

# Towards MPI progression layer elimination with TCP and SCTP

Brad Penoff and Alan Wagner  
University of British Columbia, Vancouver, BC, Canada  
{penoff,wagner}@cs.ubc.ca

## Abstract

*MPI middleware glues together the components necessary for execution. Almost all implementations have a communication component also called a message progression layer that progresses outstanding messages and maintains their state. The goal of this work is to thin or eliminate this communication component by pushing the functionality down onto the standard IP stack in order to take advantage of potential advances in commodity networking. We introduce a TCP-based design that successfully eliminates the communication component. We discuss how this eliminated TCP-based design doesn't scale and show a more scalable design based on the Stream Control Transmission Protocol (SCTP) that has a thinned communication component. We compare the designs showing why SCTP one-to-many sockets in their current form can only thin the communication component. We show what additional features would be required of SCTP to enable a practical design with a fully eliminated communication component.*

## 1 Introduction

Middleware is an essential part of any MPI system. MPI was designed from the onset to be independent from the execution environment and its communication primitives were designed to take advantage of specialized interconnects. As a result, MPI has needed middleware for starting the execution, runtime support for managing processes, and a communication middleware component to support interoperability across different interconnects. The communication middleware component typically has a module associated with each different interconnect that is specific to the characteristics of that particular interconnect. In addition to specialized interconnects, the public domain versions of MPI (MPICH [15] and LAM [2]), have always supported TCP/IP, to allow MPI to operate in local area networks.

In this paper we focus on the alternative network architecture for MPI shown in Figure 1. Rather than including extensive support for different interconnects in the middleware, we investigated middleware designs that could take advantage of standard transport protocols that use IP for in-

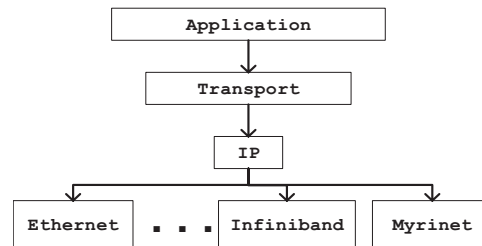


Figure 1. IP based middleware

teroperability. One advantage to the network architecture shown in Figure 1 is that it simplifies the middleware, which could lead to a more standardized MPI library that could improve the interoperability between different MPI implementations. MPI middleware has never been standardized and each implementation uses its own design. For example, MPICH's ADI [14] and LAM's RPI [11] describe their layering of the middleware. OpenMPI [4] is a more recent project with the goal of modularizing the middleware to support mixing and matching modules according to the underlying system and interconnect. The use of IP substantially reduces the complexity of designing MPI middleware in comparison to designs that try to exploit the particular features of an interconnect, which is one source of incompatibilities between implementations.

The advantage of the usual implementation of the communication middleware component is that it can fully exploit the performance of specialized interconnects. However, these interconnects all have implementations over IP and the affordability and performance of Ethernet make TCP/IP widely used for MPI. With the increased use of more commodity processors and commodity OSes, an interesting research topic becomes how much the use of commodity networking can be exploited if this commoditization trend is to continue. One initial concern is performance. If one compares microbenchmarks, TCP/IP achieves less bandwidth and significantly higher latency than customized stacks and interconnects. But microbenchmarks can be misleading so they should be carefully interpreted; improved bandwidth and latency do not always produce the same gains at the application level. For example, the performance of TCP/IP over Ethernet was measured to be within

5% of the performance of Myrinet on the NAS benchmarks and even closer with more efficient TCP implementations [9]. The advantage of using commodity networking (i.e., TCP/IP) is that it can continue to leverage the advances in mainstream uses of networking that share many of the same performance demands as MPI programs.

