Using SCTP to hide latency in MPI programs

H. Kamal, B. Penoff, M. Tsai, E. Vong, A. Wagner

University of British Columbia
Dept. of Computer Science
Vancouver, BC, Canada
{kamal, penoff, myct, vongpsq, wagner}@cs.ubc.ca

Abstract

A difficulty in using heterogeneous collections of geographically distributed machines across wide area networks for parallel computing is the huge variability in message latency that is orders of magnitude larger than parallel programs executing on dedicated systems. This variability is in part due to the underlying network bandwidth and latency which can vary dramatically according to network conditions. Although such an environment is not suitable for many message passing programs there are those programs that can take advantage of it.

Using SCTP (Stream Control Transmission Protocol) for MPI, we show how to reduce the effect of latency on task farm programs to allow them to effectively execute in high latency environments. SCTP is a recently standardized transport level protocol that has a number of features that make it well-suited to MPI and our goal is to reduce the effect of latency on MPI programs in wide area networks. We take advantage of SCTP's improved congestion control as well as its ability to have multiple independent message streams over a single connection to eliminate the head of line blocking that can occur in TCP-based middleware.

The use of streams required a novel use of MPI tags to identify independent streams rather than different types of messages. We describe the design of a task farm template that exploits streams, uses buffering and pipelining of task requests to improve its performance under network loss and variable latency. We use these techniques to improve the performance of two real-world MPI programs: a robust correlation matrix computation and mpiBLAST.

1 Introduction

There is considerable interest in taking advantage of large numbers of geographically distributed machines connected across wide area networks (WANs) for computing distributed and parallel applications. One large source of programs that already exist is MPI message passing programs. MPI (message passing interface) has been widely adopted for use in high performance computing and there are tools and runtime systems to support its use in heterogeneous environments, like the Internet [11, 2, 5]. In these environments, implementations of MPI typically rely on the standard Internet protocol stack, TCP and UDP over IP. The use of these ubiquitous standard protocols allows MPI programs to seamlessly execute in very diverse environments.

Using MPI in wide area networks is challenging because of the latency, which can be one or two orders of magnitude larger than a LAN or a dedicated parallel machine. Variability in latency is a problem where the delay of even a single message can delay the entire computation. In TCP, variability in delay or round trip time is due to flow control and congestion avoidance, as indicated by segment loss. Segment loss in particular causes a spike in latency, slowing down all messages in the stream, as well as reducing the available bandwidth.

SCTP (Stream Control Transmission Protocol) is a newly standardized transport protocol [16] similar to TCP but has been shown to perform much better than TCP under segment loss [15]. Although there are several reasons for the improved performance, one feature of SCTP of particular interest is the ability to have multiple independent message streams inside a single connection. Messages on a stream are guaranteed to be delivered to the application in-order, but messages on different streams can be delivered in the order in which they are successfully received. Under loss conditions,

1-4244-0054-6/06/$20.00 ©2006 IEEE

this ensures that a delayed message on one stream does not block the delivery of messages on the other streams (i.e., no head of line blocking). For example, in TCP-based MPI implementations, when there is a single connection between two processes and several messages in flight, packet loss in the first message delays the arrival of all subsequent messages, even when these messages could be independently completed by the receiver.

We have implemented an SCTP-based version of MPI that takes advantage of SCTP streams [9]. In [9] we described the design of the MPI communication middleware and compared TCP to SCTP under loss for the NAS benchmarks, which did not use the multiple stream feature of SCTP, and a synthetic farm program which did. Although we were able to demonstrate head of line blocking, it was not clear how easy or to what extent multiple streams could be used to improve the latency tolerance of real programs. Multiple streams are advantageous only when there are sufficient number of independent messages in flight. In an MPI program this requires the extensive use of asynchronous communication and, as we propose in this paper, the use of tags to identify messages that can be completed independently at the receiver. This is a novel use of tags since tags are usually used to denote message types. Our use of tags is consistent with the message-ordering semantics of MPI and does not preclude other uses of tags. But, by identifying those messages that are independent, it is makes it possible to have multiple streams active at the same time, thus reducing the effect of message loss. The use of tags for the purpose of identifying independent message streams may in general be of use for any middleware that is able to manage or schedule messages (for example OpenMPI [7]).

We demonstrate how our use of tags can lead to better performance for a class of MPI programs that can hide latency, namely farm programs. We introduce a simple model to characterize the effect of latency on the performance of farm programs. We then describe the design of a farm template that uses tags, buffering and asynchronous communication to hide latency. We experimentally show that SCTP programs using these techniques are less sensitive than TCP to fluctuations in latency due to the conditions in the network. Some of these performance improvements are the result of better SCTP congestion control and our use of SCTP streams to eliminate the head of line blocking that occurs with TCP. The farm template and a discussion of the performance results comparing SCTP to TCP under varying network loss and latency conditions is given in Section 3. Finally, we experimented on real-world programs by applying our techniques to two existing MPI programs: a robust correlation matrix computation and mpiBLAST [4]. In Section 5 we briefly describe some related work and give our conclusions in Section 6.

2 Background

In this section we give a brief overview of MPI middleware, SCTP and our implementation of an SCTP module for LAM-MPI. A more complete description of the design can be found in [9].

2.1 MPI middleware

MPI has a rich variety of message passing routines. These include `MPI_Send` and `MPI_Recv` along with various combinations such as blocking, nonblocking, synchronous, asynchronous, buffered, unbuffered versions of these calls. It is the responsibility of the middleware to progress messages from a send call in one process to the matching receive call in another process. The *message progression layer* is the part of the middleware responsible for message progression, matching and delivery. Message matching is based on three values inside the message envelope: (i) context, (ii) source/destination rank, and (iii) tag . The context identifies a set of processes that can communicate with each other. Within a context, each process has a unique identification called rank and messages can be further specified by a tag value. A receive call matches a message when the context, rank and tag specified by the call matches the corresponding values in a message envelope. MPI semantics dictate that messages belonging to the same tag, rank and context (TRC) must be received (i.e., completed) in the order in which the sends were posted.

