Designing Dynamic and Adaptive MPI Point-to-point Communication Protocols for Efficient Overlap of Computation and Communication *

H. Subramoni, S. Chakraborty, and D. K. Panda
Department of Computer Science and Engineering, The Ohio State University, Columbus, OH
{subramoni.1, chakraborty.52, panda.2}@osu.edu

Abstract. Broadly, there exist two protocols for point-to-point data transfer in the Message Passing Interface (MPI) programming model - Eager and Rendezvous. State-of-the-art MPI libraries decide the switch point between these protocols based on the trade-off between memory footprint of the MPI library and communication performance without considering the overlap potential of these communication protocols. This results in sub-par overlap of communication and computation at the application level. While application developers can manually tune this threshold to achieve better overlap, it involves significant effort. Further, the communication pattern may change based on the size of the job and the input requiring constant re-tuning making such a solution impractical. In this paper, we take up this challenge and propose designs for point-to-point data transfer in MPI which accounts for overlap in addition to performance and memory footprint. The proposed designs dynamically adapt to the communication characteristic of each communicating pair of processes at runtime. Our proposed full in-band design is able to transition from one eager-threshold to another without impacting the communication throughput of the application. The proposed enhancements to limit the memory footprint by dynamically freeing unused internal communication buffer is able to significantly cut down on memory footprint of the MPI library without affecting the communication performance.
Experimental evaluations show that the proposed dynamic and adaptive design is able to deliver performance on-par with what exhaustive manual tuning provides while limiting the memory consumed to the absolute minimum necessary to deliver the desired benefits. For instance, with the Amber molecular dynamics application at 1,024 processes, the proposed design is able to perform on-par with the best manually tuned versions while reducing the memory footprint of the MPI library by 25%. With the 3D-Stencil benchmark at 8,192 processes, the proposed design is able to deliver much better overlap of computation and communication as well as improved overall time compared to the default version. To the best of our knowledge, this is the first point-to-point communication protocol design that is capable of dynamically adapting to the communication requirements of end applications.

Keywords: MPI, Point-to-point communication, Overlap of Communication and Computation

1 Introduction

Message Passing Interface (MPI) [16] is a very popular parallel programming model for developing high-performance scientific applications. The MPI Standard [18] offers

* This research is supported in part by National Science Foundation grants #CNS-1419123, #CNS-1513120, #ACI-1450440 and #CCF-1565414.

various point-to-point, collective, remote memory and synchronization operations. The point-to-point operation is a fundamental building block in MPI as one can orchestrate almost all higher level primitives that MPI provides using point-to-point operations. Point-to-point operations can be broadly classified as blocking and non-blocking depending on when the buffer that has been posted to the MPI library is available for reuse. While the semantics of blocking primitives (eg: MPI_Send, MPI_Recv) is geared towards delivering the best communication performance, non-blocking primitives (eg: MPI_Isend, MPI_Irecv) have the dual objective of delivering best performance while ensuring that applications can achieve overlap of computation and communication.

Over the last several years, point-to-point non-blocking communication has emerged as a popular method for application scientists to hide the communication overhead by overlapping communication and computation. While modern primitives like the Remote Memory Access (RMA) semantics proposed by the MPI-3 standard are specifically geared towards this, many popular application kernels and applications like conjugate gradient solvers [15], adaptive mesh refinement [12], multi-physics [26], molecular dynamics [7], and earthquake prediction codes [9] still take advantage of non-blocking point-to-point communication primitives to hide the communication overhead.

Although the concept of non-blocking point-to-point primitive seems simple and the benefits obvious, there are several caveats that need to be addressed before end applications can reap the benefits offered by this programming interface. One needs to carefully match the semantics expected by the programming interface to that offered by the underlying communication protocol in order to ensure optimal performance.

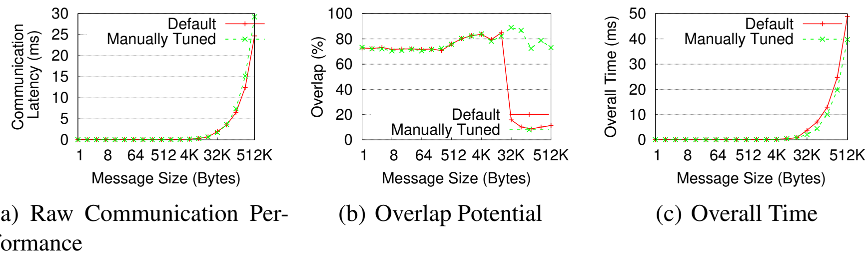
1.1 Motivation

There broadly exists two protocols for point-to-point data transfer in MPI — Eager and Rendezvous. Eager protocol sends data to the peer without waiting for an acknowledgment first and is thus used to transfer data of limited size to the receiver (typically small messages). The rendezvous protocol, on the other hand, uses control messages to ensure that the receiver has enough memory available to accommodate the incoming message. Thus it is typically used to transfer large messages. More details about these protocols are available in Section 2.1.

With modern multi-/many-core architectures and high-performance interconnects, there is always a "sweet-spot" where 1) the cost of exchanging the control information is not large enough to have an impact on the overall time of data transfer, and 2) the cost of memory copies to the internal buffer starts to be higher than the cost of exchanging control information. Most open source high-performance implementations of the MPI standard such as OpenMPI [8], MVAPICH2 [14], and MPICH [11] switch to the rendezvous protocol from eager protocol at this "sweet-spot" typically referred to as the "eager-threshold". In order to avoid the performance penalties seen with packetized data transfers, high-performance MPI libraries typically match the size of the internal communication buffers to be same as that of the eager-threshold. Thus, a secondary factor of consideration is the size of the internal communication buffers used to stage data in the eager protocol. Designers need to ensure that this is not so large that the memory footprint of the MPI library becomes too high.

