Received: 1 January 2021 | Revised: 12 October 2021 | Accepted: 5 November 2021

DOI: 10.1002/cpe.6769

RESEARCH ARTICLE

WILEY

Reconfigurable switches for high performance and flexible MPI collectives

Pouya Haghi[1] | Anqi Guo[1] | Qingqing Xiong[1] | Chen Yang[1] | Tong Geng[1] | Justin T. Broaddus[2] | Ryan Marshall[2] | Derek Schafer[2] | Anthony Skjellum[2] | Martin C. Herbordt[1]

[1]Department of Electrical and Computer Engineering, Boston University, Boston, Massachusetts, USA

[2]Simcenter & Department of Computer Science and Engineering, University of Tennessee at Chattanooga, Chattanooga, Tennessee, USA

Correspondence
Pouya Haghi, Department of Electrical and Computer Engineering, Boston University, Boston, MA, USA.
Email: haghi@bu.edu

Funding information
National Science Foundation, Grant/Award Numbers: CCF-1919130, CNS-1925504

Abstract

There has been much effort in offloading MPI collective operations into hardware. But while NIC-based collective acceleration is well-studied, offloading their processing into the switching fabric, despite numerous advantages, has been much more limited. A major problem with fixed logic implementations is that either only a fraction of the possible collective communication is accelerated or that logic is wasted in the applications that do not need a particular capability. Using reconfigurable logic has numerous advantages: exactly the required operations can be implemented; the level of desired performance can be specified; and new, possibly complex, operations can be defined and implemented. We have designed an in-switch collective accelerator, *MPI-FPGA*, and demonstrated its use with seven MPI collectives and over a set of benchmarks and proxy applications (MiniApps). The accelerator uses a novel two-level switch design containing fully pipelined vectorized aggregation logic units. Essential to this work is providing support for sub-communicator collectives that enables communicators of arbitrary shape, and that is scalable to large systems. A streaming interface improves the performance for long messages. While this reconfigurable design is generally applicable, we prototype it with an FPGA-centric cluster. A sample *MPI-FPGA* design in a direct network achieves considerable speedups over conventional clusters in the most likely scenarios. We also present results for indirect networks with reconfigurable high-radix switches and show that this approach is competitive with *SHArP* technology for the subset of operations that *SHArP* supports. *MPI-FPGA* is fully integrated into MPICH and is transparent to MPI applications.

KEYWORDS

collectives, FPGA-centric clusters, high performance computing, in-network computing, MPI

1 | INTRODUCTION

Collectives are often a large fraction of total communication in high performance computing (HPC) applications[1-6] and have been shown to bottleneck performance.[7-9] As collectives are integral to HPC, and since much communication in production HPC is based on MPI (see e.g., Reference 6), addressing the acceleration of collectives necessarily means dealing with them within an MPI framework. Collectives in MPI implementations (such as MPICH[10]) generally consist of point-to-point messages with computations in between. Thus much support has been added at the software level;[7,11,12] however, the addition of these algorithms has greatly complicated the software stack.[13]

Parts of this work have been presented at workshops with no proceedings, published in an extended abstract, and published in a conference proceedings.

In this work we offload MPI collectives into FPGA hardware (*MPI-FPGA*) and, in particular, into logic appended to the communication switches. In general, handling collectives in hardware has several benefits. First, it removes those extra layers of software (e.g., Reference 13); second, the hardware implementations can be an order-of-magnitude or more faster than the software they replace; and third, it frees up the processor for other work (e.g., Reference 14). Handling collectives in the switch, rather than in the NIC,[15-19] adds other advantages: it distributes the execution of collective computation throughout the network, rather than serializing it in the source (for broadcast) or the destination (for reduction); and it reduces network load as messages often only travel a single hop before being merged or duplicated.

Compute-in-the-network has been studied since the early days of computing through structures such as adder trees and sorting networks;[20] it is also fundamental to the more powerful PRAM models explored in the 1980s.[21] However, there appear to be just two modern commercial versions of in-switch computing: the IBM BlueGene family[8,22] and, in current use, certain switches from Mellanox.[9] Both of these have limitations (described below) that are, in part, the result of being ASIC-based and so having strictly bounded capabilities with limited flexibility. Moreover, being commercial products, few details are available about their implementation, which curtails their use as the basis of further research.

There are at least two plausible architectural targets for using reconfigurable logic for in-switch compute-in-the-network. One is to use reconfigurable logic in the router (already common for other purposes[23]) in an indirect network. A second is in FPGA-centric clusters with direct FPGA-FPGA interconnects.[24-27] In this work we consider both indirect and direct networks.

Implementations with reconfigurable logic have several inherent advantages over those with fixed logic. First, they are not limited to a small, fixed set of operations, for example, *SHArP* only supports MPI_Allreduce and MPI_Barrier operations[9] and a few primitives. Second, hardware resources can be configured to match application requirements, for example, by increasing the arithmetic support as needed. This can even be done at runtime for applications in which the communication data volume is unpredictable.[28] Third, support can be extended beyond simple datatypes to more complex structures such as matrices, tensors, and user defined datatypes.[4,28] Fourth, compute-in-the-network can be generalized still further to support *altruistic* or *opportunistic*computing.[29] And finally, since reconfiguration time is similar to program load time, only resources that will actually be used need to be configured; the remaining logic can be used for other purposes. All of these scenarios can be accomplished without incurring the cost of designing and fabricating ASIC chips for each application, or devoting ASIC resources to functions that rarely occur.

An essential part of implementing MPI collectives is handling the critical MPI feature of the *communicator*; these are used to define a safe communication context for message passing within a specific group of processes. Communicators have significant scalability issues,[30] meaning we cannot implement them in hardware with the same methods used for managing communicators in software; for this reason in-switch implementations sometimes limit communicator geometry.[22] In this work we introduce an in-switch design for general communicator support that consumes minimal memory resources. Moreover, as 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.

The main contribution is the design, implementation, and evaluation of a set of FPGA-based in-switch MPI collectives. We believe this to be the first FPGA version to be fully integrated into a general router. To achieve this end, a novel 2-level in-switch design is proposed with full-pipeline capability. Essential to this design are the seamless integration to a general router, added hardware support for aggregating (reordering) packets for reduce-type (gather-type) operations, and the flexibility to add more computational resources as the switch bandwidth increases. *MPI-FPGA* is fully integrated into MPICH; *MPI-FPGA* is therefore transparently usable by any MPI application. It is also easily extended to support additional collectives or integrated into other MPI implementations. A second contribution is the finding that all collective routing decisions—including those with arbitrarily complex communicators—can be made using only a small amount local information. Finally, an efficient streaming interface is provided to ensure a fine-grained communication specially for long message sizes.

Our experiments show that *MPI-FPGA* in a direct network can achieve a significant improvement for MPI collectives over a CPU cluster for a large range of message sizes. For FPGA-based high-radix switches in an indirect network it is possible to make comparisons with *SHArP* technology: *MPI-FPGA* is competitive and improves the performance of Allreduce operation. We have also experimented with NAS parallel benchmarks and two MiniApps from the Mantevo project[31] (miniFE and HPCCG). The most significant result, however, is that this performance is possible while simultaneously supporting reconfigurable in-switch computing in MPI collectives and many extensions (e.g., Reference 4).

The rest of this article is organized as follows. Section 2 provides background information on some of the key concepts used throughout this article. Section 3 introduces in-network communicator support and it sheds light on some of the design choices. Section 4 describes in detail the proposed switch and the modules used in the design. Section 5 analyzes and compares the performance of our approach (in-network collective offload) with that of the existing approach. Section 6 presents experimental results. In Section 7, related work is discussed and Section 8 concludes the article.

2 | CONCEPTS

We review the MPI software stack to identify opportunities for, and the benefits of, offloading collectives. Next, we discuss our hardware model. We then cover MPI communicators and the difficulties they create for hardware implementations. Finally, we discuss the proposed streaming interface needed to efficiently support long messages.

2.1 | MPI collectives

MPI collectives are generally implemented so that processes execute sequences of point-to-point messages and computations. Various algorithms are used, but priority is given to those that avoid congestion and minimize the total number of packets. However, these may translate into more work in software for deciding which chunks of data to send and receive, and the processes with which to communicate. For example, a trivial implementation of *MPI_Reduce* has every process send data directly to the root and leads to congestion in a large network. In contrast, with a binomial tree algorithm, as seen in Figure 1A, each process could be either a leaf, intermediate, or root process. Leaf processes simply send data to their parent, but intermediate processes must determine the identities of their children, receive data from them, and perform the reduction operation on the received data. They then determine their parent and then send their intermediate result. In summary, this algorithm lessens the number of packets in the network and unclogs the root, but it forces much additional work in software. Other algorithms that are commonly used in collectives—such as recursive halving and recursive doubling—can similarly improve the performance of the collective, but also require that each process perform extra work.

2.2 | MPI-FPGA—overview

MPI-FPGA removes the need for this software and moves its functions into the network. Throughout this article, we refer to the CPU (leaf nodes) as the *processor* and to the FPGA-augmented switches as the *network*. *MPI-FPGA* requires no changes to the MPI API and so is transparent to application; this makes it completely portable: it can be integrated into HPC applications without requiring the programmer to have any knowledge of the underlying hardware or make any changes to existing programs. Rather, *MPI-FPGA* constructs access capabilities automatically through enhanced middleware. The design makes no assumptions about the type of end-systems being used, as it only affects data as it is routed through the switches in the network. We create new functions for each offloaded collective (e.g., *MPI-FPGA_Reduce*), and place these underneath existing MPI collective functions (see Figure 2). If the hardware supports the offload of a particular collective, then the *MPI-FPGA* replacement function is used. If a collective does not have offload support, then it is performed by the software as usual.

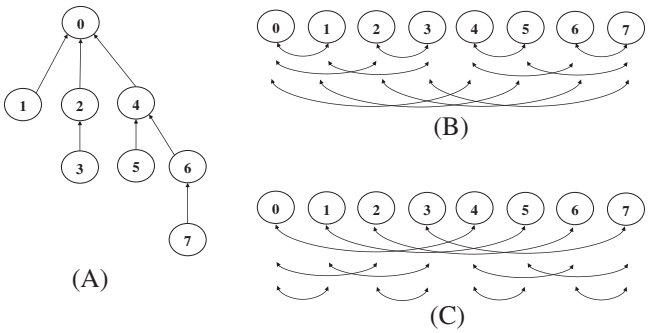

FIGURE 1 Subset of commonly used MPICH collective algorithms. (A) Binomial tree; (B) recursive doubling; (C) recursive halving

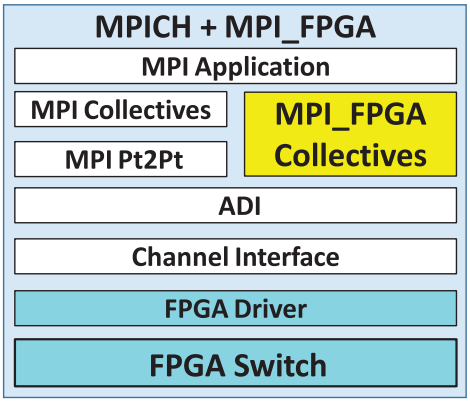

FIGURE 2 *MPI-FPGA* software stack

The basic operation of *MPI-FPGA* is as follows. Upon receiving a valid packet header from the processor, the switch begins the collective operation and performs 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 control returns to the user application as the network is responsible for progressing the message. If the calling process *does* need to receive the result—for instance, all processes in an *Allgather* operation—then the process waits until it is interrupted, which happens when the final collective results have been received and passed to the processor from the network.

