Communication Performance Issues for Two Cluster Computers

Francis A. Vaughan **Duncan A. Grove** **Paul D. Coddington**

Department of Computer Science
University of Adelaide,
Adelaide, SA 5005, Australia
Email: {francis,duncan,paulc}@cs.adelaide.edu.au

Abstract

Clusters of commodity machines have become a popular way of building cheap high performance parallel computers. Many of these designs rely on standard Ethernet networks as a system interconnect. We have profiled the performance of some standard message passing communication on commodity clusters using MPIBench, a tool for benchmarking the performance of MPI routines that uses a highly accurate, globally synchronised clock. The results suggest that existing methodologies of performance characterisation are inadequate. Tests were performed on two clusters, one with a conventional network architecture of switches connected via a high bandwidth backbone, the other with a tetrahedral network topology that potentially provides for lower contention and higher bandwidth. Where packet loss does not occur, performance in either system is good and degrades smoothly with load. However, packet loss is found to occur at any load and the consequent invocation of the TCP/IP timeout and congestion control mechanisms affect performance to a much greater than expected level. The nature of many parallel programs causes overall performance to drop to the worst case rather than the average. The value of MPIBench in profiling communication in parallel systems is clearly demonstrated, particularly through its generation of probability distributions which allow detailed analyses of performance problems.

Keywords: Communication networks, MPI, benchmarking, performance analysis, cluster computing, parallel computing.

1 Introduction

High performance computer design has always required a balance between three factors: raw compute power, memory size, and I/O capacity. The advent of parallel, and particularly distributed parallel systems, has added communication capacity to this mix. As commodity computer systems have become more powerful and cheaper, this technology has been adapted for use in distributed memory supercomputers. The 1990's exemplified such machines, for example Thinking Machines' CM-5 and Cray's T3-E. The advent of reasonably high performance commodity communication technology (in the form of Ethernet) has allowed the construction of distributed memory parallel computers entirely from off-the-shelf components. Such machines have become known as Beowulf clusters (Ridge, Becker, Merkey & Sterling 1997).

Despite the relative success of Beowulf clusters, communication performance has not kept pace with processor power. Although 100Mb/s Ethernet is most commonly used for networking Beowulf clusters, more expensive technologies, such as Myrinet (Boden, Cohen, Felderman, Kulawik, Seitz, Seizovic & Si 1995) or QsNet (Petrini, Coll, Frachtenberg & Hoisie 2002), are often used. Key aspects of these more expensive technologies are their low latency, high bandwidth and ability to scale well to large numbers of nodes. Some innovative Ethernet-based networking topologies have also been devised to improve performance in Beowulf clusters without incurring the significant expense of using these more advanced network (Reschke, Sterling, Ridge, Saverese, Becker & Merkey 1996).

In parallel computers built from specialised network interconnects, the underlying hardware is designed to provide highly optimised and reliable communication and synchronisation primitives. Beowulf clusters built using Ethernet, however, typically rely upon TCP/IP to provide reliable communication. TCP/IP was mostly designed for use in long haul networks, not tightly coupled networks. As we will see, the standard mechanisms for coping with congestion and packet loss are not well-adapted to use in a cluster, and can sometimes lead to very poor performance.

In order to understand how congestion and packet loss affect performance, it is important to measure the distribution of communication times, rather than average communication time. To achieve this, we have developed MPI benchmarking software, called MPIBench (Grove 2001), that provides very fine grained timing of MPI communication events. In particular, MPIBench can measure the probability distribution function of the intrinsic communication mechanisms and also find the distribution of outlier events. Such timings yield insight into the interaction of the various components of the communication system, and highlight significant performance characteristics that are hidden in more simplistic tests.

The remainder of this paper compares the performance that can be obtained using the conventional approach to networking commodity clusters with the performance that can be obtained using an innovative networking topology. It describes some MPIBench tests that were carried out on two different cluster computers, and draws some conclusions about the true performance of each system.

2 Cluster Computer Architectures

The two clusters profiled in this paper were the first two large computational clusters commissioned in Australia. They became operational at about the same time, in early 2000. They are built from very similar commodity components, but differ in their communication topology.

Copyright ©2003, Australian Computer Society, Inc. This paper appeared at the Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 16. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

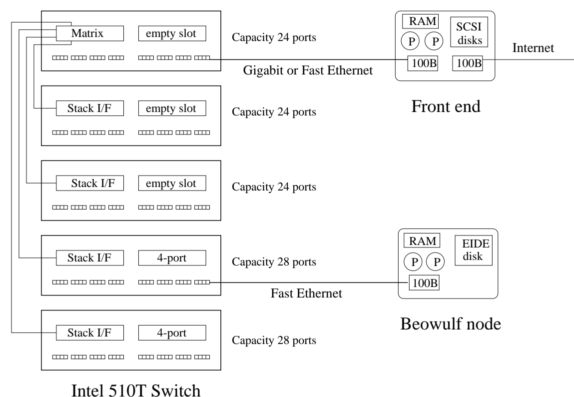

Figure 1: Network infrastructure for the Perseus Beowulf. Each Beowulf node has two Pentium processors (labelled P in the diagram).

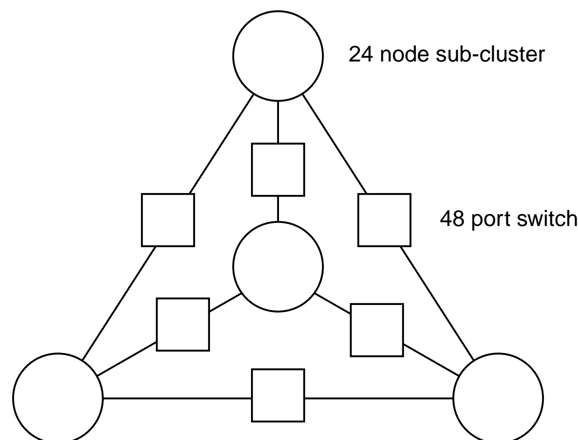

Figure 2: Network topology of the Bunyip cluster.

2.1 Perseus

The Perseus cluster (Hawick, Grove, Coddington & Buntine 2000) is an example of a typical Beowulf cluster. Developed as a dedicated resource for computational chemistry, it is comprised of 116 dual processor nodes, mostly 500 MHz Pentium III processors, connected by a stack of five Intel 510 24-port Fast Ethernet switches. These switches are connected by a proprietary high speed link of 2.1 Gb/s per switch through a Matrix module seven port backplane, as shown in Figure 1. The relevant software environment on Perseus consists of Linux version 2.2.12 and MPICH version 1.2.0.

