OIT Technical Report

MPI Performance of the UMBC *hpc.rs.umbc.edu* Cluster

Samuel G Trahan
straha1@umbc.edu
UMBC OIT

Abstract: This technical report examines the performance of OpenMPI 1.2.6, MVAPICH2 1.0.3 and MVAPICH 1.0.1 on the UMBC *hpc.rs.umbc.edu* cluster. This report presents the bandwidth of MPI_Alltoall, MPI_Allreduce, MPI_Bcast, MPI_Send, MPI_Recv and MPI_Reduce, as well as the round trip time of MPI_Send/MPI_Recv pairs. Due to recent hardware issues, this report is restricted to 1 to 12 machine cases. A bad Infiniband card on *node032* ended the study before the planned 16, 24 and 32 machine cases. Additionally, this paper examines a bug in MVAPICH 1.0.1 that prevents its use for jobs that require more than seven nodes, as well as other issues in MVAPICH2 and OpenMPI. Lastly, this report explains the need for load balancing on *hpc.rs.umbc.edu*.

Introduction

The Message Passing Interface (MPI) is the industry standard for communication between distributed memory jobs. MPI-based programs execute simultaneously on several machines at once. The various instances of the program (referred to as *processes*) communicate by passing messages between each other. MPI makes this easy by providing subroutines that perform message passing for the program. The MPI implementation handles the details of the particular underlying communication interface, and algorithms for efficient communication, allowing scientists and other programmers to worry about matters more relevant to their work.

MPI supports a number of communication patterns through C, Fortran and C++ application programing interfaces (APIs; essentially sets of callable functions). Most of these communication patters involve one process sending a message to all other processes (or receiving a message from all other processes), some involve communication between only two processes and others involve all processes communicating with each other simultaneously. Some of the more common MPI functions are the ones benchmarked by this study:

- *MPI_Send, MPI_Recv* – MPI_Send sends a single message from one process to another specific process. MPI_Recv receives the sent message.
- *MPI_Bcast* – MPI_Bcast sends a message from one process to all other processes.
- *MPI_Reduce* – MPI_Reduce takes an array of numbers on one process, splits the array up into several parts and sends each part to a different process. The processes then perform some task on the data (such as summing it or finding the maximum value). They then send the result back to the originating process, which combines the results (such as by summing the sums, or getting the maximum of the maximums).
- *MPI_Alltoall* – MPI_Alltoall sends a distinct message from every process to every other process. (That is, if you have *n* processes, then each one sends *n-1* messages – one to each other process.)

- *MPI_Allreduce* – MPI_Allreduce is somewhat of a combination of MPI_Alltoall and MPI_Reduce. As in MPI_Reduce, one process provides a list of numbers which is then split up evenly among all other processors. The processors then work together to perform a single computation on the data, such as summing the numbers. Unlike MPI_Reduce, the result is not merely sent back to the originating process. It is also sent to all other processes.

In addition to bandwidth measurements, we also will examine unidirectional latency of MPI_Send and round trip time of MPI_Send/MPI_Recv pairs. The unidirectional latency measures how long it takes to tell the MPI implementation to send a message out. The round trip time measures how total time it takes to send a message out (via MPI_Send) and receive one back again (via MPI_Recv).

Throughout this document, *node* or *machine* will refer to one physical computer. *Processor* or *core* will refer to one processor core on one machine. The *hpc.rs.umbc.edu* cluster is made up of thirty-two nodes, each with two processor cores. Due to hardware failures, this study is limited to jobs running on twelve nodes or less.

Methodology

This study uses the Low Level Characterization Benchmark (*llcbench*) benchmarking suite. It is designed to benchmark the smallest possible subset of the MPI implementation at a time. It does this through the use of tight loops surrounded by timers. For example, here is the loop in the MPI_Send/MPI_Recv bandwidth test:

```
TIMER_START;
for (i=0; i<cnt; i++)
     mp_send(dest_rank, 1, sendbuf, bytes);
mp_recv(dest_rank, 2, destbuf, 4);
TIMER_STOP;
```

Here, *mp_send* and *mp_recv* are *llcbench*'s wrappers around MPI_Send and MPI_Recv, respectively. TIMER_START and TIMER_STOP are C Preprocessor macros that calculate the current time via the use of the UNIX clock_gettime function's real time mode, which returns the time in nanoseconds. The single call to *mp_recv* ensures that all of the *mp_send* calls finish before the call to TIMER_STOP. The reason that works is that the remote process does not call *mp_send* until after it has called *mp_recv cnt* times (and process 0's *mp_recv* cannot complete until someone has sent data to process 0).

