A performance comparison of myrinet protocol stacks

Ron Brightwell[1], William Lawry[1], Mike Levenhagen[1], Arthur B. Maccabe[2] & Rolf Riesen[1]
[1]*Sandia National Laboratories*
[2]*Department of Computer Science*
University of New Mexico

Abstract

This paper describes a portable benchmark suite that assesses the ability of cluster networking hardware and software to overlap MPI communication and computation. The Communication Offload MPI-based Benchmark, or COMB, uses two different methods to characterize the ability of messages to make progress concurrently with computational processing on the host processor(s). COMB measures the relationship between overall MPI communication bandwidth and host CPU availability. In this paper, we describe the two different approaches used by the benchmark suite, and we present results three different Myrinet protocol stacks that are used to support the Portals 3.0 message passing interface. We demonstrate the utility of the suite by examining the results and comparing and contrasting the different protocol stacks.

1 Introduction

Recent advances in networking technology for cluster computing have led to significant improvements in achievable latency and bandwidth performance. Many of these improvements are based on an implementation strategy called Operating System Bypass, or OS-bypass, which attempts to increase network performance and reduce host CPU overhead by offloading communication operations to intelligent network interfaces. These interfaces, such as Myrinet [2], are capable of "user-level" networking, that is, moving data directly from an application's address space without any involvement of the operating system in the data transfer.

Unfortunately, the reduction in host CPU overhead, which has been shown to be

the most significant factor in effecting application performance [6], has not been realized in most implementations of MPI [7] for user-level networking technology. While most MPI microbenchmarks can measure latency, bandwidth, and host CPU overhead, they fail to accurately characterize the actual performance that applications can expect. Communication microbenchmarks typically focus on message passing performance relative to achieving peak performance of the network and do not characterize the performance impact of message passing relative to both the peak performance of the network and the peak performance available to the application.

We have designed and implemented a portable benchmark suite called COMB, the Communication Offload MPI-based Benchmark, that measures the ability of an MPI implementation to overlap computation and MPI communication. The ability to overlap is influenced by several system characteristics, such as the quality of the MPI implementation and the capabilities of the underlying network transport layer. For example, some message passing systems interrupt the host CPU to obtain resources from the operating system in order to receive packets from the network. This strategy is likely to adversely impact the utilization of the host CPU, but may allow for an increase in MPI bandwidth. We believe our benchmark suite can provide insight into the relationship between network performance and host CPU performance in order to better understand the actual performance delivered to applications.

2 Approach

Our main goal in developing this benchmark suite was to be able to measure overlap as accurately as possible while still being as portable as possible. We have chosen to develop COMB with the following characteristics:

- One process per node
- Two processes perform communication
- Either process may track bandwidth
- One process performs simulated computation
- Both processes perform message passing
- Primary variable is the simulated computation time

The COMB benchmark suite consists of two different methods of measuring the performance of a system, each with a different perspective on characterizing the ability to overlap computation and MPI communication. This multi-method approach captures performance data on a wider range of the systems and allows for results from each benchmark to be validated and/or reinforced by the other. The first method, the *Polling Method*, allows for the maximum possible overlap of computation and MPI communication. The second method, the *Post-Work-Wait Method* tests for overlap under practical restrictions on MPI calls. The following sections describe each of these methods in more detail.

```
read current time
for( i = 0 ; i < work/poll_factor ; i++ ){
   for( j = 0 ; j < poll_factor ; j++){
      /* nothing */
   }
   if(asynchronous receive is complete){
      start asynchronous reply(s)
      post asynchronous receive(s)
   }
}
read current time
```