2.2 Bunyip

The Bunyip cluster (Aberdeen, Baxter & Edwards 2000) consists of 96 dual processor nodes, originally 550 MHz Pentium III processors, although many have since been upgraded to 800 MHz. Our performance tests were mainly done on the nodes with 550 MHz processors to allow a better comparison with Perseus. Bunyip's communication network is constructed from Hewlett Packard ProCurve 4000M Fast Ethernet switches. The relevant software environment on Bunyip consists of Linux version 2.4.18-3 and MPICH version 1.2.0. This version of the Linux kernel has a slightly more efficient network and shared memory implementation than the version of the Linux kernel on Perseus.

Bunyip uses a novel interconnection architecture. The design is based upon two insights:

1. A switch may connect two sub-clusters, where the number of nodes in each sub-cluster equals half the number of ports available on the switch.
2. With multiple network interfaces per node, sub-clusters can be arranged in a fully connected mesh

In particular, the Bunyip cluster is implemented as a tetrahedron of four 24 node sub-clusters, connected by six 48-port switches. Each node has 3 network interface cards which allow it to connect to any other node via a single switch. This topology, shown in Figure 2, scales to large systems, but its equipment requirements grow quite quickly. For a system with p sub-clusters, each node requires $p-1$ network interfaces, and $p(p-1)/2$ switches.

Bunyip's tetrahedral topology avoids any bottleneck in network bandwidth that might occur with a central backplane (such as is used in Perseus). Since each node has multiple network interfaces the opportunity exists to increase the effective bandwidth between nodes in a manner similar to channel bonding. However this requires messages to be relayed through intermediate nodes in addition to transiting through an interconnecting switch. Such messages will suffer a significant additional latency since they must traverse part of the IP stack in the intermediate node and may introduce extra load on that node. However with careful design of group communication some success might be expected. Some efforts to exploit this are detailed in (Tan & Strazdins 2002).

3 Existing MPI benchmarking techniques

All previous performance benchmarking tools for MPI systems (Anglano 2001) suffer from one or more of three main inadequacies. The first inadequacy is the use of relatively coarse grained clocks for timing measurements. This forces benchmarks to average results over a high number of test repetitions, thereby losing detailed performance insight. In particular, none of the current synthetic benchmarks can generate distributions that show the variability in completion times and the outliers that occur, both of which may have a large effect on program performance. Moreover, this important factor has been almost completely ignored in the literature; Mraz (Mraz 1994) and Tabe *et al.*'s (Tabe, Hardwick & Stout 1995) statistical examinations of communication time on the IBM SP2 seem to be the only significant works that have made an attempt to quantify its importance. Tabe *et al.*'s main conclusions were that "it must be emphasised that using only means to model communication performance of parallel computers is inadequate" and that "understanding the tail behaviour of relevant distributions is critical to the development of good simulators".

Unfortunately, this praise-worthy advice appears to have been, until now, either unheard or unheeded. The assumption that a mean is useful implicitly assumes that the distribution is simple and well behaved. In a parallel program the impact of delaying a single node will cascade. For instance, if computation proceeds through a set of barrier synchronisation operations it does not matter that the majority of participating nodes are released from the barrier quickly. If one node is delayed, it will be unable to complete its share of the computation by the time other processors have reached the next barrier. Thus the entire distributed computation is delayed to the

same extent as if every process involved in the barrier synchronisation was to take as long as the slowest process. This cascading effect is exhibited in most group operations, since most implicitly require some level of synchronisation.

Existing benchmarks either rely on MPI's internal time-keeping function to provide a globally synchronised clock; or simply use ping-pong type tests to measure the total round-trip time, often with only one pair of processors, which avoids problems with contention. While relying on `MPI_Wtime` may be reasonable for high end systems that have a hardware-synchronised global clock, it is not acceptable on low end cluster systems where the accuracy of `MPI_Wtime` is bounded by, at best, half of the round-trip time of a zero byte message. Alternatively, measuring only round-trip message-passing times negates the need for a globally synchronised clock, but it does not allow the direct measurement of the performance distribution of individual message-passing calls that begin at one process and complete at another. In the case of round-trip point-point messages, only the convolution of the completion time of two messages can be measured, which tends to broaden the distribution observed. Further, in the case of collective communication, it is almost impossible to measure anything significant other than completion time at the root process - for example, insight into the actual completion times of collective calls seen by individual processes.

4 Measuring Communication Performance

As described earlier, MPIBench is able to make very accurate measurements of standard MPI operations. The key to MPIBench is the use of a very high resolution distributed clock. This relies on the availability of a cycle counter, which is available on most modern CPUs, for local time measurements on each processor. These local clocks are used to construct a global reference clock, given two assumptions: firstly, that over a useful time period, each local clock is relatively stable, and that there is a hard minimum transit time for point-to-point communication.

Each node implements a local, free running clock that is related to the master clock on the lead node by an offset and a rate factor. The lead node synchronises a peer by sending the peer messages that contain the value of its clock. The peer node calculates the offset between its clock and that of the lead node, and replies. Successive exchanges of messages allow refinement of the calculated offset. Given a lower limit on the transit time, a message with close to the minimum flight time will eventually be received. The peer node uses the lowest flight time seen as its offset. A second synchronisation is needed to estimate the difference in clock rates between nodes; even if nodes contain identical hardware, their clocks can drift significantly due to the reaction of their clock crystals to small temperature changes. Further, modern CMOS processors draw significantly different amounts of power dependent upon the computation being performed. This can lead to small changes in the voltage of power rails and changes of the temperature within a node's chassis. Variations of the clock rate have even been observed to correlate with the cycling of a machine room's air-conditioning (Maillet & Tron 1995). Therefore, we found it necessary to perform two back to back test runs of each experiment, where the first was used as the calibration environment for determining relative clock speeds, while the second was used to actually measure results.

