Understanding Properties of I/O between Compute Nodes in a Parallel System

Barton, Alan

National Research Council Canada
Conseil national de recherches Canada

Institute for Information Technology
Institut de technologie de l'information

NRC-CNRC

Understanding Properties of I/O between Compute Nodes in a Parallel System *

Barton, A.
April 2003

* published in NRC/ERB-1102. NRC 45820.

Canada

National Research Council Canada
Conseil national de recherches Canada

Institute for Information Technology
Institut de technologie de l'information

ERB-1102

NRC-CNRC

Understanding Properties of I/O between Compute Nodes in a Parallel System

Barton, A.
April 2003

Canada

NRC 45820

Understanding Properties of I/O between Compute Nodes in a Parallel System.

Alan J. Barton
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
alan.barton@nrc-cnrc.gc.ca

April 14, 2003

Abstract

An attempt is made to understand the properties of I/O between compute nodes in a parallel system. A theoretically plausible framework to emulate an application's I/O communication pattern (without implementation) is presented. The focus, however, lies in a literature review, in conjunction with a set of performance experiments (using LAM, a known MPI implementation, and `mpptest`), which quantify and expose some of the characteristics of the communication within two radically different Beowulf clusters.

1 Introduction

Science is holding of multiple working hypothesis – *Tukey.*

When an existing sequential algorithm running on a commodity workstation[1] cannot compute fast enough, researchers move towards parallel computer[2] systems. These sequential algorithms need to undergo a transformation to a parallel implementation[3] ([5] explains a methodology for and gives a mapping between 3 conceptual classes and 3 methods that can be used by a real programmer on a general-purpose asynchronous parallel machine). This transformation may be the identity, in the case of distributed systems [26], or more complex, in the case of shared memory shared data parallel systems [16]. Parallel algorithm design, therefore, depends on the underlying parallel computer model under consideration ([50], [47], [10], [11], [1], [42], and [49]) and the particular parallel machine's architecture. A major bottleneck for a Beowulf parallel machine architecture is the time consuming input/output (I/O) operations when transferring data from one compute node to another remote compute node using a homogeneous interconnect via processor farming [50].

The project goal is to understand the I/O characteristics (not the storage issues [15]) of a Beowulf cluster by sending and timing different sized data chunks to remote compute nodes via different libraries, which use different strategies for intercommunication. That is, a broad overview of I/O issues that demonstrate some parallel concepts – for an indepth coverage

[1]Example of a Single Instruction, Single Data (SISD) architecture

[2]A parallel computer is *a collection of processing elements that communicate and cooperate to solve large problems fast* ([6] - p.15 citing from Almasi and Gottlieb 1989)

[3]Parallel computers run parallel algorithms using multiple processors [12]

of distributed algorithms please see [34]. For example, locality of reference, bandwidth, latency, synchronization, and overhead are all issues that need to be understood and the tradeoffs must be resolved in the context of a real application workload [6].

The specific question that this project would like to partially answer is the following: How does the number of compute nodes (p) and the data size (d) being transferred between compute nodes affect time (t), when using a particular communication strategy (s) (i.e. via a particular library implementing a message passing paradigm, rather than a live data structure or distributed data structure concept [5]) on a given Beowulf cluster configuration (c)? (See Fig-1)

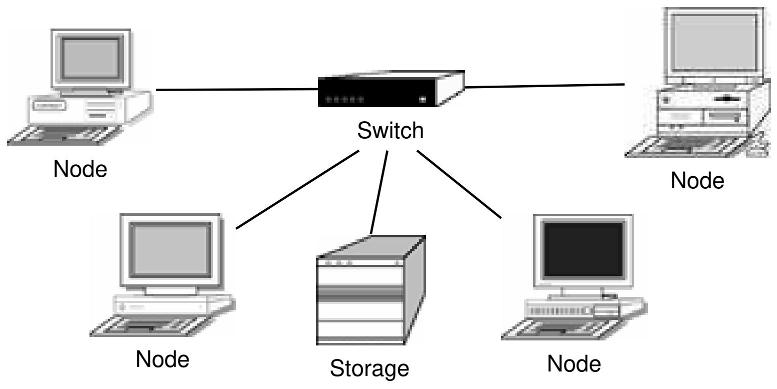


Figure 1: Example of One Possible Switched Heterogeneous Beowulf Cluster

The achievements within the project goal are *(i)* a possible algorithm for trying to understand multiple concurrent application behavior is described, *(ii)* two presentations were given describing a set of issues exposed by the project research ([3], [4]) and *(iii)* performance characteristics of a set of MPI communication methods are reported using `mpptest` [22] for two different Beowulf cluster configurations, including, but not limited to, sending different sized data chunks to remote compute nodes via LAM/MPI.

In Section 2 the relevant literature will be reviewed and related to the project, including a discussion of various strategies for minimizing overall computation time, message passing model variants, message passing standard, other libraries and related experiments on clusters. Section 3 will present the results of the project including a discussion of a parallel program emulator, experimental setup, and experimental results. Section 4 concludes the paper.

2 Literature Review

An essential idea for high throughput parallel computation, is to have all p_n processors computing while data is being transferred. This way, no processor $p_i, i \epsilon [1..n]$ is being blocked while waiting for data from processor $p_j, j \neq i, j \epsilon [1..n]$ [7]. This would be the theoretical optimal use of resources. (Another definition of an optimal parallel solution is if $T_{parallel} = O(\frac{T_{sequential}}{p})$, where $T_{sequential}$ is the sequential time complexity of the problem [11]). However, in practice, it may not be easy to achieve. See, for example, the Pipeline Solution of the Travelling Salesman Problem in [50].

2.1 Minimizing Overall Computation Time

There are various strategies for minimizing overall computation when dealing with data on a parallel (heterogeneous) computer.

1. **Use of Compute Nodes:** How should the compute nodes be used at the most abstract level of understanding? For example, Processor Farming [50] uses the master-worker (Message Passing method [5]).

2. **Placement of Processes:** [48] remarks that faster compute nodes should have more processes (if homogeneous processes). This is a scheduling problem [7].

3. **Collective I/O for Large Data Snapshots:** All processes cooperate to carry out large-scale I/O operations [48] (see also [41]). However, they point out that this may not be appropriate for heterogeneous clusters as the I/O servers are placed in a predetermined manner (e.g. first k processors have I/O servers.). Therefore, I/O server placement should be done at runtime, preferably by schedulers or I/O libraries. [48] mentions that their approach to server placement is implemented within the Panda parallel I/O library. The Panda library is optimized for large data snapshots that need to be written to disk, as in scientific applications.

4. **Placement of I/O Servers:** [48] shows that their heuristics for placing I/O servers (based on reads, which assumes read and write operations are similar) improves overall parallel I/O performance up to a factor of $2-4$ on a cluster of heterogeneous SMPs[4].

5. **Message Passing:** When moving data from one compute node to another, message passing itself may be a bottleneck if all messages must use the same interconnect [48].

6. **Message Size:** [11] explains a Course Grained Multicomputer Model (CGM) in which only a small number of very large messages are used. The idea is that the overhead of communication protocols between processors is reduced.

7. **Communication (Route Balancing):** [7] describes a technique for message traffic distribution that attempts to minimize total time cost by making communication latency uniform over the interconnection network, attempting to avoid large latency variations. That is, messages are sent via the shortest known path at any given time. The technique tries to minimize monitoring and decision overhead. Note *i)* Usually saturation of the network occurs at $\leq 50\%$ of the network's maximum capacity, and *ii)* a bounded amount of latency can be tolerated by hiding it via having enough processes per processor, and scheduling any ready process while other processes wait for their messages [7], Sect-2. Also see [33] for a survey of other methods for message routing attempting to solve the problem of getting the right data to the right place within a reasonable amount of time.

8. **Use of Local Memory:** [11] mentions that the entire data set for a given problem may be loaded into memory and remain there until the problem is solved. For massive data sets, this is not feasible [45].

9. **Use of Local Disk:** [14] mentions that one way to reduce I/O time, when processor farming is used, is for the master to do all of the I/O.

[4]A Symmetric Multi-Processor (SMP) contains homogeneous CPUs.

Observation 1 *The absolute minimal communication time that an algorithm with a set of processors can have is zero. That is, no processor should communicate with any other processor; only computation is performed. However, is this really a parallel algorithm? or simply a set of sequential programs that happen to be running concurrently?*

2.2 Message Passing

Message Passing model variants according to [5]:

1. **synchronous send and receive:** For example, Hoare's influential fragment CSP (1978), which inspired a language Occam (1983)

2. **monitor and remote-procedure-call languages and systems:** That is, one process communicates with another by invoking a procedure defined within some other process or within a passive, globally accessible module. Arguments are shipped out in one message, results are returned in another. For example, Modula, a concurrent language (assumes that multiple processes inhabit the same address space). And Ada (US Dept. of Defense 1982), a parallel object-oriented programming language.

3. **streams:** sender appends messages to the end of a message stream; receiver inspects the stream's head, (1974 Kahn). For example, Prolog (Shapiro 1987), a concurrent logic language.

2.3 Message Passing Standard

There is a standard that industry has agreed upon for message passing, namely MPI [21]. There are many implementations of the standard, one such is Local Area Multicomputer (LAM) [29], which is targeted towards heterogeneous computers on a network.

One problem in a heterogeneous environment, is data conversion. MPI makes heterogeneous data conversion a transparent part of its services by requiring datatype specification for all communication operations (Gropp, Lusk, Doss, Skjellum 1996). There are many different data exchange mechanisms possible; some, according to [46], as implemented in the channel interface a part of MPICH (another implementation of MPI) are:

1. **Eager:** In the *eager* protocol, data is sent to the destination immediately. If the destination is not expecting the data (e.g. no MPI_Recv has yet been issued for it), the receiver must allocate some space to store the data locally.

2. **Rendezvous:** In the *rendezvous* protocol, data is sent to the destination only when requested (the control information describing the message is always sent). When a receive is posted that matches the message, the destination sends the source a request for the data.

3. **Get:** In the *get* protocol, data is read directly by the receiver. This choice requires a method to directly transfer data from one process's memory to another. A typical implementation might be `memcpy`.

Another issue is related to whether separate processes should be spawned or separate threads of execution on a processor – MPI_INIT_THREAD may be used instead of MPI_INIT, if, for example, on a cluster of SMPs [13].

2.3.1 MPI-IO

The new MPI 2.0 standard has a chapter for I/O and more specifically, for Parallel I/O. Implementations of the I/O chapter include ROMIO ([23], [39]) and MPI-IO/GPFS, which is optimized for the IBM General Parallel File System ([32]). There are some libraries that try to build on the MPI-2 standard (more specifically, a particular implementation). For example, *MPI_Connect.MPI_Conn_getFile*(...) [14] is such an API method call, to get a file from a remote storage node, which uses dedicated file serving daemons. There is also a library where MPI primitives are supported through a distributed shared memory, namely MPI_SH (implementation of MPI on top of Distributed Shared Areas DVSA) [8].