I compiled *llcbench* using OpenMPI 1.2.6, MVAPICH 1.0.1 and MVAPICH 1.0.3, each with the PGI compiler and the GCC compiler (six different combinations). I ran the *llcbench* benchmark suite for all six combinations on 1, 2, 3, 4, 6, 8 and 12 machines, using 1, 2 and 4 processors per machine.

Results

The results are split into three sections. The first section discusses the latency and two process communication bandwidth (the good news). The second section discusses the multiprocess communication bandwidth (the bad news). The last section discusses the critical bug in MVAPICH 1.0.3 and other MPI-related issues.

Two Process Communication

The two process communication methods analyzed here use the MPI_Send and MPI_Recv commands to send messages between the rank 0 process and the rank *n*-1 process, where *n* is the number of processes.

The first result I will present here is the latency and round trip time – two aspects of the same benchmark. The latency benchmark calls MPI_Send five thousand times and calculates the average time taken by an MPI_Send call. That benchmark can be deceptive since the MPI implementation has the freedom to store multiple MPI_Send messages and send them all at once in a single communication.

Latency and Round Trip Time

The roundtrip benchmark does not have that problem. It calls MPI_Send followed by MPI_Recv, repeating that pair of calls five thousand times. It then calculates the average time taken by a pair of calls. In order for the MPI_Recv to succeed, the remote process must have received and acknowledged the MPI_Send. That ensures that the MPI implementation cannot simply cache five thousand MPI_Send calls and send them all at once. The results of the round trip benchmark do not vary noticably between MPI implementations, machine counts or processor per node variations. OpenMPI displays slightly better performance in the latency benchmark for small message sizes, indicating that it does a better job of caching multiple messages before sending them (see Figure 1).

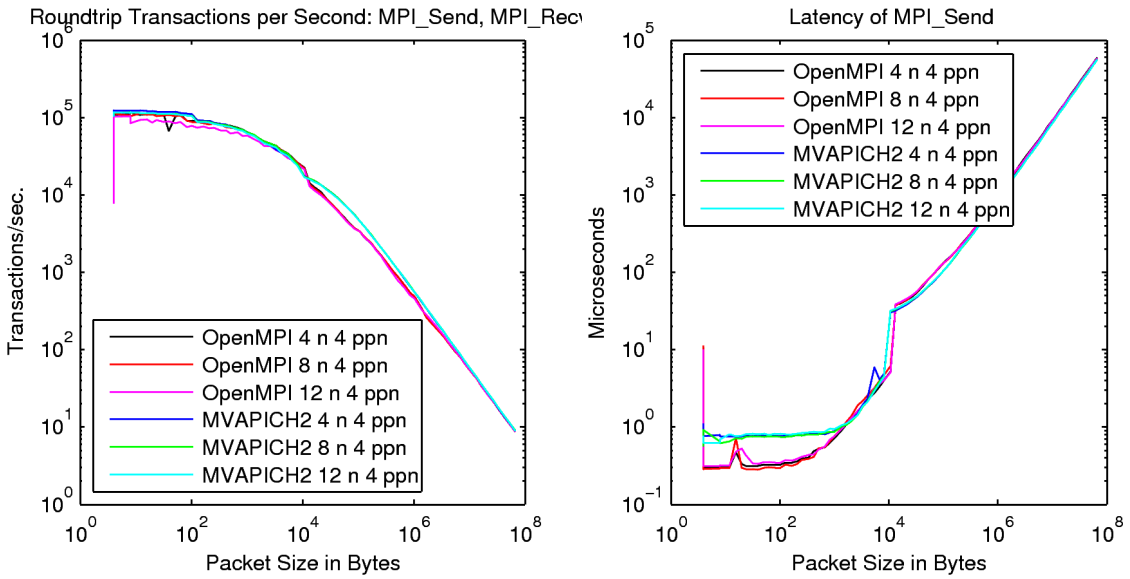


Figure 1: The round trip performance (left) is measured in transactions per second. It starts at a rate of 90,000 to 115,000 transactions per second, which corresponds to a round trip time of about 10 microseconds – equivalent to a unidirectional latency of only 5 microseconds. The round trip time remains quite fast until around 1 kB messages, when it begins to drop off roughly linearly with the message size. There are no noticeable differences between the different MPI implementations. There are also no differences between different numbers of processors per node (not shown). The latency (right) displays similar behavior, except that OpenMPI has about half of the latency of MVAPICH2 for small message sizes. Note that the latency benchmark, unlike the round trip benchmark, allows the MPI implementation to cache several MPI_Send calls before sending the messages. Thus the latency benchmark indicates that OpenMPI does a better job of caching MPI_Sends than MVAPICH2 does.

