TCP Adaptation for MPI on Long-and-Fat Networks

Motohiko Matsuda, Tomohiro Kudoh, Yuetsu Kodama, Ryousei Takano
Grid Technology Research Center
National Institute of Advanced Industrial Science and Technology

Yutaka Ishikawa
University of Tokyo, and
Grid Technology Research Center
National Institute of Advanced Industrial Science and Technology

Abstract

Typical MPI applications work in phases of computation and communication, and messages are exchanged in relatively small chunks. This behavior is not optimal for TCP because TCP is designed only to handle a contiguous flow of messages efficiently. This behavior anomaly is well-known, but fixes are not integrated into today's TCP implementations, even though performance is seriously degraded, especially for MPI applications. This paper proposes three improvements in the Linux TCP stack: i.e., pacing at start-up, reducing Retransmit-Timeout time, and TCP parameter switching at the transition of computation phases in an MPI application. Evaluation of these improvements using the NAS Parallel Benchmarks shows that the BT, CG, IS, and SP benchmarks achieved 10 to 30 percent improvements. On the other hand, the FT and MG benchmarks showed no improvement because they have the steady communication that TCP assumes, and the LU benchmark became slightly worse because it has very little communication.

1 Introduction

TCP/IP is widely used as the transport layer of many MPI communication library implementations for commodity clusters and the Grid environment. However, the traffic of MPI applications is non-optimal for the TCP protocol due to the following characteristics.

Typical MPI applications use various communication patterns, one-to-one, all-to-all, and so on. This causes the available network bandwidth per node to change quickly. TCP only monitors the network behavior per connection and does not know the overall traffic patterns of the application. If TCP can observe and adjust itself to communication patterns, TCP will be able to effectively utilize the available bandwidth. Thus, in this paper, we designed a parameter switching mechanism in which the network bandwidth parameter is saved and restored at changes of communication patterns.

Secondly, MPI applications work in phases of computation and communication, and thus, the message flow is not contiguous. TCP is basically designed for a contiguous flow of messages, and it is not efficient for non-contiguous communication. When no packets have been sent for a long period of time in the TCP sense (i.e., longer than 200 msec), TCP changes to the Slow-Start state. This causes low bandwidth utilization, especially in long-and-fat networks. Due to the MPI application characteristics, TCP frequently changes to the Slow-Start state.

Moreover, the TCP Fast-Retransmit mechanism [1] does not work well against packet losses of MPI applications again, because it is designed for a contiguous flow of messages. When the last packet of a message is lost, which TCP assumes is a rare case, TCP cannot detect the loss by duplicate acknowledgements. Therefore, the sender has to wait for the Retransmit-Timeout time, and then recover from the loss of the packet. In most existing TCP implementations, such as Linux and Solaris, the Retransmit-Timeout time is set to hundreds of milliseconds. Therefore, this also causes low bandwidth utilization.

In order to overcome the issues of a non-contiguous flow of messages, two improvements have been designed. First, a pacing mechanism is employed so that TCP can start steady communication in the congestion avoidance state as soon as possible. Second, the TCP Retransmit-Timeout time is reduced to the value calculated based on the round-trip time. These modifications are made only to the start-up

Part of this research was supported by a grant from the Ministry of Education, Sports, Culture, Science and Technology (MEXT) of Japan through the NAREGI (National Research Grid Initiative) Project.

behavior, and no changes are made to the behavior of steady communication.

The proposed improvements are implemented in the Linux TCP stack and evaluated. Evaluation using the NAS Parallel Benchmarks shows that the BT, CG, IS, and SP benchmarks achieved 10 to 30 percent improvements. The FT and MG benchmarks showed no improvement because they have the steady communication that TCP assumes, and the LU benchmark becomes slightly worse because it has very little communication.

In the following paper, we first discuss the basic behavior of TCP and its problems in Section 2, and then present our three small improvements in Section 3. Then, the results of our evaluation are shown in Section 4. We briefly mention related work in Section 5, then we conclude the paper in Section 6.

2 TCP Problems with MPI Traffic

2.1 Available Bandwidth Parameter Mismatch

TCP monitors basic network parameters, such as latency and bandwidth, and controls the amount of sending messages using those parameters [8]. One important parameter is *cwnd* (congestion window size), which is the maximum amount of in-flight messages sent during the round-trip time, and roughly corresponds to the bandwidth-delay product of the connection.

An MPI application generates differing traffic in various communication patterns. For example, let us assume an MPI application which uses both one-to-one and all-to-all communication primitives, and that it is run on two clusters connected by a wide-area network. In the one-to-one communication phase, one node can consume the whole of the bandwidth, and *cwnd* becomes large. On the other hand, in the all-to-all collective communication phase, the participating nodes must share the bandwidth, and *cwnd* becomes small. Starting one-to-one communication immediately after all-to-all results in a small *cwnd* and traffic is reduced. Starting all-to-all communication immediately after one-to-one results in a large *cwnd* and congestion occurs. That is, since TCP needs multiple round-trip time to find out a proper *cwnd* value, TCP cannot adapt to such a quick change of the traffic in phase changes from one-to-one to all-to-all, and vice versa, happening as computation progresses.

