TupleQ: Fully-Asynchronous and Zero-Copy MPI over InfiniBand *

Matthew J. Koop, Jaidev K. Sridhar and Dhabaleswar K. Panda

Department of Computer Science and Engineering, The Ohio State University
{*koop, sridharj, panda*}*@cse.ohio-state.edu*

Abstract

The Message Passing Interface (MPI) is the defacto standard for parallel programming. As system scales increase, application writers often try to increase the overlap of communication and computation. Unfortunately, even on offloaded hardware such as InfiniBand, performance is not improved since the underlying protocols within MPI implementation require control messages that prevent overlap without expensive threads.

In this work we propose a fully-asynchronous and zero-copy design to allow full overlap of communication and computation. We design TupleQ with novel use of InfiniBand eXtended Reliable Connection (XRC) receive queues to allow zero-copy and asynchronous transfers for all message sizes. Our evaluation on 64 tasks reveals significant performance gains. By leveraging the network hardware we are able to provide fully-asynchronous progress. We show overlap of nearly 100% for all message sizes, compared to 0% for the traditional RPUT and RGET protocols. We also show a 27% improvement for NAS SP using our design over the existing designs.

1. Introduction

Commodity cluster computing has been growing at a furious speed over the last decade as the need for additional computational cycles continues to exceed availability. One of the most popular interconnects used on these clusters is InfiniBand [1], an open-standard with one-way latencies approaching 1μsec and bandwidth near 1.5GB/sec. Introduced in 2000, InfiniBand was developed as a general system I/O fabric, however, it has found particularly wide acceptance in High Performance Computing (HPC). The latest list of the Top500 [2] list shows over 28% of systems are now using InfiniBand as the compute node interconnect.

The Message Passing Interface (MPI) [3] is the dominant parallel programming model on these clusters. Given the role of the MPI library as the communication substrate for application communication, the library must take care to provide scalability both in performance as well as resource usage.

MPI provides synchronous and asynchronous data transfer methods, the most basic examples of these being the `MPI_Send` and `MPI_Isend` functions. An `MPI_Isend` does not need to complete when the function returns, instead it can wait for completion at some time in the future. This type of functionality allows for the MPI library to send the data in the background while computation performed. When using an interconnect that provides off-loading of data transfer, meaning the sending and receiving of data can be done without processor involvement it is even more important that this progress be able to be done in the background. InfiniBand is one such interconnect.

Due to this capability of data transfer offload in many high-performance interconnects, application writers have often tried to restructure their code to use asynchronous MPI calls. Unfortunately for application performance, many MPI libraries do not take advantage of this hardware capability.

In this paper we explore this problem in depth with InfiniBand and propose *TupleQ*, a novel approach to providing communication and computation overlap in InfiniBand without using threads and instead relying on the network architecture. Instead of using the traditional protocols with RDMA Put and Get operations, we propose using the network hardware to directly place the data into the receive buffers without any control messages or intermediate copies. We show that our solution is able to provide full overlap with minimal overhead and is able to achieve performance gains of 27% on the SP kernel in the NAS Parallel Benchmarks for 64 processes.

The rest of the paper is organized as follows: In Section 2, we give an overview of the related work to this topic. In Section 3 we provide the requisite background information on InfiniBand. Following in Section 4, we describe the current implementations of MPI over InfiniBand. In Section 5 we motivate our work by describing the problems with current MPI design for overlap of communication and computation. TupleQ, our fully-asynchronous design is presented in Section 6. We evaluate our prototype on both an overlap test and the NAS Parallel Benchmarks in Section 7. We

** This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; NSF grants #CNS-0403342, #CCF-0702675 and #CCF-0833169; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Mellanox, Intel, Cisco, QLogic and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Advanced Clustering, Appro, QLogic and Sun Microsystems.*

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

Authorized licensed use limited to: The Ohio State University. Downloaded on September 9, 2009 at 03:19 from IEEE Xplore. Restrictions apply.

conclude and discuss future work in Section 8.

2. Related Work

Achieving good overlap between computation and communication in MPI libraries has been a hot topic of research. Brightwell et al. [4] have demonstrated the application benefit from good overlap. Eicken et al. [5] have proposed hardware mechanisms to provide better overlap between computation and communication.

