Efficient Collective Operations using Remote Memory Operations on VIA-Based Clusters

RINKU GUPTA, PAVAN BALAJI, JAREK NIEPLOCHA, AND DHABALESWAR K. PANDA

Technical Report
OSU-CISRC-1/03-TR03

Efficient Collective Operations using Remote Memory Operations on VIA-Based Clusters*

Rinku Gupta* Pavan Balaji* Dhabaleswar Panda* Jarek Nieplocha†

*The Ohio State University
{guptar, balaji, panda}@cis.ohio-state.edu

†Pacific Northwest National Lab
jarek.nieplocha@pnl.com

Abstract

High performance scientific applications require efficient and fast collective communication operations. Most collective communication operations have been built on top of point-to-point send/receive primitives. Modern user-level protocols such as VIA and the emerging InfiniBand architecture support remote DMA operations. These operations not only allow data to be moved between the nodes with low overhead but also allow the user to create and provide a logical shared memory address space across the nodes. This feature demonstrates potential for designing high performance and scalable collective operations. In this paper, we discuss the various design issues that may be the basis of a RDMA supported collective communication library. As a proof of concept, we have designed and implemented the RDMA-based broadcast and the RDMA-based allreduce operations. For RDMA-based broadcast, we get a benefit of 14%, when compared to send/receive-based broadcast for 4KB data size on a 16 node cluster. We also introduce a new reduce algorithm called as the Degree-k tree-based reduce algorithm. Combining the RDMA mechanism with the new reduce algorithm shows a benefit of 38% for 4 byte messages and 9% for 4KB messages on a 16 node cluster for the allreduce operation. We also introduce analytical models for broadcast and allreduce to predict the performance of this design for large-scale clusters. These analytical models yield a performance benefit of about 35-40% for 4 bytes and around 14% for 4KB messages for 512 and 1024 node clusters for the allreduce operation.

1 Introduction

High Speed interconnection networks and exponentially increasing microprocessor performance have made Networks of Workstations (NOWs) an increasingly appealing alternative to mainstream supercomputing for a variety of computational needs of computation intensive applications. Commonly known as Cluster Computing systems, these collections of commodity based components offer a high performance to price ratio to the end user, attributing to it's immense success. High Performance Parallel Programs on clusters often involve a lot of point-to-point and collective communication between them in addition to the computation being carried out.

Past works in the collective communication area have primarily focused on development of optimized and scalable algorithms on top of point-to-point send/receive operations [14]. The send/receive model requires explicit host intervention at both the sender and the receiver side. Modern user-level protocols such as the Virtual Interface Architecture (VIA) [8] and the InfiniBand Architecture (IBA) [1] offer a variety of models for data transfer. Together with the send/receive model, they also support the Remote Direct Memory Access (RDMA) model. The concept of Remote DMA is used for direct transfer of data between user spaces without any intervention from the receiving host. In other words, the RDMA operation is transparent to the receiver. Remote memory capability through RDMA operations allows the programmer to define a set of buffers across the nodes of a cluster which can be used as a logical shared address space to exchange data efficiently. This raises the following open question.

Can remote memory operations be used to design and implement efficient collective communication operations?

In our earlier work, we provided RDMA support for implementing fast barrier synchronization [6]. In this paper, we take up the challenge of exploring ways to design data-intensive collective operations such as broadcast and allreduce using the RDMA mechanism. We analyze various design issues and alternatives for supporting such collective operations with RDMA supported shared memory. For the broadcast and allreduce operations, we demonstrate how

*This Research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #EIA-9986052 and #CCR-0204429.

these issues have been resolved in practice in the design of high-performance collective communication libraries.

We introduce a new reduce algorithm called the *Degree-k* tree-based reduce algorithm. The implementation of allreduce using this new algorithm along with the RDMA mechanism gives significant performance benefits compared to the traditional send/receive-based allreduce operation. This benefit was found to be 38% and 9% for small (4 bytes) and large (*4KB*) messages respectively on a 16 node GigaNet cLAN cluster. To allow MPI applications take advantage of the new implementation, we linked the RDMA-based broadcast and allreduce algorithms with MVICH (a popular MPI implementation for VIA) [5].

We also present analytical models to find the optimal RDMA-based allreduce algorithm for a given configuration and data size, and to estimate the performance of the RDMA-based broadcast operation for a given data size. We use this to predict the performance benefits of using the RDMA-based collective operations for large clusters. The analytical model predicts a 20% improvement in the broadcast latency for 512-node systems. The predicted performance for RDMA-based allreduce shows a benefit of about 35-40% for small messages of 4 bytes and around 14% for messages of *4KB* size for 512 and 1024 node clusters. These results demonstrate that efficient collective operations can be built on next generation clusters with networks (such as InfiniBand and Quadrics) supporting RDMA-based mechanisms.

The remaining part of the paper is organized as follows. Section 2 provides an overview of VIA [8] and MPI. Section 3 discusses the motivation for this work. In Section 4, we discuss the basic design issues. The broadcast and allreduce, along with their design issues are discussed in Section 5. Section 6 provides the analytical models for broadcast and allreduce. We present the performance results (experimental and analytical) in Section 7 and conclude the paper in Section 8.

2 Overview of VIA and MPI

2.1 Virtual Interface Architecture

The Virtual Interface Architecture (VIA) has been standardized as a low latency and high bandwidth user-level protocol for System Area Networks (SANs).

The VIA architecture mainly aims at reducing the system processing overhead by decreasing the number of copies associated with a message transfer and removing the kernel from the critical path of the message. This is achieved by providing every consumer process a protected and directly accessible interface to the network named as a Virtual Interface(VI). Figure 1 illustrates the Virtual Interface Architecture model.

Each VI is a communication endpoint. Two VIs on different nodes can be connected to each other to form a logical bi-directional communication channel. An application can have multiple VIs. Each VI has a work queue consisting of send and a receive queue. Applications post requests to these queues in the form of VIA descriptors.

A VI descriptor is a data structure which contains all the information needed by the VIA provider to process the request. Each VI descriptor contains a Control Segment (CS), zero or more Data Segments (DS) and possibly an Address Segment (AS). The Data segment of the descriptor contains the information related to a registered user buffer. The registered memory can be referenced by the virtual address of the buffer and the handle that is obtained while registering that buffer. The VIA specifies two types of data transfer facilities: the traditional Send and Receive messaging model and the Remote Direct Memory Access (RDMA) model. In the send and receive model, each send descriptor on the local node has to be matched with a receive descriptor on the remote node. Failure to post a receive descriptor on the remote node may result in a message being dropped or a reliable connection broken.

In the RDMA model, the initiator specifies both the virtual address of the local user buffer and that of the remote user buffer. In this model, a descriptor does not have to be posted on the receiver side corresponding to every message. The exception to this case is when the RDMA Write is used in conjunction with immediate data, a receive descriptor is consumed at the receiver end.

VIA provides the RDMA Write and RDMA Read features. In the RDMA Write operation, the node writes directly to the remote node's memory. Similarly in the RDMA Read operation, the node reads directly from the remote node's memory. VIA does not support scatter of data, hence the destination buffer in the case of RDMA Write and RDMA Read has to be a contiguously registered buffer. The RDMA Read is an optional VIA feature. Hence, the work done in this paper exploits only the RDMA Write feature of the VIA.

Since the introduction of VIA, many software and hardware implementations of VIA have become available. Berkeley VIA [3], Firm VIA [2], M-VIA [10], Server Net VIA [13], GigaNet VIA [9] are among these implementations. In this paper, we use GigaNet VIA, a hardware implementation of VIA for experimental evaluation.

2.2. MPICH/MVICH

Message Passing Interface [12] is the most popular and widely used standard library specification for developing message passing high performance applications. It provides

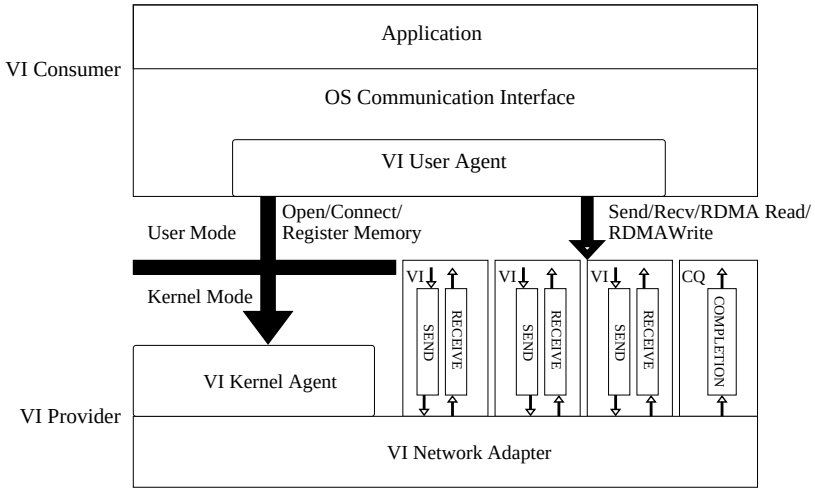

Figure 1. The Virtual Interface Architectural Model

portability and ease of use for the parallel programs written using the distributed memory programming model.

The MPI standard specifies a rich set of functions for point-to-point communication and collective communication, all scoped to a user specified group of processes.

MPI provides abstractions for processes at two levels. First, processes are named according to the rank of the group in which the communication is being performed. Second, virtual topologies allow Graph or Cartesian naming of processes that help relate the application semantics to the message passing semantics in a convenient and efficient way.

A key concept in MPI is that of a communicator, which provides a safe message-passing context for the multiple layers of software within an application that may need to perform message passing. For example, messages from a support library will not interfere with the other messages in the application, provided the support library uses a separate communicator. Communicators, which house group and communication context (scope) information, provide an important measure of safety that is necessary and useful for building up library-oriented parallel code.

Within a communicator, point-to-point and collective operations are also independent. An application can post several non blocking receive operations and then call a barrier collective operation. Messages used to complete the barrier operation will be processed independently from the posted receive operations. Most implementations of MPI simply use an additional *hidden* collective communicator to distinguish between peer communication and collective communication.

MVICH-1.0, a modification of MPICH [4] is an implementation of the MPI-1.0 standard for VIA platforms. It follows a layered structure with the *ADI* layer being at the lowest level. MVICH has no explicit primitives to support the RDMA write operation. We add explicit support for RDMA write in the *ADI layer*. To provide RDMA write support, we override the *MPI_Send()* primitive itself as and when needed. We cannot add any new constant parameters to *MPI_Send()* and hence this modification is done by setting a *global variable* before calling *MPI_Send()*. This *global variable*, visible below the *ADI layer*, allows the ADI to choose between message send and RDMA write. The *global variable* is by default set to *FALSE*, which indicates a *message send* to the *ADI* layer. We set this primitive to *TRUE* when we wish to perform a *RDMA write*.

3 Background Work

VIA and the emerging InfiniBand architecture support remote DMA operations, which allow the data to be moved between the user space of the communicating nodes with low overhead. This concept can be used to create and provide a logical shared memory address space across the nodes. Many efficient collective communication algorithms have been developed that are based on the send-receive paradigm. But the idea of providing a logical shared memory address space using the concept of RDMA on a distributed collection of nodes which has no shared memory has not been explored in the past. In a shared memory system, collective algorithms are simple and easy to implement. Consider the barrier collective operation which is essentially a synchronization operation.

In a shared memory system, a barrier operation can be done very easily. A section of memory (with multiple locations) can be reserved for the barrier and initialized to

'0'. Every process writes a '1' to a specified location in this memory region when it reaches the barrier. Next, the process reads from other memory locations to see if other nodes have reached the barrier. This concept is illustrated in Figure 2 with four processes (P0, P1, P2 and P3) and four memory locations. The figure shows the memory location corresponding to each process. In this figure, P0, P2, and P3 have already set the byte in their respective locations when they encounter the barrier and are waiting for P1 to set the value in its location. As soon as P1 sets the value in its location, all processes return from the barrier.

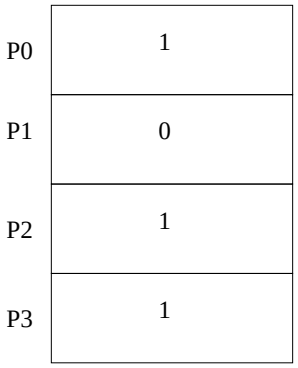

Figure 2. Illustration of a simple barrier scheme using multiple shared memory locations

If the shared memory is cache coherent, the barrier implementation turns out to be considerably simpler and faster. The processes obtain the data by a simple local read operation without additional complexities.

In a cluster with distributed memory organization, when an operation like barrier takes place, the nodes typically send and receive explicit messages. Barrier algorithms (pair-wise exchange with recursive doubling or gather-followed-by-broadcast [11]) with multiple phases (steps) are used to implement the barrier. Each of the communication step typically uses a send and receive primitive to communicate. Receiving a message from a node is typically an expensive operation. For example, an MPI over VIA implementation has to take care of unexpected receive messages. When messages come in, the relevant descriptor has to be searched for. If there is no descriptor posted, data is sent to an intermediate buffer. When the actual descriptor gets posted, the data has to be copied from the temporary buffer to the user buffer. In addition, the layering structure of the libraries like MVICH adds considerable overhead on the message latency, making each of the communication step slower and the entire barrier operation slower.

The method of RDMA communication offers a new mechanism for transferring data, by directly writing into the memory of a remote node. Consider a set of buffers being allocated at each remote node and their addresses being exchanged at the start of the program. The collection of these buffers (together with their addresses) provide a logical shared memory region (without coherency) for all nodes. Now, the nodes can exploit the advantages associated with shared memory-based algorithms to implement the barrier.

We use this concept to implement the RDMA supported barrier in one of our earlier works [6]. In this paper, we extend the concept to implement RDMA-based broadcast and allreduce.

4 Design Issues for RDMA Collective Communication Operations

The RDMA mechanism and memory registration constraints in VIA open up several major issues for designing a RDMA based collective communication library. In this section, we discuss the design issues and present some solutions. In the subsequent chapters we will discuss the design choices for the particular collective communication operation and its implementation.