The message progression layer in our MPI middleware uses SCTP for transport and is similar in design to LAM's TCP progression layer. As is typical, it uses a short and long message protocol and also maintains two queues, an expected message queue and an unexpected message queue. The expected message queue contains posted receives that have not yet been matched. The unexpected queue contains messages that have arrived for which a receive has not been posted. Short messages are sent eagerly. When the message arrives at the destination the middleware first searches the expected message queue for a match and if unsuccessful, then adds the message to the unexpected queue. A receive for a short message causes the middleware to first search the unexpected queue for a match and if unsuccessful, then adds the receive request to the expected message queue. In this manner, as long as there are sufficient message slots in the queues, messages can

be delivered independently of whether the send or the receive was posted first. Synchronous short sends are also sent eagerly but the send does not complete until an acknowledgment is received. Long messages are not sent eagerly but use a rendezvous mechanism so that the send does not complete until after the receive has been posted. For long messages, this ensures that large messages can be directly transferred into the receive buffer.

2.2 SCTP

One feature that makes SCTP well suited for MPI is the ability to have multiple streams within a single association. An association in SCTP is similar to a TCP connection but has broader scope. Streams are multiplexed on an association and sequenced. Hence messages within a stream are delivered in-order to the application, while messages on different streams are delivered according to their order of arrival. In the case of loss, multiple streams makes it possible to deliver a message on one stream before messages on another. In our implementation of the MPI middleware MPI message ranks are mapped to separate associations and the context and tag field are mapped to a stream within the association for a given rank (see Figure 1). This

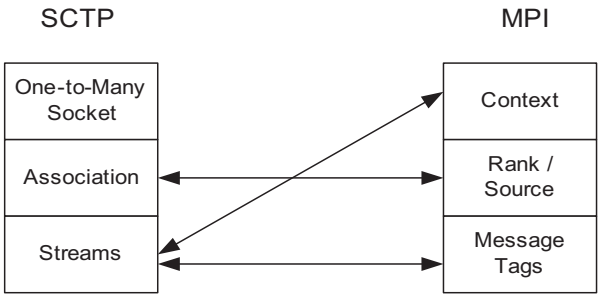

Figure 1. Mapping of MPI tags, rank and context to SCTP streams.

is consistent with the message-ordering semantics of MPI and in combination with non-blocking sends and receives, makes it possible to have several independent message streams active at the same time.

Endpoints in SCTP are more general than those in TCP and in an SCTP association an endpoint, by default, binds the IP addresses of all the interfaces to a port. An SCTP association is thus represented by a set of IP addresses and a port on both the source and destination endpoints. This is SCTP's multi-homing feature and allows for multiple paths between two endpoints. In SCTP the multiple paths are used for automatic failover in the case of failure and it is also used for congestion control where re-transmissions are sent over a different path.

SCTP can operate in a one-to-one manner where, like TCP, there is a association (i.e. connection) between two machines, or it can operate in a one-to-many manner where a single socket can receive messages from multiple processes where there are multiple associations, one for each process. In both cases, SCTP provides reliable in-order delivery of messages and uses TCP-like congestion control mechanisms to react to network conditions. We used one-to-many SCTP sockets to implement the MPI middleware. This avoids having a separate socket and connection for each process as occurs in TCP-based MPI middleware. Like UDP, SCTP is message-based and each socket call returns a complete message.

There are several other features of SCTP that make it an attractive target for MPI in open network environments. SCTP has several security enhancements that make it more secure than TCP. Another property of SCTP which makes it conducive to being widely deployed is that it is TCP-friendly. When several SCTP and TCP sources are sharing the same network, they share the network resources fairly [15]. Although relatively new, SCTP is currently available for all major operating systems and is part of the standard Linux kernel distribution.

2.3 SCTP module for LAM-MPI

In our SCTP-based middleware we mapped each tag into a separate stream within an association. In the case of loss, it allows messages in the same association (i.e. rank) with different tags to overtake each other. In addition during message progression on the send side, we multiplex short messages with long messages on a given association. As a result, it is possible for short messages to be sent and received during the transmission of a long message.

As mentioned, SCTP is message based and this fact both simplifies and complicates the middleware. It simplifies the handling of short messages, since message framing is done by SCTP and each receive socket call returns a compete message. For long messages, it is necessary to fragment and re-assemble them in the user's receive buffer. Also, as the result of using one-to-many style SCTP sockets, we did not use `select()`, and we need to pull messages from socket buffers as long as there are messages. This eliminates the costly `select()` call and also makes it easier to empty socket buffers reducing the chance that flow control closes the advertised window thus reducing the available bandwidth. But, the one-to-many style complicates the

handling of those primitives where it would be easier to simply block on the appropriate connection. As a result, we need to poll. As well, the number of unexpected messages tends to be larger in SCTP because of the need to pull messages from the socket buffers.

Like the TCP-based layer in LAM-MPI, our SCTP-based layer in LAM-MPI runs as a single thread with the application program and thus only progresses messages during MPI calls (weak message progression). For protocols using standard TCP-like flow control this can cause performance problems because the advertised window may shrink when the middleware is not running often enough to empty the socket buffers. For MPI non-blocking communication this suggests that we need to carefully plan how often we poll for messages at the application program. In the case of weak progression it is important to consider how messages are advanced during each call.