In terms of InfiniBand, Surs et al. [6] proposed a RDMA read based rendezvous protocol with a separate communication thread to achieve overlap. Kumar et al. [7] have proposed a lock free variant of the RDMA based design based on signals and a thread.

Our work is different as we do not use any threads to perform progress, as both of the previous designs have done. As a result we do not require any locking, signals, or other library interaction. Additionally, we do not use RDMA operations, which all other MPI libraries over InfiniBand use to implement large message transfer.

3. InfiniBand Architecture

InfiniBand is a processor and I/O interconnect based on open standards [1]. It was conceived as a high-speed, general-purpose I/O interconnect, and in recent years it has become a popular interconnect for high-performance computing to connect commodity machines in large clusters.

3.1. Communication Model

Communication in InfiniBand is accomplished using a queue based model. Sending and receiving end-points have to establish a Queue Pair (QP) which consists of Send Queue (SQ) and Receive Queue (RQ). Send and receive work requests (WR) are then placed onto these queues for processing by InfiniBand network stack. Completion of these operations is indicated by InfiniBand lower layers by placing completed requests in the Completion Queue (CQ). To receive a message on a QP, a receive buffer must be posted to that QP. Buffers are consumed in a FIFO ordering.

There are two types of communication semantics in InfiniBand: channel and memory semantics. Channel semantics are send and receive operations that are common in traditional interfaces, such as sockets, where both sides must be aware of communication. Memory semantics are one-sided operations where one host can access memory from a remote node without a posted receive; such operations are referred to as Remote Direct Memory Access (RDMA). Remote write and read are both supported in InfiniBand. Both communication semantics require communication memory to be registered with InfiniBand hardware and pinned in memory.

3.2. Transports

There are four transport modes defined by the InfiniBand specification, and one additional transport that is available in the new HCAs from Mellanox: Reliable Connection (RC), Reliable Datagram (RD), Unreliable Connection (UC), Unreliable Datagram (UD), and eXtended Reliable Connection. Of these, RC and UD are required to be supported by Host Channel Adapters (HCAs) in the InfiniBand specification. RD is not required and is not available with current hardware. All transports provide a checksum verification.

For our work we are most interested in the Reliable Connection and eXtended Reliable Connection transports:

- **Reliable Connection (RC)**: This is the most popular transport service for implementing MPI over InfiniBand. The RC transport is feature-rich and provides RDMA operations, atomic operations, and reliable channel semantics. As a connection-oriented transport, each connected process requires a dedicated QP.
- **eXtended Reliable Connection (XRC)**: XRC is a new transport introduced with the Mellanox ConnectX HCA to address scalability problems with the RC transport. It addresses the scalability problem by connecting processes to nodes rather than processes to processes. It also provides a unique addressing mechanism that will be discussed in Section 3.4 that will enable our work.

3.3. Shared Receive Queues

Introduced in the InfiniBand 1.2 specification, Shared Receive Queues (SRQs) were added to help address scalability issues with InfiniBand memory usage. As noted earlier, in order to receive a message on a QP, a receive buffer must be posted in the Receive Queue (RQ) of that QP. To achieve high-performance, MPI implementations pre-post buffers to the RQ to accommodate unexpected messages.

When using the RC transport of InfiniBand, one QP is required per communicating peer. To prepost receives on each QP, however, can have very high memory requirements for communication buffers. To give an example, consider a fully-connected MPI job of 1K processes. Each process in the job will require 1K - 1 QPs, each with n buffers of size s posted to it. Given a conservative setting of $n = 5$ and $s = 8KB$, over 40MB of memory per process would be required simply for communication buffers that may not be used. Given that current InfiniBand clusters now reach 60K processes, maximum memory usage would potentially be over 2GB per process in that configuration.

Recognizing that such buffers could be pooled, SRQ support was added so instead of connecting a QP to a dedicated RQ, buffers could be shared across QPs. In this method, a smaller pool can be allocated and then refilled as needed instead of pre-posting on each connection.