2.4 Other Libraries

There are also many other parallel libraries available. See, for example, TPIE: Transparent Parallel I/O Environment [25], ViC∗: Virtual-Memory C∗ Compiler [24], or PASSION: Parallel And Scalable Software for Input-Output [38].

2.5 Relevant Experiments on Clusters

Benchmarks are one way that specific communication implementations may be analyzed in terms of their performance w.r.t other implementations [17]. SKaMPI [40] (and the more recent SKaLIB [20], which supports development of benchmarks) discuss benchmarking MPI. For example, [44] discusses benchmarking collective MPI operations and [18] discusses the development of MPIBench that uses a very precise and portable distributed timing mechanism to accurately measure the performance of a single MPI communication routine without having to average over many calls. Note also that [30] provides information about computer system performance benchmarks, benchmark results, and benchmark code and that the Transaction Processing Performance Council ([31]) defines transaction processing and database benchmarks. For example, the relationship between performance and scale of business enterprise can be seen in the on-line transaction processing (OLTP) benchmarks, which rate the performance of a system in terms of its throughput in transactions per minute (tpm) on a typical workload [6].

Results with NT clusters are shown in [48] and with Linux clusters in [37]. The Linux case study, uses the Parallel Virtual File System, the ROMIO [23] MPI-IO implementation, and the Hierarchical Data Format library (HDF5 [19]) in the context of a FLASH application I/O benchmark. See also [46] for information related to testing MPI methods.

Rather than benchmark an existing application, another approach when trying to understand the I/O, is to build a performance model [9]. The approach presented in Section 3.1 could be thought of as a way to build multiple performance models.

3 Project Report

Experiments were designed in order to partially understand the characteristics of two Beowulf clusters using a message passing system. The experiments could have been designed to be very low level in order to understand the communication protocol itself (e.g. TCP or UDP), or much higher level in order to understand a performance model (e.g. Section 3.1). However, it was decided that a medium level was more appropriate; one that specifically focussed on a particular implementation of the MPI standard. Further, the presented results could have been compared with results for other clusters, but this was eliminated from the project due to time constraints (See [46] for a discussion of results obtained from other parallel systems).

A possible algorithm for understanding any parallel message passing program is discussed in Section 3.1, followed by experimental results for a homogeneous cluster of SMPs in Section 3.3.1, and concluding with heterogeneous cluster results in Section 3.3.2.

3.1 Parallel Program Emulation

The intuition behind the following emulation[5] algorithm, is that a parallel algorithm may be represented as computation and communication, where the computation could be emulated on each compute node by waiting a certain amount of time and communication could be actually implemented and measured for a particular set of communication topologies (See Fig-2). To get more and more accurate emulations of parallel algorithms, the distribution of the waiting time (e.g. Gaussian, Poisson, etc.) of the computation could be implemented s.t. the overall emulation would represent a particular parallel program. For this project, only the idea for this algorithm is discussed, due to time constraints, as the more foundational experiments need to be performed (See Section 3.3).

Also, [43] states: *Performance is largely a function of the frequency and nature of inter-component communication, in addition to the performance characteristics of the components themselves, and hence can be predicted by studying the architecture of a system [Clements 1996].* From this point of view, the following algorithm may be thought to be a way of modelling many possible parallel algorithms, with the goal of studying their I/O performance, and, hence, the parallel algorithm's performance.

Observation 2 *In one sense, only one figure of merit matters, the wall clock run time of a particular application on a particular system [2], hence this statement by David H. Bailey, provides some supporting motivation for this section.*

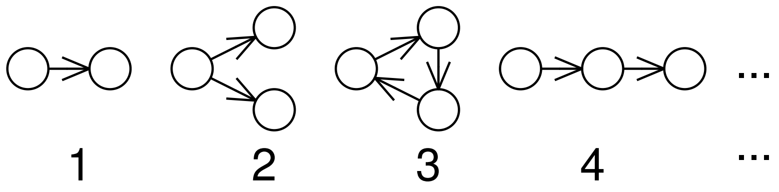


Figure 2: Examples of possible processor communication topologies (patterns), which are independent of the underlying parallel system architecture; nodes represent processors, while directed edges represent data being transferred: 1) Client/Server, 2) Task Farming, 3) Token Ring, 4) Pipelined, 5) others.

Algorithm: Let $P = \{p_1, p_2, ..., p_p\}$ be the set of all available (virtual and real) processors in the chosen parallel system (e.g. Beowulf Cluster), and let $P_{Starters}(\subseteq P) = \{p_i \epsilon P$ s.t. p_i is going to receive a $t_i \epsilon T = \{t_1, t_2, ..., t_t\}$, which will determine pi's computation and communication behaviour$\}$. Note that $P_{Starters}$ are like the set of concurrently running applications (parallel algorithms). Let there be associated with each $p_i \epsilon P_{Starters}$ a graph

[5]The term *emulation* was used by Prof. Dehne during a brainstorming session, while the author was using the term *simulation*.

$G_{Targets} = (P_{Targets}, E_{Targets})$ representing a communication topology of the parallel algorithm $p_i \epsilon P_{Starters}$. Note that *i)* $P_{Targets} \cap P_{Starters} = \emptyset$ (i.e. each parallel algorithm has a master on a separate compute node) , *ii)* $P_{Targets} \cup P_{Starters} \subseteq P$, and *iii)* $P_{Targets}$ are like the set of processors supporting a particular parallel algorithm, which receive a communiqué ($t_i \epsilon T$) from their associated *root* processor. Not all $p_i \epsilon P_{Targets}$ will receive t_i from the *root* processor ($p_i \epsilon P_{Starters}$) , because $E_{Targets}$ determines the communication topology for that particular parallel algorithm. That is, $e_{jk} \epsilon E_{Targets}$ is an edge in the communication topology such that communication information (data) will be transferred from $p_j \epsilon P_{Targets}$ to $p_k \epsilon P_{Targets}$ (For example, in MPI this could be implemented using a communicator with topology; possibly MPI_Graph_create).

Definition 1 $t_i \epsilon T$ *contains the following information:*

1. $|m| =$ *size of randomly generated data message – m – to transmit*
2. $c =$ *communication method. For example, some possibilities are:*
   i) *message passing [21],*
   ii) *shared memory, [8]*
   iii) *shared file on a parallel file system [35].*
3. $|P_{Targets}| =$ *number of target processors*
4. $G_{Targets} = (V_{Targets}, E_{Targets})$ *communication topology of the parallel algorithm*
5. $P_{Targets} =$ *set of target processors to which m will be directly sent;*
6. $P_{Targets} \subseteq V_{Targets}$

3.2 Experimental Setup

Two Beowulf clusters were used for the experiments, with the emphasis of the experiments on Cluster (i), as more detailed information was known, at the time of writing, than for Cluster (ii).

Cluster (i) is located at the Institute for Information Technology (IIT), of the National Research Council. This is a homogeneous cluster of SMPs; 20 PowerEdge 2600 processor units, each having 2 Intel Xeon processors running at 2.4GHz with Hyper-Threading Technology (2 logical CPU's per physical CPU) and having 2 GB of SDRAM and a 36GB hard drive spinning at 15K RPM (See [27]). The interconnect is a PowerConnect 5224 Gigabit Ethernet Managed switch with 24 ports (See [28]). This cluster uses LAM/MPI 6.5.9 and a shared network drive.

Cluster (ii) is located at the Institute for Biodiagnostics (IBD), of the National Research Council. This cluster is heterogeneous, and consists of 4 dual Athlon processor units operating at 1.666Ghz frequency, each with 2Gb RAM, 5 Xeon processor units operating at 1.6Ghz with 1Gb RAM, 2 Pentium III units at 1Ghz with 256Mb RAM, 1 Pentium III units at 1Ghz with 512Mb RAM, 1 Pentium III units at 930Mhz with 256Mb RAM, and 4 Pentium III units at 860Mhz with 512Mb RAM. This cluster uses Red Hat Linux 7.3, LAM/MPI 6.5.8 and a shared network drive.

The program used for the experiments is MPPTest [22], which was compiled and configured separately for each cluster. Note that LAM/MPI *daemon* mode was used, as opposed to LAM/MPI *client to client* mode [29] and that gcc 2.96 was used for compilation on Cluster (i), while gcc 3.2 was used on Cluster (ii). It should be noted that the different versions of gcc, will produce different object code, and may make the intra-cluster results incomparable. Note also, that the command:

```
basetest -maxnp=x -ttime=00:10 -echo -long
```

was used to collect the results, where $x =< 20, 40, 80 >$ for Cluster (i) and $x =< 17 >$ for Cluster (ii). Future experiments should include $x =< 21 >$ for Cluster (ii).

The following experiments were made using MPPTest [22]:

1. **Point-to-point Performance:** Performance of messages sent from one processor to one other processor were measured – short and long (blocked) messages as well as long (non-blocked) messages were used.
   (a) **short:** `mpirun -np 2 mpptest -auto -gnuplot -givedy`
   (b) **long:** `mpirun -np 2 mpptest -size 262,144 1,048,576 32,768 -gnuplot -givedy`
   (c) **longnb:** `mpirun -np 2 mpptest -async -size 262,144 1,048,576 32,768 -gnuplot -givedy`
2. **Short Messages out of Cache:** Performance of caching effects were measured by sending short messages from one processor to one other processor.
   (a) **shortc:** `mpirun -np 2 mpptest -gnuplot -givedy -cachesize 4,194,304`
3. **Short Messages with Vectors:** Performance of short messages containing vectors were measured.
   (a) **shortv:** `mpirun -np 2 mpptest -gnuplot -givedy -vector -vstride 128`
4. **Bisection:** Performance of sending data from half of the processors to the other half of the processors was measured. Note that Cluster (i) and Cluster (ii) have a different number of processors and the implementation of `basetest` only measures information up to and including 32 processors.
   (a) **bshortX** $x =< 4, 17, 20, 32 >$**:** `mpirun -np x mpptest -auto -bisect -gnuplot -givedy`
   (b) **blongX** $x =< 4, 20, 32 >$**:** `mpirun -np x mpptest -size 262,144 1,048,576 32,768 -gnuplot -givedy -reps 25 -tgoal 1.0 -autoreps -sample_reps 5 -bisect`
5. **Collective:** Performance of collective experiments – **bcast** measures scatter effects and **dsum** measures double precision reduction effects.
   (a) **bcast** **(**$x =< 4, 17, 20, 32 >$**):** `mpirun -np x goptest -gnuplot -givedy -bcast`
   (b) **dsum** **(**$x =< 4, 17, 20, 32 >$**):** `mpirun -np x goptest -gnuplot -givedy -dsum -size 0 1,024 256`

3.3 Experimental Results

The results focus on the latency and bandwidth for a set of MPI functions as mentioned in Section 3.2. High variance in the bandwidth results are a possible indicator that other processes were executing concurrently (i.e. contention for resources – the interconnect). Detailed analysis of the results cannot be performed, as the causes for the observed effects (as seen in the figures) would be speculative; however, abnormal patterns will be discussed. Many of the results display expected behavior and are included to demonstrate the specific constants that the different MPI methods exhibit on the two clusters.

3.3.1 Cluster (i) Experimental Results

Results of experiments on the homogeneous cluster of SMPs – Cluster (i) (See Section 3.2) – are discussed in this section.