2.2 Frequent Slow-Start

At the beginning of the TCP traffic, the Slow-Start mechanism is used to let the sender initially probe for the available bandwidth to avoid sending at a high rate, which may overload the network. Slow-Start starts with the initial minimum *cwnd* (one or two packets) and increments *cwnd* by one MSS (Maximum Segment Size) at each receipt of an acknowledgement. Slow-Start ends when *cwnd* reaches a predetermined threshold value, *ssthresh*. In Slow-Start, *cwnd* grows at an exponential rate, but it takes a long time to reach *ssthresh* in a long-and-fat network, because each step is triggered by a receipt of an acknowledgement. Thus, the effective bandwidth is very low during Slow-Start [2].

TCP also enters the Slow-Start state after a long quiescent state to avoid generating burst traffic in the absence of ACK-clocking. In other words, a long quiescent state makes TCP behave as if it started a new connection and it enters the Slow-Start state, which resets *cwnd* to the minimum value. This frequently happens in MPI applications, because they work in phases of computation and communication.

2.3 Retransmit-Timeout

The receiver observes the sequence numbers of the received packets, to reassemble the packets in order. When there is a hole in the sequence, the receiver recognizes a packet loss, and notifies the sender by a duplicate acknowledgement. This mechanism is called Fast-Retransmit [1]. The Fast-Retransmit mechanism works only if the traffic is contiguous. So, when no following packets arrive after the lost one because the sender has finished its data transmission, Fast-Retransmit cannot recover from the packet loss, since there is no hole in the sequence. In such a case, Retransmit-Timeout is detected by the sender passively, noticing it has not received the acknowledgement from the receiver [2].

An MPI application exchanges messages in chunks, and may wait for the end of a chunk. When the last packet of a chunk is lost, the loss cannot be recovered by the Fast-Retransmit mechanism, but can be recovered by the Retransmit-Timeout mechanism. Timeout, however, is relatively long by default (over 200 msec in the Linux implementation). This causes low utilization of the network.

3 Design and Implementation

3.1 Available Bandwidth Parameter Switching

To avoid *cwnd* parameter mismatch at a computation phase change, parameter switching has been designed. For switching, *ssthresh* is saved and restored before/after each communication pattern change. Only the *ssthresh* value is changed, but this is satisfactory because *ssthresh* holds a good estimation of the bandwidth-delay product before congestion. *cwnd* is rapidly changing, but *ssthresh* is stable. *ssthresh* is held in the TCP control structure of each connection, and saving and restoring it is easy.

MPI defines a set of collective communication primitives, and parameter switching is invoked at the start and the end of collective communications. At the start of a collective communication, a *ssthresh* value for the point-to-point communication is saved and one for the collective communication is restored. At the end of a collective communication, the values are swapped in reverse. Some of the collective communication primitives, such as barrier, which do not share the bandwidth, are not classified as the collectives.

In our implementation, TCP is forced to the Slow-Start state in addition to setting *ssthresh* in parameter switching. When TCP moves to the Slow-Start state, the pacing function described next is activated, and communication begins at a rate specified by *ssthresh*. This makes the implementation very simple in that it just calls the **tcp_cwnd_restart()** procedure, after setting *ssthresh* in Linux.

3.2 Pacing at Start-up

To avoid the performance degradation of Slow-Start, the pacing mechanism is employed at the Slow-Start state. The pacing mechanism is used to pace outgoing packets without depending on ACK-clocking [16].

The target rate for pacing is calculated by RTT/*ssthresh*. Thus, the target bandwidth is (*ssthresh*$*$MSS)/RTT, (where MSS is Maximum Segment Size). In Linux, *cwnd* and *ssthresh* are counted using the number of packets, instead of bytes as specified in the RFC [14]. As stated before, *ssthresh* holds a good estimation of the bandwidth-delay product, and it is suitable for the target rate.

Pacing is started when the TCP stack notices it has entered the Slow-Start state. Pacing is started when the following condition holds:

$$\begin{array}{c}\text{no in-flight packets}\\ \text{and}\\ \textit{cwnd} < \textit{ssthresh}\end{array}$$

A hook function is inserted in the TCP stack so that the condition is checked at sending of each packet in the hook function.

A very fast timer of 10 µsec is used to generate a clock for sending a packet. At each timer interrupt, the target rate is checked and a decision is made whether or not to send a packet. Pacing is stopped immediately upon receiving an acknowledgement. When pacing for all connections is stopped, the fast timer is also stopped. That is, the pacing is only activated during a short period of start-up of communication, and its overhead is small.

The timer should be fine enough to suppress burst traffic, and not to overflow a router buffer. A cycle of 10 µsec is fine enough because an MTU (Maximum Transfer Unit) size packet is sent in 12 µsec for 1 Gpbs Ethernet. A cycle of 10 µsec is also fine enough for 10 Gbps, where pacing can be performed at a unit of ten packets. Thus, the timer value of 10 µsec was chosen considering 10 Gbps networks as well as 1 Gbps networks.

The IA32 systems are equipped with APIC and HPET timers which both have finer granularity than 1 µsec. In this experiment, the APIC Timer is used because it is easier to use. Although the Linux-2.6 kernel includes the support code for both timers, we did not use them, but rewrote the interrupt vectors directly instead, because the full timer facility is not needed.

3.3 Reducing Retransmit-Timeout Time

The Retransmit-Timeout time (RTO) is specified in the RFC [14] as:

$$\text{RTO} = \text{SRTT} + 4 * \text{RTTVAR}$$

where, SRTT is Smoothed Round-Trip Time and RTTVAR is Round-Trip Time Variation, and both SRTT and RTTVAR are sampled and measured in the TCP stack.

