A Novel Approach to Supporting Communicators for In-Switch Processing of MPI Collectives

Joshua Stern
ECE Department, Boston University

Qingqing Xiong
ECE Department, Boston University

Anthony Skjellum
University of Tennessee, Chattanooga

Martin C. Herbordt
ECE Department, Boston University

ABSTRACT

MPI collective operations can often be performance killers in HPC applications; we seek to solve this bottleneck by offloading them to hardware within the switch itself. We have seen from previous work that moving collectives into the network offers significant performance benefits. However, there has been little advancement in providing support for sub-communicator collectives. We introduce a novel mechanism that enables the hardware to support a large number of communicators of arbitrary shape that is scalable to very large systems. We have integrated this support into an in-switch hardware accelerator to implement support for MPI communicators and full offload of MPI collectives. While this mechanism is universally applicable, we implement it in an FPGA cluster; FPGAs provide the ability to couple communication and computation and so provide an ideal testbed. Preliminary results show that we can achieve substantial performance improvement at acceptable hardware cost, including a 10× speedup over conventional clusters for short message collectives over irregular intra-communicators.

KEYWORDS

MPI Performance, Collective Communication, MPI Communicators, In-Network Processing, FPGAs

.

1 INTRODUCTION

High performance computing (HPC) applications often rely on collective communication for performing operations that require interaction among multiple processes. Simple examples of collectives are the broadcast of data from one process to many, or the gathering of data from many processes into one, usually combined (reduced) with an operator such as *add* or *max*. As collectives are integral to HPC programming, they are necessarily a key part of the Message Passing Interface (MPI) [3]. Collectives may bottleneck application performance [12]. Since implementations such as MPICH [7] consist of point-to-point messages with computations in between, an abundance of support has been added at the software level [2, 12, 21]. This includes new algorithms that can improve the performance of collectives by optimizing them either for low latency with small data sets or for high throughput when dealing with large arrays [21]. However, these algorithms have increasingly complicated the software stack.

This work was supported in part by the NSF through awards #CNS-1405695, #CCF-1618303 and #CCF-1617690, and through a grant from Microsoft Research. Email: (jstern19|qx|jysheng|herbordt)@bu.edu ; tony-skjellum@utc.edu .

In this work we seek to offload the execution of MPI collectives into hardware; in particular, into the communication switches. This has at least five benefits. First, it removes those extra layers of software; second, the hardware implementations are generally at least an order-of-magnitude faster than the software; third, it frees up the processor for other work; fourth, it distributes the execution of collective computation throughout the network, rather than forcing it into source (for broadcast) or destination (for reduction); and fifth, it reduces network load as messages generally only travel a single hop before being merged or duplicated.

Previous work in offloading collective support into hardware has been mostly limited to processing in the NIC [1, 10, 11, 15]. While valuable, the NIC-only approach still leaves much performance benefit on the table, in particular, the fourth and fifth benefits just described. Most obviously, the NIC is an end-point and subject to serialized processing of packets as they arrive, rather than being able to distribute the processing across the network as is possible with in-switch processing. There is one known work on offloading parts of collectives into the switch by BlueGene/Q [9]. It offloads mainly the arithmetic operation in collectives to in-switch hardware accelerators; there are limitations on the number of communicators one node can be in [5].

A critical capability of MPI is the concept of the *communicator*; these are used to define a safe communication context for message passing within a specific group of processes. They are primarily used for performing collective operations over a subset of processes in the network. However, handling sub-communicator collectives in hardware does not come without its share of complications. Communicators unfortunately have significant scalability issues that were explored in [8], meaning we cannot implement them in hardware with the same methods used for managing communicators in software. As we approach exascale computing, the added latency and memory costs of managing communicators would become far too expensive for the switch, exceeding any realistic hardware constraints, and preclude in-switch processing of collectives.

In this work we introduce an in-switch architecture capable of efficiently supporting communicators and the collectives that run on them. We are able to achieve this with a new Communicator Table design, which provides communicator support while consuming minimal memory resources. Since the resources are guaranteed to grow no faster than the log of the number of nodes, this solution is likely to remain relevant far beyond exascale. This project builds off our previous work where we moved `MPI_Reduce` into the network [20], and we have since expanded our design to now include support for the following collectives: Allreduce, Allgather, Gather, Broadcast, and Scatter. While our focus is on intra-communicators

(due to their popularity and simplicity), extending this solution to inter-communicators is straightforward and does not require substantial additional resources.

Since this solution works best if the internal switch hardware is augmented, we use FPGAs to demonstrate and test its viability. This has the added benefit of being useful immediately on clusters with FPGAs interconnected directly through a secondary network [4, 13]. In such clusters, the network has a *direct* topology, such as used by the BlueGene series of computers, with switches being associated with nodes. But the solution also works with *indirect* topologies used in most current high end systems.

The major contribution of this work is the finding that all collective routing decisions–including those with arbitrarily complex communicators–can be made using only a small amount local information. We demonstrate the benefit of this solution: it can be used for a complete offload of MPI collectives. It outperforms conventional networks, while at the same time frees the CPU from having to do any heavy computation for the collective. Our simulations show that we can achieve 10× speedups for MPI collectives over OSU benchmarks for smaller messages, and that we can run these collectives over sub-communicators without sacrificing performance. These results should remain useful far beyond exascale.

2 CONCEPTS

In this section we examine the MPI software stack to identify opportunities for, and the benefits of, offloading collectives. We then cover MPI communicators and the difficulties that they create for a hardware implementation. We explain how placing communicator support in the network would normally exceed the limits of the hardware constraints, thus motivating a novel in-switch design.