4.1 Registration of buffers and Address Exchange

It is a requirement in VIA that data be sent and received from registered buffers. A flexible buffer management scheme is required for this purpose in the context of collective operations. In our scheme, we can register the buffers statically before the operation or dynamically during the operation.

Static Buffer Registration: We statically register a contiguous region in memory for each communicating group for various types of collective operations. This region is generally registered when the communicating group is being created. This contiguous region is split into fixed size buffers (also called as blocks). Since the memory allocated is contiguous, only the starting address of the memory (the address of the first buffer) needs to be communicated to the other nodes. This address is communicated only once during the initialization phase. The length of the buffer space is the same for all the nodes in the communicating group for a given operation. There will be certain constraints on the order of using these buffers, which are discussed in the later sections. Since the total number of buffers are constant we will need to provide a mechanism for safe reuse of these buffers. Since the buffers are pre-registered, the data has to be copied from these registered buffers to the user buffers when they become available. Hence, there is a copying cost involved.

Dynamic Buffer Registration: In the dynamic registration scheme, we allow the use of non-contiguous buffers. This will make it mandatory to communicate the addresses of all the buffers to all the nodes for every collective operation instance. Dynamic registration is not done at the start of the program, but as a part of the operation itself after the requisite user buffers have been declared. However, in this approach the buffer addresses need to be communicated whenever the buffers are created dynamically. Hence, if we register the buffer in the collective operation, we have the additional overhead of address communication with the destination set in the collective operation before sending the actual data to the destination set. However, in this scheme as the user buffers can be registered during the operation, there is no additional copying cost involved. The dynamic buffer registration is also known as the *rendez-vous* scheme.

4.2 Data Validity at the Receiver end

RDMA write is receiver transparent. It does not require that the receiver post a descriptor or perform any action in anticipation of the incoming data and the receiver process receives no indication that any new data has been written. When the destination needs the data it goes to the memory location and fetches the data from there by performing a local read operation. Thus, we need a mechanism for indicating to the receiver that the data in the memory is valid data.

There are various ways in which this can be done. One method is to let the receiver NIC interrupt the receiver once it receives an RDMA message. But this is a very expensive operation and thus detrimental to high performance. Another approach is to use the immediate field in the RDMA descriptor and set the field when the last RDMA write operation has taken place. However, this requires consumption of a descriptor at the receive end. This also requires that the receiver be aware of the data coming and post a receive descriptor in advance. This approach disturbs the illusion of shared memory and is not feasible.

Another approach is to write a special value, known to the receiver at a pre-defined location in the receiver's memory for each buffer in the pool. The value will be written after the sender has finished writing to the destination memory. This special value will indicate the data validity at the destination end. RDMA write supports the reading of data from non-contiguous locations but does not support scattering of data in a single RDMA write operation. Thus in a single operation, writing the data to the destination buffer and writing the special value to a separate buffer location at the destination end is not possible. To perform the data transfer, we will need to perform two RDMA writes, the first for sending the data to destination buffer and the second for updating the special byte which indicates the validity of data at the destination end. But performing two RDMA writes is very expensive. Also, the order in which the destination NIC will write the data in the destination memory is not fixed. The destination NIC, on receiving both RDMA writes may decide to write the special byte first, thus defeating the entire purpose of using special byte for indicating data validity at the receiver end.

Another approach would be to attach the special byte to the end of the data. Thus the sender sends the data and an extra byte with a special value to the destination. The destination knows when and how much data is arriving and thus it checks the byte at the end of data and determines the validity of data.

4.3 Safely reusing the buffers

In the static buffer allocation scheme, buffers are allocated during the initialization time. No new buffers are allocated in the course of the program. These finite number of buffers need to be reused. Before the buffer can be reused, the sender needs some confirmation from the receiver that the data from the buffer has been read and the buffer is safe to be reused. The buffers are contiguous in nature and are used contiguously. Thus for the same communicating group and the type of collective operation, the receiver knows when exactly the buffer is going to be reused by a sender. The receiver can then explicitly RDMA write a notification to the sender and the sender can proceed with the writing of data after it has received the notification.

In the dynamic buffer allocation scheme, the problem of reusing buffers does not arise as the buffers are allocated separately for each collective communication operation.

5 Algorithms and Design Choices

In this section, we discuss the design choices with relation to the RDMA-based broadcast and RDMA-based allreduce operations.

5.1 The Broadcast Algorithm

Broadcast is a frequently used collective communication operation involving data distribution. Given a collection of nodes, the broadcast operation distributes the data present at one node (called as the *root*) to all the other nodes in the communicator.

Higher level libraries use various algorithms to implement the broadcast operation. Libraries like MVICH use the binomial algorithm which is very efficient for small-large clusters. Some libraries use the linear method for implementing broadcast in very small clusters. Currently, binomial broadcast is implemented using the send/receive prim-

itives. We implement the same algorithm in RDMA and we compare the performance of the two.

In the binomial broadcast algorithm, sending of the data is divided into steps. Consider a cluster of 4 nodes P0, P1, P2, P3 where node P0 is the root with *id 0*. Nodes P1, P2, P3 have *ids 1, 2, 3* respectively. In Figure 3, in the first step, root P0 sends the the data to the node at size/2 away i.e to node P2. In the second step, root P0 sends the data to node P1. At the same time, node P2 becomes the root of a new subtree and forward the data to node P3. The process of forming new subtrees continues till the data reaches all the nodes. Figure 4 shows the steps in the binomial broadcast for a 8 node cluster.

For power of 2 nodes, $logN$ steps are needed for performing the binomial broadcast, where N is the number of nodes. For non-power of 2 nodes, the steps taken are $logN'$ where N' is the immediate higher power of 2 than N, where N is the total number of nodes.

The data structures and the working of the algorithm is discussed in the following subsections. The RDMA-based broadcast works in two modes. These modes are chosen to obtain the best performance for all data sizes. For smaller messages typically less than *5KB*, the data copy time is less , so we use a static buffer management scheme. As the data size goes on increasing, the data copying time increases and hence after a certain data size, the memory copying time typically kills the benefit obtained by the saved extra round trip time in the static registration scheme as compared to the dynamic registration. The sending the data is done by RDMA writing into specific buffers at the receiver's end. Our modifications and comparisons are based with the MVICH implementation of MPI. The MVICH reverts to a dynamic registration scheme for data size beyond *5KB*. Hence, for messages greater than *5KB*, when the data copy time is high we also use the dynamic registration scheme.

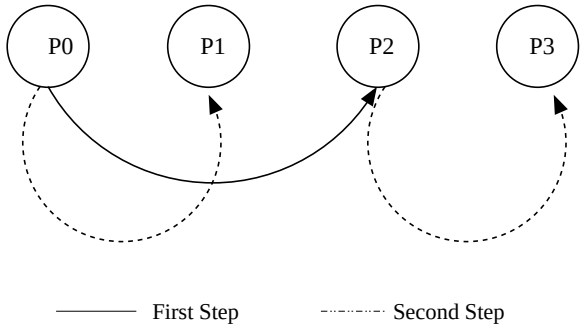

Figure 3. Broadcast using Binomial Algorithm in a 4 node cluster

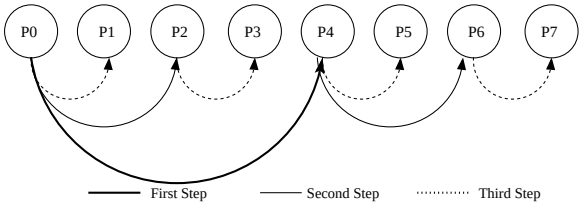

Figure 4. Broadcast using Binomial Algorithm in a 8 node cluster

5.1.1 Registration of buffers : Message Size $<$ *5KB*

For messages less than *5KB*, we follow the static buffer registration scheme. As mentioned in the previous section, a contiguous region in memory is allocated as *broadcast buffer* and is registered during initialization time. We divide the contiguous memory into blocks of fixed size denoted by *block_size*. The blocks are numbered from 0 onward. Every node has the addresses of the other broadcast buffers belonging to other nodes of the communicating group. Data has to be copied to the user specified buffers once it reaches these pre-registered broadcast buffers.

In addition to the main broadcast buffers, we also reserve a buffer called notification buffer which is used to indicate the safe reuse of the broadcast buffers. Every node registers its notify buffer. The size of the notify buffer in bytes is equal to the number of nodes in the communicating group involved in the broadcast operation. The address of the notify buffers is also exchanged during the initialization time. The broadcast and the notify buffers are initialized to -1 during the initialization time.

5.1.2 Registration of buffers : Message Size $>$ *5KB*

For messages greater than *5KB*, we adopt the *rendezvous* dynamic buffer registration approach. The destination buffers are registered and the addresses are exchanged during the operation itself using a request-reply mechanism. Thus, there are no pre-registered buffers or extra copying taking place. For large messages, typically greater than *5KB*, copying in static scheme becomes more expensive compared to the round trip overhead in *rendezvous* scheme. Hence we use the *rendezvous* scheme for larger message sizes.

The dynamic registration scheme is similar to the one adopted by MVICH implementation of MPI for messages over *5KB*. Hence, we discuss the remainder of the section with respect to the static broadcast scheme for messages smaller than *5KB*.

5.1.3 Data Validity at the Receiver end

To understand the working of the RDMA-based broadcast, consider Figure 5 with 4 processes P0, P1, P2, P3, where process P0 is the root and the broadcast instance shown is between the processes P0 and P2.

For every communicating group, we have a static counter called *broadcast counter* which is incremented by 1 for every broadcast operation, by every process within that communicating group. Consider the first broadcast of data size *block_size/2* bytes. The *broadcast counter* for the first broadcast is set to 1. The sender appends the *broadcast counter* byte at the end of the data to be written. The root P0 can RDMA write the data of size *block_size/2 + 1*, which includes the appended *broadcast counter*, to block #0 of process P2. For a communicating group, every node is involved in the collective operation. Hence, every node can keep track of the number of blocks used for that particular collective operation.

The data is written in a *bottom-fill* manner. To write 8 bytes in a 10-byte block in a *bottom-fill* manner, we start writing these 8 bytes from the 2nd byte onward in the block. Hence the data is always filled in the bottom portion of the block.

Figure 5 shows the broadcast data being written in the destination blocks in the *bottom-fill* manner, so that the sent *broadcast counter* is always written in the last byte of the block.

Since the receiver shares the communicating group with the sender, the *broadcast counter* at the sender and receiver will have the same value. Thus, the receiver can poll for the *broadcast counter* on the last byte of the received block and check for the validity of the data.

If the data to be sent is greater than the *block_size*, the data is split up into blocks of size *block_size - 1*, the *broadcast counter* is appended to each block and the data is then written to the remote node. Figure 5 also shows the second broadcast of *2 * block_size* bytes. Root P0 writes the first and second blocks of size *block_size - 1*, with *broadcast counter* of 2 appended, to block #1 and block #2 (of Process P2) respectively. The additional 2 bytes are written in block #3, again with the *broadcast counter* of 2 attached for data validity.

Writing large messages by breaking them into blocks at the lowest level enables pipelining of messages and overlapping of the copy to the user buffers at the destination. However, there is a trade-off involved between the *block_size* and number of RDMA writes. It takes 1 RDMA write to send each block. If the *block_size* is too large, the number of RDMA writes will be low but the copying cost to the buffers will be high. If the *block_size* is small, the copying cost will be low, but the number of RDMA writes and contention at the switches and NICs will be high. The processing of a large number of *RDMA writes* might offset the benefit obtained by overlapping the copies. Thus, it is desirable to find an optimal *block_size* where the cost of processing multiple *RDMA writes* does not kill the benefit achieved by overlapping memory copies. In our results section, we evaluate our RDMA-based broadcast algorithm with different *block_sizes* to obtain an optimal one.

Once the data is read by the receiver, the receiver, if needed, can forward the data to the other nodes directly from the received buffer. In our scheme, a node will forward the message only after it receives all the blocks of that message. After sending the data to the other nodes, the receiver will need to reset the last polling byte of the received blocks to -1, so that they can be safely reused at a later stage.

5.1.4 Buffer reusing

We need an explicit mechanism in the static buffer registration scheme for the broadcast operation to indicate safe reuse of buffers. We implement this by using the pre-registered notification buffers.

When the receiver realizes that data to be written is in the first block of the broadcast buffer, it writes a special value in the senders notification buffer as shown in Figure 6. Before writing, the last byte of each block has to be set to -1. This is because we might encounter a situation where a *broadcast counter* with a given value (*for ex. 20*) was written in an earlier broadcast. As the static *broadcast count* is incremented and wrapped around when its limit is reached, during a later broadcast, we may end up writing the same broadcast value, in which case the receiver might end up reading the old broadcast data.

Writing data in a *bottom-fill* manner requires us to reset only the last byte of each block. If data was written starting from the top of the block, the polling byte i.e *broadcast counter* would have been written at an arbitrary location depending on the size of data being sent and hence all the bytes of all blocks would have to be reinitialized which may be an expensive operation.

5.2 RDMA-based Allreduce

The allreduce collective operation is a global reduction operation, which takes place across all the members in the communicator. It combines values from different processes based on the reduction operation and the result is communicated back to all the nodes involved in the communication.

