Designing MPI Library with Dynamic Connected Transport (DCT) of InfiniBand: Early Experiences*

Hari Subramoni, Khaled Hamidouche, Akshey Venkatesh,
Sourav Chakraborty, and Dhabaleswar K. Panda

Department of Computer Science and Engineering, The Ohio State University,
Columbus, OH, USA
{subramoni.1,hamidouche.2,akshayvenkatesh.1,
chakraborty.52}@osu.edu, {panda}@cse.ohio-state.edu

Abstract. The Dynamic Connected (DC) InfiniBand transport protocol has recently been introduced by Mellanox to address several shortcomings of the older Reliable Connection (RC), eXtended Reliable Connection (XRC), and Unreliable Datagram (UD) transport protocols. DC aims to support all of the features provided by RC — such as RDMA, atomics, and hardware reliability — while allowing processes to communicate with any remote process with just one DC queue pair (QP), like UD. In this paper we present the salient features of the new DC protocol including its connection and communication models. We design new verbs-level collective benchmarks to study the behavior of the new DC transport and understand the performance / memory trade-offs it presents. We then use this knowledge to propose multiple designs for MPI over DC. We evaluate an implementation of our design in the MVAPICH2 MPI library using standard MPI benchmarks and applications. To the best of our knowledge, this is the first such design of an MPI library over the new DC transport. Our experimental results at the microbenchmark level show that the DC-based design in MVAPICH2 is able to deliver 42% and 43% improvement in latency for large message All-to-one exchanges over XRC and RC respectively. DC-based designs are also able to give 20% and 8% improvement for small message One-to-all exchanges over RC and XRC respectively. For the All-to-all communication pattern, DC is able to deliver performance comparable to RC/XRC while outperforming in memory consumption. At the application level, for NAMD on 620 processes, the DC-based designs in MVAPICH2 outperform designs based on RC, XRC, and UD by 22%, 10%, and 13% respectively in execution time. With DL-POLY, DC outperforms RC and XRC by 75% and 30%, respectively, in total completion time while delivering performance similar to UD.

Keywords: Dynamic Connected Transport, InfiniBand, High Performance Computing, Network technology.

1 Introduction

Two key drivers fueling the growth of supercomputers over the last decade are the current trends in multi-/many-core architectures and the availability of commodity,

* This research is supported in part by National Science Foundation grants #OCI-0926691, #OCI-1148371, #CCF-1213084, and #CNS-1347189.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2014, LNCS 8488, pp. 278–295, 2014.
© Springer International Publishing Switzerland 2014

RDMA-enabled, and high-performance interconnects such as InfiniBand [1]. As supercomputing systems head to exascale levels, the core densities of current generation multi-/many-core architectures are expected to increase manyfold. Consequently, the amount of memory available per core on such systems is expected to be low compared to current generation systems. Thus, the focus is being increasingly given to reducing the memory footprint of applications and communication middlewares that targets these systems.

MPI [2] is a popular programming model for parallel scientific applications running on current generation multi-petaflop supercomputers. As such, the MPI library design is crucial in supporting high-performance and scalable communication for applications on these large-scale clusters. Over the last decade, InfiniBand has become an increasingly popular interconnect for deploying modern supercomputing systems. InfiniBand offers several communication protocols like Reliable Connection (RC), eXtended Reliable Connection (XRC), and Unreliable Datagram (UD) which all have different performance and memory characteristics. Several implementations of MPI over InfiniBand like MVAPICH2 [3], Open MPI [4], and Intel MPI [5] are already delivering good performance for applications on these systems using the RC transport. While RC delivers the best performance at a small scale, earlier work has shown that the RC transport requires several Kilobytes of memory per connected peer, leading to significant memory usage and performance degradation at large-scale runs [6].

To alleviate this, MPI libraries like MVAPICH2 take advantage of protocols like XRC and UD to reduce the memory footprint while delivering comparable or better performance [7, 6]. UD-based solutions enable the MPI library to keep the memory required for creating communication queue pairs (QPs) constant. However, implementing MPI over UD requires software-based segmentation, ordering, and re-transmission within the MPI library. Furthermore, UD transport does not support several novel features like hardware-based atomic operations. XRC-based solutions, on the other hand, avoid these software overheads and also have hardware support for atomic operations. However, the QP memory footprint scales linearly with the number compute nodes. With exascale systems expected to have O(100,000) to O(1,000,000) nodes [8], this presents a huge overhead in terms of memory.

Recently, Mellanox has introduced the Dynamic Connected (DC) transport protocol. The DC transport attempts to give the same feature set of RC while providing UD-like flexibility in terms of communicating with all peers using one QP. Given this capability, the connection memory required for DC can potentially reduce by a factor equal to the *number cores per node * number of nodes* compared to RC. Similarly, the connection memory required can reduce by a factor of *number of nodes* compared to XRC. These represent potentially large savings as system size and core densities continue to increase. In this paper, we present the salient features of the new DC protocol. We provide a comparison of the connection model of DC with other IB transport protocols. We compare and contrast the connection model of DC with other IB transport protocols. We design a micro-benchmark suite at the verbs [9] level to study the performance, and memory tradeoffs of the new DC transport. We then use this knowledge to propose a dynamically adaptive scheme which uses multiple designs for implementing MPI over DC. We implement our designs in the MVAPICH2 MPI library and evaluate

it using standard MPI benchmarks as well as applications. Additionally, memory usage is also measured. To the best of our knowledge this is the first such study of the DC protocol and design of MPI over DC. To summarize, this paper makes the following contributions:

- Understanding the behavior of Mellanox DC transport protocol for different messaging patterns and designing an MPI library to make efficient use this protocol with multiple design variations
- Providing an analysis of the benefits and shortcomings of DC-based MPI operations and comparing with other state of the art protocols
- Showing the memory, performance, and scalability benefits with microbenchmarks and applications

Experimental results at the microbenchmark level show that the DC-based design in MVAPICH2 is able to deliver 42% and 43% improvement in latency for large message All-to-one exchanges over XRC, and RC respectively. DC-based designs are also able to give 20% and 8% improvement for small message One-to-all exchanges over RC and XRC respectively. For the All-to-all communication pattern, DC is able to deliver performance comparable to RC/XRC while outperforming them in memory consumption. At the application level, for NAMD on 620 processes, the DC-based designs in MVAPICH2 outperform designs based on RC, XRC, and UD by 22%, 10%, and 13%, respectively, in execution time. With DL-POLY, DC outperforms RC and XRC by 75% and 30% respectively in total completion time while delivering performance similar to UD.