Figure 1: Polling Method Psuedocode For Worker Process

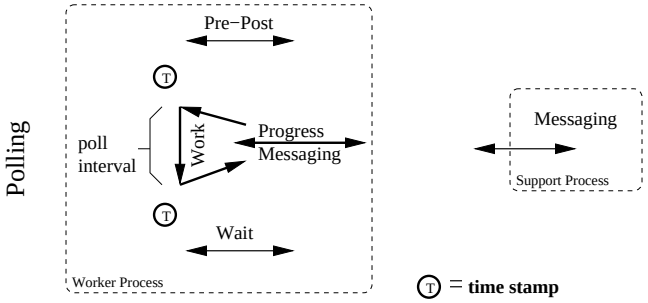

Figure 2: Overview of Polling Method

2.1 Polling Method

The *polling method* uses two processes, one process, the *worker process*, counts cycles and performs message passing. A second, *support process*, runs on the second node and only performs message passing. Figure 1 presents pseudo code for the worker process. All receives are posted before sends. Initial setup of message passing as well as conclusion of same are omitted from the figure. Additionally, Figure 2 provides a pictorial representation of the method.

This method uses a ping-pong communication strategy with messages flowing in both directions between sender node and receiver. Each process polls for message arrivals and propagates replacement messages upon completion of earlier messages. After a predetermined amount of computation, bandwidth and CPU availability are computed. The polling interval can be adjusted to demonstrate the trade-off between bandwidth and CPU availability. Because this method never blocks waiting for message completion it provides an accurate report of CPU availability.

As can be seen in Figure 1, after a fixed number of iterations in the inner loop the worker process polls for receipt of the next message. The number of iterations of the inner loop determines the time between polls and, hence, determines the polling

interval. If a test for completion is negative, the worker process will iterate through another polling interval before testing again. If a test for completion is positive, the process will post related messaging calls and will similarly address any other received messages before entering another polling interval. The support process sends messages as fast as they are consumed by the receiver.

We vary the polling interval to elicit changes in CPU availability and bandwidth. When the polling interval becomes sufficiently large all possible message transfers may complete during the polling interval and communication then must wait, resulting in decreased bandwidth.

The polling method uses a queue of messages at each node in order to maximize achievable bandwidth. When either process detects that a message has arrived, it iterates through the queue of all messages that have arrived, sending replies to each of these messages. When we set the queue size to one, a single message passed between the two nodes then the polling method acts as a standard ping-pong test and maximum sustained bandwidth will be sacrificed.

The benchmark actually runs in two phases. During the first, *dry run*, phase the amount of time to accomplish a predetermined amount of work in the absence of communication is recorded. The second phase records the time for the same amount of work while the two processes are exchanging messages. The CPU availability is reported as:

$$\text{availability} = \frac{\text{time(work without messaging)}}{\text{time(work plus MPI calls while messaging)}}$$

The polling method reports message passing bandwidth and CPU availability, both as functions of the polling interval.

2.2 Post-Work-Wait Method

The second method, the *post-work-wait method* or PWW, also uses bi-directional communication. However, this method serializes MPI communication and computation. The worker process posts a collection non-blocking MPI messages (sends and receives), performs computation (the work phase), and waits for the messages to complete. This strict order introduces a significant (and reasonable) restriction at the application level. Because the application does not make any MPI calls during its work phase, the underlying communication system can only overlap MPI communication with computation if it requires no further intervention by the application in order to progress communication. In this respect, the PWW method detects whether the underlying communication system exhibits application offload. In addition, as we will describe, this benchmark identifies where host cycles are spent on communication.

Figure 3 presents a pictorial representation of the PWW method. This method is similar to the polling method in that each process sends and receives messages, but only the worker process monitors CPU cycles.

With respect to communication, the PWW method performs message handling in a repeated pair of operations: 1) make non-blocking send and receive calls and 2) wait for the messaging to complete. Both processes simultaneously send and receive a single message. The worker process performs work after the non-blocking calls before

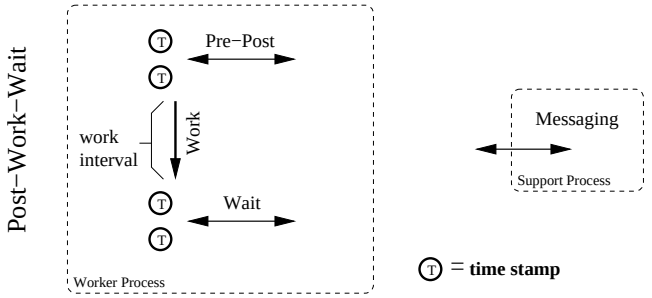

Figure 3: Post-Work-Wait (PWW) Method

waiting for message completion. As in the Polling method, the work interval is varied to effect changes in CPU availability and bandwidth.