2.4 Experimental setup

Our experimental setup consists of a dedicated cluster of eight identical Pentium-4 3.2GHz, FreeBSD-5.3 nodes connected via a layer-two switch using 1Gbit/s Ethernet connections. Kernels on all nodes are augmented with the Kame.net SCTP stack [10]. The experiments were performed in a controlled environment and Dummynet was configured on each of the nodes to allow us to vary the latency and loss on the links between the nodes. We have separate Dummynet pipes for TCP and SCTP where each one is setup such that packets are dropped at both ends (inbound + outbound) of each node. We also used an instrumented version of the middleware to determine the effect of latency and loss on the number of expected and unexpected messages and the frequency of calls to the `advance()` function which progresses messages.

In order to make the comparison between SCTP and TCP as fair as possible, the following settings were used in all the experiments discussed in subsequent sections:

1. By default, SCTP uses a larger `SO_SNDBUF/SO_RCVBUF` buffer size than TCP. In order to prevent any possible effects on performance due to this difference, the send and receive buffers were set to a value of 220 Kbytes in both the TCP and SCTP modules.

2. Nagle's algorithm is disabled by default in LAM-TCP and this setting was used in the SCTP module as well.

3. An SCTP receiver uses Selective Acknowledgment `SACK` to report any missing data to the sender, therefore, the `SACK` option for TCP was enabled on all the nodes used in the experiment.

4. In our experimental setup SCTP's multi-homing feature was not used so as to keep the network settings as close as possible to that used by the LAM-TCP module.

5. TCP is able to offload checksum calculations on to the NICs on our nodes and thus has zero CPU cost associated with its calculation. SCTP uses a CRC32c checksum which adds overhead in terms of CPU cost. We modified the kernel to turn off the CRC32c checksum in SCTP to better compare the two.

6. Unlike Linux, TCP in FreeBSD does not use `maxburst`. The `maxburst` parameter is part of a rate pacing mechanism that limits retransmissions when exiting *Fast Recovery*. If `maxburst` is set too high, bursting will stress the message queues in the network and lead to increased packet loss. Whereas if `maxburst` is set too low, packets will be send out at a slower pace limited by the `maxburst` value. The SCTP stack in FreeBSD implements its own version of `maxburst` while TCP under FreeBSD does not and essentially sends out segments as quickly as possible. Therefore we set `maxburst` in SCTP from the default of 8 to 148 segments, which corresponds to the maximum size of SCTP's socket buffer.

3 Latency tolerant processor farm

Although many MPI programs are not suitable for execution in a WAN environment, there is one class of programs that is a good candidate, namely task farms. Task farming is a commonly used strategy for executing problems that are pleasingly parallel consisting of some large number of independent tasks. A task farm may be part of a larger computation; granularity (task execution time) can often be varied and tasks are often dynamically distributed to load-balance the computation. If one can sufficiently overlap communication with computation, then it is possible to effectively compute these types of message passing programs in WANs. We describe a farm template that uses tags to identify independent streams. In addition we provide a simple model for the execution of task farms in a WAN environment and use the model to demonstrate where multiple streams can be used to hide latency.

The farm template is demand-driven where workers request work from the manager and return results as

soon as they are completed. The basic structure of the farm is shown in Figure 2.

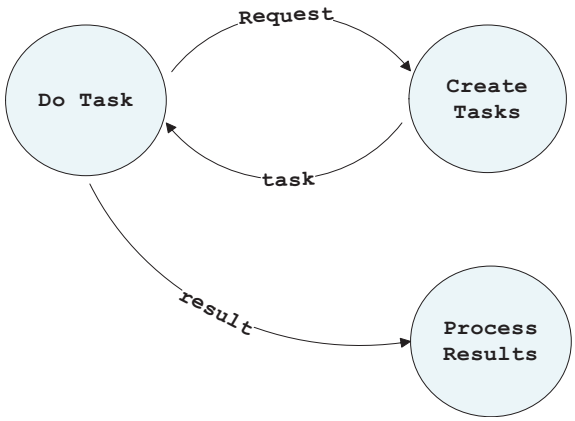

Figure 2. Structure of the farm template.

There are three main routines: one to create tasks, one to do the task, and finally one to process the results. Typically the `do_task` routine requires the most computation time and is the routine performed by a separate worker process. The `create_task` and `process_results` routines are often combined into a single manager process that distributes tasks and gathers results from the workers.

Assuming a separate process for each of the routines leads to the type of interactions shown in Figure 3.

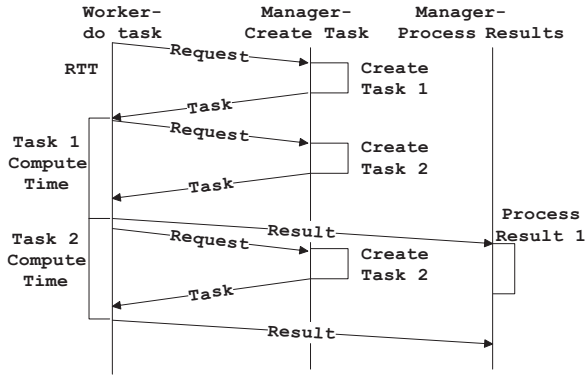

Figure 3. Execution scenario between a worker process and two manager processes.

Figure 3 assumes that the execution of a task can be overlapped with the request for the next. This is the ideal case since network latency, which is part of the time to obtain another task, can be entirely overlapped with task execution. If task granularity is too fine or the request time becomes too large, then the worker idles waiting for its next task and we get the type of execution scenario depicted in Figure 4.

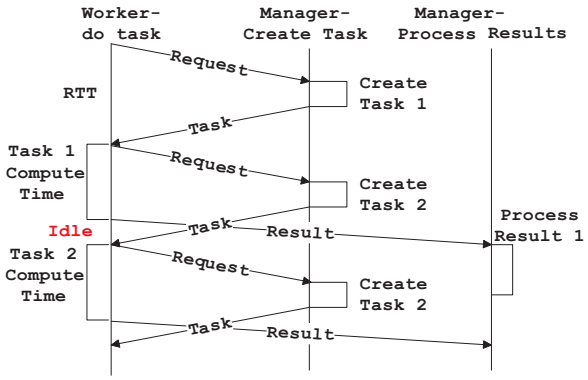

Figure 4. Execution scenario with higher latency and finer grain tasks.