In this work, we focus on blocking MPI collectives; the extension to nonblocking operations is simple. In the MPICH implementation of MPI middleware,[10] all the functionality of the abstract device interface (ADI) is maintained. This work currently uses MPICH-3.2;[32] tasks such as packing and computing predefined reduction operations are performed identically in this design. At the MPICH channel interface, we add code to transfer data into the network, with the actual FPGA hardware sitting below the channel interface (see Figure 2).

2.3 | Hardware model

For integrating *MPI-FPGA*, we consider both direct and indirect networks. For simplicity, in both cases the network switches consist entirely of FPGAs. Since the configurable parts of the design are modular, the extension to systems that combine fixed and reconfigurable logic is straightforward. For other advantages of fully configurable switches see Reference 33. For direct networks, this is identical to clusters with direct FPGA-FPGA interconnects (e.g., References 24,25). Each FPGA switch is attached to the processor through a PCIe interface and FPGAs themselves are connected directly together through a secondary network (3D-torus in this work). For indirect networks the FPGAs are proxies for high-radix switches with integrated configurable logic and are attached to multiple processors through NICs. We currently use a fat-tree network (spine-leaf architecture[34]). Switches in the direct (indirect) model are homogeneous (non-homogeneous) between leaf and spine switches.

Figure 3 shows the hardware model of the switch. To simplify the figure, transceivers and DMA engines are duplicated on the left and right. A collective control module (CCM) manages the coordination between different MPI processes for a collective operation. Inside the 2-level switch are, first, configurable gather reduction (CGR) units that perform the gather or reduction operation on incoming packets, and, then, hierarchical switches for reordering or redirecting packets. The CCM and CGR units are discussed in depth in Section 4. Processor-FPGA and FPGA-FPGA communication, as well as in-switch processing, are based on a streaming interface: data is pushed to the DMA engine, transceivers, and buffers, respectively, without having to wait for the entire packet to arrive. User logic units surround the switch for pre- and/or post-processing. Although this design is implemented on FPGAs, its portability ensures that it is independent of the type of hardware used. Units added in support of any typical network switch can be inserted or removed seamlessly.

2.4 | Communicators

Communicators are an essential MPI capability and must be supported in any useful system; yet, little work on collective offload into the network addresses it. While the default communicator is *MPI_COMM_WORLD*, where the process group is the entire set of processes,[35] many MPI programs use multiple communicators having subsets of processes. A typical example where sub-communicators are used is when the workload is partitioned among an array of MPI processes and collectives are performed on rows or columns of processes. The most common way to create these partitions is to call a function like *MPI_Comm_split*.

All communicators have a *context ID*, identifying the communicator, and a *process group* containing 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

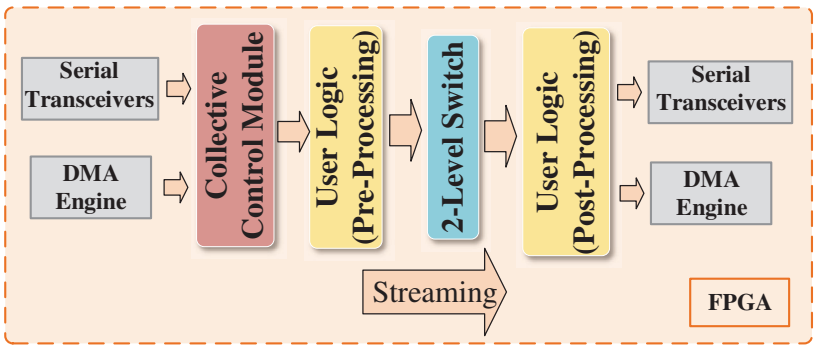

FIGURE 3 FPGA switch hardware model

large communicators, the memory consumption of these process groups leads to scaling issues.[30] To have an entire process group in FPGA memory would require storing the list of all MPI processes included in the communicator. The number of bits required would be the product of *COMM_SIZE*and *BITS_PER_PROCESS*, meaning that the resource utilization would grow linearly with the communicator size. For a system with thousands of nodes, it would require many thousands of bits in the switch for *each* of many communicators in a single application. Since high performance routing depends on having routing information close to the switching logic, replicating information about these entire process groups would quickly use up these resources, even for mid-sized clusters.

2.5 | Streaming interface

While current MPI implementations rely on bulk transfers involving large buffers,[36] we adopt a streaming interface for both processor-FPGA and FPGA-FPGA communication. The advantage of providing this support is three-fold. First, the latency of collective operations is reduced as data is processed/transferred cycle by cycle. Second, the streaming interface facilitates fine-grained communication-computation overlap. And third, it avoids the congestion caused by messages with a large payload.

To ensure the finest granularity we convert (inside the switch) processor packets into multiple small-sized network packets. Decoupling these two kinds of packets is beneficial as processor packets may contain a large payload that could block other packets from progressing. Network packets, on the other hand, are small-sized with replicated headers. The size of the packet is itself one of the fields in the new header. The structure of processor and network packets is discussed in more detail in Section 3.1. We set the phit size to 256 bits; however, the DMA engine can process two phits on each cycle.

Flow control is implemented in a standard way using back-pressure and stalling without incurring costly handshakes. There is a margin m in the FIFOs; upon reaching this threshold the upstream node is notified to stop sending packets. When the buffer gets depleted past the margin the upstream node is notified to resume sends.

3 | IN-NETWORK COMMUNICATOR SUPPORT

In this section we introduce the communicator table (CT) design, which supports collectives across any intra-communicator. This design takes advantage of the existing MPI collective algorithms in order to minimize resource utilization and latency. For simplicity, in this section and the next section we assume a direct network. The design is analogous for indirect networks; necessary modifications are given in Section 4.4.

3.1 | Communicator table (CT) design

The purpose of the CT is to manage communicator information that is needed by the CCM 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 MPI processes with which it must communicate to perform each collective. The contents of this list, for a given communicator, can be determined immediately after its initialization.

Table 1 shows a subset of MPICH algorithms for widely used collectives.[11] The three most commonly used are binomial tree, recursive halving, and recursive doubling. The ring algorithm is also common, but since its implementation is trivial we focus on the others. By being able to implement these three algorithms, we can perform all of the collectives that use them. Returning to Figure 1, for each MPI process we can identify the subset of processes with which a given MPI process must communicate. For example, process 0 must communicate with the following process set in all three

TABLE 1 Algorithms commonly used by MPICH for processing collectives[11]

MPI collective algorithms		
Reduce	Binomial tree	Recursive halving and doubling
Allreduce	Recursive doubling	Recursive halving and doubling
Broadcast	Binomial tree	Binomial tree and ring
Scatter	Binomial tree	Binomial tree
Gather	Binomial tree	Binomial tree
Allgather	Recursive doubling	Ring

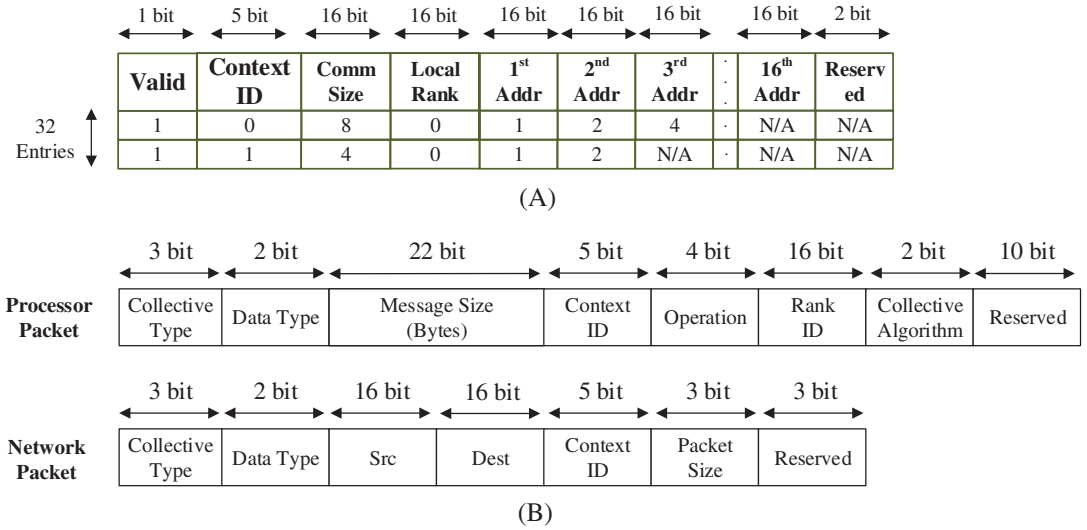

FIGURE 4 (A) CT structure, (B) processor and network packet structure

algorithms: 1, 2, 4. Storing this subset in switch memory is much more efficient than storing an entire process group as it is equal to the log of the communicator size (which can be proved directly from the properties of binomial trees).

As shown in Figure 4A, the CT holds a row for each communicator of which the current switch is a member. In each row, we store a small amount of meta-information: the communicator size, the MPI process in the communicator with which the current switch is associated, and the subset of processes with which the switch will be communicating. If there are multiple processes per node, one of the processes (the parent) is designated as the local MPI process. Each communicator entry is indexed into the table using its context ID. For any incoming packet, it is thus easy to look up its communicator as the context ID is a field in the packet header. The current design supports 32 outstanding communicators with a table size of around 1 KB; this is sufficient for up to 64 K MPI processes.

Once the switch 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. For any collective algorithm in a communicator, each MPI process will communicate with the same subset of MPI processes regardless of how many times the collective is called. Once a valid entry is loaded into the table, no updates on that entry are required until the communicator is freed.

3.2 | Communicator support constraints and design choices

Number of outstanding communicators: To maintain line-rate packet processing and low latency for table lookup we constrain the memory depth of the CT. Block RAM instances can be configured with different combinations of memory depth/width. For instance, in Xilinx FPGAs, the largest width is 512 × 72 bits, which translates to 512 outstanding communicators. To make optimal use of buffering, we currently use 32 queues which translates to supporting 32 outstanding communicators. However, this is a parameter that can be modified to support a smaller/larger number of outstanding communicators.

CT size: Assuming that the maximum supported number of MPI processes and the number of outstanding communicators are *MAX_R* and *N_C*, respectively, we can derive the following equation for the size of CT (according to Figure 4A):

$$Size = (Log(MAX\_R) \times (Log(MAX\_R) + 2) + 8) \times N\_C \quad (1)$$

where *OC* represents the CT depth and the other terms are used to compute the width. Assuming the 512 × 64 memory instances of Xilinx FPGAs (the extra 8 bits can be used for error correction), we limit the number of memory instances to achieve *MAX_R*, while making sure that it is small enough to make the packet overhead negligible. We currently use *MAX_R* of 64K (16 bits wide), which leads to five block RAM (BRAM) instances (in current generation Xilinx FPGAs).