Allreduce is generally implemented as a combination of the reduce operation followed by the broadcast operation. Reduce operation is a variation of the allreduce operation, in which the result is present at only one node labeled as

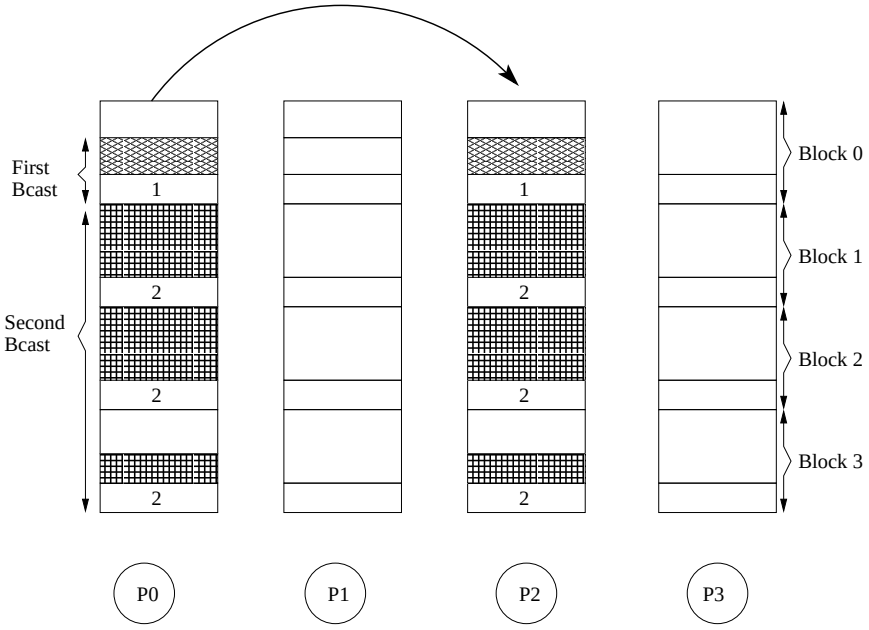

Figure 5. Two Consecutive RDMA-based Broadcast instances at Root P0

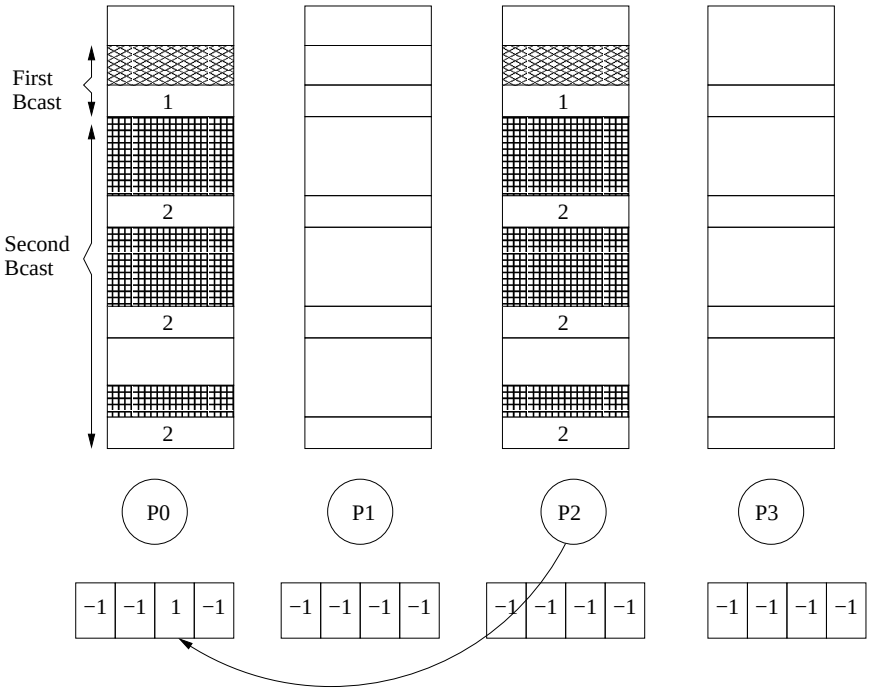

Figure 6. Notification by Process 2 to Process 1 before reusing first block

the *root* at the end of the operation. The reduce algorithm can be implemented in different ways. Current libraries like MVICH implement reduce using a recursive binomial algorithm. We introduce a new algorithm for reduce called the *Degree-k* algorithm. The new algorithm when implemented for the allreduce operation with the RDMA mechanism was found to give good performance

Definition: An *Degree-k tree-based Reduce* defines a tree where any node can receive messages from at most k nodes in any step of the reduce operation. The variable k is a *(power of 2) - 1* value. For a cluster of size N, where N is a *power of 2*, we can use all those *Degree-k tree-based* reduce schemes, where k is *(power of 2) - 1* and $k < N$. The binomial reduce algorithm is similar to the Degree-1 tree-based reduce algorithm. Since, most computing clusters generally have a *power of 2* size, we constrain the Degree-k tree-based reduce scheme to a cluster having *power of 2* nodes.

Hence, in the remaining part of the paper, the cluster size is implicitly assumed to be a *power of 2*, when it is mentioned in relation to the Degree-k RDMA-based allreduce scheme.

We implement a Degree-k RDMA-based allreduce as a combination of the Degree-k tree-based reduce and the binomial broadcast with the RDMA mechanism. To understand the Degree-k RDMA-based allreduce concept, let us consider a 4 node cluster. An allreduce operation on such a cluster can be implemented using Degree-1 or Degree-3 RDMA-based allreduce scheme. Figure 7 shows the tree for 4 processes P0, P1, P2, and P3 having *ids 0, 1, 2 and 3 respectively*. The square brackets indicate the *step number* for that node. In a Degree-1 RDMA-based allreduce scheme, every node will receive data from at-most 1 node in each step. Hence, in the first step Process P1 and Process P3 send data to Process P0 and Process P2 respectively. P0 and P2 will perform the required computation with the data acquired from P1 and P3 respectively and store the result. In the second step, Process P2 will forward its computed result to Process P0. Process P0 will then perform the computation with its own result of the previous step and the newly received result from P2 to get the final result. This final result will then be broadcast to all the other nodes involved in the allreduce operation.

Consider a Degree-3 RDMA-based allreduce scheme on the same cluster. In the Degree-3 RDMA-based allreduce scheme, a node can receive data from at-most 3 nodes in a step. Hence, as seen in Figure 8, processes P1, P2 and P3, having *ids 1, 2 and 3* respectively send the data to process P0, which has *id 0*. P0 will first perform the reduction operation on its own data and on the data sent by the node having *id 1*. The second operation is performed by P0, on this new result and the data sent by Process P2, having *id 2*. The last operation is done on the most recently computed result by P0 and the data sent by Process P3, with *id 3*. Thus, P0 will choose the order of evaluating the data based on the ascending order of the ranks of the sending nodes. The result at P0 is then broadcast to all the nodes in the group.

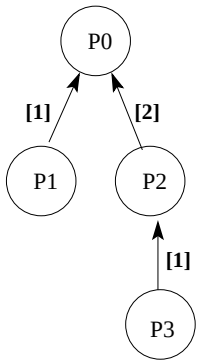

Figure 7. Degree-1 RDMA-based allreduce in a 4 node cluster

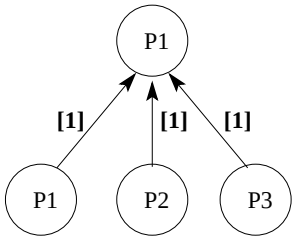

Figure 8. Degree-3 RDMA-based allreduce in a 4 node cluster

A Degree-3 RDMA-based allreduce for a 32 node cluster will have the tree as described in Figure 9. There is a trade-off involved between the number of steps in the Degree-k RDMA-based allreduce collective operation and the overhead incurred by the node in performing the reduction operation. For example, in a Degree-1 RDMA-based allreduce scheme for a 4 node cluster, there are 2 steps involved and in each step, a receiver node receives only 1 message and hence performs only 1 operation. In a Degree-3 RDMA-based allreduce scheme for a 4 node cluster, there is 1 step but the receiver nodes does 3 operations. So, depending upon the number of nodes, number of steps and the number of operations involved, we can choose different Degree-k RDMA-based allreduce algorithms.

The design solutions for all the Degree-k RDMA-based allreduce algorithm are the same. We explain the design solutions for a Degree-1 RDMA-based allreduce algorithm in the following few subsections. The RDMA-based allreduce algorithm works in 2 modes, depending on the size of the data to be transferred. These modes are similar to the

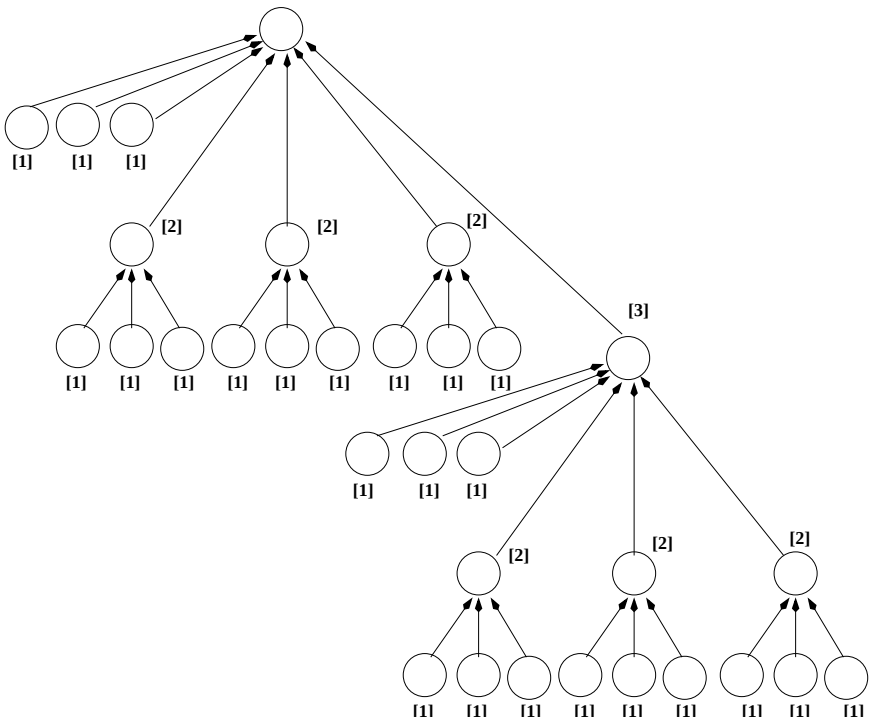

Figure 9. Degree-3 RDMA-based allreduce in a 32 node cluster

RDMA-based broadcast modes. We choose the static registration scheme for data size lesser than *5KB* because data copy for smaller bytes is not very expensive. For data size larger than *5KB*, data copy becomes expensive and so we use the dynamic registration scheme.

5.2.1 Registration of buffers: Message Size $<$ *5KB*

We allocate a contiguous registered section of memory called *allreduce buffers*, split into *block_size* of *(5K + 1)* bytes, because *5KB* is the maximum size of data that can be transferred in the static scheme. Also, the total memory region reserved need not be greater than $block\_size * N$, where N is the number of processes in the communicator. This is because, in the Degree-(*N-1*) RDMA-based allreduce algorithm, a maximum of $N - 1$ processes can write to a receiver node. For sending the data in the static scheme, the sender RDMA writes the data to the receiver's allreduce buffers. Figure 10 shows the 4 processes P0, P1, P2 and P3 with *ids 0, 1, 2, 3* respectively, each having 4 contiguous blocks of memory reserved and registered for the allreduce operation.

5.2.2 Registration of buffers: Message Size $>$ *5KB*

For messages greater than *5KB*, we follow the *rendezvous* scheme as described in the RDMA-based broadcast section. Since MVICH-1.0 also implements the same scheme for message size $>$ *5KB*, the remainder of the allreduce discussion will be only for messages less than *5KB*.

5.2.3 Data Validity at the Receiver end

Consider the node P1 sending data to node P0 and node P3 sending data to node P2. The node P1, with *id 1*, RDMA writes the data to block #1 of the receiver's allreduce buffer. A node always writes the data to the block having the same number as its *id*. Any node with any *id* can write to any other node's allreduce buffer at a location indicated by its *id*. This indicates to the receiver the identification of the sender of the data and also enables an ordered evaluation of the data.

Node P1 sends the entire data in 1 single RDMA write to the node P0. Similar to the broadcast operation, the allreduce operation has a static *allreduce counter* which is appended to the block of data when it is sent. The data is written in a *bottom-fill* manner. The receiver performs the required operation and the result is stored in the same location as that of the latest received data from the node having

the greatest id. Figure 10 shows node P1 with *id 1* and node P3, with *id 3*, writing the data to node P0's block #1 and to node P2's block #3 respectively. Assuming this is the first allreduce, *allreduce counter byte* is set to 1.

Figure 11 shows node P0 and node P2 performing the summation operation and writing the intermediate results in blocks #1 and block #3, respectively. Data is not broken down into smaller blocks and sent. This is to avoid the additional overhead of assembling and packing the data together before performing the required computation. The overhead of copying and packing data was found to be larger than the overhead of sending the entire data in a single RDMA write operation.

In the second step of the Degree-1 RDMA-based allreduce, node P2 with *id 2* will RDMA write its latest computed result to the allreduce block #2 of node P0 with the *allreduce counter* of 1 attached at the end of the data as shown in Figure 12.

Figure 13 shows node P0 performing the operation on the newly received data from node P2 and its own computed result obtained in the previous step. The result of this operation is stored in block #2 at P0. The result is copied by the root, i.e., node P0 to its receive buffer after it is done with its final computation. The result is broadcast from this receive buffer to all the nodes.

5.2.4 Buffer reusing

The blocks can be safely reused by the nodes without any additional messages being sent. In an allreduce, the reduce operation is followed by a broadcast. The second allreduce operation starts only after all the nodes have received the broadcast results of the first operation. Hence, the nodes can RDMA write the data to the same locations, as the data previously written has been used for successful computation.

6 Analytical Models

In this section we develop and present analytical models for binomial RDMA-based broadcast and degree-k RDMA-based allreduce operations. These models help to estimate performance of collective operations for large scale systems.

6.1 Binomial RDMA-based broadcast Analytical model

A message transfer in binomial RDMA-based broadcast operation consists of many events and overheads due to the communication stack. At the sender side, there is a copying overhead due to the need to copy data in registered buffers, the MPI library overhead, the cost involved in posting the send descriptor, DMAing the data from the host, the time for performing the processing by the NIC and the time taken by the acknowledgment message. The data is then transmitted to the destination. If the destination is connected to the source via a switch, the messages are sequentialized at that switch. Data can be broken into smaller blocks and sent which will enable pipelining and will overlap the memory copying with the sending side events. When the receiver NIC receives the data, it processes it and obtains the destination memory address (in case of RDMA write), to which the data is then DMAed. Copying into the user buffer takes place after the destination has forwarded the data to other nodes if necessary.