As described above, while significant attention has been given to ensure that these protocols deliver best trade-off between performance and memory footprint, not much

attention has been paid to the overlap aspect of these protocols. We employ a simple case-study with a 3D-stencil benchmark (described in Section 2.3) to clearly motivate the need to account for overlap of computation and communication. Note that this communication pattern is representative of several large applications mentioned in Section 1. Figure 1(a) compares the raw communication performance of the 3D-stencil benchmark run with the default eager-threshold value of 17 KB against a version where we manually forced the eager-threshold and the internal communication staging buffer to be 1 MB. These numbers were taken with 8,192 processes (512 nodes) on the Stampede supercomputing system at TACC [25]. As expected, a smaller eager threshold forces use of rendezvous protocol which provides better raw communication time for large messages. However, as seen in Figure 1(b), this does not take into account the overlap potential of the different protocols. When there is computation that can be overlapped, use of the eager protocol is able to deliver better overall performance due to the higher overlap obtained. This is basically due to the fact that, with rendezvous transfer, the data transfer (which consumes the most time) does not start until an MPI_Wait or MPI_Waitall operation is called. However by switching to the eager protocol, the small loss of raw communication performance due to multiple memory copies are more than compensated by overlapping the most time-consuming data transfer part with computation, thereby reducing overall execution time 1(c).



(a) Raw Communication Performance (b) Overlap Potential (c) Overall Time

Fig. 1. Performance and overlap offered by eager and rendezvous protocols for 3D-Stencil benchmark at 8,192 processes on Stampede

While it is possible for application developers to manually tune this threshold to achieve better overlap, it involves significant effort and complexity. Modern high-performance MPI libraries have hundreds of tunable parameters each impacting a different aspect of communication. Thus, it is rather difficult for an application developer to effectively optimize a particular application using such manual tuning. Further, the communication pattern may change based on the size of the job and the input requiring constant re-tuning making such a solution impractical. To make matters worse, blindly increasing the eager-threshold can also have the negative consequence of increasing the overall memory footprint of the MPI library leaving less memory for the application to perform its science. Large internal communication buffers can also negatively affect the communication performance of small message operations due to poor cache locality on the sender and the receiver sides. Figure 2 shows the adverse effect of larger eager-threshold (and consequently larger communication buffers) on the message throughput. These issues lead us to the following broad challenge: **Can we design an adaptive and dynamic point-to-point communication mechanism for high-performance MPI libraries that can deliver the best communication performance, overlap of computation and communication, and memory footprint for all classes of applications?**

1.2 Contributions

In this paper, we take up this challenge and explore multiple point-to-point communication protocol designs to enable efficient overlap of computation and communication. We highlight the merits and deficiencies of each design and evaluate its performance with microbenchmarks and applications on modern HPC systems. Finally, we propose a dynamic and adaptive design for point-to-point communication that enables efficient overlap while ensuring basic communication performance and memory footprint is not adversely impacted. Our proposed full in-band design is able to transition from one eager-threshold to another without impacting the communication throughput of the application while taking care of all possible corner cases. The proposed enhancements to limit the memory footprint by dynamically freeing unused internal communication buffers is able to significantly cut down on memory footprint of the MPI library without affecting the communication performance. Our experimental results show that, our proposed dynamic and adaptive approach is able to deliver performance on par with what exhaustive manual tuning provides while cutting down on the overall memory footprint of the MPI library. For instance, with the Amber molecular dynamics application at 1,024 processes, the proposed design was able to perform on-par with the best manually tuned versions while reducing the memory footprint of the MPI library by 25%. With the 3D-Stencil benchmark at 8,192 processes, the proposed design is able to deliver much better overlap of computation and communication as well as improved overall time compared to the default version. To the best of our knowledge, this is the first point-to-point communication protocol design that is capable of dynamically adapting to the communication requirements of end applications. To summarize, the major contributions of this paper are:

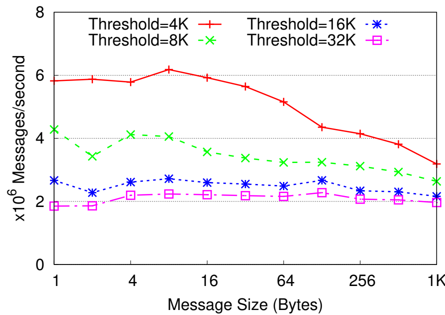


Fig. 2. Impact of changing Eager Threshold on performance of multi-pair message-rate benchmark with 32 processes on Stampede

- Study the interplay between communication pattern of applications and point-to-point communication protocols
- Propose, design, implement and study multiple dynamic and adaptive point-to-point communication protocols to deliver better overlap of computation and communication
- Explore alternate design approaches to overlap computation and communication and study its benefits and deficiencies
- Propose secondary designs to tackle the additional challenge of limiting memory footprint of the MPI library
- Demonstrate the benefits of the proposed scheme on performance with microbenchmarks and applications

Figure 3 compares the default, manually tuned, and the new designs along the metrics of performance, productivity, memory scalability, and overlap achieved. In all axes, the higher value is better. As we can see, the proposed dynamic and adaptive design performs the best when all metrics are considered. For instance, the proposed design is able to deliver overlap of computation and communication and overall application performance comparable to the best manually tuned version while providing a high degree of productivity similar to the default versions. It is also able to significantly cut down on the memory requirement of the MPI library when compared to best manually tuned version.

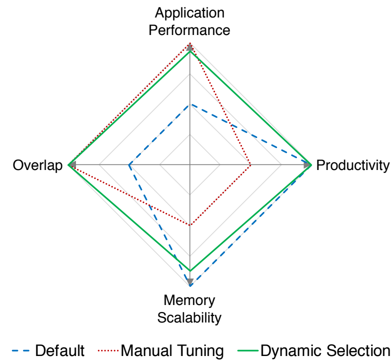


Fig. 3. Comparison of existing and proposed designs

2 Background

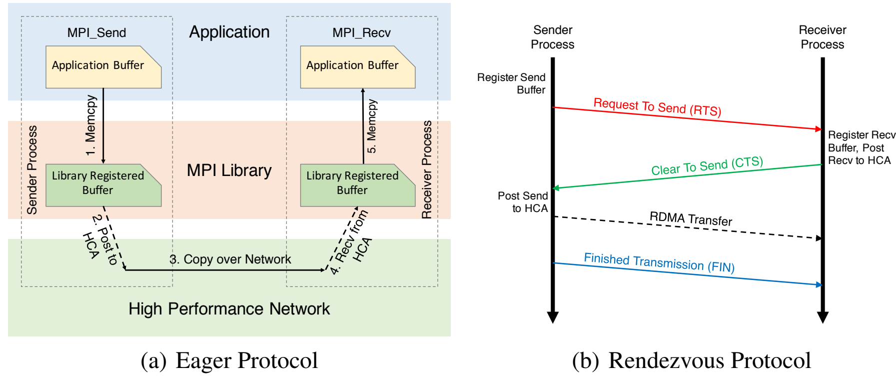
In this section, necessary background information for this paper is provided.