These scenarios give rise to following formula (Figure 5) for the performance of one worker executing k tasks.

```
T = RTT + k × T(do_task)
        + max((k − 1)×(RTT − T(do_task)),0)

where

 T(do task)= time for the do_task routine

 k = number of tasks executed

 RTT (round trip time)
       = T(create_task)+C(req)+ C(reply)

 T(create_task)= time for create_task routine

 C(req)= communication time for a request

 C(reply)= communication time for a reply
```

Figure 5. Formula for the execution time of one worker.

The term $\texttt{RTT} - T(\texttt{do\_task})$, when positive, is the time the worker idles waiting for another task.

The execution time of the entire farm will be the maximum time taken by all the workers, where the time for the worker includes the time to send and process the last result ($C(\texttt{result}) + T(\texttt{process\_results})$). We assume that results are sent as soon as they are computed and that, as we assumed with tasks, the communication for sending the results is overlapped with the computation. The model simplifies the communication and assumes asynchronous communication where it is possible to overlap communication with computation. As well, although communication time and task computation time in general varies, we expect that for

a sufficient number of tasks, the steady-state performance of the farm to follow the given formula.

The formula was validated by choosing various task compute times and task create times which ranged from small to large. We also varied the message sizes from short messages to long (greater than 64Kbytes). By adjusting the ratio between task compute and task create time, we tested the farm template under the different scenarios that could occur. Under the assumption that fairness is strictly enforced, the experiment was performed using the manager with a single worker. The experimental results were within 5% of the expected values. We did not test the model for large number of processes, where scheduling may be a factor, nor for dynamically varying compute times and message sizes. The same model applies to the case of heterogeneous workers since the task compute rates can vary and the request-driven farm template will adjust just as it does to varying task compute times.

3.1 Variation in Latency

In a less ideal setting, where task execution time and latency varies, then $[\texttt{RTT} - T(\texttt{do\_task})]$ may fluctuate above zero and degrade performance. In these situations, buffering can help smooth out the variation and ensure that the worker does not idle waiting for a task. Buffering along with asynchronous communication makes it possible to have multiple MPI communications actively sending requests, receiving tasks, and sending results. Separate MPI tags will be used to identify the different instances of requests, tasks and results that are independent from each other.

The use of separate tags to identify independent message streams makes it possible for our SCTP-based middleware, should loss occur, to complete messages in the order in which they are successfully sent or received by the transport layer. This eliminates the head of line blocking that can occur when a packet is lost and allows out of order delivery to the MPI middleware layer. This has the potential of reducing or eliminating idle time on instances when RTT become larger than $T(\texttt{do\_task})$ since instead of waiting, the execution of buffered tasks can proceed. In addition to the elimination of head of line blocking, there are other features of SCTP that reduce the effect of loss on latency and bandwidth. SCTP's congestion control mechanism has improvements over TCP that allow it to achieve larger average congestion window size and faster recovery from segment loss [15] .

3.2 Design of the buffered farm template

The goal of our buffered farm template is to use SCTP to reduce the effect of loss and latency on the application. In the design of the buffered farm template, individual workers determined the number of buffers to use and also the number and frequency of task requests. Our implementation of the manager combined the `create_task` and `process_results` into one process and was also responsible for distributing tasks based on the requests it receives. The manager uses a simple round-robin scheduling scheme that distributes tasks fairly according to the demand.

As mentioned, MPI tags were used as task identifiers (IDs) which made it possible to map each task to its own SCTP stream. We used a pool of task IDs so that the manager could execute an arbitrarily large number of tasks. Using a program defined bound on the number of tags made it possible to maintain other tag uses while at the same time reserve a range of tags for SCTP streams. Although tags are 32 bits integers, when other uses are taken into account, it is important to be able to reuse the tags. The manager also maintained a dictionary using the tag as its key to match results to tasks. This may not always be necessary. IDs can be managed locally by the worker since tags only need to be unique with respect to a particular association.

Figure 6 illustrates the overall structure of the worker program. While there are tasks to execute, the worker blocks on `MPI_Waitany` to receive the first available task (j) from all the posted `MPI_Irecv`. The worker immediately sends off another request, `MPI_Isend`, and then executes the task. Once finished, it uses `MPI_Isend` to send the result and then again posts an `MPI_Irecv` for the next task.

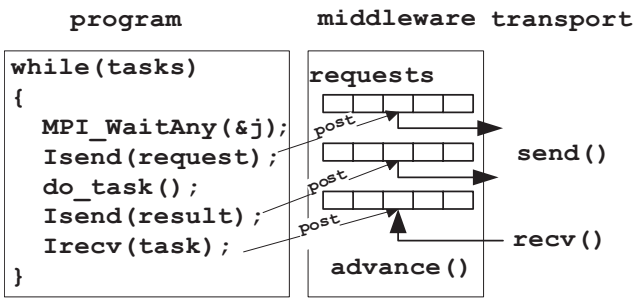

Figure 6. Overview of farm software.

In our implementation, each worker used a fixed number of maximum outstanding requests, `MAXREQS`, and we also added the ability for the worker to batch requests by allowing the worker to request k tasks rather than simply one. Batching requests reduces the num-

ber of messages and also tends to send more tasks earlier, which increases the potential for head of line blocking that SCTP can avoid.

We do not batch tasks by sending more than one task at a time. Although there may be less overhead in sending one larger message, it increases the negative effects of message loss. If it is possible to batch tasks, then it is often easy to increase the granularity of the task and avoid the cases where `RTT` is large enough to cause performance problems.