Based on the above events, we construct the analytical model with the following parameters :

1. Message size (given in *bytes*)
2. The MPI overhead time (given as *Tm*)
3. The descriptor posting time (given as *Td*)
4. The DMA startup time (given as *Ts*)
5. NIC processing time after the data is received by the NIC (given as *Tn*)
6. The data transmit time (given as *Tt*)
7. The acknowledgment time (given as *Ta*)
8. Memory copy time (given as *Tc*)

We assume that *Tn* is less than *Ts*, which is true for most current generations systems. Also, *Tt* and *Tc* depend upon the total bytes (*bytes*) that are being communicated in that message. Also, *Td* is generally a very small value.

Nodes in a large scale systems may be interconnected to each other through more than one switch. Every message that passes through the switch is delayed slightly by the switch latency. The switch latency however is only a factor of the data size and is independent of the mechanism used to transfer the data. Thus, in our analytical models for collective operations taking place via RDMA or Send/Receive, we choose to ignore the switch latency because of its *unvarying* contribution to the message latency.

Figure 14 shows the RDMA-based broadcast of 1 block of data between two nodes. If the number of nodes increases, the *STEP1* parameter will be multiplied by the number of steps in the broadcast operation. Hence for an *N* (where *N* is power of 2) node system, we will have the total time as : *2*Tc + log(N)*STEP1*.

Figure 15 shows the RDMA-based broadcast of a message sent as 2 blocks of data between a pair of nodes. When a message is broken into smaller blocks, the last block sent may have size lesser than the *block_size*. GigaNet cLAN

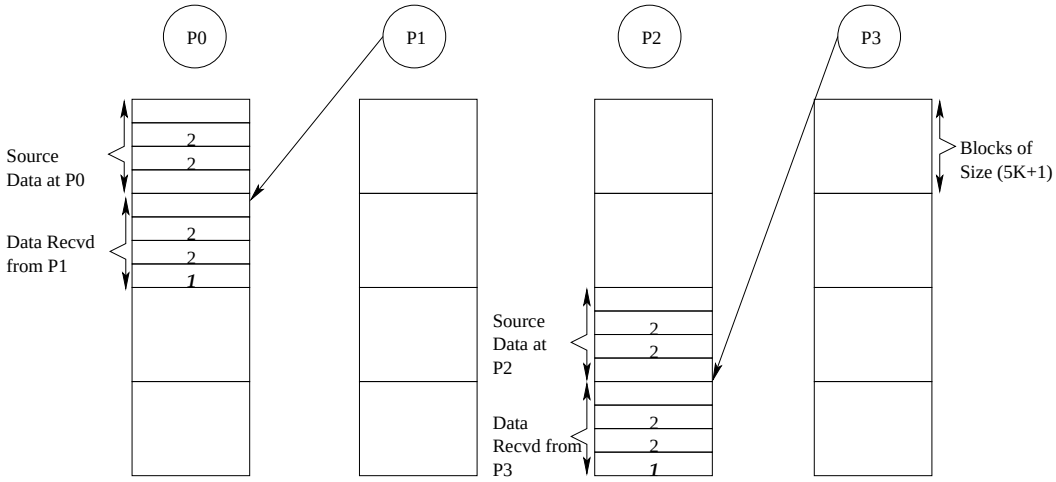

Figure 10. Step 1 of Degree-1 RDMA-based allreduce

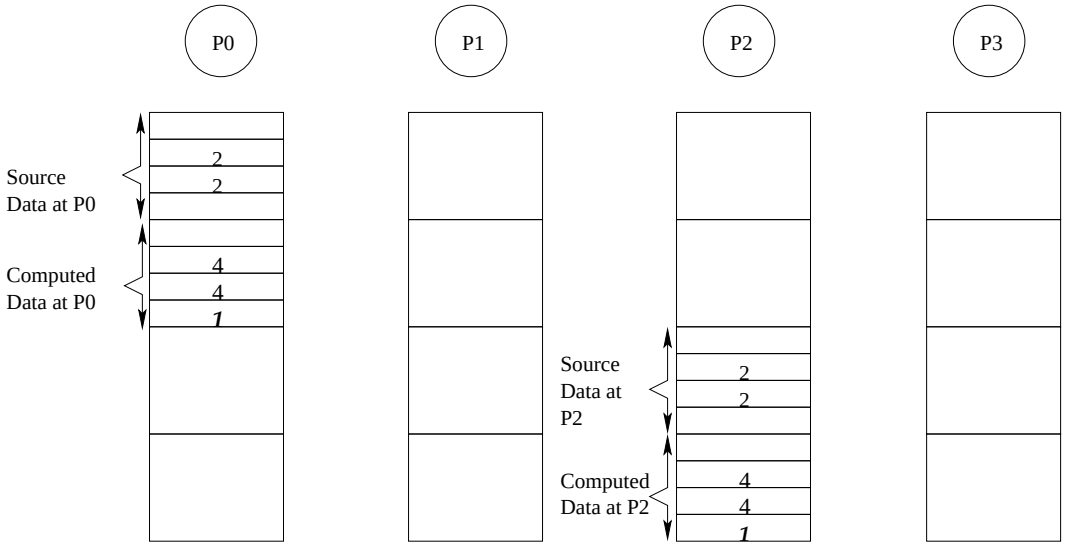

Figure 11. Reduce Computation at P1 and P2

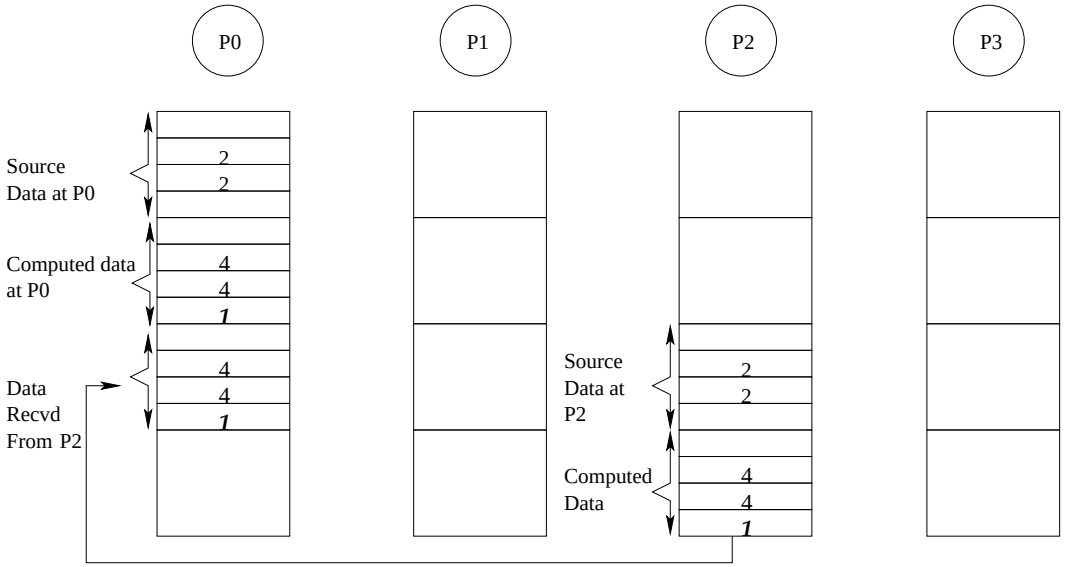

Figure 12. Step 2 of Degree-1 RDMA-based allreduce

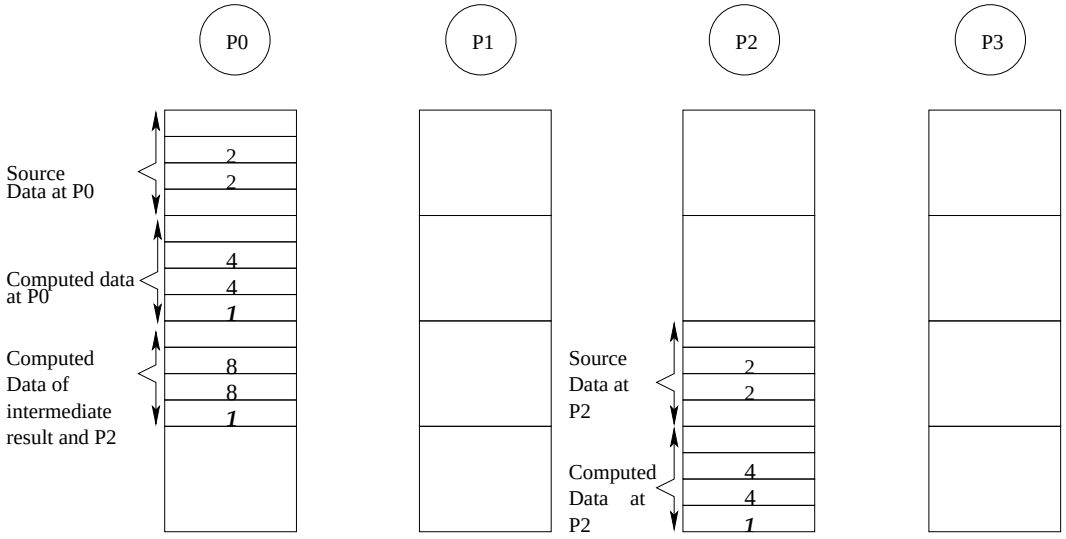

Figure 13. Final reduce Computation at P0

sends an acknowledgment for each send for reliable reception mode. The second block of data can only be sent after the acknowledgment has been received. Thus, for an *N* (where *N* is a power of 2) node system, we will have the total time as : *2*Tc + log(N)*STEP2*.

In general, the total broadcast time for *N* nodes, where the message is broken into *num* blocks, can be represented by the equation : *Tc1 + (Tm + Td + 2*(Ts+Tn) + (num - 1)*Tt1 + (num*Ta) + Tt2)*log(N)+ Tc2*. *Tc1* is the copying time for the first block (of size *block_size*) and *Tc2* is the copying time for the last block, which may have a size lesser than *block_size*. *Tt1* is the transmit time for all blocks but the last block. *Tt2* is the transmit time for the last block. Approximately, considering all blocks to be of the same size, the equation can be given by : *2*Tc + (Tm + Td + 2*(Ts + Tn) + num*(Tt + Ta))*log(N)*.

6.2 Analytical model for RDMA-based allreduce

For a given set of *power of 2* nodes, we have different *degree-k* algorithms available. In this section, we present an analytical model for Degree-k RDMA-based allreduce, which enables us to choose the optimal value of k for a given number of nodes and data size on the basis of the parameters discussed in the previous section.

In addition to the parameters given in the previous subsection, we also have to take into account the time spent in performing the reduction operation on the given data size. This total time can be indicated by *To*. The assumptions made in the previous section hold true for allreduce too. Typically for large messages *To* is much lesser as compared to *Tt*. However this is not true for small messages having small counts. A message transfer in allreduce operation consists of similar events as described in the previous section. The only difference being that once the data is DMAed by the destination NIC, the reduction operation is performed. The result can then be broadcast to all the nodes in the communicating group.

The analytical model has various cases based on the values of the above parameters. In the following subsections we present the analytical equations with the time diagrams for all these cases. The cases outlined below give the analytical equations for the Degree-k RDMA-based Reduce. The analytical model presented in the previous subsection for the broadcast operation can be combined with the analytical model in this section for the Degree-k RDMA-based reduce to give the overall analytical model for the degree-k RDMA-based allreduce operation.

For all the examples, we consider a Degree-3 RDMA-based allreduce scheme in a 4 node cluster. Consider nodes P0, P1, P2 and P3 with *ids 0, 1, 2, 3* respectively. The Degree-k reduce part of the allreduce operation takes place as shown in Figure 8. It involves one step with P1, P2, P3 sending to P0 and all the operations take place at P0.

In all the following sections when we refer to the Degree-k allreduce analytical model, we implicitly assume that it is the sequentialized combination of the degree-k reduce analytical model and the broadcast analytical model.

6.2.1 Handling Large Messages

Messages where *Tt > (Tn + Ts)*, are categorized as large messages. In such messages, the transmit time dominates all the other parameters.

For the degree-k RDMA-based allreduce case, the receiver performs reduction operation based on the order of *ids*, starting with the data sent by the node with the lowest *id*. Consider P1, P2 and P3 with *ids 1, 2 and 3* respectively are sending the data to node P0. The receiver P0 polls for the data from the node with *id 1*, i.e. P1 to arrive first and operates on this data. When many nodes are performing an RDMA write to a single destination simultaneously, the order in which the messages will arrive at the destination NIC and hence in the destination buffer is indeterminate. Thus, if a NIC is waiting for allreduce data to arrive from the lowest *id* node, there is a fair probability that the data might not arrive first. Hence, the analytical model for RDMA allreduce gives the best and the worst time estimates.

The best time estimate assumes that the required data is the first to arrive, the DMA startup and the NIC processing can be overlapped with each other. The worst time estimate assumes that the required data is the last to arrive. It also assumes that the NIC processing and the DMA startup can't be overlapped due to sharing to the system bus.

To understand this concept, consider the Degree-3 RDMA-based allreduce scheme in a 4 node cluster, where P1, P2 and P3 write to P0. Figure 16 shows the time line chart for the events that happen at the sender side. Since P1, P2 and P3 send data at the same time, the copying of data, MPI overhead, posting of descriptor, DMA startup and NIC processing for the various processes gets overlapped at the sender side. During this period, the Process P0 is polling for data from Process P1, which has the *id 1*, to arrive. The sender side cost can be termed by the parameter *T_sender* given by : *T_sender = Tm + Td + Tn + Ts*. The total sender cost : *Total_sender_cost = Tc + T_sender*.

In every step of the degree-k RDMA-based allreduce algorithm, every possible destination can receive from *at most* k nodes. A node can receive from less than k nodes in the last step of the allreduce operation. The number of nodes sending the data to the destination in each step plays an important role in the analytical model.

Figure 17 shows the best case scenario at the receiver end.