Bandwidth: Unidirectional and Bidirectional

The next type of benchmark is two process bandwidth benchmarks. The *llcbench* suite contains unidirectional and bidirectional two process bandwidth benchmarks. The unidirectional bandwidth benchmark calls MPI_Send many times, followed by a single MPI_Recv (to ensure that the MPI implementation actually sends all of the MPI_Send messages). It then reports the average bandwidth in kB/s. The bidirectional bandwidth benchmark calls MPI_Send once, then MPI_Recv once, and repeats the pair thousands of times. It then reports the average bandwidth. In the unidirectional bandwidth benchmarks, MVAPICH2 consistently outperforms OpenMPI by a factor of two to four for message sizes of less than 10 kB. In the bidirectional bandwidth benchmark, the two have approximately equal performance (Figure 2).

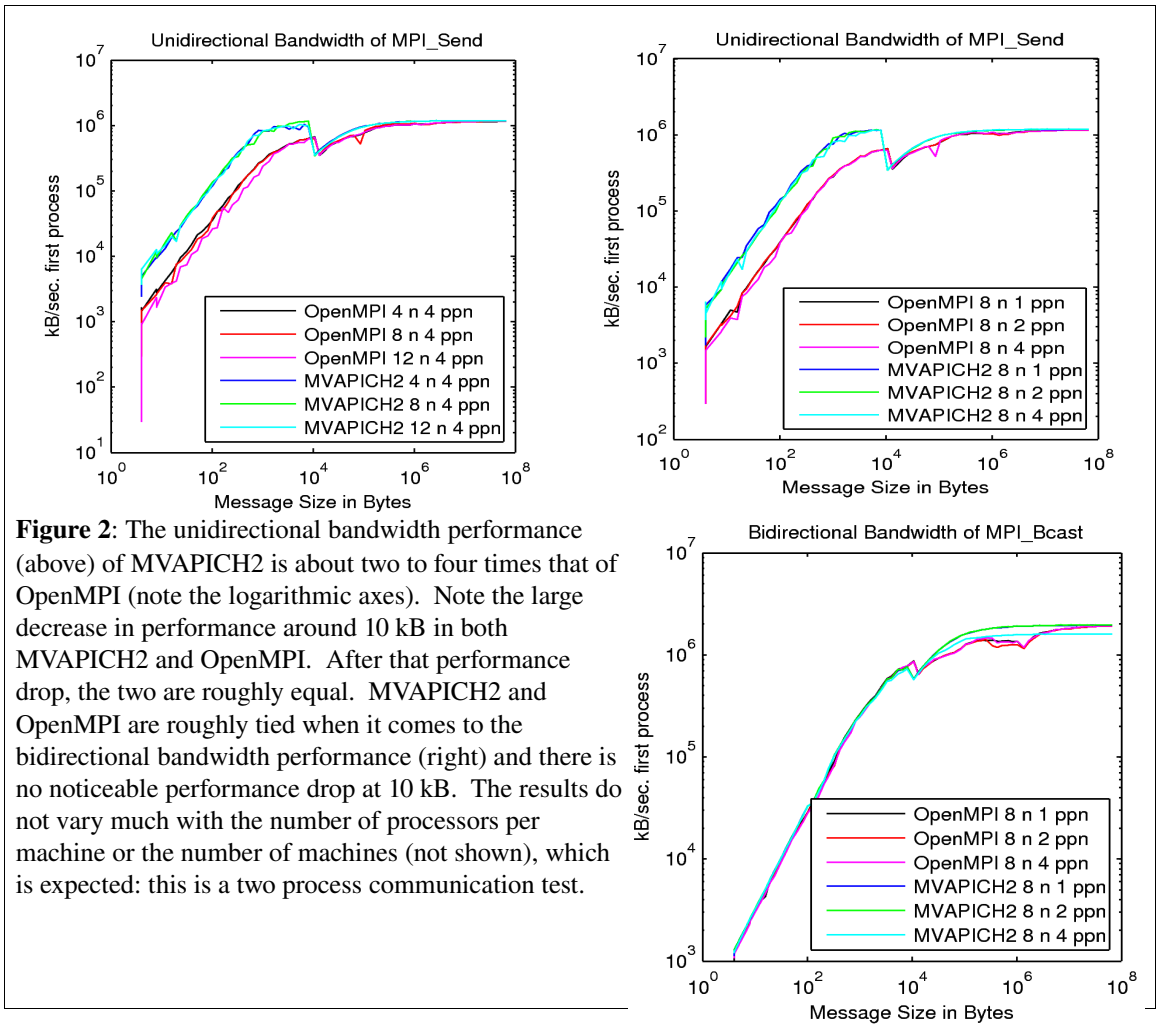


Figure 2: The unidirectional bandwidth performance (above) of MVAPICH2 is about two to four times that of OpenMPI (note the logarithmic axes). Note the large decrease in performance around 10 kB in both MVAPICH2 and OpenMPI. After that performance drop, the two are roughly equal. MVAPICH2 and OpenMPI are roughly tied when it comes to the bidirectional bandwidth performance (right) and there is no noticeable performance drop at 10 kB. The results do not vary much with the number of processors per machine or the number of machines (not shown), which is expected: this is a two process communication test.

