FPGAs in the Network and Novel Communicator Support Accelerate MPI Collectives*

Pouya Haghi* Anqi Guo* Qingqing Xiong* Rushi Patel* Chen Yang* Tong Geng*
Justin T. Broaddus† Ryan Marshall† Anthony Skjellum† Martin C. Herbordt*
*Dept. of Electrical and Computer Engineering, Boston University
†Simcenter & Dept. of Computer Science and Engineering, University of Tennessee at Chattanooga

***Abstract*—MPI collective operations can often be performance killers in HPC applications; we seek to solve this bottleneck by offloading them to reconfigurable hardware within the switch itself, rather than, e.g., the NIC. We have designed a hardware accelerator MPI-FPGA to implement six MPI collectives in the network. Preliminary results show that MPI-FPGA achieves on average 3.9× speedup over conventional clusters in the most likely scenarios. Essential to this work is 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, and that is scalable to very large systems. We show how communicator support can be integrated easily into an in-switch hardware accelerator to implement MPI communicators and so enable 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 are an ideal testbed and have a number of other architectural benefits. MPI-FPGA is fully integrated into MPICH and so transparently usable by MPI applications.**

***Index Terms*—MPI, Collectives, FPGA-Centric Clusters, High Performance Computing, In-Network Computing**

I. Introduction

High performance computing (HPC) applications often rely on collective communication for performing operations that require interaction among multiple processes; collectives comprise a large fraction of total HPC communication [4], [5], and in many applications they bottleneck performance [6]–[8]. 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) [9]. And since virtually all communication in production HPC is based on MPI [5], addressing the acceleration of collectives necessarily means dealing with them within that framework.

Collectives in MPI implementations (such as MPICH [9]) generally consist of point-to-point messages with computations in between. Thus much support has been added at the software level [6], [10], [11]; this includes new algorithms that can improve performance by optimizing, e.g., either for low latency with small data sets or for high throughput when dealing with large arrays [10]. However, the addition of these algorithms has greatly complicated the software stack [12].

In this work we offload MPI collectives into FPGA hardware (MPI-FPGA); in particular, into logic appended to 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 [13]–[17]. While valuable, the NIC-only approach still leaves much performance on the table, in particular, the fourth and fifth benefits just described. Most obviously, the NIC is an endpoint 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.

General compute-in-the-network has been studies since the early days of computing through structures such as adder trees and sorting networks; it is also fundamental to the more powerful PRAM models explored in the 1980s [18]. However, there appear to be just two recent commercial versions of in-switch computing: the IBM BlueGene family [7], [19] and certain switches from Mellanox [8]. Both of these have limitations (described below) that are, in part, the result of being ASIC-based and so having strictly bounded and inflexible capabilities. Moreover, being commercial products, the details available about their actual implementation are very limited.

There are at least two plausible models for using FPGAs for in-switch compute-in-the-network. One is to use reconfigurable logic in the router (already common for other purposes [20]) in an indirect network. A second is in FPGA-centric clusters [21]–[30] with direct FPGA-FPGA interconnects. In this work we assume the latter model as it allows us to evaluate working prototypes; the major results, however, are applicable to the first model as well. FPGA implementations have several inherent advantages: first, they are not limited to a small, fixed set of operations; second, for any application, they only need to implement the operations that are substantially used;

∗ Parts of this work have been presented at workshops with no proceedings [1], [2] and as a published extended abstract [3].

third, support can be extended beyond simple datatypes to higher order structures such as matrices, tensors, etc.; and fourth, compute-in-the-network can be generalized still further to support *altruistic* or *opportunistic* computing.

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. They are primarily used for performing collective operations over a subset of processes in the application. Handling sub-communicator collectives in hardware does not come without its share of complications. Communicators have significant scalability issues [31], meaning we cannot implement them in hardware with the same methods used for managing communicators in software. As we approach exascale, the added latency and memory costs of managing communicators would soon exceed any realistic hardware constraints. In this work we introduce an in-switch design capable of efficiently supporting communicators and the collectives that run on them. We are able to achieve this with a new Communicator Table (CT) design, which provides general communicator support while consuming 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 FPGA in-switch MPI collectives. We believe this to be the first FPGA version to be fully integrated into a general router. Also, MPI-FPGA is fully integrated into MPICH with publicly available code and API; MPI-FPGA is therefore currently transparently usable by any MPI application. It is also easily extended to support additional collectives or integrated into other MPI implementations. The 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.