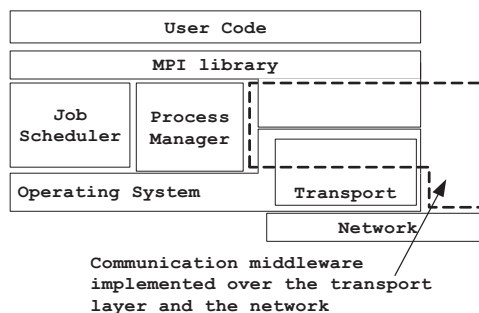
Given the network architecture shown in Figure 1, then it is interesting to investigate re-designs of the middleware that can leverage standardized transport protocols over IP with a view to simplifying and eliminating the communication middleware. This approach differs from the design of current public domain implementations of MPI in that it attempts to push functionality down into the transport layer rather than pull it up into the middleware. For example, OpenMPI sequences all its data within the middleware and stripes it across all available interconnects, managing message assembly and connection failures within the communication component. If one were to limit the interconnects to those only atop IP, it could be possible to prevent doing such management in the middleware if networking research such as Concurrent Multipath Transfer [7] were exploited, where the data is sequenced and striped within the transport layer instead.

We introduce a new TCP socket per message stream design, and later we consider an SCTP-based design. SCTP ([13]) is a newly standardized transport protocol for IP that has a number of new features that make it an interesting target for MPI middleware. In previous work [8], we designed and implemented our own SCTP-based version of the middleware with LAM; here we review its design but more from a networking point of view since we are focusing on the overall goal of communication component elimination.

The contribution of this work is that it identifies the extent to which we can take advantage of standardized transport protocols and commodity networking by simplifying and eliminating MPI middleware functionality. We discuss the requirements of MPI messaging in terms of demultiplexing, flow control, and the communication management properties of the underlying transport protocol. The socket per message stream design leads to a simple implementation of MPI that completely eliminates the communication middleware layer, however, it does not scale. We show how using SCTP thins the middleware and avoids the scalability problems, however SCTP one-to-many sockets in their current form present limitations that make it difficult to completely eliminate the communication middleware component. We discuss what would be required of SCTP one-to-many sockets for full elimination.

## 2 MPI middleware

Middleware is an essential piece of MPI. As with any parallel programming model, an MPI application requires a



**Figure 2. Communication Middleware**

variety of services in order to properly run. These services are each glued together by the middleware [3]; they can be conveniently split into three interacting components: (a) a job scheduler to determine which resources will be used to run the MPI job, (b) a process manager is needed during runtime for process initialization/shutdown, signal delivery, error detection as well as process stdin/stdout/stderr redirection, and finally (c) a parallel library such as MPI to provide a mechanism for parallel processes to effectively exchange messages. In this paper, we focus on the MPI library.

### 2.1 MPI library

The MPI library offers a variety of routines to help application programmers effectively develop parallel programs. The middleware exports the API as defined by the MPI standard and the application links to this library. The MPI API routines are typically implemented with the help of other functions which in turn call the appropriate lower layer system and device routines. Often MPI implementations further modularize these other functions into components to allow for maximum flexibility of hardware and algorithm choice. For example, amongst other things, OpenMPI lets you select at runtime which hardware to run on, which datatype implementation to use, and which reduction operation algorithm to perform.

As shown in Figure 2, one obtains a layered system where the MPI library code calls into lower level modules for job scheduling, process management and communication (represented by the dashed line). The communication middleware consists of one more modules that call the operating system in the case of TCP and possibly lower level network layer functions in the case of specialized interconnects like Infiniband. Our intent is to remove the direct access to the network layer and remove the communication middleware by dividing its functionality between the MPI library and an IP transport protocol.

The existence of non-blocking communication, in addition to other types of messaging, and the need to match sends and receives makes it difficult to completely execute the MPI communication routine as a simple library function. Typically message progression state has to be

maintained in-between MPI calls. The message progression layer ensures that all messages sent are eventually received.

## 2.2 Message Progression Layer

The communication component, also called the message progression layer, copies the message, allocates and frees memory when necessary, enforces the MPI semantics for the different message types, performs message matching, executes the lower level communication protocol, constructs the appropriate message header, and finally sends and receives data from the underlying transport layer. The various public domain versions of MPI define their interfaces for message progression in a variety of ways. In LAM, there is one layer for this purpose called the request progression interface (RPI), while in MPICH it is a combination of two layers, an abstract device handling MPI semantics and a lower level channel that acts as a byte mover. Similarly, OpenMPI splits the MPI semantics off into its own layer called the PML (point-to-point management layer). However, underneath this layer it has two other layers for message progression, a simple byte transfer layer (BTL) as well as a BTL management layer that attempts to stripe data across all underlying interconnects([16]).