Multiprocess Benchmarks

The next class of benchmarks examine the bandwidth of MPI_Allreduce, MPI_Alltoall, MPI_Bcast, MPI_Reduce. All four of those MPI calls communicate between all processes in the MPI program, unlike MPI_Send and MPI_Recv. As with MPI_Send and MPI_Recv, *llcbench* calculates the bandwidth in terms of the number of bytes per second sent out from process 0. However, in these benchmarks, I convert the bandwidth to kB/s for the whole machine by dividing *llcbench*'s bandwidth result by the number of processors used per node.

The performance of MPI_Reduce (Figure 3) and MPI_Allreduce (Figure 4) is reasonable. MPI_Alltoall (Figure 5) suffers serious performance problems, while MPI_Bcast (Figure 6) displays similar performance to MPI_Reduce and MPI_Allreduce.

MPI_Reduce Bandwidth Per Machine

In the MPI_Reduce benchmark, process 0 assembles an array of MPI_INT values and instructs the processes in the MPI program to collectively sum the array. The message size reported in Figure 3 is the total size of that array. There are no guarantees made about the algorithm the MPI implementation will use to divide up the array among the available processes – the only guarantee is that the processes will collectively sum the data.

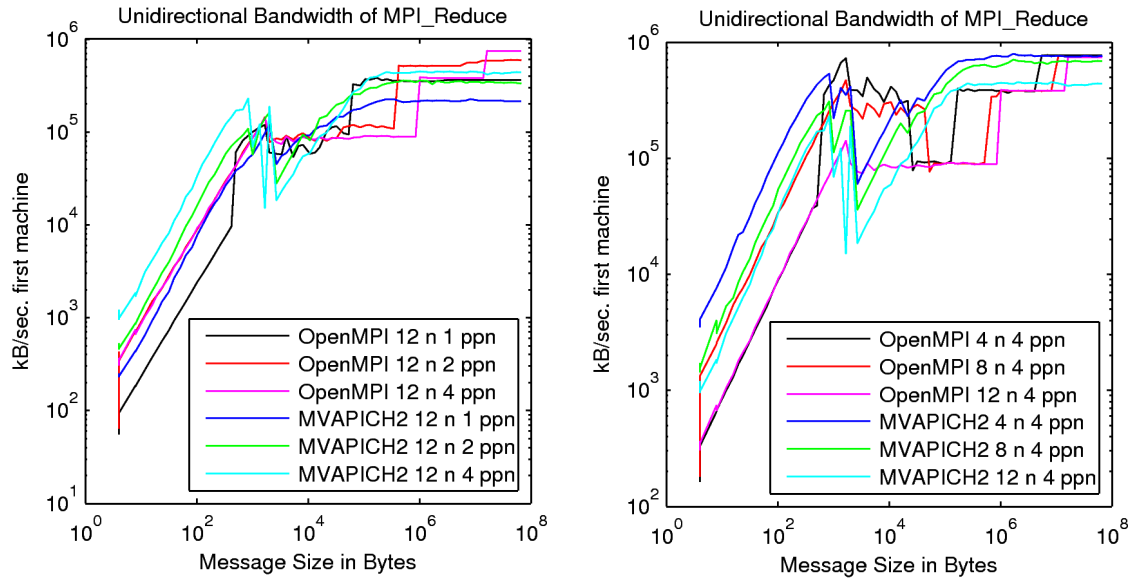
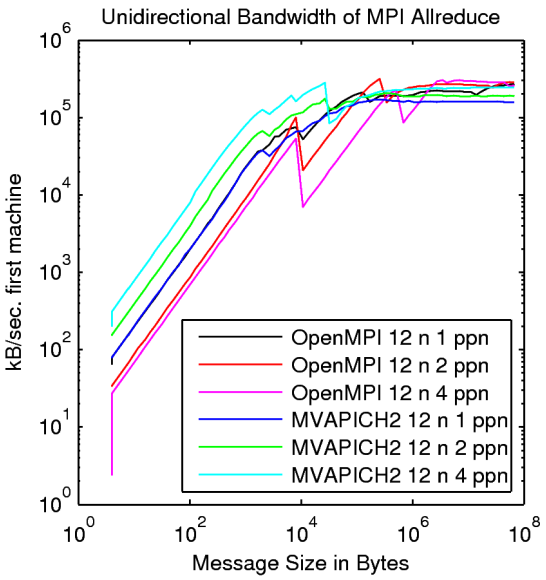
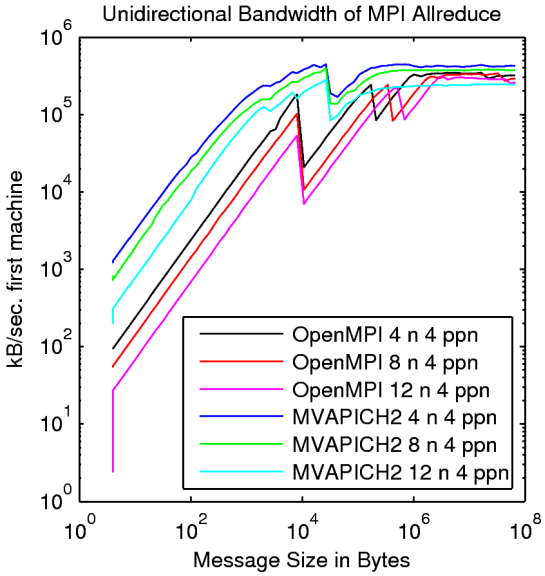


