Performance Comparison of Open Source MPI Implementations

Erik McClements

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2006

Abstract

An investigation and analysis of Open Source MPI implementations, comparing and assessing performance across a variety of HPC platforms and interconnects. An investigation into the reasons for performance difference is also undertaken, along with suggestions on improvement for specific MPI libraries.

Contents

List of Figures

Acknowledgements

Project proposer David Henty, and for his help starting the project.
Project supervisor Stephen Booth, for continued help and advice throughout the project.
The Open MPI, MPICH and LAM/MPI developers, for providing the MPI implementations.
Intel, for providing their MPI benchmark suite.
EPCC, Daresbury Laboratories and the eScience Centre for providing the various machines involved with this project.

Chapter 1

Introduction

High Performance Computing (HPC) machines are increasingly constructed using commodity processors and interconnects. On such systems it is common practice to install a free, Open Source, Message Passing Interface (MPI) implementation to facilitate parallel computing.
Traditionally there has been a trade-off between proprietary MPI implementations and the free alternatives. These fee-based implementations often supply specific installation instructions or binary packages, are expected to be tuned for particular architectures and even provide user support. It is therefore of interest to see what the overheads of getting a free implementation up and running are, and if this lack of tech support is really of significance. If the "you get what you pay for" saying is true, then these free implementations may not function at all, or be severely limited in function or performance.
Open Source implementations have existed for many years, but recently the "next-generation" of implementations have emerged, in the wake of the new MPI-2 standard. With MPI-2, some existing MPI-1 implementations such as LAM/MPI decided to build this into the previous implementation. However with MPICH a new implementation, that shared no MPICH1 code was developed. LAM/MPI has since ceased development in favour of the new collaborative project Open MPI, completely redesigning from the ground up with MPI-2 compliance.
The main aim of this project is to compare and contrast various Open Source MPI implementations, identifying their strengths and weaknesses across multiple machine architectures commonly used for HPC. Any found performance differences will then be investigated, in part by any associated literature but also by utilising the ability to view the source code of these libraries to provide reasons why performance varies. This access to MPI library internals should enable identification and assessment of each implementation's chosen algorithms. The varying qualities of different implementations will be illustrated with performance data, obtained by benchmarking the various MPI defined functions.

Chapter 2

Background

2.1 Hardware

Commodity HPC is centred around cluster technologies. A HPC cluster can range from standard desktop machines networked with Gigabit Ethernet to a HPC manufacturers server with Infiniband, Myrinet or other advanced interconnects. To further the study a Symmetric Multiprocessing (SMP) machine provides an alternative HPC architecture to see if performance is sustained across very different interconnects. The available machines are as follows.

2.1.1 Low-grade Cluster *'eScience'*

A 20 node Intel Pentium 4 3.0GHz cluster housed at the eScience department, University of Edinburgh. The interconnect is Gigabit Ethernet, and the nodes run RedHat Enterprise Linux. This is akin to a home made cluster, where multiple desktop machines with single core CPUs and 1GB RAM are simply networked together with Gigabit Ethernet. It is primarily used for simple cluster usage, infrequently, and so components used are relatively inexpensive and are expected to be inefficient.

2.1.2 High-grade Cluster *'Scaliwag'*

An IBM e325 server, 'Scaliwag' is a 32-node dual-core AMD 2.0GHz Opteron64 Cluster at Daresbury Laboratories, Daresbury. The interconnects are Gigabit Ethernet, Infiniband, and SCI / Dolphin. The nodes run SuSE Linux Enterprise Server. The system features larger quantities of RAM, 2GB per node, and is designed for intensive HPC codes and real-world multiprocessor utilisation. This machine is used for research by the UK academic community, and as such has an expensive, high quality interconnect that has undergone testing and tuning for performance, particularly for the Gigabit Ethernet interconnect.
Unfortunately this machine is only available for a period of less than 7 days, quite early on in the life-cycle of the project. A total of eight benchmarking operations are planned, four implementations and two interconnects, which means work is intensive. There is a possibility

that bugs or installation problems could not be resolved in this time, causing a smaller than expected set of results. There is some duplication however, having the low-grade eScience cluster with Gigabit Ethernet available for the whole term of the project.

2.1.3 Shared Memory Machine *'Lomond'*

A SunFire E15K, located at Edinburgh Parallel Computing Centre (EPCC), University of Edinburgh. Comprised of 48 900MHz Sun Ultrasparc III CPUs, running Sun Solaris 9.
While not traditionally associated with free, Open Source MPI Implementations, using a large purpose built shared memory machine provides interest when used with such libraries. More commonly used with vendor supplied or recommended implementations, it is of interest to see if free implementations support fully, or possible even offer comparable performance when used on shared memory machines. Furthermore, with a vendor supplied proprietary implementation, in this instance the Sun MPI library, performance data can be collected to which Open Source implementations can be assessed relative to.

2.2 Benchmarks

Using a freely available benchmark, timings and throughput of a particular implementation can be easily obtained. A popular and long-standing benchmark has peer-approval, and we can be confident as to accuracy while also having data that can be compared in the future with different machines results. Benchmarks can automate and organise tests for the functions defined by the MPI-1 standard, without any knowledge of the actual MPI implementation. This is easily automated to run on varying CPU counts and interconnects, with various sizes and types of data.

2.2.1 Intel MPI Benchmarks

The Intel MPI Benchmarks (IMB), formerly known as Pallas MPI Benchmarks, are a "concise set of benchmarks targeted at measuring the most important MPI functions" [5] reporting timings and throughput data for latency and bandwidth comparisons.
This data is formed in a simple text file, that is consistent across all platforms, and so data can be easily extracted for graphing purposes.
The benchmarks focus on recording timings across all involved processes, with the aim to producing an average recorded time. This is implemented using the following system.

```
for ( i=0; i<N_BARR; i++ ) MPI_Barrier(MY_COMM)
time = MPI_Wtime()
for ( i=0; i<n_sample; i++ )
      execute MPI pattern
time = (MPI_Wtime()-time)/n_sample
```

Here processes are synchronised, timing starts, and the MPI benchmark begins its repetitions of the testing formula. After all the repetitions are completed, the time is divided by the number of repetitions to form a recorded time local to that process. The number of repetitions is initially 1000, but for the largest message sizes is progressively smaller. IMB then reports the maximum and minimum recorded time and calculates the mean time from the result on each process, in order to provide the approximate time of one operation across the system. For runs on large CPU counts, further benchmarks are taken with smaller CPU counts, while other CPUs wait using MPI_Barrier. For example benchmarking 16 CPUs also produces benchmarks for 8, 4 and 2 CPUs, where such a benchmark is meaningful.
By timing the entirety of the repetitions, unfortunately the benchmark does not give statistical error reporting. This is certainly a deficiency, and a more productive benchmark strategy would be to multiple runs of smaller repetition counts. Providing similar data for these results would further analysis, producing minimum and maximum values measured over different runs rather than for different processors. The repetition count would still need to be sufficiently large to avoid producing inaccurate timings and compensate for any lack of granularity with the implementation of MPI_Wtime.

The Intel MPI Benchmarks User Guide [5] organises the benchmarks into three subsets.

Single Transfer Benchmarks

This group benchmarks a single message, sent between two processes. There are two variants in this group

- PingPong
  Using MPI_Send and MPI_Recv, a message is sent from processor A to B, and then a similar message back from B to A. The time reported by the benchmark is half this timing.
- PingPing
  With PingPing, processes A and B simultaneously execute MPI_Isend, a non-blocking send, followed by MPI_Recv, a blocking receive, and MPI_Wait, waiting on the earlier MPI_Isend. The time reported is the entirety of this sequence.

Parallel Transfer Benchmarks

These benchmarks measure performance with all processors executing similar routines concurrently.

- Sendrecv
  With Sendrecv, the processes form a periodic chain, with each individual process sending data to it's right, and receiving from it's left. This is achieved using the MPI_Sendrecv function.

- Exchange
  Similar to Sendrecv, except each process sends data both right and left, and receives from both right and left. This is implemented by the benchmarks as two MPI_Isend instructions, two MPI_Recv and an MPI_Waitall.

Collective Benchmarks

These benchmarks are best suited for analysing the implementation and algorithms used in MPI collective operation, i.e. those where information from all processes is operated upon. While the benchmark suite is very thorough, essentially with a benchmark for each of the MPI-1 standard's defined operations, this investigation focuses on a subset of commonly used functions. Furthermore the speed of each message transferring is not the only concern, the mathematical operations that manipulate the collected data have a small effect. The MPI library may have optimised operations on data types but the core numerical operations are the same, having used the same compilers. The data from collective operations is reported in raw latency timings.

- Reduce
  Benchmarking the MPI_Reduce function, this performs a MPI_SUM operation upon MPI_FLOAT data.
- Reduce_scatter
  "The MPI_Reduce_scatter function firstly performs an element-wise reduction on a vector of elements, then the vector is split into segments which are distributed around the process group." [1]
- Gather
  Benchmarks the MPI_Gather function, where data from each process is collected and concatenated at the root process.
- AllGather
  The MPI_AllGather function is similar to an MPI_Gather operation, however in this instance the result is available on all processes.
- Bcast
  This benchmarks the MPI_Bcast function, broadcasting one complete message to every process.

2.3 MPI Implementations

2.3.1 Open MPI

Open MPI is a new implementation of MPI, attempting to create the best MPI library available. It uses resources and technology from existing or defunct MPI libraries such as FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI, developing an all new implementation influenced by these

codes.
Open MPI aims to provide support for the majority of established high performance interconnects, TCP/IP, shared memory, Myrinet(GM and MX), and Infiniband (MVAPI and OpenIB). [2] In addition the modular component system enables easy extension and addition, one such example is that of Quadrics/Elan4 interconnect support. [4]
Open MPI has only recently begun to have stable versions of the library available.[1] With extensive development ongoing, and ambitious aims to support all common interconnects and full MPI-2 compliance, the performance of Open MPI against established MPI implementations is not expected to be comparable; the refinement process taking second place to functionality.
While in it's infancy, Open MPI has been tested against all our target architectures, and so should pose no real problem to compile and run the benchmarks. As the Open MPI developers do not test with Sun Solaris regularly, it is possible that some difficulty may be had with the Sun machine.

2.3.2 LAM/MPI

Local Area Multicomputer is an MPI implementation (LAM/MPI, LAM) ceased development with the first release of Open MPI, with developers and resources becoming major contributors with the new library. LAM is currently a Indiana University project, having had several bases over it's seventeen year history. [2]
LAM/MPI supports the MPI-2 standard, and multiple HPC platforms and interconnects. Infiniband support however is in it's infancy, only being available since version 7.1. The User Guide comments that such support has only been adequately tested on "Mellanox VAPI thca-linux-3.2-build-024" and that mVAPI versions prior to 3 will definitely not work. [6]

2.3.3 MPICH

Developed by the Argonne National Laboratory, MPICH1 (MPI Chameleon) is a high-performance portable implementation of the MPI standard. The primary goal of MPICH is portability, similar to the chameleon's adaptability.[3] Supporting only MPI-1 MPICH1 development has since halted while an entirely new implementation supporting MPI-1 and MPI-2 is developed.
MPICH2 is a new implementation of MPI, designed to further development and support research into high-performance implementations of the MPI-1 and MPI-2 standards. With no code reuse from MPICH1, the core operation of MPICH2 is significantly different, building on the experience from many years research into MPICH1. Development is ongoing, but at a slower pace than Open MPI, with thoroughly tested releases and only with significant changes. Interconnect support is more limited with MPICH, with neither versions supporting Infiniband, developers instead adapt a particular version of MPICH to add Infiniband support. The MVAPICH project (MPI over Infiniband VAPI based on MPICH)at Ohio State University provides MVAPICH1 and MVAPICH2, which are designed in this manner. [3]

[1] Open MPI 1.0 released in November 2005
[2] LAM's History : http://www.lam-mpi.org/about/devel/
[3] MVAPICH Home : http://nowlab.cse.ohio-state.edu/projects/mpi-iba/

2.3.4 Native or proprietary MPI implementations

With Sun SMP machine a proprietary, native MPI library developed by Sun is already installed. By collecting benchmark data for this in addition to the Open Source implementations, a suitable base reference can be achieved. This marker can aid analysis of the performance, enabling interpretation relative to some theoretical best.

2.4 MPI Function Implementation

For some of the MPI functions being benchmarked analysis of the libraries show differences in implementation, most notably with the choice of algorithm used for complex collective operations. With these complex operations there are ofter different ways to implement, by using varieties of other, simpler MPI functions.
While *Improving the Performance of Collective Operations in MPICH* [7] details the algorithms used by MPICH, there is no such clear document for every implementation; indeed the paper is dated 2003 so it is entirely possible the implementation has changed somewhat. Extensive analysis of the source code of all libraries was therefore undertaken, where comments and actual code have revealed the algorithm and implementation decisions taken by each library.

2.4.1 Single Transfer Benchmarks

Point-to-point functions are implemented directly, without calls to other MPI functions, involving only 2 processes and the direct transfer of data. These implementations are effectively the core transport system of the library, from which the algorithms for more complex operations are built upon.

Parallel Transfer Benchmarks

- Sendrecv
  With the Sendrecv benchmark, the benchmarked operation is the MPI_Sendrecv function. Open MPI & LAM/MPI implement MPI_Sendrecv as MPI_Irecv , MPI_Send, MPI_Wait. MPICH1 & MPICH2 implement this as MPI_Irecv, MPI_Isend, MPI_Waitall. Interestingly the MPICH2 code contains comments noting that

  > Performance for small messages might be better if MPID_Send() were used here instead of MPID_Isend().