The PWW method collects wall clock durations for the different phases of the method. Specifically, the method collects individual durations for i) the non-blocking call phase, ii) the work phase, and iii) the wait phase. Of course, the method also records the time necessary to do the work in the absence of messaging. These phase durations are useful in identifying communication bottlenecks or other causes of poor communication.

It is worth emphasizing here that the terms "work interval" and "polling interval" represent the foremost difference between the PWW method and the Polling method. After the polling interval, the Polling method checks whether or not there are arrived messages that require response but *in either case* "computation" then proceeds via the next polling interval. In contrast, after PWW's "work interval," the worker process waits for the current batch of messages even if the messages have not begun to arrive, such as in the case of a very short work interval. This is one of the most significant differences between the two methods and is key to correctly interpreting the results.

3 Platform Description

In this section we provide a description of the hardware and software systems from which our data was gathered.

Each node contained a 617 MHz Alpha EV67 processor with 256 MB of main memory and Myrinet [2] LANai 9 network interface card (NIC). Nodes were connected using a 64-port Mesh64 switch.

Results were gathered using the Portals 3.0 [5, 3] software designed and developed by Sandia National Labs and the University of New Mexico. Portals is an interface for data movement designed to support massively parallel commodity clusters, such as the Computational Plant [4]. We have also ported the MPICH implementation of MPI to Portals 3.0.

We gathered results from two different implementations of Portals for Myrinet. The first implementaiton of Portals for Myrinet used in our experiments is kernel-based.

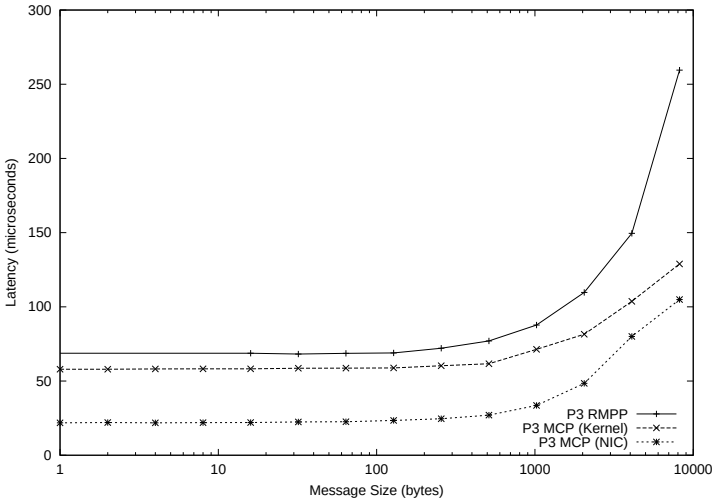

Figure 4: Half round trip latency performance.

The user-level Portals library interfaces to a Linux kernel module that processes Portals messages. This kernel module in turn interfaces to another kernel module that provides reliability and flow control using a protocol called RMPP (cite Rolf's dissertation). This kernel module works with any Linux network device that can send raw packets. For this kernel-based implementation, we have a Sandia-developed Myrinet Control Program (MCP) running on the Myrinet NIC that simply acts as packet engine. This particular implementation of Portals does not employ any OS-bypass techniques. This implementation is currently running in production on all the Cplant™ clusters at Sandia, the largest of which is 1792 nodes.