2.1 Protocols for High-Performance Point-to-point Communication in MPI

Figures 4(a) and 4(b) depict how the eager and rendezvous protocol respectively are typically implemented.The eager protocol consists of four steps — 1) copying the data from application buffer to buffers internal to the MPI library, 2) initiating the data transfer to the remote process, 3) detecting the reception of data in buffers internal to the MPI library, and 4) copying the data back to the application buffer. With most high-performance networks like InfiniBand, the network itself takes care of the actual data transfer. Thus, initiating the data transfer at the sender and detecting the reception of the data at the receiver are low overhead tasks. So, apart from the time to transfer data over the network, the main costs involved in an eager transfer are the memory copies at the sender / receiver. Note that steps #1 and #2 happen inside the send function call itself. With a rendezvous protocol on the other hand (Figure 4(b)), MPI designers take advantage of the RDMA feature that high-performance interconnects like InfiniBand offers and transfers data directly from the source application buffer to the target application buffer (with appropriate exchange of control information), thereby avoiding the extra large memory copies from the application buffer to internal communication buffers within the library.

2.2 Amber

Amber [7] is a molecular dynamics package including numerous programs that work in conjunction to perform end-to-end molecular dynamics simulation (from the creation of input files to the analysis of results).



(a) Eager Protocol (b) Rendezvous Protocol

Fig. 4. Point-to-point communication protocols in MPI

2.3 3D-Stencil Benchmark

The processes in the benchmark are mapped onto a 3D grid and each process talks to its neighbors in each dimension (6 neighbors). In every step, each process posts MPI_Irecv operations for all of the messages it expects and then posts all of the MPI_Isend calls. It waits for all of the transfers to complete with one MPI_Waitall call. At the end of each iteration, the benchmark executes a call to MPI_Allreduce to collect boundary information from all processes participating in the job. To calculate the overlap of communication and computation, we first measure the time to perform all the MPI_Irecvs and MPI_Isends immediately followed by a MPI_Waitall. The benchmark also computes the overall latency (the total time taken when computation is overlapped with communication), the communication and the computation time, and the overlap percentage. In addition, we are also time the initialization overhead and the wait time.

3 Common Challenges in Designing Dynamic and Adaptive Point-to-point Communication Protocols

Several applications tend to communicate with its peer processes using varying message sizes. Thus, one of the first design challenge is to enable the ability to have different eager-threshold values for different process pairs. To this end, we introduce two adaptive and dynamic designs — 1) partial in-band and 2) fully in-band that are capable of updating the eager-threshold for a pair of processes. However, there are some common design challenges that need to be addressed before such a change of eager-threshold can occur. We enumerate these challenges and our solutions to address these challenges in the following sub-sections.

3.1 Triggering Eager-Threshold Change

It is important for the MPI library to correctly identify when it needs to migrate to a higher eager-threshold in order to obtain better overlap of computation and communication. We find that two conditions need to hold for such a change of eager-threshold to have a positive impact on the performance of the end application:

1. The use of non-blocking send and/or recv operation (eg: MPI_Isend, MPI_Irecv) by the application

2. The time elapsed between posting non-blocking send / recv operation and polling for completion of the operation (through MPI_Wait, MPI_Waitall etc) should be a reasonable proportion (50% or more) of the total estimated time for data transfer

If either one of these conditions does not hold, then it is unlikely that the application will see any benefits because of the eager-threshold change. For instance, in an application using blocking send / recv operations there is no potential for overlap. On the other hand, if the time between posting the non-blocking operation and polling for completion is very fast (like in the case of a typical bandwidth benchmark), the potential for overlap is significantly reduced. Further, as shown in Section 1.1, incorrectly increasing the eager-threshold can negatively impact the performance of small messages due to the paging behavior at the receiver process. Finally, the initiating process must also ensure that there are enough resources available locally to allocate the resources necessary to perform an eager-threshold switch as described in Section 3.3.

3.2 Identifying the New Eager-Threshold

As the average size of messages being sent from process A to process B need not be the same as those being sent in the opposite direction, a "handshake" or "agreement" protocol is required to ensure that both processes settle on the same value for eager-threshold. This is very critical as different values for eager-threshold for a process pair can result in undefined communication behavior (like a hung data transfer). Further, it is possible that any one of the processes is unable to honor the eager-threshold change request (due to lack of resources or some internal errors). In this scenario, the process encountering the failure needs a mechanism to inform the peer process of its inability to proceed with the eager-threshold change.

We introduce two new packet types "NEW_CONN_HANDSHAKE_REQ" and "NEW_CONN_HANDSHAKE_REP" to address these issues. When a process decides to trigger an eager-threshold change (as identified in Section 3.1), it sends out a "NEW_CONN_HANDSHAKE_REQ" packet to its peer and marks the virtual communication channel that exists between the two processes to indicate that an eager-threshold change is in progress. This packet contains the new value of eager-threshold the initiating process wants the communication channel to be moved to. The new eager-threshold is calculated using the following equation:

$$Threshold_{new} = 2^{\left\lceil \log_2 \left(\frac{\sum sizeof(Rndv\ Msg + Pkt\ Header)}{Number\ of\ Rndv\ Msgs} \right) \right\rceil} + offset$$

The new threshold is chosen based on the average size of rendezvous messages being sent from the initiating process to the peer process. The goal here is to allow most of the large messages to go through the eager path while not increasing the eager-threshold to an unnecessarily large value. An "offset" of 1,024 bytes is added to ensure that messages falling right on the boundary of the new eager-threshold can also be accounted for with this change.

The remote process on receiving the "NEW_CONN_HANDSHAKE_REQ" packet, first checks if it can allocate the resources necessary to proceed with the eager-threshold change (as described in Section 3.3). If so, it proceeds to identify the local eager-threshold value using the formula described above. It then compares the local value with the value sent by the remote peer and uses the maximum of the two values as the

new eager-threshold for the communication channel. This value is communicated to the peer process using a "NEW_CONN_HANDSHAKE_REP" message and the communication channel is marked as "in-active" indicating that an eager-threshold change is in progress. If, for some reason, the process is unable to allocate the necessary resources or is unable to proceed with the eager-threshold change for any other reason, it responds back with an eager-threshold value of "-1".