Processor and network packet structure: Figure 4B shows the structure of these two kinds of packets. *Message size* in the processor packet is replaced with the *packet size* in the network packet which requires fewer bits. *Packet size* stores encoded values (power of two numbers). The *Dest* field is computed in the switch (by a route computation unit) and is 16 bits wide (equal to the bitwidth of the process rank). The context ID field is 5 bits wide as the maximum number of outstanding communicators is 32. The *Operation* field determines the supported operations f (sum, min, max,

and logical operations) and the width is set to 4. Finally, 2 bits are reserved for supporting different collective algorithms. Generally, the bitwidths of the fields are chosen to minimize packet header size; for the processor and network packets these are 8 and 6 bytes, respectively.

3.3 | CT entry creation

When a new communicator is created in software, the switch needs a way of obtaining the CT entry from the processor. A new CT entry creation function has been inserted at the end of each MPICH communicator creation function. This added function checks whether an MPI process is a member of a new communicator and, if so, calculates the subset of MPI processes with which it communicates. This requires that, for each communicator creation function call, the processor calculates the physical addresses of the subset of MPI processes that will be stored in the table entry. Once the new entry is created, the switch can handle new collectives occurring within this communicator. The addresses are then packaged alongside communicator metadata (see Figure 4A) to be sent to the switch. Although this operation does lead to a small amount of overhead in creating communicators, this overhead is only paid for once during communicator creation.

In addition to the CT, there is a forwarding table in each switch which stores a dictionary of global ranks (used by *MPI_COMM_WORLD*) as the keys with the processor physical addresses as the values. When there are multiple processes in a node, multiple global ranks associated with the processor appear in the forwarding table. This information is collected during `MPI_Init` (called only once at the start of the program). It is then successively used in the routing phase. Since each new communicator could assign a new rank for the same process, storing the information in the forwarding table for each new communicator has the potential to exhaust memory resources. Hence, the CT entry creation function sends the global ranks of the processes instead of the actual rank of that specific communicator. As a result, the switch deals with the global MPI process ranks for packet forwarding while the software on the processor considers the actual MPI process ranks (based on the communicator).

4 | IMPLEMENTATION

4.1 | Overview

The base design is a standard virtual channel router (see Figure 5 and References 33,37) extended with the proposed streaming interface. It uses the classic four stage pipeline:[38] route computation, virtual channel allocation, switch allocation, and switch traversal. In this example it is designed to be used in an FPGA cluster interconnected in a 3D torus and so has six input and six output ports, each connected to serial transceivers.

To minimize back-pressure from switch to processor, the size of input FIFOs should be sufficient to tolerate the latency of the collective tree. This happens if the tree is deep and the root has to wait a considerable amount of time for the leaf processes. We calculate the minimum buffer size as in Equation (2):

$$min\_buf\_size = \left(m + (d+q) \times \left(\frac{L}{T\_clk}\right)\right) \times Ph \tag{2}$$

where m is the flow control margin, d is the maximum depth of the tree-based collective algorithm (accounting for the number of nodes used in the FPGA cluster), and q is a constant (set to 2) to provide slack for other operations to proceed. L, T_clk, and Ph are the link latency, clock period, and phit size, respectively. The input FIFO size in the current design is 48 KB.

The MPI offload support is designed to keep the overall design modular: the accelerator architecture is portable to any other standard router. The two key modules are the CCM and the in-switch collective support module. The former determines new forwarding and multicast destinations for collective packets; it also contains the communicator support. The module is placed before the router so that the packets' output ports can be calculated during the route computation stage after it has been assigned a new destination. The latter is integrated into the L1 switch (see Section 4.3) and is used for reordering, redirecting, and in-switch processing. Details follow in the next subsections.

4.2 | Collective control module (CCM)

The CCM (Figure 6) performs all of the algorithmic work found in the software of *MPI_Reduce*, *MPI_Allreduce*, *MPI_Bcast*, *MPI_Scatter*, *MPI_Gather*, *MPI_Allgather*, and *MPI_Reduce_scatter*, as well as any other collectives implemented in the future. When packets enter the router, they first go through the CCM. If they are not part of a collective operation, or are not destined for the current process, 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 process, then the CCM uses the CT to determine new destinations for the packet.

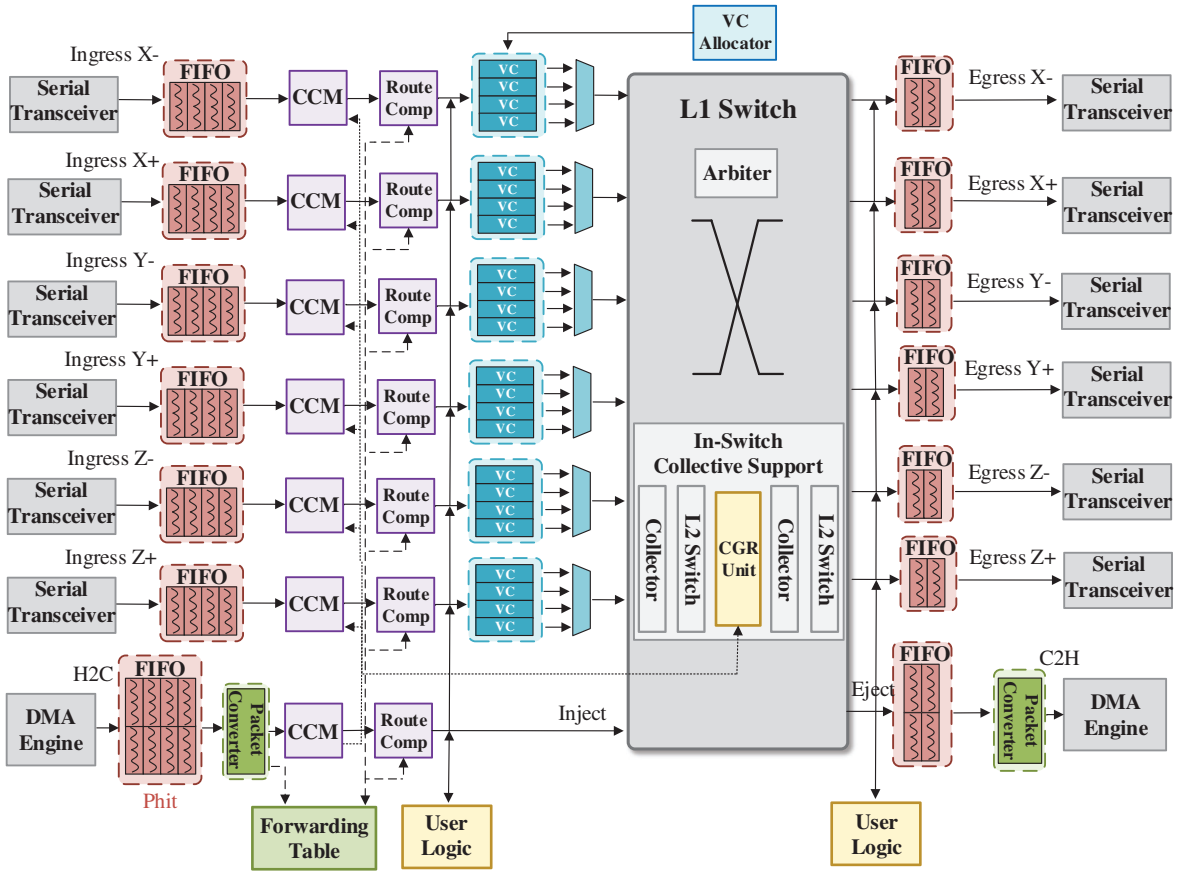

FIGURE 5 *MPI-FPGA* switch with collective support: Collective control module (CCM) and configurable gather/reduction (CGR) unit

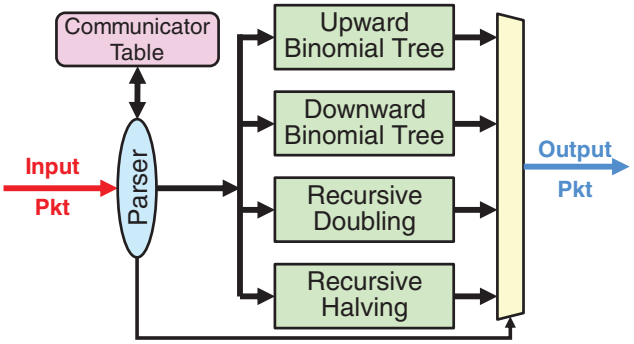

FIGURE 6 Collective control module

In order for the CCM to determine which collective a packet is a part of a collective, an opcode field has been added to the packet header. With this *MPI-FPGA* can perform work for each collective algorithm in parallel and then use the opcode to decide which algorithmic results to use for the packet (see Figure 6). Within each of these algorithm blocks, *MPI-FPGA* performs computations using input from the packet header and the CT entry. For a reduction, the router calculates the parent node to send the packet to; or, for a broadcast, all of the child nodes to multicast the packet to.

The CT also eases the computation required to calculate these destinations. When a packet needs to be sent to multiple destinations, these are also adjacent in the table entry. For multicast a bit vector is used to keep track of these destinations; this results in much less work than if destinations were calculated repeatedly. Once a packet passes through the CCM, it is passed to the route computation stage (in the routing pipeline) where its output port is calculated.

As in MPICH-3.2, the implementation also supports multiple algorithms for the same collective operations. The choice of algorithm is often determined by packet size. In *MPI-FPGA* we support algorithm selection by adding bits to the packet header opcode field.

4.3 | Two-level switch

As shown in Figure 5, the overall switch design consists of hierarchical switches (L1 and L2) for directing packets. The L1 switch directs packets to the correct output port and is followed by output FIFOs. The current design supports a round-robin arbitration policy. Note that the arbiter is disabled for the *Eject* port; once the communication between the switch and the processor is established, packets in other input ports do not contend for the *Eject* port as the network packets are again converted into processor packets. To support processing of the collectives there is an in-switch collective support module that is seamlessly integrated to the L1 switch. This module performs the gather-type (*Gather* and *Allgather*) or reduce-type (*Allreduce*, *Reduce*, and *Reduce_scatter*) operations. If network packets are identified as part of one of these collectives (by their opcode field), they are transferred to the in-switch collective support module. It has two collectors, two L2 switches, and CGR unit(s). The first collector is responsible for waiting for all packets corresponding to the child MPI processes. The first L2 switch and the last collector are only used for gather-type operations. The former reorders incoming packets according to their sender's rank, while the latter serializes the gathered data. Finally, the last L2 switch is responsible for directing the CGR outputs to the correct switch output port.

The CGR unit consists of configurable vectorized aggregation logic units (AgLUs), as shown in Figure 7. Each vectorized AgLU is cascaded to match the number of switch ports to perform concurrent reductions with full pipelining. The aggregation is made up of two separate paths with vectorized AgLUs chained together which are combined at the end. Each vectorized AgLU is 256-bits wide (in this design) which can accommodate multiple AgLUs based on the bitwidth of the datatype being used. The CGR unit also has gather and reduction tables that are indexed and capable of supporting multiple gathers and reductions, respectively, and with different communicators. There is a *counter* field inside the tables to track the number of processed phits; this is then compared with the *count* field (total number of phits according to the message size). When the threshold is reached, a completion notification is sent to the DMA engine. The CCM configures the value of the *count* field in the reduction (gather) table according to the processor packet header. *Count* and *counter* fields have the same bitwidths as the message size in the processor packet header. There are six columns for the reduction (gather) matching the switch radix for the current design (3d-torus), each with a bitwidth equal to the phit size.