2 InfiniBand Transport Services

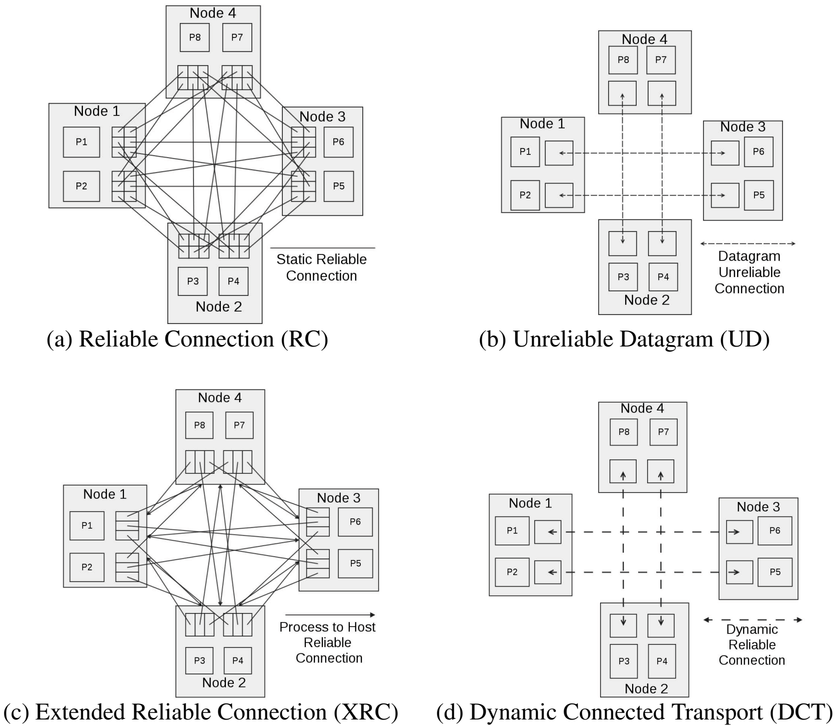
InfiniBand is a popular switched interconnect fabric used by 41% of the Top500 Supercomputing systems [10]. InfiniBand Architecture [9] defines a switched network fabric for interconnecting processing and I/O nodes, using a queue-based model. It supports two communication semantics: Channel Semantics (Send-Receive communication) over RC, XRC, and UD; and Memory Semantics (Remote Direct Memory Access communication) over RC and XRC. Both semantics can perform zero-copy transfers from source-to-destination buffers without additional host-level memory copies. RC is connection-oriented and requires dedicated QP for destination processes while the connection-less UD transport uses a single QP for all [6, 11]. XRC optimizes QP allocation by requiring each process to create only one QP per node [7].

3 Dynamic Connected (DC) Transport

In the following sections, we describe the connection model, the communication model, and the destination addressing scheme used for DC.

3.1 Connection Model

Figure 1 depicts the connection models for the different transport protocols InfiniBand supports for a fully-connected job. Each node has two processes that are fully connected to the all processes on other nodes. We do not account for intra-node IB connections since the focus of this paper is on MPI and MPI libraries generally use a shared memory channel for communication within a node instead of network loopback. For generality, we assume that the cluster has N nodes with C cores per node in all equations.



(a) Reliable Connection (RC)

(b) Unreliable Datagram (UD)

(c) Extended Reliable Connection (XRC)

(d) Dynamic Connected Transport (DCT)

Fig. 1. Connection models for different transport protocols in InfiniBand

Figure 1a depicts the connection model for RC. To maintain full connectivity in a cluster, each process must have *(N — 1)*C* QPs. Figure 1c shows a fully-connected XRC job. Instead of requiring a new QP for each process, now each process needs to only have one QP per node to be fully connected. In the best case the number of QPs required for a fully-connected job in a cluster, is only N QPs. This reduces the number of QPs required by a factor of C. However, on next-generation exascale systems where the number of hosts is expected to reach O(100,000) to O(1,000,000) nodes [8], this still presents a huge overhead in terms of memory. Figures 1b and 1d represent the connection model for UD and DC respectively. As we can see, the number of QPs

required for a fully-connected job reduces to a constant value irrespective of the number of processes in the job. This presents a very compelling reason to use these protocols to design next-generation communication middlewares.

3.2 Communication Objects, Addressing Scheme and Communication Model

Communication Objects: The main communication objects used in DC are DC-Initiators (DCINI) and DC Targets (DCTGT). DCINIs are analogous to the send QPs used in other IB protocols. Any process that has to transmit data to a peer using DC must create a DCINI for this purpose. Processes can use DCINIs as they would use a UD QP, i.e. 1) it can be used to transmit data to any peer as seen in Section 3.1 and 2) we can transition the state of the DCINI without providing the information about the remote QP. DCTGTs are analogous to the receive QPs. But unlike RC and UD where the use of a Shared Receive Queue (SRQ) [12] is optional, the DCTGTs *must* be backed by an SRQ.

Addressing Scheme: Each DCTGT will be assigned a number called the *dct_number* by the IB HCA. This value will be unique to the HCA. Thus one can uniquely identify a DCTGT on the network by a combination of the Local Identifier (LID) that is unique to the IB subnet and the DCTGT number. As we saw before, a DCINI does not require the target information while transitioning to the Ready To Send (RTS) state. Hence, the protocol uses a combination of an address handler field and the number of the remote DCTGT to route the packet to the correct destination. The address handler field contains the remote LID to aid in the routing decision in the network. An additional parameter known as the *dc_key* must be provided to all the DC objects to enable communication. This parameter must be same across all processes that wish to communicate with each other.