The initiating process on receiving the "NEW_CONN_HANDSHAKE_REP" packet extracts the value of eager-threshold indicated by peer. If the value is "-1", the peer has indicated that it cannot proceed with the eager-threshold change and the process marks the communication channel as being incapable of processing an eager-threshold change so that no future eager-threshold changes are initiated by this process for the communication channel with the peer. If the value is non-negative, the process initiates either the partially in-band or the fully in-band mechanism (described in Sections 4.1 and 4.2 respectively) to establish a new connection with the larger eager-threshold.

3.3 Allocating Resources for Eager-Threshold Change

High-performance MPI libraries for InfiniBand typically use shared receive queues (SRQ) for improved scalability [22, 24]. With this technology, a process can have just one queue to receive data from any peer process. However, the buffers that are posted to receive data on the shared receive queue must be large enough to hold the data any sender may possibly send to it in a gratuitous fashion. In other words, the buffers posted to the SRQ must be equal to the new eager-threshold size the process wants to use identified by the "handshake" protocol described in Section 3.2. We introduce a pool based design where each process creates a set of internal communication buffers whose size is equal to the new eager-threshold agreed upon by the pair of processes. If such a pool already exists (from a previous dynamic eager-threshold change with another process), the pool and the associated SRQ is reused. Otherwise, a new pool and the SRQ are created and added to the list of available pools. We limit the number of such pools that a process can create, to a value that can be set at runtime by the user through an environment variable (default value: 20).

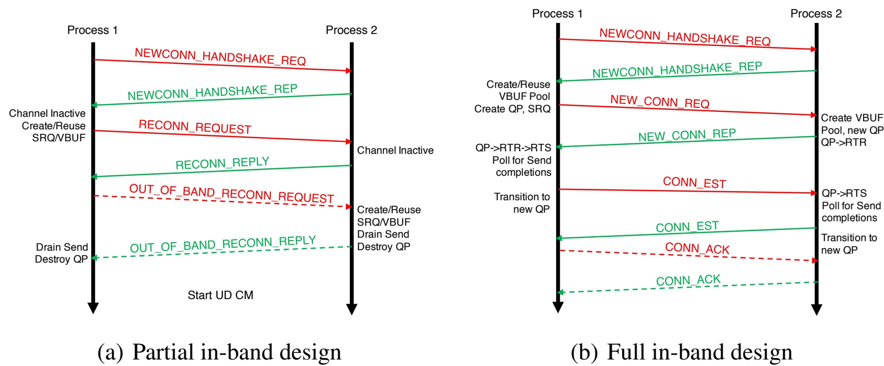
4 Dynamic and Adaptive Design for Point-to-point Communication Protocols

In this section we discuss the various alternative designs as well as their benefits and deficiencies. We use the open-source MVAPICH2 [14] library for the proposed designs and studies in this paper. However, the proposed designs are generic and can be incorporated into other MPI libraries.

4.1 Partial In-band Design

We first explore a partial in-band design to re-establish the connections between the process pairs with an increased eager-threshold. The various messages exchanged in this design are depicted in Figure 5(a). One of the major benefits of this approach is its ability to take advantage of the existing on-demand connection establishment design in MVAPICH2 [28, 27]. Once the new eager-threshold has been successfully identified as described in Section 3.2, the design marks the communication channel as inactive to

prevent the application from sending any further messages. The initiator process then allocates resources as described in Section 3.3 and sends a "RECONN_REQUEST" to the peer process. This will be the last message the initiator process transmits over the existing IB connection (also know as a queue pair - QP). The peer process on receiving the "RECONN_REQUEST" proceeds to mark the communication channel as inactive and replies back with a "RECONN_REPLY". This will be the last message that the peer process transmits over the existing QP. When the initiator process receives the "RECONN_REPLY", it initiates an "OUT_OF_BAND_RECONN_REQUEST" through the QP being used for the on-demand connection management design. On reception of the "OUT_OF_BAND_RECONN_REQUEST", the peer process allocates resources as described in Section 3.3, drains all the send completions from the existing QP and destroys it. After completing this successfully, it transmits a "OUT_OF_BAND_RECONN_REPLY" to the initiator. The initiator on receiving the request drains all the send completions from the existing QP, destroys it, and starts the on-demand connection establishment using the newly allocated resources. As we can see, one of the major cons of this approach is the duration of time for which the communication channel is marked as inactive. Such a long duration of inactivity has the potential to negatively impact the application throughput. Thus we discard this design from further consideration.



(a) Partial in-band design (b) Full in-band design

Fig. 5. Connection establishment in partial and full in-band designs

4.2 Full In-band Design

In the full in-band design, we completely avoid using the on-demand connection management design in MVAPICH2 and instead use the regular communication channel to exchange messages. The various messages exchanged in this design are depicted in Figure 5(b). The major benefit of this design when compared to the partial in-band design is that the communication channel is always active and thus does not affect the communication throughput of the application. Once the new eager-threshold has been successfully identified as described in Section 3.2, the initiator process allocates resources as described in Section 3.3, creates a QP for new eager-threshold and transmits a "NEW_CONN_REQ" message to the peer. The message contains the end-point information of the initiator process so that the peer can begin the process of IB connection management. The peer on receiving the "NEW_CONN_REQ" allocates resources as

described in Section 3.3, extracts the initiator end-point information from the message, transitions the newly created IB QP to ready-to-receive (RTR) state, and responds to the initiator with a "NEW_CONN_REP" message containing the local end-point information.

The initiator process on receiving the "NEW_CONN_REP" extracts the peer processes end-point information and uses that to transition the new local QP to RTR and ready-to-send (RTS) states. At this point, the new QP is capable of gratuitously sending messages to the peer at the increased eager-threshold size. Once this is complete, the initiator will ensure that all the previous send operations on the existing QP has completed and will send out a "CONN_EST" message to the peer with the local end-point information. This will be the last message that is sent on the old QP. All future messages are sent over the newly created QP. The peer on receiving the "CONN_EST" message will transition its newly created QP to the RTS state and is thus capable of gratuitously sending messages to the peer at the increased eager-threshold size. After this, the process will ensure that all the previous send operations on the existing QP has completed and will send out a "CONN_EST" message to the peer with the local end-point information. This will be the last message that is sent on the old QP. All future messages are sent over the newly created QP.