In our second implementation of Portals for Myrinet, all reliability and flow control is performed within the MCP. Processing of Portals messages can occur either in the MCP or via a Linux kernel module. A process can choose to have all processing of Portals messages occur on the card, which is expected to incur minimal host processor overhead, or have intitial processing done in an interrupt handler on the host. Once Portals processing has occurred in the interrupt handler, data is transferred directly from the the network into user space. Both of these methods of processing a Portals message employ OS-bypass, since the OS is not involved in the transfer of data once the final destination is determined. This is an experimental implementation of Portals that is currently running on a 10-node development system. It is currently limited to being able to send and receive messages into a 4 MB region of memory with a physically contiguous address space.

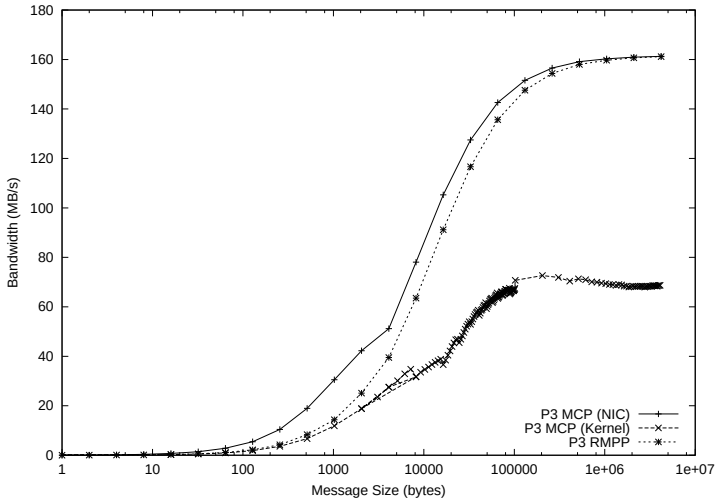

Figure 5: One-way bandwidth performance.

4 Results and Analysis

Figure 4 shows the MPI one-half round trip latency performance using a standard ping-pong microbenchmark. The zero-length MPI latency is 17μsecfor the Portals-based MCP where processing is handled entirely in the NIC, 48μsecfor the Portals MCP with processing in the kernel, and 66μsecfor the Portals over RMPP implementation. These numbers indicate that the cost of having the host processor involved in processing Portals messages via an interrupt routine in the kernel is significant.

Figure 5 shows the MPI one-way message-passing bandwidth. The asymptotic MPI bandwidth is 161 MB/s for both NIC and kernel processing for the Portals MCP, and 68 MB/s for Portals over RMPP. The curve for NIC processing in the Portals MCP is slightly steeper for smaller messages. For the RMPP implementation, the cost of copying packets from kernel-space to user-space is evident.

Figures 6 and 7 show the bandwidth calculated by the polling method. Initially , these bandwidth graphs exhibit a plateau of maximum sustained bandwidth until a point of steep decline. The point of steep decline occurs when the poll interval becomes large enough that all messages in flight are completed during the poll interval. When this happens, messages are delayed until the occurrence of the next poll. We see similar behavior for the PWW bandwidth calcuation in Figure 8.

Figures 9 and 10 show results of the polling method for 5 KB and 100 KB respectively. In these graphs, availability remains low and relatively stable until it rises steeply. Before the steep increase, polling is so frequent that messages are processed as soon as they arrive. This keeps the system active with message handling and availability is kept low. CPU availability steeply climbs when the poll interval becomes infrequent enough to cause stops in the flow of messages. For the RMPP-based imple-

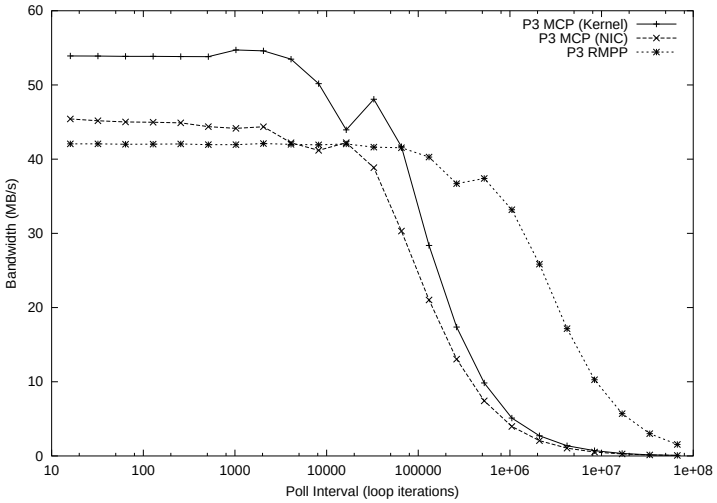

Figure 6: Polling method: bandwidth for 5 KB messages.