Collective Benchmarks

- Reduce
  Open MPI has a basic implementation of MPI_Reduce, where all processes send to the root, and the root performs the reduction. There is however an optimised Shared Memory specific routine that performs the operation in memory without going down to lower level point-to-point MPI operations.
  MPICH1 and MPICH2 use a binomial tree algorithm for small message sizes, and Rabenseifer's Algorithm for message of 2 kilobytes and above. This algorithm improves on tree algorithms by distributing computation across all processes, unlike a tree's subset of processes.[8]
  LAM/MPI uses a wide area optimal associative algorithm for some Shared Memory cases, and a combination of a linear (like Open MPI) and logarithmic (like MPICH's binary tree) algorithm for all other interconnects.

- Reduce_scatter
  The MPI specification[1] for MPI_Reduce_scatter comments on possible implementations.

  > The MPI_REDUCE_SCATTER routine is functionally equivalent to: A MPI_REDUCE operation function with count equal to the sum of recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster.

  For Open MPI this simple, functionally equivalent algorithm of MPI_Reduce followed by MPI_Scatterv has been chosen.
  LAM/MPI has some optimisations for message size, and a choice of algorithms selected based on the operations variables.
  MPICH2 has optimised for commutative and non commutative operations. "All the predened reduction operations in MPI (such as MPI SUM, MPI MAX) are commutative" [7] so for this benchmark a recursive-halving algorithm is used for messages with total data less than 512 kilobytes, while at or above this size a pairwise exchange algorithm is used.

- Gather
  With the MPI_Gather function and Open MPI, the root process receives data while all others send to it. MPICH1 and MPICH2 use a binomial tree to gather information upwards to the root. LAM/MPI has a direct shared memory implementation, and in other instances uses a similar algorithm to Open MPI.

- Allgather Again the MPI specification[1] makes recommendations.

  > Although the discussion of collective communication in terms of point-to-point operation implies that each message is transferred directly from sender to receiver, implementations may use a tree communication pattern. Messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather), if this is more efficient.

  Open MPI implements this as an MPI_Gather followed by an MPI_Broadcast, a solution that is functionally equivalent but simple and inefficient. MPICH1 and MPICH2 are

identical, choosing different algorithms based on function arguments.
Below 32 kilobytes in total, the total bytes calculated by multiplying the datatype count with the datatype size and again with the number of processors, a recursive doubling algorithm is used. On or above 32 kilobytes total, a ring algorithm is used.

LAM/MPI implements this as a MPI_Gather followed by an MPI_Broadcast, with specific shared memory implementation that is essentially the same algorithm - all processes copy into contiguous shared memory, a barrier for all to complete, then all copy from shared memory.

- Bcast
  To implement MPI_Bcast, MPICH1 and MPICH2 use a binomial tree for messages less than 12 kilobytes; a Scatter followed by AllGather for messages of size 12 kilobytes and above. Open MPI uses a basic tree algorithm, with a custom shared memory implementation. LAM/MPI has a similar direct shared memory implementation, and a similar basic tree algorithm as Open MPI for other interconnects.

Chapter 3

Results and Analysis

To ensure comparable results, the same version of each MPI implementation has been used across all platforms. While some implementations have unstable and development releases, such as Open MPI, the latest stable release of each library was used. This ensures a better tested implementation, that would provide a more accurate picture of that implementations ability. These versions were:

- MPICH1 1.2.7p1, Released in June, 2005
- MPICH2 1.0.3, Released in November, 2005
- Open MPI 1.0.2, Released in April, 2006
- LAM/MPI 7.1.2, Released in March, 2006

In all instances the libraries were compiled from source. Compilers vary across platforms, however on each platform where possible the same compiler has been used for all implementations. Different compilers upon different architectures should not affect the ability to compare performance across machines, due to the relatively small amount of computation that the library actually performs. Most of the time will be spent in Operating System (OS) system calls, where the compiler has little impact.
These were coupled with the latest release of the Intel MPI Benchmarks, version 2.3. This was compiled from source on all platforms, using the compilers and MPI implementations as above.

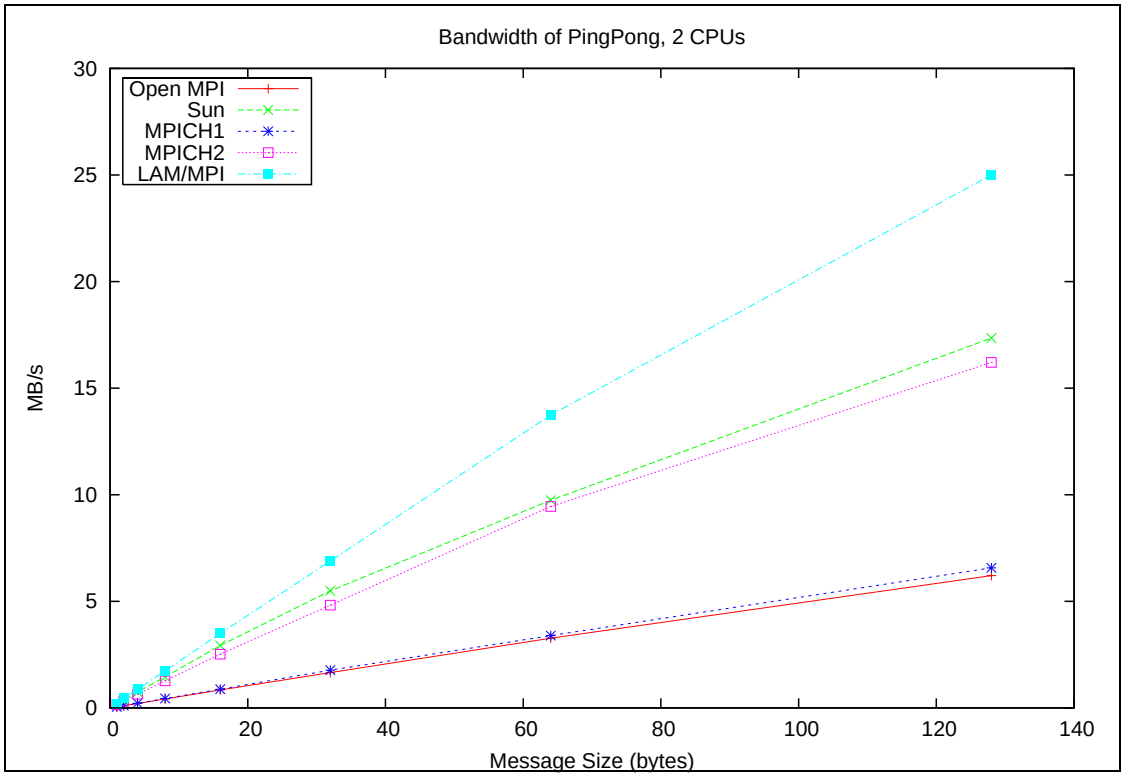

Figure 3.1: Bandwidth of PingPong on Lomond, up to 128 bytes

3.1 *Lomond*

Figure 3.1 of the benchmark's PingPong routine shows the single message throughput results. Focusing on small message sizes, three levels of performance are apparent. The superseded MPICH1 and the newly developed Open MPI have initially very similar rates, slow in comparison with the next band of MPICH2 and the native Sun implementation. LAM/MPI however outperforms everything, and has more than four times greater throughput than Open MPI with a 128-byte message.

Increasing the message size causes a cessation of the linear bandwidth increases, illustrated in Figure 3.1; instead several curves become evident. Figure 3.2 shows how Open MPI employs a single algorithm for all message sizes, illustrated by the gradual continuous curve. MPICH1 is similar, although performance of it's single algorithm is an improvement on Open-MPI. MPICH1's smooth curve eventually meets that of LAM/MPI, which is no longer the best performer after the algorithm changes dramatically when the message size reaches 10KB. Sun MPI's steady curve results in the best performance after 10KB; at this size MPICH2's performance fluctuates due to a change of algorithm before it's performance rejoins that of Sun MPI.

Eventually with large message sizes, as shown by Figure 3.3, the performance forms two bands. MPICH1, Open MPI and LAM/MPI make up the poorer performers, with bandwidth of approx. 400 MB/s. At the higher tier however, Sun MPI performs around 550 MB/s and MPICH2 proves itself superior at large message sizes with throughput of around 600 MB/s.