Figure 3: MVAPICH2 outperforms OpenMPI by about a factor of three for small message sizes. MVAPICH2's bandwidth drops abruptly by a factor of two to five at messages around 3 kB in size. A similar drop is visible in the OpenMPI performance when using four or eight nodes (right) though that feature is not present in the one and two processor per machine. cases (not shown). One important result to take away from this is that MPI_Reduce's performance is very poor for messages less than 1 kB in size for both OpenMPI and MVAPICH2.

MPI_Allreduce Bandwidth Per Machine

MPI_Allreduce works much like MPI_Reduce – process 0 creates an array of MPI_INT values and orders the collection of processors to sum the results. Unlike MPI_Reduce, MPI_Allreduce forwards the results of that sum to all processors. The performance of MPI_Allreduce is decent, though there are some ranges of message sizes over which OpenMPI performs far more poorly than MVAPICH2.





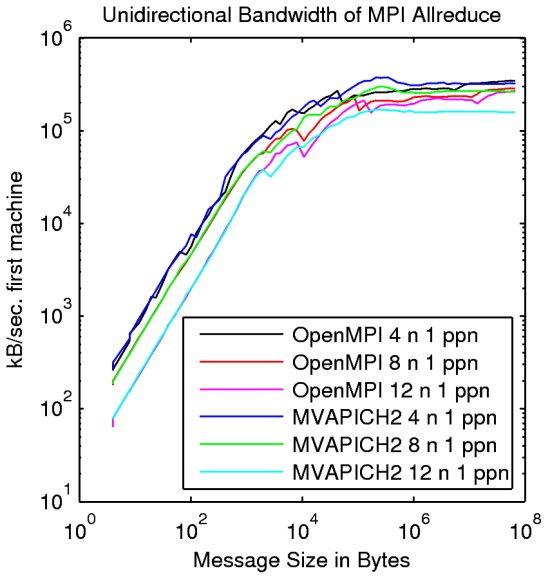
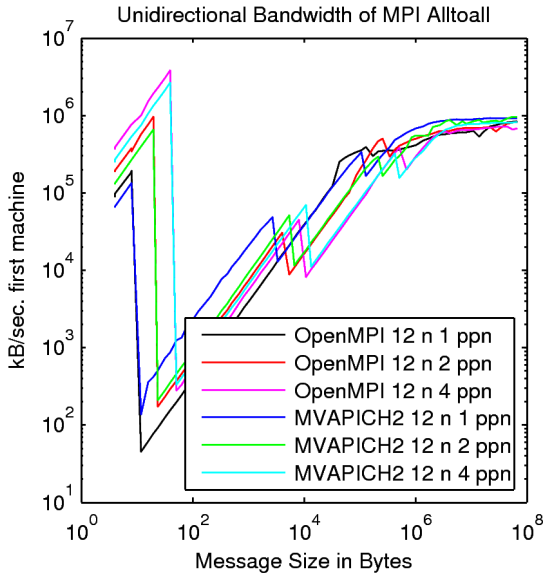


Figure 4: MVAPICH2's MPI_Allreduce performance is nearly as good as its MPI_Reduce performance. As before, the performance is poor for messages less than 1 kB in size, but decent (100 MB/s) for larger messages). Notable though is the factor of three drop in performance around 30 kB messages. In the two and four processor per node cases, OpenMPI's performance is consistently one third to one tenth of MVAPICH2 for small messages. OpenMPI's bandwidth drops by a factor of ten at around 8 kB messages and again by a factor of three at around 100-1000 kB messages (depending on the number of nodes).

Note that for one processor per node cases (right), the performance of the two is almost identical. Drops in OpenMPI's bandwidth are still visible around 8 kB and 100 kB, the drops are only enough to produce about a factor of two difference in performance for a small range of message sizes.