Authorized licensed use limited to: The Ohio State University. Downloaded on September 9, 2009 at 03:19 from IEEE Xplore. Restrictions apply.

Note that a QP can only be associated with one SRQ for RC. So any channel traffic on a QP will consume a receive buffer from the attached SRQ. If another SRQ is desired instead, a second QP must be created.

3.4. SRQ Addressing in XRC

With the addition of the XRC transport, a QP is no longer connected to a single SRQ. The sender can now specify the SRQ to which a message should be sent. Each message is sent with the SRQ number of the SRQ that it should be sent into. This is a key difference that will be the basis for our design.

4. Existing Designs of MPI over InfiniBand

MPI has been implemented over InfiniBand by many implementors and organizations, however, all of them generally follow the same set of protocols and design:

- *Eager Protocol:* In the eager protocol, the sender task sends the entire message to the receiver without any knowledge of the receiver state. In order to achieve this, the receiver must provide sufficient buffers to handle incoming unexpected messages. This protocol has minimal startup overhead and is used to implement low latency message passing for smaller messages.
- *Rendezvous Protocol:* The rendezvous protocol negotiates buffer availability at the receiver side before the message is sent. This protocol is used for transferring large messages, when the sender wishes to verify the receiver has the buffer space to hold the entire message. Using this protocol also easily facilitates zero-copy transfers when the underlying network transport allows for RDMA operations.

4.1. Eager Protocol

This mode is generally used for small messages and is designed for low-latency and overhead. In this mode the sender pushes the data over to the receiver without contacting the receiver. In this case the sender has no knowledge of the receiver state. This is important for two reasons:

- *Unknown Receive Address*: The address of the application receive buffer is not known. Since the address is not known the RDMA capability of InfiniBand cannot be used for a zero-copy transfer.
- *Unknown Availability*: The sender also has no idea if the receiver has posted a receive for this send operation yet.

As a result of this, the eager protocol for InfiniBand-based MPI libraries is done using copies on both the sending and receiving sides:

- *Sender Side*: Take a pre-allocated buffer (*send buffer* and place header information (tag, context, and local source rank) at the beginning. Next the application send buffer is copied into the send buffer.
- *Receiver Side*: The message is received into a *receive buffer* by the network hardware. On reception the receiver reads the header and checks the receive queue for a matching receive. If it is found then the data is copied into the corresponding application receive buffer. If it is not available the message is buffered and will be copied when the receive is posted.

This process is shown in Figure 1(a). This shows there are two copies as well as the network transfer.

4.2. Rendezvous Protocol

The rendezvous protocol is generally implemented with one of the RDMA operations of InfiniBand, RDMA Write or RDMA Read.

In each case the sender sends a "Request to Send (RTS)" message to the receiver. Upon receipt of the RTS message the queue of posted receives is searched. Then depending on the protocol different steps are taken:

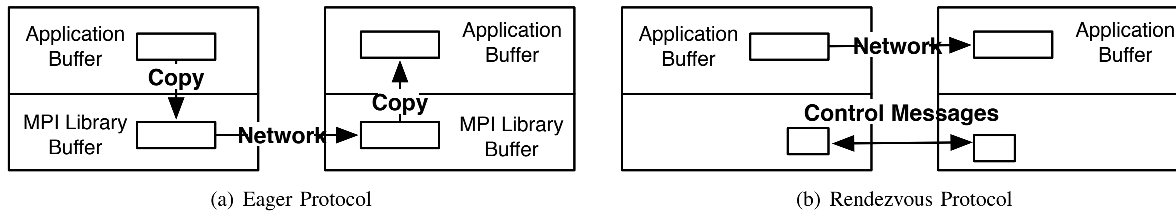
- *RDMA Write (RPUT)*: If the receive for this send has already been posted the address of the application receive buffer is sent back to the sending process. If the receive has not been posted the address is sent whenever the receive is posted. Upon receipt of the receiver buffer, the sender directly RDMA writes the data from the sender application buffer to the receiver application buffer on the remote node. A finish message (FIN) is also sent to the receiver to notify the completion of the send operation.
- *RDMA Read (RGET)*: If the receive has already been posted, the receiver directly reads the data from the senders application buffer and places it into the receive buffer with an RDMA Read operation. If the receive has not been posted then the operation will occur after the receive has been posted. After completion of the RDMA Read the receiver sends a FIN to the sender to indicate that the send can be marked complete.