To support multiple simultaneous reductions (gathers), there are *SA* (simultaneous aggregations) number of CGR units. Clearly, *SA* can be equal to the radix of the switch; but it also can be fewer depending on the available hardware resources (discussed in Section 6.1). There are $R\times4$ AgLUs (64-bit) for each CGR unit (with a double data type), where *R* is the radix of the switch. This results in $SA\times R\times4$ AgLUs in the L1 switch.

The arithmetic unit is constructed using standard methods including the use of vendor IP. The current design supports sum, min, max, XOR, AND, and OR but is trivially extendable for other operations, including user specified functions.

This approach is flexible enough to support multi-user processing. When the number of jobs is small—not larger than the number of AgLUs in a VAgLU—it is possible to divide phit size channels (each arrow in Figure 5), into 64 bit sub-channels and assign each of them uniquely to each job. In this case, there is no resource/time sharing involved in the switch and each AgLU is mapped to a single job. When there are more jobs, it is possible to employ time slotting.[39,40] In either case, if the computation requirements of a certain application surpass the operations that are supported by the AgLUs, or there is a need to add user logic for inline pre/post-processing (see Figure 5), it is possible to take advantage of partial reconfiguration.[41] For the former case, partial reconfiguration does not disturb other existing jobs but, for the latter, it would introduce minimal idle time (depending on the amount of new logic) as only the AgLUs and the user logic would be reconfigured, rather than the entire design.

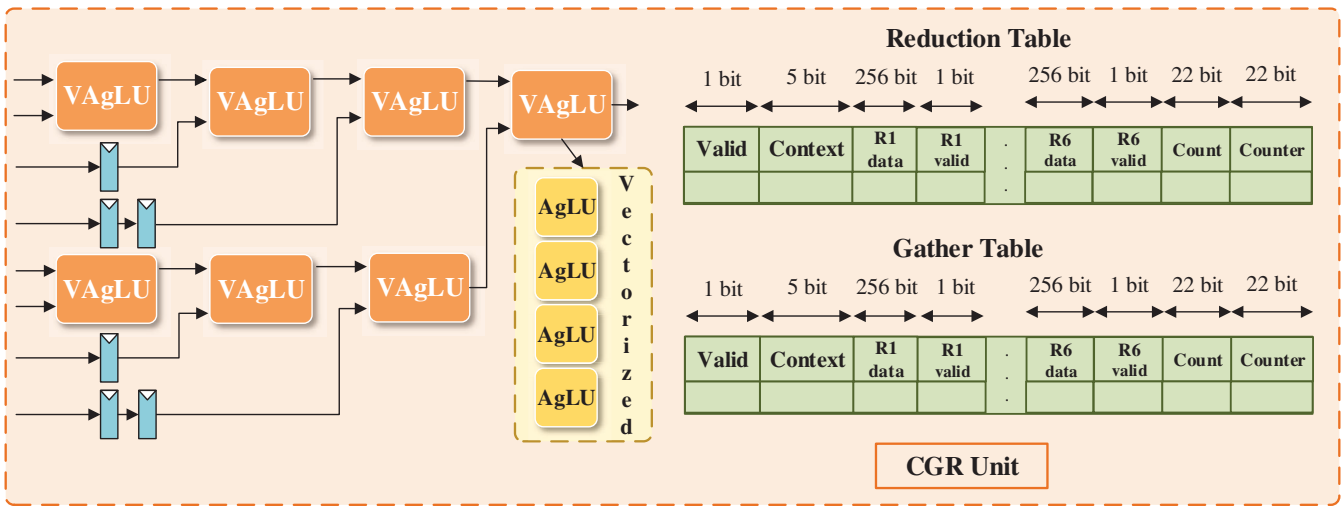

FIGURE 7 Configurable gather/reduction (CGR) unit: Aggregation has two paths with vectorized AgLUs chained together and combined at the end

4.4 | Considerations for indirect networks

The *MPI-FPGA* switch design for indirect networks follows the same overall design as Figure 5 but with some differences:

- In the direct model, each FPGA card is attached to a processor through a PCIe interface. In the indirect network model, FPGA switches are attached to the processors through network ports.
- For the indirect model, storing information in the forwarding table is a two-step process. During MPI_Init, a dictionary—where the keys are the global process IDs and the values are the port numbers—is sent from the processors to each of the corresponding leaf switches and then from leaf switches to spine switches. As a result, each spine switch maintains a dictionary of all the processors within the system inside its forwarding table;
- The *count* and *context ID* fields in the reduction/gather tables of the spine switches are configured by sending the information from the leaf switches to the spine switches based on the collective operation shown in the incoming packet header;
- For the leaf switches, the number of packet converter modules is equal to the number of downlink ports.

5 | PERFORMANCE ANALYSIS OF IN-NETWORK COLLECTIVE OFFLOAD

In order to develop a general understanding of how this approach could improve the performance of MPI collectives and the conditions that need to exist for this approach to benefit application performance, we consider a LogGP model.[42] L, o, g, G, and P represent the latency, overhead, gap between messages, gap per byte, and number of processors, respectively. As is standard in reports on hardware implementations of communication operations, including collectives, we assume that all of the processes start at the same time with no skew. As it has been stated previously, substantial process skew necessarily diminishes the performance benefit of any aggregation algorithm.[9] Skew is discussed further in Reference 43.

We compare two models: a baseline with no compute-in-the-switch and the new one where the switch is able to perform the computation, multicast, and reordering.

Since the total latency of an MPI operation is determined by the slowest process; that is, the maximum latency of any process, processes having different roles during the collective need to be considered. For *MPI-FPGA*, we consider the latency of two processes in an MPI_Bcast for a binomial tree (see Figure 1A): one receiving a number of messages (e.g., process 7) and one sending multiple messages consecutively (process 0).

We first derive expressions for the time spent on the second process type. Let $T_{Bl,i}$ and $T_{ls,i}$ denote the time needed for the ith process to finish the MPI operation in baseline and in-switch models, respectively.

$$T_{Bl,P-1} = \lfloor log(P) \rfloor \times (O_s + (k \times G) + L + O_r) \tag{3}$$

$$T_{ls,P-1} = O_s + (k \times G) + L + O_r \tag{4}$$

We distinguished the overhead time on the sender side (O_s) from that on the receiver side (O_r); these are the times it takes for the processor to transmit and receive a message, respectively. Note that $\lfloor log(P) \rfloor$ is reduced to 1 in Equation (4). These equations indicate that the *MPI-FPGA* approach effectively eliminates the overhead time (O_s and O_r) for intermediate MPI processes since communicator support is offloaded into the FPGA fabric. Also, the above model suggests that the in-switch approach unlocks higher performance for collectives involving many-to-many communication patterns (Allreduce, Allgather, Reduce_scatter, and Barrier) as the number of times that CPUs are bypassed is higher in these collectives. The in-switch approach also scales better with the number of nodes. Finally, note that the term G in the baseline is greater than for in-switch: the cost of the transport protocol is high compared to that of the light-weight transport engine.

We now analyze the time spent on the root process (process 0):

$$T_{Bl,0} = O_s + ((\lceil log(P) \rceil - 1) \times max\{g, O_s\}) + (\lceil log(P) \rceil \times k \times G) \tag{5}$$

$$T_{ls,0} = O_s + (k \times G) \tag{6}$$

In the in-switch model the g term is eliminated by the pipelining communication. For example, for MPI_Bcast the message is sent from the processor to the switch only once where it is then multicast to the different MPI processes. For MPI_Scatter, different messages are lumped into a large message. Consequently, for the in-switch model $g=0$ and assume only G for both short and long messages. Clearly if the message size is small, then the G terms in all of the above equations are irrelevant. One observation is that, for the in-switch model, the coefficient $\lceil log(P) \rceil$ is reduced to 1 in Equation (6) as the switch benefits from in-network multicast. Another observation is that a multi-port switch attached to each node in the direct network reduces the *effective* G for intermediate MPI processes as multiple messages can be sent simultaneously.

One major benefit of the in-switch model is in-network reduction; *inline* high-throughput reduction is performed in the switch, which eliminates the need to perform the reduction in the CPU. This is especially beneficial for large messages. Assuming *n* MPI processes involved in the reduction collective, in the baseline model the operation could be performed *n-1* times. In the in-switch model nearly all of these operations are eliminated owing to the pipelined, high-throughput AgLUs. Another advantage of the in-network reduction is reducing network traffic and the number of data copies in the processor. From the above discussion, we infer that the application characteristics that maximize *MPI-FPGA* benefit are, first, a large number of MPI collectives, especially Allreduce and Allgather, and, second, reductions with large message sizes.

6 | EVALUATION

We implemented and tested *MPI-FPGA* on a two-node FPGA system using Xilinx Alveo U280 boards and the standard Xilinx development tool suite. Each board exposes two 100 Gb/s QSFP128 interfaces and a PCIe Gen3 ×16 interface. Vendor-provided IPs (Xilinx QDMA with AXI4-Streaming interface and Aurora) were used to implement the DMA engine and transceivers. The design is coded in *Verilog HDL*. We have provided the FPGA-to-FPGA latency and peak bandwidth of the Alveo U280 card in Table 2. The bandwidth starts to saturate at a message size of just under 1 KB. The information in this table is used during the simulation, which is discussed in the next paragraph.

Given the challenges of HDL coding, an efficient cycle-accurate simulator is essential for exploring a large number of nodes. The simulator used in this article is an version of that described in References 33,44 updated by changing the transceiver parameters collected from the two-node testbed (Table 2). The simulator is implemented in C++; every hardware module in the RTL model has a corresponding class in the simulator. These classes are organized in the same hierarchical structure as the RTL model. To give the cycle-accurate simulator good extensibility, we define an interface standard for all hardware modules. We adopt a producer-consumer model with every module being both a producer and consumer. The simulator has been validated with respect to the RTL code for the two-node FPGA system; the behavior of RTL simulation matches the simulation.

For cluster tests, we target Xilinx XCVU13P FPGA devices. The performance results for the FPGA cluster are obtained from a cycle-accurate simulator. Resource utilization is reported using *Vivado Design Suite 2019.2*. The operating frequency of the current implementations is 250 MHz. For the CPU reference, benchmarks were run on the Stampede2[45] Skylake (SKX) compute cluster, accessed through XSEDE, with 48-cores per node (2 sockets) 2.1 GHz Intel Xeon Platinum 8160 CPUs, and a 100 Gb/s Intel Omni-Path (OPA) network (fat tree topology). We used Intel MPI 18.0.2 as an Intel-compatible MPI compiler and launcher as recommended for the TACC Stampede2 cluster. We found that it usually gives a better performance compared to MPICH 3.2.