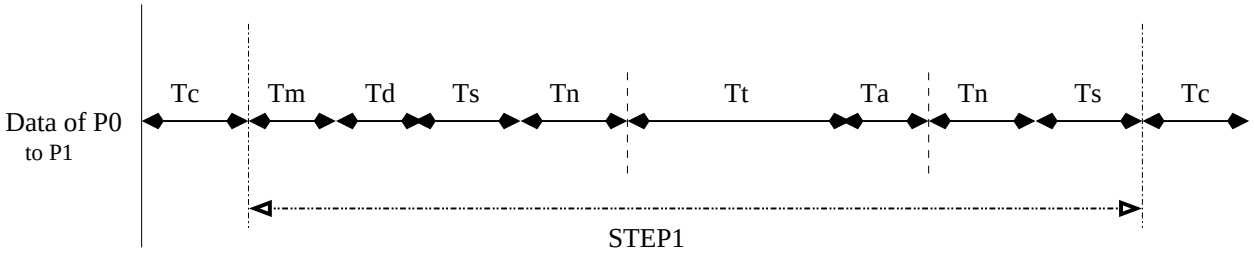

Figure 14. RDMA-based broadcast with 1 block

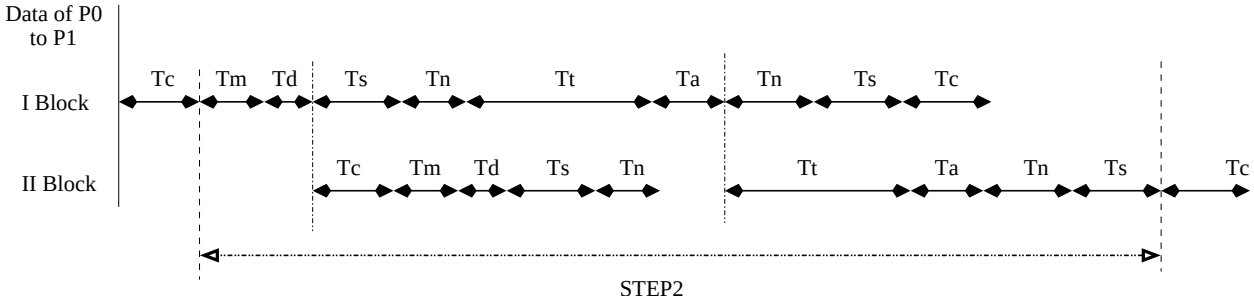

Figure 15. RDMA-based broadcast with 2 blocks

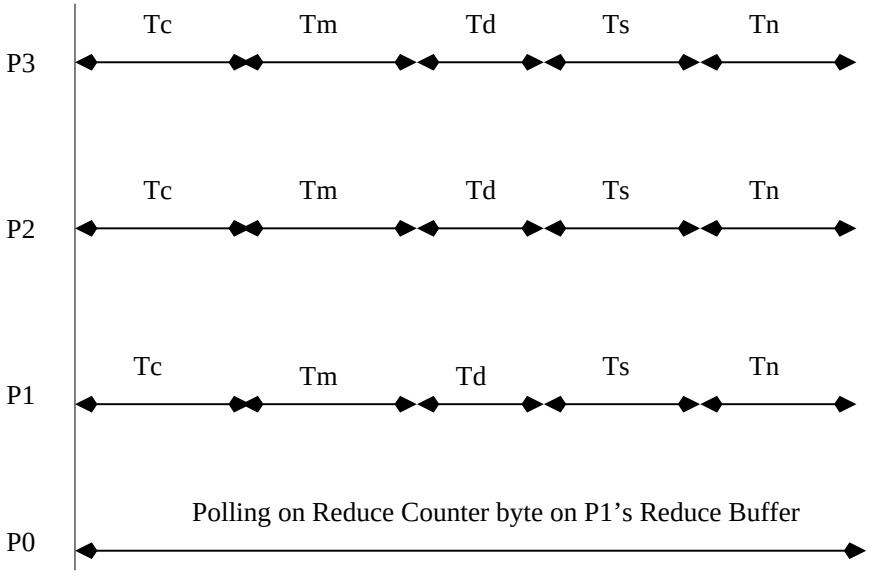

Figure 16. Sending side events

The message from P1 arrives first to the switch. When this message reaches node P0, it does the NIC level processing, DMAing and starts the reduction operation. The transmission of the second message happens in parallel with the NIC processing, DMAing and operation of the first message. By the time the destination NIC receives the next message, it has already finished processing the first one. By the time the host process receives the second message, it has finished the operation on the first one. For large messages, the *To* parameter is very small as compared to the *Tt* parameter. Also, data has to be copied from the user-specified source user buffer to the allreduce registered buffer. Hence, there is a copying cost involved. The time taken at the receiver end in each step for this best case is : *Tt*(No. of Sending Nodes in that step) + Tn + Ts + To*

The overall best case equation for very large messages is given as : *2*Tc + (Total Number of Steps)*[2*(Ts + Tn) + Tm + Td + Tt*(No. of Sending Nodes in that step) + To]*, which includes the events at both the sender and the receiver ends. The *Total Number of Steps* variable specifies the total number of steps that will be required to complete the allreduce operation in that cluster.

The worst case is depicted in Figure 18 where the message from P1 reaches last. P0 can start performing the operations only after receiving data from P1. Hence, the time at the receiver end for each step is given as : *Ts + Tn + (Tt + To)*(No. of Sending Nodes in that step)*

The total time which includes the sender and the receiver time is given by : *2*Tc + (Total number of Steps)*[2*(Ts + Tn) + Tm + Td + (Tt+To)*(No. of Sending Nodes in that step)]*

6.2.2 Handling Small Messages

Messages where *Tt < (Tn + Ts)* are categorized as small messages. The events at the sender side remain the same as shown in Figure 16. The events at the receiver are shown in Figures 19 to 21. At the receiver side, the messages from various nodes destined for one node are still sequentialized, however since the amount of data to be transmitted is less, hence the transmit time is very less. When the transmit time is less, the operation time *To* for such messages is also very less. In many cases, the NIC processing and the DMA startup cost will be the major contributors of the overhead.

The evaluation of the best cases for small messages are divided into two sections. We consider that case first where *Tt <= Ts* and this best case is shown in Figure 19. Here we assume that the NIC processing can be done in parallel with the DMA startup and that the required data is always obtained first. Hence, the message from P1 is the first to reach P0. When P0's NIC is processing the message, DMAing it and performing the operation, the message from P2 can be transmitted and processed. The time taken at the receiver end by each step for the best case scenario for this case is given by : *Tn + Tt + Ts*(No. of Sending Nodes in that step) + To*

The total time including the sender and the receiver is given by : *2*Tc + (Total number of Steps)*[Tm + Td + 2*Tn + Tt + Ts*(No. of Sending Nodes in that step + 1) + To]*

The second case is when *Tt > Ts*, as shown in Figure 20. The message from node P1 reaches node P0 first. The transmit time and the NIC processing of the second message is overlapped with the first one. However, as *Tt > Ts*, hence the *Ts* time for the second message is delayed till the *Ts* of the first message has been completed. The duration of this delay is given as the *Tt - Ts* time.

Hence, the best estimate at the receiver end for each step is given by : *(Tt*No. of Sending nodes in that step) + Tn + Ts + To*

The total time for allreduce for the best case scenario for *Tt > Ts* is : *2*Tc + (Total number of Steps)*[Tm + Td + 2*(Tn + Ts) + (Tt*No.of Sending nodes in that step) + To]*

If the NIC processing and the DMA startup can't be overlapped, then we have a worse case scenario as shown in Figure 21. Here, we also assume that the required data is the last to arrive. Hence the data sent by Process P3 reaches first and is processed by the NIC and DMAed. Meanwhile, the message from node P2 arrives but it can't be processed because the NIC is busy processing and DMAing the first message received from node P3. So the NIC processing and DMAing are sequentialized for each message with no overlap happening. When the data from node P1 arrives, the first operation is done.

Thus the worst case scenario at the receiver end for each step is : *Tt + (Tn + Ts + To)*(No. of Sending Nodes in that step)*.

The total time for the worst case scenario for small messages is : *2*Tc + (Total number of steps)*[Tm + Td + Tt + (Tn + Ts)*(No. of Sending Nodes in that step + 1) + To*(No. of Sending Nodes in that step)]*

The pseudo code for getting the values analytically for the various Degree-k RDMA-based allreduce algorithms is shown in Figure 22.

The variable k stands for Degree-k RDMA-based allreduce value,i.e., the maximum number of nodes from whom a node can receive data in one step. The function calculates the best and the worst time estimates based on the various parameters provided. The first *while loop* iterates for the total number of complete steps, where *complete steps* are defined as steps where a destination in that step receives