In both of these cases the application buffer undergoes no intermediate copies, "zero-copy transfer," but does require at least two control messages to be sent. This data movement path is shown in Figure 1(a). As we will show in the next section, these control messages often prevent communication and computation overlap.

5. Semantic Differences Between MPI and InfiniBand

In this section we discuss the matching semantic differences MPI and InfiniBand matching.

Authorized licensed use limited to: The Ohio State University. Downloaded on September 9, 2009 at 03:19 from IEEE Xplore. Restrictions apply.



(a) Eager Protocol

(b) Rendezvous Protocol

Figure 1. Data Movement of Protocols

5.1. MPI Matching

Matching in MPI is based on the communicator, tag and the rank that sent the message:

- *Communicator:* A communication context, such that any message in one context does not interact with each other. This allows different "groups" to be created that do not interfere.
- *Rank:* Within each communicator, each process is ordered and given an integer rank from 0 to n-1. Using this rank information, messages can be send to the appropriate process.
- *Tag:* An MPI tag is a message identifier to allow different ordering.

Each message is matched on these characteristics. Generally, this matching is implemented by giving a unique integer ID to a communicator Then each message is matched on the tuple of the communicator ID, rank, and tag. Ordering is based in order on this matching information and not just the order that messages arrive.

5.2. InfiniBand Matching

As noted in Section 3, InfiniBand has multiple transports. The RC and XRC transports are those most used in implementing MPI. The semantics for RC and XRC are such that messages on a single QP are ordered.

Receive buffers in InfiniBand are consumed in a FIFO manner. The only matching done is based on the QP that it is sent to and no other identifier. When using XRC messages can be routed by SRQ number, but there is no other form of matching. Buffers are otherwise always consumed in the order they are posted.

Thus, there is a significant semantic gap between InfiniBand message matching and MPI message matching. Due to this, message matching for MPI over InfiniBand has typically been entirely in software.

6. Motivation

As noted earlier, MPI allows the application writer to use non-blocking communication with `MPI_Isend` to overlap communication and computation. Unfortunately, the method of implementing large message transfer, is often done with control messages and an RDMA operation.

Since MPI libraries generally poll for incoming messages, an incoming control message cannot be discovered unless it is in the progress engine (within an MPI call). If the application is trying to achieve overlap of communication with computation, the application will by definition not be in the MPI library. As a result, the control messages can be delayed, leading to no overlap in many cases.

Figure 2 shows the overlap that can be achieved with the current designs. The RDMA Write-based design has 3 control messages and leads to poor sender and receiver side overlap. If the sends immediately goes into a computation loop after sending the message there will be no overlap. Similarly, the receiver has no overlap as well. For RDMA Read, the sender has good overlap, however the receiver will have very poor overlap if the receiver has gone into a computational loop. Neither of these existing designs provides good performance.

Others have proposed using threads, however, this increases the overhead since signaled completion in InfiniBand is quite a bit slower than that of polling. This also can decrease performance due to locking required. Signaling has also been suggested to avoid locks, however, many calls within the MPI progress engine are not signal-safe.

7. Proposed Design

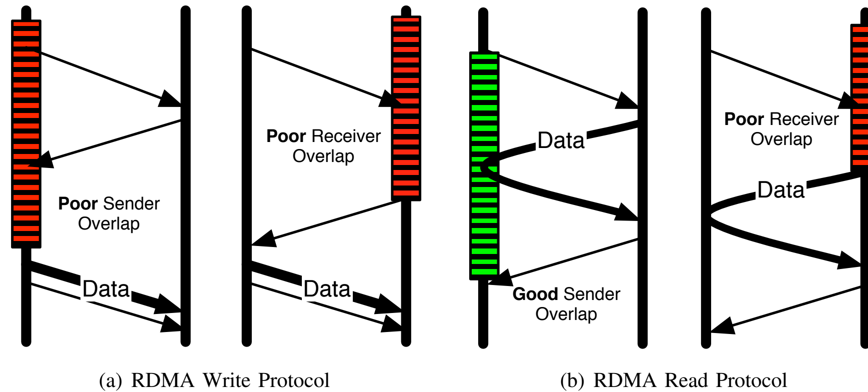
In this section we describe our design that allows full overlap of communication and computation for all message sizes. We first describe the mechanism that we use to provide zero-copy transfers without needing to exchange control messages, then we discuss the option to use sender-side copies, creation of receive queues, and discuss handling of the MPI wildcards.