Although the RFC specifies the RTO calculation, the RFC also defines the minimum value of RTO as 1 second (RTO *should* be rounded-up to 1 second when RTO is less than 1 second). Linux TCP uses a slightly different definition, that is, the minimum of $4*$RTTVAR is set to 200 msec. Thus, RTO is approximately RTT + 200 msec in Linux.

To reduce the pause in transmission when waiting for Retransmit-Timeout, the Retransmit-Timeout time is shortened. In reducing Retransmit-Timeout time, RTTVAR is set to the calculated value using the definition from the RFC, without rounding up the minimum value to 200 msec.

The RFC uses such a large minimum RTO, because old TCP implementations use a coarse timer of hundreds of milliseconds for processing jobs, such as delayed acknowledgements. This means that a small Retransmit-Timeout value causes retransmission of packets not actually lost. Recently, timers have become relatively finer at a millisecond level, and the retransmit algorithm has also been improved to be robust for a few extraneous retransmissions. In addition, retransmitting a few packets is no longer a problem for a Gbps class high-bandwidth environment.

4 Evaluation

4.1 Experimental Setting

In order to evaluate the proposed improvements, two PC-based clusters were connected in a WAN-emulated environment as shown in Figure 1. Each cluster consists of 8 nodes. Table 1 shows the specifications of the PCs and the network switch.

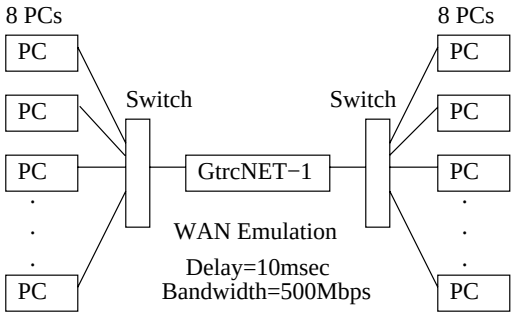


Figure 1. Experimental Setting

Table 1. PC Cluster Specifications

Node PC	
CPU	Pentium4 2.4C (2.4 GHz)
Motherboard	Intel D865GLC
Memory	512KB DDR400
NIC	Intel 82547EI (on-board CSA)
OS	Fedora Core 2 (Linux-2.6.9-1.6_FC2)
Switch	
	CISCO Catalyst 3750G-24T

The WAN emulator is GtrcNET-1 (formerly named GNET-1) [10], which is an FPGA-based configurable network testbed which can precisely control the traffic. The WAN emulator emulates a wide-area network connection: inserting delay, limiting bandwidth, and emulating router behavior. It also enables observation of the traffic with high-precision without affecting the observed traffic. The router emulation performs drop-tail buffer control, which discards packets when the router buffer overflows.

The settings for the WAN emulation are shown below:

Delay	10 msec
Bottleneck	500 Mbps
Router buffer	128 KB (drop-tail)

The delay of 10 msec was chosen as a realistic upper limit for MPI applications. We plan to run MPI applications in a wide-area network without modifying the code developed for PC clusters. Preliminary benchmark evaluation in a wide-area network emulation environment revealed that the performance does not scale for delay larger than 10 msec [11].

WAN emulation is disabled in some experiments, and it is so stated in the results. In that case, the setting is an ordinary PC cluster, and the emulation settings are: 0 msec delay, 1 Gpbs bottleneck, and 16 MB router buffer (which never overflows in the experiment).

GtrcNET-1 observes the traffic at 1 msec intervals, that is an average of 1 msec for each 1 msec. Observed traffic is from a cluster with lower ranks (node numbers) to the other cluster with higher ranks.

The TCP implementation is the default TCP of Linux-2.6.9, based on New-Reno [1, 4], using BIC-TCP [17] for the congestion avoidance algorithm. The MPI library is YAMPII [7], which is an MPI implementation on PC clusters, and is a base for GridMPI [5] for wide-area network communication. The MPI library uses plain sockets and eagerly sends messages to make it efficient in a large-latency environment. The compiler used is GCC (version 3.3.3) for both C and Fortran, and the optimization is -O4 for all tests.

In the experiment, a socket option, NODELAY, is set. NODELAY disables the Nagle algorithm which delays a send to merge small packets into one. Normally, MPI libraries disable the Nagle algorithm. The size of a socket buffer is set to 20 MB, which is enough for the bandwidth-delay product of the experiment environment, the value necessary to fill the network capacity.

While standard TCP sets *ssthresh* to infinity immediately after establishment of a connection, our implementation uses a predetermined value for pacing. In the experiment, an ad hoc value of 250 Mbps is used as the initial target rate because the bottleneck is known in advance as 500 Mbps.

4.2 Impact of Parameter Switching

Figure 2 shows the effect of parameter switching. One-to-one communication is started at time 0, just after collective communication. [1] The X-axis is time (sec), and the Y-axis is the bandwidth in 1 msec samplings. Traffic is limited to 60 to 70 MB/s, because the WAN emulator limits the bottleneck bandwidth to 500 Mbps.