2.1 MPI Software Stack

As previously mentioned, MPI collectives force processes into executing long sequences of point-to-point messaging and computation. This is because the new collective algorithms being developed and implemented in MPI are designed to reduce the number of packets that have to traverse the network and avoid congestion. This translates to more work in software for figuring out which chunks of data to send and receive, and which processes with which to communicate. For example, a trivial implementation of MPI_Reduce has every process send data directly to the root, leading to serious congestion in a large network. With a binomial tree algorithm, as seen in Figure 1a, each process is either a leaf, an intermediate node, or the root. Leaf processes simply send data to their parent, but intermediate nodes must compute who all of their children are, receive the data from them, and perform the reduction operation on the received data. They then compute who their parent is in order to then send their intermediate result. The algorithm lessens the number of packets in the network and unclogs the root, but it forces additional work in software. Other algorithms that are commonly used in collectives, such as recursive halving and recursive doubling, can improve the performance of the collective, but they again require that each process perform extra work to determine where to send the data.

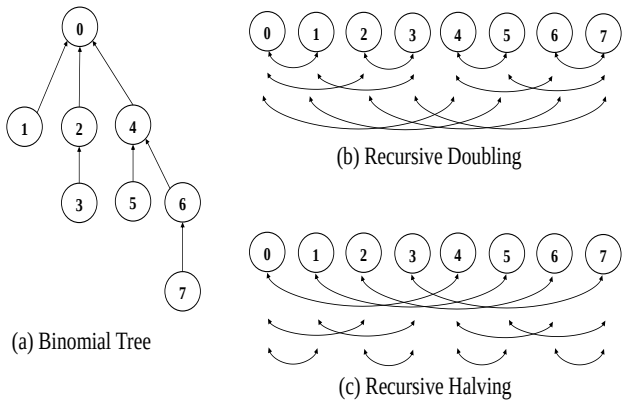


Figure 1: Algorithms used for MPICH Collectives

MPI_FPGA Software Support. Our design, which we call MPI_FPGA, aims to remove all of this software from the responsibility of the CPU and pass the functionality onto the FPGA switch. MPI_FPGA assumes complete transparency with MPI middleware, making it completely portable: it can be integrated into HPC applications without requiring the programmer to have any knowledge of the underlying hardware. Instead, constructs automatically access MPI_FPGA capabilities through enhanced middleware. The design also makes no assumptions about the types of end systems being used, as it is only affects data as it is routed through the FPGAs in the network. We create new functions for each collective that we offload (e.g., MPI_FPGA_Reduce, MPI_FPGA_Scatter), and we place these underneath existing MPI collective functions. If the hardware supports the offload of a particular collective, then the MPI_FPGA replacement functions are used. They take all of the software work for a collective and replace it with a single message to be sent to the FPGA. If a collective does not have offload support, then it is performed normally by the software. All of this is hidden from the application layer, meaning that no changes need to be made to MPI programs.

Upon receiving this message, the FPGA begins the collective operation and perform all of the necessary steps to complete it. If an MPI process is not required to receive the final data, such as the root process in a broadcast operation, then it can return to the application and continue doing work. If the calling process does need to receive results, such as any process in an AllGather operation, then the process can still continue doing other work, but will be interrupted when the final collective operation results have been received and passed up to the CPU from the network. In terms of the MPICH implementation of MPI middleware [7], all of the functionality of the ADI is maintained. We are currently using MPICH-3.2 to design MPI_FPGA [6]; tasks such as packing and computing predefined reduction operations are performed identically in our design. At the channel interface of MPICH we add-in the FPGA communication code that transfers data into the FPGA network, with the actual FPGA hardware sitting below the channel interface (see Figure 2).

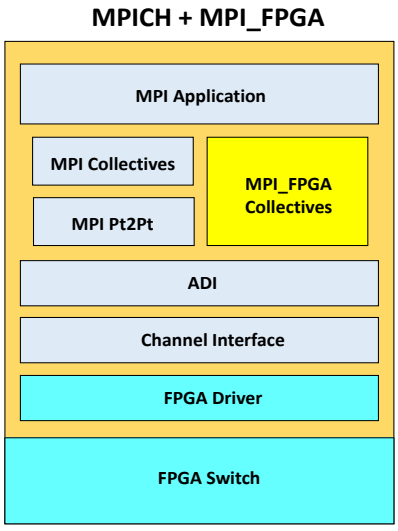


Figure 2: Top-down approach to the MPI_FPGA software stack

Hardware Model. The collective execution logic is placed adjacent to the routing logic so that it can perform computations on data as it passes through the network. As shown in Figure 3, the accelerator logic is broken up into two components: the Collective Control Module and the Reduction Unit (details are given in the Implementation section). By placing the accelerator around the switch rather than integrating it into the switch, we keep the two separate and allow the accelerator to be portable to any type of network switch. When the Collective Control Module receives packets, it operates on the packets if necessary. Otherwise, it simply passes the packet into the switch without modification. The Reduction Unit on the other side of the switch again only operates on packets if they are a part of a reduction operation. If not, the packet is just sent back out into the network. Again, although we have implemented our design on an FPGA, its portability ensures that it is not only independent of the type of switch used, but also the type of hardware used.

2.2 Communicators

Communicator support is absolutely essential in performing collectives in the network, yet little work on collective offload into the network addresses it. It is generally assumed that the only communicator is MPI_COMM_WORLD, meaning the number of ranks involved in any collective operation is the same throughout a program. This was one of our assumptions in our previous work on accelerating MPI_Reduce, but many MPI programs involve multiple communicators, and, in any case, communicators are a central MPI capability and must be supported. A common example of using sub-communicators is when partitioning workload among a matrix of MPI processes and needing to perform collectives on an entire row of column of processes. The most feasible way to do so would be to call a function like MPI_Comm_Split, one of the many functions MPI offers for creating communicators, to divide up the global communicator into sub-communicators. Another motivation

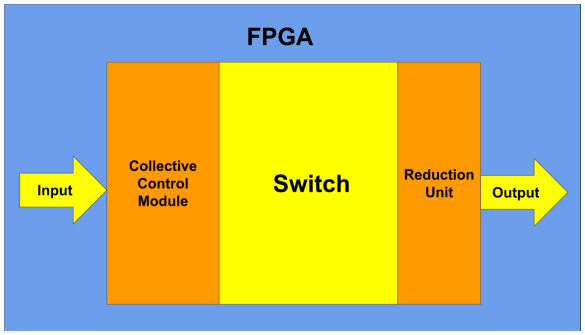


Figure 3: MPI_FPGA Hardware Model

for using multiple communicators is dividing MPI processes into masters and slaves, and then performing collectives on these groups separately.