7.1. Providing Full Overlap and Zero-Copy

As noted in Section 3, InfiniBand offers two sets of semantics to transfer data, channel and memory. Traditionally memory semantics (RDMA) has been seen as the only efficient method by which to transfer large messages without copies for MPI. Our design shows that this is not the case.

In channel semantics the receive buffers are consumed in order from a receive queue. In contrast, MPI semantics

Authorized licensed use limited to: The Ohio State University. Downloaded on September 9, 2009 at 03:19 from IEEE Xplore. Restrictions apply.



(a) RDMA Write Protocol

(b) RDMA Read Protocol

Figure 2. Overlap of Existing RDMA Protocols

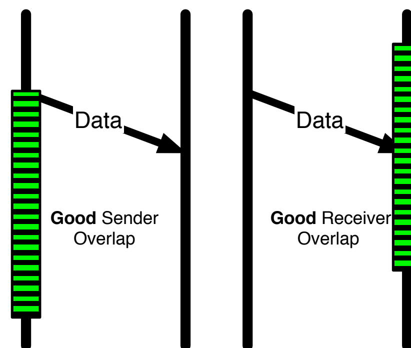


Figure 3. TupleQ Overlap

require messages to be matched in order, not necessarily in the same order that they are posted. This semantic gap has been seen as a need for exchanging control messages.

After studying various applications and patterns we have found that relatively few tags and communicators are used in MPI libraries. As such, we propose to have *separate queues for each matching tuple*. This design we term *TupleQ*. The matching tuple contains the tag, communicator ID and rank. If each matching tuple has a separate queue, MPI semantics can match those of InfiniBand as long as only messages with that tuple are sent to that receive queue. In this way we are able to provide full overlap as seen in Figure 3.

In addition to being zero-copy without control messages, this is fully asynchronous and receives are now "matched" by the hardware. If a receive buffer is not posted to a queue and a send operation is sent to that queue, the sender HCA will block until the receive buffer is posted. In this case is HCA is handling all operations and neither the sender or receiver CPU has any involvement with these retries.

7.2. Creating Receive Queues

As noted in Section 3.3, with the RC transport of InfiniBand a Queue Pair (QP) can only be connected to a single receive queue. As a result, a new connection would be required for each new queue. This approach would not be scalable. In contrast, the XRC transport allows addressing of receive queues, so multiple QPs are not required. Instead we just create a new SRQ for each matching tuple. When sending a message, the sender simply sends to the previously agreed upon SRQ number of the matching tuple.

Given the very large space of possible matching tuples, the receive queues are created "on demand." Only when the sender needs to send a message of a given tuple is an out-of-band message sent to the receiver to setup a queue for that tuple. The receiver responds with the SRQ number for the tuple. This value is then cached on the sender, so all future sends of that tuple will use that queue. Similarly, the receiver will post all application receive buffers directly to the receive queue for that tuple. If the tuple has not been created when the receive queue will be created for that tuple.

7.3. Sender-Side Copy

In MPI buffer availability determines when a blocking send call (`MPI_Send`) call can complete. So as long as the send buffer is available to be overwritten, the call can complete. As a result, as described earlier, since many MPI implementations already copy the send buffer to another buffer the send can be marked complete directly after the

Authorized licensed use limited to: The Ohio State University. Downloaded on September 9, 2009 at 03:19 from IEEE Xplore. Restrictions apply.

buffer has been copied. This option is also possible in TupleQ as well since there may be benefits to allowing the sender to go on even if the receiver has not posted the receive. We evaluate this option in Section 8.