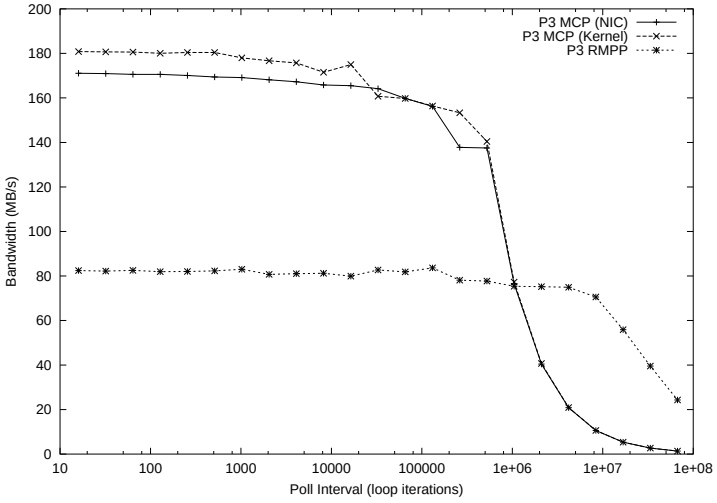

Figure 7: Polling method: bandwidth for 100 KB messages.

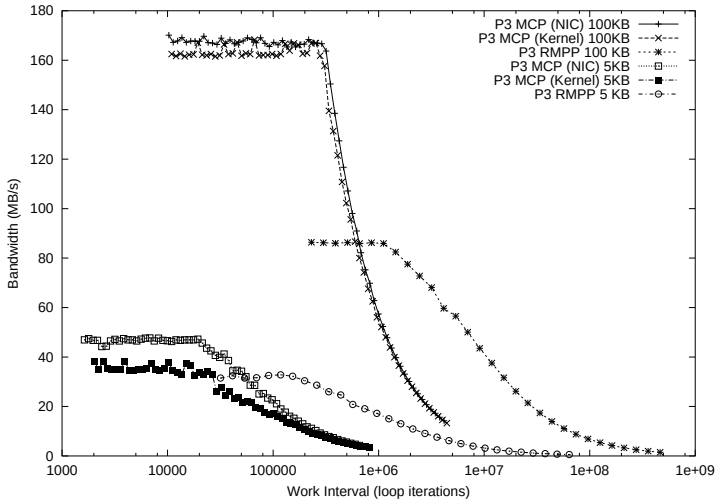

Figure 8: PWW method: bandwidth for 5 KB and 100 KB messages.

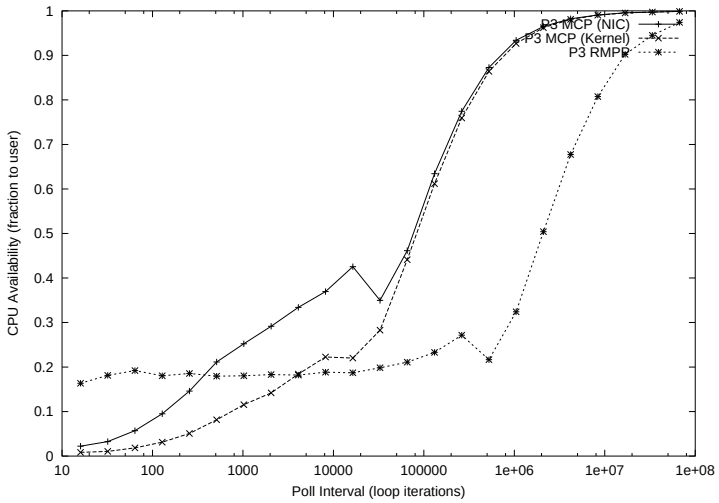

Figure 9: Polling method: CPU availability for 5 KB messages.