The initiator on receiving the "CONN_EST" message responds back with a "CONN_EST_ACK" message over the new QP to indicate that there are no more messages in flight on the old QP. The peer on receiving this message will destroy the old QP and send a "CONN_EST_ACK" message over the new QP to indicate that the initiator can destroy the old QP as well thus completing the transition to the new eager-threshold size. As this design proceeds without having to throttle application communication, it has the potential to deliver the best performance. Thus we use the "Full In-band" design for all experimental evaluations in the paper.

4.3 Avoiding Message Loss During Threshold Migration

It should be noted that in our design, eager thresholds are not bidirectional, i.e. a message from Process A to Process B could go over the eager protocol while a message of the same size from Process B to Process A could go through the rendezvous path. Furthermore, each process can independently decide to change the eager-threshold for a peer process based on its past communication with said peer. To prevent loss of messages, both peers must be able to identify when to switch to the new QP as well as handle in-flight messages during the handshake. The handshake protocol used in the "Full In-band" design achieves this by ensuring that a) the initiator process starts sending messages through the new QP only after the target process has acknowledged that it is ready to receive on the new QP, and b) the target process destroys the old QP only after getting a confirmation from the initiator that it has processes send completions for all messages sent through the old QP.

4.4 Mitigating Memory Footprint Requirements

A major concern with manually and exhaustively tuning the eager-threshold and the proposed dynamic and adaptive design is the potential increase in memory footprint of the MPI library due to the increased size of internal communication buffers. To address

this issue, we propose a design that dynamically identifies unused internal communication buffers and reclaims them. However, unless performed carefully, this operation can lead to a continuous cycle of allocation and freeing of internal communication buffers leading to poor performance.

Most high-performance MPI libraries dynamically allocate internal communication buffers as and when the library runs out of these buffers due to communication pressure from the application. Further, most applications have phases where the communication activity increases and decreases. Thus, if proposed design is too aggressive in freeing internal communication buffers, it could free a large number of buffers in the phase with low communication only to reallocate them when the communication pressure increases again. To avoid such a cycle, we add weights to the communication buffers and only free them if they have not been used for a specific period of time which is tunable. We have done extensive tuning of this value on multiple supercomputing systems and identified appropriate values for it to ensure that such cycles of allocation and freeing are avoided as much as possible. With such a design, we are able to significantly reduce the memory overhead caused by the dynamic and adaptive designs to less than 50% of what the manually tuned designs can offer.

4.5 Alternate Design Approaches

In this section, we explore possible alternate designs to avoid the requirement to exchange the control information in the rendezvous exchange which is the major cause for the lack of overlap. The ideal solution here would be to have a hardware component to which the rendezvous exchange can be offloaded. Unfortunately, such technology is still not available in the public domain. In this context, we create a new design that uses the "Receiver-Not-Ready" or RNR mechanism of InifniBand to avoid (1) the need to exchange control information and (2) the need for intermediate memory copies from the application buffer to internal MPI buffers and back. Figure 6 depicts how the communication proceeds in the RNR-based design.

In the RNR design, the sender and receiver create a special QP for each tag used for communication. Once the QP has been created, the sender directly registers the send buffer with the IB HCA and sends the data to the receiver's QP. At this point, the IB HCA takes over and continually checks with the target HCA as to whether it is ready to receive the data. The target HCA becomes ready to receive data when the receiver arrives and posts a corresponding receive operation to the special QP created earlier. Until this event occurs, the receiver HCA will respond back to the sender HCA's queries indicating that the receiver is not ready (RNR). Although there is a timeout after which the sender will stop trying to transmit the data to the receiver, we increase

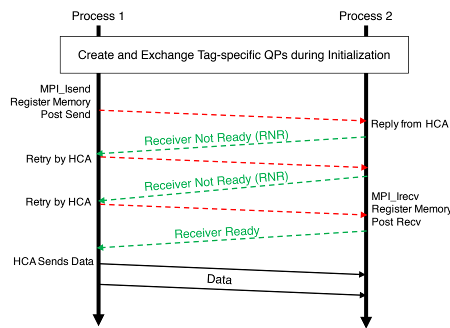


Fig. 6. Communication in RNR-based design

it so that the sender keeps retrying infinitely. Finally, when the receiver arrives, the target HCA indicates that the receiver is ready causing the sender to place the data in the buffer pointed to by the receive QP which happens to be the application level buffer in the RNR design. This design eliminates the need for the application to explicitly use control messages to stage the rendezvous transfer or to use intermediate memory copies to transfer the data using the eager protocol.

However, this design has several functionality and performance issues making it more constrained to use in real world applications. (1) Due to the lack of hardware-based tag matching, it cannot support wild cards such as
MPI_ANY_SOURCE and MPI_ANY_TAG which is very common in MPI as well as application communication. Further, as described above each communicating process pair needs to use a separate QP for communication creating a scalability bottleneck. On the performance side, the RNR design is unable to send a continuous stream of data as a typical eager or rendezvous protocol does. This is mainly due to the fact that it cannot start a subsequent transfer until the previous transfer is complete. Thus, it is hard to keep the communication pipeline resulting in sub-par communication throughput. Due to these performance and functionality constraints, we discount this design from further performance evaluations.

5 Experimental Results

In this section, we describe the experimental setup, provide the results of our experiments, and give an in-depth analysis of these results. All numbers reported here are averages of a minimum of five runs. There was little to no variance between the different runs. As described in Sections 4.1 and 4.5, we discard the Partial In-band design and RNR-based design from further performance evaluation due to their inherent design limitations. Thus, for the remainder of the performance evaluation section, "dynamic threshold" refers to the Full In-band design described in Section 4.2.