7.4. MPI Wildcards

MPI specifies two wildcards that may be used when posting a receive. `MPI_ANY_SOURCE` and `MPI_ANY_TAG`. The first, `MPI_ANY_SOURCE`, means that the source of the message is unknown and can match any source. The second, `MPI_ANY_TAG`, allows a receive operation to match any tag. Both can be used together to match any incoming send operation.

These wildcards are a challenge for a number of reasons:

- There is no "wildcard" receive that can be placed in many receive queues at a time and then removed from all when it is consumed.
- A sender will try to always send to the designated tuple queue and the hardware will not complete the message until a receive is posted. The receiver will not know what buffer to post the receive to since it has been given a wildcard.

To address this issue we introduce a wildcard fallback mechanism. When a wildcard receive is posted all connections to the receiver are shutdown. This prevents any messages from being retried by the HCA and all will end up connecting back to the receiver with a traditional mechanism described in Section 4. After the wildcard has been matched the receiver will notify the senders and the tuple queues can be used again.

This fallback can be quite expensive if it occurs too often, so TupleQ will fall back to a traditional implementation model if it occurs too frequently. Alternatively, it can be disabled for applications that are known to contain wildcards. Further, the MPI-3 standard that is under discussion may contain support for "asserts", in which the application could inform the MPI library that it will or will not be using wildcards [8].

To provide full functionality we would like to see a hardware matching mechanism that allows a single receive descriptor to be posted to multiple queues simultaneously and be removed from all queues once it is consumed.

7.5. InfiniBand Matching Details

As noted earlier, there is no explicit "matching" in InfiniBand. When a message arrives, it is received into the QP or SRQ that it is addressed. This is done with strictly first-in-first-out semantics. Figure 4 shows the difference between a traditional design and the new TupleQ design. In the first design a message is received into a temporary buffer, matched, and then copied into the end user buffer. This design requires a copy. The TupleQ design by contrast allows a message to be directly placed into the receive buffer. Only the small receive descriptor that points to the end buffer is in the MPI library.

8. Evaluation

In this section we evaluate the design described in the previous section. We first measure the overlap potential of our design as well as the overhead incurred as compared to the traditional implementation. We also evaluate our design with the SP kernel of the NAS Parallel Benchmarks.

8.1. Experimental Platform

Our experimental platform is a 128-core InfiniBand Linux cluster. Each of the 8 compute nodes has 4 sockets each with a Quad-Core AMD Opteron 8350 2GHz Processor with 512 KB L2 cache and 2 MB L3 cache per core. Each node has a Mellanox MT25418 dual-port ConnectX HCA. InfiniBand software support is provided through the OpenFabrics/Gen2 stack [9], OFED 1.3 release.

8.2. Experimental Combinations

We evaluate four different combinations to observe their effect on overlap and performance. For the MPI library, we use MVAPICH [10], a popular open-source MPI implementation over InfiniBand. It is based on MPICH [11] and MVICH [12] and is used by over 840 organizations worldwide. We implement our TupleQ design into MVAPICH as well.

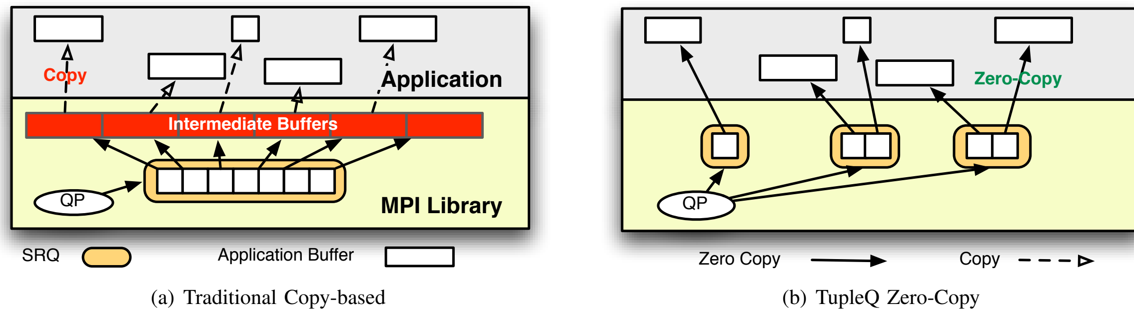
The combinations we evaluate are:

- *TupleQ*: This is the design described in Section 7 with no data copies.
- *TupleQ-copy*: This is the design described in Section 7 with sender-side copies for messages under 8KB.
- *RPUT*: This is the RDMA Write based design from MVAPICH.
- *RGET*: This is the RDMA Get based design from MVAPICH.