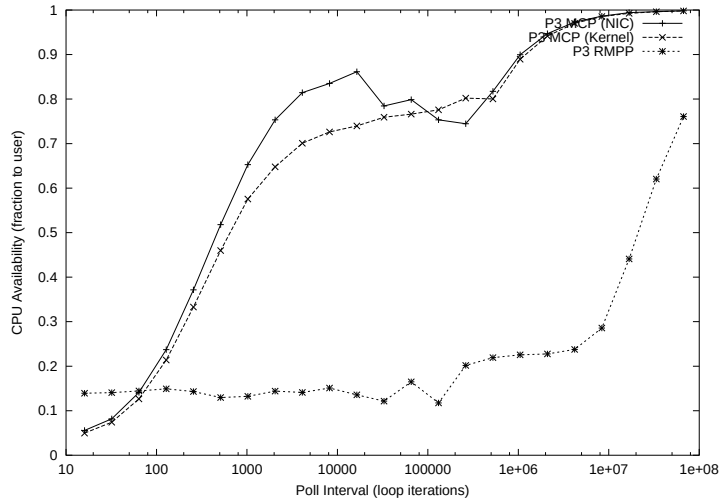

Figure 10: Polling method: CPU availability for 100 KB messages.

mentation, lack of message handling equates to lack of interrupts and the application no longer competes for CPU cycles.

The PWW availability graphs, Figures 11 and 12, lack the initial plateau as seen in the polling availability graphs. This difference is due to the fact that the polling method returns to work (i.e., to another polling interval) if a message has not yet arrived, whereas the PWW method waits regardless of what the cause is for the delay. This *wait while delayed* functionality suppresses apparent CPU availability until the work interval becomes sufficiently long to fill the delay period of time.

4.1 CPU Overhead

We now examine the work phase of the PWW method. The duration of the work phase is of interest when considering communication overhead. Depending on the system, a separate process or the kernel itself could facilitate communication while competing with the user application for CPU time. In such cases, the time to complete the work phase during messaging will take longer than the time to complete the same work in the absence of a competing process.

Figure 13 shows the time to complete work as a function of work interval. Recall that both methods time the duration needed to complete work with and without communication. In Figure 13, the work with message handling takes a greater amount of time relative to work without message handling; the difference is due to the overhead of interrupts needed to process Portals messages.

In contrast, Figure 14 displays results for the Portals MCP using kernel processing and shows much less communication overhead in that the time to do work is the same regardless of the presence or absence of communication. As expected, Figure 15 shows

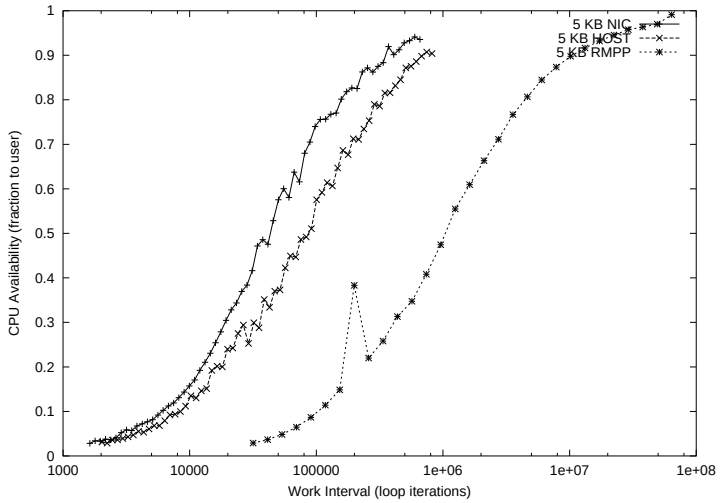

Figure 11: PWW method: CPU availability for 5 KB messages.