An alternative approach would be to post process the results to correct the time measurements. However, the precision and sensitivity of MPIBench requires careful scheduling of the tests, which made this impossible (due to causality). Initial measurements of group operations were found to be contaminated by a barrier synchronisation that was performed at the start of each trial. Since this operation does not actually complete at the same time on each node, the results were skewed by the distribution of completion times for the barrier synchronisation. Therefore, MPIBench's globally synchronised clock was used to start of each individual test; and hence the results could not be post-processed because the scheduling of individual tests themselves depended on the global clock. Each process was required each node to wait until a nominated global time before proceeding. The time between tests was selected to ensure that the network would be free of any residual messages that may cause unexpected congestion. The delay between individual measurements in the following tests was set to be 0.05 seconds, except for the large all-to-all tests where 0.25 seconds was used. Once synchronisation was achieved, it was found to remain useful for the duration of test runs, varying at a rate of about 0.5ms/hour. Very occasionally, synchronisation would not achieve the required accuracy, leading to systematic mis-timings for some trials. These mis-timings manifested themselves as a linear trend in completion times, which were easily observable and accounted for.

5 `MPI_Isend` Results

Point-to-point tests were run to characterise the performance of unstructured communication between co-operating processes. In these tests, pairs of processors exchanged messages freely. Processes were arranged to ensure an evenly spread load across the network. As the load of messages increased, contention effects in the switches became evident.

The communication time for point-to-point messages on Perseus and Bunyip are shown in Figures 3-10. The 2-dimensional plots show the average completion times of point-to-point tests versus message size, which rise steadily at a rate commensurate with the increased time required to send larger. messages. The different curves represent the different number of participating processes. Note that the switches used in Bunyip have 48 ports, while those used in Perseus have only 24 ports.

The switches used on Perseus show a significant decrease in performance as the amount of traffic in the network is increased. Also, Figure 5 shows the manner in which the variance of completion times increases as message size is increased. In contrast, even with a significant number of communicating processes, Bunyip's switches perform well, with only a marginal increase in completion times. This increase is independent of message size, and hence number of packets (since the largest message is less than the maximum Ethernet packet size), which indicates it is probably the result of increasing load on the packet routing system in the switch.

The same tests were run across Bunyip's tetrahedral network topology, and the results are plotted in Figures 8 and 9. The average communication times were slightly lower than those obtained using a single switch in Bunyip, probably because the traffic load was spread across multiple packet engines as opposed to a single one. Figure 10 shows a similar but smaller increase in completion time and variance on Bunyip's tetrahedral network topology compared to the results obtained on Perseus.

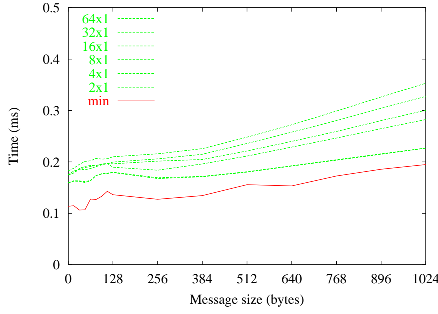

Figure 3: Average times for small `MPI_Isend` messages on Perseus.

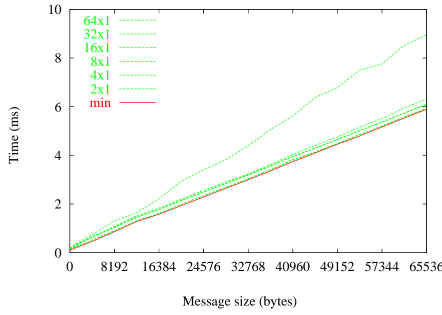

Figure 4: Average times for large `MPI_Isend` messages on Perseus.

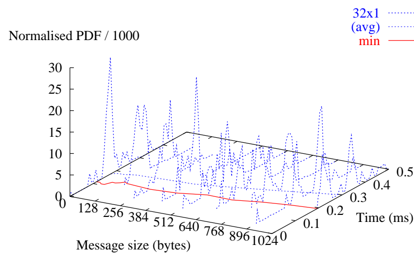

Figure 5: Distributions of times for `MPI_Isend` using small `MPI_Isend` messages on Perseus.

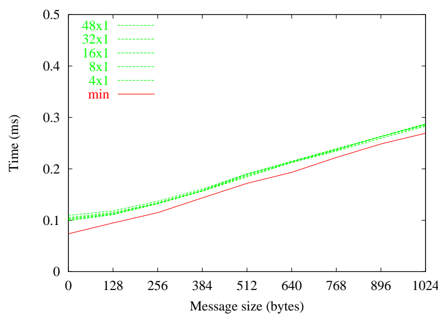

Figure 6: Average times for small `MPI_Isend` messages on Bunyip using only one switch.

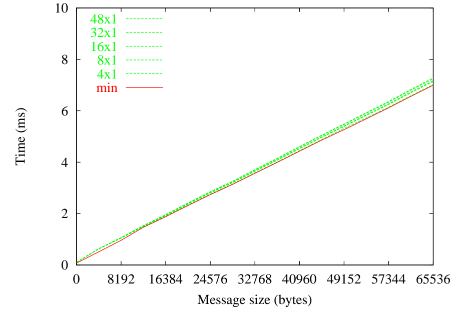

Figure 7: Average times for large `MPI_Isend` messages on Bunyip using only one switch.

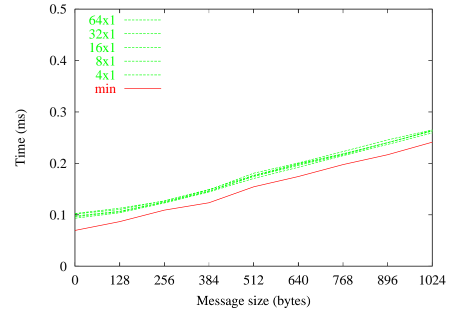

Figure 8: Average times for small `MPI_Isend` messages on Bunyip using its tetrahedral network topology.

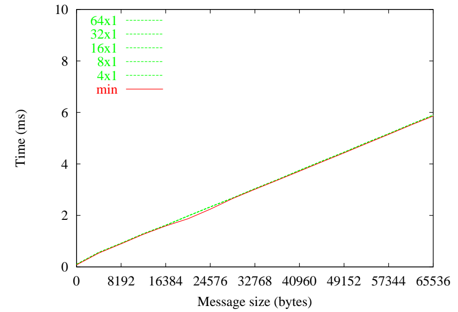

Figure 9: Average times for large `MPI_Isend` messages on Bunyip using its tetrahedral network topology.