The three main tasks we will focus on in the message progression layer are the message matching, managing of expected/unexpected messages, and the long/short message protocols that are typically used.

*Message Matching:* Message matching in MPI is based on three values specified in the send and receive call: (i) context, (ii) source/destination rank, and (iii) tag. The context identifies a group of processes that can communicate with each other. Within a context, each process has a unique identification called rank that is used to specify the source and destination process for the message. Finally there is a user defined tag value that can be used to further distinguish messages. A receive call matches a message when the tag, source rank and context (TRC) specified by the call matches the corresponding values. MPI semantics dictate that messages belonging to the same TRC must be received (i.e. completed) in the order in which the sends were posted.

Message matching is a form of demultiplexing that identifies the particular receive call that is the target of the sent message. It is also possible that wildcards are used in a receive call to match messages from any source and/or any tag. Wildcards provide a form of non-deterministic choice where a process can receive from some collection of messages sent to the process.

*Expected/Unexpected Messages:* Most implementations of MPI perform message matching by managing two structures: one for expected messages (local receive requests) and one for unexpected messages (remote send requests). When the message progression layer processes a receive request, it can search the unexpected queue for a matching

send request and, if not found, add itself to the expected message structure. In the case of a send request, the message progression layer sends the request to the remote process where the message progression layer on the remote process checks its expected message structure for a match and, if not found, adds the request to the unexpected message structure.

The management of the expected and unexpected message structures is simple in principle but there are several subtle issues that complicate it. First, in order to avoid potential deadlock, the message progression layer must be able to accept incoming send requests. The MPI standard defines a “safe program” as a program that correctly finishes when every communication is replaced by a synchronous communication, which effectively implies that the progression layer needs to be able to accept only a single send request. The guaranteed envelope resource protocol [1] extends this guarantee to a bounded number of requests. Second, because of weak message progression<sup>1</sup>, each MPI call needs to progress messages by updating the state of the expected and unexpected message structures and sending and receiving data to and from the transport layer below. The message progression layer needs to maintain state between the MPI calls since it will need to progress messages for other calls; for example, it needs to keep checking the transport layer for remote send requests from other processes to be matched or added to the unexpected message structure.

*Short/Long Messages:* In terms of network level functions, long and short messages in MPI introduce problems associated with message framing and flow control. The previous discussion only considers requests and assumes that once requests have been matched the transfer can now be completed. In MPI this is complicated by the fact that messages can in principle be arbitrarily long. Since the system does not have an unlimited amount of buffers, at some point in the transfer, parts of the message must use the user level buffer space given as a parameter to the receive call. As a result, for a sufficiently large message, the communication becomes synchronous where the send call cannot complete before the receive has been posted. Furthermore, depending on the transport layer used, large messages may need to be fragmented and reassembled on the receive side. Fragmentation may also be necessary for fair message delivery to ensure that the message progression layer doesn't block for long periods of time while engaged in the transfer of one large message.

Given that the message progression layer must accept send requests and also manage the transfer of large messages, then a natural solution is to handle short and long messages differently. Short messages are bundled with the request and thus can be immediately copied into the user's

---

<sup>1</sup>We focus on implementations that assume a weak message progression rule where messages are progressed only during MPI calls.

memory when matched. Long messages use a rendezvous protocol that first matches the two requests and then arranges a transfer of the message from the user's send buffer on the remote side to the user's receive buffer. There is an additional benefit to this approach for performance. As Gropp describes for MPICH [14], if one considers memory copying and the cost of the rendezvous, then when message copying costs exceed rendezvous costs it is more efficient to use rendezvous, assuming it can avoid the extra copying, rather than using eager send.

### 3 TCP socket per Message Stream

In this section we describe a TCP-based implementation of the MPI library that eliminates the message progression layer altogether, following Figure 2. By elimination, we mean that such a layer is no longer necessary to maintain the hidden state required to progress messages. Since messages still need to be advanced, we eliminate the message progression layer by moving some functionality into the MPI library routines and the remainder down into TCP as illustrated in Figure 2. Although this design is not very scalable, for reasons to be discussed, it serves to illustrate the features that are needed in the library and transport layer when having an MPI implementation where the message progression layer is eliminated. After sketching the design, we discuss the problems.