All communicators are composed of a context id, identifying the communicator, and a process group that contains the list of processes in that communicator. When a new communicator is created, a new process group is created and stored in memory. In large systems, with correspondingly large communicators, the memory consumption of these process groups can lead to scaling issues [8]. To have an entire process group in FPGA memory would require the storing of the list of all ranks included in the communicator. The number of bits required would be *COMM_SIZE*×BITS_PER_RANK, meaning that the resource utilization would grow linearly with the communicator size. For a system with millions of nodes, it would require millions of bits in FPGA for *each* communicator. Having replicated storage of these entire process groups in the network would quickly use up memory resources and slow down the FPGA, so we must only store the minimum amount of information that is needed by the FPGA to complete the collectives.

2.3 Related Work

Previous work has shown that significant performance speedups can be achieved by offloading collectives onto hardware. These generally assume that the hardware is located in the NIC [1, 10, 11, 15], tightly connected with the host CPU via interconnects such as PCI, whereas we add hardware support in the switch. For instance, Arap, et al. [1] offload collectives onto an FPGA cluster; however, they do not mention any communicator support, nor do they integrate into a switch. Their reduction unit also differs from ours as theirs waits until all reduction data is received before performing the reduction, whereas ours begins reductions as soon as data is received. Schimidt, et al. [15] implement `MPI_Reduce` in an FPGA cluster for the AIREN network. Their reduction core consists of floating point units and the output can be looped back as the inputs for further accumulations. This architecture is simple, but lacks flexibility in its reduction capabilities; it can only support one reduction at a time, while our design can support multiple reductions occurring simultaneously. There are several other hardware offload designs implemented on FPGAs; they also lack communicator support, and their collective hardware, e.g., the reduce unit, can only handle one operation at a time [11, 14].

A general solution was provided by Voltaire [22] which included processing support in the router for collectives; this work differs from ours in that the offload is to an in-router CPU rather than a hardware augmentation of the switch.

Among all previous work, the IBM BlueGene project [9] is the most relevant since it offloads collectives in the network router and also takes communicators into account. For instance, BlueGene/Q [5] provides a summing unit for accelerating collective operations, which is available for subcommunicators. BlueGene/Q requires class routes for collective operations. However, there are only 13 class routes available, so that a node can only be in 13 communicators before hardware acceleration for collectives becomes unavailable. More importantly, it does not support packet processing in the network where the accelerator must maintain its own memory [9].

Overall, the BlueGene solutions show the difficulties in implementing in-switch collective support in fixed logic. While high wire utilization is achieved, there are many limitations. Collectives are supported in a separate network and not integrated into the primary router; this may be difficult to replicate. The number of communicators is bounded. The collectives and the operations on those collectives are a fixed subset. And the type of communicators are limited to be either the whole network or a rectangular subset.

In distinction from the previous work, we are the first one to offload both communicator tables and the processing of an entire collective operation on hardware, supporting irregular communicators and providing hardware acceleration of collective packets' processing. Comparing with the NIC offload solutions, our in-switch solution is able to make the shortest collective routes with the ability to process and distribute packets across the network. In our previous work, we implemented MPI_Reduce across MPI_COMM_WORLD without any additional communicator support. In this new paper, we continue using the same hardware model as in [20], but we have obviously implemented new features such as support for 5 new collectives and irregular communicators.

3 DESIGN

In this section we introduce the Communicator Table design, which extends our previous work of collective implementation by now enabling us to implement these collectives across any type of intra-communicator. This design takes advantage of the existing algorithms used to implement MPI collectives in order to minimize resource utilization and latency.

3.1 Communicator Table

The purpose of the Communicator Table is to manage communicator information that is needed for the Collective Control Module to make packet forwarding decisions. To minimize the resources required, the table only holds the local data that is necessary to complete the implemented collectives. This means that each switch needs a way of obtaining this local data, which is a list of the other ranks with which it must communicate to perform each collective. The contents of this list, for a given communicator, can be determined immediately after its creation in software.

In Table 1, we show the different algorithms that are used in the MPICH-3.2 implementations of 6 popular collectives. We see that

Table 1: MPICH-3.2 Collective Algorithms for short and long messages

MPI Collective Algorithms		
	Short Messages	Long Messages
Reduce	Binomial Tree	Recursive Halving and Recursive Doubling
Allreduce	Recursive Doubling	Recursive Halving and Recursive Doubling
Broadcast	Binomial Tree	Binomial Tree and Ring
Scatter	Binomial Tree	Binomial Tree
Gather	Binomial Tree	Binomial Tree
Allgather	Recursive Doubling	Ring

the 3 most used algorithms are binomial tree, recursive halving, and recursive doubling. The ring algorithm is also sometimes used, but its implementation is trivial so we focus on the others for now. By being able to implement these 3 algorithms, we can perform all of the collectives that use them. Looking back at Figure 1 (where we show the structure of these algorithms), for each rank, we see that we can identify the subset that the given rank must communicate with. For example, rank 0 must communicate with the following set in all 3 algorithms: 1, 2, 4. For rank 5, although it only communicates with rank 4 in the binomial tree algorithm, its communicating set for all algorithms is 4, 7, 1. This means that, for each rank, we can identify a subset of processes to communicate with, and this subset will meet the communication needs of all collectives that use these algorithms.

Storing this subset in FPGA memory is much more efficient than storing an entire process group: it is equal to the log of the communicator size. We can prove this directly from the properties of binomial trees. In a binomial tree, the node that communicates with the most other nodes is the root. If we can determine the maximum number of root children of a given binomial tree, then we can translate this to the number of ranks that can be stored in one of these rank subsets. In a binomial tree of order m, the root has m children, and the total amount of nodes in the tree is 2^m. For $m = 0$, we have a binomial tree that is $2^0 = 1$ node in size. The root has 0 children, thus making the number of other ranks for it to communicate with during a collective equal to $\lg(1) = 0$. If we have a binomial tree of order $m + 1$, we know from the basic property of binomial trees that this tree is composed of 2 binomial trees each of order m. The root will have $m + 1$ children, making the size of its communicating subset equal to $m + 1$. We can also determine this by calculating that the number of nodes in the tree is $2 \times 2^m = 2^{m+1}$. If we apply our previous formula to calculate the size of the subset, we get $\lg(2^{m+1}) = m + 1$. This proves that for any communicator, the subset size will be equal to lg(COMM_SIZE).