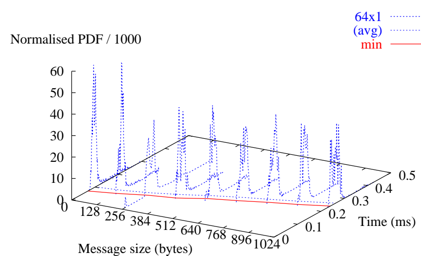

Figure 10: Distributions of times for `MPI_Isend` with small `MPI_Isend` messages on Bunyip using its tetrahedral network topology.

6 `MPI_Barrier` Results

Barrier synchronisation is a fundamental building block of parallel programs. Participant processes enter the barrier and wait until all the participants have entered the barrier, at which point the barrier is released and all the processes can proceed with the computation. Logically, the implementation of the barrier synchronisation can be viewed as nothing more than an `MPI_Sendrecv` operation where the lead process is the peer of every other process. In practice, to reduce the number of steps required, barrier synchronisations are implemented using a binary tree-based communication structure, where each process peers not (in general) with the lead process, but with another process, determined by its position in the tree. In MPICH 1.2.0, a `MPI_Barrier` is performed in two phases. A fan-in operation is used to collect messages from each participant at the root process and then a fan-out stage is used to send messages back out to each participant, signalling that the barrier has been released. The precise structure of the the fan-in/fan-out mechanism has important consequences for performance. The fan-in/fan-out implementation is based upon a binary tree. In the fan-out stage, the root process begins by sending a message to a participant process. Upon receipt, this process will send a copy of the message to another participant. At the same time the root process continues to send messages to other participating processes, who in turn forward their messages on. Half of the participant processes will not forward the message, since they are leaf nodes of the tree. The fan-in stage is identical to this, except that it is performed in reverse order. Figure 11 shows the message flow for an eight way barrier synchronisation. A process: 1) receives messages from its children; 2) performs a send/receive operation with its parent; and 3) sends messages to its children.

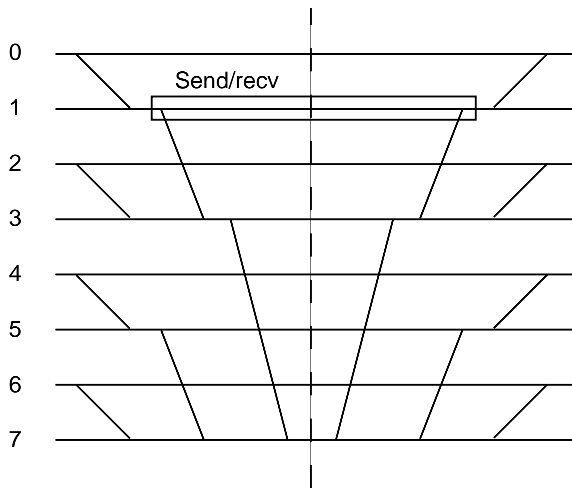

Figure 11: Message flow in an 8 process barrier synchronisation. The rectangle highlights two of the communication operations that can be paired using a send/receive operation.

Significant care is needed when measuring the performance of barrier synchronisations. To avoid contaminating the measurements with random or systematic offsets in start time, the participants must all be started at the same time. This was achieved using MPIBench's globally synchronised start option, described in section 4. Figure 12 shows that `MPI_Barrier` performance on Perseus is well behaved, with the completion time gradually increasing as

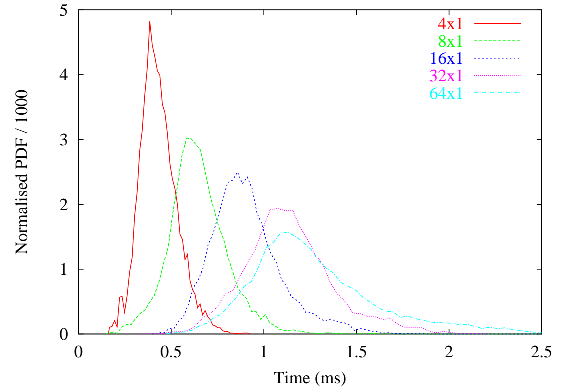

Figure 12: Distribution of times for `MPI_Barrier` on Perseus.

more processes participate. The extra time required to synchronise more processes is due to two effects. Firstly, the deepening of the binary tree results in a time overhead proportional to $\lceil log_2 n \rceil$, which can be observed as a steady increase in the minimum time to achieve a barrier synchronisation. Secondly, and the major contributor to increased completion time, is network contention. This seems almost entirely due to the rate at which the network switches can buffer, process and forward on incoming packets. Worthy of note, it can be seen that the average completion times for both 32 and 64 processes are almost the same, although the spread of times for 64 processes is wider. In the 32 process case, one switch was fully loaded while the other was only 1/3 loaded. In the 64 process case, two switches were fully loaded, while the other was 2/3 loaded. In this second case, the load was better distributed over the available packet processing capability, hence bucking the trend seen up until that point.

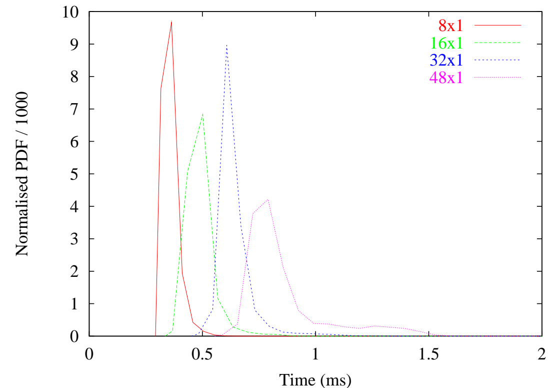

Figure 13: Main distribution of times for `MPI_Barrier` on Bunyip using only one switch.