Communication Model: Figure 2 depicts the communication model used by DC to establish / tear-down connections and progress communication. As describe above, the first step is for the processes to create DCINIs / DCTGT and transition them to the appropriate state. Once this is done, the processes are ready to send and receive data. Lets say Process-2 (P2) enqueues WQEs to send two data items to Process-1 (P1) and one data item to Process-3 (P3). The HCA at P2 will first send out a connect message in an *inline* fashion to P1 ahead of sending the actual data. On receiving the *Connect* message, the target HCA temporarily allocates a context to receive the incoming data. Once the data arrives, it looks up the DCTGT number and places it in the SRQ specified by the user when the DCTGT was created. The connection is kept alive by the target HCA in anticipation of receiving further messages from the same sender. If such messages are not received, the target times out and releases the context thereby breaking the connection. In this case, due to the back-to-back nature of the sends from P2, the connection will still be alive when the second packet arrives. Thus the HCA at P1 places the data in the appropriate SRQ as before. As P2 is using the same DCINI to send data to both P1 and P3, it will issue a *Disconnect* message to P1 as soon as it receives the ACK for the last packet from P1. We use the term ***sender-initiated disconnect*** to refer to this behavior in the rest of the paper. P2 will then send a *Connect* message in an

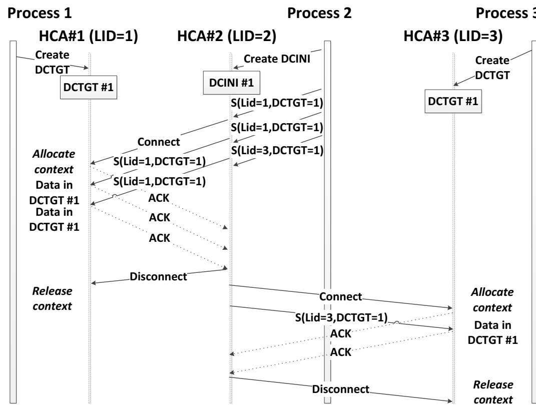


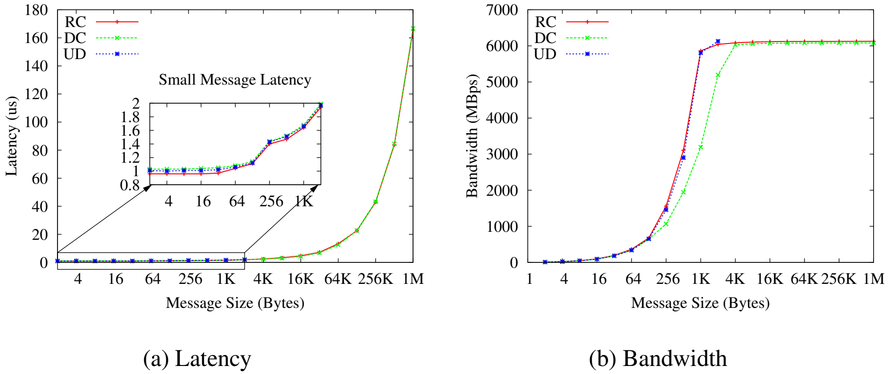
Fig. 2. Communication model(Courtesy [13])

inline fashion as before to P3 followed by the actual data. The HCA at P3 follows the same pattern as the HCA at P1 for the first message it received, as depicted in Figure 2.

4 Understanding Basic DC Performance

We use several verbs-level benchmarks in order to understand the performance tradeoffs associated with using DC protocol. We first use verbs-level point-to-point latency and bandwidth benchmarks to understand and analyze the performance one can obtain by using the DC protocol. We do not show numbers for the XRC protocol as the behavior will be similar to the RC protocol for small systems sizes. Notably, the verbs-level benchmarks only measure latency and bandwidth using the UD protocol up to 2,048 bytes. Beyond this the time to perform segmentation and re-assembly of packets must be taken into account to accurately measure latency and bandwidth using UD.

Figure 3a compares the verbs-level, point-to-point latency observed with RC, DC, and UD protocols for different message sizes. As seen, RC gives the best latency for small messages. We can also observe that DC adds an additional 100 nanoseconds of latency on top of RC. We believe that this is due to overhead involved with establishing connections for each send operation. Due to the ping-pong nature of the benchmark, the DCTGT will get timed out after each send operation, thereby breaking the connection and forcing the DCINI to establish connection with each send operation. With large messages, this overhead is not significant as the time taken for transmitting the data is much greater than the overhead of connection establishment. Figure 3b depicts the verbs-level, point-to-point bandwidth obtained for RC, DC, and UD protocols. As we can see, the RC delivers peak performance for all message ranges. DC is able to deliver performance on par with RC for small and large messages. However, it performs worse than RC in the medium message range. We are still investigating the reasons behind this performance drop.