Once the FPGA obtains this subset of ranks for a communicator (we discuss later how this is done), it stores the addresses of these ranks in a table along with the subsets from other communicators. As shown in Figure 4, the Communicator Table holds a row for each communicator that the current FPGA is a member of. In each row, we store a small amount of meta-information such as the communicator size and the rank within the communicator that the current FPGA is associated with, followed by the subset of

Valid	Context ID	Comm Size	Local Rank	First Address	Second Address	Third Address
1	0	8	0	1	2	4
1	1	4	0	1	2	NULL

Figure 4: Communicator Table Structure

processes that the FPGA will be communicating with for collective operations. Each communicator entry is indexed into the table using its context id. This makes it easy to look up the communicator that an incoming packet is from, because the context id is sent within the packet header.

Once the FPGA has a table entry for a given communicator, it can use that data to perform any collective that uses a binomial tree, recursive halving, or recursive doubling. The reason that we can use this table is that for any collective algorithm in a communicator, each rank will communicate with the same subset of ranks regardless of how many times the collective is called. This is what allows us to store this communicator information in the table whenever the communicator is created, because these subsets do not change. This means that once a valid entry is loaded into the table, no updates on that entry are ever required until the communicator is freed. The Collective Control Module (see next section) uses the Communicator Table along with packet header information to decide how to deal with packets in a collective operation.

3.2 Communicator Table Entry Creation

When a new communicator is created in software, the FPGA needs a way of obtaining the Communicator Table entry from the host CPU. If an MPI process is a member of a newly created communicator, then the software generates a special message containing the Communicator Table entry data and sends it to the FPGA. This requires that, for each communicator creation function, the CPU calculates and retrieves from memory the physical addresses of the subset of ranks that will be stored in the table entry. Once the new entry is filled in, the FPGA can handle new collectives occurring within this communicator.

In order for the CPU to obtain the necessary addresses, we have written a hook function and inserted it at the end of MPICH communicator creation functions. This hook function checks whether an MPI process is a member of a new communicator, and if so, calculates the subset of ranks for it to communicate with. Then, for each rank in the subset, it obtains the rank's connection string from the key-value space in memory which is used to hold virtual connections. From this connection string, the physical address is extracted and packaged alongside communicator meta-data into a message to be sent to the FPGA. Although this operation does lead to a small amount of overhead in creating communicators, this overhead is only paid for once during communicator creation. The MPI application can then perform any number of collectives on the communicator at no additional cost, making the cost of communicator creation negligible when compared to the cost of later performing large collectives on these communicators.

The hook function is designed so that it is first run after MPI_Init(); this allows the first entry in the Communicator Table to be filled

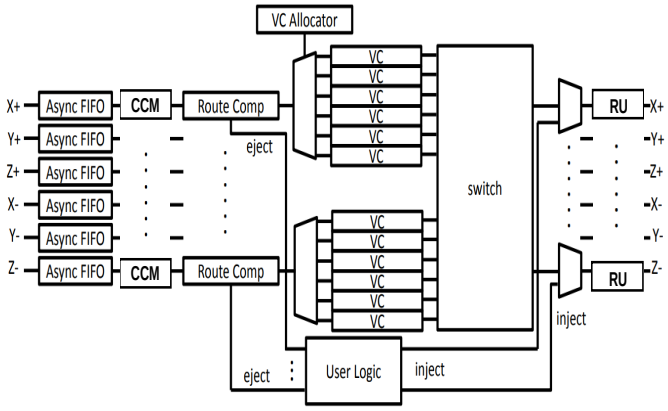


Figure 5: Dynamic router combined with MPI collective offload support: Collective Control Module and Reduction Unit

with communicator information for MPI_COMM_WORLD. With the creation of all other communicators, the CPU will only call this hook function if it is a member of the new communicator. This means that the Communicator Table in the FPGA will only contain information on communicators that it is a member of, and the creation of all other communicators will be ignored. We include a valid bit with each table entry; these are initialized to invalid. Whenever a new communicator packet is received from the CPU, the valid bit is set so that the table entry can be used for packet processing of future collective packets from that communicator.

4 IMPLEMENTATION

4.1 Overview

This section describes the different components of the hardware architecture, which are displayed in Figure 5. The base of our design is a virtual-channel dynamic router that was previously built by our group, implemented in programmable logic, and used for inter-FPGA communication [16–19]. The router is designed to be used in an FPGA cluster interconnected in a 3D torus and has 6 input and 6 output ports. These ports are connected to Multi-Gigabit Transceivers (MGTs), which allows FPGAs to be directly connected to each other. The router normally has a 4 stage pipeline: route computation, virtual channel allocation, switch allocation, and switch traversal. To be clear, the route computation stage is responsible for determining a packet's output port based off the destination field in its header, but it does not have any effect on the packet's destination address. As seen in Figure 5, with the added MPI support, we have extended the pipeline to six stages.

The MPI offload support was designed around the router in order to keep the overall design modular: the accelerator architecture is portable to any other standard router. The MPI support is divided into 2 modules, the Collective Control Module (CCM) and the Reduction Unit (RU). The former is responsible for calculating new forwarding and multicast destinations for collective packets; it contains the communicator support. The module is placed before the router so that the output port of the packet can be calculated in

the route computation stage after it is assigned a new destination. The Reduction Unit sits on the output end of the router and is used for performing MPI_Reduce and MPI_Allreduce computations. It maintains a reduction table of buffers that store temporary reduction results. Once all of the necessary packets for a reduction are received, the resulting packet is released to its output port. This unit is placed after the switch due to the fact that all packets going into any particular reduction unit will exit using the same output port. This prevents the reduction unit from having to manage the output ports of each packet.