Figure 13 shows the majority of measurements made for the `MPI_Barrier` test across one switch of Bunyip. Since any single participant can delay the progress of the barrier, either in the fan-in or fan-out stage, the completion time can be significantly extended. This can be seen in Figure 14, which shows the remainder of the events that were recorded for the same `MPI_Barrier` test on Bunyip. It can be seen that these outlier events cluster in the range of 30 to 40ms, compared to less than 1ms for the majority of the measurements, shown in Figure 13. There is

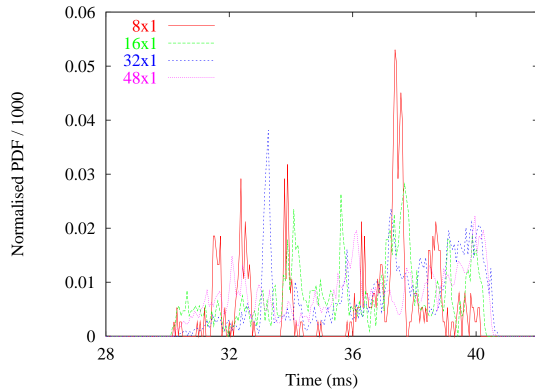

Figure 14: Distribution of outlying (retransmit) times for `MPI_Barrier` on Bunyip using only one switch.

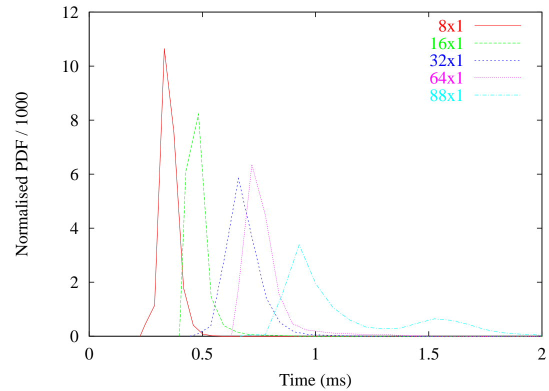

Figure 16: Main distribution of times for `MPI_Barrier` on Bunyip using its tetrahedral network topology.

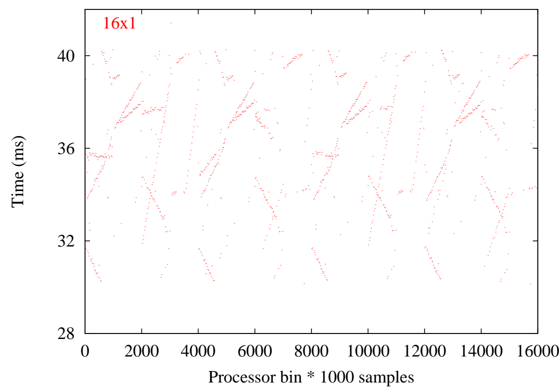

Figure 15: All sampled times for a 16 process `MPI_Barrier` on Bunyip using only one switch.

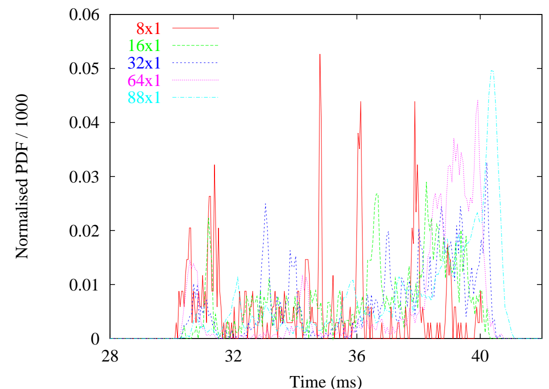

Figure 17: Distribution of outlying (retransmit) times for `MPI_Barrier` on Bunyip using its tetrahedral network topology.

clear evidence of some internal structure in the distributions.

Figure 15 presents the individual completion times for a 16 process barrier synchronisation on a single switch. The results for each process are arranged in blocks, from process 0 on the left to process 15 on the right. The results of 1000 individual trials are recorded in each block, also ordered from left to right. There are number of repeated patterns in this structure, that seem to be consistent with the location of the process in the broadcast tree. This pattern would appear to occur when timeouts occur on messages that form part of the broadcast tree. Such a timeout will delay one branch of the tree and cause a distribution of completion times that is offset by the time taken to reach that part of the broadcast tree before the timeout occurred. Thus the distribution of outlying events is primarily determined by which message in the broadcast tree was delayed.

The main completion time distributions and outlier distributions for the barrier on the tetrahedral network are plotted in Figures 16 and 17. The distributions are identical to those for a single switch. The distribution for the case of 88 participants shows a second slight peak at around 1.5ms which is probably because 88 is not a power of 2, resulting in an unbalanced binary tree. The distribution of outliers for these trials, again evenly distributed between 30 and 40ms, is indicative of the effects of singular retransmit timeouts.

7 `MPI_Alltoall` Results

A fundamental communication pattern, used for example in parallel sorting and Fast Fourier Transformations, is all-to-all data exchange. In this operation each process exchanges information with every other process. The results of three `MPI_Alltoall` operations are analysed in this section – one with a very small process to process payload (4 bytes), one with a medium sized payload (4096 bytes) and one with a relatively large payload (16K bytes). We used a fixed process to process message size, rather than fixed aggregate data size, because we wished to determine the effect of constant-size packets on the network switches.

An important aspect of the MPICH 1.2.0 implementation of `MPI_Alltoall` is its use of asynchronous messages. In principle, messages bound for hosts on different interfaces should be overlapped by the kernel, leading to maximum use of the multiple interfaces present in each of Bunyip's nodes. Tests were performed on both a single switch (up to 48 participating processes) and across the full tetrahedral network topology (up to 88 participating processes) to compare the relative performance of the flat and tetrahedral network topologies.

Figures 18 and 19 show the completion times observed for `MPI_Alltoall` operations using 4 byte messages. It can be seen that processes almost always

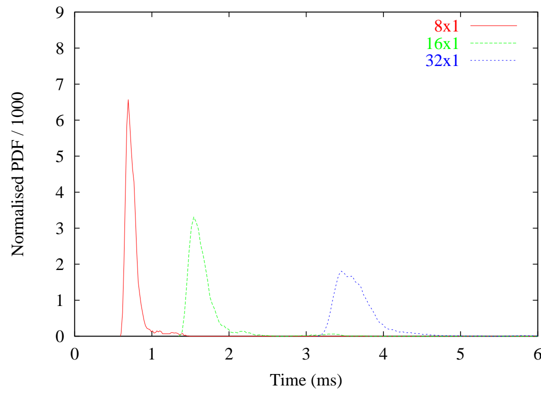

Figure 18: Distribution of times for a 4 byte `MPI_Alltoall` on Bunyip using only one switch.