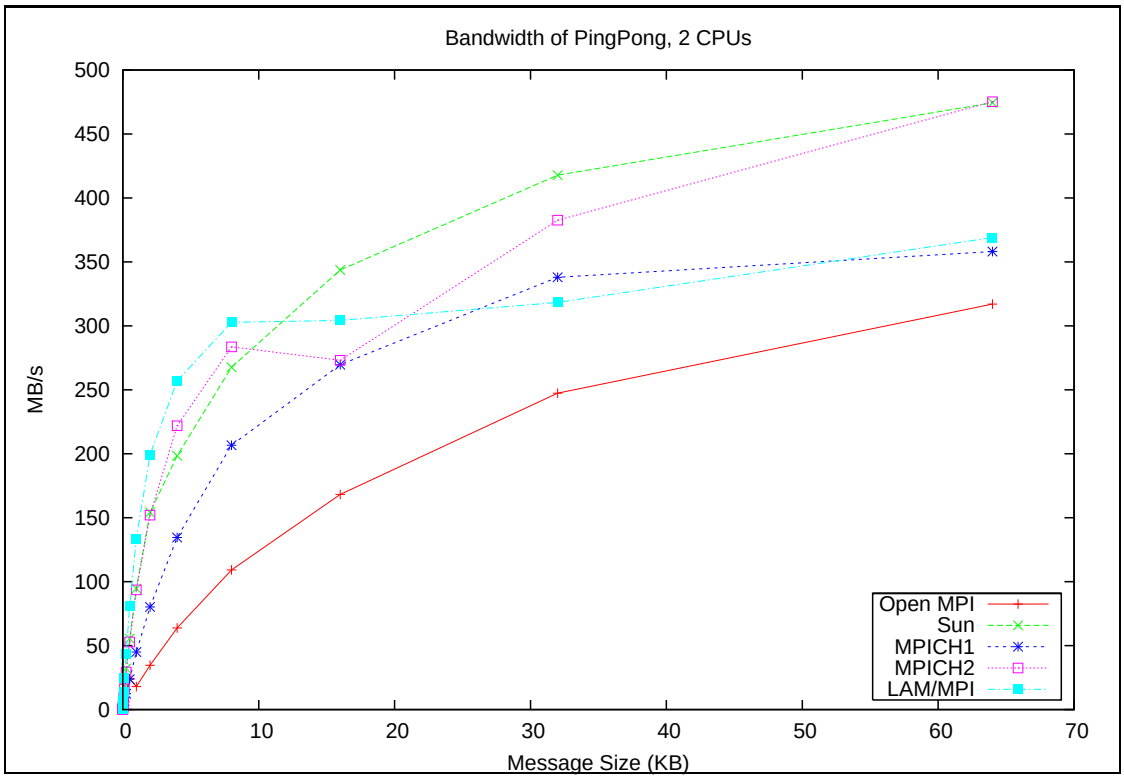

Figure 3.2: Bandwidth of PingPong on Lomond, up to 64 KB

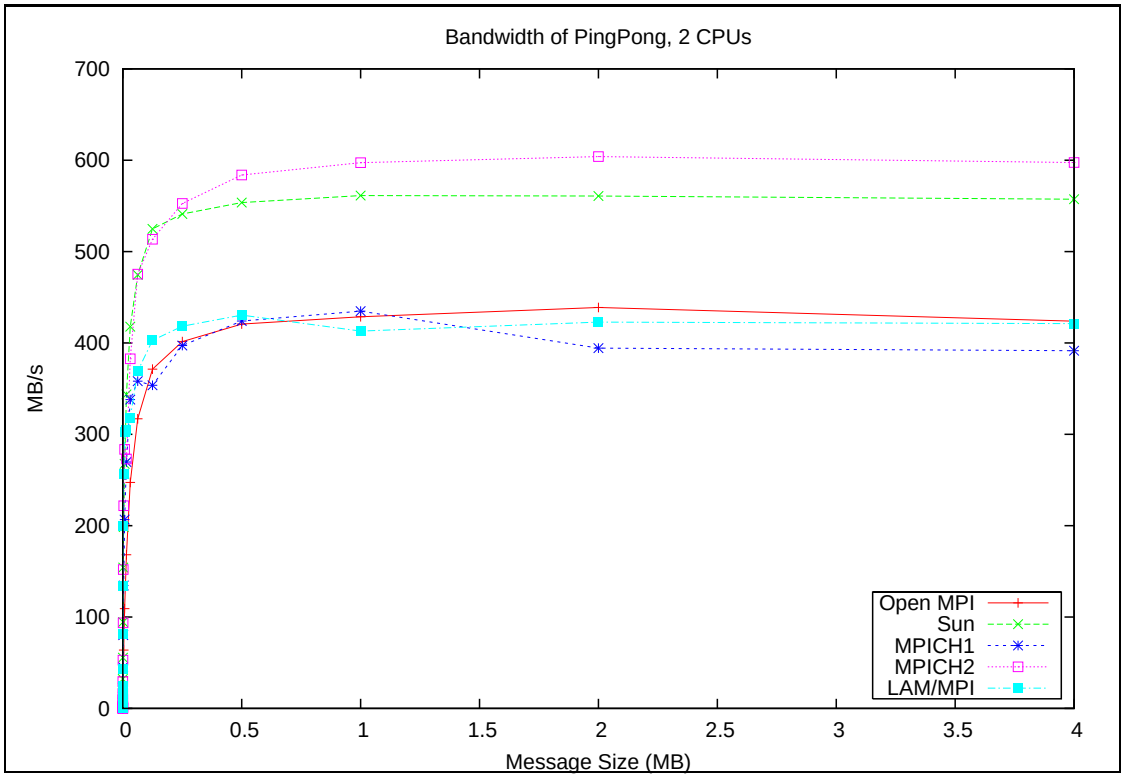

Figure 3.3: Bandwidth of PingPong on Lomond

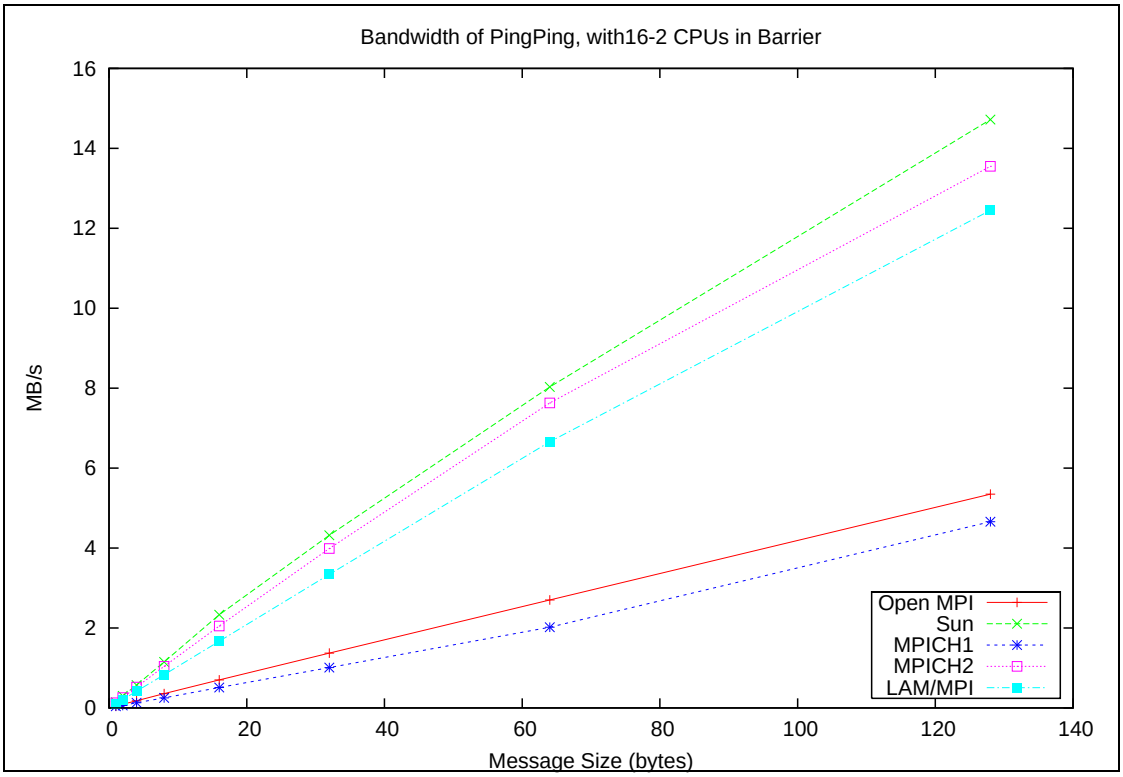

Figure 3.4: Bandwidth of PingPing on Lomond, up to 128 bytes

To compare this in terms of system theoretical maximum, the *STREAM : Sustainable Memory Bandwidth in High Performance Computers* benchmark was used.[1] When benchmarking Lomond this highly efficient copy routine performed at 750MB/s. The MPI implementations are somewhat off this mark, where the top-end implementations this is in line with a slightly inefficient copy routine, achieving near 80% efficiency. The lower tier results, at 400MB/s seem to low to be as a result of an inefficient routine alone, and are consistent with a bandwidth result from two copy operations rather than the one of the higher performing MPI implementations.

When there are simultaneous messages, using the benchmark PingPing, initial performance with small message sizes is similar to that of PingPong; Figure 3.4 shows a similar linear increase in bandwidth. Again MPICH1 and Open MPI have lower throughput than the others, grouping into a lower tier with MPICH1 slightly the poorest. With these simultaneous non-blocking messages LAM/MPI does not mirror it's success with the blocking communications of PingPong; it performs slightly worse than MPICH2, which is still not as efficient as Sun MPI.

Figure 3.5 illustrates how increasing the message size causes similar results to PingPong. The poorer performing implementations of MPICH1 and Open-MPI continue on a smooth curve, showing no change in algorithm. With the leading bunch changes to the curves occur at similar message sizes as before: on or after a message size of 10KB. Of the three, LAM/MPI's change at 10KB is drastically worse, causing a drop from 250MB/s to 100MB/s and lowering it's performance to below that of all others. Sun MPI also takes a dip, performing worse than the steady implementation of MPICH1 after a 30KB message size. MPICH2 has the only change

[1]STREAM Home : http://www.cs.virginia.edu/stream/

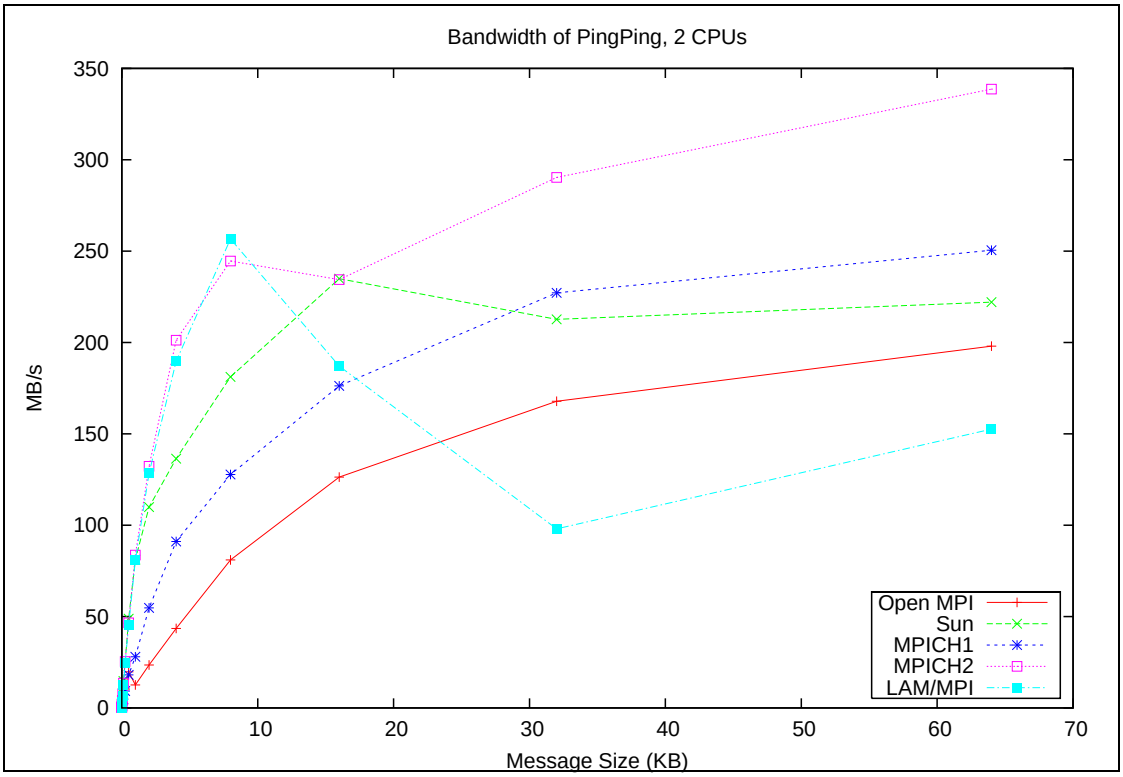

Figure 3.5: Bandwidth of PingPing on Lomond, up to 64 KB

of algorithm that is for the better, it's new curve an exact match for the Sun MPI curve before 10K. This results in MPICH2 outperforming all others.

Figure 3.6 shows the PingPing benchmark with larger message sizes. LAM/MPI recovers slightly from it's drop in performance, but MPICH2 still outperforming all others. Sun MPI is second to last with messages over 1 megabyte, only managing to improve on MPICH1's result. MPICH1 suffers a large downturn at 1MB message size, eventually achieving throughput of 150MB/s - half that of MPICH2.

When analysing the parallel transfer benchmarks there is a similar division of performance, with MPICH2, LAM/MPI and Sun MPI all comparable, with LAM/MPI slightly worse, as Figure 3.7 shows. In this example of MPI_Sendrecv a significant gap exists between these implementations and MPICH1 and Open MPI, with their much lower throughput values.

Collective benchmarks further illustrate the emerging pattern than MPICH1 and Open MPI are not up to scratch. Figure 3.8 shows the latency of MPI_Bcast operations, with Sun MPI showing a very flat curve - the algorithm and base implementation working well as more CPUs are used. LAM/MPI has a similar shape, but a near constant 3 microsecond increase. MPICH2 is vastly superior to MPICH1, in this instance having the same shape graph but approximately 15 microseconds slower across all processor counts. This is indicative of a similar algorithm but with a increase in efficiency of the underlying core of the library.

Sun MPI is not always the superior platform. With Figure 3.9 the choices of algorithm for each implementation's MPI_Reduce_scatter operation is clear to see. MPICH2, with many different