4.2 Collective Control Module

The Collective Control Module (Figure 6) is responsible for performing all of the algorithmic work found in the software of MPI_Reduce, MPI_Allreduce, MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Allgather, and any other collectives implemented in the future. By moving all of the software functionality into hardware, we avoid the large costs of software and free the CPU from having to switch back and forth between computation and communication. This allows the host CPU to start a collective by sending a single message to the FPGA and then return to the MPI program. If the CPU needs the results of the collective, it will receive them from the FPGA.

When packets enter the router from the input MGTs, they first go through the Collective Control Module. If they are not part of collective operation or are not destined for the current FPGA, then they simply pass through unchanged. If they are part of an offloaded collective and the destination address in the packet header matches that of the current FPGA, then the module uses the Communicator Table to determine new destinations for the packet.

In order for the Collective Control Module to determine which collective a packet is a part of, we added a collective opcode field to the packet header. With this, we can perform work for each collective algorithm in parallel and then use the opcode to decide which algorithmic results to use for the packet. In Figure 6, we display the parallel architecture. Within each of these algorithm blocks, we perform computations using input from the packet header and Communicator Table entry. For a reduction, we might have to calculate the parent node to send the packet to, or for a broadcast, all of the children nodes to multicast the packet to.

The communicator table also eases the computation required to calculate these destinations. In the previous design, destinations had to be determined on-the-fly, but now the problem has been simplified to calculating table entry indexes on the fly. In the case that a packet needs to be sent to multiple destinations, these destinations are also adjacent in the table entry. A bit vector is used for keeping track of these destinations for multicast, which results in much less work than if destinations were repeatedly calculated on-the-fly. Once a packet passes through the Collective Control Module, it is passed to the route computation stage (in the routing pipeline) where its output port is calculated.

The implementation also supports multiple algorithms for the same collective operations, which is also how MPICH-3.2 works. This requires the packet header opcode field to include an additional bit about whether the collective message is classified as short or long, but it allows the Collective Control Module to use the optimal

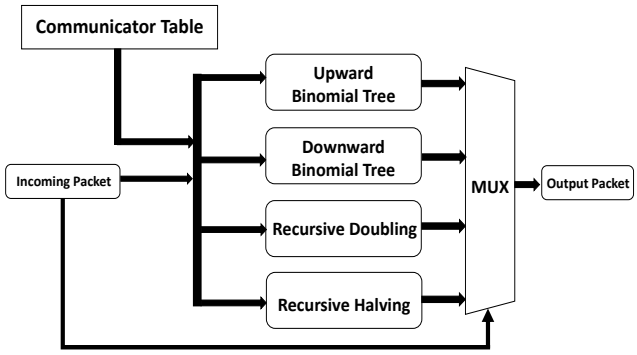


Figure 6: Collective Control Module

algorithm for a collective. We show the algorithms used for short and long messages in Table 1, and we see that the same set of algorithms is used for both large and short message collectives. This means that if we can implement long message collectives for free, because we are already performing same the algorithmic computations for the small message collectives.

4.3 Reduction Unit

The Reduction Unit is used for performing the reduction computations necessary for MPI_Reduce and MPI_Allreduce. Once packets pass the switch traversal stage of the router pipeline, if they are identified as part of one of these collectives by their opcode field in the packet header, then they are transferred to the Reduction Unit. This unit performs and manages all reduction computations, thus freeing the host CPU from having to do so. The unit consists of a reduction table, which is indexed and capable of supporting multiple reductions simultaneously. Once a reduction packet enters from the switch output, the unit's control logic examines the packet header and places the data in the appropriate table slot. If the reduction table slot for an incoming packet is empty, then the packet is simply copied into the reduction table slot. If the table slot is not empty, it means that the reduction has already begun. In this case, the data payload of the incoming packet and the data already contained in the reduction table are combined, with the result later being fed back into the reduction table entry (see Figure 7).

The arithmetic unit is constructed from using standard methods including use of vendor IP. The default design supports addition, multiplication, maximum, minimum, but is trivially extendable for other operations; eventually we will extend this design to include user specified functions. All functions are run in parallel, both pipeline and superscalar. The reduction opcode in the packet header is used to select which output is to be stored back in the reduction table.

Each reduction table entry slot also keeps track of the number of child nodes for any given reduction. Whenever an incoming packet enters the unit, the reduction table slot records how many child nodes are required for that particular reduction and keeps track of the number of child nodes remain as the reduction continues. When a reduction table entry has received packets from all child nodes,

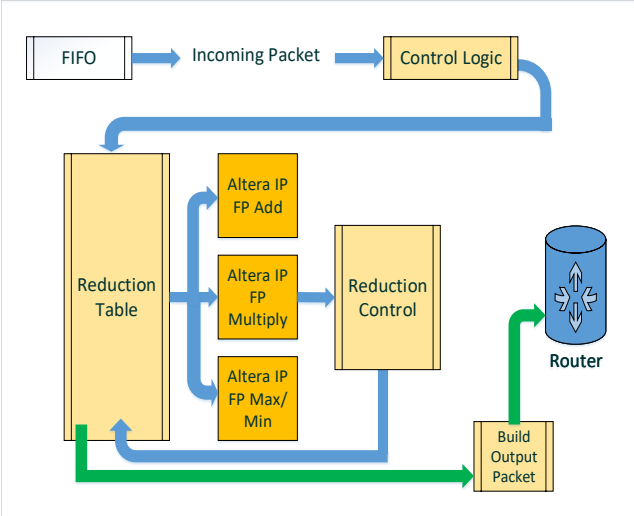


Figure 7: Flow of packets through the inner components of the reduction computation unit

and the reduction has been completed, a new packet is built and sent back to the router. The router is then notified that a reduction has been completed and gives the result to the proper output port.

By designing the reduction table to have multiple entry slots, we can support multiple reductions taking place at the same time. If the reduction is of a large data set and needs to be divided into smaller reductions, each occupies a different slot of the reduction table. By pipelining the floating point units, these multiple reductions can all be completed one after another. This unit can also support separate unrelated reductions, and is flexible enough to allow any order of reductions occurring throughout the reduction table. If the reduction table has 100 entry slots, then this module can support up to 100 different reductions occurring together, all possibly performing different types of reduction operations. To handle the case of the reduction table filling up due to a reduction of a large enough array, a local control unit keeps track of the capacity of the reduction table and buffers incoming reduction packets until the reduction table has open slots.