The `short` Fig-3(a,b) experiments exhibit high latency and low bandwidth when 1 process is used per compute node (containing 2 CPUs). High variability of the latency is also observable. The experiments with 2 processes per compute node Fig-3(c,d) have a almost horizontal latency (and linear bandwidth) indicating that the interconnect may be reaching the saturation point, while the 4 processes per compute node Fig-3(e,f) display reduced bandwidth.

The point-to-point communication results for `long` Fig-4 display different characteristics to that for `short`. The latency is almost 100 times increased and the linear relationship also increases more rapidly. When Fig-4(a), containing results for 1 process/compute node is compared with Fig-4(b,c), containing 2 and 4 processes/compute node, respectively, one can see a reduction by not quite one half (as would be expected with the compute nodes containing 2 physical CPUs) due to possible intra-SMP contentions (e.g. OS context switching).

The non-blocking point-to-point communication results for `longnb` Fig-5, when compared to Fig-4, indicate similar latency characteristics. However, the bandwidth is different (e.g. compare Fig-5(d) with Fig-4(d)). That is, the initial high bandwidth has been replaced by a much more horizontal curve.

Short messages (out of cache) results for `shortc` contained in Fig-6 don't indicate radical departure from those results contained in Fig-3 except for the reduced latency variation in Fig-6(a).

The short messages with vectors `shortv` contained in Fig-7 exhibit radically different behavior. For example, Fig-7(d,f) start to show a decrease in bandwith as message size increases. This may be due to another (unknown) process starting, or may, in fact, be due to the overhead inherent within vectors.

Bisection experiments for `bshort4` Fig-8, `bshort20` and `bshort32` Fig-9, `blong4` Fig-10, and `blong20` and `blong32` Fig-11, show very linear bandwidth relationships for the short messages, while contain very horizontal relationships for the long messages. This may indicate that the long message experiments completely utilize the interconnect. However, it is interesting to note that the maximum obtained bandwidth for the short messages is above that obtained for the long messages; possibly indicating that the bandwidth increases and then decreases and becomes stable when the overhead and contention effects no longer play a role.

Collective `bcast` (scatter) Fig-12(a,c,e) experiments indicate that, overall, as the number of processes per compute node increases from 1 to 2 to 4, the latency decreases and the variability decreases. While `dsum` (reduction) Fig-12(b,d,f) experiments show the same decrease in latency, but with increased variability as the number of processes increases.

3.3.2 Cluster (ii) Results

The heterogeneous cluster results are not as complete or reliable as those in Section 3.3.1. For example, the long message tests did not complete, possibly indicating the tests were too ambitious for this cluster, or that this cluster was already being heavily used. This last point is worth stressing, as the cluster is located in Winnipeg, Manitoba, and control over the experimental variables (such as who is using the system) are not controllable. Also, the bisection tests only completed for 4, 8 and 16 processors (this cluster only has 17 compute nodes). Refer to Fig-13 and Fig-14 for detailed results.

4 Conclusion

A deeper understanding of more of the issues involved in transferring knowledge from one compute node to another in a parallel system has been obtained through a search of the relevant literature, along with a set of performance experiments on two Beowulf clusters and a plausible framework to emulate an application's I/O communication pattern.

The author has learned there are many other interesting avenues for further research. For example, David H. Bailey has written an interesting paper about Beowulf Clusters [2], which lead to an investigation of his other papers, which lead to a paper about parallel integer relation detection, giving a new formula for the calculation of π. A very interesting paper, which will need to be read in more depth in the future, but completely unrelated to the subject of this project.

In terms of this project, other interesting things that could have been included are: *(i)* more experiments related to clusters (e.g. the Carleton cluster and sunfire machine results could have been collected and compared against those reported in this paper), *(ii)* other standard benchmarks, like the parallel Linpack Benchmark by Jack Dongarra and the NAS Parallel Benchmark (NPB) suite designed at NASA Ames Research Center [2], *(iii)* measurements of the long-term throughput of a Beowulf, instead of the conventional benchmark performance testing [2], *(iv)* develop a formal model of communication patterns and test them within the framework of Section 3.1 (including an implementation), *(v)* more experiments related to gathering information about throughput versus the number of processors ([6] p.10), *(vi)* Maria de los Santos Perez Hernandez (Spain) was contacted about MAPFS [36], from which information was obtained. However, this information could not be used for this project as the project would have to focus on file systems for it to be appropriate. *(vii)* Laura Ricci (Italy) was contacted to try to obtain MPI_SH. The authors of the software were willing to share the information, but could not supply the information fast enough within the constraints of this project. It would have been nice to be able to compare MPI_SH (based on a shared memory model) directly against a message passing model of the algorithm mentioned in Section 3.1.

As more experience is gained by the author, more specific knowledge may be used in conjunction with relevant experiments on the Beowulf Cluster, to help come to a deeper scientific understanding of the issues involved in understanding properties of I/O between compute nodes in a parallel system.

5 Acknowledgements

The author is very grateful to Robert Orchard, Julio J. Valdés, A. Fazel Famili, Robyn Paul, Marc Léveillé from the Integrated Reasoning Group, Institute for Information Technology, National Research Council of Canada. R. Orchard, Valdés and A. Famili gave their support for this project, while R. Paul gave helpful suggestions and M. Léveillé provided some technical support related to running experiments on Cluster (i). The author is also very grateful to Professor Frank Dehne and David Bickle of the School of Computer Science at Carleton University. F. Dehne taught the course and D. Bickle informed me about NPACI Rocks and Ganglia. Finally, and certainly of no less importance, the author appreciates

Nicolino Pizzi from the Institute for Biodiagnostics, National Research Council Canada for the possibility of using Cluster (ii).

References

[1] Y. Ben-Asher A. Agbaria and I. Newman. Communication-processor tradeoffs in limited resources pram. In *Eleventh ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 74–82, Saint Malo, France, June 1999.

[2] David H. Bailey. *How Fast Is My Beowulf.* LBNL-48598. MIT Press, (See also http://www.nersc.gov/~dhbailey/dhbpapers/beowulf-perf.pdf), 2001.

[3] Alan Barton. Presentation: Understanding properties of i/o between compute nodes in a parallel system. Rm 278, Integrated Reasoning Group, Institute for Information Technology, National Research Council Canada, Building M-50, Ottawa, Ontario, Canada, March 17 2003.

[4] Alan Barton. Presentation: Understanding properties of i/o between compute nodes in a parallel system. Rm 415, Southam Hall, Carleton University, Ottawa, Ontario, Canada, March 28 2003.

[5] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. In *ACM Computing Surveys*, volume 21, pages 323–357, September 1989.

[6] David E. Culler and Jaswinder Pal Singh with Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc., 340 Pine Street, Sixth Floor, San Francisco, CA, USA, 1999. ISBN: 1-55860-343-3.

[7] I. Garcés D. Franco and E. Luque. A new method to make communication latency uniform: Distributed routing balancing. In *ACM International Conference on Supercomputing (ICS)*, pages 210–219, Rhodes, Greece, 1999.

[8] P. Mori D. Guerri and L. Ricci. Mpi_sh: a distributed shared memory implementation of mpi. In *4th International Conference on Massively Parallel Computing Systems*, pages 129–134, Ischia, Italy, April 2002. ISBN: 0-9669530-0-2.

[9] P.D. Coddington D.A. Grove. A performance modeling system for message-passing parallel programs. Technical Report DHPC-105, Adelaide University, October 2001.

[10] W.F. McColl D.B. Skillicorn, Jonathan M.D. Hill. Questions and answers about bsp. *PRG-TR-15-96, Oxford University Computing Laboratory*, November 1996.

[11] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. Preliminary version published in the proceedings of the 1993 ACM conference on Computational Geometry., 1994.

[12] Professor Frank Dehne. Comp 5704, parallel algorithms and their implementation, carleton university. January 2003.

[13] Message Passing Interface Forum. Mpi-2: Extensions to the message-passing interface. July 18 1997.

[14] M. Resch G. Fagg, E. Gabriel and J. Dongarra. Parallel io support for meta-computing applications: Mpi_connect io applied to pacxmpi. In Yiannis Cotronis and Jack Dongarra, editors, *8th European PVMMPI Users' Group Meeting, Lecture Notes in Computer Science 2131.* Springer Verlag, Berlin, 2001.

[15] J. Vitter G. Gibson and J. Wilkes et al. Strategic directions in storage i/o issues in large-scale computing. In *ACM Computing Surveys*, volume 28, pages 779–793, December 1996.

[16] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Press Syndicate of the University of Cambridge, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, 1988.

[17] William Gropp and Ewing Lusk. Reproducible measurements of mpi performance characteristics. Proceedings of PVMMPI'99, 1999.

[18] Duncan Grove and Paul Coddington. Precise mpi performance measurement using mpibench. Technical Report DHPC-104, Adelaide University, August 2001.

[19] http://hdf.ncsa.uiuc.edu/HDF5/. Hdf5 - the next generation of the hdf library and tools homepage. February 2003.

[20] http://liinwww.ira.uka.de/ skampi/. Skampi homepage. March 2003.

[21] http://www unix.mcs.anl.gov/mpi/. Mpi - the message passing interface standard homepage. February 2003.

[22] http://www unix.mcs.anl.gov/mpi/mpptest/. Mpptest - measuring mpi performance web page. March 2003.

[23] http://www unix.mcs.anl.gov/romio/. Romio: A high-performance, portable mpi-io implementation homepage. February 2003.

[24] http://www.cs.dartmouth.edu/ thc/ViC/index.shtml. Vic* homepage. February 2003.

[25] http://www.cs.duke.edu/TPIE/. Tpie, a transparent parallel i/o environment homepage. February 2003.

[26] http://www.cs.wisc.edu/condor/. Condor project homepage. February 2003.

[27] http://www.dell.com/us/en/biz/products/model_pedge_pedge_2600.htm. Poweredge 2600 compute node. April 2003.

[28] http://www.dell.com/us/en/biz/products/model_pwcnt_networks_pwcnt_5224.htm. Powerconnect 5224 interconnect (switch).

[29] http://www.lam mpi.org/. Lam/mpi homepage. February 2003.

[30] http://www.netlib.org/benchweb/. Benchweb homepage. March 2003.

[31] http://www.tpc.org/. Transaction processing performance council homepage. March 2003.

[32] R. Hedges B. Jia J. Prost, R. Treumann and A. Koniges. Mpi-io/gpfs, an optimized implementation of mpi-io on top of gpfs. In *ACM Super Computing (SC2001)*, Denver, USA, November 2001.

[33] T. Leighton. Methods for message routing in parallel machines. In *24th Annual ACM Symposium on Theory of Computing (STOC)*, Victoria, British Columbia, Canada, May 1992.

[34] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., 340 Pine Street, Sixth Floor, San Francisco, CA, USA, 1996. ISBN: 1-55860-348-4.

[35] R. B. Ross P. H. Carns, W. B. Ligon III and R. Thakur. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. (Best Paper Award).