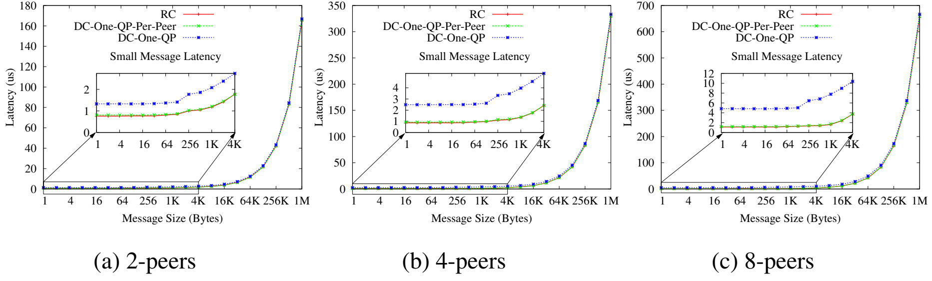


(a) Latency (b) Bandwidth

Fig. 3. Verbs-level point-to-point performance

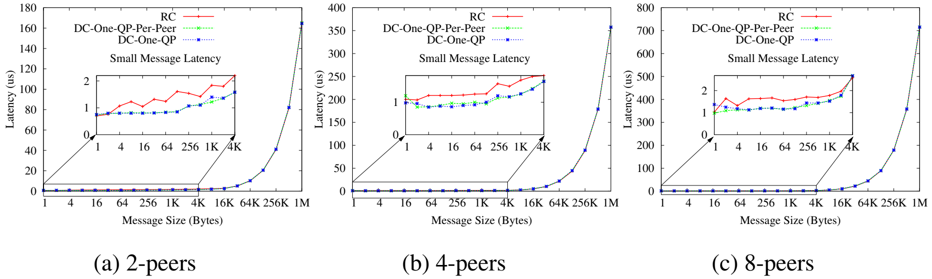
Next we evaluate two collective communication patterns that are common in scientific applications - *One-to-all* and *All-to-one*. We design and develop two new verbs-level benchmarks based on the basic point-to-point, verbs-level latency benchmark for this purpose. Due to the overheads of maintaining reliability and flow control, we do not enable the use of UD protocol for this benchmark. As we saw in Section 3.2, the use of just one DCINI has the potential to create significant serialization at the sender side. To evaluate the impact of this serialization on the performance of One-to-all and All-to-one communication patterns, we run experiments with DC in two modes - 1) sender using only one DCINI to perform all send operations (indicated by DC-One-QP) and 2) sender using separate DCINI per communicating peer for communication (indicated by DC-One-QP-Per-Peer). Figure 4 compares the performance of RC, DC-One-QP-Per-Peer, and DC-One-QP for the One-to-all pattern. As we can see, with an increasing number of peers, the performance of DC-One-QP for small messages degrades significantly. On the other hand, DC-One-QP-Per-Peer is able to deliver performance comparable to RC. Next we evaluate the performance of the All-to-one communication pattern with RC, DC-One-QP and DC-One-QP-Per-Peer. Figure 5 depicts the performance of the All-to-one benchmark with increasing number of peers. As we can see, both DC-One-QP and DC-One-QP-Per-Peer perform slightly better than RC for small messages. For all other message sizes, we do not observe any significant difference in performance between the RC and the two DC variants.

In summary, we observe four major trends with DC - 1) DC adds an overhead of 100 nanoseconds when compared with RC if there are frequent disconnections, 2) One-to-all performance of small messages can be affected significantly if the sender is only using one DCINI, 3) DC performs better for All-to-one pattern with small messages, and 4) the number of DCINIs does not affect the performance of All-to-one communication pattern. Another side effect of serialization when using just one DCINI can come in the form of head-of-line blocking. This can happen if a small message destined for peer A gets blocked by an ongoing large message transmission to peer B on the same DCINI.



(a) 2-peers (b) 4-peers (c) 8-peers

Fig. 4. Verbs-level One-to-all latency



(a) 2-peers (b) 4-peers (c) 8-peers

Fig. 5. Verbs-level All-to-one latency

5 Design of DC-Aware MPI Library

As we saw in Section 4, there are several design challenges that need to be overcome to create a high performance MPI library that uses DC. The design needs to achieve four major goals:

1. Avoid the negative effects of serialization with one DCINI
2. Avoid the performance penalty of sender-initiated connection tear down
3. Reduce the communication memory footprint of the MPI library and,
4. Avoid head-of-line blocking with DCINIs where small messages may get blocked behind an ongoing large message transfer to a different peer

Notably the overhead of connection tear down due to the sender having no data to send is a separate issue and needs to be handled at the HCA. With these goals in mind, we propose three different DC-aware designs for the MVAPICH2 MPI library.

Design-1 (*DC-E-RC*): As we saw in Figures 4 and 5, by using one DCINI per peer, DC-One-QP-Per-Peer is able to perform on par with RC. Hence, the simplest approach would be to create a design where we dedicate one DCINI per communicating peer. We call this design "DC-Emulating-RC" or *DC-E-RC* in short. As we are dedicating one DCINI per peer, we will avoid the negative effects of serialization. We will also avoid

the performance penalty of sender-initiated disconnects as one DCINI will only be used to send data to one peer. Thus, using *DC-E-RC*, we will be able to achieve goals 1, 2, and 4 that we identified above.

Design-2 (*DC-E-UD*): Although the One-to-all communication pattern requires multiple DCINIs to be created to avoid the effects of serialization, the basic point-to-point communication pattern, as well as the All-to-one pattern, performed equally well with DC-One-QP. Hence, depending on the communication pattern and the number of processes involved, it is possible that we may not require multiple DCINIs. However, using just one DCINI for small and large messages can cause head-of-line blocking we identified in Section 4. Hence we use two DCINIs - one for small messages and one for large messages to avoid this. Furthermore, using just two DCINIs will present us with significant savings in memory especially as the scale increases. Since we emulate the UD transport protocol by using just one set of QPs, we term this “DC-Emulating-UD” or *DC-E-UD* in short. Thus, using *DC-E-UD*, we achieve goals 3 and 4. We may also achieve goals 1 and 2 depending on the communication pattern.

Design-3 (*DC-Pool*): As noted above, the *DC-E-RC* design has the potential to alleviate the negative effects of serialization at the DCINI. However, it will cause a huge bloat in memory due to the over-provisioning of DCINIs. *DC-E-UD* on the other hand will reduce the memory footprint but may fail to alleviate the effects of serialization for all communication patterns. Hence, an approach that takes a middle path between these two extreme scenarios may be able to achieve the first three goals. With this in mind, we propose the third hybrid scheme. Here, we use a pool of DCINIs for sending data. When a process wants to send data, it picks a free DCINI from the pool, transmits data and then returns it back to the pool. As we choose a different DCINI for each transmission, it can avoid the negative effects of serialization we identified in Section 4. Furthermore, by limiting the number of DCINIs created to a small value, we also limit the communication memory footprint of the MPI library to a constant value. Thus, we will be able to achieve goals 1 and 3 with this design.

However, due to the fact that the DCINIs get used for both large and small message transmission, head-of-line blocking scenarios may still occur. Furthermore, as each send uses a separate DCINI, it is virtually guaranteed that sender-initiated disconnects will happen with each send operation even if subsequent sends are to the same peer. To remedy these two issues, we propose two optimizations to the pool-based approach: 1) We split our DCINI pool into two - one for small messages and one for large messages. Thus we ensure that a small message never gets blocked by a large message transfer to a different peer. 2) Once a process selects a DCINI for performing communication with a peer, it stores the information of the DCINI in the Virtual Channel (VC) data structure called VC table (VCT) maintained in MVAPICH2. The VC table is used to keep track of the current state of communication with each peer a process has. For future transmissions to the same peer, the process will just re-use the same DCINI. By doing this, we avoid sender-initiated disconnects from happening if a process is continually communicating with a particular peer. We call this hybrid design as *DC*.