5.1 Experimental Setup

We used multiple high-performance computing systems to obtain the results for this paper:

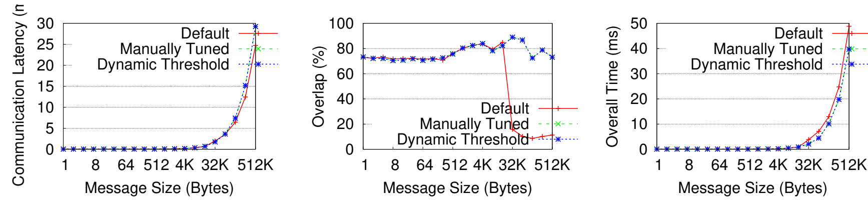
Gordon @ SDSC [21]: Each node contains two 8-core 2.6 GHz Intel EM64T Xeon E5 (Sandy Bridge) processors and 64 GB of DDR3-1333 memory. The operating system used is CentOS release 6.4(Final), with kernel version 2.6.32-431.29.2.el6. The network topology is a 4x4x4 3D torus with adjacent switches connected by three 4x QDR InfiniBand links (120 Gb/s). Compute nodes (16 per switch) and I/O nodes (1 per switch) are connected to the switches by 4x QDR (40 Gb/s).

Stampede @ TACC: Each node is equipped with an Intel SandyBridge series of processors using Xeon dual eight-core sockets, operating at 2.70 GHz with 32 GB RAM. Each host is equipped with MT4099 FDR ConnectX HCAs (54 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is CentOS release 6.7 (Final), with kernel version 2.6.32-431.17.1.el6, and OpenFabrics version 1.5.4.1. The network is a five-stage partial Fat-Tree with 5:4 oversubscription on the links.

5.2 Performance of 3D-Stencil Benchmark

In this section, we analyze the performance results of the 3D-stencil benchmark (described in Section 2.3) for different process counts on Stampede. In Figure 7(a), we

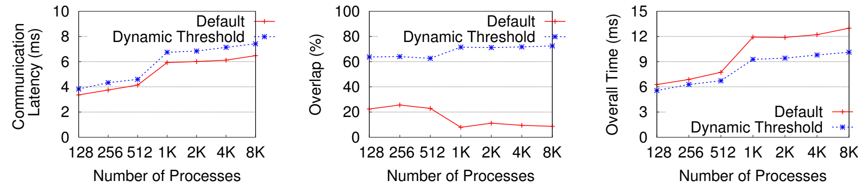
compare the raw communication performance of the default, manually tuned, and the proposed dynamic design at 8,192 processes. The default scheme provides the best pure communication time. However, when seen in conjunction with the overlapping computation, the default scheme performs worse than the manually tuned and proposed dynamic schemes as indicated by the overall time and overlap numbers in Figures 7(c) and 7(b) respectively. We also observe that, while the default scheme is able to offer better communication initialization time by avoiding the large memory copies associated with eager protocol indicated in Section 2.1, it loses out significantly in the time spent in MPI_Wait. This is because, with the rendezvous protocol being used in the default scheme, almost all of the time consuming data transfer happens in the MPI_Wait operation with little or no overlap of computation and communication actually happening. Due to lack of space, we are unable to include the figures describing the initialization and wait times in the paper.



(a) Raw Communication Performance (b) Overlap Potential (c) Overall Times

Fig. 7. Performance and overlap offered by various point-to-point communication protocols for 3D-Stencil benchmark at 8,192 processes on Stampede

Finally in Figure 8, we study the pure communication performance, overlap obtained, and overall communication time for the 128KB message size of the 3D-stencil benchmark for various system sizes on Stampede. As we can see, while the proposed dynamic threshold design takes a slight hit in raw communication performance (depicted in Figure 8(a)), is able to provide much better overlap (depicted in Figure 8(b)) and overall time (depicted in Figure 8(c)) when compared to the default design.

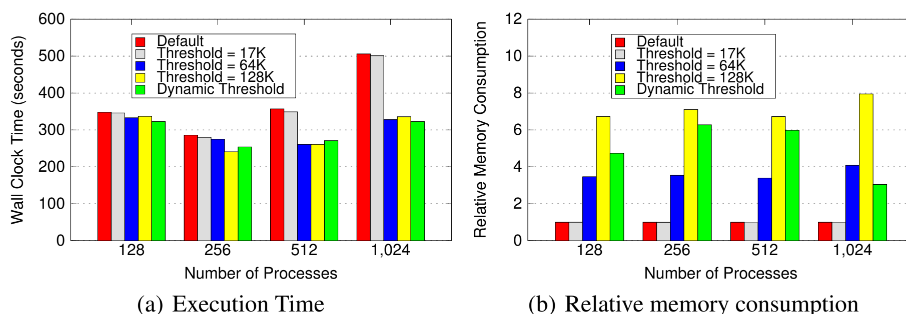


(a) Raw Communication Performance (b) Overlap Potential (c) Overall Times

Fig. 8. Comparison of performance and overlap offered by Default and Dynamic Threshold design for 128 KB messages in 3D-Stencil benchmark at different scales on Stampede

5.3 Performance of Amber

We perform an in-depth study and analysis of the performance of the Amber molecular dynamics code with the different point-to-point communication protocol designs in this section. Figure 9(a) shows the overall application execution time with the default design, various manual tuning options, and the proposed dynamic design. As we can see, for different system sizes, best performance is given by different manual tuning options. Such unpredictable behavior (as indicated in Section 1.1) makes this kind of manual tuning cumbersome, error prone, and impractical. The proposed dynamic design, on the other hand, is able to deliver performance on par with the best manual tuned design in a user transparent way making it a high-productivity and high-performance option for application developers.



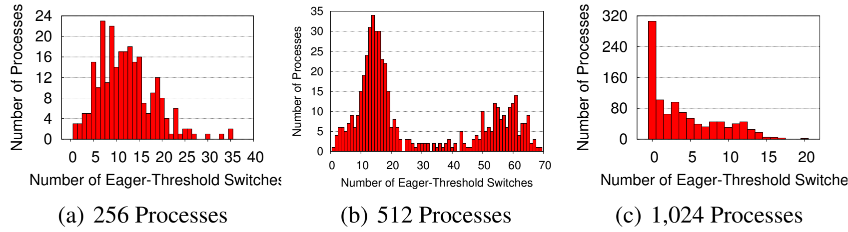
(a) Execution Time (b) Relative memory consumption

Fig. 9. Performance of Amber on Gordon with different values of Eager-Threshold

Figure 9(b) illustrates the memory used for the internal communication buffers used by the MPI library to stage the data transfers. The data is plotted relative to the amount of memory taken by the default design for internal communication buffers. As we can see, the default design gives best memory scalability. However, we also observe from Figure 9(a) that, it is unable to deliver the best performance due to its inability to effectively overlap communication and computation. The proposed dynamic design, on the other hand, is able to keep the memory footprint to the absolute minimum required by the design described in Section 4.4.