#### 3.1 Implementation

The standard socket library provides routines that can be used to implement the MPI communication primitives. There are several differences between the standard MPI send and receive and the corresponding socket library calls. Some of the differences can be easily accommodated. For example, message length is expressed in terms of MPI types, which can be converted within the MPI library to its actual byte length using information contained in the `MPI_Datatype`. The other part of the message is the communicator, source/destination, and tag fields. The design attempts to map these into a socket descriptor. As a result, every TRC will have its own socket and TCP connection, which is consistent with the message ordering semantics of MPI.

The `MPI_Init()` routine can be used to exchange the information necessary to initiate the system. In particular, at a minimum, at runtime we need the IP addresses of all machines running MPI processes. We assume there is a pre-assigned control port for each MPI process. This is used for the `accept()` socket call when creating new connections initiated by a `connect()` call to that port from another MPI process. When a new connection is created on this control port using `accept()`, a new socket using its own port is returned and the control port can continue to be reused

as a control port to initiate other connections. The control IP and port are made available for each rank by way of the LAM out-of-band daemons. The `MPI_Init()` routine creates the `MPI_COMM_WORLD` communicator, an opaque data structure, which for each process, contains the table of the machines in `MPI_COMM_WORLD`. More specifically, the table gives a mapping from MPI ranks to IP addresses and control port. Since the communicator object is an argument to all communication calls, every call has access to this table.

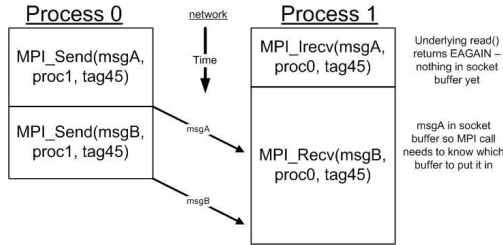
As well as the rank-IP table, communicators also maintain a separate "connections" table that is a mapping from a TRC to a socket descriptor. Each MPI send or receive uses the TRC value as a key to find the corresponding socket for the send and receive socket call. If there is no entry for that key, then it uses the rank to determine the IP address and port. The control port is then used to connect to the remote machine and create a new connection for that TRC. To create a new connection, one end must execute `accept()` whereas the other end must execute `connect()`. Because of wildcards, the receive side executes the `accept()` and the send side executes a corresponding `connect()`. Note, this is only done for the first connection. Having sketched out the basic scheme, we will now consider the message progression layer functionality with respect to the requirements described in Section 2.2.

*Message Matching:* The basic scheme works for the standard send and receive, however, matching is complicated by the existence of wildcards. Wildcards are only used in MPI receive calls (`MPI_ANY_SOURCE`, `MPI_ANY_TAG`). The `select()` socket call allows us to create a set of socket descriptors that can be used to block while waiting for data on one of the connections. The MPI library code scans the connections table to create a set of sockets that can match the receive and can then be used in a `select()` call.

It is possible that there is a matching TRC for which a connection does not yet exist so a new connection will need to be established. New connections can be handled by adding the socket associated with the control port to the `select()` call. However, there may be connect requests from one or more processes not associated with the receive call. As a result, once a new connection is accepted, the TRC information from the remote send side needs to be sent to the receive side in order for it to process the connection request and update the connections table associated with the communicator. This was the rationale for having receive do `accept()`, since these calls can receive from multiple processes and need to use `select()`, whereas the sends do not use wildcards and they either already have a pre-assigned socket or the send needs to create a new connection using `connect()`.

The receive call that executes the `select()` may be





**Figure 3. Motivation for expected queues**

notified of data on an existing connection or notification for a new connection, which must be added to the connection table. Depending on the implementation, it is possible that the request is for a different communicator, which implies that the tables for `MPI_COMM_WORLD` and all communicators derived from it must be accessible. An alternative approach, since creating a communicator is a global operation over the ranks in the new communicator, is to negotiate a new control port to be stored with each new communicator. This eliminates the possibility of receiving a connection request for a communicator different from the one specified in the receive call since every communicator will have its own control port and communicators can never be wildcarded.

*Expected/Unexpected Messages:* The global expected and unexpected message structures are no longer needed under this design. Because each MPI message stream (TRC) now has its own connection, for the unexpected queue the implementation relies instead on the socket buffers and the TCP flow control mechanism to manage the flow of messages on a connection once it has been created.