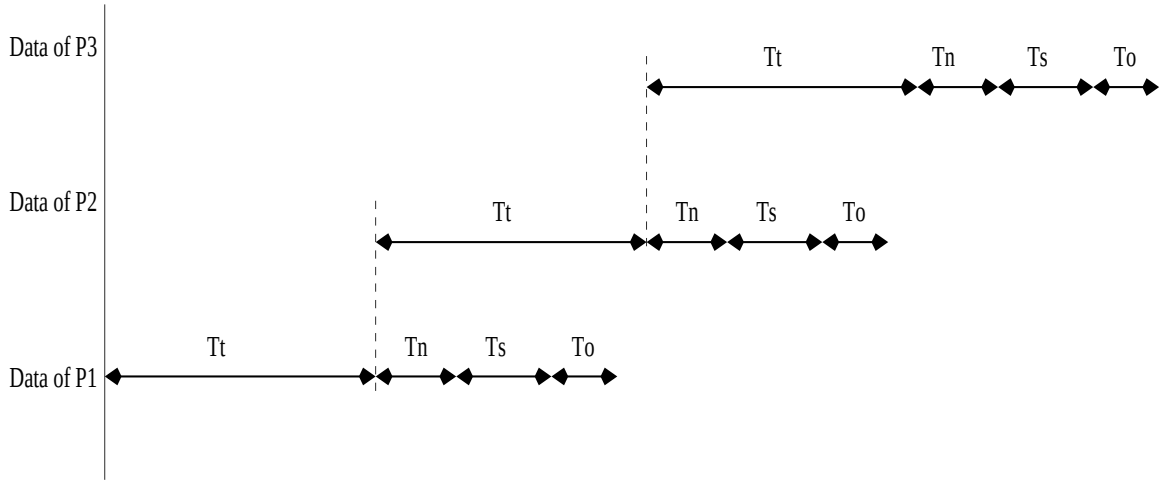

Figure 17. Best case receiver scenario for large messages

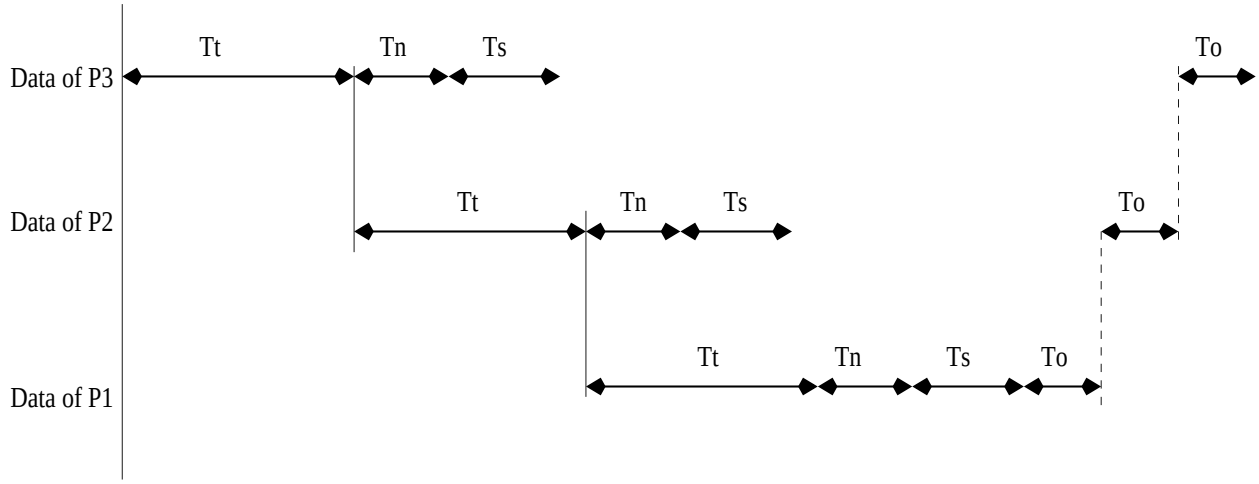

Figure 18. Worst case receiver scenario for large messages

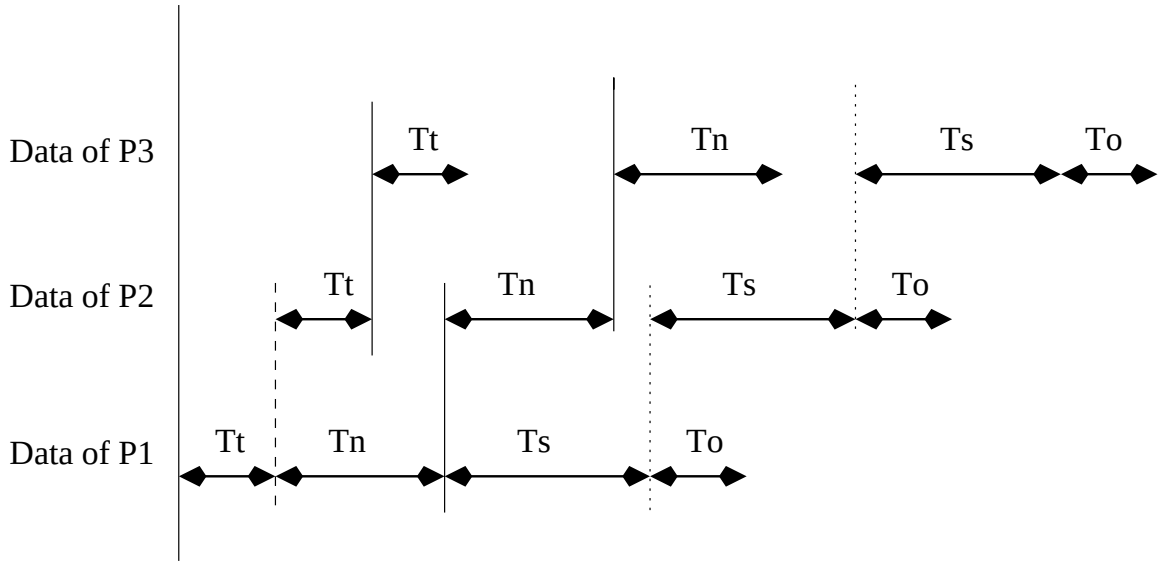

Figure 19. Best case receiver scenario for small messages where *Tt <= Ts*

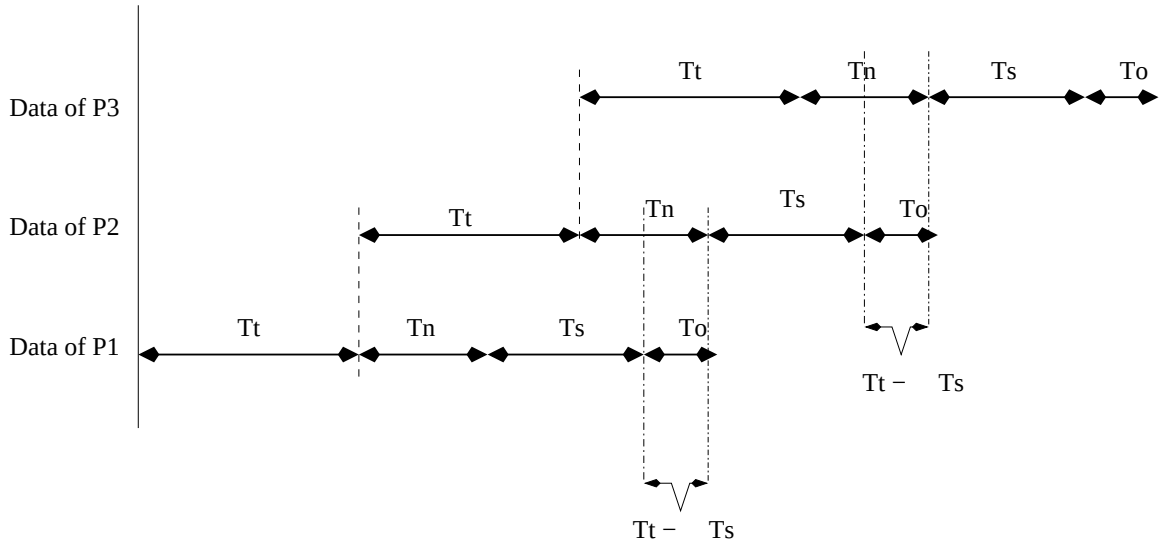

Figure 20. Best case receiver scenario for small messages where *Tt > Ts*

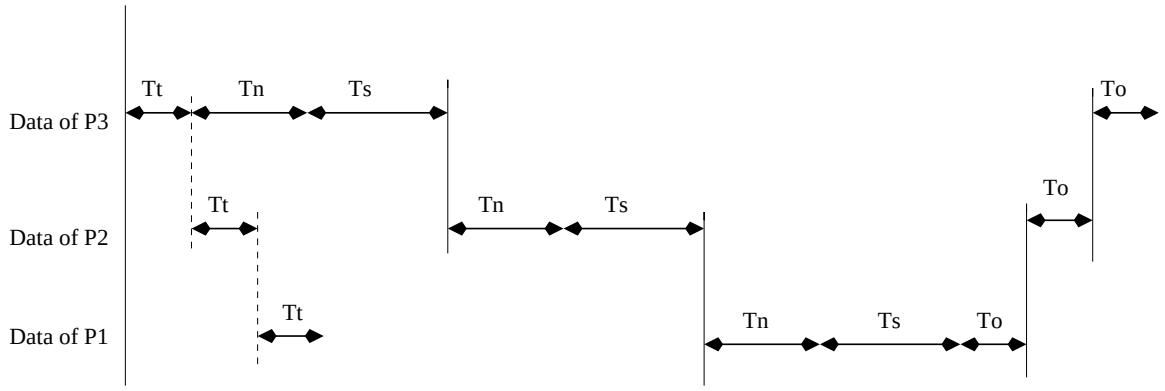

Figure 21. Worst case receiver scenario for small messages

from exactly k nodes in an Degree-k RDMA-based allreduce algorithm. The second part of the code is called a destination node receives data from less that k nodes. We add the *bcast_time*, which is the time taken for a binomial broadcast to complete for the given set of nodes involved in the allreduce operation, in the final worst and best time estimates of the allreduce analytical model.

This function works for *power of 2* clusters, i.e. when the variable *Nodes* is a *power of 2* and we are testing for a Degree-k tree-based RDMA allreduce where the value of k is a *power of two - 1* and $k < Nodes$. It gives us the best and the worst estimates between which the actual values might lie.

7 Performance Evaluation

In this section, we discuss the results that have been obtained for RDMA-based broadcast and RDMA-based allreduce. We evaluated our implementations on a cluster of 16 nodes, each with a 33MHz PCI bus, 1.0GHz Pentium III machines, 512MB of Main memory and Linux version 2.2.17. The machines are connected using a GigaNet 5300 switch. In addition to the experimental results, we also present results for larger systems using the analytical model. For the analytical model, we use the following values for the system dependent parameters of RDMA-based broadcast and allreduce. We set *Tt_per_byte=0.010 us, Ts=2us, Tc_copy_rate=0.0027us, Tn=1.52us, Td=0.6us, Tm=1.3us. Tt* and *To* are calculated depending on the total number of *bytes* and *count* of the operation.

7.1 RDMA-based Broadcast Performance

The RDMA-based binomial broadcast is compared with the send/receive-based binomial broadcast, provided by the MVICH-1.0 implementation of MPI. The broadcast latency, averaged over 5000 iterations, is calculated between the root node and the last leaf node receiving the message. The last leaf is typically chosen to be the one having the maximum depth from the root in the binomial tree structure.

7.1.1 RDMA-based broadcast v/s Send/Receive-based broadcast on small clusters

Using the RDMA scheme, a large message is broken into multiple blocks depending upon the *block_size*. MVICH-1.0, in the send/receive-based binomial broadcast, sends data in blocks of *1KB*. To ensure fair comparison, we tested the RDMA-based binomial broadcast with different block_sizes starting with 1025 (1 extra byte for the counter) bytes.

Figure 23 shows a comparison of RDMA-based broadcast with *block_sizes* of 1025, 2049, 3073 and 4097 bytes and the send/receive-based broadcast for small messages of 4-1024 bytes for a 16 node cluster. Small messages from 4-1024 bytes using RDMA-based broadcast show the same timings because all the messages use 1 block to write to the remote node as the least *block_size* is 1025 bytes. The difference can be seen in Figure 24. We see that to transmit 4096 bytes with *block_size* of 1025 bytes, we need 4 blocks. For higher *block_sizes*, the number of sends decreases and so do the timings. We obtain the best result for a *block_size* of 3073 bytes. In fact, a *block_size* of 3073 bytes gives the most optimal result for all message sizes from 1025 to 5000 bytes. RDMA-based broadcast for all the given *block_sizes* gives better performance as compared to the message send/receive-based binomial broadcast.

With RDMA-based broadcast with *block_size* of 3073 bytes, we get a benefit of around 19.7% for small messages of 4 bytes. For large messages we see a benefit of around 14.4% for 4608 bytes.

We also use the broadcast analytical model, presented in the previous section, to estimate the timings for RDMA-based binomial broadcast with varying block sizes. Fig-

```
Quotient      = Nodes / ( k + 1 );
T_transmit  =  bytes  *  T_transmit _per_byte;
Best_time    =  Worst_time  =  0;

while ( Quotient  !=  0 ) {
      Sender_side = T_mpi + T_descriptor + T_startup + T_nic
      If  ( T_transmit >= ( T_startup  + T_nic ) ) then
            Receiver_side  = (  T_transmit  *  Nodes  )  +  T_startup  +  T_nic  +  T_operation
            Best_time    +=  Sender_side  +  Receiver_side ;
            Worst_time +=  Sender_side  +  Receiver_side  + T_operation * ( k  − 1 )
    else
            If  ( T_transmit <  T_startup  ) then
                  Receiver_side = T_transmit +  T_nic + (T_startup * k) + T_operation
            else
                  Receiver_side = T_transmit +  T_nic + (T_startup * k) + (T_transmit − T_startup) *
                                        (k−1)  + T_operation
          Best_time    +=  Sender_side + Receiver_side
          Worst_time +=  Sender_side + Receiver_side +  (T_nic + T_operation ) * (k − 1)
          Endif
      Endif
      Dividend = Quotient ;
      Quotient = Dividend/(k+1);
      Remainder = Dividend % (k+1)
End− while

If ( Remainder != 1 ) then
      Sender_side  = T_mpi + T_descriptor + T_startup + T_nic
      If  (  T_transmit  >=  ( T_startup + T_nic ) ) then
            Receiver_side  = ( T_transmit  * ( Remainder  −  1 )  ) + T_startup + T_nic + T_operation
            Best_time    +=  Sender_side  +  Receiver_side
            Worst_time  +=  Sender_side  +  Receiver_side  +  T_operation  *  ( Remainder  −  2 )
      else
          If  ( T_transmit <  T_startup  ) then
                Receiver_side = T_transmit +  T_nic + (T_startup * (Remainder −1)) + T_operation
          else
                Receiver_side = T_transmit +  T_nic + (T_startup * (Remainder −1 )) +
                                    + ( ( T_transmit − T_startup) * (Remainder −2 ) )  + T_operation
            Best_time    +=  Sender_side + Receiver_side ;
            Worst_time +=  Sender_side + Receiver_side +  ( T_nic + T_operation) * (Remainder − 2)
          Endif
      Endif
Endif
Worst Time += ( bytes * T_copy_rate ) * 2 + bcast_time
Best_Time   += ( bytes * T_copy_rate ) * 2 + bcast_time
```

Figure 22. Pseudo code for Optimal All Reduce Algorithm

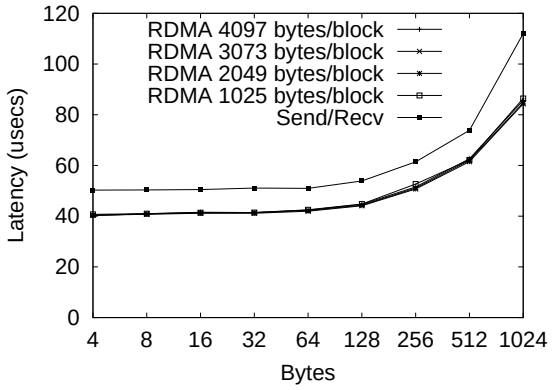

Figure 23. RDMA-based vs Send/Receive-based broadcast comparison (4-1024 bytes)-16 node cluster

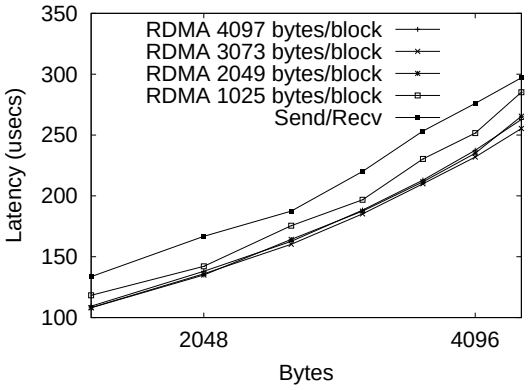

Figure 24. RDMA-based vs Send/Receive-based broadcast comparison (1025-4608 bytes)-16 node cluster

ure 25 shows the comparison of the analytically obtained RDMA-based broadcast with the experimentally obtained RDMA-based broadcast using *block_size* of 3073 bytes for a 16 node cluster. For all data sizes (4-4096 bytes), the analytically obtained results closely match the experimental ones with an error rate of below 10%.

We also estimated the performance of RDMA-based broadcast on 2,4 and 8 node cluster of the above configuration. The analytical results were found to closely match the experimental ones.

7.1.2 RDMA-based broadcast V/s Send/Receive-based broadcast on large clusters

We use this analytical mode to estimate the performance of RDMA-based binomial broadcast on 512 and 1024 node systems. We compare the estimated RDMA-based broadcast timings with send/receive-based broadcast for the same cluster size. Extrapolating the timings obtained for broadcast between a pair of nodes, we obtain the latency for the send/receive-based broadcast for large clusters of size N to be approximately *log(N)*t1*, where *t1* is the time for send/receive-based broadcast between 2 nodes.

Figure 26 shows the graphs for RDMA-based broadcast and send/receive-based broadcast for 512 node system for data size 4-4608 bytes. The RDMA-based broadcast timings have been taken with an optimal block size of 3073 bytes. We achieve a benefit of about 21% for *4KB* messages. For small messages of 4 bytes, we get a performance benefit of about 16%.

Figure 27 shows the analytical graphs for the RDMA-based broadcast and send/receive-based broadcast comparison for 1024 node system with optimal *block_size* of 3073 bytes for data size 4-4096 bytes. Here again, for message size of *4KB*, we obtain a performance benefit of about 21% and for small messages a benefit of 16%.

7.2 RDMA-based Allreduce Performance

The timings were obtained by running 5000 iterations of allreduce and taking the average of the iterations over all nodes. The operation used was MPI_SUM with the data type MPI_INT of size 4 bytes and the count of the elements was varied from 1 (4 bytes) to 1024 (4096 bytes).

7.2.1 RDMA-based allreduce V/s Send/Receive allreduce for small clusters

We presented the analytical model for degree-k RDMA-based allreduce in Section 6. Depending upon the network topology, the analytical model gives the best and the worst time estimates for the allreduce operation. In this section we present the analytical results for RDMA-based allreduce for a 16 node cluster for data size (4-4096) bytes. We show the graphs for degree-15, degree-7, degree-3 and degree-1. We also compare various analytical and experimental results.