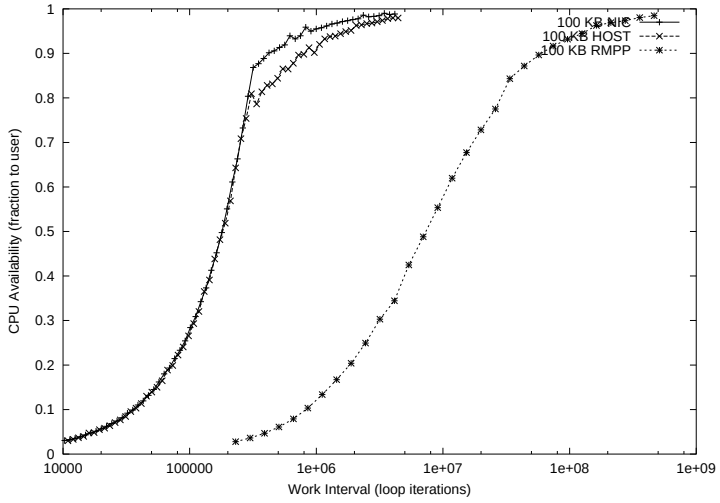

Figure 12: PWW method: CPU availability for 100 KB messages.

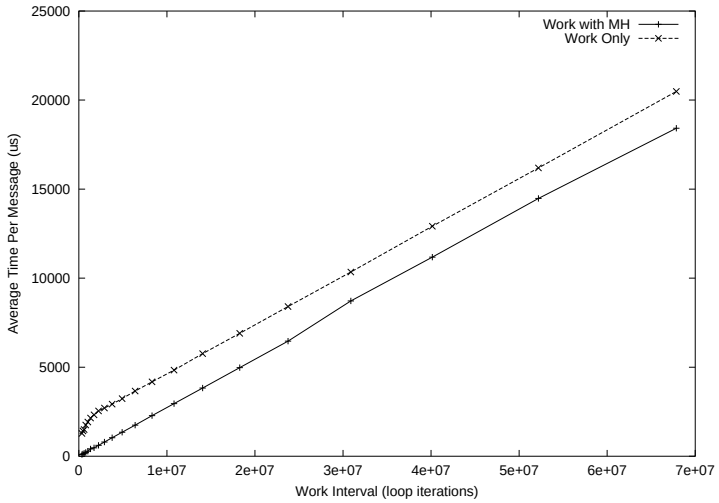

Figure 13: PWW Method: CPU overhead for RMPP.

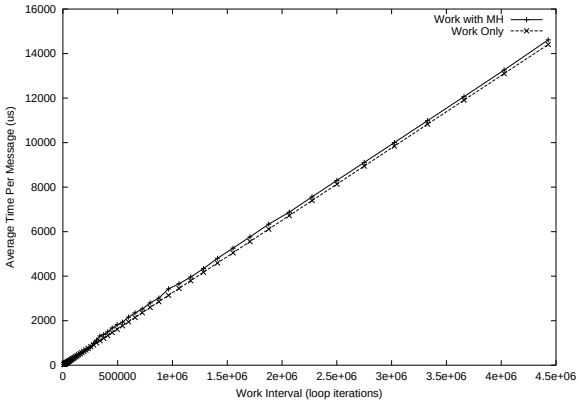

Figure 14: PWW Method: CPU overhead for P3 MCP (kernel).