Figure 6 depicts how the *DC-Pool* design is implemented inside the MVAPICH2 MPI library. MVAPICH2 uses an *on-demand* method to establish connections. In this

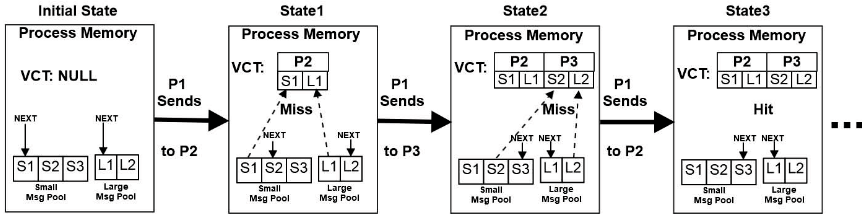


Fig. 6. DC-Pool-based design

method, a connection is established with a peer only if a process needs to communicate with it. Hence at the initial state (just after job startup), the VC table would be empty or NULL. We create two pools of DCINIs - one for small messages and one for large messages. We also initialize *NEXT* pointers that point to next DCINI that needs to be used in the pool. The DCINIs are selected in a round robin fashion to distribute the load. From this initial state, lets assume that the sender process (P1) wants to communicate with P2 (represented by State 1 in Figure 6). It will check the VC table to see if a communication channel already exists for that process. If it does not, it is considered as a “Miss”. It will then retrieve the next small message and large message DCINIs from the pool and store the information of the DCINIs in the entry of VC table for the peer. It will also increment the *NEXT* pointers for both pools. The next time P1 wants to communicate with P2, it will retrieve the DCINI information directly from the VC table entry for P2 instead of using a new DCINI from the pool (represented by State 3). State 2 in Figure 6 depicts the state of the various data structures after P1 establishes connection with a new peer P3.

6 Experimental Results

In this section, we describe the experimental setup used to conduct micro-benchmark and application experiments to evaluate the efficacy of existing transport protocols and that of the proposed DC designs. An in-depth analysis of the results are also provided to correlate design motivations and observed behavior. All results reported here are averages of multiple runs to discard the effect of system noise.

6.1 Experimental Setup

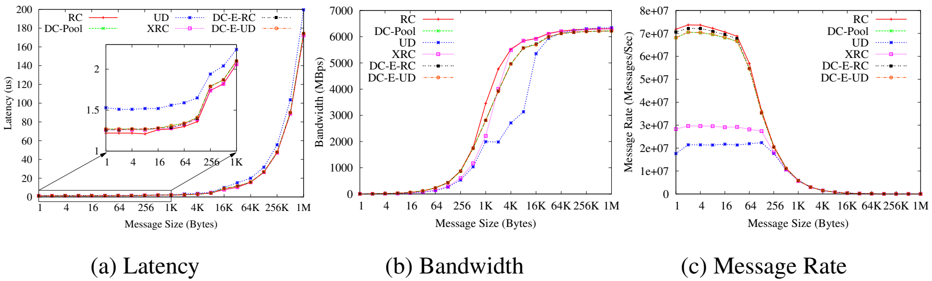
The setup consists of 32 Ivy Bridge Compute nodes interconnected by Mellanox FDR switch SX6036. The Intel Ivy Bridge processors consist of Xeon dual ten-core sockets operating at 2.80 GHz with 32 GB RAM. Each node is equipped with MT4113 Connect-IB HCAs (54 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is RHEL 6.2 with kernel version 2.6.32-220.el6, and Mellanox OpenFabrics version 2.1-1.0.0. All micro-benchmark results have been obtained through the use of OSU Micro-Benchmark suite (OMB) [14].

6.2 MPI Level Point-to-Point Results

In this section, we show the influence of transport protocol choice on the performance of communication between either a pair or multiple pairs of MPI processes through micro-benchmarks.

In Figure 7a, the effect of protocol choice on the latency of communicating messages is shown as a function of message size. As this experiment involves a single communicating pair, each process from a pair uses a maximum of one QP and hence the performance of DC-Pool, DC-E-UD, and DC-E-RC are on par. Further, any QP thrashing effects can be ruled out. It can be observed that the performance of RC in the short message range outperforms that of DC-Pool by around 100 nanoseconds. We attribute this degradation to connection re-establishment after the implicit teardown that occurs upon transmission of a message and the subsequent timeout that occurs when using DC-Pool. RC, on the other hand, maintains the connection until explicit teardown instruction and hence shows the best latency. Also, it is to be noted that use of UD transport mechanism has an overhead in comparison with the rest of the available mechanisms for the entire message range. This is an expected result owing to the combination of UD being a connection-less transport, being unreliable and in the large message range, suffering from packetization owing to MTU limits (on current Mellanox hardware, MTU is 2KB).

In Figure 7b, the effect of protocol choice on the bandwidth of communicating messages between a single pair of MPI process is shown as a function of message size. The experiment involves posting multiple non-blocking send operations and waiting for their the completion. It can be seen in the small message range that the bandwidth of DC-Pool, DC-E-UD, and DC-E-RC are on par with RC. This behavior is explained by noting that connection teardown occurs in DC if messages are not posted to the DC QP within a certain timeout interval. Within the short message range, this is an unlikely event for the bandwidth experiment due to multiple sends being posted and the relatively short amount of time needed to transfer each of them. However, for the rest of the range DC-Pool, DC-E-UD, and DC-E-RC show marginally smaller bandwidth in comparison with RC and XRC. This can be attributed to the effect of connection teardown that are likely to occur during the progression of medium and large message transfers with DC protocol owing to the larger latency involved in their transmission.



(a) Latency (b) Bandwidth (c) Message Rate

Fig. 7. MPI-level point-to-point performance

Lastly, in Figure 7c, the effect of protocol choice on the messaging rate between 20 pairs of MPI processes (20 processes on one node) is shown as a function of message size. As expected, due to connection teardown overheads, DC-Pool and its variants have marginally smaller messaging rates in comparison with RC. In the small message range, DC-E-RC performs slightly better than DC-E-UD or DC-Pool. From the point of view of the node HCA, the aggregate number of QPs to be managed increases from 20 to 40 as transport protocol is switched DC-E-RC from DC-E-UD or DC-Pool. For small messages, subsequent cache trashing leads to performance degradation in case of DC-E-UD or DC-Pool.