Figure 28 and 29 show the worst and best analytical and the actual practical timings for Degree-15 RDMA-based allreduce for 16 nodes. We see that for all data size the actual timings lie between the best and the worst case estimated timing. The greatest error rate is around 7.8% for 512 bytes. All other error rates, if any, lie below 2%. Figures 30 and 31 show the timings for the same configuration but now with a Degree-7 RDMA-based allreduce. We see a slight

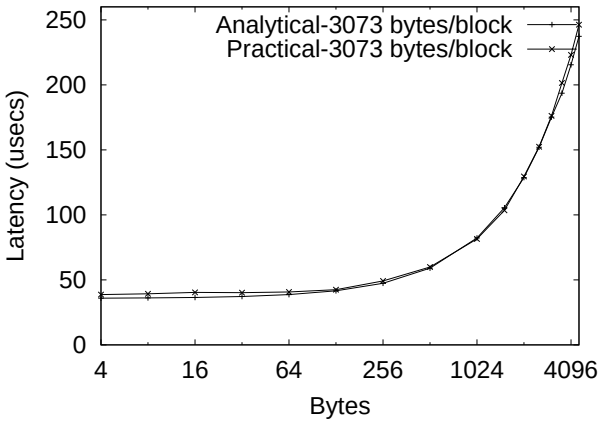

Figure 25. RDMA-based binomial broadcast Analytical and Experimental comparison (4-4608 bytes)-16 node system

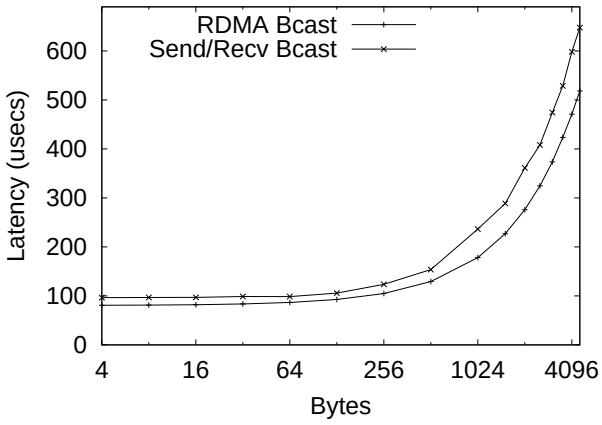

Figure 26. RDMA-based and Send/Receive-based binomial broadcast estimations for (4-4608 bytes)-512 node cluster

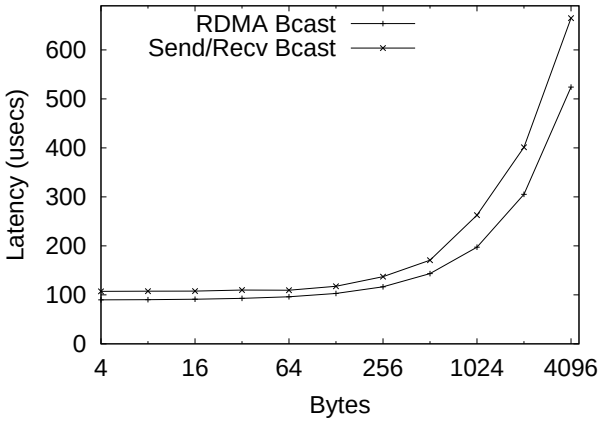

Figure 27. RDMA-based and Send/Receive-based broadcast estimations for (4-4096 bytes)-1024 node cluster

deviation of the actual values from the best and the worst cases, for small data size. However, the error rate of deviation from estimation is still below 3%. Figures 32 and 33 show the best and the worst case analytical and the experimental timings for the Degree-3 RDMA-based allreduce. We see that the actual experimental timings lie between the best and the worst analytical range. The maximum error rate for Degree-3 RDMA-based allreduce is about 4.3% for 4 bytes. Figures 34 and 35 show the timings for Degree-1 RDMA-based allreduce. The maximum error rate was around for Degree-1 RDMA-based allreduce was 6.21% for 4 bytes.

A comparison of the best and worst different degree-k allreduce algorithms for 16 nodes with small data sizes (4 bytes) using the analytical model gives Degree-3 RDMA-based allreduce scheme as the optimal algorithm which we verified practically. For large messages, the analytical model clearly shows Degree-1 RDMA-based allreduce to be the most optimal one.

We also evaluated the analytical model on 4 and 8 nodes of the same cluster. We summarize the results in Figure 40. For a cluster of 4 nodes, the Degree-3 RDMA-based allreduce algorithm performs well for small messages till 1024 bytes. The Degree-3 RDMA-based allreduce gives good performance for small messages in the 16 node and also in the 8 node cluster. But in the 8 node cluster, we obtain a slightly improved performance if we use Degree-7 RDMA-based allreduce for very small messages. For large messages above 1024 bytes, the Degree-1 RDMA-based allreduce always gives the best performance. This is because in the *degree-3* RDMA-based allreduce case, 3 nodes write to 1 node and 3 operations are done at the receiving node. As the data size increases, the number of operations increase and computation becomes very expensive. The *degree-1* RDMA-based allreduce fares better as the computation is distributed to a greater number of the nodes. We verified both sets of results. Given here are the results for a 16 node and a 8 node cluster.

For a 16 nodes cluster, we implemented Degree-1, Degree-3, Degree-7 and Degree-15 RDMA-based allreduce schemes. Figures 36 and 37 give the comparison of the experimental evaluation of the all the degree-k RDMA-based allreduce algorithms possible for the 16 node cluster for various data sizes. As predicted by the analytical model, the Degree-3 RDMA-based allreduce gives the best performance for small message up to *1*KB. Beyond *1*KB, Degree-1 gives the best performance. Figures 38 and 39 give the comparison of the experimental evaluation of all degree-k RDMA-based allreduce algorithms for an 8 node cluster. These results were again found to be similar to the analytical results that were summarized.

Figure 41 compares the results of send/receive-based binomial allreduce, most optimal *degree-k* RDMA-based allreduce and *degree-1* RDMA-based allreduce. The most optimal *degree-k* RDMA-based allreduce uses *degree-3* algorithm for small messages and *degree-1* algorithm for messages greater than *1KB*. The *degree-1* allreduce is exactly the same as the binomial allreduce algorithm. We see that the RDMA-based binomial algorithm (i.e *degree-1*) always performs better than the send/receive-based binomial algorithm. The *degree-3* RDMA-based algorithm outperforms both the send/receive-based and RDMA-based binomial allreduce algorithms for small messages (4-1024 bytes). On a 16 node cluster, we obtain a 38% benefit for 4 byte messages, when we use the *degree-3* RDMA-based allreduce. For large messages of size *4KB*, we get a 9% improvement on using the optimal *degree-1* RDMA-based allreduce scheme. The benefits obtained are due to an optimal algorithm implemented with an efficient RDMA mechanism.

	4−256 Bytes	256−1024 Bytes	>1024 Bytes
1024 nodes	Degree−3	Degree−3	Degree−1
512 nodes	Degree−3	Degree−3	Degree−1
16 nodes	Degree−3	Degree−3	Degree−1
8 nodes	Degree−7	Degree−3	Degree−1
4 nodes	Degree−3	Degree−3	Degree−1

Figure 40. Choosing the Right algorithm for varying configurations

7.2.2 RDMA-based allreduce V/s Send/Receive-based allreduce for large clusters

We evaluated the analytical model for clusters of 512 and 1024 nodes. The degree-k RDMA-based allreduce analytical model gives the best and the worst case allreduce latency.

An analysis of the best and worst case timings shows that *degree-3* RDMA-based allreduce performs optimally for small messages (4-1024 bytes) and *degree-1* RDMA-based allreduce performs the best for large messages (1025-4096 bytes) (Figure 40). Thus we can generalize by saying that a *degree-3* RDMA-based allreduce algorithm should give good performance for small data size (from 4 to 1024 bytes) and a *degree-1* RDMA-based allreduce scheme can be used

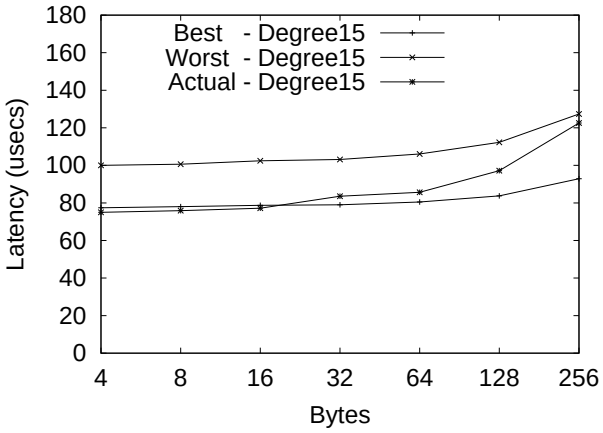

Figure 28. RDMA-based allreduce Analytical and Experimental Comparison for 16 node system: Degree-15 (4-256) bytes

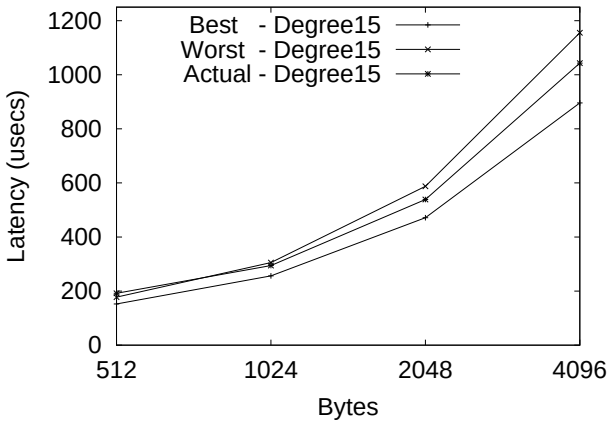

Figure 29. RDMA-based allreduce Analytical and Experimental Comparison for 16 node system: Degree-15 (512-4096) bytes

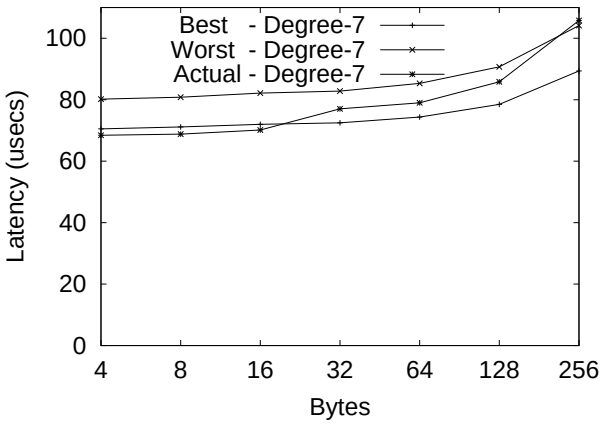

Figure 30. RDMA-based allreduce Analytical and Experimental Comparison for 16 node system: Degree-7 (4-256) bytes

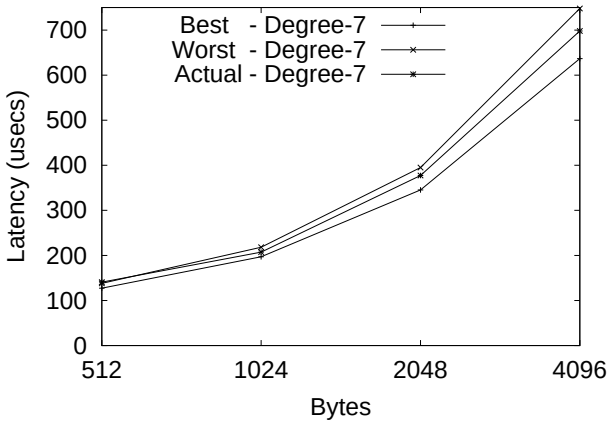

Figure 31. RDMA-based allreduce Analytical and Experimental Comparison for 16 node system: Degree-7 (512-4096) bytes

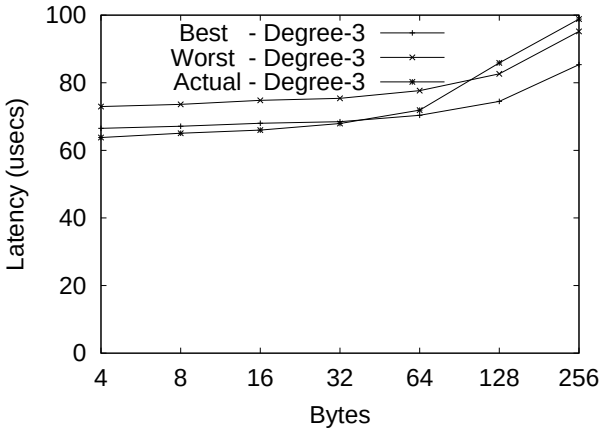

Figure 32. RDMA-based allreduce Analytical and Experimental Comparison for 16 node system: Degree-3 (4-256) bytes

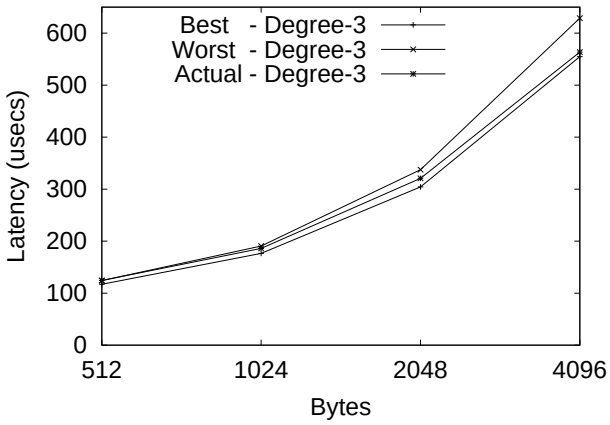

Figure 33. RDMA-based allreduce Analytical and Experimental Comparison for 16 node system: Degree-3 (512-4096) bytes