Our experiments show that MPI-FPGA can achieve an improvement of about 10x for MPI collectives over a CPU cluster for medium sized messages, with greater speedup for smaller messages and less for larger messages. These collectives can be run over sub-communicators without sacrificing performance. In addition, there is little added cost over the general router itself and the enhanced router only takes a small fraction of the total device resources on current FPGAs.

II. Concepts

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

A. MPI Software Stack

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 1(a), 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.

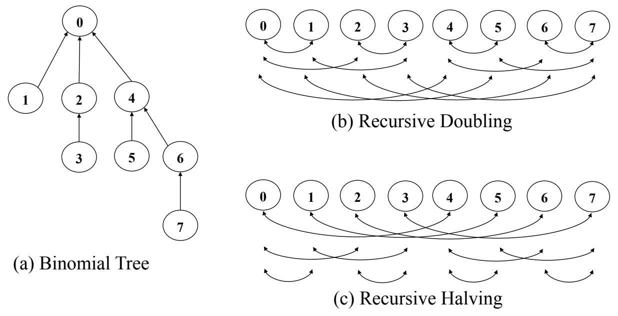

Fig. 1: Algorithms used by MPICH Collectives

*a) **MPI-FPGA Software Support:*** **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 is transparent making it completely portable: it can be integrated into HPC applications without requiring the programmer to have any knowledge of the underlying hardware or making any changes to existing programs. 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*), 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. If a collective does not have offload support, then it is performed as usual by the software.

Upon receiving the function 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

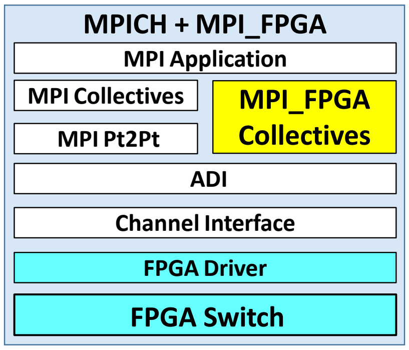

Fig. 2: MPI-FPGA Software Stack

been received and passed up to the CPU from the network. In the MPICH implementation of MPI middleware [9], all of the functionality of the ADI is maintained. We are currently using MPICH-3.2 [32]; 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).

*b) **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 (**CCM**) and the Reduction Unit (**RU**) (details in Section IV). 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 CCM receives packets, it operates on the packets if necessary. Otherwise, it simply passes the packet into the switch without modification. The RU on the other side of the switch, again, only operates on packets if they are a part of a reduction operation. Although we have implemented the design on an FPGA, its portability ensures that it is also independent of the type of hardware used.

B. 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. However many MPI programs use multiple communicators: they are a central MPI capability and must be supported in creating a useful system. A common example of using sub-communicators is when partitioning workload among an array of MPI processes and performing collectives on an entire row or column of processes. The most common way to do so is to call a function like *MPI_Comm_Split*, one of the many for creating communicators, to divide the global communicator into sub-communicators. Another common reason for having multiple

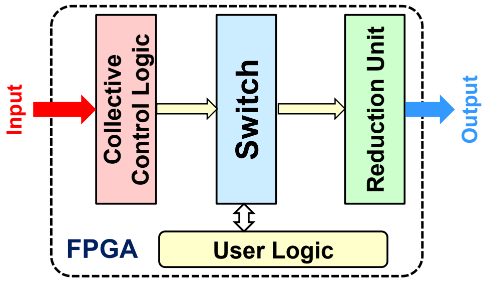

Fig. 3: MPI-FPGA Hardware Model

communicators is to divide MPI processes into masters and slaves and then performing collectives on these separately.

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 large communicators, the memory consumption of these process groups leads to scaling issues [31]. To have an entire process group in FPGA memory would require storing the list of all ranks included in the communicator. The number of bits required would be the product of *COMM_SIZE* and *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* of many communicators in a single application. Since high performance depends on having routing information on the device, replicating information about these entire process groups would quickly use up memory resources, even for mid-sized clusters.

C. 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 [13]–[16], tightly connected with the host CPU via interconnects such as PCI, whereas we add hardware support in the switch. For instance, Arap, et al. [13] 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 can begin reductions as soon as data is received. Schmidt, et al. [14] 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 [15], [33]. In [34], [35] collectives on FPGA clusters is studied, but the emphasis is on scheduling algorithms. Other work accelerating MPI with FPGAs includes [36], [37].