[36] María S. Pérez, Félix García, and Jesús Carretero. A new multiagent based architecture for high performance I/O in clusters. In *Proceedings of the 2nd International Workshop on Metacomputing Systems and Applications MSA'2001*, September 2001.

[37] A. Cheng R. Ross, D. Nurmi and M. Zingale. A case study in application i/o on linux clusters. In *ACM Super Computing (SC2001)*, Denver, Colorado, USA, November 2001.

[38] A. Choudhary R. Ponnusamy R. Thakur, R. Bordawekar and T. Singh. Passion runtime library for parallel i/o. In *Scalable Parallel Libraries Conference (SPLC94)*, Mississippi State, Mississippi, October 1994.

[39] W. Gropp R. Thakur and E. Lusk. On implementing mpi-io portably and with high performance. In *ACM Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, Atlanta, Georgia, USA, May 1999.

[40] Lutz Prechelt Ralf Reussner, Peter Sanders and Matthias Muller. Skampi: A detailed, accurate mpi benchmark. In *Recent Advances in Parallel Virtual and Message Passing Interface. 5th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science 1497, Liverpool, UK, September 1998. Springer-Verlag.

[41] Y. Cho J. Lee S. Kuo, M. Winslett and Y. Chen. Efficient input and output for scientific simulations. In *ACM Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, Atlanta, Georgia, USA, May 1999.

[42] D. Skillicorn and D. Talia. Models and languages for parallel computation. In *ACM Computing Surveys*, volume 30. ACM, June 1998.

[43] Conniew U. Smith and Lloyd G. Williams. *Performance Solutions A Practical Guide to Creating Responsive, Scalable Software*, volume ISBN 0-201-72229-1. Addison-Wesley, 2002.

[44] Ralf Reussner Thomas Worsch and Werner Augustin. On benchmarking collective mpi operations. In *European PFM/MPI 2002*, Lecture Notes in Computer Science 2474, pages 271–279. Springer-Verlag, 2002.

[45] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. In *ACM Computing Surveys*, volume 33, pages 209–271, June 2001.

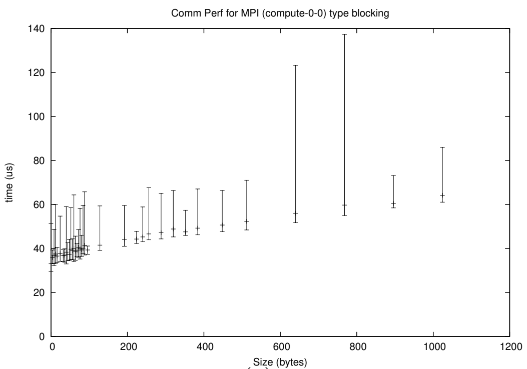
[46] N. Doss W. Gropp, E. Lusk and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. Preprint MCS-P567-0296, Argonne National Laboratory and Mississippi State University, July 1996. http://www-fp.mcs.anl.gov/division/publications/abstracts/abstracts96.htm.

[47] Tiffani L. Williams. *A General-Purpose Model For Heterogeneous Computation.* PhD thesis, Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, Fall 2000.

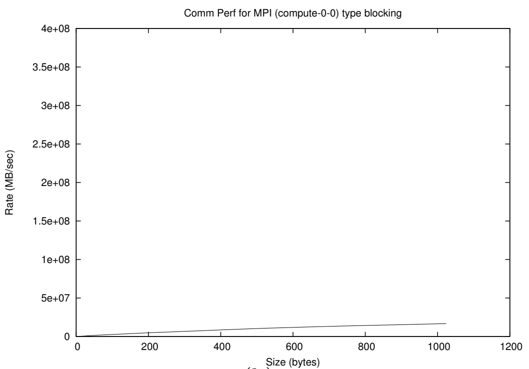
[48] S. Kuo J. Lee Y. Cho, M. Winslett and Y. Chen. Parallel i/o for scientific applications on heterogeneous clusters: A resource-utilization approach. In *ACM International Conference on Supercomputing (ICS99)*, pages 253–259, Rhodes, Greece, 1999.

[49] N. Nisan Y. Mansour and U. Vishkin. Trade-offs between communication throughput and parallel time. In *ACM Symposium on Theory of Computing (STOC)*, pages 372–381, Montreal, Quebec, Canada, May 1994.

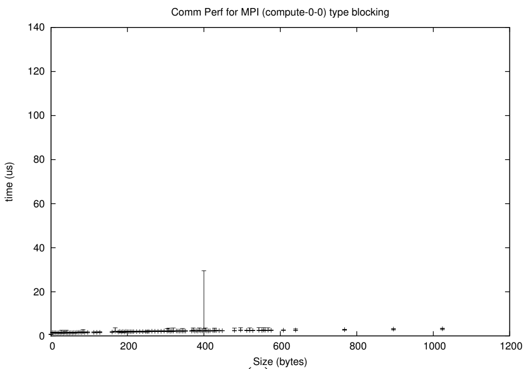
[50] L. Yang and L. Jin. Integrating parallel algorithm design with parallel machine models. pages 131–135, New York, NY, USA, 1995. ACM Press.



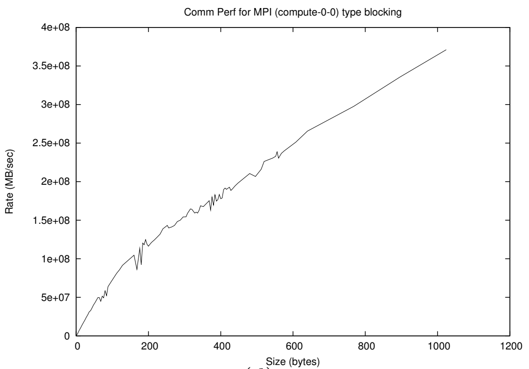
(a)



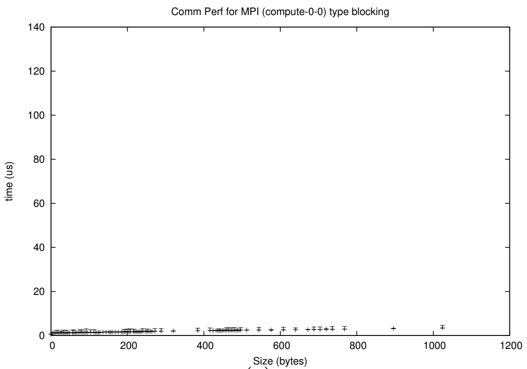
(b)



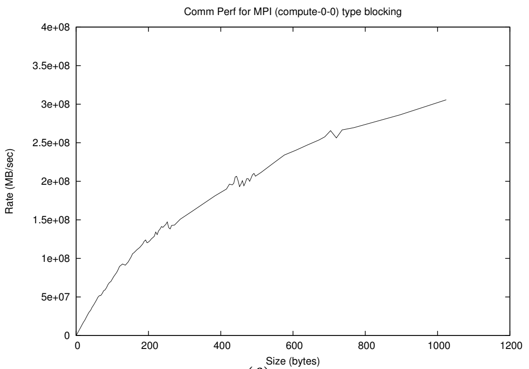
(c)



(d)

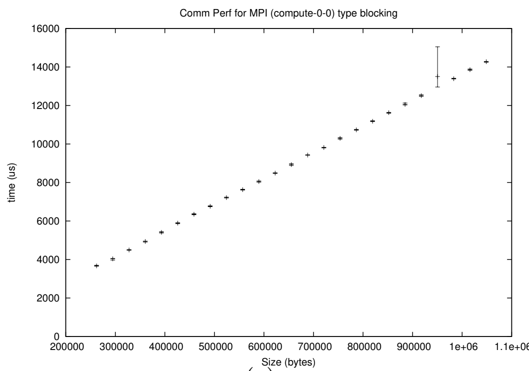


(e)

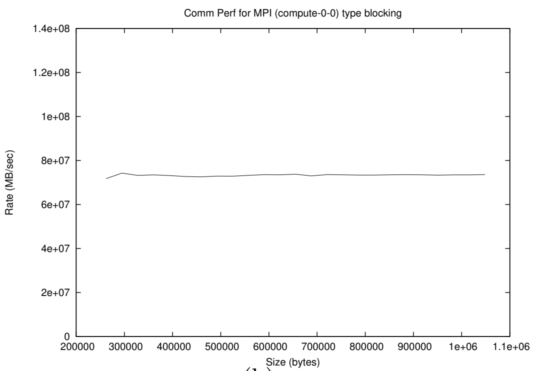


(f)

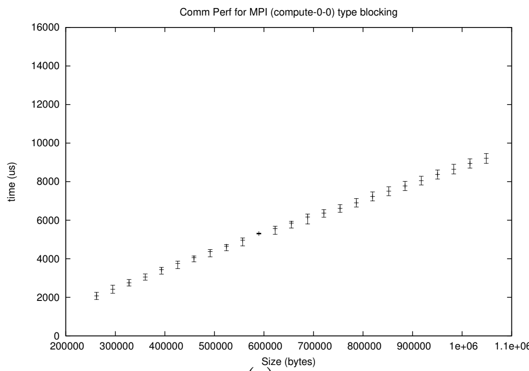
Figure 3: short – Cluster (i) lam gcc2.96 c0-19 (a,b) latency and bandwith for 1 process/compute node (c,d) latency and bandwidth for 2 processes/compute node (e,f) latency and bandwidth for 4 processes/compute node



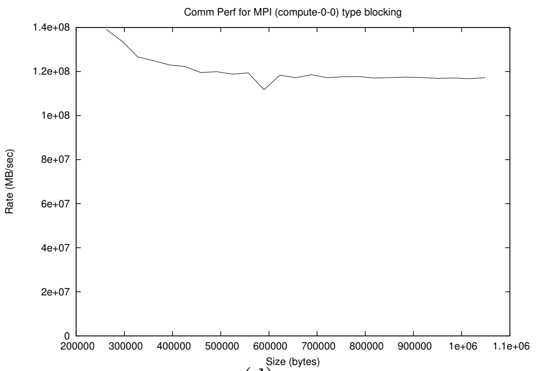
(a)



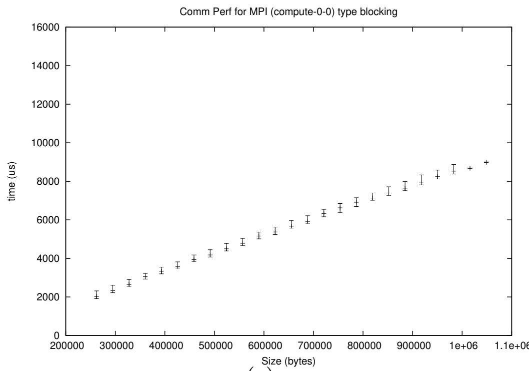
(b)



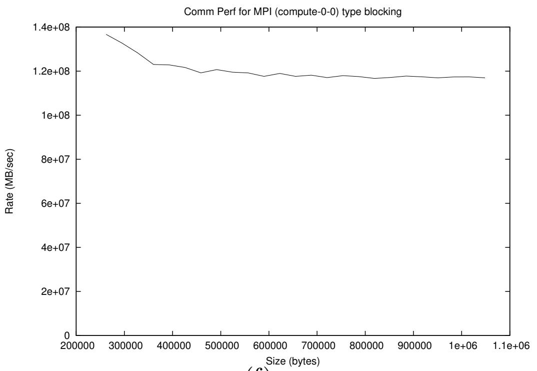
(c)