The graph on the left is the non-switched case, where one-to-one is started at time 0, without restoring the *ssthresh* parameter. The graph on the right is the parameter switched case, where one-to-one is started at time 0, after restoring the parameter. The restored *ssthresh* is the one saved after the previous one-to-one communication. The graph on the right apparently shows the traffic starts with a stable state, while the graph on the left shows the bandwidth gradually increases following the application of the congestion avoidance algorithm. Note that the congestion avoidance algorithm is BIC-TCP [17] and the curve is not linear. The results sometimes show larger traffic than the bottleneck

[1] Collective communication here is a modified version of **MPI_Alltoall**, which only performs communication passing through a bottleneck link. This modification makes the experiment generate stable results. **MPI_Alltoall** is heavily dependent on the timing because **MPI_Alltoall** includes intra-cluster traffic, and the amount of traffic passing through the bottleneck varies greatly as timing changes.

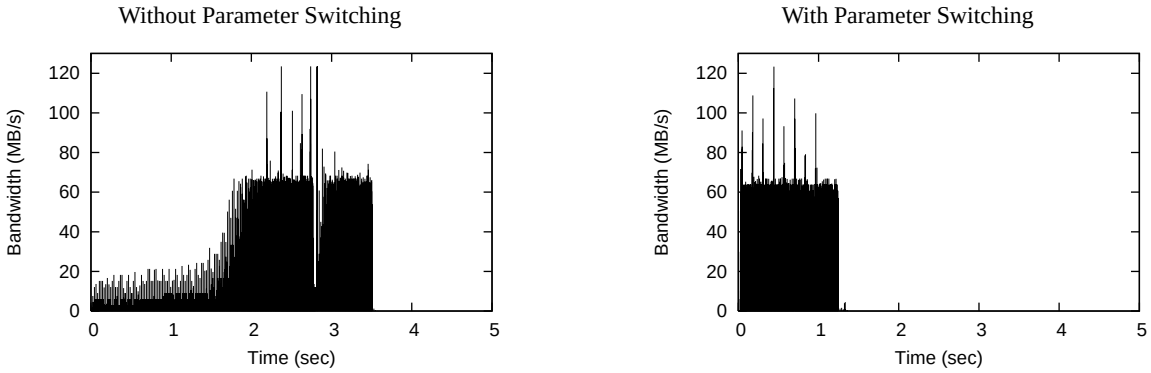


Figure 2. Effect of parameter switching

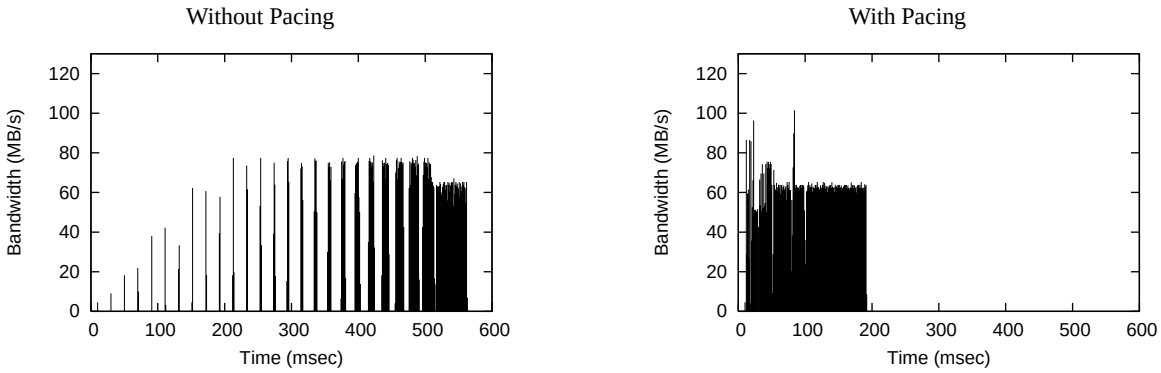


Figure 3. Effect of pacing at start-up

link. This is because the traffic is observed at the input of GtrcNET-1.

In the experiment, reducing Retransmit-Timeout time is applied for all connections, including intra-cluster communication. This is because a large RTO value also affects intra-cluster communication, as previously stated. Pacing at start-up is also enabled for all connections, although it has very little effect because the latency inside a cluster is short (some 10 μsec).

The data size of one-to-one communication is 50 MB. At the start of communication (time=0), *ssthresh* values were 49 and 582 for the non-switched case (left graph) and the switched case (right graph), respectively.

4.3 Impact of Pacing at Start-up

Figure 3 shows the effect of pacing at start-up. The graphs show the traffic in intermittent communication, where one-to-one data transfer is repeated with pauses. The experiment performed repeatedly sending chunks of 10 MB data at 2 second intervals. Each graph shows the traffic of one chunk. The X-axis is time (msec), and the Y-axis is the bandwidth in 1 msec samplings.

Figure 3 on the left-hand side (without pacing) shows that the traffic gradually increases for each round-trip time (20 msec) with Slow-Start. It is apparent that the available bandwidth is not utilized. Figure 3 on the right-hand side shows the case with pacing. The bandwidth is high from the beginning.

4.4 Impact of Reducing Retransmit-Timeout Time

Figure 4 shows the bandwidth of **MPI_Alltoall**, varying the message size. The bandwidth is the byte count of messages divided by the communication time. The curve labeled with *Modified TCP* is the case with reducing Retransmit-Timeout time, and it shows good improvement from standard TCP. Note that no delay and no limiting of the bandwidth at the bottleneck was applied in this experiment. The setting was chosen to show that the effect is independent of the behavior of the router emulation. It means that reducing the Retransmit-Timeout time is effective in a LAN environment as well as in a WAN environment.