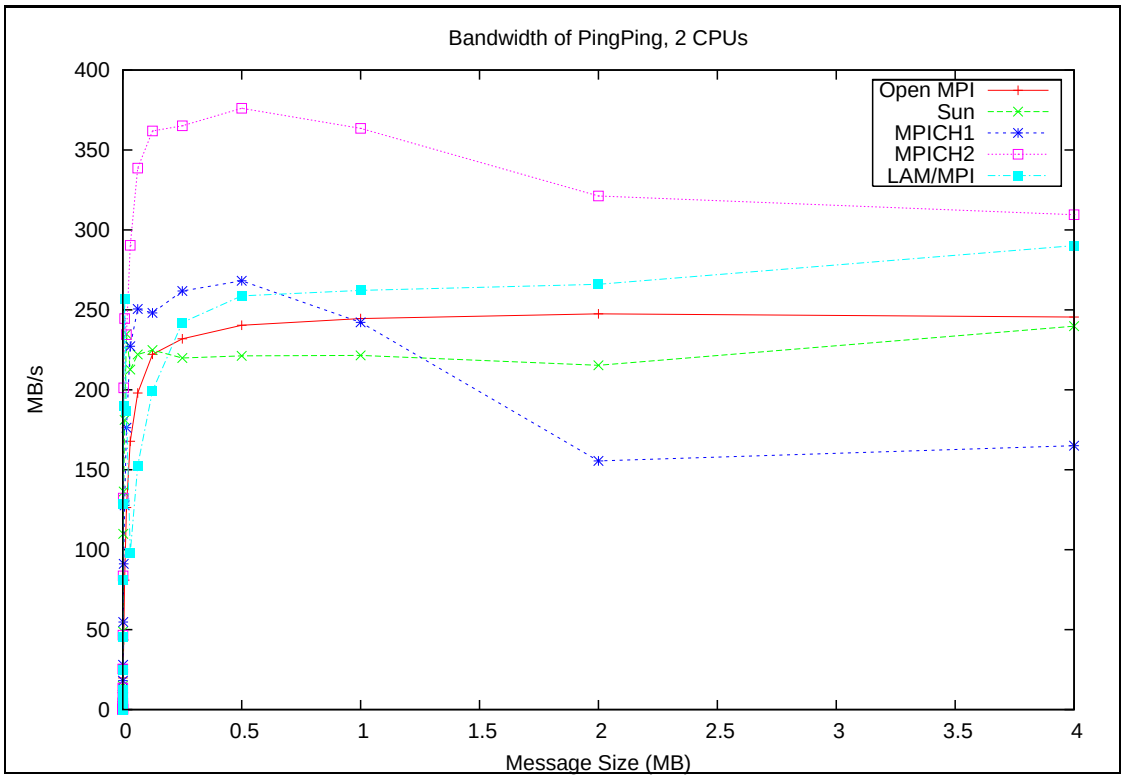

Figure 3.6: Bandwidth of PingPing on Lomond, up to 4 MB

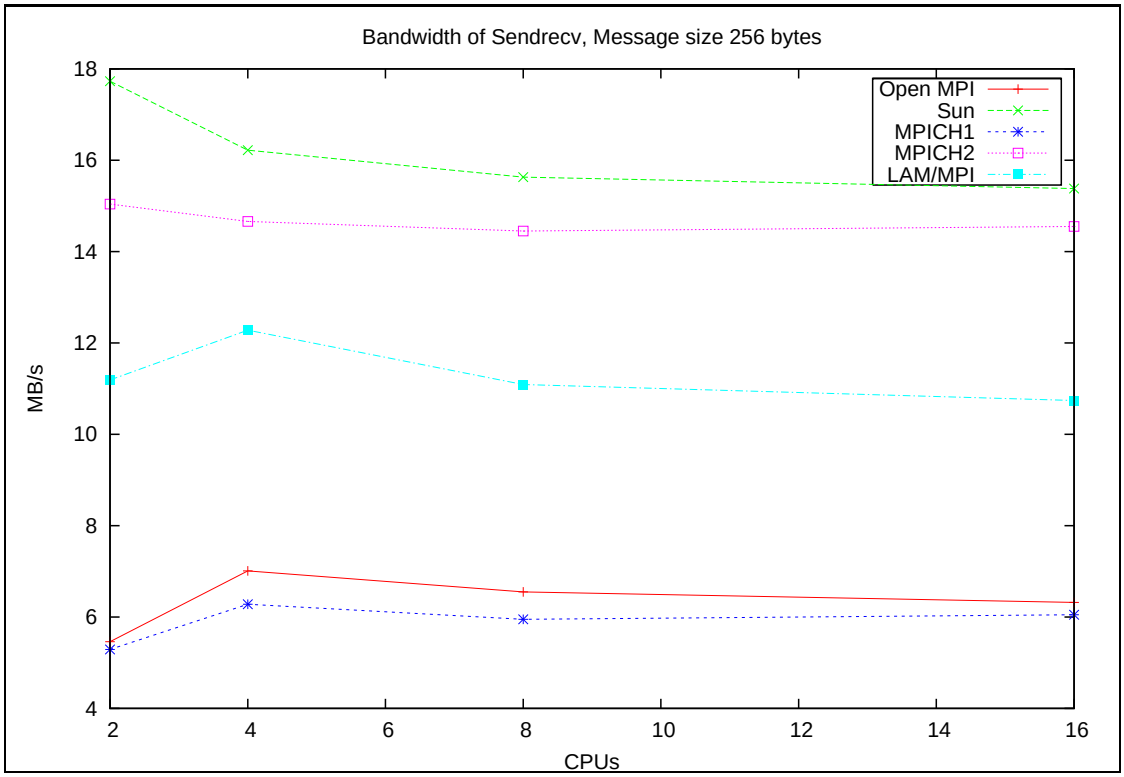

Figure 3.7: Bandwidth of Sendrecv on Lomond

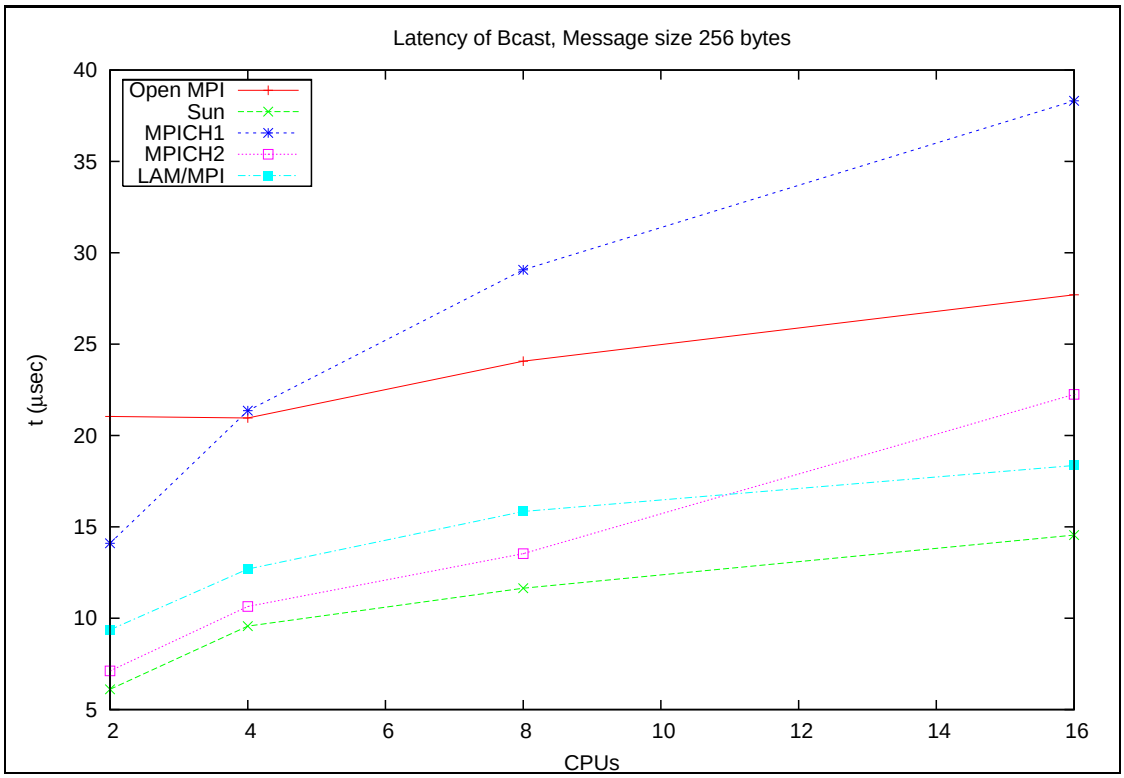

Figure 3.8: Latency of Bcast on Lomond

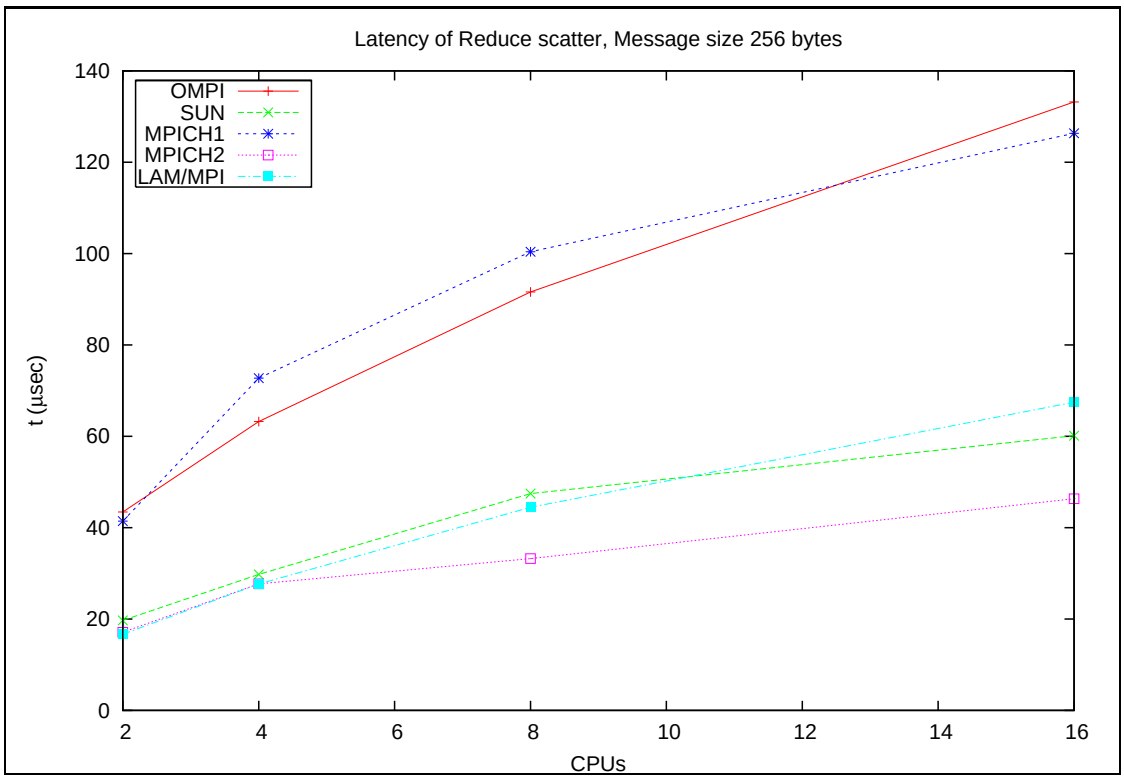

Figure 3.9: Latency of Reduce_scatter on Lomond

algorithms and optimisations performs best; Sun MPI is next, with it's proprietary closed-source implementation. LAM/MPI is a half-way house between simple and complex, with few optimisations, but still a significant improvement on the basic implementation of MPICH1 and Open-MPI which perform poorly.

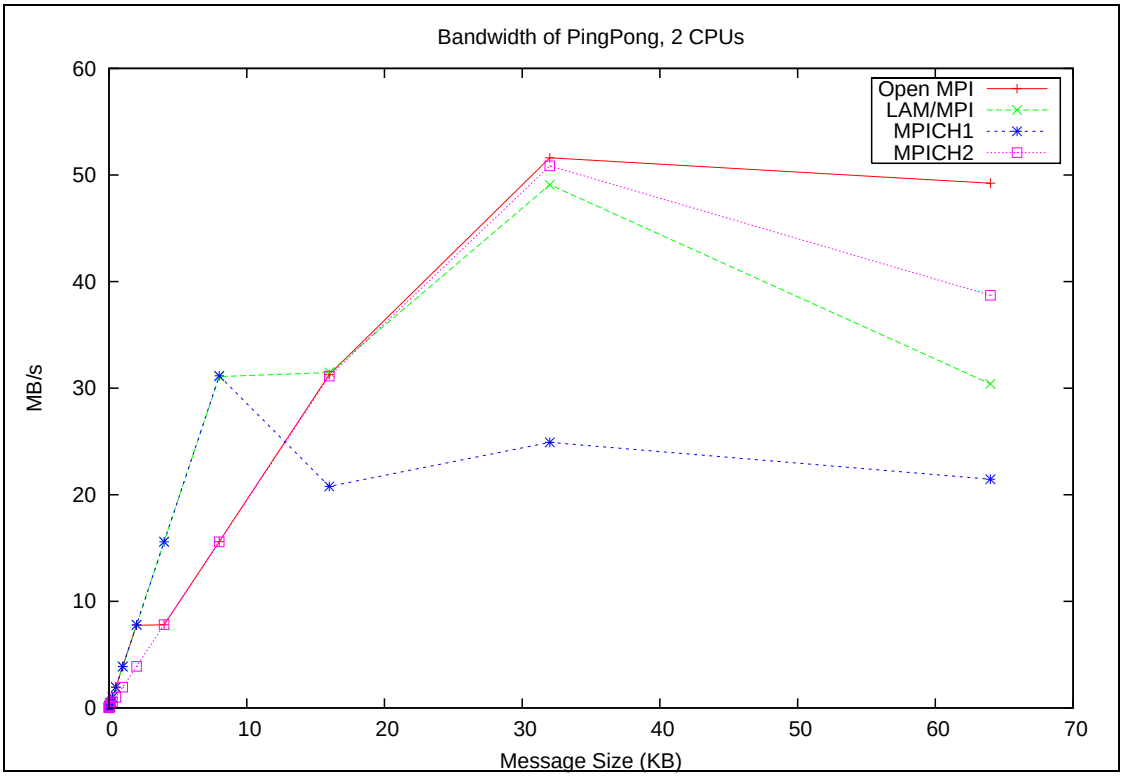

Figure 3.10: Bandwidth of PingPong on eScience, up to 64 KB

3.2 *eScience*

With Gigabit Ethernet, and it's relatively low bandwidth and high latency, there is a possibility that the performance of the interconnect itself will be poor. This gives concern that differences in implementation and the associated performance differences could be masked by the interconnect's performance, as the hardware struggles with the intensive benchmarks. However Gigabit Ethernet is the most accessible and cheapest interconnect that can be feasibly used for HPC, so analysis of MPI implementations is worthwhile. The low cost hardware of *eScience* will give a good indication of performance a the lower end of the HPC market.

The point-to-point bandwidth for small message sizes, benchmarked using PingPong, is displayed in Figure 3.10. Here a difference in implementation is apparent, with MPICH1 plateauing early at only 15MB/s. The other implementations perform better, with similar rates of bandwidth increase.

With the larger message sizes of Figure 3.11 the maximum throughput of each implementation is realised. While MPICH1 remains low at around 20MB/s, LAM/MPI and MPICH2 have increased their bandwidth to similar levels around 45MB/s. Open MPI has achieved a maximum throughput of more than double this, at around 95MB/s. Considering the theoretical maximum for Gigabit Ethernet is 125MB/s, this is very encouraging.