(d)

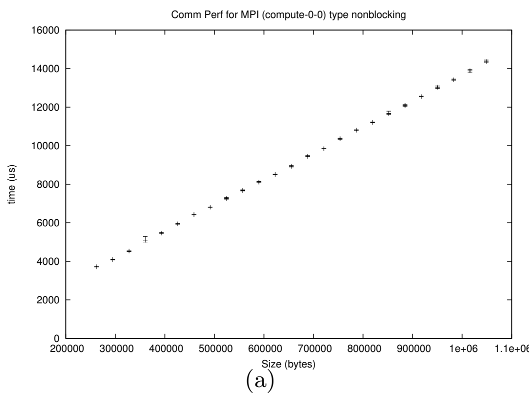


(e)

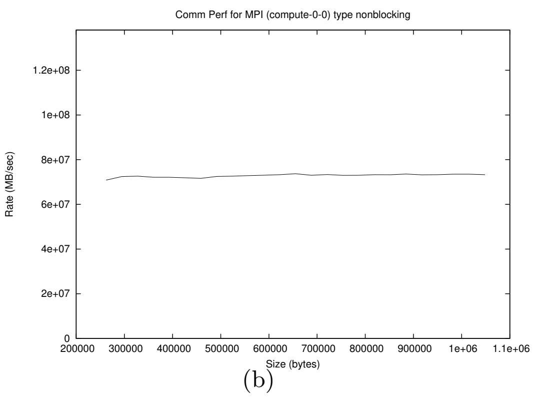


(f)

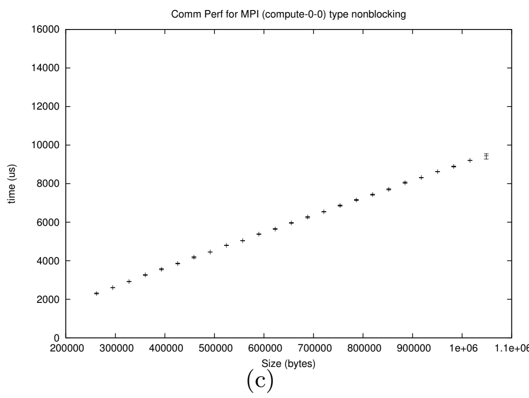
Figure 4: long – Cluster (i) lam gcc2.96 c0-19 (a,b) latency and bandwidth for 1 process/compute node (c,d) latency and bandwidth for 2 processes/compute node (e,f) latency and bandwidth for 4 processes/compute node



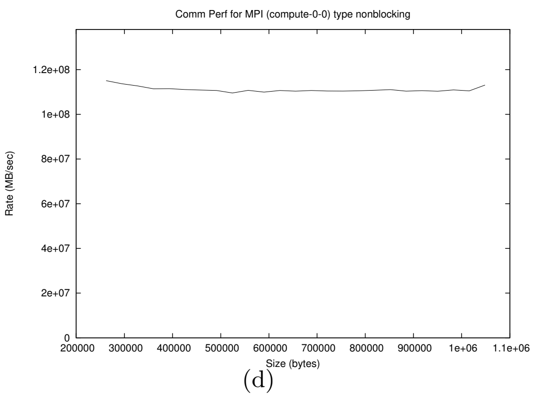
(a)



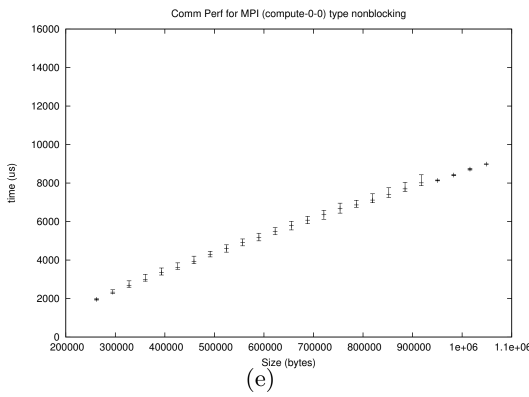
(b)



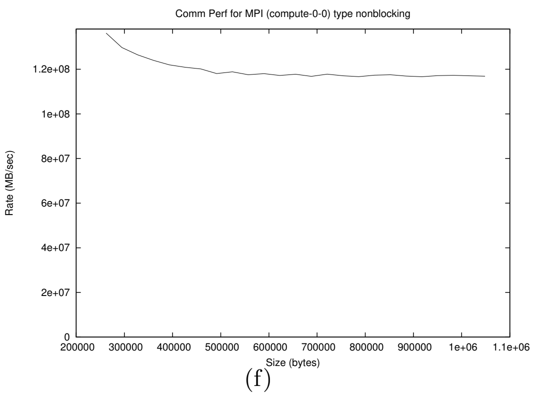
(c)



(d)

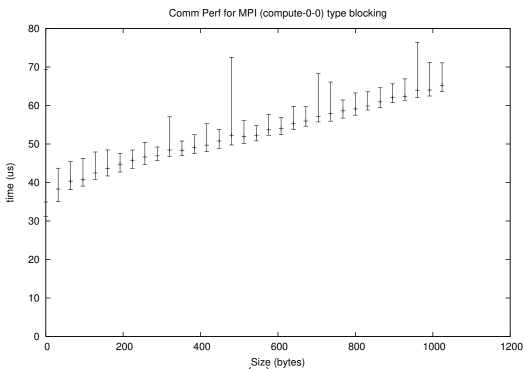


(e)

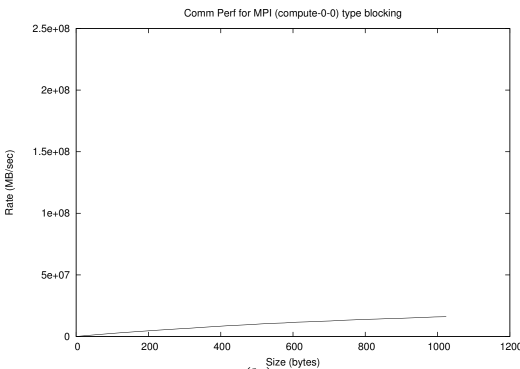


(f)

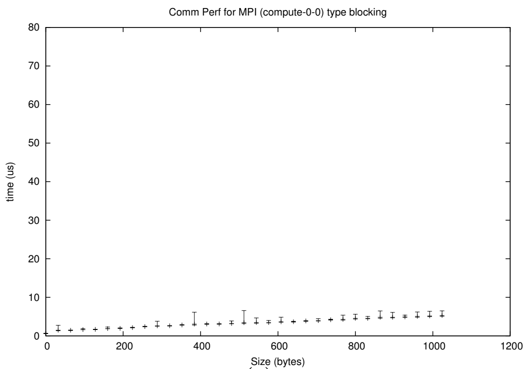
Figure 5: longnb – Cluster (i) lam gcc2.96 c0-19 (a,b) latency and bandwidth for 1 process/compute node (c,d) latency and bandwidth for 2 processes/compute node (e,f) latency and bandwidth for 4 processes/compute node



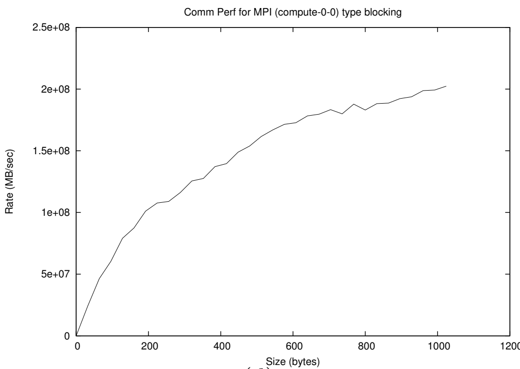
(a)



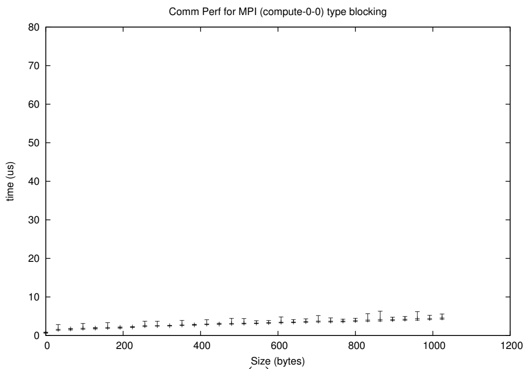
(b)



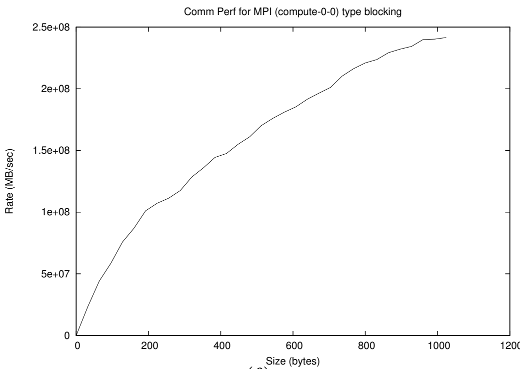
(c)



(d)

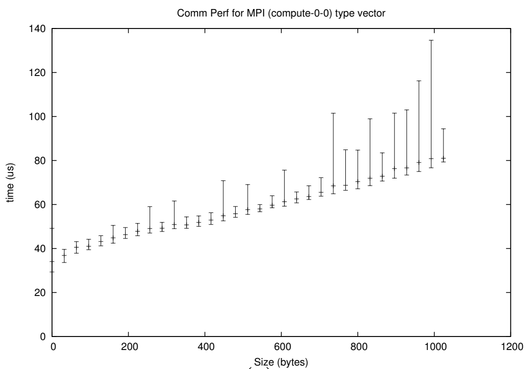


(e)

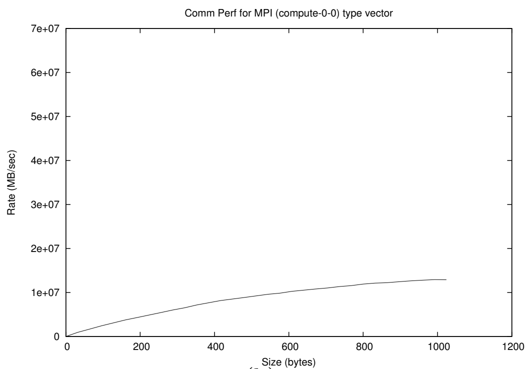


(f)

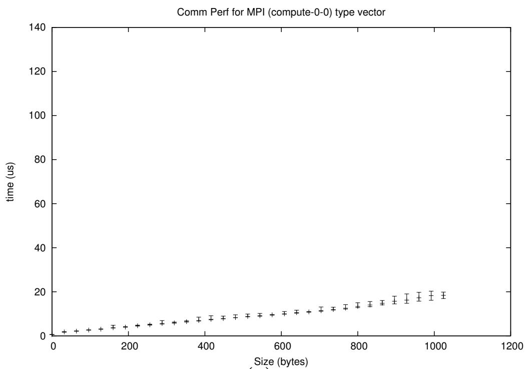
Figure 6: shortc – Cluster (i) lam gcc2.96 c0-19 (a,b) latency and bandwith for 1 process/compute node (c,d) latency and bandwidth for 2 processes/compute node (e,f) latency and bandwidth for 4 processes/compute node