Figure 10 depicts the number times various processes performed switches of eager thresholds during the execution of the program for different system sizes. Although, due to space limitations, we only show details of the larger system sizes, note that the trends for smaller system sizes are similar. As we can see, at 256 and 512 processes the processes perform a lot of eager-threshold changes. This is reflected as increased memory usage for the dynamic design for the corresponding system sizes in Figure 9(b). However, at 1,024 processes, we observe that a large percentage of the processes perform no eager-threshold changes (indicated by the large value for zero in Figure 10(c)). This trend translates to relatively lower memory consumption for the dynamic design (when compared to 256 and 512 processes) as seen in Figure 9(b).

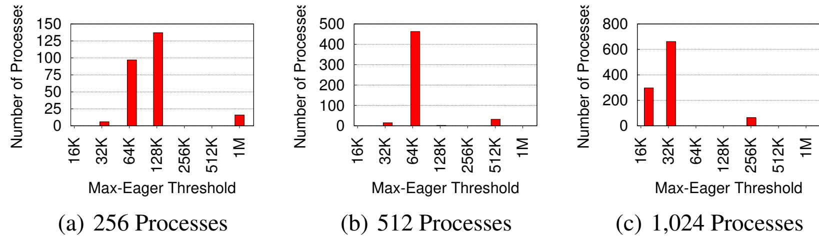
Figure 11 shows the maximum value of eager message size that various processes in the job end up with for different job sizes with Amber. We can see a clear trend with the maximum value of the message continually decreasing as the size of the system increases. While the median max eager-threshold size with 256 processes is 128KB,



(a) 256 Processes (b) 512 Processes (c) 1,024 Processes

Fig. 10. Number of eager-threshold switches performed by different number of processes with Amber on Gordon

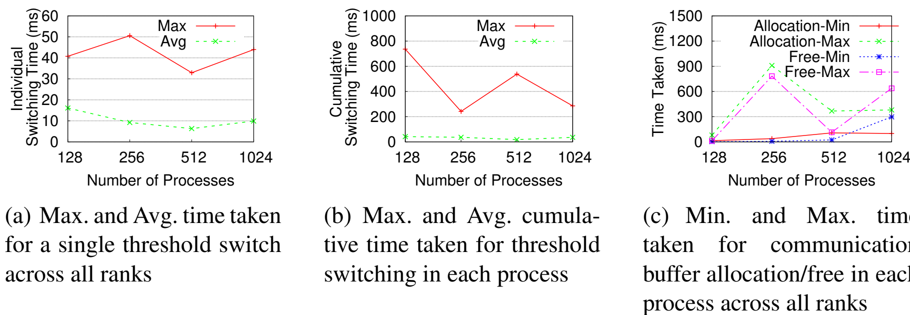
it reduces to 64KB at 512 processes and further reduces to 32KB at 1,024 processes. This indicates a distribution of computation load among the different processes - in other words, strong scaling. Another interesting trend to observe is the similarity in the number of processes whose max eager-threshold value is 17KB (the default value) at 1,024 processes as depicted in Figure 11(c) and the number of processes that perform no eager-threshold changes indicated by the large value for zero in Figure 10(c). These values corroborate each other indicating that the processes that do not have to perform eager-threshold switch are actually those whose desired max eager-threshold value happens to be equal to the default value the MPI library is configured with.



(a) 256 Processes (b) 512 Processes (c) 1,024 Processes

Fig. 11. Maximum value of eager-threshold used by different number of processes with Amber on Gordon

Figure 12(a) depicts the average and maximum time taken for one eager threshold switch across all processes for various system sizes. As we can see, the maximum overhead of establishing a new connection is very low (of the order of 40 ms) indicating the efficiency of the proposed design. Figure 12(b) on the other hand measures the maximum and average cumulative time spent by each process for eager-threshold switching during the lifetime of the job. As we can see, while the maximum value fluctuates a little, it is still very low (< 0.5 seconds). Note that this is for jobs that take 300 seconds to 350 seconds to execute on average (depicted in Figure 9(a)). Thus, we can clearly see that even the maximum cumulative time for eager-threshold switching only forms a negligible percentage (0.1%) of the overall execution time. Finally in Figure 12(c), we measure the time taken to perform dynamic de-allocation of internal communication buffers described in Section 4.4 for various system sizes with Amber on Gordon. As with the cumulative time for eager-threshold switching, we see that the maximum time taken to handle the overheads associated with dynamically allocating and freeing internal communication buffers are always less than one second. As we mentioned above, this only forms a negligible percentage (0.3%) of the overall execution time of the application.



(a) Max. and Avg. time taken for a single threshold switch across all ranks

(b) Max. and Avg. cumulative time taken for threshold switching in each process

(c) Min. and Max. time taken for communication buffer allocation/free in each process across all ranks

Fig. 12. Time taken for switching eager-threshold and communication buffer allocation/free with different number of processes with Amber on Gordon

6 Related Work

The use of extra threads, commonly known as asynchronous progress threads, to progress communication in the background has been a popular method to progress communication in the background while the application is computing in the foreground. While this allows progression of messages without the application having to enter the MPI library to progress the communication, it takes away a valuable computation core away from the application process which can adversely affect the overall performance of the application.

Brightwell et al [5] showed that eagerly sending large messages can improve latency for pre-posted receives. However, this scheme has to resend unexpected large messages in presence of application skew, which does not affect our design. In [3], Barrett et al proposed the use of triggered operations in the portals [20] interface to perform large message rendezvous operations. This method is similar to the RNR-based design proposed in Section 4.5 and also suffers from the same drawbacks as the RNR-based design in that it is not able to handle wild cards in the MPI library (e.g: MPI_ANY_SOURCE).

Researchers have also explored the use hardware-assisted tag matching techniques to offload rendezvous transfers to the hardware. While several high-performance network interconnect such as Intel Omni-Path, Myrinet [4], Quadrics [2], Bull BXI [10], and Mellanox [1] have proposed and are proposing solutions that expose such capabilities to high performance MPI libraries.