To assess the efficiency of *MPI-FPGA*, we compare the performance with respect to the CPU cluster for *Allgather*, *Allreduce*, *Broadcast*, *Gather*, *Reduce*, *Reduce_scatter*, and *Scatter* operations using the OSU micro-benchmarks (v5.6.2).[46] The study investigates a range of message sizes. For FPGA-based indirect networks, we considered three switch designs: indirect-8, indirect-16, and indirect-28, which denote switches with radix-8, -16, and -28, respectively. Radix-28 was the highest radix we could implement on the XCVU13P FPGA according to the number of available GTY transceivers.

6.1 | Resource utilization

FPGA resource consumption is shown in Table 3 for four different designs (direct and indirect networks). As discussed previously, each AgLU supports a different set of operations. Depending on application requirements, the user can remove the support for unused operations or add user-defined operations; these actions decrease or increase resource utilization, respectively. The current result is based on using floating point add

TABLE 2 Inter-FPGA latency and bandwidth for Xilinx Alveo U280

Parameter (FPGA-to-FPGA)	Latency (ns)	Peak bandwidth (Gb/s)
Value	440	95.9

TABLE 3 Resource usage on Xilinx XCVU13P FPGA devices

Design	LUT	FF	DSP	BRAM	URAM	#SA	#AgLU
Direct	401,852 (23.3%)	448,117 (13.0%)	1441 (11.7%)	454 (16.9%)	6 (0.4%)	6	144
Indirect-8	700,409 (40.5%)	1,062,680 (30.7%)	2561 (20.8%)	523 (19.4%)	0 (0%)	8	256
Indirect-16	1,408,301 (81.5%)	2,126,184 (61.5%)	5121 (41.7%)	987 (36.7%)	0 (0%)	8	512
Indirect-28	1,551,853 (89.8%)	2,349,600 (68.0%)	4481 (36.5%)	1543 (57.4%)	0 (0%)	4	448

operation (double-precision). Overall, BRAM utilization is dominated by serial transceivers, input FIFO buffers, and the DMA engine, while the L1 switch accounts for a large fraction of DSP, LUT, and FF utilization.

According to Table 3, the overall utilization increases with the radix of the switch. For all designs except indirect-28, the number of SAs is equal to the number of ports. There the number of SAs needed to be reduced in order to fit the entire design onto the FPGA. This is because the number of AgLUs for each CGR unit increases with the radix of switch. The critical resource is LUTs with the CGR units constituting a large fraction of total LUT utilization.

6.2 | Performance of MPI collectives

We evaluated *MPI-FPGA* for seven different MPI collectives using the OSU Micro-benchmarks. For all collectives, double precision floating point was used. The results were averaged over 1000 iterations (with 200 warm-up iterations) and five different runs (different node allocations).

6.2.1 | Overall collective latency

Figures 8 and 9 show the simulation results of MPI and *MPI-FPGA* collectives for small (4 Bytes to 4 KB) and medium-to-large message sizes (4 KB to 4 MB) for 32, 64, and 128 nodes using the OSU benchmarks. The reported average latency is the average time it takes for the processes to finish

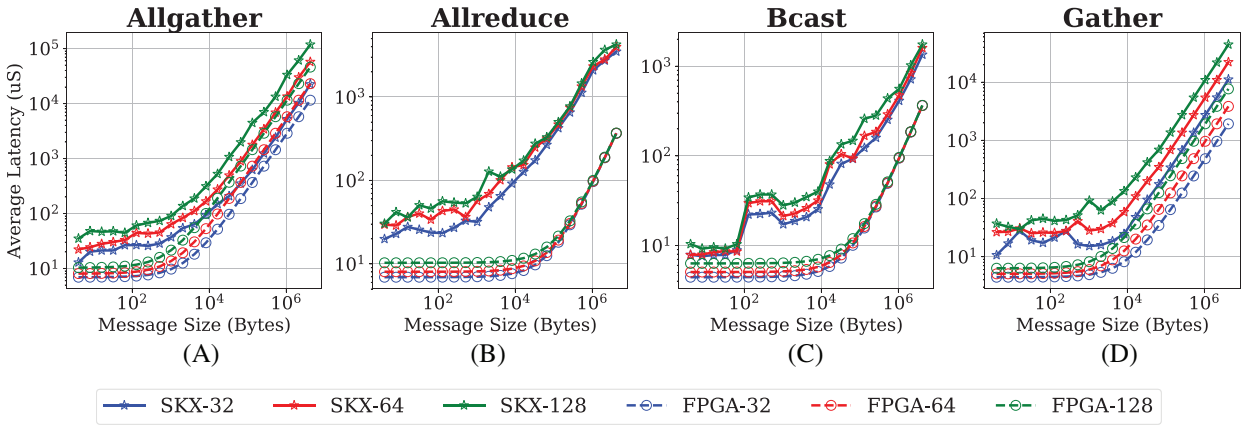

FIGURE 8 MPI CPU cluster (SKX) versus *MPI-FPGA* execution times for 32, 64, and 128 nodes: (A) osu_allgather, (B) osu_allreduce, (C) osu_bcast, and (D) osu_gather

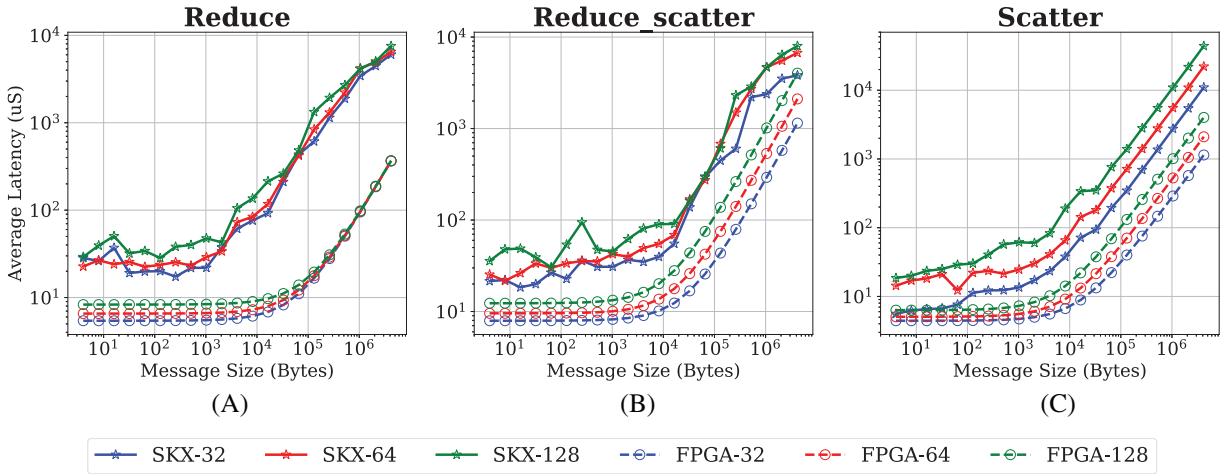

FIGURE 9 MPI CPU cluster (SKX) versus *MPI-FPGA* execution times for 32, 64, and 128 nodes: (A) osu_reduce, (B) osu_reduce_scatter, and (C) osu_scatter

the operation. Processor-FPGA communication latency is included in the time. To isolate the impact of the design under study, for example, from contention at the PCIe interface, we focused simulations with one process per node.

One of the advantages of *MPI-FPGA* is that utilization of the application layers in the network stack (such as MPI) can be bypassed for the nodes associated with root and intermediate processes because communicator support is offloaded, while reduction operations (if any) can be performed by a network switch. Although having a low-latency network topology (such as a fat tree) for the reference CPU cluster (as opposed to 3D torus for the FPGA cluster) can offset the aforementioned benefits, we observe that *MPI-FPGA* has a higher overall performance. As is evident from Figures 8 and 9, *MPI-FPGA* speedup relative to the CPU cluster is higher for *Allreduce*operation, since a greater number of nodes corresponding to the intermediate processes are involved and data volume is reduced during transfer. For the Reduce_scatter operation, the switch corresponding to the root process is able to perform the reduction on incoming data received by ingress ports and then scatter the result directly. This effectively bypasses the processor as it eliminates the need to transfer data to the processor to perform reduction and scattering.

To view the results from a different perspective, Figure 10 shows the average speedups for each of these collectives on 32, 64, and 128 nodes. The geometric mean is used to summarize *MPI-FPGA* speedup ratios with respect to different message sizes.[47] The speedup ratio ranges from 2.0× to 32.6×.

MPI-FPGA achieves higher performance than the CPU cluster for both small and medium to large message sizes. The central reason is that *MPI-FPGA* does compute-in-the-network. This bypasses the processor in the nodes corresponding to the root and intermediate processes, which in turn reduces the latency, especially for short messages. Also, *MPI-FPGA* utilizes a streaming interface with fine-grained communication as opposed to processor-based bulk transfers involving large buffers; this helps for long messages.

With respect to problem size, of note is that the *MPI-FPGA* speedup is maintained as the number of processes grows; this indicates the expected benefit for larger systems through reduced network traffic. Interestingly, the speedup usually increases for larger systems, especially for medium-to-long message sizes, showing that the advantage of *MPI-FPGA* scales. According to the results in Figure 10, the speedup in medium-to-large messages is higher than for small messages (except Allgather) due to the efficient streaming-based interface. An exception is Allgather where there is more wait time and traffic, the latter since the volume of data expands as it traverses the network.

In order to characterize the latency variation of MPI collectives across different iterations and different runs, we have tabulated the standard deviation (std), maximum, and minimum latency for messages of size 1 KB and 1 MB on 32, 64, and 128 nodes (SKX cluster), see Table 4. As expected MPI collectives for large messages (1 MB) have larger variation than those of small messages (1 KB) and the deviation usually increases as we scale out.