With simultaneous messaging, displayed in Figure 3.12, a similar trend is apparent. MPICH1 again plateaus early, in this instance at around 15MB/s; others outperform it, their bandwidth

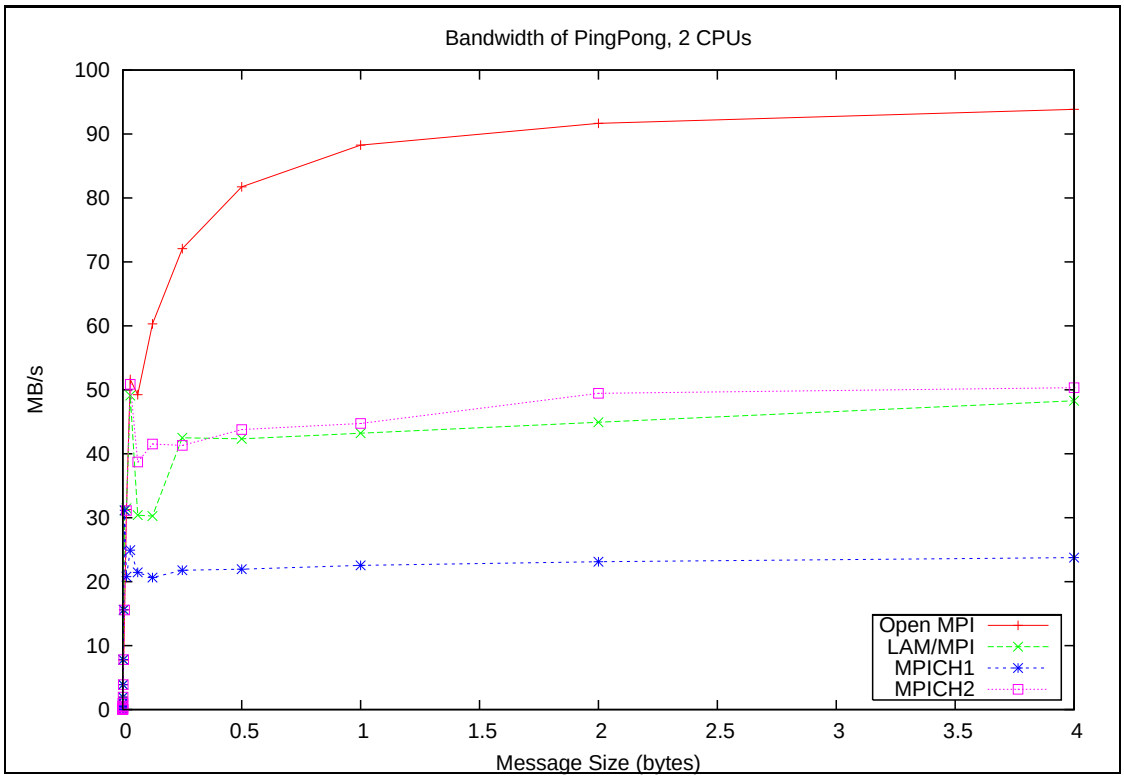

Figure 3.11: Bandwidth of PingPong on eScience

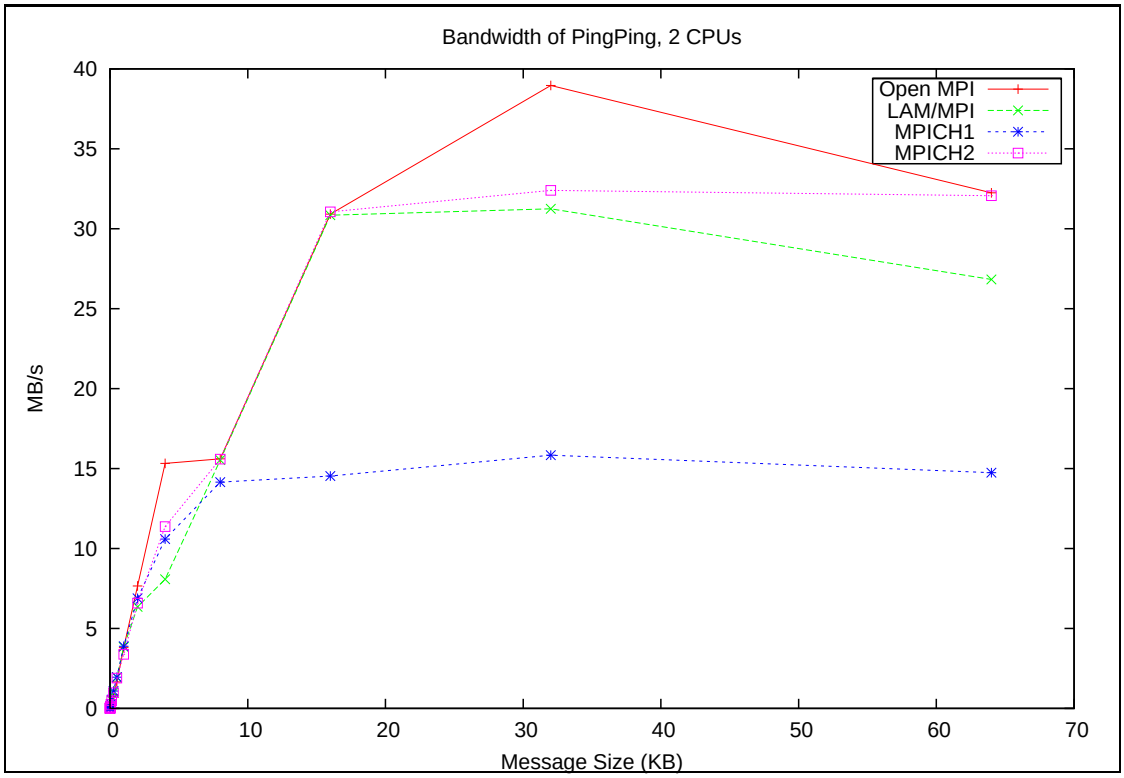

Figure 3.12: Bandwidth of PingPing on eScience, up to 64 KB

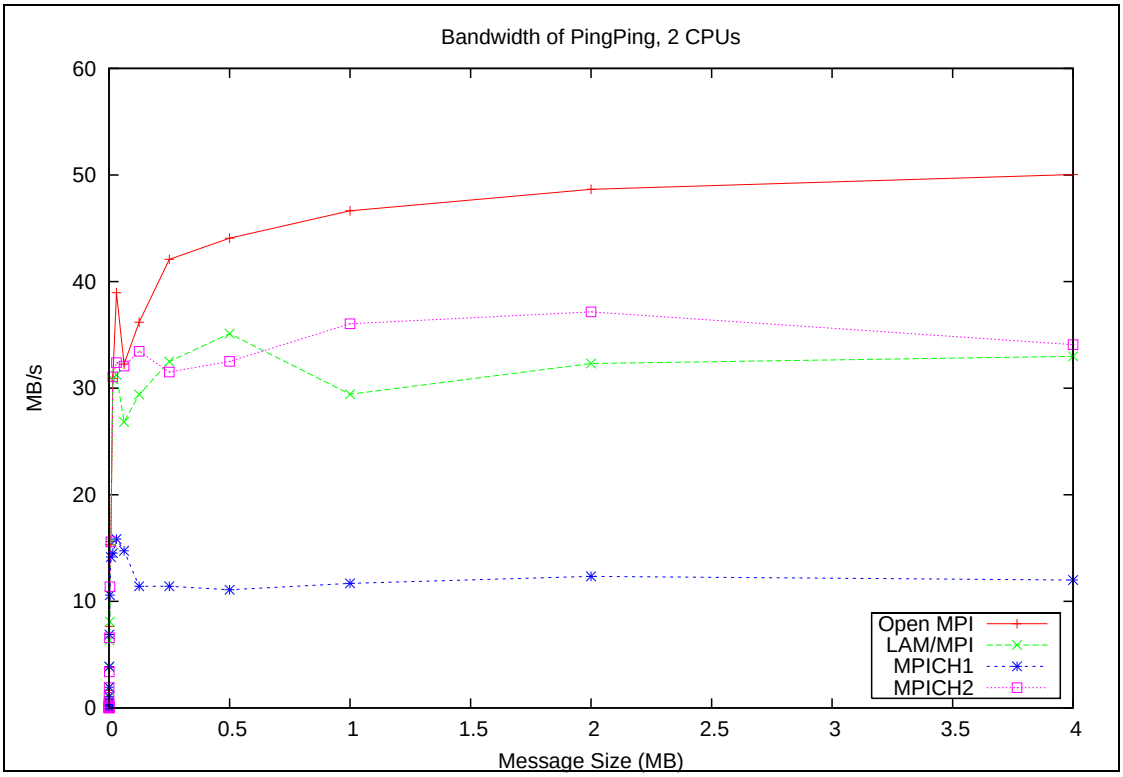

Figure 3.13: Bandwidth of PingPing on eScience

rate rising in unison as message size increases. With these small messages Open MPI is at points identical to MPICH2, while LAM/MPI performs slightly worse.

The larger message sizes illustrated in Figure 3.13 show that the similarity of MPICH2 and Open MPI was short lived, with Open MPI rising to 50MB/s. MPICH2 performance dips at 500KB, a similar message size to where LAM/MPI peaks, showing changes in implementation that reduce and improve respective performances. Overall LAM/MPI is comparable to MPICH2, and MPICH1 shows no marked change - continuing with it's earlier 15MB/s ceiling.

Parallel transfer benchmarking results in erratic performance, as shown by MPI_Sendrecv performance in Figure 3.14. With bandwidth between 300 and 600 KB/s in all instances, this cap in performance is most likely as a result of the Gigabit Ethernet and the associated switch. The interconnect, especially the switch, now has to work even harder than the 2 messages of PingPing, with all processes sending and receiving messages.

With collective benchmarks, there is no clear pattern either. All collective operations have a minimum latency of 200 microseconds, and there is no clear performance winner from the data. Multiple messages per processor and complex patterns of messages increase demand on the interconnect, which was struggling even with simpler and less frequent messages. This further shows that the low-quality hardware impinges massively on MPI benchmarks, becoming a far greater factor than the individual implementations.

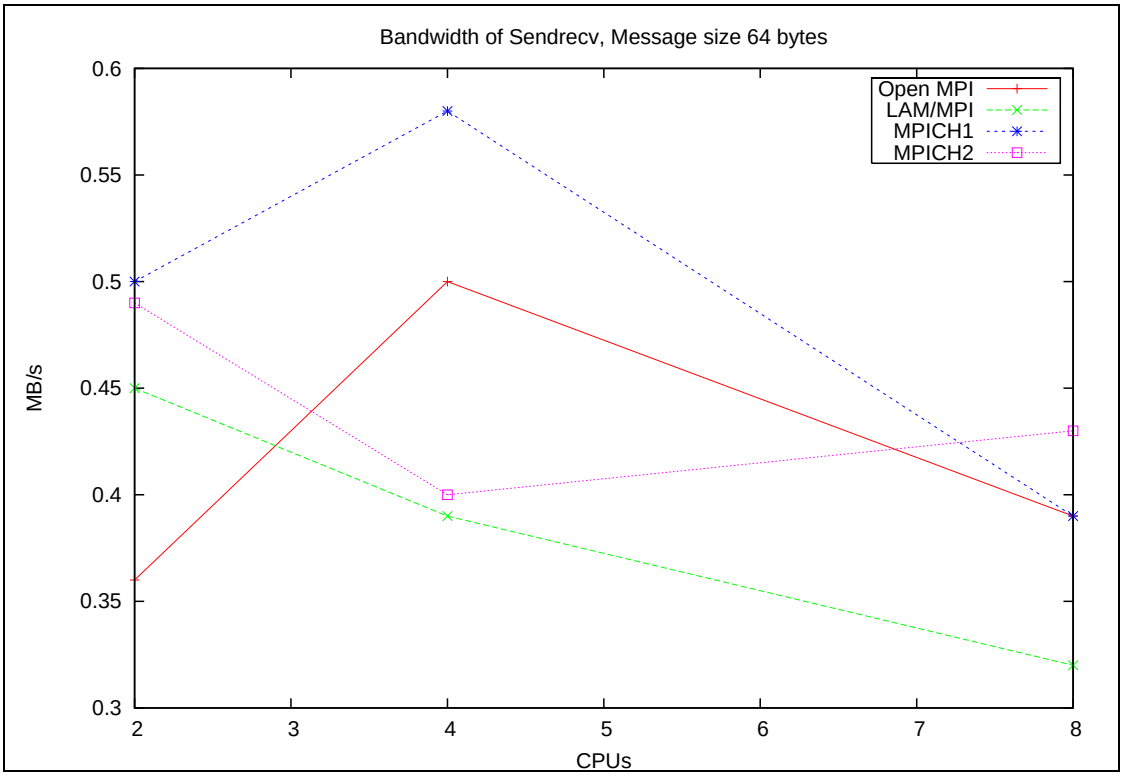

Figure 3.14: Bandwidth of Sendrecv on eScience

3.3 *Scaliwag*

The high-grade cluster 'Scaliwag' initially proved quite a challenge to install some MPI implementations. Access to the machine was unfortunately restricted to just one week, with the unfortunate consequence that results from all implementations across both Gigabit Ethernet and Infiniband were not available in the short timescale.
LAM/MPI proved to be the most temperamental, functioning neither with Gigabit Ethernet nor with Infiniband interconnects. Open MPI, despite having both Infiniband and Gigabit Ethernet support selectable at run-time, did not run as expected. Unfortunately in this environment, only Infiniband ran as planned. MPICH1 and MPICH2 do not natively support Infiniband, however both functioned correctly with Gigabit Ethernet. MVAPICH1 and MVAPICH2, the Infiniband compatible forks of both MPICH development branches were installed without fault, although unfortunately they did not function and debugging could not be undertaken due to the time constraints.
The single message bandwidth of the various implementations shows clearly the performance gain from Infiniband, with a three times speed gain with single messages of Figure 3.15.

MPICH2 outperforms MPICH1 by around 10 MB/s with large message sizes, but it takes a message greater than 30KB before any difference is noted. Both perform well, with the raw maximum for Gigabit Ethernet 125MB/s, the values of 100 MB/s for MPICH1 and 110 MB/s for MPICH2 with message sizes over 250KB are very close to the maximum possible.