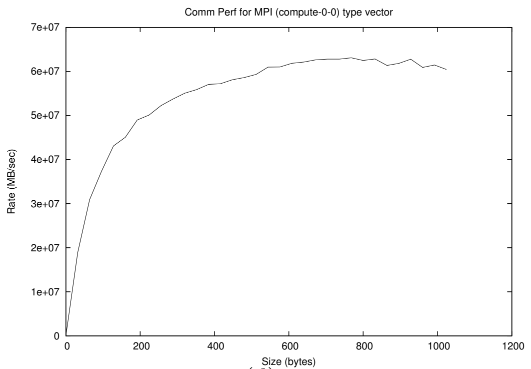
(a)



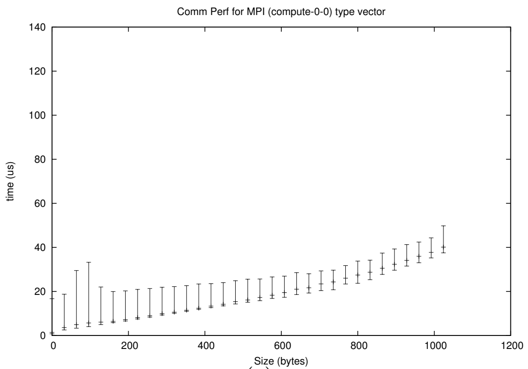
(b)



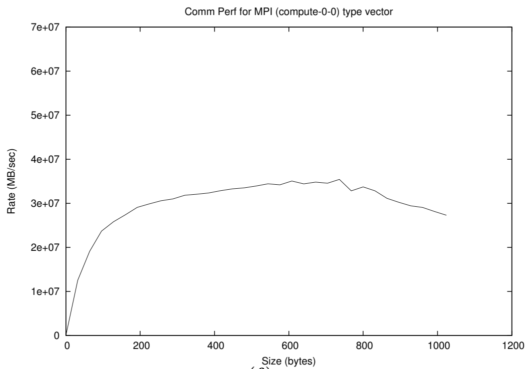
(c)



(d)

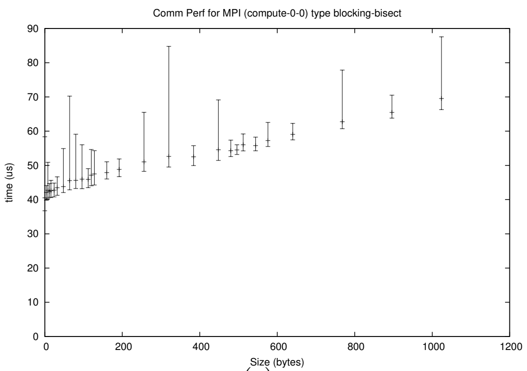


(e)

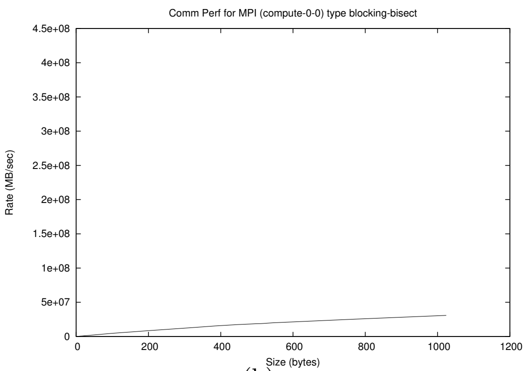


(f)

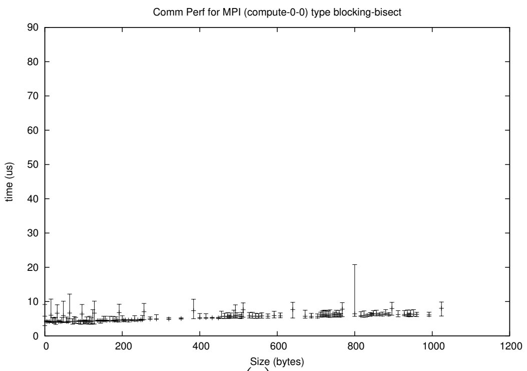
Figure 7: shortv – Cluster (i) lam gcc2.96 c0-19 (a,b) latency and bandwidth for 1 process/compute node (c,d) latency and bandwidth for 2 processes/compute node (e,f) latency and bandwidth for 4 processes/compute node



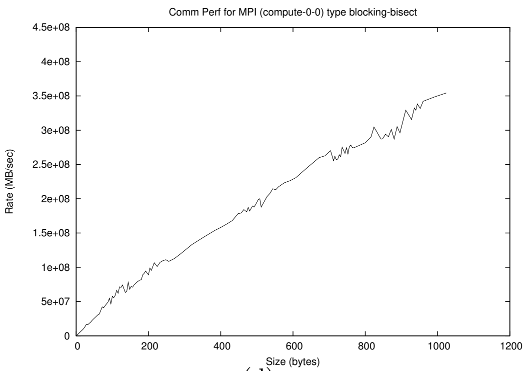
(a)



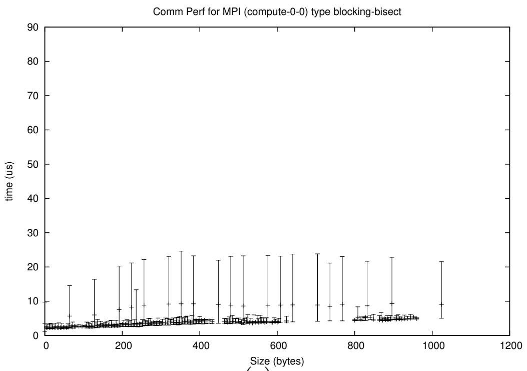
(b)



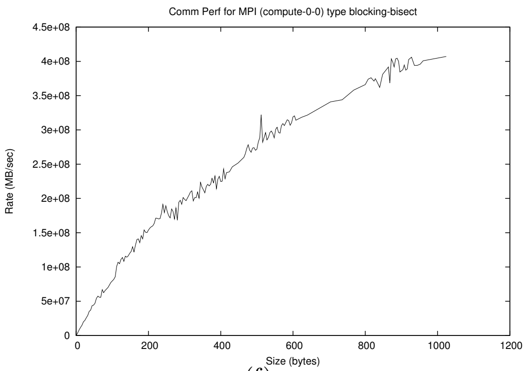
(c)



(d)

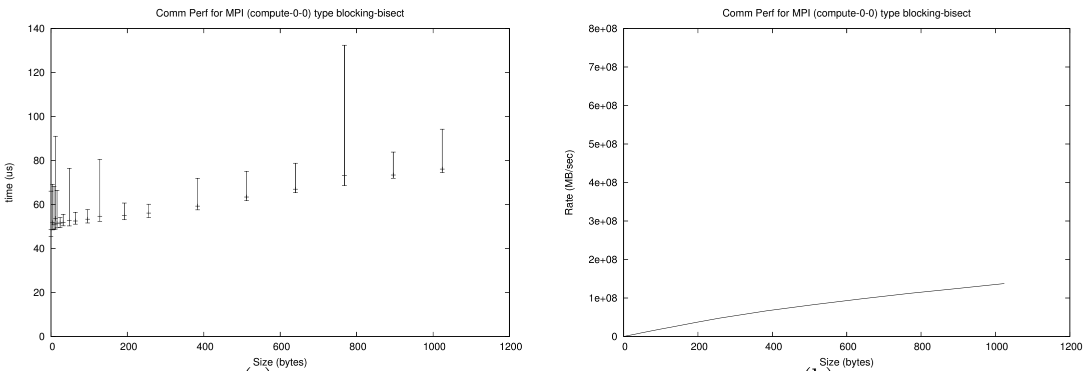


(e)

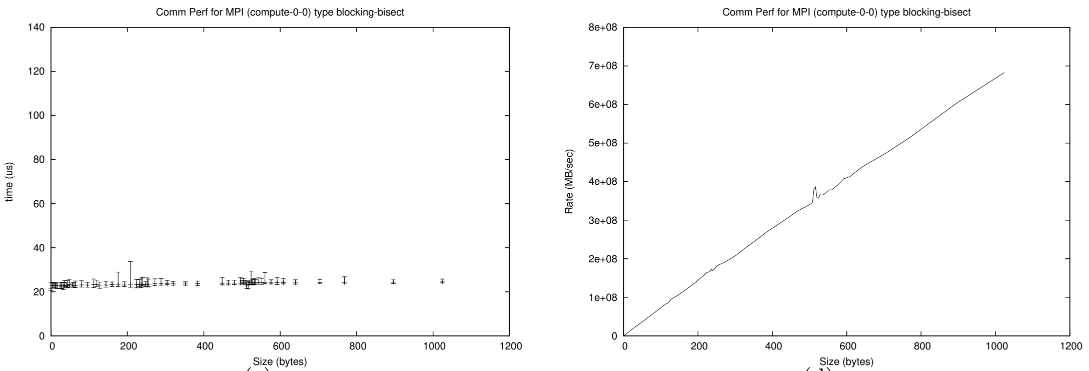


(f)

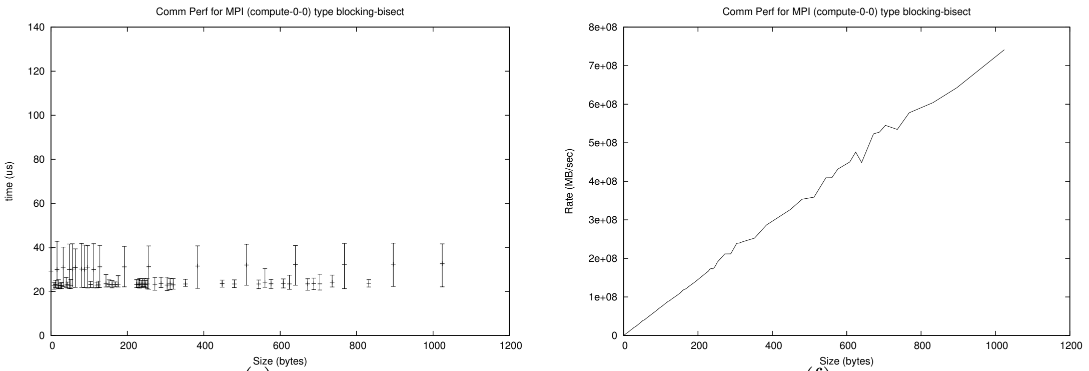
Figure 8: bshort4 – Cluster (i) lam gcc2.96 c0-19 (a,b) latency and bandwidth for 1 process/compute node (c,d) latency and bandwidth for 2 processes/compute node (e,f) latency and bandwidth for 4 processes/compute node



(a) (b)