The dotted line in the graph at 29 MB/s shows the ideal bandwidth in the experimental setting, where eight streams share the 1 Gbps bottleneck link.

Figure 5 shows the traffic at a data size of 32 KB. Com-

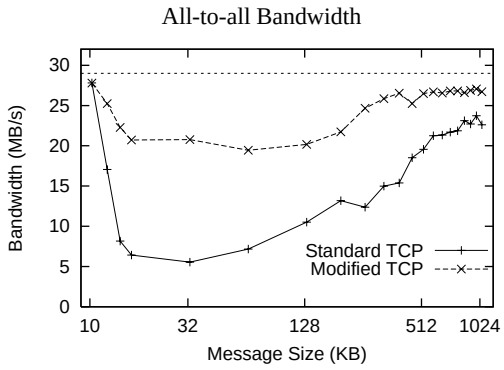


Figure 4. Effect of reducing Retransmit-Timeout time

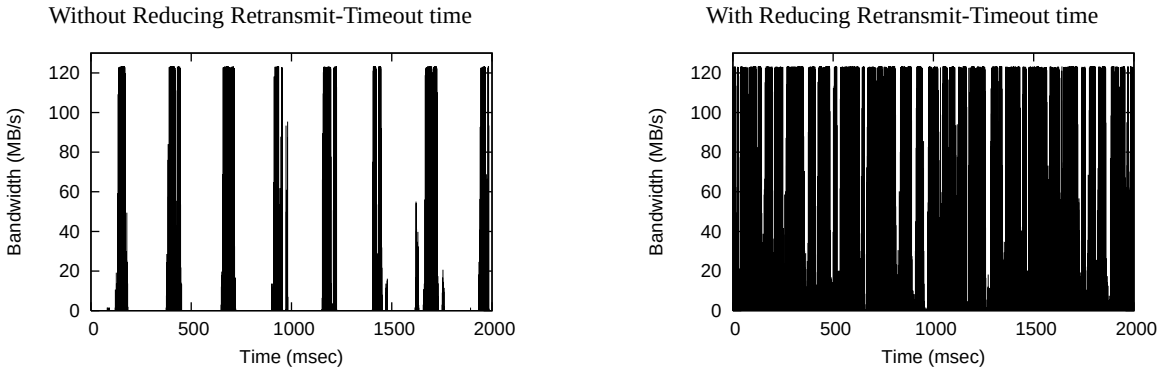


Figure 5. All-to-all Traffic (data size 32 KB)

paring the two graphs shows an apparent effect of the modification. Although **MPI_Alltoall** is called continuously, large gaps occur in the left graph. All-to-all communication does not progress until each node receives the tail of a message, which causes Retransmit-Timeout to happen. The gaps are over 200 msec, because Retransmit-Timeout is over 200 msec in the Linux TCP implementation. Note that routers and switches tend to discard the tail of a message which causes Retransmit-Timeout in high probability.

4.5 NPB Benchmark Results

Next, the effect of the modifications is evaluated using application benchmarks to show the effect in real applications. The benchmarks used are the NAS Parallel Benchmarks (NPB 2.3). The data set size is CLASS=B, and the number of processes is NPROCS=16. Measurement is taken from the best of three runs, because variance of around 10 percent was observed for the standard TCP case.

Figure 6 shows the relative performance normalized to standard TCP. Table 2 shows the absolute values. In both the graph and the table, *STD* is standard TCP, and *PCE* is pacing at start-up. *RTO+PCE* is the combination of pacing at start-up and reducing Retransmit-Timeout time. *ALL* is all combinations of the modifications, including parameter switching.

Table 2. NPB absolute value (Mop/s total)

	STD	ALL	RTO+PCE	PCE
BT	2835.48	3571.02	3887.74	3520.48
CG	293.86	371.99	372.48	325.94
EP	82.23	82.23	82.22	82.22
FT	1168.01	1182.69	1191.58	1182.74
IS	23.61	26.25	27.60	26.15
LU	4143.90	3781.36	3535.71	4198.00
MG	1130.55	1107.41	1112.02	1160.35
SP	995.72	1323.11	1368.42	1243.75

The results of reducing Retransmit-Timeout time alone and parameter switching alone are omitted, because some modifications depend on others. Reducing Retransmit-Timeout time causes frequent Slow-Start as a side-effect and it always degrades performance without pacing. Also,

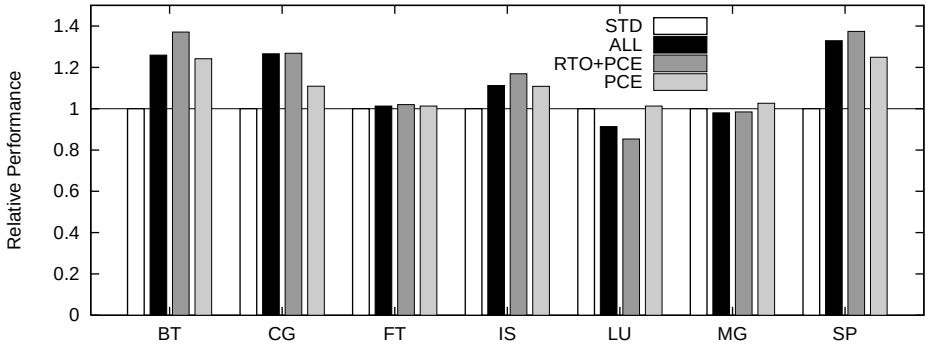


Figure 6. NPB performance comparison

the implementation of parameter switching depends on pacing.