6.2.2 | Indirect network study

Figure 11 shows the average latency of two MPI collectives, Allgather and Allreduce, for small message sizes (4 Bytes to 4 KB). Four different kinds of networks are considered: direct, indirect-8, indirect-16, and indirect-28. These are a direct network with 3D torus topology and indirect networks with radix-8, -16, and -28 switches, respectively. The switches in the indirect network have a default over-subscription of 3:1. Overall, use of an indirect network reduces the average latency compared with the direct network. As expected, switches with higher radix result in more performance

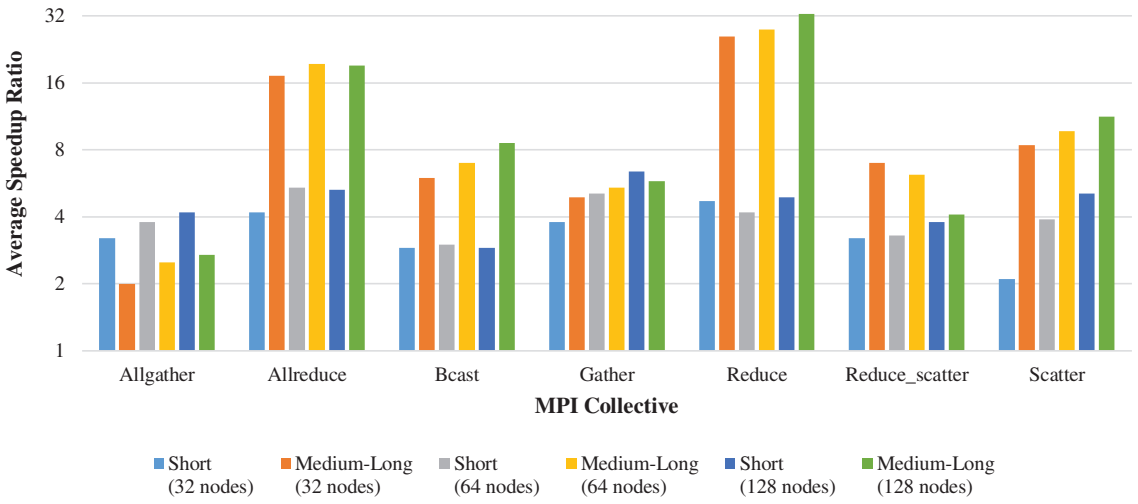

FIGURE 10 Average *MPI-FPGA* speedup ratios for OSU benchmarks running on 32, 64, and 128 nodes of Stampede2 for short and medium to long messages

TABLE 4 MPI collective latency variation on the Stampede2 compute cluster for message sizes 1 KB and 1 MB on 32, 64, and 128 nodes

	32 nodes						64 nodes						128 nodes					
	1 KB			1 MB			1 KB			1 MB			1 KB			1 MB		
MPI collective	Std	Max	Min	Std	Max	Min	Std	Max	Min	Std	Max	Min	Std	Max	Min	Std	Max	Min
Allgather	6.7	48.8	27.1	250.3	5466.9	4645.1	12.2	87	44.8	808.5	14164.9	11,327	17.2	130.2	66	990.7	26,439	23233.2
Allreduce	6.3	46	21	81.4	2224.9	1896.1	10.7	74.1	36.9	182.9	2614	1940	11.6	86.1	33.2	291.4	2985	1997.9
Bcast	0.6	18.1	16	39.5	493	349	2.1	26.2	17.8	35.2	546.2	412	3.4	36	22.8	116	816.8	402.9
Gather	4.2	26.9	9.8	5	2739	2712	4.4	41	21	7.4	5465	5425.9	25.9	151.1	44.1	35.3	11,007	10,847
Reduce	6.8	39.1	11.9	151.6	3720.1	3089.9	11.2	57	9.7	217.3	4524.9	3695.1	12.2	72	21.9	255.8	4709.9	3807
Reduce_scatter	6.6	38.2	16.9	113.4	2577.9	2033.9	10.1	62	25	187.2	4908.9	4150.1	8.4	61	31	203.6	4904.2	4195.7
Scatter	3.1	24.8	11	12.1	2781.8	2729.1	3.9	34.1	21.9	16.6	5549.2	5483.8	10	95.1	45.8	20.8	11,018	10,937

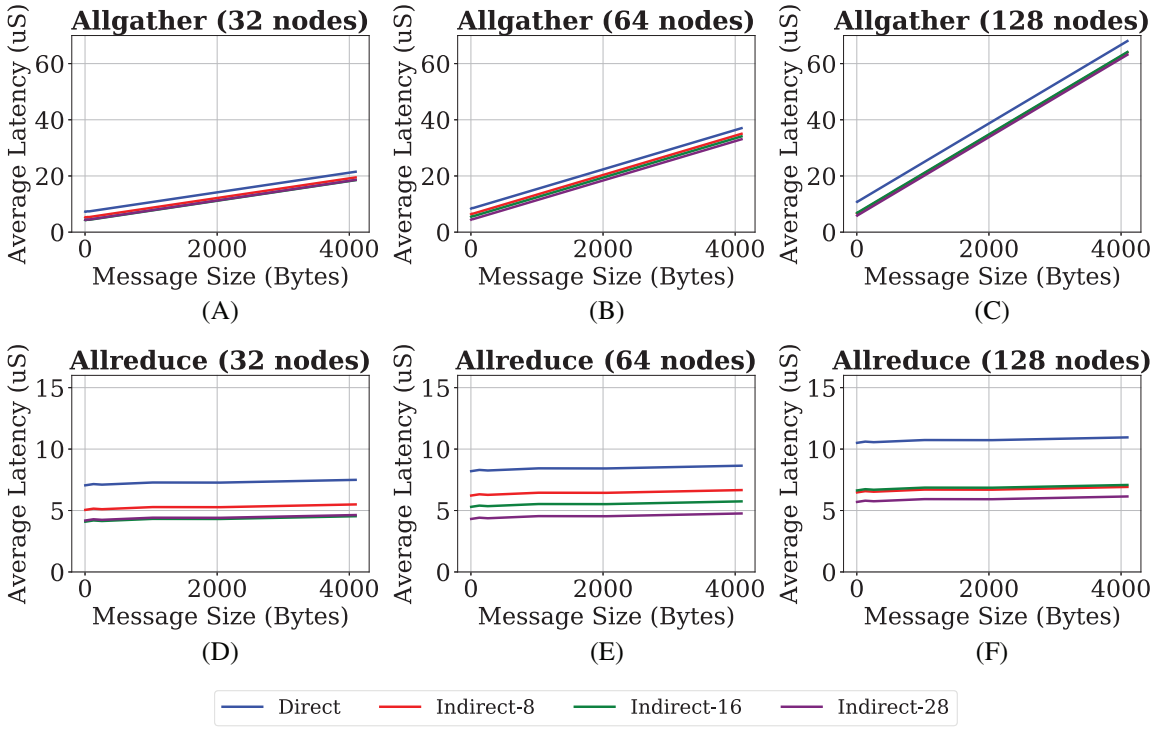

FIGURE 11 Performance comparison for short messages of *MPI-FPGA* with direct and indirect networks on 32-, 64-, and 128-node systems for Allgather and Allreduce operations. (A) Allgather (32 nodes), (B) Allgather (64 nodes), (C) Allgather (128 nodes), (D) Allreduce (32 nodes), (E) Allreduce (64 nodes), (F) Allreduce (128 nodes)

benefit. There are cases, however, in which this does not hold, for example, in (Figure 11F). There, having radix-16 switches could not reduce the number of hops compared to the radix-8 switches. This slightly affects the performance as higher radix switches are more complex and have more AgLUs. It should be noted that, for Allgather, average latency increases more rapidly compared to Allreduce as packets are combined throughout the network.

Of interest is the comparison of *MPI-FPGA* with the commercial ASIC-based version of in-switch offload of collectives from Mellanox. Unfortunately gaining access to SHArP-based systems is still extremely challenging. SHArP only supports Allreduce and Barrier collectives[9] while the reconfigurable approach embraces a diverse and extensible set of collectives. Although results published in SHArP may not be directly comparable to those described here, as they are independent sets of experiments, it appears that the *MPI-FPGA* approach is competitive with SHArP and could outperform SHArP beyond a certain message size.[9] One likely reason is that reduction of messages that are larger than SHArP hardware maximum message size are performed by posting multiple reductions. In contrast, in *MPI-FPGA*only one request is sent. Also, *MPI-FPGA* has an efficient streaming interface where large processor packets are converted internally to multiple small network packets. There is a newer version of the

SHArP protocol which is based on streaming aggregation interface, but it is not optimized for small message size[48] and the latency is higher than that of the original work.[9]

Figure 12 summarizes the average speedup of indirect network switches over the baseline direct network switch (from Figure 11) for short messages. The geometric mean is used to summarize speedup ratios over different message sizes.[47] The overall speedup ratio is between 1.2× and 1.9×.

As a further evaluation we compared *MPI-FPGA* for an indirect network based on radix-28 switches with a SHArP-based MVAPICH2-X implementation.[49] Figure 13 depicts the scaling of *MPI-FPGA* with radix-28 switches (FPGA-indirect-28) and the SHArP-based solution in the MVAPICH implementation (MVAPICH2-X-SHArP) for Allreduce with 2048 byte messages. The number of nodes was varied from 2 to 128 with one MPI process per node. The data for MVAPICH2-X-SHArP is collected from Reference 49 running on TACC Frontera HPC system. The figure demonstrates that the FPGA-based approach may be able to outperform high-end CPU clusters optimized with SHArP technology.

6.3 | MiniApp-level benchmarking

To show the efficacy of *MPI-FPGA* in application performance, we have benchmarked various HPC kernels and pseudo applications using NAS parallel benchmarks (NPB)[50] as well as two MiniApps from the Mantevo project (miniFE[51] and HPCCG[52]).

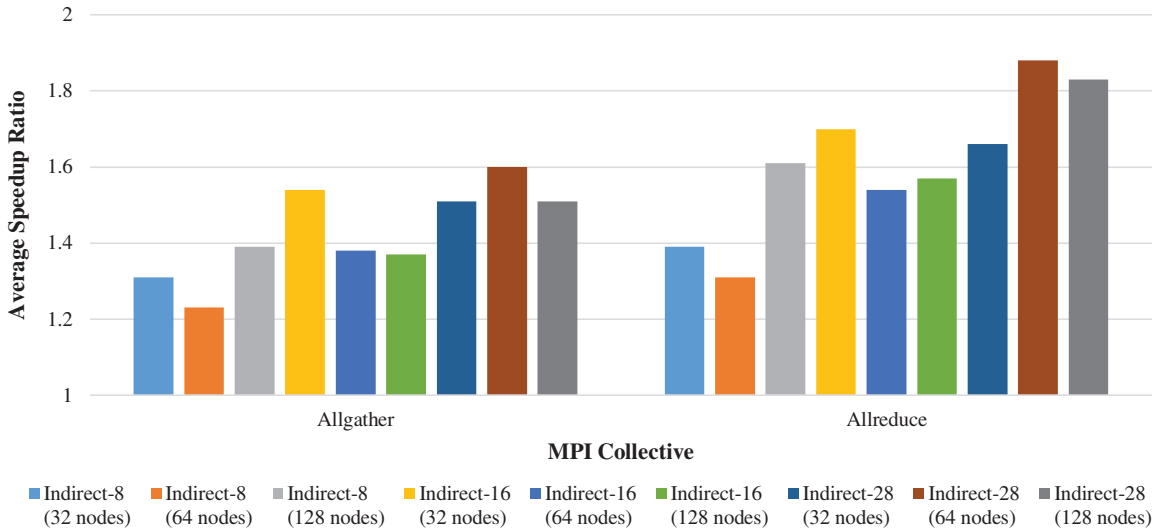

FIGURE 12 Average speedup ratios of indirect network switches over direct network switches for short messages

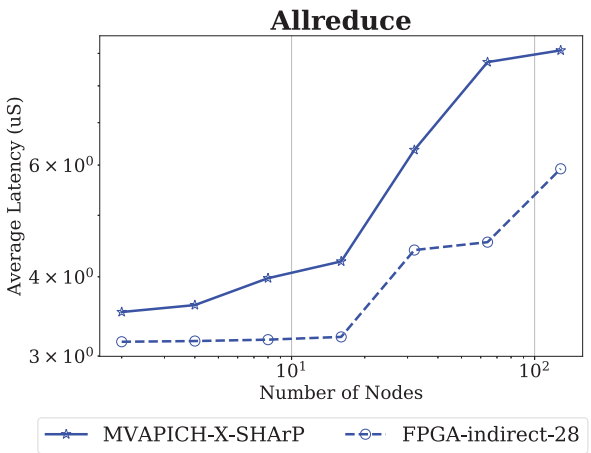

FIGURE 13 Scaling comparison of the Allreduce operation between MVAPICH2-X-SHArP[49] and *MPI-FPGA* with radix-28 switches for 2048 bytes up to 128 nodes

TABLE 5 Performance comparison of NAS parallel benchmark, miniFE, and HPCCG, for MPI CPU cluster (SKX) versus MPI-FPGA for 64 and 128 nodes

Benchmark/ application	64 nodes			128 (121[a]) nodes		
	SKX (ms)	FPGA (ms)	Improvement (%)	SKX (ms)	FPGA (ms)	Improvement (%)
CG	40.3	39.7	1.45	69.9	68.5	2.06
IS	151.7	150.3	0.89	88.3	85.7	2.96
MG	536.7	532.4	0.80	303.3	294.5	2.92
SP[a]	708.3	707.7	0.08	626.7	625.5	0.18
miniFE	32886.9	32868.2	0.06	71052.9	71037.8	0.02
HPCCG	24890.9	24858.9	0.13	58268.6	58247.4	0.04

[a] For the SP benchmark the number of processes must be a perfect square.