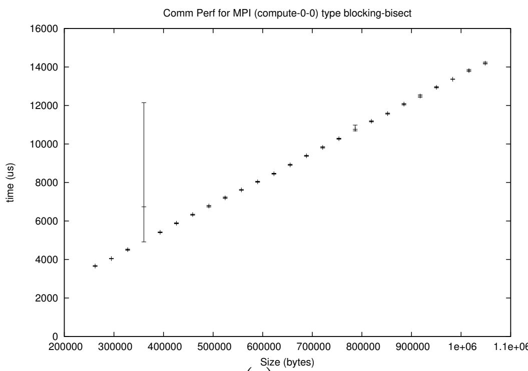


(c) (d)

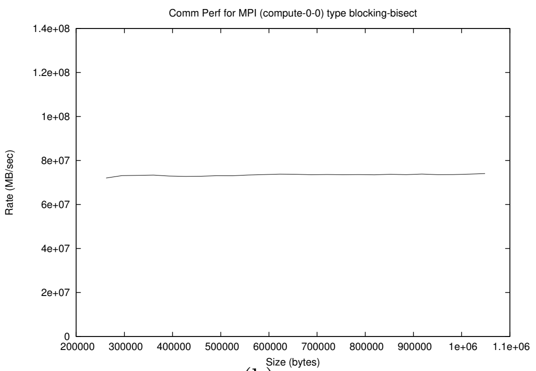


(e) (f)

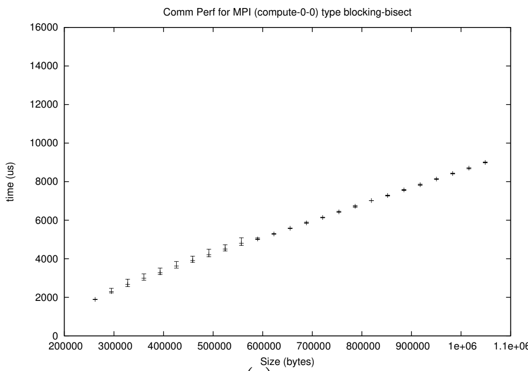
Figure 9: bshortX – Cluster (i) lam gcc2.96 c0-19 (a,b) latency and bandwidth for 1 process/compute node (bshort20) (c,d) latency and bandwidth for 2 processes/compute node (bshort32) (e,f) latency and bandwidth for 4 processes/compute node (bshort32)



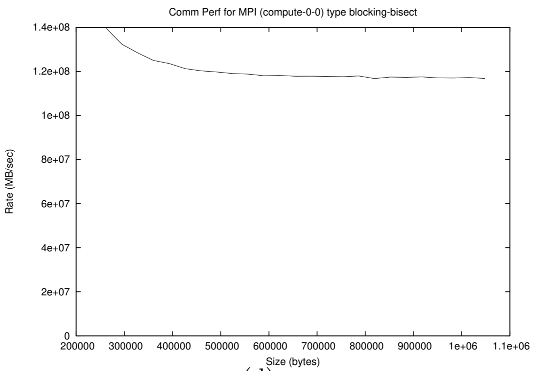
(a)



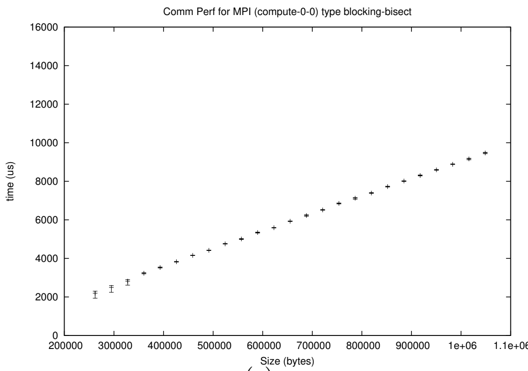
(b)



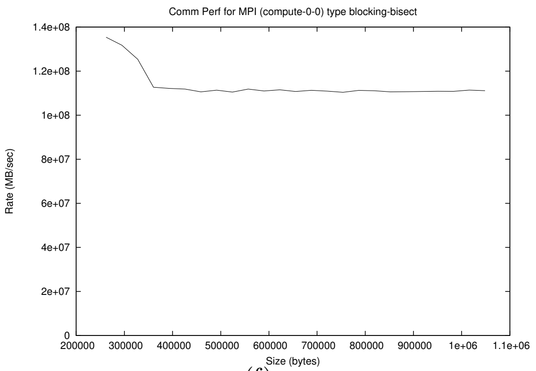
(c)



(d)

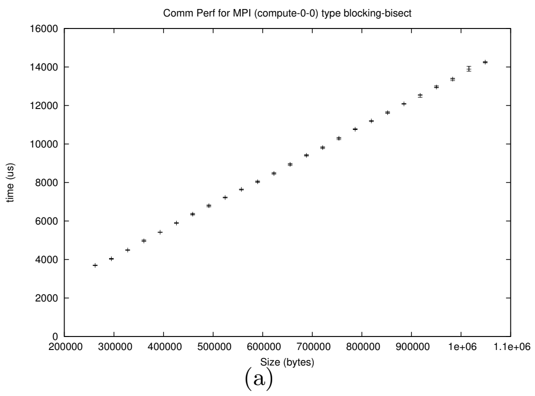


(e)

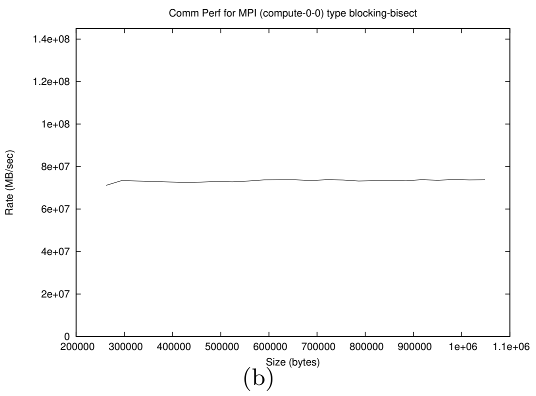


(f)

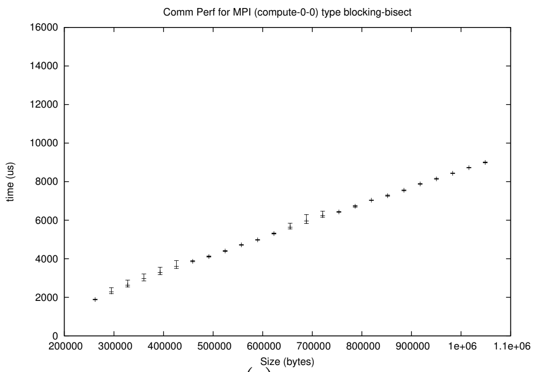
Figure 10: blong4 – Cluster (i) lam gcc2.96 c0-19 (a,b) latency and bandwidth for 1 process/compute node (c,d) latency and bandwidth for 2 processes/compute node (e,f) latency and bandwidth for 4 processes/compute node



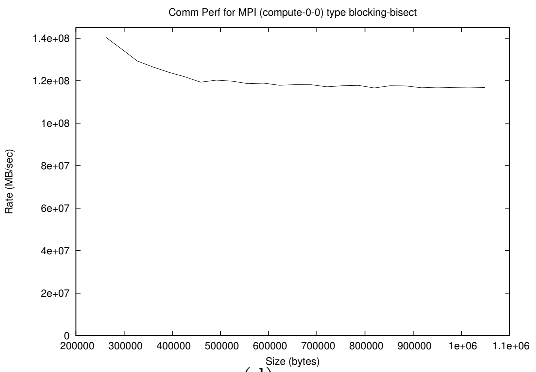
(a)



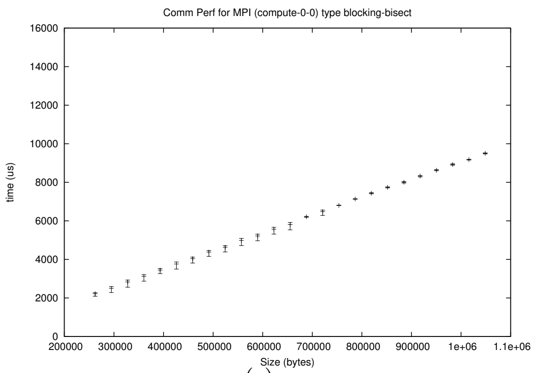
(b)



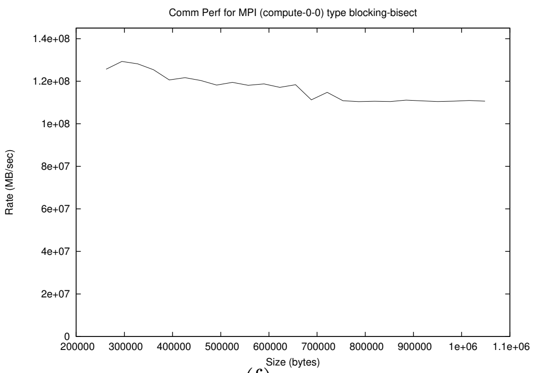
(c)



(d)

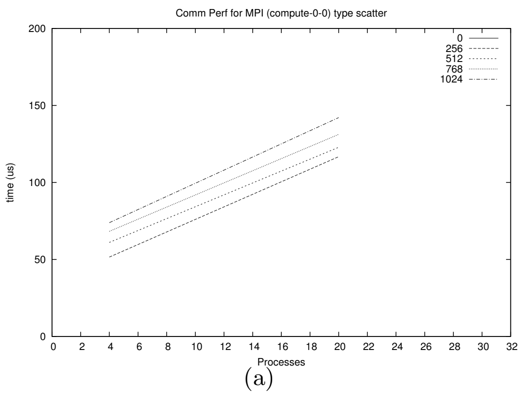


(e)

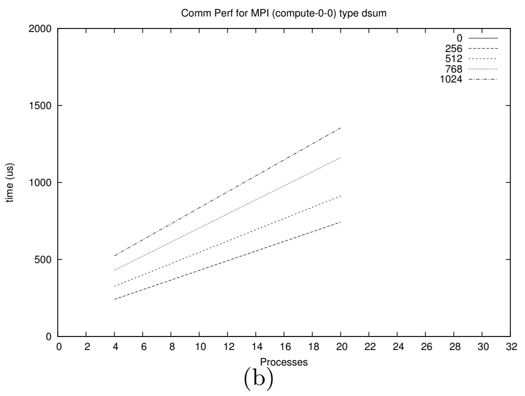


(f)

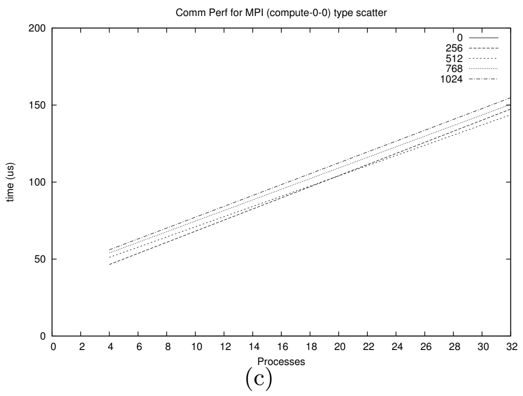
Figure 11: blongX – Cluster (i) lam gcc2.96 c0-19 (a,b) latency and bandwidth for 1 process/compute node (blong20) (c,d) latency and bandwidth for 2 processes/compute node (blong32) (e,f) latency and bandwidth for 4 processes/compute node (blong32)