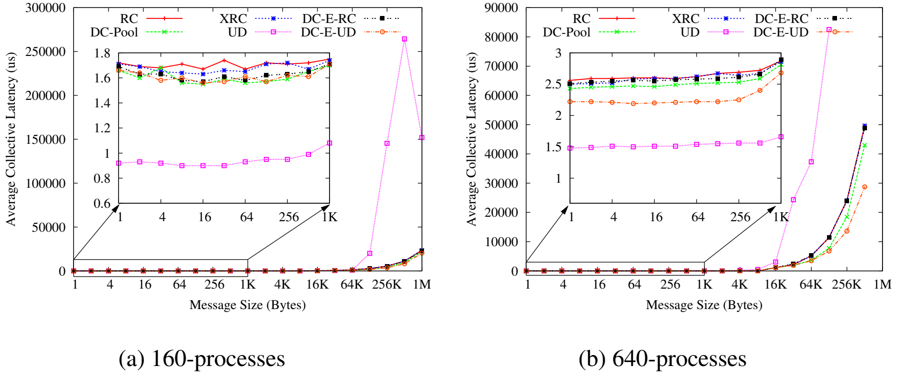
6.3 MPI Level Collective Results

MPI collectives predominantly rely on MPI point-to-point operations as their base primitives. As seen in Section 6.5, transport selection has an effect on performance on MPI point-to-point primitives and in this section we depict the effect of transport protocol choice on the performance of MPI collective communication patterns such as All-to-one, One-to-all, and All-to-all. All the collective experiments have been conducted with either 160, 320, or 640 processes to study the effect on scalability as well. All the figures depict the average of latencies experienced by all processes. Due to lack of space and the repetitive nature of the figures, we drop the results for 320 processes and just show results for 160 and 640 processes.

In Figures 8a and 8b, the performance of All-to-one collective is shown. For all message ranges, the root process receives a message directly from every other process involved in the collective. In the short message range, it can be seen that UD followed by DC-E-UD achieve the shortest latency among all the protocols. To understand this, the mechanism undertaken by the root must be considered. With RC/DC-Pool/XRC schemes, when the HCA at the root receives a message from a non-root, it has the overhead of sending back an ACK before processing the next message which is avoided in a UD-based scheme. However, beyond a message size of 2 KB, explicit software packetization and reliability overheads add to the latency and cause the latency to deteriorate. DC-E-UD performs relatively well in the entire message range owing to two reasons. First, it avoids cache thrashing experienced by RC, DC-E-RC, and XRC. Second, the serialization effects are avoided at a receiving DC QP, further bringing down the average latency of the All-to-one operation.

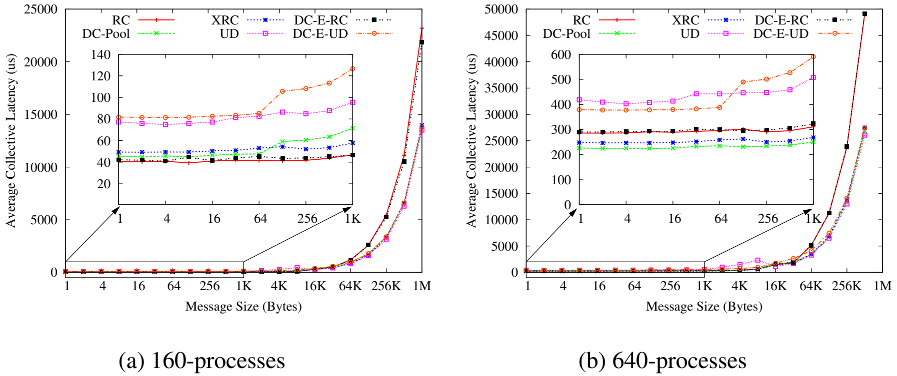
In Figures 9a and 9b, the performance of an One-to-all collective is shown. For all message ranges, the root process sends a message directly to every other process involved in the collective. The trends here are inverse of that noted for the All-to-one collective. In the short message range, the use of UD and DC-E-UD causes serialization due to the use of a single QP and hence these two schemes have the greatest latencies among all the protocols. Beyond the 16KB message range, RC and DC-E-RC schemes suffer from QP cache thrashing at the HCA which causes a significant increase in the average latency in comparison to the UD, XRC, and DC-E-UD.

Finally, Figures 10a and 10b depict the performance of the All-to-all personalized collective. The operation involves pairs of MPI processes taking turns to communicate with each other. Hence, it is clear that the performance of the operation largely hinges on the performance of the basic point-to-point primitive performance of underlying



(a) 160-processes (b) 640-processes

Fig. 8. MPI-level All-to-one performance



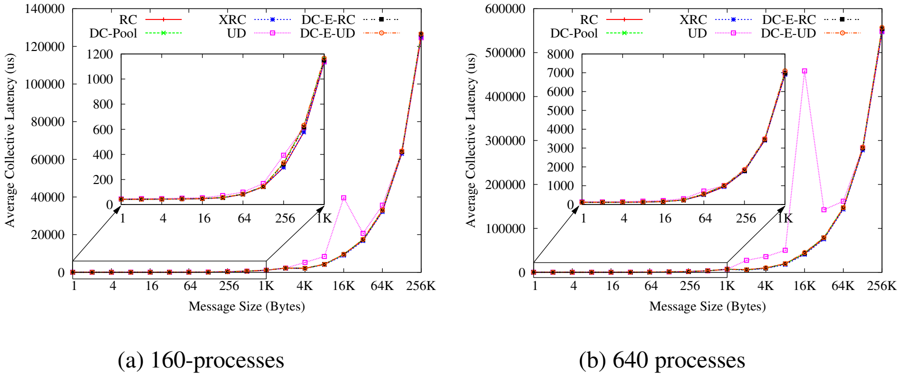
(a) 160-processes (b) 640-processes

Fig. 9. MPI-level One-to-all performance

transport protocols. As seen Section 6.5, UD is the only transport mechanism that shows considerable deviation from the rest of the protocols and the effect of this shows up in Figures 10a and 10b. At around the 16KB mark, UD starts showing a considerable increase in point-to-point latency (Figure 7a) and so do the All-to-all trends. It is to be noted that the abrupt peak and dip in case of UD is a result of switch from degraded packetized sends to zero copy UD rendezvous transfers [15].