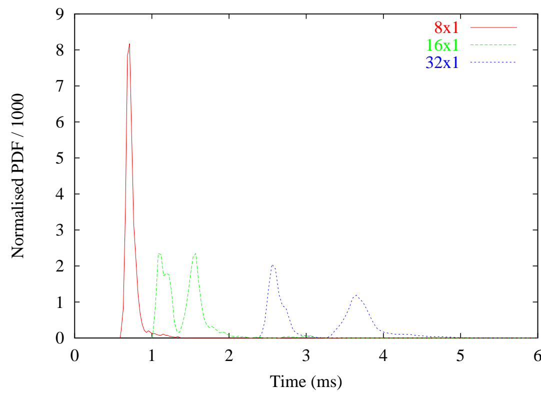

Figure 19: Distribution of times for a 4 byte `MPI_Alltoall` on Bunyip using its tetrahedral network topology.

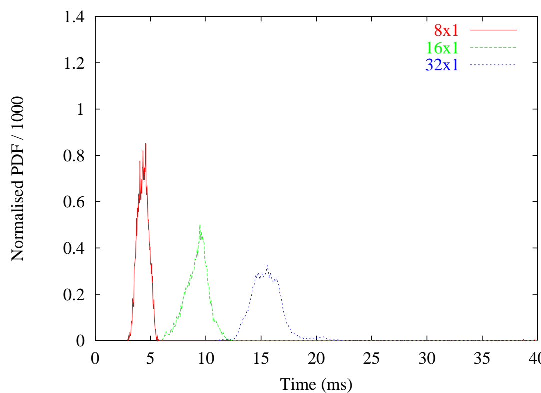

Figure 20: Distribution of times for a 4096 byte `MPI_Alltoall` on Bunyip using only one switch.

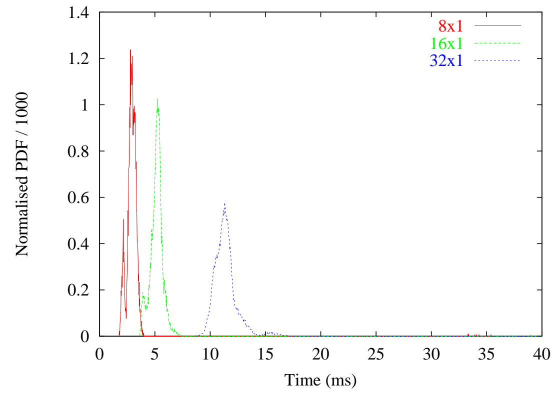

Figure 21: Distribution of times for a 4096 byte `MPI_Alltoall` on Bunyip using its tetrahedral network topology.

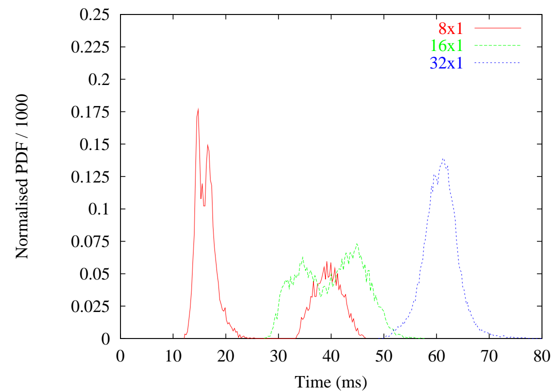

Figure 22: Distribution of times for a 16K byte `MPI_Alltoall` on Bunyip using only one switch.

complete quickly and the distributions are well behaved. There are, again, some outliers present in the 30 to 40ms range.

Figures 20 and 21 present histograms of completion times for messages of 4096 bytes on both flat and tetrahedral network topologies. This is especially revealing since it shows essentially perfect performance. The distributions are well behaved and the average times are commensurate with the time taken to send the payload. Note that the aggregate number of packets and data sent and received by each process are proportional to the total number of processes – hence the separation of results. It seems that payloads of this size decrease the packet rate seen by the switch enough to reduce the number of packet it drops, yet the bandwidth requirements are not so great as to run afoul of the congestion limiting algorithms that will be seen to affect larger message sizes.

Figures 22 and 23 present the results for 16Kbyte `MPI_Alltoall` tests. The results are strikingly different. The most important feature is the very large number of processes completing between 30 and 40ms. Again, this is the range of times commensurate with a process having been delayed by one packet retransmission timeout. For an `MPI_Alltoall` of 8 participating processes, the number of processes completing in the expected time of just under 20ms is almost equally balanced by processes completing after one timeout. When 16 processes participate the distributions of processes competing without packet loss and those suffering a timeout start to overlap. When 32 processes participate the actual expected time for completion is greater than a single timeout and the distribution becomes well behaved again. Due to the asynchronous message sending in the all-to-all implementation, a resent message does not delay the overall completion of the operation.

When the same tests were performed utilising the tetrahedral network topology the results were consistent with the single switch case. As can be seen,

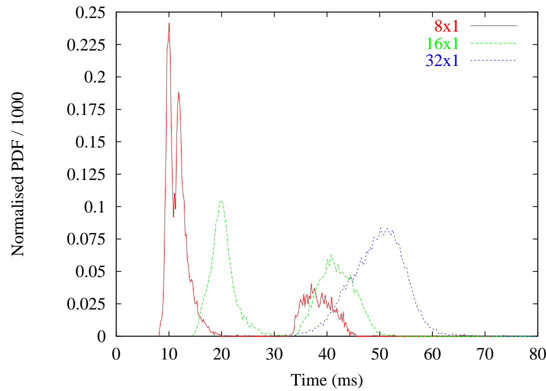

Figure 23: Distribution of times for a 16K byte `MPI_Alltoall` on Bunyip using its tetrahedral network topology.

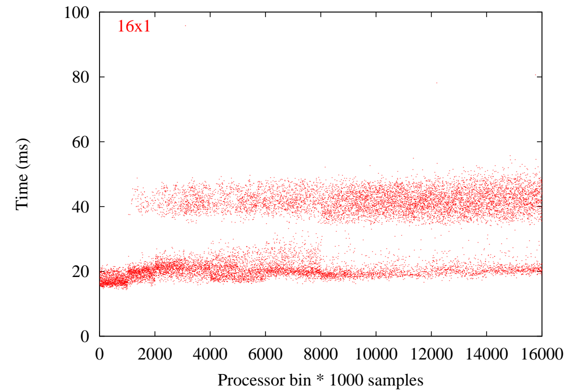

Figure 25: All sampled times for a 16K byte `MPI_Alltoall` using 16 processes on Bunyip using its tetrahedral network topology.

the tetrahedral communication network does allow some improvement in performance, and the completion times for processes that were not affected by timeouts were reduced by a little over 50%. However the proportion of processes affected by timeouts remained much the same. The change in completion times has the effect of separating the peaks for the 16 node case. The 32 node case remains as a single peak, but has improved from an average completion time of 60ms to 50ms.