MPI_Alltoall Bandwidth Per Machine

MPI_Alltoall sends a distinct message from each process to each other process. The MPI implementation is free to distribute these messages by any algorithm it wishes. Unlike the past benchmarks, the results here are very disappointing. Over a large range of message sizes, MVAPICH2 and OpenMPI MPI_Alltoall bandwidth is atrocious nearly to the point of being useless.



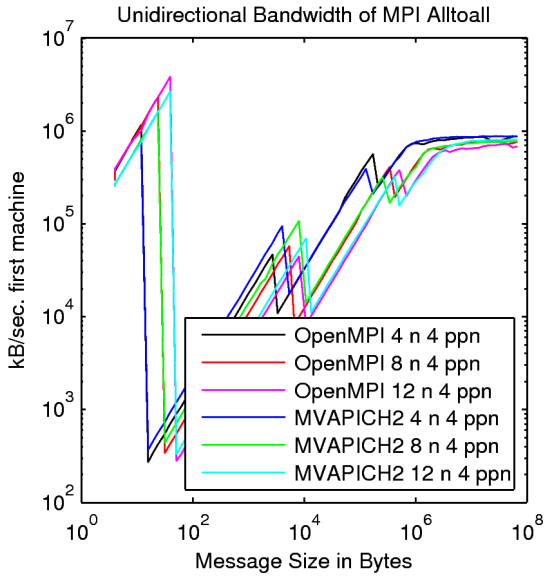
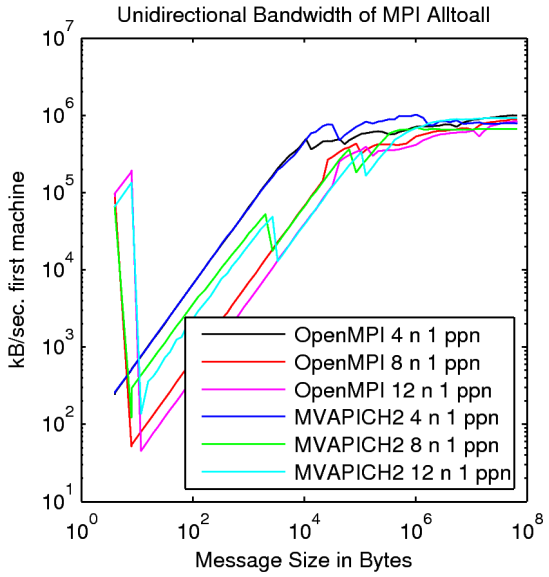


Figure 5: The performance of both MVAPICH2 and OpenMPI is very poor in this case. The bandwidth does not reach a usable level until 10-100 kB messages (depending on the number of nodes and processors per node). The bandwidth starts out, very briefly, at a usable level until two to eighty byte messages, after which the bandwidth crashes to 60-300 kB/s – a level comparable to what one might expect from Comcast, but not from Infiniband. There are two other noticeable drops in performance at around 1-10 kB and again at 100-1000 kB.

For the most part, the performance of OpenMPI and MVAPICH is roughly the same, indicating that the problem may be in hardware rather than software. However, MVAPICH2 has roughly two to four times the performance of OpenMPI for messages of 10-1000 bytes. (Its performance in that regime is still terrible though.)



MPI_Bcast Bandwidth Per Machine

MPI_Bcast broadcasts a message from process 0 to all other processes. As with the other calls, there are no guarantees about how the message will get to its destinations. The *llcbench* suite calculates MPI_Bcast bandwidth by running MPI_Bcast five thousand times, and then MPI_Recv once for each process whose rank is greater than zero (to ensure that the broadcasts complete rather than being cached somewhere). I then multiply the bandwidth by the number of processors used per machine to calculate the total bandwidth of the first machine.