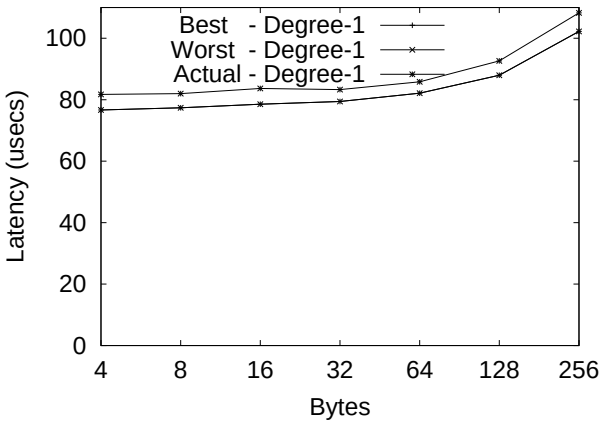

Figure 34. RDMA-based allreduce Analytical and Experimental Comparison for 16 node system: Degree-1 (4-256) bytes

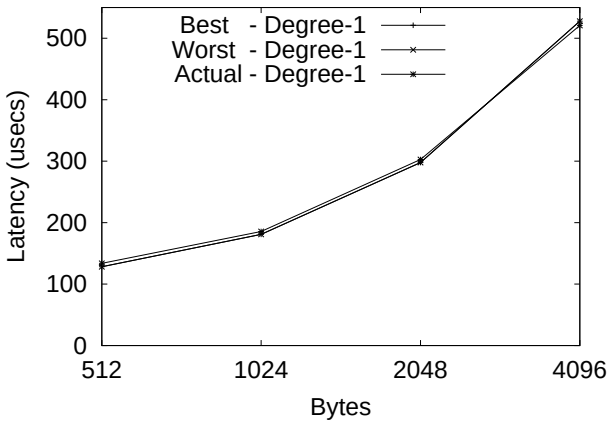

Figure 35. RDMA-based allreduce Analytical and Experimental Comparison for 16 node system: Degree-1 (512-4096) bytes

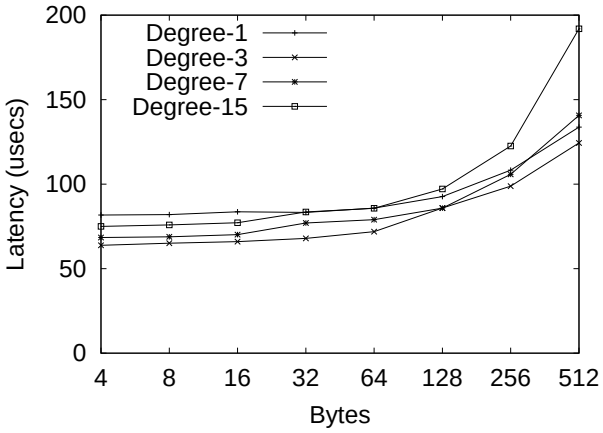

Figure 36. Degree-k RDMA-based allreduce Performance comparison for (4-1024 bytes)-16 node cluster

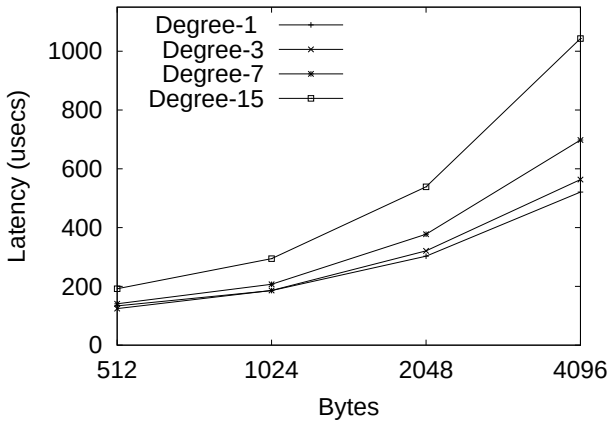

Figure 37. Degree-k RDMA-based allreduce Performance comparison for (1024-4096 bytes)-16 node cluster

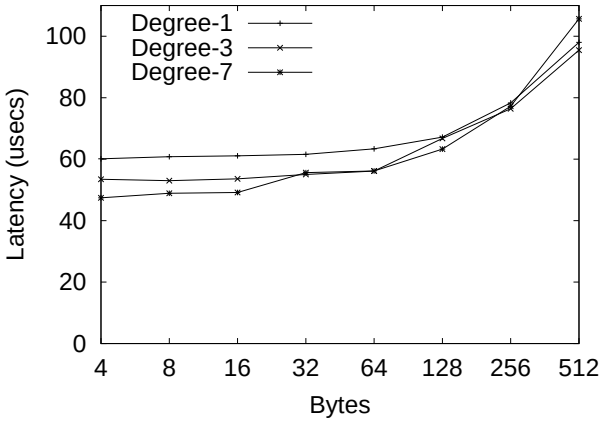

Figure 38. Degree-k RDMA-based allreduce Performance comparison for (4-1024 bytes)-8 node cluster

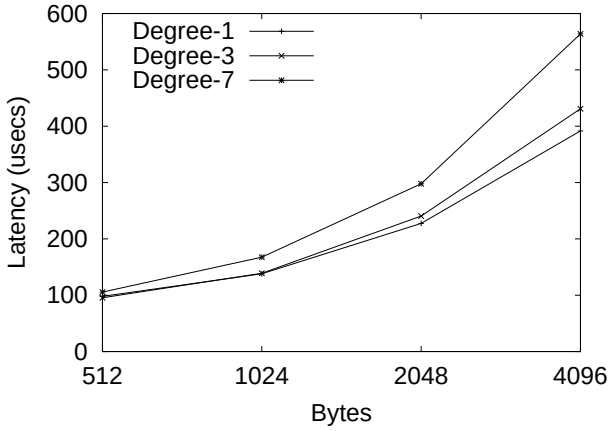

Figure 39. Degree-k RDMA-based allreduce Performance comparison for (1024-4096 bytes)-8 node cluster

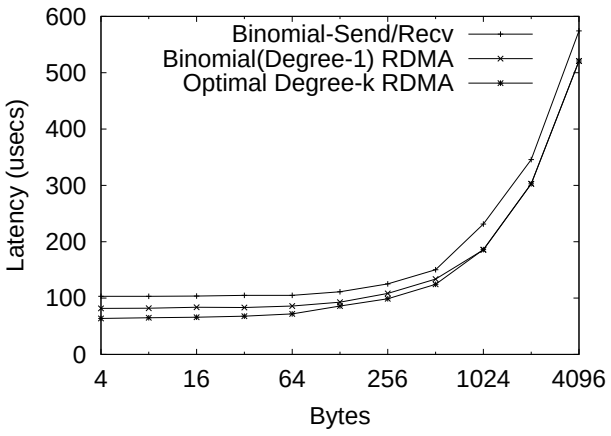

Figure 41. Send/Receive-based binomial, RDMA-based binomial and the most optimal degree-k RDMA-based allreduce comparison - 16 node Cluster

for large data sizes while implementing the reduce part of the allreduce collective operation.

We also use the RDMA-based allreduce analytical model to predict the performance achievable in large clusters. We obtain the send/receive-based binomial reduce latency by extrapolating the reduce latency between 2 nodes. If the time for reduce between 2 nodes is *t1*, then the time taken for the same reduce between N nodes, which involves $log(N)$ steps is approximately *log(N) * t1*. The timings for the send/receive-based allreduce can be obtained by adding the reduce and the broadcast timings.

Figure 42 shows the comparison between the estimated send/receive-based binomial, the best and the worst case of the most optimal *degree-k* RDMA-based allreduce and the *degree-1* (binomial) RDMA-based allreduce. We use the most optimal *degree-3* algorithm for message sizes lesser than *1KB* and *degree-1* algorithm for messages greater than *1KB* in the *degree-k* RDMA-based allreduce. The analytical models predicts a 14% performance benefit of the best case of the most optimal *degree-k* RDMA-based allreduce for message size of *4KB* and around 40% for small messages of 4 bytes as compared with the send/receive-based binomial allreduce. When send/receive-based binomial allreduce is compared with the worst case of the most optimal *degree-k* RDMA-based allreduce, we still obtain a benefit of about 35% for 4 byte messages and 14% for *4KB* messages. The *degree-3* RDMA-based allreduce outperforms the RDMA-based binomial as well the send/receive-based binomial algorithm for small messages. Similar timings are obtained for 1024 nodes (Figure 43). The analytical model predicts a benefit of 41% for 4 bytes and 14% for *4KB*.

8 Conclusions and Future work

Traditionally, collective operations have been implemented on the send/receive message passing primitives. In this paper, we introduce a novel method to implement fast broadcast and allreduce communication operations on VIA based clusters using RDMA operations. We analyzed the different design issues that are a part of such an implementation focusing mainly on buffer registration, safe reuse of buffers and indication of data validity at the receiver end. We implemented *RDMA-based broadcast* using the binomial algorithm which gives a 14% benefit for a 16 node cluster for 4608 bytes and 19% for 4 byte messages as compared to the send/receive-binomial broadcast algorithm.

For the *allreduce* operation, we introduced a new concept of *degree-k tree-based* allreduce algorithms which when combined with the RDMA mechanism gives improved performance as compared to the send/receive-based algorithms. A comparison of the most optimal degree-k RDMA-based allreduce with the send/receive-based binomial allreduce gives us a benefit of about 38% benefit for a small data size of 4 bytes and about 9% for data size of *4KB* for a 16 node cluster. We also presented analytical models for broadcast and allreduce that give an estimate of the broadcast time for large clusters and the most optimal *degree-k RDMA-based* allreduce algorithm that can be used for a given cluster and data size. We have used the analytical model to predict the performance benefits achievable by our RDMA implementation of allreduce on large clusters. The predicted performance shows a benefit of about 35-40% for small messages of 4 bytes and around 14% for messages of *4KB* for 512 and 1024 node clusters.

In future, we plan to perform in-depth analysis of the global buffer management for other collective operations. We plan to explore and develop efficient algorithms dealing with user-defined communicators used in context with these collective operations. We also plan to extend this framework to the emerging InfiniBand architecture.

References

[1] Infiniband Trade Association. http://www.infinibandta.org.

[2] M. Banikazemi, V. Moorthy, L. Hereger, D. K. Panda, and B. Abali. Efficient Virtual Interface Architecture Support for IBM SP switch-connected NT clusters. In *the Proceedings of the International Parallel and Distributed Processing Symposium, pages 33-42*, May 2000.

[3] P. Buonadonna, A. Geweke, and D. E. Culler. BVIA: An Implementation and Analysis of Virtual Interface Architecture. In *the Proceedings of Supercomputing '98*, 1998.

[4] W. Gropp and E. Lusk. MPICH Working Note: The Second Generation ADI for the MPICH Implementation of MPI.

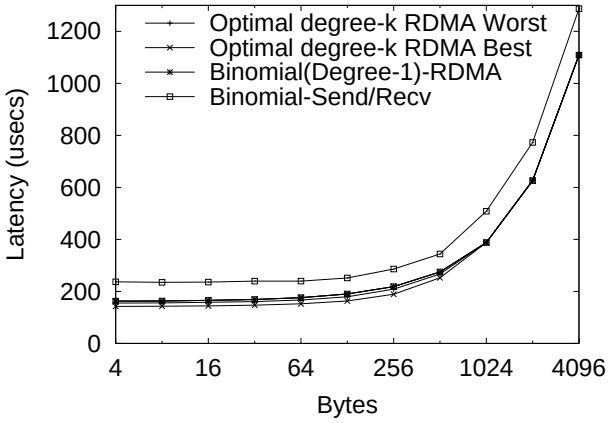

Figure 42. Send/Receive-based binomial, RDMA-based binomial and the most optimal degree-k RDMA-based allreduce Estimations - 512 node Cluster

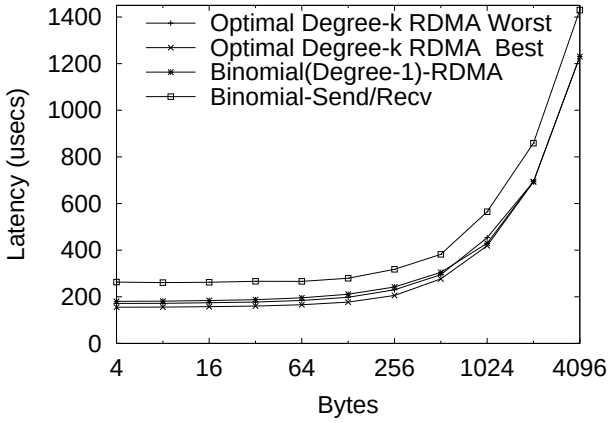

Figure 43. Send/Receive-based binomial, RDMA-based binomial and the most optimal degree-k RDMA-based allreduce Estimations - 1024 node Cluster

[5] Future Technology Group. MVICH: MPI for Virtual Interface Architecture. In *http://www.nersc.gov/research/FTG/mvich*.

[6] R. Gupta, V. Tipparaju, J. Nieplocha, and D. K. Panda. Efficient Barrier using Remote Memory Operations on VIA-Based Clusters. In *IEEE Cluster Computing*, 2002.

[7] Rinku Gupta. Thesis : Efficient Collective Communication using Remote Memory Operations on VIA-Based Clusters, The Ohio State Univeristy, August 2002.

[8] http://www.viarch.org/. Virtual Interface Architecture Specifications.

[9] GigaNet Incorporations. cLAN for Linux: Software Users' Guide. 2001.

[10] M-VIA: A High Performance Modular VIA for Linux. http://www.nersc.gov/ research/FTG/via.

[11] P. K. McKinley, Y.-J. Tsai, and D.F.Robinson. A Survey of Collective Communication in Wormhole-Routed Massively Parallel Computers. In *Technical Report MSU-CPS-94-35, Michigan State University*, 1994.

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

[13] E. Speight, H. Abdel-Shafi, and J. K. Bennett. Realizing the Performance Potential of the Virtual Interface Architecture. In *Proceedings of the International Conference on Supercomputing*, June 1999.

[14] Robert van de Geijn, David Payne, Lance Shuler, and Jerrell Watts. *A Streetguide to Collective Communication and its Application*. Jan 1996.