Bunyip's performance in all-to-all tests are poor enough to be of significant concern. It is apparent that the onset of significant packet loss will limit any all-to-all communication, not by the intrinsic bandwidth of the network, but by the delay incurred by retransmitting lost packets.

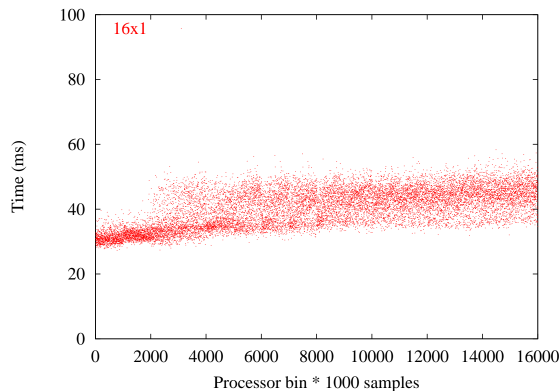

Figure 24: All sampled times for a 16K byte `MPI_Alltoall` between 16 processes on Bunyip using only one switch.

Figures 24 and 25 show the distribution of completion times for 16 nodes performing an all-to-all. The distribution of completion times for both a single switch case and for a Tetra network are striking. It can be seen that in both cases process zero completes quickly and with almost no delays. Each successive process is subject to increasing numbers of timeouts in its distribution of completion times, until in the case of the tetra network the number of trials affected by a timeout exceeds those that do not.

These distributions are indicative of a number of issues in switch performance. The switches used in Bunyip have individual input buffers of 512K bytes per port. The total of all messages sent by each process is only 256K bytes, and thus the port buffers will never overrun, eliminating one possible source of message loss. However, The MPICH implementation of the all-to-all operation is somewhat lacking. It is constructed from a series of asynchronous sends, where each node sends to the participating nodes in order. Thus all the nodes send to process 0, then to process 1 and so on. Since the sending is asynchronous each process will send the messages as fast as it is able. This sequence of message arrival will act to change the switches operating conditions as the all-to-all operation proceeds. Initially each buffer will receive a 16K byte message directed to process zero. These messages can be drained at the rate of 100 Mb/s. By the time one message has arrived the buffers will have received the set of messages directed at process 1, which will in turn begin to drain. At each message arrival interval the switch will contain more messages up to a worst case proportional to $n^2/2$ messages. This peak occurs at the point where those messages destined for the last process have just been sent. There seems to be a clear correlation between the number of messages currently buffered in the switch to the packet loss rate. It is possible that this is due to increasing contention for access to the buffers between the packet engine, the output ports and the input port.

For both the single switch and tetrahedral network there are significant numbers of delayed messages. While the tetrahedral network reduces the effects of this and provides a useful decrease of latency through the reduction of serialisation, the dominant effect of TCP timeouts remains.

Figure 26 shows the performance of the tetrahedral network on a 64 way all-to-all of 16K byte messages. This operation has entered a different performance regime. It can be seen that the majority of processes complete at about 100ms, which is roughly the performance to be expected if a single 100Mb/s link was used to full effect. Quite a number of these complete in less than 100ms, but do not come close to the 33ms that could perhaps be expected if each network interface in every node was used to full effect. Also note that there are a large number of outlying completion times that are banded intervals at 30ms increments, which appear to have been the subject of multiple (exponentially backed-off) retransmit timeouts.

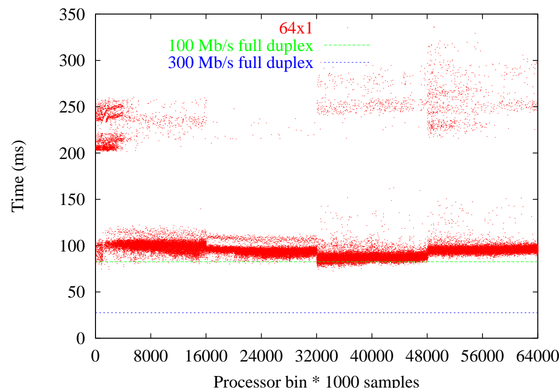

Figure 26: All sampled times for a 16K byte `MPI_Alltoall` using 64 processes on Bunyip using its tetrahedral network topology.

8 The effect of TCP/IP on Communication

The MPICH implementation of MPI assumes a reliable underlying communication protocol. Indeed MPI programs have traditionally been run over specialised high performance networks (such as that of the Cray T3D and T3E, or Myrinet or QsNet), which have hardware support for reliable delivery. In particular it has been possible to provide hardware detection of packet loss.

Since Ethernet is subject to packet loss, this mandates the use of a reliable protocol at a higher level. In general, Beowulf clusters utilise the standard TCP/IP protocol implemented in the Linux kernel. This has important ramifications, many of which stem from the fact that Ethernet is unable to detect or report packet loss. If a packet is lost it is assumed that the higher level network protocol will detect this and re-send lost data.

Delayed ACK

All messages are expected to be acknowledged, so that the sender is aware of safe receipt. Acknowledgement messages may be piggy-backed with other message traffic from the receiver to the sender if such traffic exists. However if no return message is available an ACK must be separately sent. In order to avoid too eager activity in sending ACK messages many TCP/IP stacks, including that in the Linux kernel, use a simple heuristic to delay the sending of ACK messages. Delaying an ACK may cause an unacceptable delay in sending messages in a distributed program, thus to avoid this MPICH uses a mechanism provided in the Linux kernel (the TCP_NODELAY socket option) to disable ACK delay.