One enhancement to the template we did not explore was to vary the number of requests according to network conditions. The number of outstanding requests can easily be adjusted dynamically, however, under varying network conditions, the challenge is to avoid buffering too many tasks yet to buffer enough to avoid idle time. Buffering too many task is a problem towards the end of the computation [18]. If the number of tasks are known, then towards the end of the computation the workers can request fewer tasks. The manager does record the number of outstanding tasks for each worker. We did not use this information, however, it could be used to implement a scheduling regime other than round-robin to better ensure load-sharing.

There is a complication that occurs with long messages. Long messages result in a rendezvous, which does not allow the overlap of communication with computation. However, the application level protocol used in the worker and manager ensures that the receives are always posted before the corresponding send. This reduces wait time, but cannot avoid the synchronous transfer of the message from the send slot to the receive slot, which does add extra communication costs to receiving tasks or sending results. Our use of multiple streams makes it possible to multiplex long and short messages over the same association. However, it may result in added delays in cases where the long message needs to be received as early as possible.

3.3 Performance of the buffered farm program

We experimented with the buffered farm program on the setup described in Section 2.4 using a synthetic workload. The experiments were run at loss rates of 0%, 1% and 2% and added network latencies of 0, 20, 40 and 80 milliseconds. The farm program was run three times for each of the different combinations of loss rates and message sizes and the average value of total runtimes are reported. The average and the median values of the multiple runs were very close to each other. We also calculated the standard deviation of the average value, and found the variation across different runs to be very small.

Table 1 and Table 2 compare the performance of the buffered farm program using SCTP-based middleware versus the performance of the same program for TCP-based middleware for short (12Kbyte) and long message (128Kbyte)sizes. In the tables, we have highlighted in bold the entries where SCTP performed better than TCP.

	Latency(milliseconds)			
	0		20	
Loss	SCTP	TCP	SCTP	TCP
0%	16.98	16.95	17.11	17.11
1%	**16.72**	30.85	**26.01**	59.43
2%	**17.19**	42.63	**37.35**	90.04

(a) small latencies

	Latency(milliseconds)			
	40		80	
Loss	SCTP	TCP	SCTP	TCP
0%	21.61	19.76	**36.62**	36.65
1%	**51.77**	109.06	**102.78**	200.26
2%	**69.47**	160.66	**123.72**	264.29

(b) large latencies

Table 1. Comparison of the execution time (in seconds) of SCTP and TCP for a farm using 7 workers with a synthetic workload of 5000 tasks for short messages.

As both Table 1 and Table 2 show, SCTP outperforms TCP as loss and latency increases. For short messages, TCP execution times are about 2 times longer than SCTP at loss rates of 1% and 2%. For long messages, TCP is slower than SCTP by about 70% at 1% loss rates with 20 - 80ms delay and about 80% at 2% loss rates with 20ms delay. This occurs because buffering has not been able to sufficiently hide all the latency. For each task request, RTT is a combination of the network latency and the time to create tasks at the manager. As network latency increases at some point buffering is no longer able to hide the latency and worker executes at the speed at which it can obtain tasks. As segment loss increases, this creates a spike in latency, which depends on how quickly the congestion control mechanisms of the transport layer can recover from the loss. The table shows that, when loss occurs, the improvements of SCTP over TCP increases with increased latency.

Loss	Latency(milliseconds) 0 SCTP	0 TCP	20 SCTP	20 TCP
0%	6.50	5.88	**26.10**	56.14
1%	**35.50**	40.59	**189.97**	323.67
2%	**53.53**	80.95	**287.77**	519.49

(a) small latencies

Loss	Latency(milliseconds) 40 SCTP	40 TCP	80 SCTP	80 TCP
0%	**53.29**	109.92	**97.37**	215.06
1%	**361.22**	620.22	**693.49**	1194.41
2%	**506.54**	982.24	**993.17**	1705.99

(b) large latencies

Table 2. Comparison of the execution time (in seconds) of SCTP and TCP for a farm using 7 workers with a synthetic workload of 5000 tasks for long messages.

Loss	Latency(milliseconds) 0 Streams 1	0 Streams 10	20 Streams 1	20 Streams 10
0%	16.07	16.06	16.21	16.24
1%	16.34	16.46	26	25.75
2%	25.65	17.99	35.21	35

(a) small latencies

Loss	Latency(milliseconds) 40 Streams 1	40 Streams 10	80 Streams 1	80 Streams 10
0%	17.97	24.72	32	42.37
1%	53.70	52.04	95.95	94.32
2%	74.03	68.43	149.04	137.01

(b) large latencies

Table 3. Comparison of the execution time (in seconds) of 1-Stream SCTP and 10-Stream SCTP with $k = 10$ for short messages.

3.3.1 Head of line blocking

SCTP's congestion control mechanism and the ability to use multiple streams to reduce head of line blocking both contribute to the improved performance of SCTP over TCP. In order to isolate the effects of head of line blocking in the buffered farm template, we used a version of the SCTP module, one that uses only a single stream to send and/or receive messages irrespective of the message tag, rank and context. In all other respects the modules were identical.

In general, we found the improvements due to congestion control were more consistent and more significant than the elimination of head of line blocking. The benefit of using multiple streams was not as evident because not as many tasks were actively being sent by SCTP at the same time. We attempted to improve the farm template by allowing workers to request up to k tasks for each request message.

We performed the same experiments again with $k = 10$ tasks. The farm program was run at different loss rates for short message (12Kbyte). The results are shown in Table 3.

The results obtained show the effect of head of line blocking and the advantage due to the use multiple tags/streams at higher loss and increased latency. At 2% loss and 80ms delay, multiple-stream performs about 8% faster than single-stream. This shows that head of line blocking can affect performance in highly variable environment such as WANs, where loss and latency are an issue. It is possible to reduce the effect of head of line in the farm template by adding additional buffering. As shown in the experiments, head of line blocking was still a factor with $k = 10$.