Non-blocking is possible with sockets by setting the non-blocking option on the socket (`O_NONBLOCK`). For most socket libraries, it is possible to set the option on a per call basis, which corresponds to MPI, rather than having it as part of the connection [12]. Non-blocking communication in MPI returns an opaque message request object (`MPI_Request`) that is then passed to all calls that check for completion. The request objects can be used to maintain the information about what is needed to complete the MPI call. For example, in the case of a synchronous communication, the request object stores whether the message corresponding to the send or receive has been posted.

The presence of blocking and non-blocking calls in MPI creates potential scenarios where the state of a particular message's progress will be dependent on the progress of other previously posted messages. As a result, a local form of the expected message queues must be kept. Suppose we did not have an expected queue and consider the scenario shown in Figure 3. If messages A and B were successfully put on the network from process rank 0 to process rank 1 on tag 45 using blocking sends. Process rank 1 has a non-blocking receive for message A and a blocking call for message B. When process rank 1 pre-posts a request for message A, if that message isn't in the socket buffer, an

`EAGAIN` would be returned and the application would proceed. Next, during the blocking receive call for message B, process rank 1 must be aware of the request for message A since it is going to arrive to the socket first due to TCP's full sequencing of a connection. The library must know which buffer to put it in since in this design, no global expected queue exists.

In order to process requests correctly, another table is maintained within the opaque communicator object. This table maps a TRC value to the head of a linked list of request objects. Effectively the table contains more local expected message queues. A linked list itself is outstanding requests on a given TRC; due to MPI semantics, these requests must be completed in order. For receive requests, access to this linked list is required so that data obtained from the socket buffer can be placed in the correct user-space buffer as specified in the posted MPI receive call. For send requests, messages need to be sent in the order in which they were posted so this linked list is also required to be available. This table is essentially an expected queue within the opaque object, the only difference being that instead of a linked list of requests for all TRCs as in standard message progression layers, it is a table hashed by the TRC returning a smaller, more local linked list.

*Short/Long Messages:* In this design, a connection corresponds to a particular TRC. TCP's in-order semantics ensure that messages are completed in the order they are sent. As well, TCP's flow control mechanism ensures that the eager sending of messages will not over-run the receiver's buffer and exhaust memory resources. This eliminates one of the motivations for short and long messages and the use of a separate protocol for each kind. The important point is that we can take advantage of TCP's flow control mechanisms.

This design simplifies the middleware since it is no longer necessary to have a short and long protocol that restricts the eager sending of larger messages. In the message progression layer within typical implementations, flow control is handled by complicating the protocol, restricting the eager sending of large messages.

### 3.2 Critique of the Implementation

There are several issues that arise in using a socket for each TRC. These issues generally arise in any TCP implementation of the middleware, including the standard TCP implementation used in public domain implementations of MPI([10, 8]). The issues are discussed in the following sections.

*Number of Sockets:* The number of allowed file descriptors is determined by `FD_MAX` (typically 1024 on Linux). On larger systems, even the standard TCP implementation can exhaust the supply of file descriptors, when it opens one socket for each MPI process in the system. For this reason,

both MPICH and OpenMPI open connections as they are needed rather than opening all connections during a call to `MPI_Init()`, as is the case for LAM. The socket per TRC design makes this problem even worse. The design may be acceptable for small clusters, but has difficulty scaling to a large number of processes or with programs that may use a large number of tags and contexts.

The number of file descriptors can be reconfigured in the kernel, however, there is also a significant memory cost associated with each TCP connection, which again prohibits having large number of connections. Recent work by Gilfeather and Maccabe [6] address the problem of the scalability of connection management in TCP and introduce some techniques for alleviating these problems in clusters. The origin of their work was from similar problems that arise in web servers, which also has to manage a large number of connections and where techniques have been developed to efficiently handle these connections. This work is a good example of an advantage of using commodity transport protocols like TCP for MPI since one can take advantage of state-of-the-art research on web servers.

*Number of System Calls:* The second issue is a performance issue that arises because of the way in which the middleware must poll the sockets that results in a large number of system calls. As described in [10], the message progression layer usually needs to execute a `select()` call to determine the next socket with available data and then needs one or more `read()` calls to obtain the message. This results in several system calls for each MPI call, which not only requires a context switch, but also extra processing because it is a kernel call [12]. In [10], they address this problem by managing the expected and unexpected message structures inside the kernel.