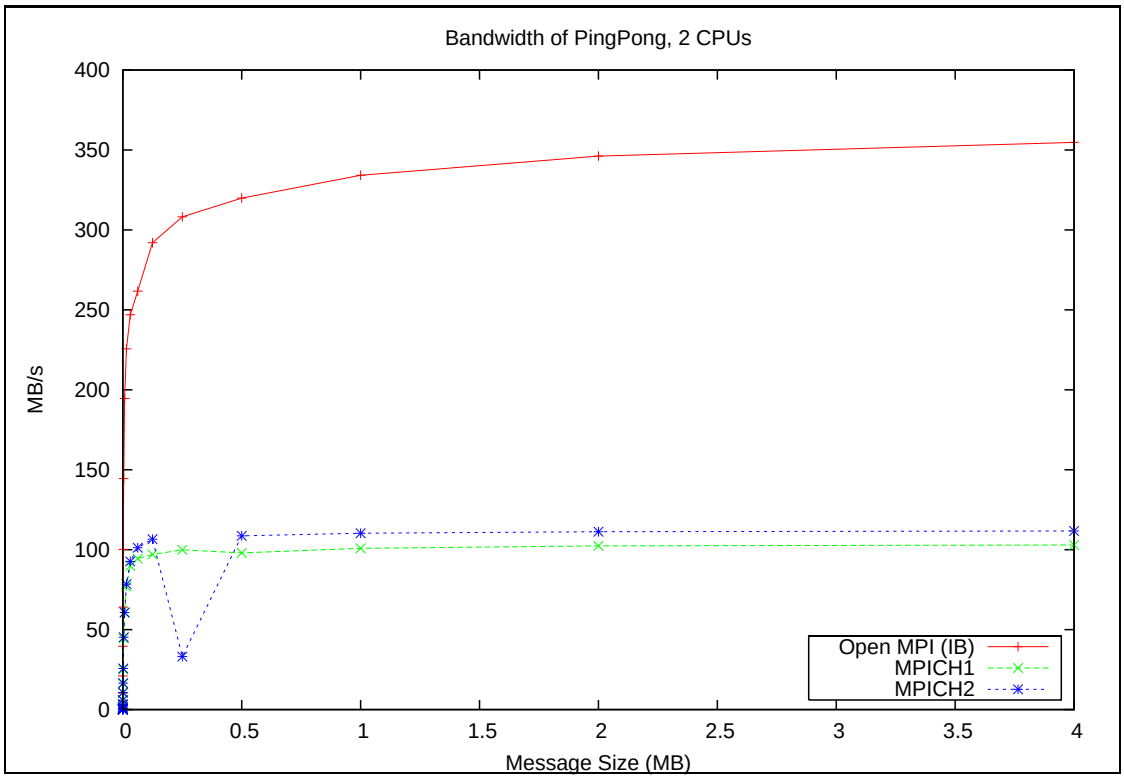

Figure 3.15: Bandwidth of PingPong on Scaliwag

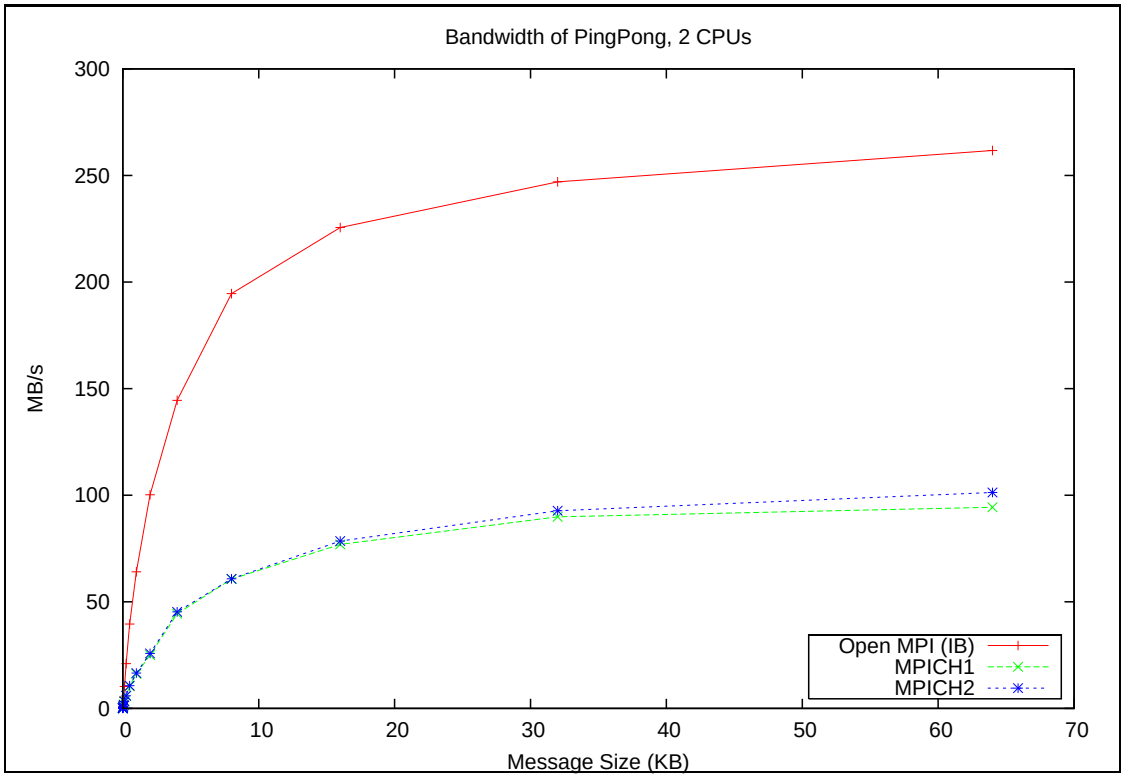

Figure 3.16: Bandwidth of PingPong on Scaliwag, up to 64 KB

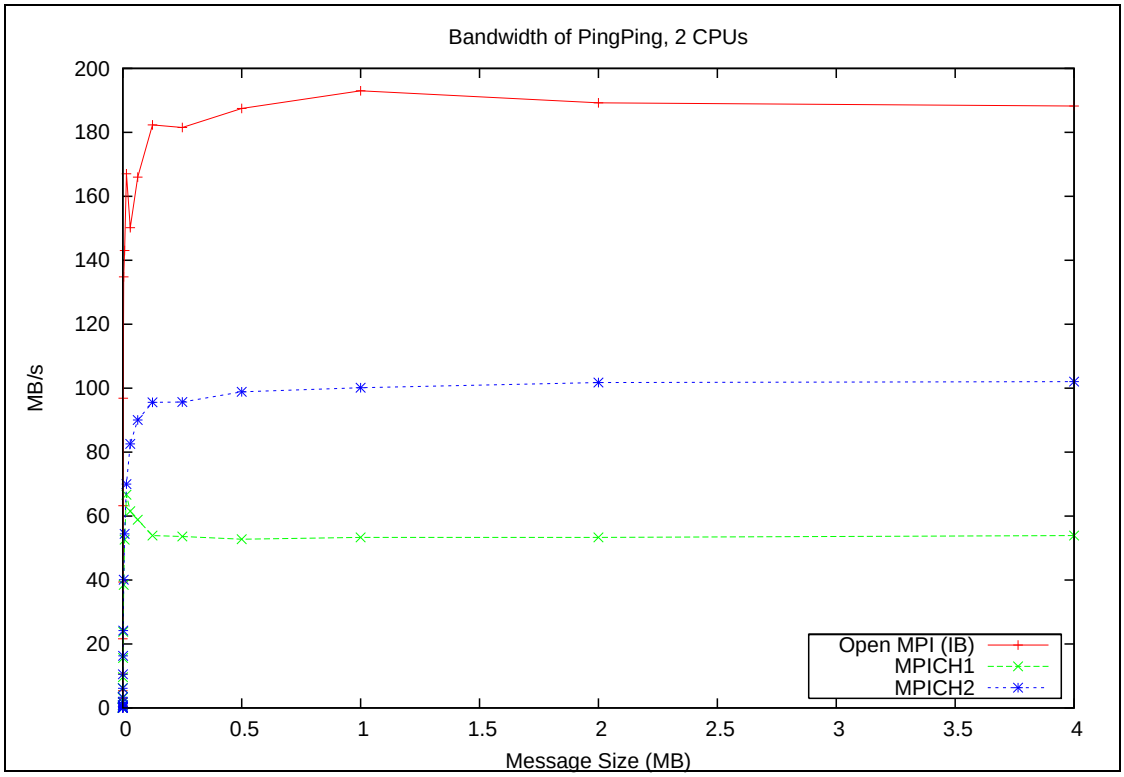

Figure 3.17: Bandwidth of PingPing on Scaliwag

The raw data max for 1X DDR Infiniband is 4 Gbit/s - 500 MB/s.[2] Open MPI achieves a 350 MB/s maximum, but importantly at small message sizes, shown in Figure 3.16 throughput is much greater than the Gigabit Ethernet implementations.

With the simultaneous communication of PingPong, the divide between MPICH implementations is more distinct. Figure 3.18 shows that after message sizes of 20 KB/s MPICH1 is no longer comparable with MPICH2, it's throughput decreasing. Figure 3.17 shows MPICH1's bandwidth from large messages stabilising at around 55MB/s, whereas MPICH2 reaches rates of 100 MB/s, only 10 MB/s less than with single communication. Open MPI again has very high bandwidth even when message size is low, and for large messages bandwidth peaks at 195MB/s. Comparing this relatively, simultaneous messaging causes only a 10% reduction in efficiency with MPICH2 but a 50% reduction for Open MPI, an indication that the Infiniband drivers within Open MPI may not yet be mature.
The full-duplex interconnect is only fully exploited by MPICH2, with both MPICH1 and Open MPI bandwidth results peaking at less than half of their single point-to-point results of PingPong. This shows the efficiency of message handling code by the various implementations. MPICH2 has clearly improved upon MPICH1, whose poor message handling routines are not evident with single messages due to the high specifications of CPU, memory and interconnect.

Benchmarking the parallel transfer operations also shows MPICH2 to be an improvement upon MPICH1. MPI_Sendrecv results graphed by Figure 3.19 illustrate a 20% improvement of

[2]Infiniband SDR, DDR, and QDR White Paper :
http://www.cisco.com/en/US/products/ps6421/products_white_paper0900aecd804c324e.shtml