4 Real world examples

In this section we consider two real-world message passing programs that are based on task farms. The first program is the parallel computation of a large correlation matrix. For this program, we used the buffered farm template described in the previous section. The second program is mpiBLAST, a parallel version of the BLAST bioinformatics tool. For mpiBLAST, rather than re-structure the program we focused on its communication and added the ability to make multiple requests and buffer tasks to the application. We describe the latency tolerance properties of each example according to model presented in the previous section. As well, we give performance results to show where and the extent to which these latency hiding techniques improved performance.

4.1 Robust correlation computation

Computing the correlation or covariance matrix is critical to many data mining activities. They are the basis for principle component analysis, dimensionality

reduction, as well as for detecting multidimensional outliers. Robust methods, like the Maronna method, are used because the classical techniques are sensitive to the presence of multidimensional outliers that distort their true values. The Maronna method is very computation intensive, however, there is a simple parallelization using task farming that can speed-up part of the computation [3]. We used the farm template from Section 3 to perform the task farming part of the computation.

The Maronna method takes as input a $m \times n$ matrix with m rows corresponding to samples and n columns corresponding to values of the variables for each sample. The method first computes the median and the median absolute deviation (MAD) for each column. These values for each column i, j are used to compute the correlation of variable i and j. Once the median and MAD have been calculated, the correlation calculation, which is a iterative process, can be performed independently for each i, j. The median and MAD computation are small relative to the correlation computation and we used the farm template to compute this phase. Once the median and MAD are computed and distributed to the workers, the farm template was used to do the correlation computation. The manager then gathered back the correlation results and assembled the final matrix.

The correlation computation was divided into a user specified number of tasks where each task was a set of correlations. It was sufficient to use a single integer to identify the range of correlations to be performed. Although it is possible to statically partition the work, experimentation had shown that it has better load-balancing when there are significantly more tasks than processors. Each task returned the correlation values of the set specified by the task. Hence, the result message is large in comparison to the task message.

For input, we used a gene expression dataset with 6028 entries that requires computing a 6028 by 6028 correlation matrix with over 18 million values. Table 4 compares the performance of the buffered farm program using SCTP-based middleware versus the performance of the same program for TCP-based middleware. As before, we have highlighted in bold the entries where SCTP performed better than TCP. The results show that as loss and latency increases, SCTP outperforms TCP by at least 50%. The Maronna results are similar to the synthetic farm results in Table 1. This is not surprising since both used same farm template, however, unlike the synthetic workload, there were other phases to the computation and tasks required different amounts of computation. Like the synthetic workload the task messages were small, short

	Latency(milliseconds)			
	0		20	
Loss	SCTP	TCP	SCTP	TCP
0%	28.90	28.90	**44.56**	102.56
1%	**56.20**	64.50	**368.48**	615.90
2%	**94.60**	142.00	**540.16**	990.42

(a) small latencies

	Latency(milliseconds)			
	40		80	
Loss	SCTP	TCP	SCTP	TCP
0%	**86.48**	178.52	**173.98**	414.50
1%	**704.73**	1187.37	**1314.30**	2092.24
2%	**1015.61**	1955.63	**1786.97**	3495.13

(b) large latencies

Table 4. SCTP versus TCP for various values of loss and latency, 6028 Gene expression dataset, split into 1000 tasks.

messages, but the results were long messages, approximately 140Kbytes. The results show the dramatic impact that loss can have on execution time, especially in the case of high latency where the execution time has gone from almost 30 seconds to almost 30 minutes. The major source of the increase is the rendezvous protocol for long messages which requires the worker to synchronize with the manager to return a result. SCTP alleviates the effect of loss or high latency on execution time, but when both loss and latency became large there simply was not a sufficient amount of computation to overcome the time needed to return results.

4.2 mpiBLAST application

mpiBLAST is an open source parallel implementation of BLAST, a widely used bioinformatics tool that searches for similarities between a given set of query sequences and a database of known DNA and protein sequences. For a given query, BLAST can be parallelized by segmenting the database among a set of machines and having each machine execute the query. It fits the processor farm template as described in Figure 2, where the manager consists of two processes, a scheduler and a writer. The scheduler sends the tasks to the worker that performs the query, and the writer is responsible for gathering the query results from each of the workers.

The original mpiBLAST has a task farm infrastructure by default, however it uses few tags and does not

buffer tasks at the workers. In executing the original mpiBLAST program, when there was no loss or added latency, the task execution time for a reasonably sized query greatly exceeded the time to obtain the next task and mpiBLAST provided excellent speed-up. In the case of segment loss there were cases where the workers idle waiting for another task, which indicates that buffering and multiple streams may improve the performance.

mpiBLAST consists of a scheduler, worker, and writer. Under little network variations, a worker's I/O search time and a writer's output time dominate a fixed portion of the total execution time, while the communication time is insignificant. However, as the network condition deteriorates, communication time increases greatly with respect to the fixed components of the total execution time. By adopting more tags and pipelining of requests, we believe that our modified mpiBLAST can adapt to the network conditions in a smoother fashion and utilize the machines better than the original.

To test these latency hiding techniques proposed in this paper, we modified the original mpiBLAST program to use tags as task IDs. In order to minimize the changes, we encoded the task number so that for any tag we can extract both the original mpiBLAST tag and the task ID with bitwise operations. We also modified mpiBLAST by adding buffers to allow each worker to have up to `MAXREQ` outstanding task requests. Requests were batched so that in response to each request message the scheduler sent `MAXREQ` tasks. In responding to requests, it is important to distribute the work fairly, otherwise initially some workers idle waiting for their first task.

In our experiments we used several different sized queries based on the Swiss protein database `swissprot`, which is about 66 Mbytes in size (segmented into 8 parts and pre-distributed). We used queries of size 30, 100 and 500 obtained from NCBI website[1]. The maximum number of outstanding requests made by each worker was 10 using 8 workers with varied latency and loss. The scheduler, writer and 8 workers processes are assigned round-robin to the 8 processors.