This presence of ACK messages in the network is a source of additional congestion that is not typically considered. In a global synchronisation operation the constituent fan-in and fan-out operations have no intrinsic reverse message traffic. Thus there is no opportunity to piggy-back ACK messages on other traffic. It is possible the fan-out stage may be able to carry some of the ACK messages from the fan-in stage. However, in general, it might be expected that this is unusual. ACK traffic during the fan-out stage will typically have no option but to be discretely transmitted. Thus an additional 1.5 to 2 times the number of packets are generated during a global synchronise compared to what might be expected by a simple analysis of the basic algorithm.

Retransmit Timeout

In the face of a lost packet, and the subsequent non-arrival of the appropriate ACK message, the sender must eventually re-send the message. This timeout period for re-sending of a message is clearly a limiting factor on the performance of the network in the face of serious congestion. The value used for timeout is typically derived from an estimate of round trip time, and is also limited to a minimum time. The Linux kernels have a minimum timeout period of 200ms. However on Bunyip it is not clear that this minimum is used, indeed the results suggest that retransmit is occurring as soon as 30ms.

These timeout periods were chosen for systems that are intended to operate in wide area networks. A cluster system is almost always implemented using a local private network that is walled off from the global Internet. However, no single value for timeout is universally applicable in a cluster. Whilst aggressively small timeout periods might seem appropriate, in the face of very heavy network load, especially with very large data transfers, they may prove too short, with drastic performance consequences. If this is the case, a retransmitted packet will be lost again, and again, and the timeout between retransmissions will back off exponentially. Some level of adaptive control, or better, an API in which the application is able to provide broad guidance as to its instantaneous needs, is needed if the basic TCP/IP protocols are to achieve good performance in such circumstances. Alternatively a specialised protocol could be implemented on top of UDP.

Slowstart

TCP/IP uses a pair of interrelated algorithms to limit congestion in a network. These are *congestion avoidance* and *slow start* (Stevens 1994). Slow start is intended to limit the rate at which packets are injected into a network, and provide a mechanism by which a node can find an equilibrium rate that will not cause congestion. This is achieved by initially starting with a small limit on the aggregate size of messages that may be transmitted before they are acknowledged. As ACK messages are received this window is increased exponentially. The cycling of ACK messages is used as a throttle to control the injection of messages into the network. However if an ACK is not received before the timeout period expires the slow-start algorithm resets and returns the congestion window size to the original size. The large number of timeout events seen in the tests described suggest that the congestion window will continually be reset, and the slow start algorithm will constantly cycle. The very bursty nature of communication in a parallel environment makes it very difficult, if not impossible for the congestion window to settle on a useful size before a change in communication activity makes the value settled on obsolete.

9 Conclusions

The performance of a cluster using a conventional stacked switch architecture versus the tetrahedral network design of the Bunyip cluster has been evaluated. In those cases where the network switches perform without packet loss the design appears to offer some useful performance advantages. However it does not offer the speed improvements that might have been expected from the increase in network bandwidth. The reasons for this are manifold, including serialisation within the kernel of I/O operations, and

the intrinsic serial nature of the implementation of some MPI operations.

The high precision timings provided by the MPIBench suite have proven to be invaluable for analysing and comparing the two clusters. Most importantly, the ability to view the distribution of communications times is quite crucial in evaluating the communications performance of the clusters, and identifying problems and bottlenecks. Simplistic metrics used by other MPI benchmarks, such as a minimum time or simple mean, gloss over significant performance anomalies and may lead to a poor understanding of the real performance, and ignorance of important aspects of cluster design.

The MPIBench results clearly demonstrate the nature of congestion onset and the very poor mismatch of the standard TCP/IP algorithms and tuning parameters when used in a cluster environment. It is clear that considerable care must be taken in using Ethernet in a cluster system, and that the standard parameters for TCP/IP are inappropriate for clusters.

It must be emphasised that the congestion results for the Bunyip design are not intrinsic to the tetrahedral topology. It is interesting to note the the switches used in the Perseus cluster do not exhibit nearly the same level of packet loss, although in general they perform worse in terms of latency and throughput under load.

The architecture of the tetrahedral network does clearly stave off the onset of significant packet loss, and does offer measurable performance improvements. However to realise the full benefit the problems in the performance and operation of the underlying TCP/IP mechanisms must be addressed.

References

Aberdeen, D., Baxter, J. & Edwards, R. (2000), 98c/mflop, ultra-large-scale neural-network training on a piii cluster, *in* 'Proceedings of Supercomputing'.

Anglano, C. (2001), Cluster benchmarks web page. `http://www.di.unito.it/~mino/cluster/benchmarks/`.

Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N. & Si, W.-K. (1995), 'Myrinet – a gigabit-per-second local-area network', *IEEE Micro* **15**(1), 29–38.

Grove, D. A. (2001), Precise mpi performance measurement using MPIBench, *in* 'Proceedings of HPC Asia'.

Hawick, K. A., Grove, D. A., Coddington, P. D. & Buntine, M. A. (2000), 'Commodity cluster computing for computational chemistry', *Internet Journal of Chemistry* **3**, article 4.

Maillet, E. & Tron, C. (1995), 'On efficiently implementing global time for performance evaluation on multiprocessor systems', *Journal of Parallel and Distributed Comnputing* **28**(1), 84–93.

Mraz, R. (1994), 'Reducing the variance of point-to-point transfers for parallel real-time programs', *Parallel and Distributed Technology* **2**(4), 20–31.

Petrini, F., Coll, S. C., Frachtenberg, E. & Hoisie, A. (2002), 'Performance evaluation of the quadrics interconnection network', *Journal of Cluster Computing* .

Reschke, C., Sterling, T., Ridge, D., Saverese, D., Becker, D. & Merkey, P. (1996), A design study of alternative network topologies for the Beowulf parallel workstation, *in* 'Proceedings of the 5th International Symposium on High Performance Distributed Computing'.

Ridge, D., Becker, D., Merkey, P. & Sterling, T. (1997), Beowulf: Harnessing the power of parallelism in a Pile-of-PCs, *in* 'Proceedings of IEEE Aerospace'.

Stevens, W. R. (1994), *TCP/IP Illustrated, Volume 1*, Addison-Wesley.

Tabe, T. B., Hardwick, J. & Stout, Q. F. (1995), 'Statistical analysis of communication time on the IBM SP2', *Computing Science and Statistics* **27**, 347–351.

Tan, W. B. & Strazdins, P. (2002), The analysis and optimization of collective communications on a beowulf cluster, *in* 'Proceedings of the International Conference on Parallel and Distributed Systems'.