report and final version.

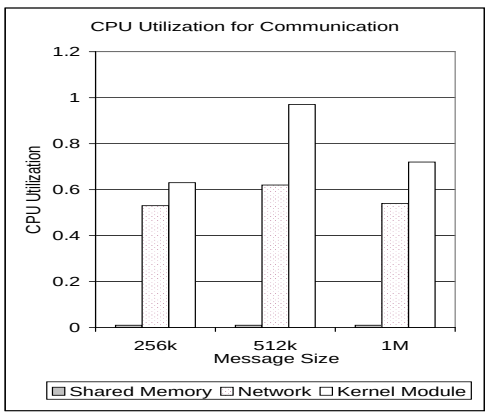
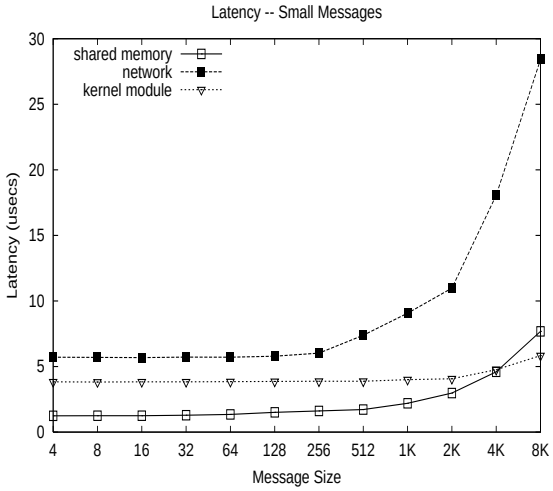
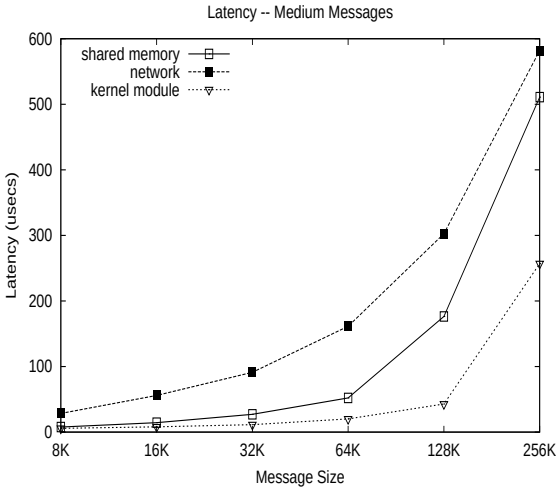
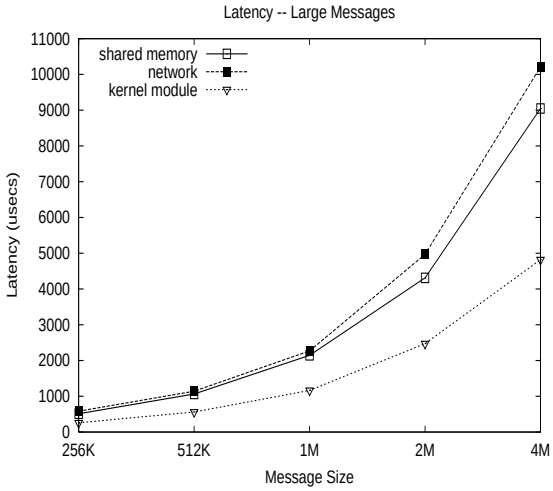
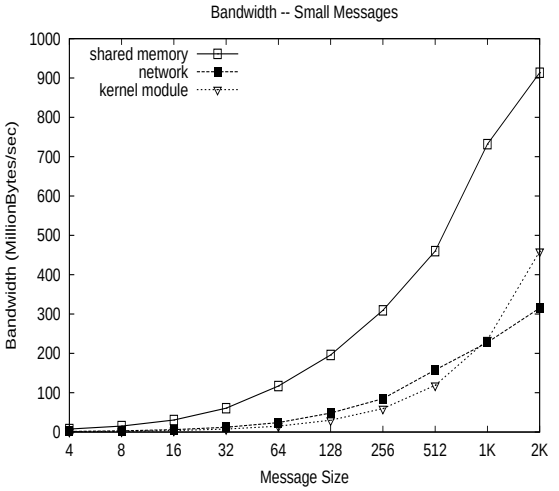