A general solution was provided by Voltaire [38] 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 [39] offload collectives into the network router and also, to some degree, handle communicators. For instance, BlueGene/Q [40] 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 [39]. 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 [8] 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 with published results in the evaluation section.

In contrast to previous work, we are the first to offload both communicator tables and the processing of an entire collective operation in hardware while supporting irregular communicators and providing hardware acceleration of collective packet processing. Compared with the NIC offload solutions, the in-switch solution is able to make the shortest collective routes with the ability to process and distribute packets across the network. Compared with ASIC solutions, the reconfigurable logic allows arbitrary functions and datatypes to be supported. Also, at any time, only those operators/functions that are needed for a particular application need to be instantiated.

III. Communicator Processing

A. Communicator Table (CT)

The purpose of the CT is to manage communicator information that is needed for 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 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 initialization.

Table I shows the different algorithms that are used in the MPICH-3.2 implementations of six popular collectives. The three 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 three algorithms, we can perform all of the collectives that use them. Looking back at Figure 1, for each rank we can identify the subset with which a given rank must communicate. For example, rank 0 must communicate with the following set in all three 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.

TABLE I: MPICH-3.2 Collective Algorithms

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

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 (which can be proved directly from the properties of binomial trees).

Once the FPGA obtains this subset of ranks for a communicator, it stores the addresses in a table along with the subsets from other communicators. As shown in Table II, the CT 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 processes that the FPGA will be communicating with. Each communicator entry is indexed into the table using its context id. For an incoming packet it is thus easy to look up the communicator it is from: the context id is a field in 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. 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. Once a valid entry is loaded into the table, no updates on that entry are ever required until the communicator is freed.

TABLE II: Communicator Table Structure

Valid	Context ID	Comm Size	Local Bank	1st Addr	2nd Addr	3rd Addr
1	0	8	0	1	2	4
1	1	4	0	1	2	N/A

B. Communicator Table (CT) Entry Creation

When a new communicator is created in software, the FPGA needs a way of obtaining the CT 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 CT 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

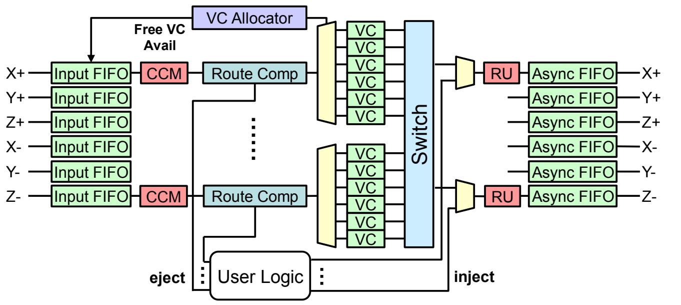

Fig. 4: Router with MPI collective support: Collective Control Module (CCM) and Reduction Unit (RU)

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.

IV. Implementation

A. Overview

The base of the design is a virtual-channel dynamic router (see Figure 4) designed to be used in an FPGA cluster interconnected in a 3D torus. It has 6 input and 6 output ports each connected to Multi-Gigabit Transceivers (MGTs); these allow FPGAs to be directly connected to each other. The base router has a classic 4 stage pipeline [41]: route computation, virtual channel allocation, switch allocation, and switch traversal. With the added MPI support the pipeline is extended to six stages.

The MPI offload support was designed to keep the overall design modular: the accelerator architecture is portable to any other standard router. It 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 packets' output ports can be calculated in the route computation stage after it is assigned a new destination. The RU 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

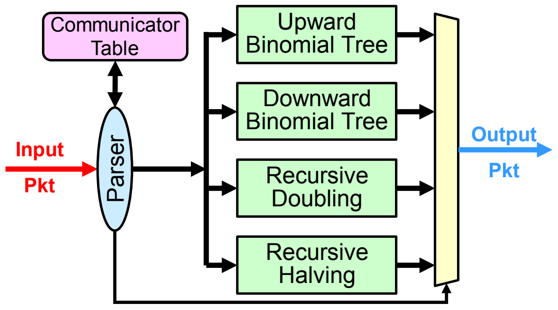

Fig. 5: Collective Control Module

that all packets going into any particular reduction unit will exit using the same output port. This frees the reduction unit from having to manage the output ports of each packet.