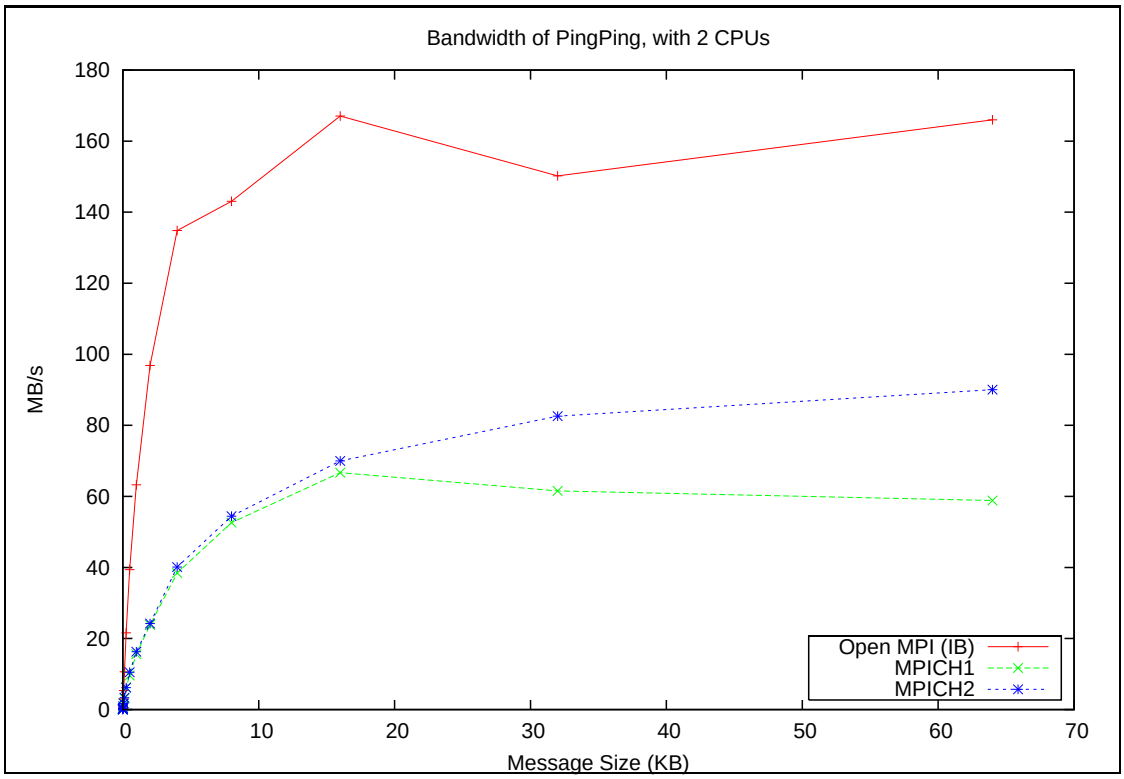

Figure 3.18: Bandwidth of PingPing on Scaliwag, up to 64 KB

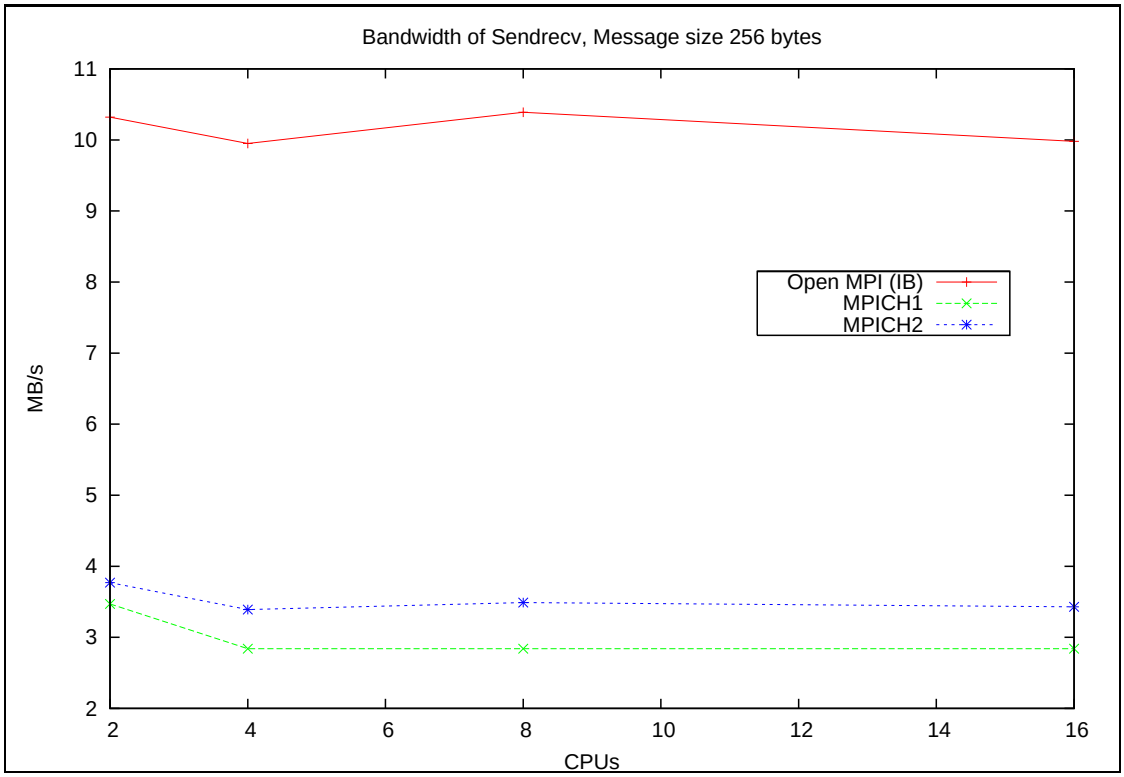

Figure 3.19: Bandwidth of Sendrecv on Scaliwag

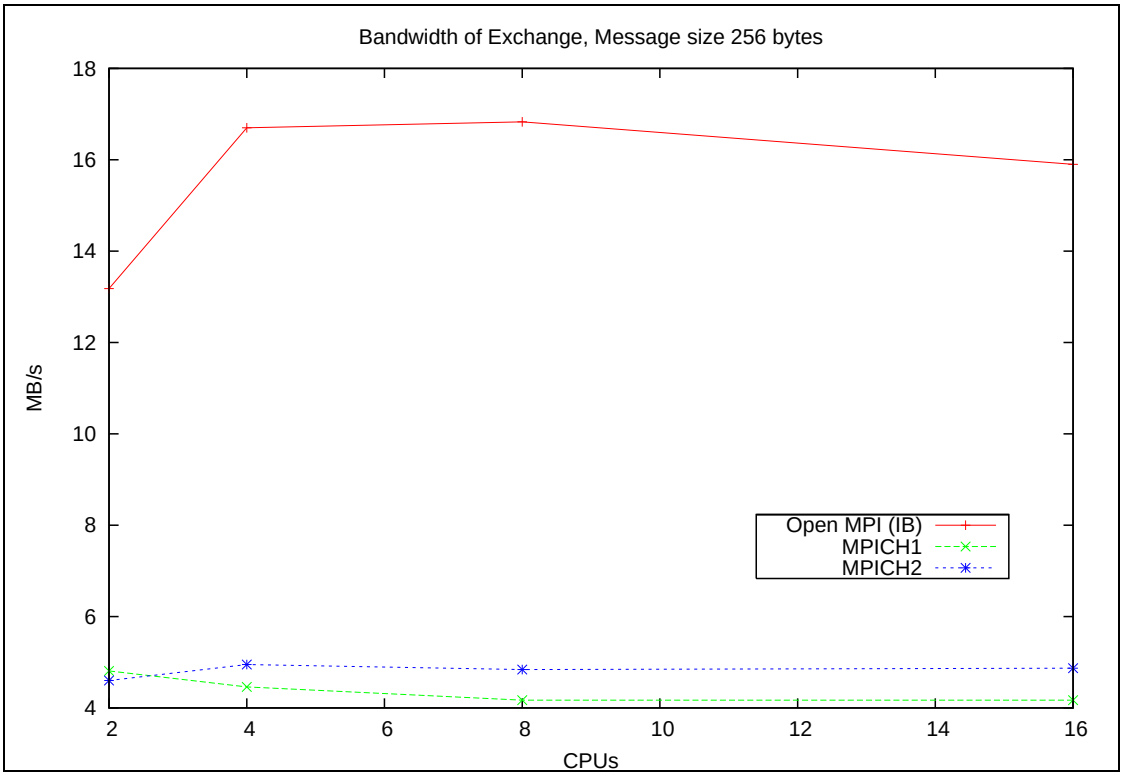

Figure 3.20: Bandwidth of Exchange on Scaliwag

MPICH2 over MPICH1. Bandwidth does not decrease as CPUs are added, a sign that the switch is functioning well. Benchmarking of Exchange in Figure 3.20 cements this, this time with MPICH2 outperforming MPICH1 by a much smaller margin.

With collective operations the difference in performance is further accentuated; the poorer performance of point-to-point operations of MPICH1 augmented by it's basic choice of algorithms for collective operations.

These collective MPI operations predominantly illustrate the same pattern. The Reduce_scatter benchmark of Figure 3.21 shows the norm, with Open MPI and it's superior interconnect having the lowest latency, usually a straight line. The straight line indicates the single algorithm used for the collective operations by Open MPI, but also that message size does not cause a change in efficiency. This is not the case for MPICH, where MPICH2 and MPICH1 are similar for small message sizes, in this case less than 5KB. At a point the poor handling of large message sizes by MPICH1 becomes more apparent, it's latency increasing in excess of that of MPICH2, despite having the same algorithm for the collective operation.

One interesting benchmark is that of Allgather, shown with small message sizes in Figure 3.22. While Open MPI has a single algorithm, resulting in a straight-line graph, the two versions of MPICH change algorithm at 32 kilobytes. This improved algorithm causes a dip in the latency of these implementations; this is sufficient to make them more efficient that Open MPI, despite being on Gigabit Ethernet compared to Infiniband.
As message size increases, in Figure 3.23, the new algorithm proves to be stable and continually below that of Open MPI. With the MPICH2 implementation, latencies are 50% of that of

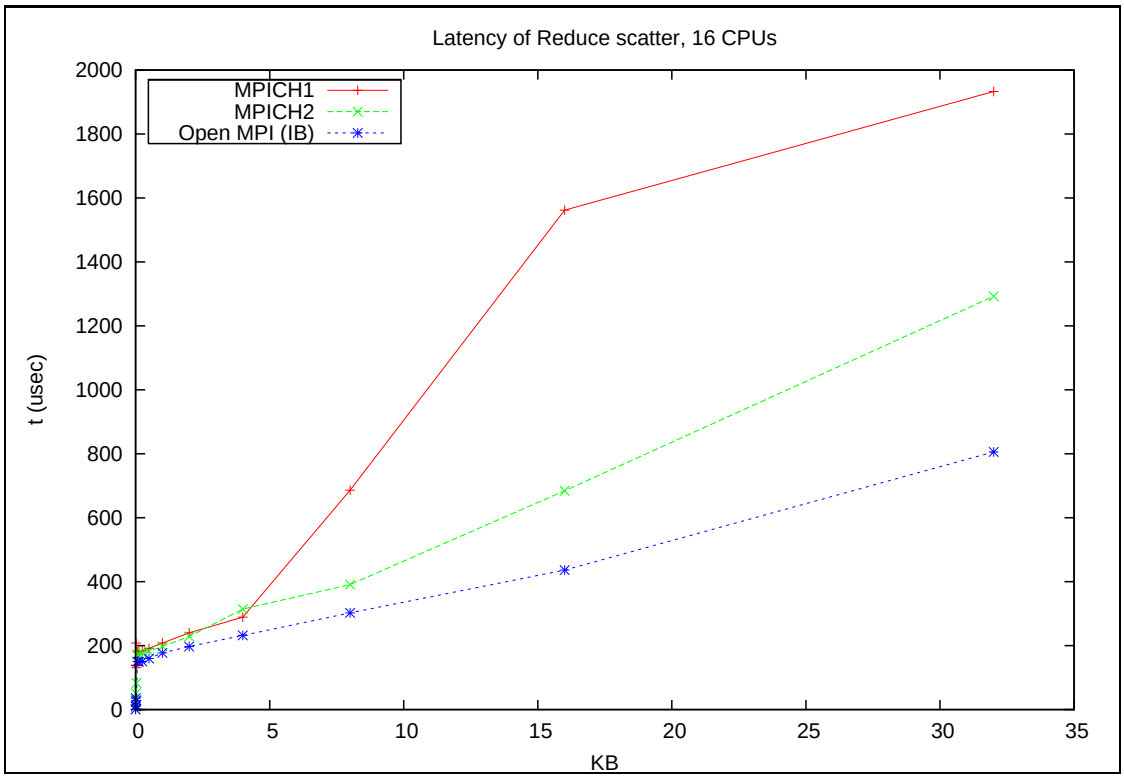

Figure 3.21: Latency of Reduce_scatter on Scaliwag, up to 32 KB

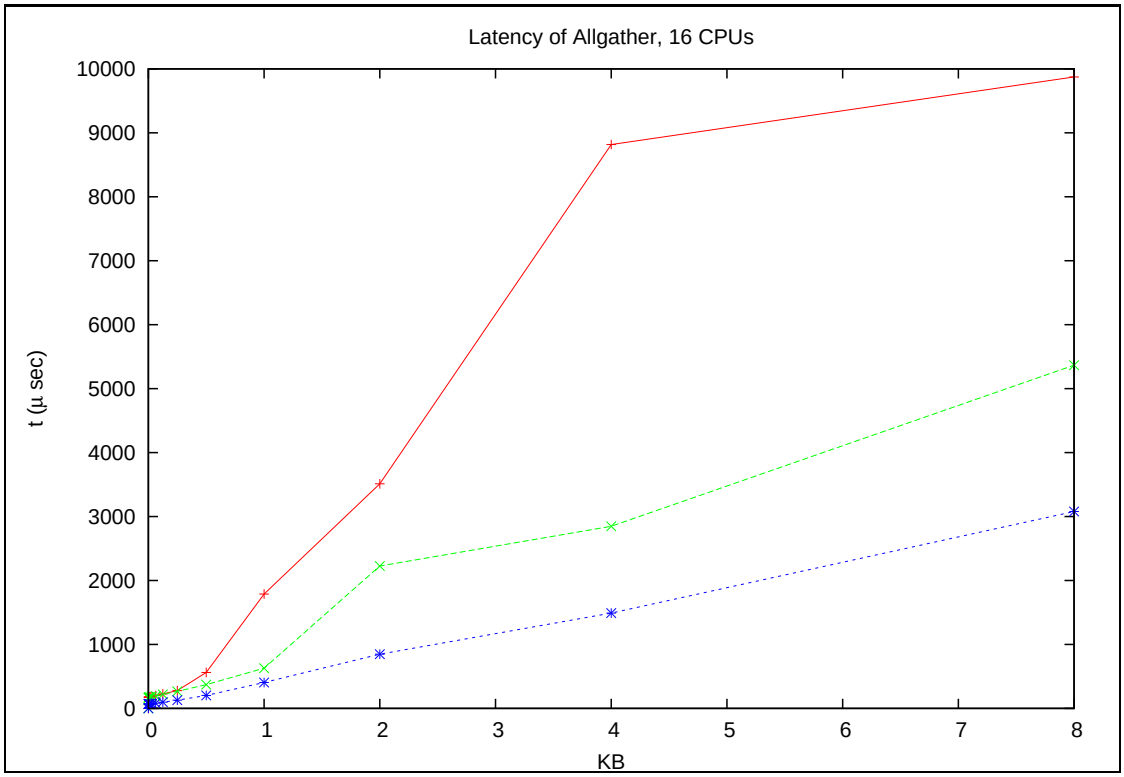

Figure 3.22: Latency of Allgather on Scaliwag, up to 8 KB

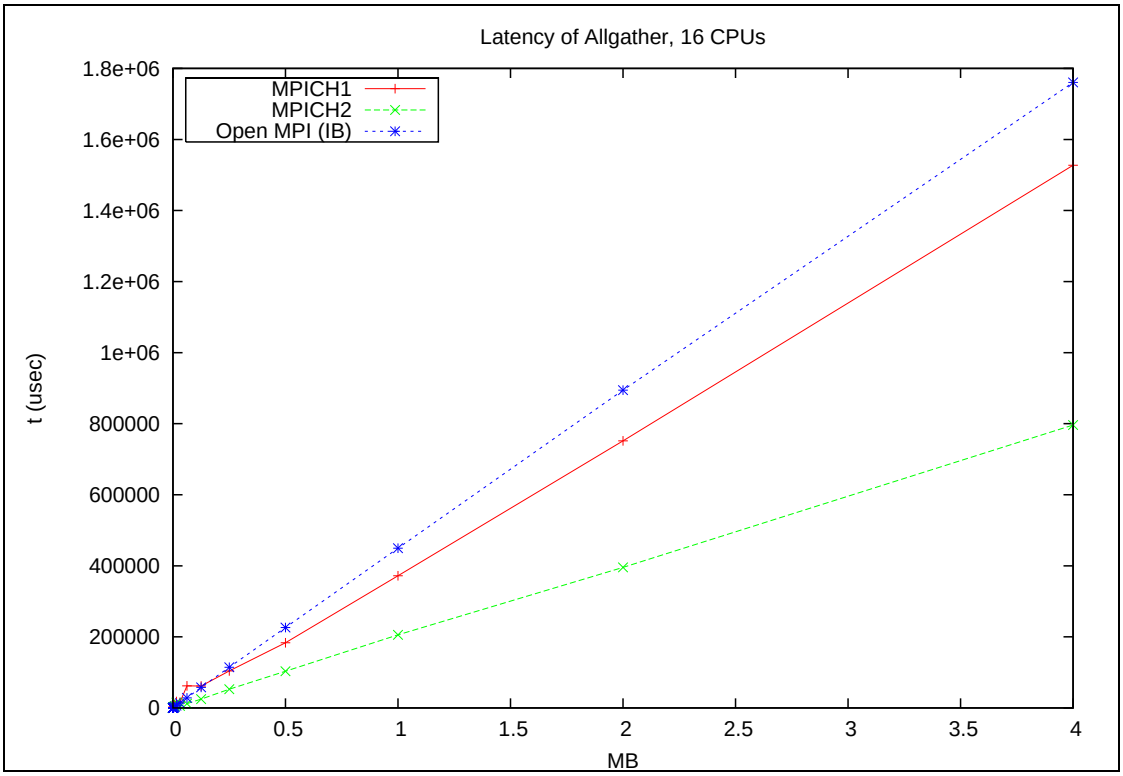

Figure 3.23: Latency of Allgather on Scaliwag

Infiniband, despite point-to-point messages being over three times as fast with Open MPI.