Automatic tuning for MPI libraries and applications has been explored by many researchers [17, 6, 19, 23]. However, such tools generally cannot perform more targeted tuning, such as changing eager-threshold for a small number of peers. Although the introduction of MPI-T [13] might enable such fine-grained tuning, good designs are still required inside the MPI library to minimize the overhead involved.

The emerging remote memory access (RMA) model semantically relieves the remote process from having to actively participate in communication. Thus it has the potential to completely overlap computation and communication. Although the use of the MPI3-RMA models is slowly catching up, the vast majority of scientific applications in use today still use the two-sided communication model. Further, collective operations, that are widely used across various scientific domains, still rely on two-sided point-to-point operations. Thus the proposed schemes are likely to remain very relevant in the future.

7 Conclusion and Future Work

In this paper, we proposed designs for point-to-point data transfer in MPI which accounts for overlap in addition to performance and memory footprint. The proposed designs dynamically adapt to the communication characteristic of each communicating pair of processes at runtime. Our proposed fully in-band design is able to transition from one eager-threshold to another without impacting the communication throughput of the application while taking care of all possible corner cases. The proposed enhancements to limit the memory footprint by dynamically freeing unused internal communication buffers is able to significantly cut down on memory footprint of the MPI library without affecting the communication performance. Our experimental evaluation showed that the proposed dynamic and adaptive design is able to deliver performance on-par with what exhaustive manual tuning provides while limiting the memory consumed to the absolute minimum necessary to deliver the desired benefits. For instance, with the Amber molecular dynamics application at 1,024 processes, the proposed design was able to perform on-par with the best manually tuned versions while reducing the memory footprint of the MPI library by 25%. With the 3D-Stencil benchmark at 8,192 processes, the proposed design is able to deliver much better overlap of computation and communication as well as improved overall time compared to the default version. To the best of our knowledge, this is the first point-to-point communication protocol design that is capable of dynamically adapting to the communication requirements of end applications.

As part of future work, we plan to study the benefits of the proposed design with multiple applications at scale.

References

1. Mellanox Technologies. http://www.mellanox.com.
2. Quadrics Supercomputers World Ltd. http://www.quadrics.com/.
3. B. W. Barrett, R Brightwell, K. S Hemmert, K. B. Wheeler, and K. D. Underwood. Using Triggered Operations to Offload Rendezvous Messages. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 120–129, Berlin, Heidelberg, 2011. Springer-Verlag.
4. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15:29–36, 1995.
5. R Brightwell and K Underwood. Evaluation of an Eager Protocol Optimization for MPI. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 327–334. Springer, 2003.
6. É Brunet, F Trahay, A Denis, and R Namyst. A Sampling-based Approach for Communication Libraries Auto-tuning. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 299–307. IEEE, 2011.
7. D.A. Case, T.A. Darden, T.E. Cheatham, C.L. Simmerling, J. Wang, R.E. Duke, R. Luo, R.C. Walker, W. Zhang, K.M. Merz, B.P. Roberts, B. Wang, S. Hayik, A. Roitberg, G. Seabra, I. Kolossváry, K.F. Wong, F. Paesani, J. Vanicek, X. Wu, S.R. Brozell, T. Steinbrecher, H. Gohlke, Q. Cai, X. Ye, J. Wang, M.-J. Hsieh, G. Cui, D.R. Roe, D.H. Mathews, M.G. Seetin, C. Sagui, V. Babin, T. Luchko, S. Gusarov, A. Kovalenko, and P.A. Kollman. Amber 2016, 2016. University of California, San Francisco.
8. Open MPI : Open Source High Performance Computing. http://www.open-mpi.org.

9. Y. Cui, R. Moore, K. Olsen, A. Chourasia, P. Maechling, B. Minster, S. Day, Y. Hu, J. Zhu, A. Majumdar, and T. Jordan. Toward petascale earthquake simulations. In *Acta Geotechnica (in press), Springer*, 2008.
10. S. Derradji, T. Palfer-Sollier, J. P. Panziera, A. Poudes, and F. W. Atos. The BXI Interconnect Architecture. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 18–25, Aug 2015.
11. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.
12. M. A Heroux, D. W Doerfler, P. S Crozier, J. M Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R Keiter, H. K Thornquist, and R. W Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.
13. T. Islam, K. Mohror, and M. Schulz. Exploring the Capabilities of the New MPI_T Interface. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 91. ACM, 2014.
14. J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004.
15. Yutong Lu, Chao Yang, and Yunfei Du. HPCG on Tianhe2.
16. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.
17. R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, et al. Autotune: A Plugin-driven Approach to the Automatic Tuning of Parallel Applications. In *International Workshop on Applied Parallel Computing*, pages 328–342. Springer, 2012.
18. MPI-3 Standard Document. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`.
19. A. Pimenta, E. Cesar, and A. Sikora. Methodology for MPI Applications Autotuning. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 145–146. ACM, 2013.
20. Portals Network Programming Interface. `http://www.cs.sandia.gov/Portals/`.
21. San Diego Supercomputing Center. Gordon Supercomputer. `http://www.sdsc.edu/services/hpc/hpc_systems.html#gordon`.
22. G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges. Infiniband scalability in open mpi. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 100–100, Washington, DC, USA, 2006. IEEE Computer Society.
23. A. Sikora, E. César, I. Comprés, and M. Gerndt. Autotuning of MPI Applications Using PTF. In *Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications*, pages 31–38. ACM, 2016.
24. S. Sur, L. Chai, H. Jin, and D. K. Panda. Shared receive queue based scalable mpi design for infiniband clusters. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 101–101, Washington, DC, USA, 2006. IEEE Computer Society.
25. Texas Advanced Computing Center. Stampede Supercomputer. http://www.tacc.utexas.edu/.
26. The MIMD Lattice Computation (MILC) Collaboration. `http://physics.indiana.edu/~sg/milc.html`.
27. J. Wu, J. Liu, P. Wyckoff, and D. Panda. Impact of on-demand connection management in mpi over via. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 152–159, 2002.
28. W. Yu, Q. Gao, and D. K. Panda. Adaptive connection management for scalable mpi over infiniband. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 10 pp.–, April 2006.