8.3. Overlap

We use the Sandia Benchmark [13] to evaluate the overlap performance of our design. Figure 5(a) shows the Application Availability that the protocol allows. Due to the control messages, the RGET and RPUT designs have poor overlap. 8KB is the threshold where the rendezvous protocol is used for RPUT and RGET and thus the steep drop in overlap. Since the TupleQ design is fully asynchronous nearly full overlap is obtained for all message sizes, including small ones. Figure 5(b) shows the overhead incurred with each

Authorized licensed use limited to: The Ohio State University. Downloaded on September 9, 2009 at 03:19 from IEEE Xplore. Restrictions apply.



(a) Traditional Copy-based

(b) TupleQ Zero-Copy

Figure 4. Transfer Mechanism Comparison

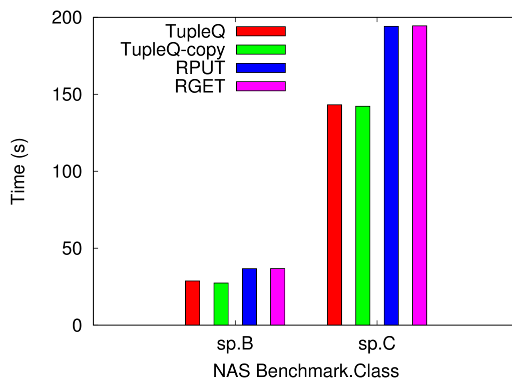


Figure 6. NAS SP Benchmark

send operation – what is not overlapped. Since the TupleQ design does full overlap there is minimal overhead, whereas the RGET and RPUT designs have high overhead since none of the message transfer can be overlapped.

8.4. NAS Parallel Benchmarks (NPB) - SP

The NAS Parallel Benchmarks [14] (NPB) are a set of programs that are designed to be typical of several MPI applications, and thus, help in evaluating the performance of parallel machines.

Of these kernels, the SP kernel attempts to provide overlap of communication and computation overlap with `MPI_Isend` operations. As such, this is the kernel that we evaluate for the performance of our new design.

We evaluate this benchmark using 64 processes. Further, we disable shared memory for all combinations since the shared memory implementation is not overlapping (future designs have been proposed that do allow overlap) and will negate some of the benefit seen.

As shown in Figure 6, the new TupleQ design does very well for both Class B and Class C. For Class B, the TupleQ design gives 27.38 seconds and the copy design gives 28.74 seconds. The RGET and RPUT designs do similar with 36.78s and 36.69, respectively. The same pattern is shown for Class C. The TupleQ design is able to give 27% higher performance. Again the RPUT and RGET designs perform similarly. There is very little difference between the TupleQ and TupleQ-copy modes. There are smaller messages being transfered, but the majority of the transfers are large with these datasets. Given that the TupleQ design requires no buffering it is the better option.