Table 5 compares the performance of modified mpiBLAST using SCTP-based middleware versus the performance of the same program for TCP-based middleware. As before, we have highlighted the entries where SCTP performed better than TCP.

When focusing on latency with no loss, the various latencies have little effect on the overall runtimes. This is due to the fact that task computation time in mpiBLAST is large and overall mpiBLAST requires little communication. In addition, what performance penalties mpiBLAST does suffer are alleviated by its new ability to make multiple task requests at once. This reduction in communication by the modified mpiBLAST is also the reason why loss with no latency similarly has little effect. There is simply not enough message passing for streams to help. However, as both loss and

[1] http://www.ncbi.nlm.nih.gov/

	Latency(milliseconds)			
	0		20	
Loss	SCTP	TCP	SCTP	TCP
0%	226.50	224.58	230.30	224.02
1%	229.72	226.46	**224.21**	237.20
2%	233.56	223.18	**228.54**	287.59

(a) small latencies

	Latency(milliseconds)			
	40		80	
Loss	SCTP	TCP	SCTP	TCP
0%	228.02	227.65	**231.92**	245.51
1%	**235.71**	325.25	**312.82**	529.30
2%	**260.09**	475.73	**410.26**	819.47

(b) large latencies

Table 5. Comparison of the execution time (in seconds) of the modified mpiBLAST program with 500 queries for SCTP and TCP.

	Latency(milliseconds)			
	0		20	
Loss	SCTP	TCP	SCTP	TCP
0%	73.2	75.0	**73.92**	77.63
1%	78.8	75.8	**77.25**	89.83
2%	75.9	75.8	**81.08**	100.82

(a) small latencies

	Latency(milliseconds)			
	40		80	
Loss	SCTP	TCP	SCTP	TCP
0%	78.4	78.0	**81.1**	89.1
1%	**84.6**	107.5	**102.9**	162.24
2%	**94.3**	149.2	**144.0**	271.6

(b) large latencies

Table 6. Comparison of the execution time (in seconds) of the modified mpiBLAST program with 100 queries for SCTP and TCP.

latency increase, the benefits of SCTP congestion control and our middleware design helps to reduce the effects of increased communication penalties. Moreover, head of line blocking was insignificant in mpiBLAST because the local compute time contributes the most to a search request.

In addition to the previous changes, we also changed the protocol used to send results to the writer. In the original mpiBLAST there is an explicit handshake where the worker sends the size of the result, the writer `ACKs` after which the sender sends the result. We eliminated the handshake protocol to make mpiBLAST less synchronous. These changes benefited mpiBLAST for SCTP and TCP. Table 7 shows a comparison between the original mpiBLAST and our modified version when running under no loss and no delay. The results shows a slight improvement in performance with the modified mpiBLAST.

	Protocol			
	SCTP		TCP	
Queries	Original	Modified	Original	Modified
100	81.80	73.27	81.18	75.06
500	240.58	229.83	241.21	224.57

Table 7. Comparison of the execution time (in seconds) of modified mpiBLAST to the original program with no loss and no added latency.

Some possible future improvements to mpiBLAST are adaptive scheduling algorithms, more efficient output writing with possible use of MPI-2 I/O features, and a more asynchronous mechanism for worker involvement in segment distribution.

5 Related work

There are several projects that have investigated the execution of MPI programs, which uses TCP, in heterogeneous environments such as the Internet or wide area networks. MPICH-G2 [11] and PACX-MPI [12]are both multi-protocol implementations of MPI for metacomputing environments that make it possible to link together clusters over wide area networks. MPICH-G2 and PACX-MPI use a two layer approach that takes advantage of vendor implementations of MPI inside a cluster and use TCP for communication between clusters. In our case, our implementation of MPI uses one protocol and the farm template we investigated supports heterogeneous collections of machines but was not specially tailored to the type of cluster of clusters environment targeted by MPICH-G2 and PACX-MPI. Some of the benefits of SCTP may be useful in this environment as well for use as the glue for communication between clusters.

Another project on wide-area communication for grid computing is NetIbis [5] that focuses on connectivity, performance and security problems of TCP in wide-area networks. NetIbis enhances performance by use of data compression over parallel TCP connections. Moreover, it uses TCP splicing for connection establishment through firewalls and can use encryption as well. The use of parallel connections in NetIbis could likely benefit from the design of our task template and our use of tags to identify independent streams. In [14] we discuss some of the challenges in using an aggressive design that attempts to map each independent communication stream into its own TCP connection.

Even though both SCTP and TCP sit atop IP, middleboxes like firewalls or boxes performing network address translation (NAT) are transport (i.e., L4 layer) aware and typically only work with TCP. However, some firewalls like the IPTables firewall within Linux, support SCTP. Even with full support in middleboxes, the fact that SCTP supports multi-homing presents some unique challenges. The IETF *behave* working group is currently researching what SCTP NAT traversal considerations are required [2].

MagPIe [13] is a library of collective communication optimized for wide-area networks that is built on top of MPICH. MagPIe constructs communication graphs that are wide-area optimal, taking the hierarchical structure of network topology into account. Their results show that their algorithms outperform MPICH in real wide-area environments. The collective communication routines used in our implementation did use SCTP, but they were the simple point-point based versions of the collectives and were not optimized to take advantage of multiple streams.

Several projects have used UDP rather than TCP [17, 1]. UDP is message based and can also be used to avoid head of line blocking and potentially avoids some of the "heavy-weight" mechanisms present in TCP and SCTP. However, when one adds reliability on top of UDP, the advantages begin to diminish. For example, LA-MPI, a high-performance, reliable MPI library that uses UDP, reports performance of their implementation over UDP/IP to be similar to TCP/IP performance of other MPI implementations over Ethernet [1]. WAMP [17] is an example of UDP for MPI over wide area networks. Interestingly, WAMP only

[2] http://www.ietf.org/internet-drafts/draft-stewart-behave-sctpnat-01.txt

wins over TCP in heavily congested networks where TCP's congestion avoidance mechanisms limit bandwidth. In any case, depending on the implementation, the techniques for latency hiding investigated in this paper could be used in these UDP-based implementations.