*select System Call:* A third issue is the performance of the `select()` call, which is known to scale linearly with the number of sockets. For example, as shown in [10] on a 3.06GHz dual Xeon machine, the cost of `select()` increases linearly from under 100 microseconds for less than 200 sockets to 900 microseconds for 1000 sockets. Newer system calls such as `epoll()` for Linux, and similar calls in other operating systems, have tried to improve the event handling performance of web-servers [5]. The socket per TRC design still requires frequent `select()` calls particularly with MPI calls having wildcards.

*TCP Flow Control:* TCP flow control is a concern in any TCP-based implementation. Since buffering is being done by the transport layer, it is possible that when MPI receive calls are delayed, the socket buffers fill and as a result trigger TCP flow control to close the receive window. Closing and opening the advertised receive window in TCP restricts bandwidth because of TCP built-in timers and slow-start mechanisms. This is particularly serious for high bandwidth connections.

There is a mismatch between the event driven operation of the transport layer and the sequential control operation of the MPI program. Flow control at the transport layer attempts to match the sender's data rate to that of the receiver. Messages in MPI are burstier and can easily trigger TCP's flow control mechanism. Flow control is an end-to-end mechanism and thus not only reduces bandwidth but also significantly adds to message latency. The standard TCP implementation of the progression layer constantly empties the socket buffers to reduce the chances of it filling. Also, since all traffic between two processes is aggregated on a single connection it tends to smooth out some of the burstiness. The socket per TRC does not aggregate traffic and is far more likely to trigger TCP flow control.

On the other hand, our socket per TRC design does make it possible for the MPI program to take advantage of TCP's flow control mechanism since it possible to push back on an individual flow by delaying receiving data on a particular socket. At the user level, flow control allows the user to configure and throttle the data rates on a particular TRC. However, the end-to-end nature of TCP's flow control mechanism makes such fine-grain flow control expensive in high latency network environments.

In LAM there are flow control mechanisms within the message progression layer itself rather than only using the flow control of the underlying transport protocol [1]. By pushing this functionality down into TCP, the socket per TRC design prevents the MPI implementor to have to design a user level flow control mechanism and instead lets them leverage the transport layer's flow control mechanism, including potential advances that may occur here by research in the networking community.

## 4 SCTP-based MPI Middleware

We have implemented a version of the MPI middleware, as a new RPI for LAM, that takes advantage of these new features. The features of SCTP that had the most influence on the design are the following: (a) SCTP introduces associations and streams, which generalizes the notion of a connection. (b) SCTP can operate in a one-to-one manner where, like TCP, there is an association between two machines, or it can operate in a one-to-many manner where a single socket can receive messages from multiple clients and where there are multiple associations, one for each client. (c) SCTP, like UDP, is message-based and each socket call returns a complete message. In the following section we present the implementation of an SCTP-based message progression layer.

### 4.1 Implementation

In SCTP, each association between endpoints can have multiple streams. Messages within a stream are delivered

in order, but messages between streams can be delivered independently of one another. As a result, unlike a TCP connection, segment loss or out-of-order delivery on one stream does not block the delivery of messages on other streams (i.e., no head of line blocking). We take advantage of streams in our SCTP-based implementation of MPI by mapping different MPI context and tag combinations onto different streams. The use of an association for each rank and a stream for each context and tag combination results in a mapping of each TRC to its own SCTP stream. This uses only a single one-to-many socket. The mapping of each TRC to its own stream is semantically equivalent to the mapping of TRCs to sockets in Section 3. Both satisfy the MPI semantics of in-order message delivery, however, SCTP avoids many of the disadvantages discussed in Section 3.2.

In the remainder of this section, we discuss the thinned SCTP-based message progression layer with respect to message matching, expected/unexpected messages, and the short/long message protocol. We describe what features are required of SCTP for full message progression layer elimination.

*Message Matching:* In SCTP, we used a one-to-many socket that is similar to a UDP socket. Using socket calls, it is possible to specify the association (destination machine) and stream on which to send messages. Again, just as in the case of one socket per TRC, we can store this information in the communicator that is a parameter to each send call. Ideally, on the receive side we would like to receive the next message on a given association and stream; this is because TRCs are mapped to streams which maintain the order that messages were sent, adhering to MPI messaging semantics. However, SCTP does not support this functionality for streams or associations in the one-to-many style and simply returns the next available message on any association and any stream.