BT, CG, IS, and SP show improvement from 10 to 30 percent, FT and MG show no improvement, and LU shows a degradation in performance. The traffic samples from the benchmarks are shown in the Appendix.

Examining the details of the traffic, the BT benchmark has a computation phase longer than 200 msec, which causes Slow-Start. Thus, the modification has a large effect. The CG benchmark has small bursts in traffic at small intervals, and some burst traffic seems to cause packet losses and congestion, which causes Slow-Start at some point in time. The FT benchmark has steady traffic and thus gained no effect. The IS benchmark has a computation phase at the start which causes Slow-Start, thus the modification has an effect. The LU benchmark has steady and very little traffic. Retransmit-Timeout and packet loss do not occur, so standard TCP performs well. The MG benchmark has small bursts, but they are smaller than the ones in the CG benchmark. Standard TCP works well without packet losses, thus, the modification has no effect. The SP benchmark starts with Slow-Start, so the modification has a large effect.

4.6 Overhead of Fast Timer

Interrupting at 10 µsec, a program with an empty loop slowed down by 1.8 to 2.0 percent. The timer interrupt is used only during the pacing, which only lasts for a round-trip time. Therefore, the performance degradation caused by the timer interrupt may be small.

5 Related Work

Modifications to the start-up behavior have been the subject of many proposals in the context of adaptation to WEB traffic [6]. Aron, et. al [2], discussed reducing Retransmit-Timeout time and pacing at start-up. This work shows that using a relatively fine timer (10 msec) has better effect on Retransmit-Timeout time, compared to the old timer (200 msec or 500 msec) used in the old BSD Unix implementations. Pacing with a 10 msec timer also shows a better effect in the simulation of a 128 Mbps bandwidth and 5 msec delay network. Our work adapts these results to a more recent environment, where using the finer timer does not reduce the Retransmit-Timeout time any more, and thus it needs to explicitly cut the time. And also pacing with a 10 msec timer is not effective in Gbps class networks, and so it is necessary to use a much faster timer.

For APIC and HPET fast timers on the IA32 systems, Oberle, et. al, discussed the APIC timer and pacing experiments of UDP packets at µsec precision in [12]. Kamezawa, et. al, discussed fast timers and burst avoidance, and experimented using trans-pacific network communication [9]. In [15], we discussed a very precise pacing mechanism implemented totally in software, where the timing is generated by the network interface at a packet level.

Papers [2, 13, 16] discussed determination of the pacing target, measurement of precise round-trip time, and a decaying factor affecting the *ssthresh* parameter due to time.

Ensemble-TCP [3] discussed the sharing of TCP parameters by connections to the same target host, taking into account that these connections may share the same network path and will share the behavior. Under that assumption, this work proposes to share TCP parameters among connections, which improves the performance. Our approach is roughly the opposite, that is, even a single connection has differing characteristics with regard to phases of computation and communication, and switching multiple sets of parameters improves the performance.

6 Conclusion

By small modifications to the intermittent behavior of TCP, it has been shown that these modifications improve the performance of communication in an MPI work load. Even though the modifications are limited to the start-up and no modifications are made to the congestion avoidance phase, four of the NPB benchmarks are improved by 10 to 30 percent.

The modifications have the following effects: reducing Retransmit-Timeout time reduces a long pause in recovery at congestion, pacing at start-up avoids a bad utilization of the bandwidth due to Slow-Start, and parameter switching improves estimation of available bandwidth at phase changes of computation.

TCP modification should consider the fairness to other streams. The modifications in this paper are just for the behavior at the start-up (within a round-trip time) and no change are made in the congestion avoidance phase where the fairness is an important issue. Thus, the fairness of the modified TCP follows that of the base TCP. In addition, the modifications suppress burst traffic and so they should reduce influence on other streams.

References

[1] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581, 1999.

[2] M. Aron and P. Durschel. TCP: Improving Startup Dynamics by Adaptive Timers and Congestion Control. Tech. Rep. TR98-318, Rice Univ., 1998.

[3] L. Eggert, J. Heidemann, and J. Touch. Effects of Ensemble-TCP. ACM Computer Communication Review, 30(1), 2000.

[4] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582, 1999.

[5] GridMPI Home Page. http://www.gridmpi.org

[6] A. S. Hughes, J. Touch, J. Heidemann. Issues in TCP Slow-Start Restart After Idle. Expired Internet Draft, draft-hughes-restart-00.txt, 2001.

[7] Y. Ishikawa. YAMPII Official Home Page. http://www.il.is.s.u-tokyo.ac.jp/yampii

[8] V. Jacobson. Congestion Avoidance and Control. Proc. SIGCOMM'88, in ACM Computer Communication Review, 18(4):314-329, 1988.

[9] H. Kamezawa, M. Nakamura, J. Tamatsukuri, N. Aoshima, M. Inaba, K. Hiraki, J. Shitami, A. Jinzaki, R. Kurusu, M. Sakamoto, and Y. Ikuta. Inter-layer Coordination for Parallel TCP Streams on Long Fat Pipe Networks. SC2004, 2004.