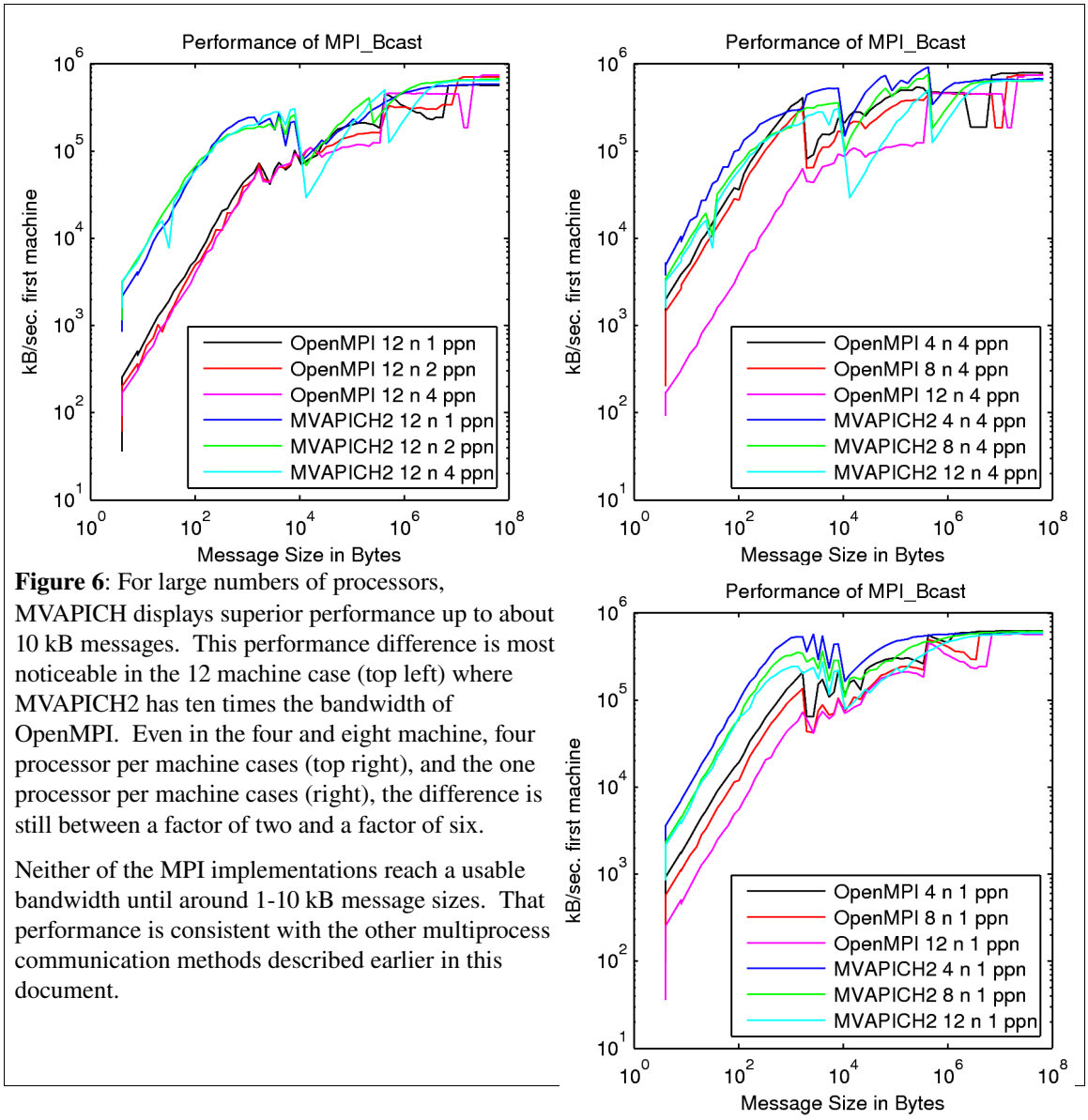


Figure 6: For large numbers of processors, MVAPICH displays superior performance up to about 10 kB messages. This performance difference is most noticeable in the 12 machine case (top left) where MVAPICH2 has ten times the bandwidth of OpenMPI. Even in the four and eight machine, four processor per machine cases (top right), and the one processor per machine cases (right), the difference is still between a factor of two and a factor of six.

Neither of the MPI implementations reach a usable bandwidth until around 1-10 kB message sizes. That performance is consistent with the other multiprocess communication methods described earlier in this document.

Other MPI Issues

MVAPICH 1.0.1

You may have wondered about the conspicuous absence of MVAPICH 1.0.1, which was prominently featured in the abstract of this technical report. MVAPICH 1.0.1 has a critical bug causing it to crash the user's program if he, she or it uses MPI_Bcast to communicate between eight or more machines (cluster nodes – not processes). This bug is a known bug that has already been fixed in a developmental branch of MVAPICH 1.0.1 as of August 21, 2008 (possibly earlier). Due to this bug, I did not test MVAPICH 1.0.1.

MVAPICH2 1.0.3

This MPI implementation displays reasonable performance in most cases and it does not have the critical bug that MVAPICH 1.0.1 has. It appears at first glance to be the best choice for an MPI implementation, however there are significant issues with MVAPICH2:

1. MVAPICH has no *mpirun* script. Thus users must call the low-level *mpdboot*, *mpiexec* and *mpdallexit* functions directly. The *mpiexec* function has an especially confusing argument parsing routine that requires certain arguments to appear in a certain order. In addition, the user must set up a *~/.mpd.conf* file in their home directory. This process is overly complex and unnecessarily so. It will likely confuse many users.
2. Dr. Gobbert has noticed that MVAPICH2 does not recover gracefully from a crashed program, unlike OpenMPI which happily exits when your program crashes. This has led to MVAPICH2 jobs getting stuck in the queue.
3. As with OpenMPI, the bandwidth of MVAPICH2's MPI_Alltoall is horrible.