Chapter 4

Conclusions

The easiest conclusion to make is are free MPI implementations are a false economy, i.e. the time taken to install and maintain a library could exceed that of purchasing a library with user support. Overall installation worked well, particularly so with both versions of MPICH. With MPICH1 and MPICH2 installation was without hiccup, apart from a small issue with preserving permissions when installing MPICH2 on the *eScience* cluster; this required minor Makefile modification. This was the case across both clusters and the shared memory machine, and was the only implementation to successfully install across all supported interconnects. It is of significance, however, that the supported interconnects were less than the other libraries, notably Infiniband is not supported directly by either MPICH.
For Open MPI installation issues were small, and was the only MPI implementation to utilise Infiniband on the *Scaliwag* cluster. The attempt to have a single Open MPI installation with both TCP, for Gigabit Ethernet, and Infiniband modules was probably the reason for the TCP modules failure in this instance. Unfortunately the short time span of access to the *Scaliwag* machine could not permit attempts at further installations. Having a working TCP over Gigabit Ethernet installation using a different Enterprise Linux cluster, eScience, supports this conclusion. Open MPI not only claimed to support all of this investigations available interconnects, but was also the only implementation tested to do so.
LAM/MPI also claimed full support for the available interconnects, but the specific Infiniband support mentioned earlier proved not to function with the high-grade cluster. Unfortunately with *Scaliwag* the Gigabit Ethernet support could also not be operated, the combination of short access time, and difficulties using LAM with the batch submission system preventing function. With success on a non-batch system cluster using Gigabit Ethernet, *eScience*, I would expect LAM/MPI to operate successfully on *Scaliwag* given further time and manipulation.

4.1 Open MPI

When assessing Open MPI's performance consideration of it's relatively short history is worthwhile. This new implementation has big design goals from the outset with full MPI-2 implementation and multiple interconnect support.
In particular Open MPI's intended support for Infiniband interconnects, through both OpenIB

and the Mellanox Verbs API, is distinct from other Open Source implementations. Having this functionality integral to the library sets it apart from the best-performing rival, MPICH2, whose Infiniband support is in the style of an add-on component, product of a forked development tree and written by different developers.

While MPICH1 has a good choice of algorithms, it performs very poorly because of it's core implementation. Open MPI has poor algorithm choice, but interestingly for simple point-to-point messages performs well, implying a good core structure. Building upon a good structured basic implementation makes algorithm choice and implementation easier, with clear indications as to the success of a different algorithm. The choice of algorithm has major consequences to the effectiveness of an implementation, this was shown by the results. Open MPI's Infiniband implementation was outperformed by both MPICH versions using Gigabit Ethernet when benchmarking MPI_Allgather, illustrated by Figure 3.23.
When using Gigabit Ethernet and the low grade cluster, Open MPI was the only implementation to reach acceptable bandwidth for point-to-point operations. As Figure 3.11 shows, achieving 95MB/s while the closest competitor, MPICH2 achieved 50MB/s, from a 125MB/s link is very respectable. The difference in performance between Ethernet (TCP) and Shared Memory is not wholly unexpected, as the focus of Open MPI is cluster technologies and Linux, for it is on those systems that most testing is undertaken.

4.2 MPICH

MPICH1 implements many collective operations with advanced algorithms, often choosing one based on the MPI function's passed variables. Despite this performance is poor, showing that regardless of the advanced choice of algorithms, the underlying system is too inadequate for this to make much of a difference. Performance results from *Scaliwag* in particular shows that the core MPICH1 system does not handle multiple simultaneous large messages very well, with performance of collective operations degrading as message size increases.
When benchmarking point-to-point MPI operations, MPICH1 had mixed performance. With the low-grade cluster *eScience* benchmarking PingPong it was shown that even with clear interconnects and waiting processes, performance is well below standard. This is in contrast to the results on the high-grade cluster *Scaliwag*, which despite both using Gigabit Ethernet had a sustained bandwidth five times that of *eScience*. This is shown in Figure 4.1.
This could lead to conclusions of inadequacy of the *eScience* Ethernet setup, but with Open-MPI's performance so much greater than that of MPICH1 this is not the case. It is far more likely that MPICH1's usage of the Gigabit Ethernet is either crude or naïve. With the expensive IBM eServer and switch which has undergone months of testing and tuning this is much more likely to be able to perform well with poorer implementations.
MPICH2 uses the same algorithms as MPICH1, yet outperforms it by some margin. The major change between MPICH1 and MPICH2 is the complete reworking of the library's architecture. This is apparent as with MPICH2 a daemon is now run on each host before execution of the MPI program, whereas before the hosts would be connected at runtime of the program. With significant performance increases on low-grade hardware this indicates either a more intelligent MPI library; one which can performs better with lower quality interconnects, with major work undertaken to achieve a better point-to-point transfer time.
It is clear that the work on MPICH2 has had a significant impact on performance with shared

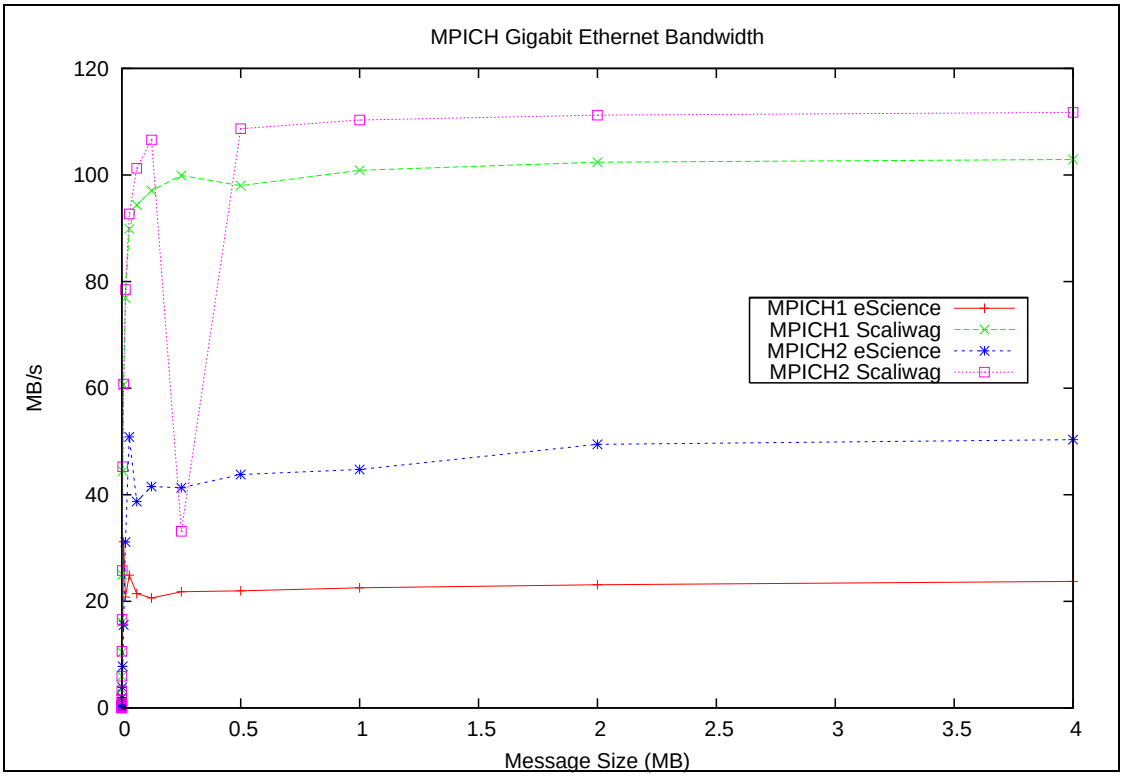

Figure 4.1: Bandwidth of MPICH using Gigabit Ethernet

memory systems, as the *Lomond* results show. MPICH2 outperforms MPICH1 by some margin, and this trend is shown across the board. In point-to-point operations such as PingPong, Figure 3.3, and collective operations such as Reduce_scatter, figure 3.9. The performance of MPICH2 on Lomond is comparable to, and indeed with Reduce_scatter and others slightly outperforms the Sun MPI implementation. This is a major achievement on MPICH2's part, achieving a comparable performance to that of the Sun MPI library, whilst running on Sun hardware with Sun compilers.

4.3 LAM/MPI

It is of little surprise that LAM/MPI was unable to function with Infiniband, Section [2.3.2] shows such support is severely limited. Aside from this, LAM/MPI performance was acceptable on a shared memory system. Complex collaborative operations performed well, i.e. Reduce_scatter, while with others LAM/MPI proved to be a halfway house between the similar MPICH2 and Sun native implementations and the also-rans of MPICH1 and Open MPI.
LAM/MPI is now effectively outdated software, with no future development taking place. The greater interconnect support of Open MPI and the better performance by MPICH2, albeit on a subset of interconnects, make LAM an unlikely choice when selecting an Open Source MPI implementation.

4.4 Future Work

There are a number of interesting points that this investigation has developed. The most obvious is regarding algorithm choice of collective operations with Open MPI. With relatively basic algorithms in place, and with an implementation of advanced algorithms available in MPICH2, a further study could re-implement a particular MPI function, then assess the performance change between old and new implementations. Further interesting data would be produced by comparing with MPICH1 and MPICH2, as three implementations with the same high-level algorithm would show the quality of the core subsystem of each library well. Given Open MPI's high point-to-point results, one would expect to see results similar to those of MPICH2. This could even be undertaken with MPICH2 itself, as the comments for the MPI_Sendrecv function detailed in section 2.2.1 show. These comments suggest an alteration for performance gain, which if undertaken would give MPICH the same implementation as other MPI libraries.
Further work on algorithms would involve the secondary algorithms used by MPICH - those that select the MPI algorithm based on the function's arguments. Frequently multiple algorithms are used, selected based upon the message size of the MPI operation, and it is entirely possible that for a particular system different values could produce better results.
The current heavy development of Open MPI, with development branches and bleeding-edge versions available to the public leads to a future investigation as to how Open MPI has improved it's algorithms. While this report was being written a new stable version was released, and it is entirely likely that this version, or subsequent versions have made improvements upon the algorithms used for the MPI operations.

A further study involves the use of user defined data types. With the MPI standard, the functions are not restricted to predefined data types. It would be interesting so see if this caused a large performance hit, especially with MPICH where many collective algorithms, including MPI_Reduce, fall back to the simple implementation when used with user-defined datatypes.

Bibliography

[1] Message Passing Interface Forum *MPI: A Message-Passing Interface Standard, Version 1.1*
http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[2] Gabriel, Fagg, Bosilca et. al. *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation.*
Euro PVM/MPI 2004, September, 2004, Budapest, Hungary. pp1-3.
http://www.open-mpi.org/papers/euro-pvmmpi-2004-overview/

[3] W. Gropp, E. Lusk, N. Doss, A. Skjellum. *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard.*
http://www-unix.mcs.anl.gov/mpi/mpich1/papers/mpicharticle/paper.html

[4] Weikuan Yu, Tim S. Woodall, Rich L. Graham, Dhabaleswar K. Panda. *Design and Implementation of Open MPI over Quadrics/Elan4.*
19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) Papers, 2005.

[5] Intel Corporation *Intel MPI Benchmarks : Users Guide and Methodology Description.*
IMB v2.3,
http://www.intel.com/cd/software/products/asmo-na/eng/cluster/clustertoolkit/219848.htm

[6] The LAM/MPI Team, Open Systems Lab. *LAM/MPI User's Guide Version 7.1.2.*
http://www.lam-mpi.org/download/files/7.1.2-user.pdf , March 10, 2006.

[7] Rajeev Thakur and William Gropp *Improving the Performance of Collective Operations in MPICH*
10th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2003)
http://www-unix.mcs.anl.gov/~thakur/papers/mpi-coll.pdf

[8] Rolf Rabenseifner *A new optimized MPI reduce algorithm*
High-Performance Computing-Center, University of Stuttgart, Nov. 1997,
http://www.hlrs.de/mpi/myreduce.html