[10] Y. Kodama, T. Kudoh, R. Takano, H. Sato, O. Tatebe, and S. Sekiguchi. GNET-1: Gigabit Ethernet Network Testbed. IEEE Intl. Conf. on Cluster Computing (Cluster2004), 2004.

[11] M. Matsuda, Y. Ishikawa, and T. Kudoh. Evaluation of MPI Implementations on Grid-connected Clusters using an Emulated WAN Environment. CCGrid2003, 2003.

[12] V. Oberle and U. Walter. Micro-second Precision Timer Support for the Linux Kernel. IBM Linux Challenge, 2001. http://www.tm.uka.de/itm/publications.php?id=61

[13] V. N. Padmanabhan and R. H. Katz. TCP Fast Start: A Technique for Speeding Up Web Transfers. Proc. IEEE GLOBECOM Internet Mini-Conference, 1998.

[14] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988, 2000.

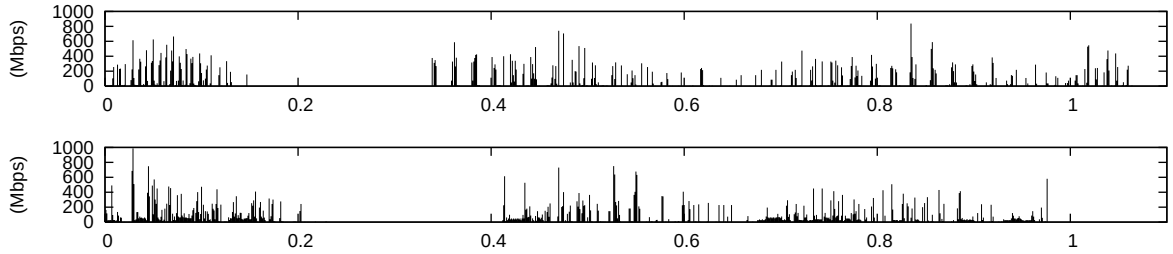
[15] R. Takano, T. Kudoh, Y. Kodama, M. Matsuda, H. Tezuka, and Y. Ishikawa. Design and Evaluation of Precise Software Pacing Mechanisms for Fast Long-Distance Networks. 3rd Intl. Workshop on Protocols for Fast Long-Distance Networks (PFLDnet05), 2005.

[16] V. Visweswaraiah and J. Heidemann. Improving Restart of Idle TCP Connections. Tech. Rep. 97-661, Univ. of Southern California, 1997.

[17] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control for Fast Long-Distance Networks. INFOCOM 2004.

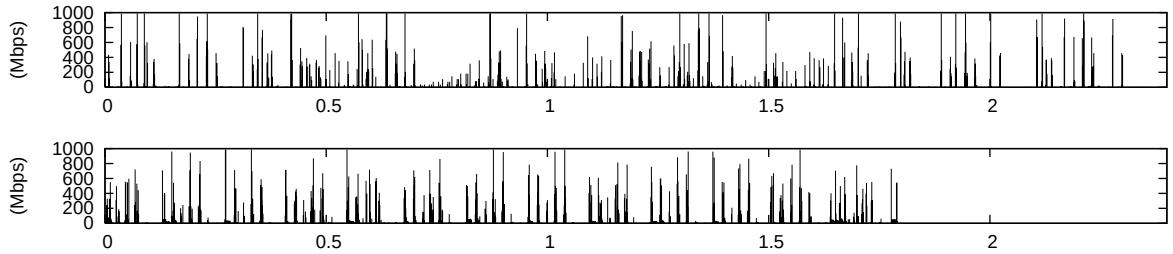
A NPB Traffic Samples

This appendix views the traffic samples of the NPB benchmarks. The graphs show the samples of one loop from the benchmarks. The upper graph is standard TCP, and the lower one is modified TCP. The X-axis is time (sec) and the Y-axis is bandwidth (Mbps). Comparing the two well reveals the behavior of TCP. Note that the graphs show the sum of the traffic from multiple streams, since the traffic is observed at the bottleneck between clusters. Also note that the start of a loop deviates sightly because of delivery of a trigger to the network emulator.



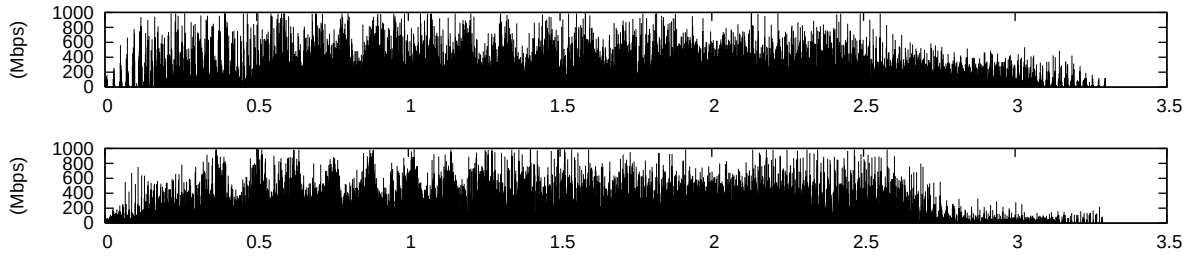
Traffic samples of BT

In BT, there is a pause in communication around 0.3 second for standard TCP. That causes the following communication starts with Slow-Start.