OpenMPI

OpenMPI is filled with features, and is quite easy to use but is substantially slower than MVAPICH2 in a number of cases. The MPI_Bcast performance is two to four times slower for messages less than about 10 kB in size. The MPI_Allreduce performance is ten to thirty times slower until around 30 kB messages. The MPI_Reduce performance is up to four times slower for messages less than 300 bytes in size. The unidirectional MPI_Send bandwidth is around one third that of MVAPICH for messages less than 8 kB in size. OpenMPI's bandwidth is littered with sudden performance drops at various message sizes – performance drops that are not present in the MVAPICH2 results (with the notable exception of MPI_Alltoall). This suggests that OpenMPI may simply be misconfigured.

Fortunately, OpenMPI is configurable at runtime by the user through various flags to *mpirun*. The OpenMPI project FAQ provides a number of suggestions about how to improve bandwidth:

http://www.open-mpi.org/faq/?category=openfabrics

That link also provides suggestions as to how system administrators can configure switches and other hardware to improve Infiniband performance.

Lack of Load Balancing

The *hpc.rs.umbc.edu* cluster is currently set up to launch jobs on the highest-numbered cluster nodes first. The first node listed in the user's job list is the highest-numbered node. That is going to wear down the

higher-numbered nodes sooner than the lower-numbered ones. It will have an especially large impact on *node032*'s Infiniband card, due to the nature of MPI programs. The first machine in the machine file gets MPI process 0. That process performs more message passing than the other processes since it tends to be the root of multiprocess communications such as MPI_Bcast, MPI_Alltoall, MPI_Reduce and MPI_Allreduce. Indeed, twice now *node032* has crashed during benchmarking, and the second time the Infiniband card failed before the rest of the machine. To make matters worse, the PBS queuing system is unable to end a job cleanly if it cannot contact the job launching demon on the job's initial node (the node on which the user's qsub script runs). That means any 32-node jobs that are running when *node032* has crashed will tie up the entire cluster until a system administrator can restart the relevant demons.

It appears that *hpc.rs.umbc.edu* is using MAUI, which includes the MOAB scheduler. According to the administrator's documentation for the MOAB scheduler, there is no load balancing. The “load balancing” page in the manual states that the scheduler only lets you mark nodes as available or unavailable – you cannot prioritize them. Marking a node as available or unavailable is considered a “load balancing” feature since, on machines that run several jobs at a time, you can mark a machine as unavailable if it has too many jobs running. That doesn't help load-balance a cluster – it only keeps a node from running more jobs than it has processors. Unfortunately, standard OpenPBS doesn't include any real load balancing either.

One possible solution to this problem is to simply reorder the node list once a week or so. There is no way to tell the *pbs_server* demon to re-read its node list file – one can only achieve that effect by killing and restarting the *pbs_server* demon. A possible solution to this problem is to create a script that will watch the PBS queue and the terminals on the head node, wait for a time when the queue and terminals are all idle, and then restart the *pbs_server* demon with a new node list. The ideal solution would be a new scheduling algorithm, possibly as part of a commercial queuing system such as the commercial version of PBS or IBM's LoadLeveler.

Conclusions and Future Work

Unfortunately, there are no good MPI implementations to chose from on the *hpc.rs.umbc.edu* cluster at this point. The best performer is MVAPICH2 1.0.3. The easiest to use is OpenMPI, but it preforms much more poorly than MVAPICH2 1.0.3. MVAPICH 1.0.1 is not usable at this time due to a bug that causes the user's program to crash if he or she uses MPI_Bcast between eight or more machines. Both MVAPICH2 and OpenMPI have extremely poor, bordering on useless, MPI_Alltoall performance between message sizes of 10 to 10,000 bytes.

Due to a failure of the Infiniband card on *node032*, I have temporarily suspended these benchmarks. The reason *node032* is critical is that PBS refuses to kill a job if the job's initial node (the one running one's qsub script) goes down. Due to the lack of load balancing, *node032* is nearly always the initial node. Thus, if any of my jobs crash, they tie up most or all of the cluster until a system administrator can restart the pbs server. I hope to continue later on if the hardware becomes more reliable.

In the future, if MVAPICH 1.0.1 is fixed, I may re-run the benchmarks with that MPI implementation. The limited benchmarks I've run for MVAPICH 1.0.1 (using one to six machines) have shown that its performance is nearly the same as MVAPICH2 1.0.3. I also intend to test OpenMPI and MVAPICH2 all the way up to 32 nodes, and I will re-test OpenMPI while tuning its performance.