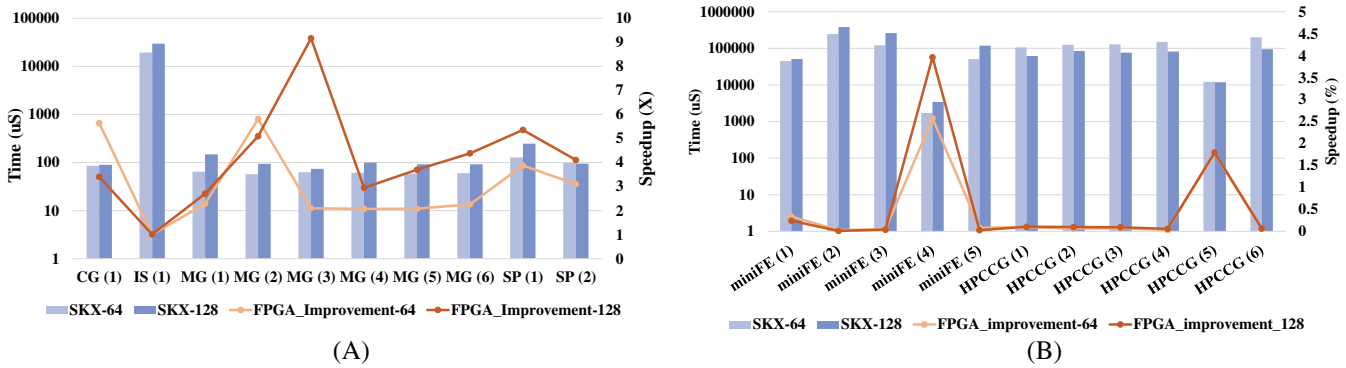

FIGURE 14 CPU cluster (SKX) execution time and *MPI-FPGA* speedup of Allreduce (Reduce) collectives used in the applications under study. Each collective instance is evaluated separately, for example, with MG having six instances of Allreduce. (A) NAS parallel benchmarks; (B) miniFE and HPCCG

When running the benchmark codes on the FPGA cluster we found that the addition of barriers simplified the instrumentation. To ensure that we compared the FPGA cluster results with the best possible baseline, we also created versions of the baseline codes with barriers. For the baseline cases, original and "barrier," we found that the performance of the two versions was indistinguishable.

For NPB, class A is used for CG and SP, the others (IS and MG) are benchmarked with class C. The results are averaged over five runs. Table 5 compares NPB, miniFE, and HPCCG for the CPU cluster (SKX) and *MPI-FPGA* for 64 and 128 nodes. Analyzing the NPB results, it can be inferred that *MPI-FPGA* improvement increases scales with the number of nodes. This aligns with the discussion of the LogGP model in Section 5. For MG, *MPI-FPGA* achieves considerable improvement (about 2% on 128 nodes) as there are a large number of MPI collective calls in this benchmark. For IS there is an MPI_Allreduce with a large message size in which *MPI-FPGA* benefits from in-network computing (about 3% improvement on 128 nodes).

For miniFE and HPCCG, we used 48 OpenMP threads on SKX as this number of threads yielded the highest performance according. The problem size used for miniFE application on 64 (128) nodes is $512 \times 256 \times 64$ ($512 \times 512 \times 64$). When scaling up, the problem size per processor is kept fixed (weak scaling). Our approach provides satisfactory speedup as one of the Allreduce collectives is called for 400 times. Weak scaling is measured for the HPCCG benchmark as well (the grid dimension on each process is $100 \times 100 \times 100$). Similar to miniFE, one of the Allreduce collectives is called for a large number of times (298).

Figure 14 shows the CPU cluster (SKX) execution time and *MPI-FPGA* speedup for Allreduce (Reduce) collectives used in (a) NPB and (b) miniFE and HPCCG. Because each application may have a large number of collectives, we only used Allreduce: these are common and account for most of the collective execution time. Each instance is evaluated separately. Also, for those collective instances with multiple executions, we take the arithmetic mean of their execution times (maximum time among all processes). The exception is CG which does not use Allreduce: Reduce was used instead.

6.4 | Discussion

We observe that *MPI-FPGA* provides significant performance improvement for some of the benchmarks. The current access to TACC Stampede2 limits the number of nodes in our experiments to 128. We also observe that the performance improvement for proxy applications (a few percent) is

much less than that of the collectives themselves (*factors* of 2.0×–32.6×). As already noted this difference is due to the time spent waiting for laggard processes (known as process skew) that affects any network enhancement.[9] In the applications used here the process skew is generally much larger than the communication latency itself.

While coping with process skew is beyond the scope of the current report, there are obvious paths forward. For example, we plan on applying stepwise refinement to the application benchmarks to transform blocking collective operations (e.g., *MPI_Allreduce*) to non-blocking and more advanced variants offered in newer versions of the MPI standard. This will lead, first, to the non-blocking operations (cf. *MPI_Iallreduce*) available in MPI-3.1, second, to the latest persistent collectives[53] offered in the newly completed MPI-4 standard (cf. *MPI_Alltoall_init*), and, finally, to proposed partitioned collective communication[54] operations planned for MPI-5 (cf. *MPIX_Pallreduce_init*).

In updating the benchmarks with newer collective primitives, we introduce greater scope for MPI to achieve overlap of communication and computation. This occurs by widening the gap between the initiation of non-blocking or persistent collective operations and their subsequent completion. MPI implementations offering asynchronous progress plus hardware offload, notably *MPI-FPGA*, can exploit such overlap efficiently for sufficiently large messages. Secondarily, the planned transfer paradigm supported in the more recent APIs is particularly good at removing setup and tear-down overheads from the critical, per-operation path, and selecting the right mode of collective implementation, such as using the *MPI-FPGA* offload system. Finally, partitioned collective operations are designed to enhance overlap when there are laggard processes by supporting fine-grained production and consumption of sub-buffers (aka, partitions) that can be pipelined into the network, thereby reducing load-imbalance penalties.

7 | RELATED WORK

Previous work has shown that significant performance speedups can be achieved by offloading collectives onto hardware. These generally enhance the NIC,[15-18] tightly connected with the processor via interconnects such as PCI, whereas the work reported here adds hardware support in the switch. For instance, Arap et al.[15] offload collectives onto an FPGA cluster; however, they do not mention any communicator support, nor do they integrate into a switch. Schmidt et al.[16] 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.[17,55] In References 37,56 collectives on FPGA clusters are studied, but the emphasis is on scheduling algorithms. Other work accelerating MPI with FPGAs includes.[57,58] Portals[59] adds programmability and hardware support for triggered operations, message steering, and atomic operations to the NIC. However, this network interface emulates limited processing capabilities.[60]

A general solution was provided by Voltaire[61] 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.

The IBM BlueGene systems[62] offload collectives into the network router and also, to some degree, handle communicators. For instance, BlueGene/Q[63] provides a summing unit for accelerating collective operations which is available for subcommunicators. BlueGene/Q, however, requires class routes for collective operations and there are only 13 class routes available: 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.[62] Overall, the BlueGene solutions show the difficulties in implementing in-switch collective support in fixed logic. While high wire utilization is achieved, there are still many limitations. Collectives are supported in a separate network. The number of communicators is bounded and restricted to either the whole network or a rectangular subset. The collectives and the operations on those collectives are a fixed subset and not extensible.

Recent work by Mellanox[9] offloads MPI collectives to fixed logic switches using reduction trees for short message size. It appears to address many of these problems, but also has similar limitations, in particular, supporting only a small number of simple operations with no extensibility; also there are no published (or generally available) design details. We compare our work with the published results in the evaluation section. A hardware-based streaming aggregation capability was later added to provide high bandwidth for large messages.[48] Finally, the authors in Reference 49 designed, implemented, and evaluated SHArP-based solutions for MPI_Reduce and MPI_Barrier in MVAPICH2-X.

In contrast to previous work, we are the first to offload both CTs and the processing of an entire collective operation in hardware while supporting irregular communicators and providing hardware acceleration of collective packet processing.

8 | CONCLUSION

We present a comprehensive solution to processing collectives in network switches with reconfigurable logic. This has the advantage over fixed logic alternatives of being able to support capabilities as needed. This includes varying the supported operation types and numbers of simultaneous

operations, but also enabling more general user-defined processing in the network. We have demonstrated the last of these advantages in other work.[4,28]

As part of this solution 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, in particular, for collectives occurring over irregular communicators. We find that storing entire process groups in the network is not a scalable solution. We then introduce the CT, 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.

A novel 2-level switch design is introduced to efficiently process in-network collectives; yet it remains flexible enough to embrace user-defined collectives. We provide a streaming interface to improve the performance for long messages. By supporting a full offload of seven popular collectives, we remove nearly all of the collective operation software from MPI and implement the functionality in the switch. The hardware support has been integrated into a reconfigurable router which remains portable enough that it is independent of the type of router. We evaluate *MPI-FPGA* with respect to a CPU cluster and find that the in-switch accelerator achieves significant and scalable speedups.

As part of future work, we aim to augment *MPI-FPGA* to support non-blocking MPI collectives to provide communication-computation overlap and address the process skew challenge. Also, we would like to further augment this approach by supporting persistent collectives and partitioned communication.

ACKNOWLEDGMENTS

This work is supported, in part, by the NSF through awards CCF-1919130 and CNS-1925504; by a Grant from Red Hat; and by generous donations from Intel and Xilinx. 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.

DATA AVAILABILITY STATEMENT

Data available on request from the authors.