5 EVALUATION

We evaluate the design on a cluster with FPGAs in the network datapath (e.g., the Microsoft Catapult I [13] and the BU/UFlorida Novo-G# [4]). We use Quartus II on a ProceV FPGA board from Gidel with an Altera Stratix V 5GSMD8. On the FPGA, we put a local unit connected to our new router design as shown in Figure 5. Both the Collective Control Module and Reduction Unit have been inserted into the router datapath. The FPGA is run at 150 MHz, which could be doubled with normal optimization and increased much further if the switch were to be implemented on an ASIC. For inter-FPGA communication, we use Multi-Gigabit Transceivers, which provide a bandwidth of 40Gbps and latency of 200ns.

5.1 MPI Collectives

In order to get the performance of our accelerator, we simulated 4-, 8-, 16-, 32-, 64-, and 128-node systems in ModelSim. We included the inter-FPGA communication delays of the MGTs and performed all collectives using double precision floating point types. For each of these different network sizes, we simulated our accelerated MPI_FPGA_Reduce, Allreduce, Broadcast, Scatter, Gather, and Allgather. We then compared these results to their respective OSU benchmarks, which is a well-known set of MPI benchmarks. The benchmarks were run on the Boston University Shared Computing Cluster (SCC) using 28 core 2.4 GHz Intel Xeon E5s connected by EDR Infiniband. We were limited to only being able to allocate 16 nodes at a time. To get around this and retrieve benchmark results for 32-, 64-, and 128- node clusters, we measured the difference in performing collectives on different cores in a single node versus on different nodes with one core on each node. We then used this delay to estimate the results for the 32-, 64- , and 128-node benchmarks.