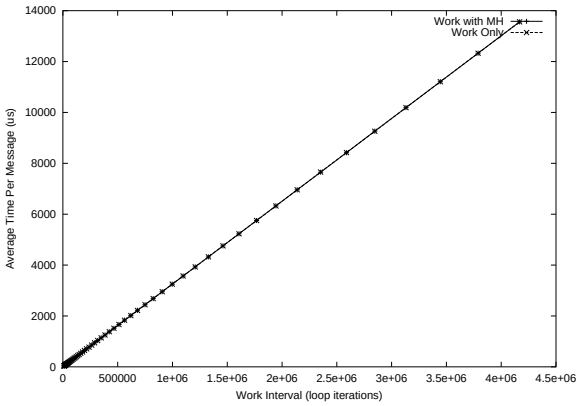

Figure 15: PWW Method: CPU overhead for P3 MCP (NIC).

virtually no overhead for communication since all message processing is done on the Myrinet NIC. The lack of a time gap between work with and without message handling is the general indicator of a system that lacks communication overhead.

5 Related Work

Previous work related to assessing the ability of platforms to overlap computation and MPI communication have simply characterized systems as being able to provide overlap for various message sizes [8]. Our benchmark suite extends this base functionality in an attempt to gather more detailed information about the degree to which overlap can occur and the effect that overlap can have on latency and bandwidth performance. For example, our benchmark suite is able to help assess the overall benefit of increasing the opportunity to overlap computation and MPI communication at the expense of decreasing raw MPI latency performance.

The netperf [1] benchmark is commonly used to measure processor availability during communication. Our benchmarks uses the same general approach as that used in netperf. Both benchmarks measure the time taken to execute a delay loop on quiescent system; then measure the time taken for the same delay loop while the node is involved in communication; and report ratio between the first and second measurement as the availability of the host processor during communication. However, in netperf, the code for the delay loop and the code used to drive the communication are run in two separate processes on the same node.

Netperf was developed to measure the performance of TCP/IP and UDP/IP. It works very well in this environment. However, there are two problems with the netperf approach when applied to MPI programs. First, MPI environments typically assume that there will be a single process running on a node. As such, we should measure processor availability for a single MPI task while communication is progressing in the background (using non-blocking sends and receives). Second, and perhaps more important,

the netperf approach assumes that the process driving the communication relinquishes the processor when it waits for an incoming message. In the case of netperf, this is accomplished using a *select* call. Unfortunately, many MPI implementations use OS-bypass. In these implementations, waiting is typically implemented using busy waiting. (This is reasonable, given the previous assumption that there is only one process running on the node.)

6 Summary

In this paper, we have described the COMB benchmark suite. We have described the methods and approach of COMB and demonstrated its utility in providing insight into the underlying implementation of communication system.

Of the two methods used in the suite, the polling method is distinguished by providing a basis for viewing a systems performance in an unfettered manner. The polling method makes periodic calls to the MPI library and logs computation whenever the user application does not need to progress messaging. The result is that maximum overlap between communication and computation is allowed As such, the polling method provides a basis for an unqualified or general comparison between different systems.

In contrast, the PWW method identifies actual limitations with respect to application offload. and provides timing information which identifies where the hosts spent time on communication, such as overhead during the work phase. As such, the PWW method provides performance comparisons in many areas and provides a means to help identify bottlenecks during the post-work-wait cycle.

We believe COMB is a useful tool for the analysis of cluster communication performance. We have used it extensively to benchmark several systems, both development and production, and it has provided new insights into the effects of different implementation strategies.

References

[1] *Netperf*. `http://www.netperf.org`.

[2] N. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet-a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995.

[3] 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.

[4] R. B. Brightwell, L. A. Fisk, D. S. Greenberg, T. B. Hudson, M. J. Levenhagen, A. B. Maccabe, and R. E. Riesen. Massively Parallel Computing Using Commodity Components. *Parallel Computing*, 26(2-3):243–266, February 2000.

[5] R. B. Brightwell, T. B. Hudson, A. B. Maccabe, and R. E. Riesen. The Portals 3.0 Message Passing Interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999.

[6] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 85–97, New YOrk, June 2–4 1997. ACM Press.

[7] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[8] J. White and S. W. Bova. Where's the Overlap?: An Analysis of Popular MPI Implementations. In *Proceedings of the Third MPI Developers' and Users' Conference*, March 1999.