B. Collective Control Module (CCM)

The CCM (Figure 5) 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*, 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 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 CCM uses the CT to determine new destinations for the packet.

In order for the CCM to determine which collective a packet is a part of, a collective 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 5. Within each of these algorithm blocks, MPI-FPGA performs computations using input from the packet header and 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 communicator table also eases the computation required to calculate these destinations. When 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.

As in MPICH-3.2, the implementation also supports multiple algorithms for the same collective operations. The algorithm used is often determined by packet size, e.g., short versus long. In MPI-FPGA we support algorithm selection by adding a bit to the packet header opcode field.

C. Reduction Unit (RU)

The RU performs the reduction computations 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, then they are transferred to the RU. RU has a reduction

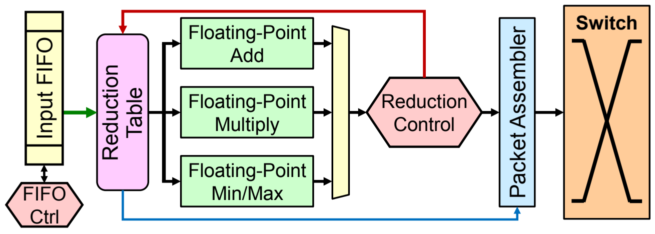

Fig. 6: Reduction Computation Unit Data-Flow

table which is indexed and capable of supporting multiple reductions simultaneously.

When a reduction packet enters the RU, 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 6).

The arithmetic unit is constructed from using standard methods including use of vendor IP. The default design supports addition, multiplication, max, and min, but is trivially extendable for other operations including user specified functions. All functions are run in parallel.

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 remaining as the reduction continues. When a reduction table entry has received packets from all child nodes, 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, MPI-FPGA can support multiple reductions executing simultaneously. If the reduction is large and needs to be subdivided, each occupies a different slot of the reduction table. This unit can also support separate unrelated reductions and is flexible enough to allow any order of reductions occurring throughout the reduction table. 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.

V. Evaluation

We have implemented, tested, and verified MPI-FPGA on a four FPGA system. Each Gidel ProcV FPGA board hosts an Intel/Altera Stratix V 5GSMD8 chip with 12 MGT ports exposed to users. The FPGAs are configured to run at 150MHz. The inter-FPGAs link is realized with Multi-Gigabit Transceivers (MGTs) with a bandwidth of 40Gbps and latency of 200ns. To give an idea of how MPI-FPGA scales with current technology we have also implemented it for an Intel Stratix 10 1SG280LU2F50E2VG FPGA; results there are post place-and-route.

A. Resource Utilization

TABLE III: Resource usage on Stratix V & Stratix 10 FPGAs

Design	Device	ALM	BRAM	DSP
Baseline	Stratix V	38K(15%)	234(9%)	0(0%)
Baseline	Stratix 10	18K(2%)	234(2%)	0(0%)
Collective	Stratix V	42K(16%)	267(11%)	12(1%)
Collective	Stratix 10	20K(2%)	267(2%)	10(1%)

FPGA resource consumption is shown in Table III. The resource consumption of the CT can vary based on a user defined parameter, *COMMUNICATOR_COUNT*, that specifies how many communicators a single MPI process can be a part of. Here it is set to 30. MPICH-3.2 has 12 predefined reduction operations; here we implement all of them. Depending on application requirements, the user can remove the support for unused operations or add user-defined operations; these actions of course decrease or increase resource utilization, respectively.

The first two rows give the resource usage for the baseline router design for Stratix V and Stratix 10 implementations. The virtual channel number per link is set to 6 (bidirectional). The next two rows give the resource usage for the MPI-FPGA router. Compared with the baseline design, the added MPI collective support increases the overall resource usage by 10% for ALMs and 14% for BRAMs. Overall resource usage for the Stratix V is less than 20% and for the Stratix 10 is around 2%. Especially in the latter case, this leaves abundant on-chip resources for applications. For implementation of all predefined reduction operations, the design consumes 12 DSP blocks on the Stratix V and 10 on the Stratix 10.