Recently OpenMPI [8] has been announced which is a new public domain version of MPI-2 that builds on the experience gained from the design and implementation of LAM/MPI, LA-MPI and FT-MPI [7]. The point-point communication modules in OpenMPI have the potential to manage messages and could also make use of tags to identify independent messages to help schedule messages across interfaces or interfaces that support multiple streams.

6 Conclusions

In this paper, we explored a class of programs suitable for WANs, namely task farms, through a simple performance model and a farm template, which maximally overlapped computation with communication. We showed that the farm template which uses buffering and our novel use of tags as task IDs, had better performance in SCTP-based middleware compared to TCP-based middleware. We also evaluated these techniques by experimenting with two real-world programs, mpiBLAST and robust correlation matrix computation. Our results have shown to be positive. The performance of farm programs can be improved in WANs by application decisions such as the ability to buffer tasks and intelligent middleware design to benefit from SCTP's multistreaming and improved congestion control mechanisms. Therefore in highly variable shared environment such as WANs, where there may be loss and increased latency, SCTP may be a better choice for MPI middleware.

6.1 Future work

We have investigated one group of MPI applications, namely the task farms in this paper. In the future, we are interested in exploring more classes of parallel programs and its real world applications that can potentially benefit with a MPI-SCTP combination. SCTP has shown to be more robust in a variable environment. Hence, we would like to simulate our experiments in different network topologies via Emulab [6], in a larger scale and a more heterogeneous environment. SCTP is gradually evolving and we believe as its functionalities mature, more benefits are yet to be explored.

References

[1] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. A. Taylor, T. S. Woodall, and M. W. Sukalski. Architecture of LA-MPI, a network-fault-tolerant MPI. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Sante Fe, New Mexico, April 2004.

[2] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G. E. Fagg. The open run-time environment (OpenRTE): A transport multicluster environment for high-performance computing, September 2005. EURO PVM MPI 2005 12th European Parallel Virtual Machine and Message Passing Interface Conference.

[3] J. Chilson, R. Ng, A. Wagner, and R. Zamar. Parallel computation of high dimensional robust correlation and covariance matrices. In *ACM International Conference on Knowledge Discovery and Data Mining, ACM KDD*, August 2004.

[4] A. E. Darling, L. Carey, and W. chun Feng. The design, implementation, and evaluation of mpiBLAST. In *ClusterWorld Conference & Expo and the 4th International Conference on Linux Clusters*, 2003.

[5] A. Denis, O. Aumage, R. Hofman, K. Verstoep, T. Kielmann, and H.E.Bal. Wide-area communication for grids: an integrated solution to connectivity, performance and security problems. In *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pages 97–106, June 2004.

[6] EmuLab. Network Emulation Testbed. http://www.emulab.net/.

[7] G. E. Fagg and G. E. Dongarra. Building and using a fault-tolerant MPI implementation. *International Journal of High Performance Computing Applications*, 18(3):353–361, 2004.

[8] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[9] H. Kamal, B. Penoff, and A. Wagner. SCTP versus TCP for MPI. In *Supercomputing '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2005. IEEE Computer Society.

[10] KAME. SCTP stack implementation for FreeBSD. http://www.kame.net.

[11] N. T. Karonis, B. R. Toonen, and I. T. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *CoRR*, cs.DC/0206040, 2002.

[12] R. Keller, E. Gabriel, B. Krammer, M. S. Mueller, and M. M. Resch. Towards efficient execution of MPI

applications on the grid: Porting and optimization issues. *Journal of Grid Computing*, 1:133–149, June 2003.

[13] T. Kielmann, H. E. Bal, S. Gorlatch, K. Verstoep, and R. F. Hofman. Network performance-aware collective communication for clustered wide-area systems. *Parallel Comput.*, 27(11):1431–1456, 2001.

[14] B. Penoff and A. Wagner. Towards MPI progression layer elimination with TCP and SCTP. In *11th International Workshop on High-Level Programming Models and Supportive Environments (HIPS 2006)*. IEEE Computer Society, April 25 2006.

[15] M. A. S. Fu and W. Ivancic. SCTP over satellite networks. In *IEEE Computer Communications Workshop (CCW 2003)*, pages 112–116, Dana Point, Ca., October 2003.

[16] R. R. Stewart and Q. Xie. *Stream control transmission protocol (SCTP): a reference guide*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[17] R. Vinkat, P. M. Dickens, and W. Gropp. Efficient communication across the Internet in wide-area MPI. In *Conference on Parallel and Distributed Programming Techniques and Applications*, Las Vegas, Nevada, USA, 2001.

[18] A. Wagner and M. Parsa. Load balancing in task farms. In *Int'l Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, August 1997.

Biographies

Humaira Kamal, received a MSc degree in computer science from the Lahore University of Management Sciences. She has completed a MSc at the UBC under the supervision of Alan Wagner. She is currently an instructor a Lahore Institute of Management Science, Lahore, Pakistan.

Brad Penoff, received a BSc degree from Ohio State University, he has worked at SUN Microsystems. He is an MSc student at UBC under the supervision of Alan Wagner. He recently completed a two month summer internship at Argonne National Labs working on MPI related software.

Mike Tsai, received a BSc degree from University of British Columbia, He is currently an MSc student at UBC.

Edith Vong, is currently completing her BSc degree in computer science from University of British Columbia.

Alan Wagner, received his PhD from the University of Toronto in 1987. He joined the Computer Science Department at the University of British Columbia in 1987, where he is an Associate Professor.