Figure 8: MPI_FPGA_Reduce vs MPI_Reduce

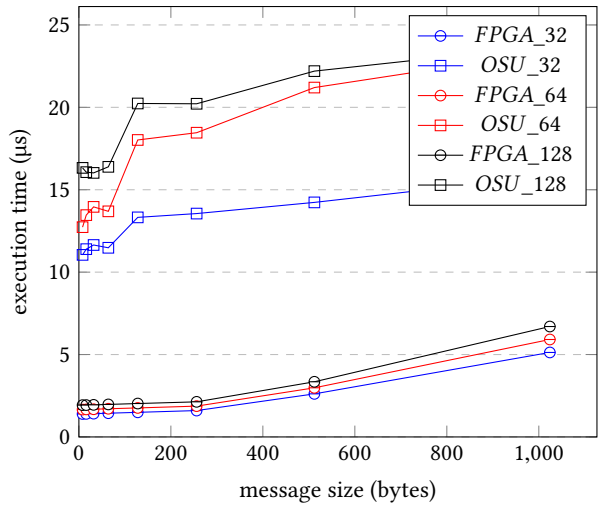


Figure 9: MPI_FPGA_Allreduce vs MPI_Allreduce

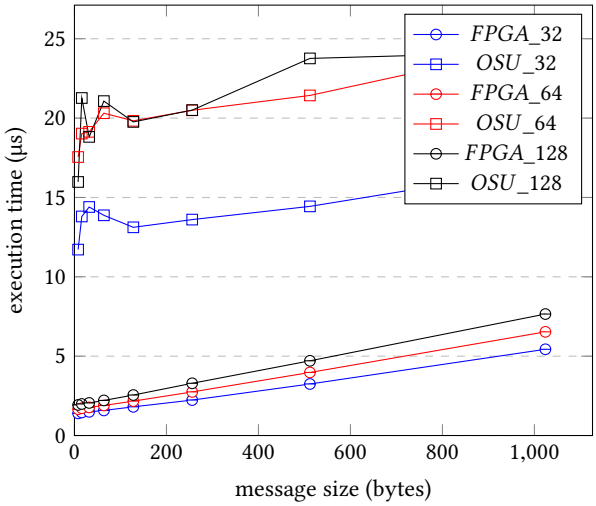


Figure 10: MPI_FPGA_Bcast vs MPI_Bcast

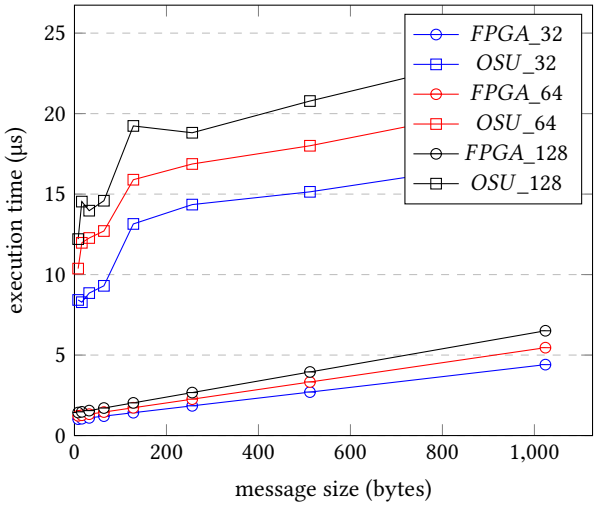


Figure 11: MPI_FPGA_Scatter vs MPI_Scatter

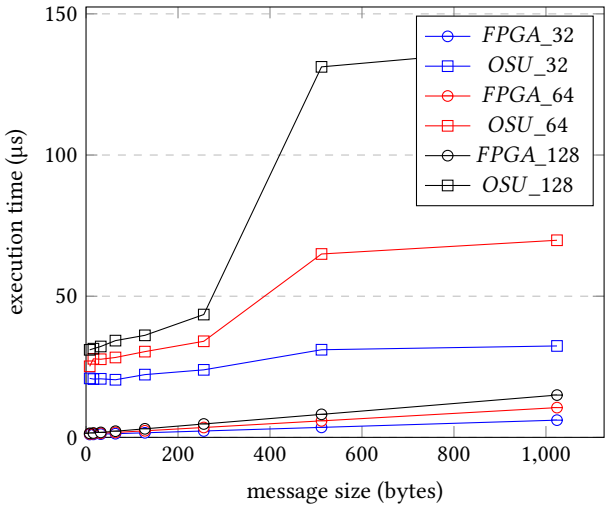


Figure 12: MPI_FPGA_Gather vs MPI_Gather

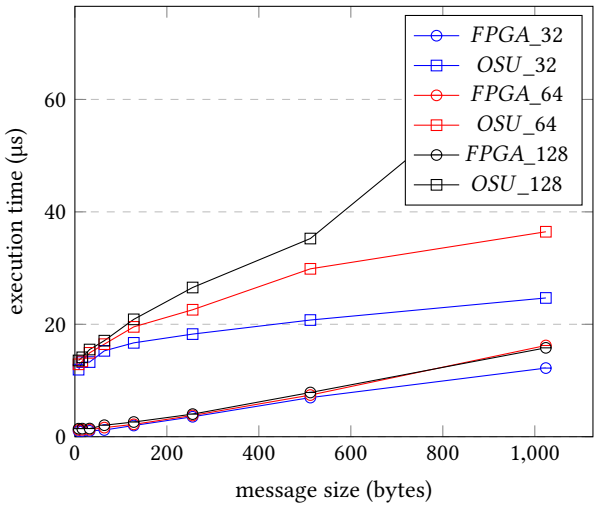
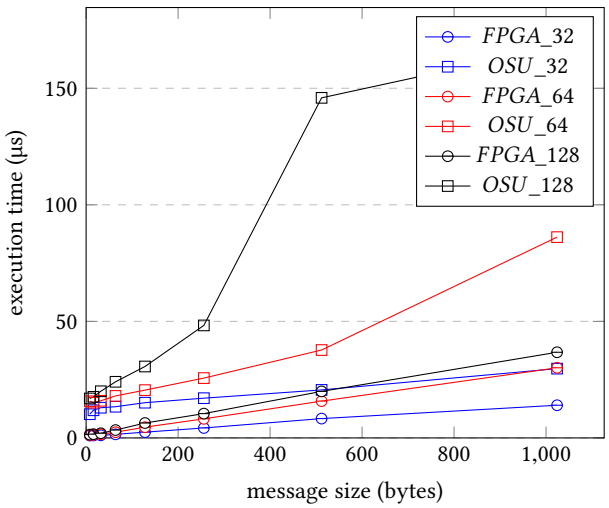


Figure 13: MPI_FPGA_Allgather vs MPI_Allgather



Overall Collective Latency. In Figures 8-13, we show the simulation results of MPI_FPGA collectives for small to medium array sizes on the 32-node, 64-node, and 128-node systems and compare our results against those of the OSU benchmarks. This graphed data represents the time that it took for the last process in a collective to complete the operation. We see that for small array sizes, MPI_FPGA achieves much better performance over the OSU benchmarks. We also see that the MPI_FPGA speedup over the OSU benchmark is maintained as the network grows, thus indicating the scalability of MPI_FPGA. As the array size grows, the MPI_FPGA speedup begins to diminish to about 2×. This is expected because of the low clock rate of FPGAs. The obvious and simple solution is add more parallelism, which translates to more resource usage. To view the results through a different perspective, in Table 2 we provide a table specifying the speedups that we achieve for each of these collectives at the message size of 128 bytes. We can achieve speedups for different collectives ranging from 5× to 15×.

Table 2: MPI_FPGA Speedups over OSU Benchmarks on BU SCC (128 byte messages)

MPI_FPGA Speedup Over OSU			
	32 ranks	64 ranks	128 ranks
Reduce	8.92	10.23	9.98
Allreduce	8.78	9.15	9.74
Bcast	9.23	9.21	9.45
Scatter	13.72	13.72	15.01
Gather	8.37	8.87	7.99
Allgather	5.86	7.34	7.15

Average Case Results. We also collected the average case results, or the average amount of time that it took for a process in a collective to finish. We found that for Allreduce and Allgather, the worst-case results and average-case results were nearly identical. This is because in these types of operations, every process must wait for data from every other process, so no process can complete the collective until all processes have at least started it. For the other collectives, the speedup is much larger for the average case. This is because if an MPI_FPGA rank does not require the results of a collective, then it just sends a special message to the FPGA and returns to the application. For example, in a gather where only the root cares about the results, every other process completes its work for the collective by just sending their data to the FPGA. These processes could then continue doing computational work for the application. We can achieve 180× and 100× speedups for MPI_FPGA_Reduce and MPI_FPGA_Gather respectively in a 128-node network.

These results should not be represented as the actual speedup of a collective, but are nevertheless significant. It is common in MPI applications to have one process be a master that sends work to the other slave processes. Whenever a slave process completes its work, it notifies the master, which will then assign the slave more work. If the slaves are performing a collective operation such as a reduction or gather, all of the non-root slaves could return from the collective immediately after sending the special collective message to the FPGA. This would then allow slave processes to complete their work faster and request additional work from the master process at a higher rate. In this case, and any other example where an MPI process has work to perform directly following a collective, the application performance would benefit greatly from MPI_FPGA.

Resource Usage. The resource consumption of the Communicator Table can vary based on a user parameter that defines how many communicators a single MPI process can be a part of. When calculating our total resource consumption, we limited this to 30, but it can obviously be changed to the user's specifications. Let NUM_COMM be the maximum amount of communicators a process can be in, let WORLD_SIZE be the number of ranks in MPI_COMM_WORLD, and let ADDR_LEN be the number of bits is a process's physical address. We earlier proved that a Communicator Table entry has lg(WORLD_SIZE) other ranks that it might need to communicate with, so the amount of bits used by the Communicator Table (not included small amount of meta-data) can be calculated as follows:

$$bits = NUM\_COMM * lg(WORLD\_SIZE) * ADDR\_LEN \quad (1)$$

The MPI_FPGA router consumes 100,768 ALMS, which accounts for about 38% logic utilization of the Stratix V FPGA. This number can grow if more IP cores are added to the Reduction Unit, but we have found that adding a single mathematical IP core increases the resource usage by less than 1% of the total ALMs. MPICH-3.2 has 12 predefined reduction operations, so for a full implementation, we would need at most 12 IP cores. Conservatively, this would amount to no more than 5% of additional ALMs needed. However, a Stratix 10 FPGA has over four times as many logic resources than a Stratix V, meaning that our new router design would consume about 11% of the FPGA resources.