ORCID

Pouya Haghi https://orcid.org/0000-0003-2893-9194
Qingqing Xiong https://orcid.org/0000-0002-3243-3949
Tong Geng https://orcid.org/0000-0002-3644-2922

REFERENCES

1. Stern J, Xiong Q, Sheng J, Skjellum A, Herbordt M. Accelerating MPI_Reduce with FPGAs in the network. Proceedings of the Workshop on Exascale MPI; 2017:1-4.
2. Stern J, Xiong Q, Skjellum A, Herbordt M. A novel approach to supporting communicators for in-switch processing of MPI collectives. Proceedings of the Workshop on Exascale MPI; 2018:1-10.
3. Xiong Q, Yang C, Haghi P, Skjellum A, Herbordt M. Accelerating MPI collectives with FPGAs in the network and novel communicator support. Proceedings of the 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines; 2020.
4. Haghi P, Guo A & Xiong Q et al. FPGAs in the network and novel communicator support accelerate MPI collectives. Proceedings of the 2020 IEEE High Performance Extreme Computing Conference; 2020:1-10.
5. Klenk B, Froening H. An overview of MPI characteristics of exascale proxy applications. Proceedings of the International Supercomputing Conference; Vol. 10266, 2017:217-236.
6. Bernholdt DE, Boehm S, Bosilca G, et al. A survey of MPI usage in the US exascale computing project. *Concurr Comput Pract Exper*. 2018;32:1-16.
7. Pjesivac-Grbovic J, Angskun T, Bosilca G, Fagg GE, Gabriel E & Dongarra JJ Performance analysis of MPI collective operations. Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium; 2005:1-8.
8. Faraj A, Kumar S, Smith B, Mamidala A & Gunnels J MPI collective communications on the blue gene/P supercomputer: algorithms and optimizations. Proceedings of the Symposium on the High Performance Interconnects, Hot Interconnect; 2009:63-72.
9. Graham RL, Bureddy D, Lui P, et al. Scalable hierarchical aggregation protocol (SHArP): a hardware architecture for efficient data reduction. Proceedings of the 1st International Workshop on Communication Optimizations in HPC (COMHPC); 2016:1-10.
10. Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput*. 1996;22:789-828.
11. Thakur R, Rabenseifner R, Gropp W. Optimization of collective communication operations in MPICH. *Int J High Perform Comput Appl*. 2005;19(1):49-66. doi:10.1177/1094342005051521
12. Chan EW, Heimlich MF, Purkayastha A, van de Geijn RA. On optimizing collective communication. Proceedings of the 2004 IEEE International Conference on Cluster Computing; September 2004:145-155.
13. Raffenetti K, Amer A, Oden L, et al. Why is MPI so slow? analyzing the fundamental limits in implementing MPI-3.1. Sc; 2017:1-12.
14. Schonbein W, Grant R, Dosanjh M, Arnold D. Inca: in-network compute assistance. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2019.
15. Arap O, Swany M. Offloading collective operations to programmable logic on a Zynq cluster. Proceedings of the IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI); 2016:76-83.

16. Schmidt AG, Kritikos WV, Gao S, Sass R. An evaluation of an integrated on-chip/off-chip network for high-performance reconfigurable computing. *Int J Reconfigurable Comput*. 2012;2012:5.
17. Peng Y, Saldana M, Chow P. Hardware support for broadcast and reduce in MPSOC. Proceedings of the International Conference on Field Programmable Logic and Applications; 2011:144-150.
18. Mellanox. Fabric collective accelerator (FCA); 2019. https://www.mellanox.com/
19. Xiong Q, Yang C, Patel R, Geng T, Skjellum A, Herbordt M. GhostSZ: a transparent SZ lossy compression framework with FPGAs. Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM); 2019:258-266.
20. Knuth D. *The Art of Computer Programming: Volume III Sorting and Searching*. Addison-Wesley; 1973.
21. Eppstein D, Galil Z. Parallel algorithmic techniques for combinatorial computing. *Annu Rev Comput Sci*. 1988;3:233-283.
22. Almàsi G, Heidelberger P, Archer CJ, et al. Optimization of MPI collective communication on BlueGene/L systems. Proceedings of the 19th International Conference on Supercomputing; 2005:253-262.
23. Arista networks; 2013. Accessed October 2013. http://www.aristanetworks.com/en/products/7100series/7124fx/
24. Putnam A, Caulfield AM, Chung ES. A reconfigurable fabric for accelerating large-scale datacenter services. Proceedings of the International Symposium on Computer Architecture; 2014:13-24.
25. George A, Herbordt M, Lam H, Lawande A, Sheng J, Yang C. Novo-G#: a community resource for exploring large-scale reconfigurable computing through direct and programmable interconnects. Proceedings of the 2016 IEEE High Performance Extreme Computing Conference (HPEC); 2016:1-7; Waltham, MA.
26. Miyajima T, Ueno T, Koshiba A, Huthmann J, Sano K, Sato M. High-performance custom computing with FPGA cluster as an off-loading engine. Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis; 2018:1-2.
27. Stewart L, Pascoe C, Davis E, Sherman B, Herbordt M, Sachdeva V. Particle mesh Ewald for molecular dynamics in OpenCL on an FPGA cluster. Proceedings of the 29th IEEE International Symposium on Field-Programmable Custom Computing Machines; 2021.
28. Haghi P, Geng T, Guo A, Wang T, Herbordt M. FP-AMG: FPGA-based acceleration framework for algebraic multi-grid solvers. Proceedings of the 28th IEEE International Symposium on Field-Programmable Custom Computing Machines; 2020:148-156.
29. Munafo R. *Cooperative High-Performance Computing with FPGAs: Matrix Multiply Case Study*. Master's thesis. Department of Electrical and Computer Engineering, Boston University; 2018. https://open.bu.edu/handle/2144/30740.
30. Kamal H, Mirtaheri SM, Wagner A. Scalability of communicators and groups in MPI. Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing; 2010:264-275.
31. Mantevo project. https://mantevo.github.io/
32. Gropp W, Lusk E, Ashton D, Ross R, Thakur R, Toonen B. 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
33. Sheng J, Yang C, Herbordt M. High performance dynamic communication on reconfigurable clusters. Proceedings of the 28th International Conference on Field Programmable Logic and Applications; 2018. doi: 10.1109/FPL.2018.00044
34. Alizadeh M, Edsall T. On the data path performance of leaf-spine datacenter fabrics. Proceedings of the 2013 IEEE 21st Annual Symposium on High-Performance Interconnects; 2013:71-74.
35. Dongarra J, Otto S, Snir M, Walker D. An introduction to the MPI standard; 1995.
36. De Matteis T, de Fine Licht J, Beránek J, Hoefler T. Streaming message interface: high-performance distributed memory programming on reconfigurable hardware. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2019:1-33.
37. Sheng J, Xiong Q, Yang C, Herbordt M. Collective communication on FPGA clusters with static scheduling. *ACM SIGARCH Comput Arch News*. 2016;44(4):2–7.
38. Dally W, Towles B. *Principles and Practices of Interconnection Networks*. Elsevier; 2004.
39. Byma S, Steffan JG, Bannazadeh H, Leon-Garcia A, Chow P. FPGAs in the cloud: booting virtualized hardware accelerators with OpenStack. Proceedings of the 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines; 2014:109-116.
40. Khawaja A, Landgraf J, Prakash R, Wei M, Schkufza E, Rossbach CJ. Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS. Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18); October 2018:107-127; USENIX Association, Carlsbad, CA.
41. Vivado design suite user guide: partial reconfiguration. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug909-vivado-partial-reconfiguration.pdf
42. Alexandrov A, Ionescu MF, Schauser KE, Scheiman C. LogGP: incorporating long messages into the LogP model-one step closer towards a realistic model for parallel computation. Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures, ser. SPAA '95; 1995:95-105; Association for Computing Machinery, New York, NY.
43. Haghi P, Guo A, Geng T, Skjellum A, Herbordt M. Workload imbalance in HPC applications: effect on performance of in-network processing. Proceedings of the 2021 IEEE High Performance Extreme Computing Conference; 2021.
44. Yang C. *High-Performance Communication Infrastructure Design on FPGA-Centric Clusters*. Ph.D dissertation. Department of Electrical and Computer Engineering, Boston University; 2019. https://open.bu.edu/handle/2144/38207
45. Stanzione D, Barth B, Gaffney N, et al. Stampede 2: the evolution of an XSEDE supercomputer. Practice and Experience in Advanced Research Computing on Sustainability, Success and Impact; 2017.
46. OSU Micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/
47. Hoefler T, Belli R. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC '15; 2015; Association for Computing Machinery, New York, NY. 10.1145/2807591.2807644
48. Graham RL, Levi L, Burredy D, et al. Scalable hierarchical aggregation and reduction protocol (SHARP)TM streaming-aggregation hardware design and evaluation. In: Sadayappan P, Chamberlain BL, Juckeland G, Ltaief H, eds. *High Performance Computing*. Springer International Publishing; 2020:41-59.
49. Ramesh B, Suresh K, Sarkauskas N, et al. Scalable MPI collectives using SHARP: large scale performance evaluation on the TACC frontera system; November 2020:11-20.
50. NAS parallel benchmarks. https://www.nas.nasa.gov/publications/npb.html#url

51. ECP proxy applications, miniFE Catalog. https://github.com/Mantevo/miniFE
52. HPCCG benchmark. https://github.com/Mantevo/HPCCG
53. Holmes DJ, Morgan B, Skjellum A, Bangalore PV, Sridharan S. Planning for performance: enhancing achievable performance for MPI through persistent collective operations. *Parallel Comput*. 2019;81:32-57.
54. Skjellum A, Bangalore PV, Holmes DJ, et al. Partitioned collective communication: a position paper; May 2021; Submitted to EuroMPI 2021, Under review.
55. Saldaña M, Patel A, Madill C, et al. MPI as a programming model for high-performance reconfigurable computers. *ACM Trans Reconfig Technol Syst (TRETS)*. 2010;3(4):22.
56. Sheng J, Yang C, Herbordt M. Application-aware collective communication on FPGA clusters. Proceedings of the IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM); 2016:197-197.
57. Xiong Q, Skjellum A, M. Herbordt. Accelerating MPI message matching through FPGA offload. Proceedings of the 2018 28th International Conference on Field Programmable Logic and Applications (FPL); 2018:191-194.
58. Xiong Q, Bangalore P, Skjellum A, Herbordt M. MPI derived datatypes: performance and portability issues. Proceedings of the 25th European MPI Users' Group Meeting; 2018.
59. Barrett BW, Brightwell RB, Grant RE, et al. The portals 4.0.2 networking programming interface; 2014:11.
60. Di Girolamo S, Jolivet P, Underwood KD, Hoefler T. Exploiting offload-enabled network interfaces. *IEEE Micro*. 2016;36(4):6-17.
61. Wachtel A. Boosting Scalability of InfiniBand-based HPC. *Clusters*. 2010. https://cw.infinibandta.org/document/dl/7259
62. Kumar S, Mamidala A, Heidelberger P, Chen D, Faraj D. Optimization of MPI collective operations on the IBM Blue Gene/Q supercomputer. *Int J High Perform Comput Appl*. 2014;28(4):450-464.
63. Gilge M. *IBM System Blue Gene Solution Blue Gene/Q Application Development*. An IBM Redbooks Publication; 2013.

How to cite this article: Haghi P, Guo A, Xiong Q, et al. Reconfigurable switches for high performance and flexible MPI collectives. *Concurrency Computat Pract Exper*. 2021;e6769. doi: 10.1002/cpe.6769