6.4 Memory Footprint

Figure 11 contrasts the memory consumed by the MPI library on QP resources for different protocols. It can be seen that RC and DC-E-RC schemes consume the most memory as each process sets up dedicated QPs for every other process in the MPI job. UD scheme uses a single QP but has a constant overhead for rendezvous-protocol based



(a) 160-processes (b) 640 processes

Fig. 10. MPI-level All-to-all performance

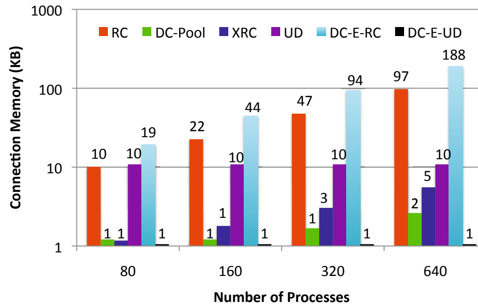


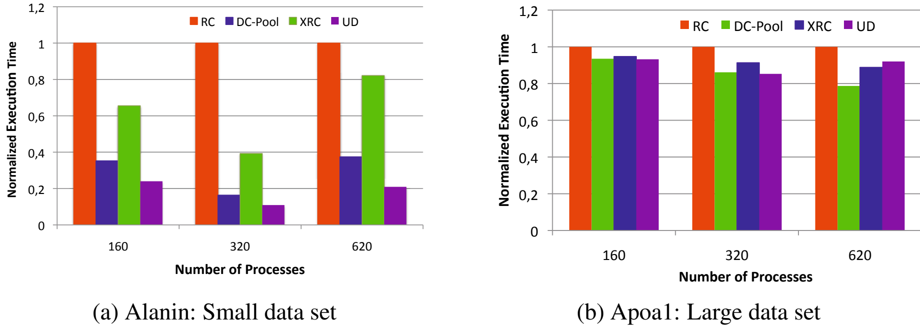
Fig. 11. Memory Footprint of Transport Protocols

transmissions. Under the XRC scheme, each process has a QP for every node and hence has a factor of core-count-per-node reduction in memory consumption in comparison to RC. The DC-Pool scheme has a further reduced memory footprint due to the use of minimal set of QPs. As DC-E-UD emulates UD scheme, there are only two DCINIs used for all connections and hence memory consumption remains constant.

6.5 Application Level Results

In this section we evaluate the different transport modes using two applications. These are more likely to model real-world use than microbenchmarks. We evaluate using the two molecular dynamics application NAMD and the DL-POLY.

NAMD: is a fully-featured, production molecular dynamics program for high performance simulation of large bimolecular systems [16]. NAMD is based on Charm++ parallel objects, which is a machine independent parallel programming system. The parallel decomposition strategy used by NAMD is a three-dimensional patchwork which



(a) Alanin: Small data set (b) Apoa1: Large data set

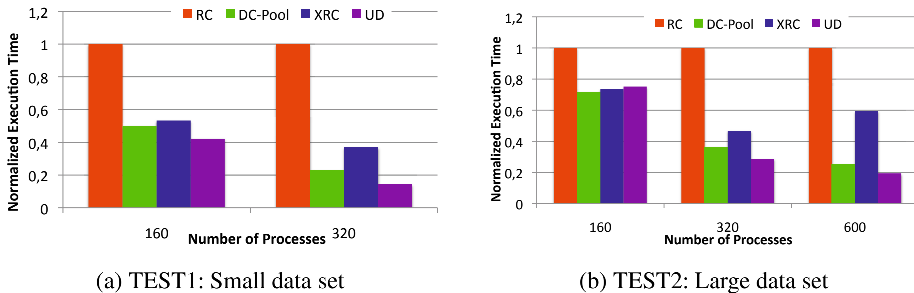
Fig. 12. NAMD application: Normalized execution time

involves a near-neighbors communication pattern. Of the standard data sets available for use with NAMD, we use the *apoa1* and *alanin* datasets.

DL-POLY: is a general purpose classical molecular dynamics (MD) simulation software developed at Daresbury Laboratory [17]. It spends 27% and 17% of its communication time which is almost 40% of the execution time on MPI_Allreduce and MPI_Send/Recv respectively. We used two standard data sets: the *TEST1: Sodium chloride (27,000 ions)* and the *TEST2: Sodium chloride (216,000 ions)*.

Figures 12a and 12b show the normalized execution time of NAMD on three different systems sizes (160, 320 and, 620 processes) using the *alanin* (small) and *apoa1* (large) data sets respectively. With *alanin* DC-Pool outperforms both RC and XRC modes by 84% and 25% respectively with 320 processes configuration, and by 63%, 35% with 620 processes configuration. With the three system size configurations, UD is performing the best as *alanin* is a small data set and hence the communications patterns use small message sizes. With large data set size, as shown in Figure 12b, DC-Pool is performing the best with 620 processes configuration. Indeed DC-Pool outperforms RC, XRC and UD by 22%, 10%, and 13% respectively.

Normalized execution time of DL-POLY with TEST1 and TEST2 is depicted in Figures 13a and 13b respectively. DL-POLY is enable to run at large scale system (600 cores), with a small data set (TEST1) as each process need to have a minimum of data to initialize. As shown in Figure 13, DC-Pool is performing better than RC and XRC by up to 77% and 13% respectively on 320 processes configuration. As with NAMD, for DL-POLY with small data set, UD is performing better than DC-Pool by 9% on the 320 processes configuration. With large data set (TEST2), DC-Pool is performing the best for 160 processes configuration. On 600 processes, DC-Pool shows 75% and 30% improvement compared to RC and XRC and almost the same performance than UD (4% benefits for UD). We are trying to understand the communication characteristics of DL-POLY better to analyze the reason for the benefit seen with UD.



(a) TEST1: Small data set (b) TEST2: Large data set

Fig. 13. DL-POLY application: Normalized execution time

7 Related Work

Scalability of MPI libraries over InfiniBand has been the focus of many recent research works. Koop, et al. proposed [6] and later improved [15] a connection-less, design based on the UD protocol. Similar designs have been proposed for iWARP as well. [18]. Hardware-based multicast over UD has been shown to improve performance of collectives in [19–21]. A hybrid scheme where UD was used for first 16 messages before setting up a RC connection was presented by Yu, et al [22]. The effects of different transport services on WAN was analyzed in [23]. A dynamic approach that uses both RC and UD protocols based on the communication pattern of the application has also been designed [11]. Using SRQ to reduce requirement of communication context has been studied in [24]. Shipman, et al. proposed using different SRQs for different data sizes to improve utilization of communication buffers on RC [25]. Erimli [26] used a method based on multiply-linked lists to share memory for WQEs. Connect-X, the first HCA to support XRC was evaluated by Sur, et al [24]. Koop, et al. explored design of MPI library, performance and memory footprint of XRC protocol in [7].