9. Conclusion and Future Work

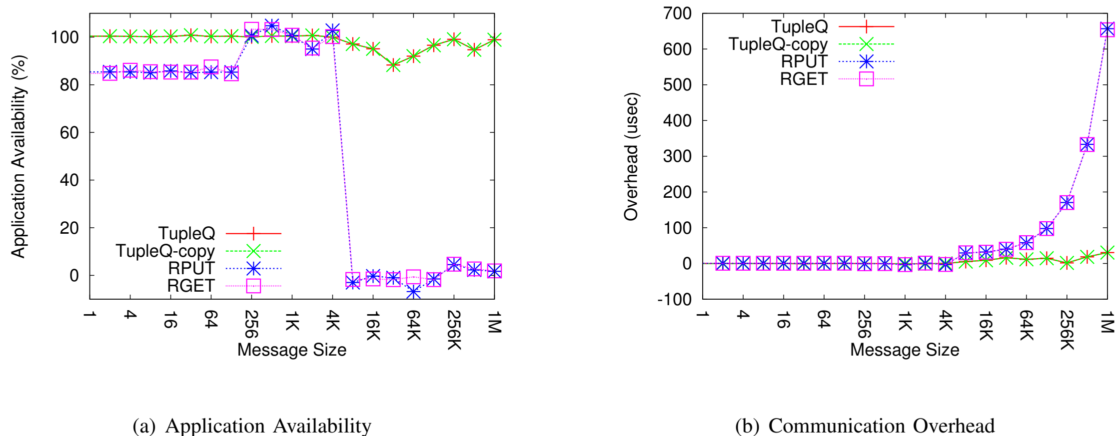
MPI is the current standard for parallel programming. As system scales increase, application writers try to increase the overlap of communication and computation overlap. Unfortunately, even on offloaded hardware, like InfiniBand, performance is not improved since the underlying protocols within MPI require control messages that prevent overlap without expensive threads.

In this work we propose a fully-asynchronous design to allow full overlap of communication and computation. We design *TupleQ* with novel use of XRC receive queues to allow zero-copy transfers for all message sizes

Our evaluation on 64 tasks reveals significant performance gains. By leveraging the network hardware we are able to provide fully-asynchronous progress. We show overlap of nearly 100% for all message sizes, compared to 0% for the traditional RPUT and RGET protocols. We also show a 27% improvement for NAS SP using our design over the existing designs.

In the future we wish to explore a wider range of applications at increased scale to see the effect of the zero-copy and asynchronous protocols. We also wish to further study applications to see how they can be modified to take care of this additional overlap capabilities. We would also like to look into the hardware modifications necessary to provide better support for the MPI wildcards.

Authorized licensed use limited to: The Ohio State University. Downloaded on September 9, 2009 at 03:19 from IEEE Xplore. Restrictions apply.



(a) Application Availability

(b) Communication Overhead

Figure 5. Sandia Overlap Benchmark

Acknowledgements

We would like to thank Pasha Shamis and Gil Bloch from Mellanox Technologies for various technical discussions on this topic with us.

References

[1] InfiniBand Trade Association, “InfiniBand Architecture Specification,” http://www.infinibandta.com.

[2] “TOP 500 Supercomputer Sites,” http://www.top500.org.

[3] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, Mar 1994.

[4] R. Brightwell and K. D. Underwood, “An Analysis of the Impact of MPI Overlap and Independent Progress,” in *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*. New York, NY, USA: ACM, 2004, pp. 298–305.

[5] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active Messages: A Mechanism for Integrated Communication and Computation,” in *ISCA '92: Proceedings of the 19th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 1992, pp. 256–266.

[6] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, “RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: design alternatives and benefits,” in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 32–39.

[7] R. Kumar, A. Mamidala, M. Koop, G. Santhanaraman, and D. K. Panda, “Lock-free Asynchronous Rendezvous Design for MPI Point-to-point communication,” in *EuroPVM/MPI 2008*, September 2008.

[8] MPI Forum, “MPI-3 Discussions,” http://www.mpi-forum.org/.

[9] OpenFabrics Alliance, “OpenFabrics,” http://www.openfabrics.org/.

[10] Network-Based Computing Laboratory, “MVAPICH: MPI over InfiniBand and iWARP,” http://mvapich.cse.ohio-state.edu.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, “A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard,” Argonne National Laboratory and Mississippi State University, Tech. Rep.

[12] Lawrence Berkeley National Laboratory, “MVICH: MPI for Virtual Interface Architecture,” http://www.nersc.gov/research/FTG/mvich/ index.html, August 2001.

[13] Sandia National Laboratories, “Sandia MPI Micro-Benchmark Suite,” http://www.cs.sandia.gov/smb/.

[14] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, “The NAS Parallel Benchmarks,” vol. 5, no. 3, Fall 1991, pp. 63–73.

Authorized licensed use limited to: The Ohio State University. Downloaded on September 9, 2009 at 03:19 from IEEE Xplore. Restrictions apply.