6 DISCUSSION AND FUTURE WORK

We present a new method for supporting MPI communicators and accelerating collectives in the network switch. We begin by considering the movement towards exascale computing and the need for offloading collectives and communicator support into hardware. Other groups, including us, have already shown the ability to implement MPI collectives in the network. However, there has been a lack of support for collectives occurring over irregular communicators. In our investigation of communicators, we find that a storing entire process groups in the network is not a scalable solution. We then introduce our Communicator Table, which takes advantage of the properties and patterns of collective communication in order to provide the accelerator hardware with the minimum amount of communicator information needed to perform collectives. By supporting a full offload of 6 popular collectives, we remove all of the collective operation software from MPI and implement the functionality in the switch. Our hardware support has been integrated into a reconfigurable VC router, but remains portable enough that it is independent of the type of router and and system infrastructure. We compare our network simulations with hardware support to OSU benchmarks running on a conventional cluster, and we and find that our in-switch accelerator achieves significant and scalable speedups over the benchmarks.

ACKNOWLEDGMENTS

Support is acknowledged from the National Science Foundation under Grants Nos. CCF-1562659, CCF-1562306, CCF-1618303, CCF-1617690, CCF-1822191, CCF-1821431. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Omer Arap and Martin Swany. 2016. Offloading Collective Operations to Programmable Logic on a Zynq Cluster. In *IEEE 24th Annual Symposium on High-Performance Interconnects*. 76–83.

[2] E. W. Chan, M. F. Heimlich, A. Purkayastha, and R. A. van de Geijn. 2004. On optimizing collective communication. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*. 145–155. https://doi.org/10.1109/CLUSTR.2004.1392612

[3] Message Passing Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. Knoxville, TN, USA.

[4] A. George, M. Herbordt, H. Lam, A. Lawande, J. Sheng, and C. Yang. 2016. Novo-G#: A Community Resource for Exploring Large-Scale Reconfigurable Computing Through Direct and Programmable Interconnects. In *IEEE High Perf. Extreme Computing Conf.*

[5] M. Gilge. 2013. *IBM System Blue Gene Solution Blue Gene/Q Application Development.* An IBM Redbooks publication.

[6] William Gropp, E Lusk, D Ashton, R Ross, R Thakur, and B Toonen. 2003. *MPICH Abstract Device Interface Version 3.3 Reference Manual.* Technical Report. Draft MCS-TM-00, Argonne National Laboratory, 2002. http://www-unix. mcs. anl. gov/mpi/mpich/adi3.

[7] William Gropp, Ewing L. Lusk, Nathan E. Doss, and Anthony Skjellum. 1996. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Comput.* 22, 6 (1996), 789–828. https://doi.org/10.1016/0167-8191(96)00024-5

[8] Humaira Kamal, Seyed M Mirtaheri, and Alan Wagner. 2010. Scalability of communicators and groups in MPI. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. 264–275.

[9] Sameer Kumar, Amith Mamidala, Philip Heidelberger, Dong Chen, and Daniel Faraj. 2014. Optimization of MPI Collective Operations on the IBM Blue Gene/Q Supercomputer. *International Journal of High Performance Computing Applications* 28, 4 (2014), 450–464.

[10] Mellanox. 2019. Fabric Collective Accelerator (FCA). https://www.mellanox.com/. (2019).
[11] Yuanxi Peng, Manuel Saldana, and Paul Chow. 2011. Hardware support for broadcast and reduce in mpsoc. In *Proc. IEEE Conf. on Field Programmable Logic and Applications*. 144–150.
[12] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. 2005. Performance analysis of MPI collective operations. In *19th IEEE International Parallel and Distributed Processing Symposium*. 8 pp.–. https://doi.org/10.1109/IPDPS.2005.335
[13] A. Putnam and et al. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *International Symposium on Computer Architecture*. 13–24.
[14] Manuel Saldaña, Arun Patel, Christopher Madill, Daniel Nunes, Danyao Wang, Paul Chow, Ralph Wittig, Henry Styles, and Andrew Putnam. 2010. MPI as a programming model for high-performance reconfigurable computers. *ACM Transactions on Reconfigurable Technology and Systems* 3, 4 (2010), 22.
[15] Schmidt, Andrew G and Kritikos, William V and Gao, Shanyuan and Sass, Ron. 2012. An evaluation of an integrated on-chip/off-chip network for high-performance reconfigurable computing. *International Journal of Reconfigurable Computing* 2012 (2012), 5.
[16] J. Sheng, B. Humphries, H. Zhang, and M.C. Herbordt. 2014. Design of 3D FFTs with FPGA Clusters. In *IEEE High Perf. Extreme Computing Conf.*
[17] Jiayi Sheng, Qingqing Xiong, Chen Yang, and Martin C Herbordt. 2016. Application-Aware Collective Communication. In *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, Vol. 3.
[18] J. Sheng, C. Yang, and M.C. Herbordt. 2015. Towards Low-Latency Communication on FPGA Clusters with 3D FFT Case Study. In *Proc. International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*.
[19] J. Sheng, C. Yang, and M.C. Herbordt. 2018. High Performance Dynamic Communication on Reconfigurable Clusters. In *Proc. IEEE Conf. on Field Programmable Logic and Applications*.
[20] J. Stern, Q. Xiong, J. Sheng, A. Skjellum, and M.C. Herbordt. 2017. Accelerating MPI_Reduce with FPGAs in the Network. In *Proc Workshop on Exascale MPI*.
[21] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.
[22] A. Wachtel. 2010. Boosting Scalability of InfiniBand-based HPC Clusters. Voltaire, Inc.. (2010).