8 Conclusion and Future Work

In this paper we presented the salient features of the new DC protocol including its connection and communication models. We designed new verbs-level collective benchmarks to study the behavior of the new DC transport and understand the performance / memory trade-offs it presents. We then used this knowledge to propose multiple designs for MPI over DC. An implementation of our design in the MVAPICH2 MPI library was evaluated using standard MPI benchmarks and applications. To the best of our knowledge, this is the first such design of an MPI library over the new DC transport. Our experimental results at the microbenchmark level showed that the DC based design in MVAPICH2 was able to deliver 42% and 43% improvement in latency for large message All-to-one exchanges over XRC and RC respectively. DC based designs were also able to give 20% and 8% improvement for small message One-to-all exchanges over RC and XRC respectively. For the All-to-all communication pattern, DC was able

to deliver performance comparable to RC/XRC while outperforming in memory consumption. At the application level, for NAMD on 620 processes, the DC based designs in MVAPICH2 outperformed designs based on RC, XRC and UD by 22%, 10%, and 13% respectively. With DL-POLY, DC outperformed RC and XRC by 75%, and 30% respectively while delivering performance similar to UD.

References

1. InfiniBand Trade Association, `http://www.infinibandta.com`
2. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (March 1994)
3. Panda, D.K., Tomko, K., Schulz, K., Majumdar, A.: The MVAPICH Project: Evolution and Sustainability of an Open Source Production Quality MPI Library for HPC. In: Int'l Workshop on Sustainable Software for Science: Practice and Experiences, Held in Conjunction with Int'l Conference on Supercomputing, SC 2013 (November 2013)
4. The Open MPI Development Team: Open MPI: Open Source High Performance Computing, `http://www.open-mpi.org`
5. Intel Coporation: Intel MPI Library, `http://software.intel.com/en-us/intel-mpi-library/`
6. Koop, M.J., Sur, S., Gao, Q., Panda, D.K.: High Performance MPI Design using Unreliable Datagram for Ultra-scale InfiniBand Clusters. In: ICS 2007: Proceedings of the 21st Annual International Conference on Supercomputing, pp. 180–189. ACM, New York (2007)
7. Koop, M., Sridhar, J., Panda, D.K.: Scalable MPI Design over InfiniBand using eXtended Reliable Connection. IEEE Int'l Conference on Cluster Computing (Cluster 2008) (September 2008)
8. Kogge, P.: ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems
9. InfiniBand Trade Association: InfiniBand Architecture Specification 1, Release 1.0, `http://www.infinibandta.com`
10. Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: TOP 500 Supercomputer Sites, `http://www.top500.org`
11. Koop, M.J., Jones, T., Panda, D.K.: MVAPICH-Aptus: Scalable high-performance multi-transport MPI over InfiniBand. In: IPDPS 2008, pp. 1–12 (2008)
12. Sur, S., Chai, L., Jin, H.-W., Panda, D.K.: Shared Receive Queue Based Scalable MPI Design for InfiniBand Clusters. In: Proceedings of the 20th International Conference on Parallel and Distributed Processing, IPDPS 2006, p. 101. IEEE Computer Society, Washington, DC (2006)
13. Crupnicoff, D., Kagan, M., Shahar, A., Bloch, N., Chapman, H.: Dynamically Connected Transport Service (July 3, 2012), US Patent 8,213,315
14. Network Based Computing Laboratory: OSU Micro-benchmarks, `http://mvapich.cse.ohio-state.edu/benchmarks/`
15. Koop, M.J., Sur, S., Panda, D.K.: Zero-copy Protocol for MPI using InfiniBand Unreliable Datagram. In: CLUSTER 2007: Proceedings of the 2007 IEEE International Conference on Cluster Computing, pp. 179–186. IEEE Computer Society, Washington, DC (2007)
16. Phillips, J.C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R.D., Kale, L., Schulten, K.: Scalable Molecular Dynamics with NAMD. Journal of computational chemistry 26(16), 1781–1802 (2005)
17. Forester, T., Smith, W.: DL-POLY Package of Molecular Simulation. CCLRC, Daresbury Laboratory: Daresbury, Warrington, England (1996)

18. Rashti, M.J., Grant, R.E., Afsahi, A., Balaji, P.: iWARP redefined: Scalable connectionless communication over high-speed Ethernet. In: 2010 International Conference on High Performance Computing (HiPC), pp. 1–10. IEEE (2010)
19. Liu, J., Wu, J., Kini, S.P., Wyckoff, P., Panda, D.K.: High Performance RDMA-Based MPI Implementation over InfiniBand. In: 17th Annual ACM International Conference on Supercomputing (June 2003)
20. Mamidala, A., Liu, J., Panda, D.K.: Efficient Barrier and Allreduce on IBA clusters using hardware multicast and adaptive algorithms. In: IEEE Cluster Computing (2004)
21. Mamidala, A.R., Narravula, S., Vishnu, A., Santhanaraman, G., Panda, D.K.: On Using Connection-Oriented Vs. Connection-Less Transport for Performance and Scalability of Collective and One-Sided Operations: Trade-offs and Impact. In: Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, pp. 46–54. ACM (2007)
22. Yu, W., Gao, Q., Panda, D.K.: Adaptive Connection Management for Scalable MPI over InfiniBand. In: International Parallel and Distributed Processing Symposium, IPDPS (2006)
23. Yu, W., Rao, N.S., Vetter, J.S.: Experimental Analysis of InfiniBand Transport Services on WAN. In: International Conference on Networking, Architecture, and Storage, NAS 2008, pp. 233–240 (2008)
24. Sur, S., Chai, L., Jin, H.W., Panda, D.K.: Shared Receive Queue based Scalable MPI Design for InfiniBand Clusters. In: International Parallel and Distributed Processing Symposium, IPDPS (2006)
25. Shipman, G.M., Woodall, T.S., Graham, R.L., Maccabe, A.B., Bridges, P.G.: InfiniBand Scalability in Open MPI. In: 20th International Parallel and Distributed Processing Symposium, IPDPS 2006, p. 10. IEEE (2006)
26. Erimli, B.: Arrangement in an InfiniBand Channel Adapter for Sharing Memory Space for Work Queue Entries using Multiply-linked Lists (March18, 2008) US Patent 7,346,707