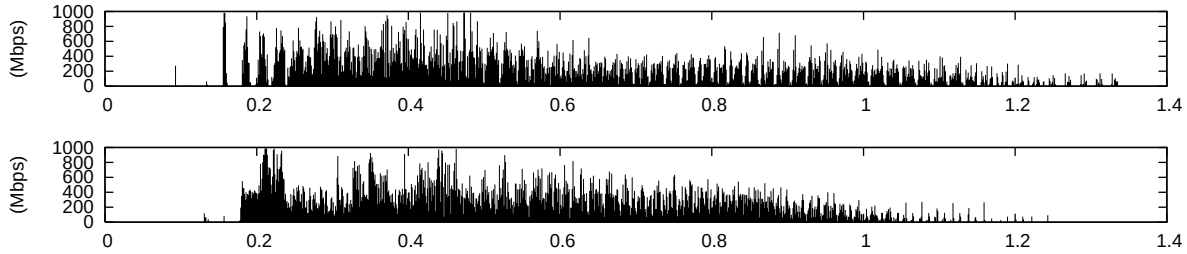
Traffic samples of CG

In CG, disruption occurs in a short period. CG repeats a short communication and ACK-clocking does not work well. Some burst traffic is observed which causes losses of packets. Examining the graph in detail shows standard TCP starts Slow-Start at around 0.7 second.



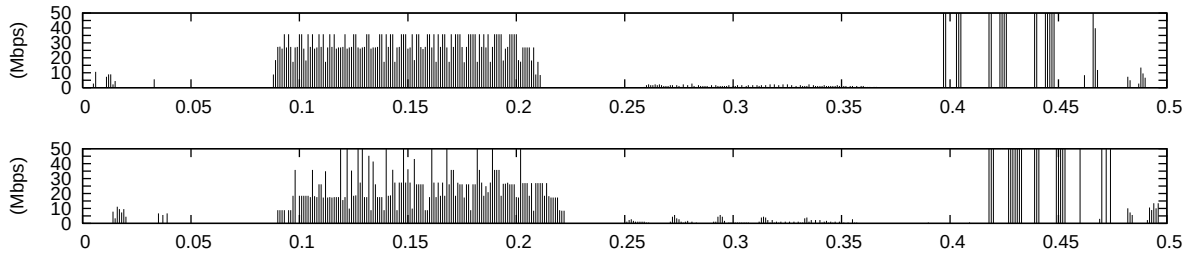
Traffic samples of FT

In FT, standard TCP and modified TCP show similar behavior.



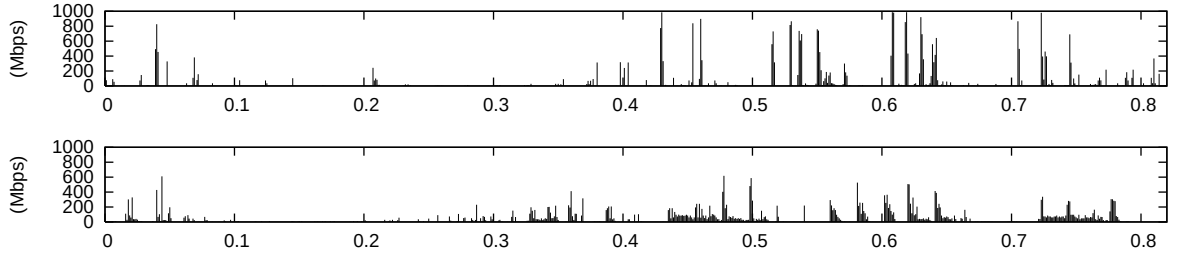
Traffic samples of IS

In IS, there is a pause in communication at the start of a benchmark round, and standard TCP seems to start by Slow-Start which slows down the performance.



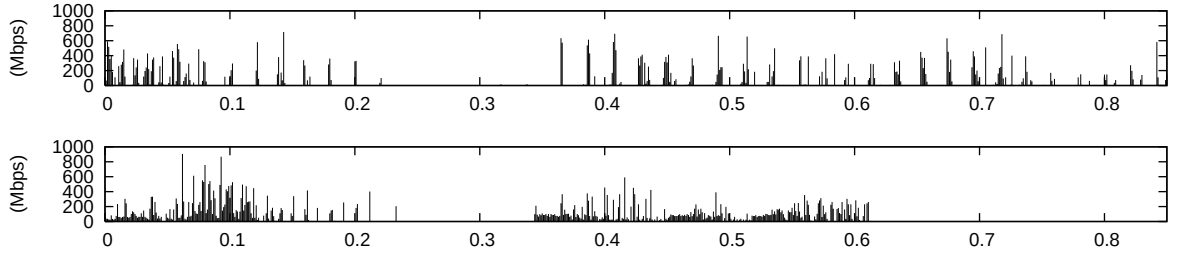
Traffic samples of LU

In LU, very small communication is maintained. Note that the scale of the Y-axis is 1/20 that of other benchmarks. Standard TCP works well for this amount of traffic without causing congestion. Modified TCP is slower because it starts communication by pacing with the target rate which is conservatively lower.



Traffic samples of MG

In MG, traffic is low and bursts are smaller than CG, and standard TCP does not seem to have packet drops. Pacing used in modified TCP lowers the peaks of the traffic, but the overall behavior is similar.



Traffic samples of SP

In SP, there is a quiescent state at around 0.3 second, and standard TCP enters Slow-Start after that period.