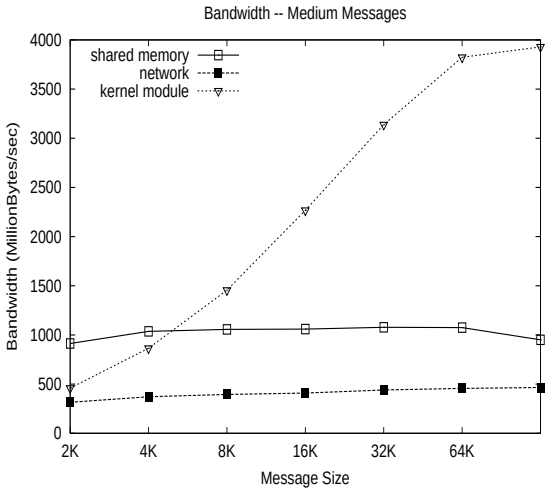
Figure 4. CPU Utilization











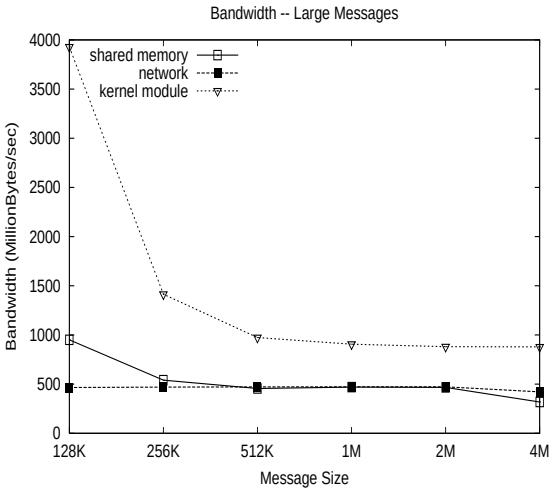


Figure 5. Threshold Selection Mechanism

5 Related Work

Several researchers have proposed different designs for efficiently implementing intra-node communication in clusters. In this section, we describe in brief their contributions. Geoffray, Tourancheau, Prylli et al [13][5][4] suggest a BIP-SMP multi-protocol layer for intra-node and inter-node communication. Takahashi et al [15][14] provide a device driver called PM/SHMEM that supports a direct memory access between processes. MPICH-GM[12] provides a User level shared memory approach for MPI. It also provided a kernel level approach but the support is withdrawn currently to the best of our knowledge. Although many researchers have suggested different channels for intra-node communication, they have not considered the design factors to optimize polling among different channels and communication/computation overlap.

In addition it is to be noted that Brightwell, et al have shown the impact of the usage of MPI queue on latency and applications[16] [3].

6 Conclusion

In this paper we proposed several different schemes for polling the communication channels in MVAPICH. We evaluated several approaches, both static and dynamic approaches to polling. In addition, we evaluated thresholds for each channel both based on raw MPI latencies and bandwidths and also CPU utilization.

We note that the static polling scheme can reduce intra-node latencies by 37%. By the adaptive polling scheme we can reduce the message discovery overhead by 37% for a message burst of 100. For a message burst of 200, we could reduce the discovery overhead to 45%. Also, we evaluated the thresholds for all the communication channels.

We would like to continue work in this direction. We would like to evaluate the impact of these schemes on end MPI applications.

References

[1] InfiniBand Trade Association. http://www.infinibandta.com.

[2] MPI over InfiniBand Project. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/.

[3] R. Brightwell, K. Underwood, and R. Riesen. An Initial Analysis of the Impact of Overlap and Independent Progress for MPI. In *Euro PVM/MPI*, 2004.

[4] P. Geoffray, C. Pham, and B. Tourancheau. A Software Suite for High-Performance Communications on Clusters of SMPs. *Cluster Computing*, 5(4):353–363, October 2002.

[5] Patrick Geoffray, Loic Prylli, and Bernard Tourancheau. BIP-SMP: High Performance Message Passing over a Cluster of Commodity SMPs. In *SuperComputing (SC)*, 1999.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[7] H. W. Jin, S. Sur, L. Chai, and D. K. Panda. Design and Performance Evaluation of LiMIC (Linux Kernel Module for MPI Intra-node Communication) on InfiniBand Cluster. Technical Report OSU-CISRC-10/04-TR58, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210, October 2004.

[8] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *SuperComputing(SC)*, 2003.

[9] J. Liu, J. Wu, , and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. volume In Press, 2004.

[10] Mellanox Technologies. Mellanox VAPI Interface, July 2002.

[11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[12] Myricom Inc. Portable MPI Model Implementation over GM, March 2004.

[13] Loic Prylli and Bernard Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. *Lecture Notes in Computer Science*, 1388, April 1998.

[14] Toshiyuki Takahashi, Shinji Sumimoto nd Atsushi Hori, Hiroshi Harada, and Yutaka Ishikawa. PM2: High Performance Communication Middleware for Heterogeneous Network Environments. In *SuperComputing (SC)*, 2000.

[15] Toshiyuki Takahashi, Francis O'carroll, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Hiroshi

Harada, Yutaka Ishikawa, and Peter H. Beckman. Implementation and Evaluation of MPI on an SMP Cluster. *Lecture Notes in Computer Science*, 1586, April 1999.

[16] K. Underwood and R. Brightwell. The Impact of MPI Queue Usage on Message Latency. In *Internation Conference on Parallel Processing (ICPP)*, 2004.