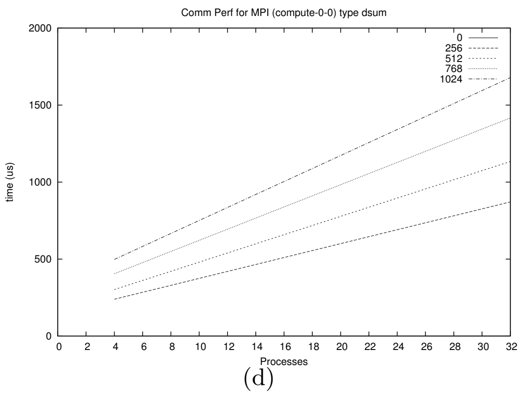
(a)



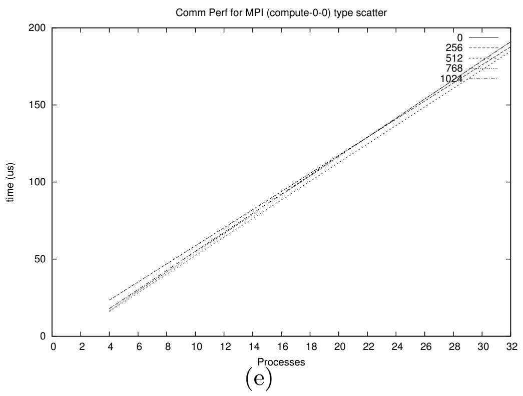
(b)



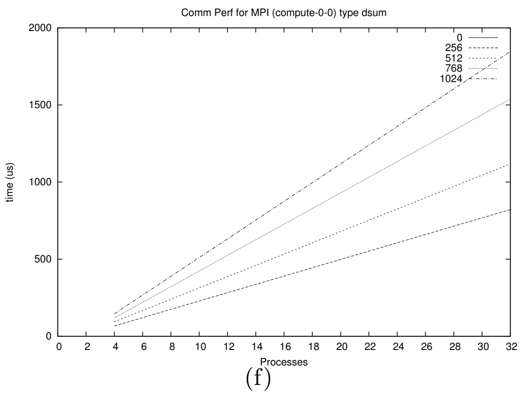
(c)



(d)



(e)



(f)

Figure 12: Cluster (i) lam gcc2.96 c0-19 (a,c,e) bcast 1,2,4 processes/compute node respectively (b,d,f) dsum 1,2,4 processes/compute node respectively

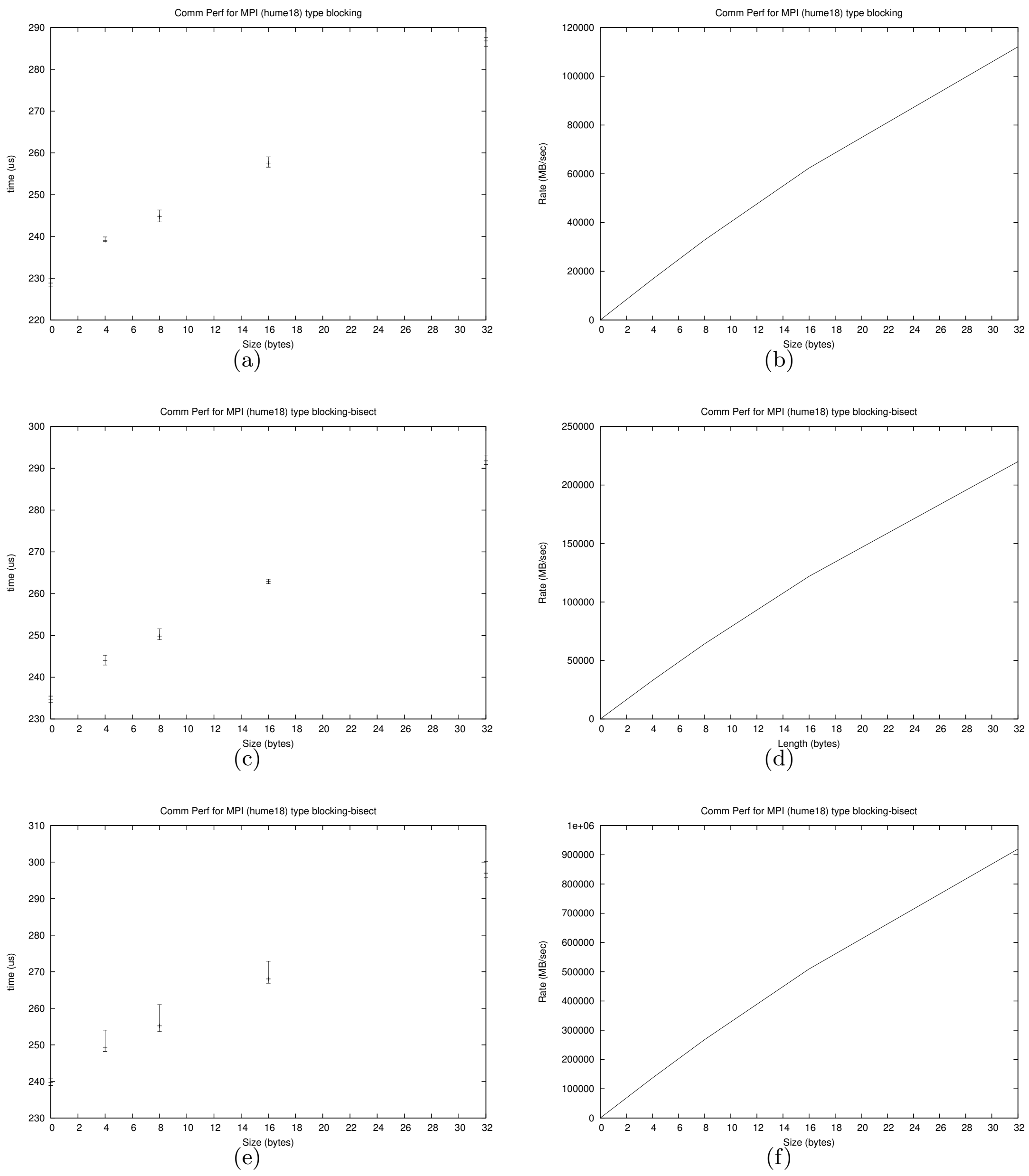
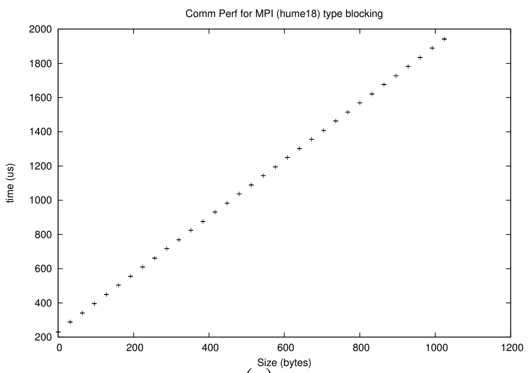
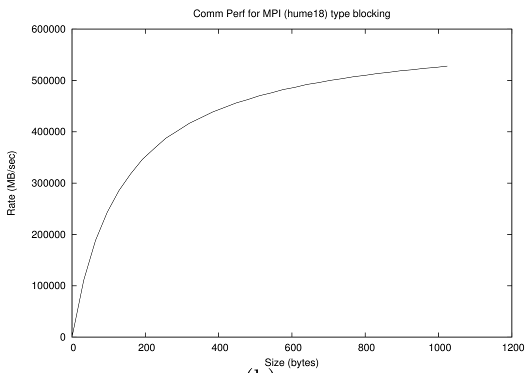


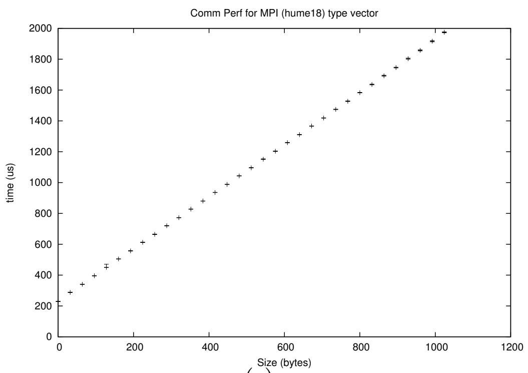
Figure 13: Cluster (ii) lam gcc3.2 c0-17 (1 process/compute node) (a) short – latency (b) short – bandwidth (c) bshort4 – latency (d) bshort4 – bandwidth (e) bshort17 – latency (f) bshort17 – bandwidth



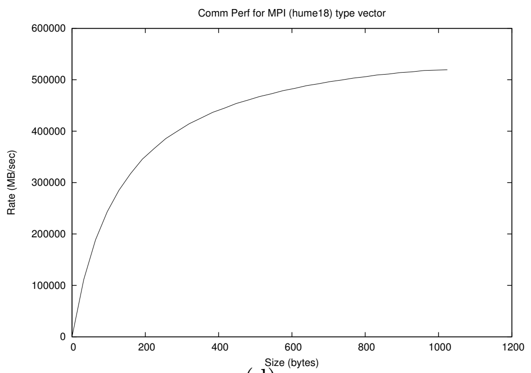
(a)



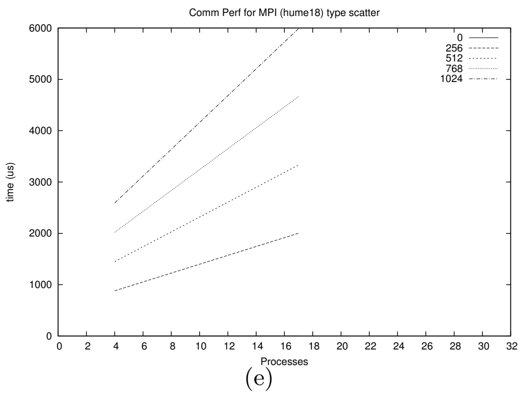
(b)



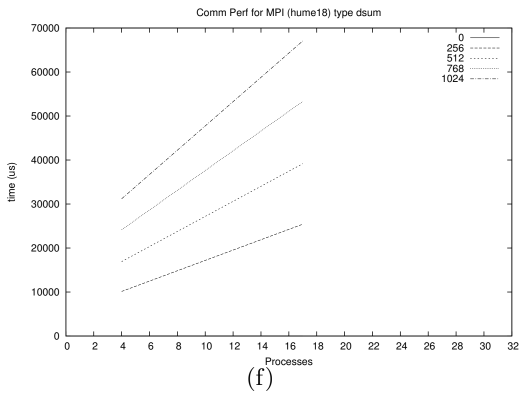
(c)



(d)



(e)



(f)

Figure 14: Cluster (ii) lam gcc3.2 c0-17 (1 process/compute node) (a) shortc – latency (b) shortc – bandwidth (c) shortv – latency (d) shortv – bandwidth (e) bcast – latency (f) dsum – latency