Semantically, the provided `sctp_recvmsg()` socket call is equivalent to an `MPI_Recv` with `MPI_ANY_RANK` and `MPI_ANY_TAG`. As a result, for MPI receive calls that do not use the wildcards, it is necessary to do message matching. It is thus not possible to eliminate message matching from the design without maintaining global expected and unexpected message queues.

SCTP one-to-many style does simplify the socket calls needed to receive messages since no `select()` call is required because only one socket is used; implementing it using a single one-to-many socket makes the message progression layer thinner. Each `sctp_recvmsg()` socket call returns a full message that has either to be delivered or added to the unexpected message structure. In comparison to the socket per TRC design in Section 3, it avoids the costly `select()` call.

*Expected/Unexpected Messages:* Since we need to do mes-

sage matching, our message progression layer still needs to maintain expected and unexpected message structures. Each of these queues needs to be global across all TRCs because the SCTP API does not let one specify the stream and association on the `sctp_recvmsg()` call. In the socket per TRC design, these queues were eliminated from within the progression layer; their functionalities were effectively split. For the expected message queue, a more local form of the queue was pulled up into the MPI library, having a hash of queues keyed by the TRC. The ability to fully specify the TRC within the `recv()` call was a trait of the transport protocol so expected message queues were split between the MPI library and the transport protocol. For unexpected message queues in the socket per TRC design, they were pushed fully down onto the transport protocol. Traditional designs such as LAM place the queues in the message progression layer. In any design, while possible to maintain these queues in the MPI library and attach them to a communicator, their mere existence and necessity shows that progression layer elimination with SCTP using a one-to-many socket can not push as much functionality down onto the transport as is possible in the socket per TRC design unless the SCTP API provided additional capabilities. These capabilities include the ability to receive messages on a particular association and stream as well as the ability to ask the socket which streams currently have data that can be read without blocking (i.e., something similar to the functionality of `select()`).

Although, every `sctp_recvmsg()` call returns a message, we continue reading from the socket if there are outstanding requests in order to obtain and deliver as many messages as possible. Like TCP, SCTP has a socket option that can be set so that I/O functions called on that socket return immediately when the socket call would block (returning `errno EWOULDBLOCK`); for non-blocking MPI calls, we can exit the message progression layer when this `errno` is returned. In addition, the SCTP receive socket buffer size can be set by the user, but all streams and associations share the same buffer, congestion control settings, and flow control settings. To avoid triggering the flow control mechanisms, it is better to empty the buffer as often as possible; this is why we read as much as we can from the socket when we enter the message progression layer. Given no other user input, this design decision is likely the best choice since it progresses all messages that are already reside at the receiver.

*Short/Long Messages:* The need to manage unexpected messages implies that we will need a short and long message protocol to avoid resource exhaustion.

The first issue that arises is handling messages which exceed SCTP's socket send buffer size. Messages that exceed this size need to be fragmented and re-assembled by the message progression layer. We use a rendezvous



mechanism where, once the rendezvous has occurred, the sender fragments the MPI message into fragments and the receiver assembles these fragments in the user's receive buffer as specified in the MPI receive call. The sender could loop sending each fragment, however, this does not allow other messages to advance. As a result, we used round robin scheduling of all out-going SCTP messages and recorded their progress in a structure pointed to from the `MPI_Request`. This allows MPI messages with different TRCs to progress concurrently on an association between two machines. Special care had to be taken to eliminate race conditions that could occur when MPI messages had the same TRC [8].

The second issue is flow control. As previously mentioned, it is not possible in SCTP to request the next message on a particular stream and association. A reason for not providing this functionality is that streams share the receive socket buffer inside the transport layer. Providing a mechanism for obtaining messages on a particular stream introduces the possibility of messages not being selected and eventually exhausting the socket buffer resulting in deadlock. The potential for deadlock would be difficult to detect because it depends on the possible ordering of data segments received by the transport layer.

The socket per TRC design was able to eliminate the short/long message protocol because it could rely on TCP's flow control mechanism. Although the consequences of closing the advertised receive window makes the use TCP's flow control costly, a push back mechanism would give the MPI program control over which message to advance. This would allow users the ability to advance messages on its critical path and potentially improve the overall execution time of the program.

SCTP cannot provide the level of fine grain flow control that would be necessary to allow the user to push back on a particular stream. SCTP streams and thus TRCs share flow control values. As a result, one TRC can affect the performance of another. To try to combat this, it is possible in the SCTP one-to-many socket style to allocate socket buffer space on a per association basis, which would give more control over flow control from a particular machine, but there still is no call provided in the API to tell which associations or streams have data ready to be read.

## 5 Conclusions

In this paper, we discussed and compared MPI designs using TCP and SCTP. The goal of these designs is to thin or eliminate the communication component from the MPI middleware. This is done by pushing functionality commonly present in this component down onto a transport protocol. As a result, MPI implementations are simplified and in addition, they can leverage advances in networking pro-

tol research instead of having to implement certain functionalities in the middleware. Our designs illustrate some limitations of TCP in terms of scalability and of SCTP in terms of missing features. For SCTP, because of the lack of stream-level flow control and the ability to receive from a particular stream, it is not possible to completely eliminate the message progression layer with something more scalable than a socket per TRC. However, SCTP scales better than TCP, it avoids the head of line blocking that can occur in standard TCP implementations and it provides more opportunity for fairer concurrent message transfer in the case of longer messages.

MPI designs that make extensive use of standard protocol stacks may provide a solution to interoperability among interconnects and can take advantage of commodity networking as it continues to gain momentum.

## References

- [1] G. Burns and R. Daoud, "Robust message delivery with guaranteed resources," in *Proc. of MPI Developer's and User's Conference (MPIDC)*, May 1995.
- [2] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In *IEEE/ACM Supercomputing*, pages 379–386, 1994.
- [3] R. Butler, W. Gropp, and E. Lusk. Components and interfaces of a process management system for parallel programs. *Parallel Computing*, 27(11):1417–1429, 2001.
- [4] E. Gabriel *et al.*, "Open MPI: Goals, concept and design of a next generation MPI implementation," in *Euro PVM/MPI 11*, Sept 2004.
- [5] L. Gammo, T. Brecht, A. Shukla, and D. Pariag, "Comparing and evaluating epoll, select, and poll event mechanisms," in *Proc. of the Linux Symposium*, July 2004.
- [6] P. Gilfeather and A. B. Maccabe, "Connection-less tcp," in *IPDPS*, 2005.
- [7] J. R. Iyengar, K. C. Shah, P. D. Amer, and R. Stewart, "Concurrent multipath transfer using SCTP multihoming," in *SPECTS 2004*, July 2004.
- [8] H. Kamal, B. Penoff, and A. Wagner, "SCTP versus TCP for MPI," *ACM/IEEE SC-05*, Nov 2005.
- [9] S. Majumder and S. Rixner, "Comparing Ethernet and Myrinet for MPI communication," in *LCR '04: Workshop on languages, compilers, and run-time support for scalable systems*. New York, USA: ACM Press, 2004, pp. 1–7.
- [10] M. Matsuda, T. Kudoh, H. Tazuka, and Y. Ishikawa, "The design and implementation of an asynchronous communication mechanism for the MPI communication model," in *IEEE Intl. Conf. on Cluster Computing*, Sept 2004.
- [11] J. M. Squyres, B. Barrett, and A. Lumsdaine, "Request progression interface (RPI) system services interface (SSI) modules for LAM/MPI," Indiana University, CS Dept. TR579, 2003.
- [12] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming, Vol. 1, 3rd Ed.*. Pearson Education, 2003.
- [13] R. Stewart *et al.*, "The Stream Control Transmission Protocol (SCTP)," IETF RFC2960 Oct 2000.
- [14] W. Gropp and E. Lusk, "MPICH working note: The implementation of the second generation MPICH ADI," Tech. Rep. ANL/MCS-TM, 2005.
- [15] W. Gropp, E. Lusk, N. Doss and A. Skjellum, "High-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sept 1996.
- [16] T. Woodall *et al.*, "TEG: A high-performance, scalable, multi-network point-to-point communications methodology," in *Euro PVM/MPI 11*, Sept 2004.