Since the design is parameterized, the user can change the communicator count based on resource budget and communication requirements. Let *NUM_COMM* be the maximum amount of communicators a process can be in, *WORLD_SIZE* be the number of ranks in *MPI_COMM_WORLD*, and *ADDR_LEN* be the number of bits is a process's physical address. As shown above, a CT entry has at most $lg(WORLD\_SIZE)$ other ranks that it might need to communicate with, so the maximum number of bits used by the CT, not included small amount of meta-data, is

$$\begin{aligned} bits =& NUM\_COMM * lg(WORLD\_SIZE) \\ & * ADDR\_LEN \end{aligned} \tag{1}$$

B. MPI Collectives Performance

We evaluated MPI-FPGA for *Reduce*, *Allreduce*, *Broadcast*, *Scatter*, *Gather*, and *Allgather* using the OSU Microbenchmarks [42], a well-known set of MPI benchmarks. For all collectives, double precision floating point was used. To generate the FPGA cluster results, we first ran MPI-FPGA on a testbed equipped with 4 FPGAs. Larger systems were simulated using ModelSim and calibrated with measured parameters. For the CPU reference, benchmarks were run on the Stampede2 [43] 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/sec Intel Omni-Path (OPA) network (fat tree topology).

For the CPU cluster, we restricted the mapping to one process per node. To observe the behavioral differences among a variety of process-node mappings, we ran a large number of experiments on different numbers of nodes (< 128) and different numbers of processes per node (< 48). Based on our observations, we found that the number of processes per node made little difference in scaling. The maximum of 128 processes in the results is due to limitations imposed by the execution environment with the FPGA-related experiments.

*a) **Overall Collective Latency**:* fig. 7 shows the simulation results of MPI and MPI-FPGA collectives for small to medium array sizes on the 32-, 64-, and 128-rank systems using the OSU benchmarks. Execution time is the time that it took for the last process to complete the operation. It should be noted that host-FPGA communication latency is included in execution time. The message sizes (number of elements being reduced) were varied from 1 - 128 elements (8 - 1024 bytes). With counts higher than 128, reductions are usually split in software [19].

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 root node and intermediate nodes 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 our experimental CPU cluster (as opposed to 3D torus for the FPGA cluster) can offset the aforementioned benefits, we can see that MPI-FPGA has a higher overall performance, especially for small messages. Moreover, as it is evident from fig. 7, MPI-FPGA speedup relative to CPU cluster is higher for *MPI_Allgather* and *MPI_Allreduce*, since a greater number of intermediate nodes are involved in these MPI collectives.

MPI-FPGA achieves a higher performance for small message sizes than the CPU benchmarks. This advantage diminishes somewhat for reductions as the number of combined elements increases because for large messages, reductions turn from communication into computation problems. The computation unit in MPI-FPGA is much smaller than that in a CPU core, and the clock rate is generally much lower. The obvious and simple solution is to add more parallelism to MPI-FPGA, which translates to more resource usage; this is likely to be worthwhile for many applications and we are currently exploring this option. When looking at problem size, of particular interest is that the MPI-FPGA speedup is maintained as the number of processes grows, thus indicating the expected benefit for larger systems of MPI-FPGA through reducing network traffic.

To view the results in a different perspective, Table IV shows the speedups for each of these collectives for a message size of 128 bytes. Overall, MPI-FPGA achieves on average 3.9× speedups for different collectives.

TABLE IV: MPI-FPGA speedups over OSU Benchmarks running on 32, 64, and 128 nodes of Stampede2

	32 ranks	64 ranks	128 ranks
Reduce	1.51	1.28	1.53
Allreduce	4.26	5.85	10.1
Bcast	4.62	6.69	5.44
Scatter	0.76	5.58	5.31
Gather	1.66	1.43	1.43
Allgather	3.83	4.14	4.34

*b) **Average Case Results**:* We also collected the average case results, or the average amount of time that it took for any process in a collective to finish. We found that for *Allreduce* and *Allgather*, the worst-case results and average-case results were nearly identical. 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, however, 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 simply sends a special message to the FPGA and returns to the application. For example, in a *Gather* where only the root process deals with 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.

The average case results should not be represented as the actual speedup of a collective, but are nevertheless significant. It is common in MPI applications for one process be a master that sends work to the other slave processes. When a slave process completes its work, it notifies the master, which will then assign more work items to the slave.

*c) **Discussion**:* We have presented initial results that indicate performance benefit of compute-in-the-network using FPGAs at modest resource cost. Especially promising is that the *harder* the problem (larger, more messages), the greater the benefit. Note performance is only one potential benefit. We have just hinted at another, that nodes that finish early can begin other computations. Other benefits remain for further study, including the secondary effects of the reduced communication load, e.g., on other applications using the cluster.

Of great interest is the comparison of MPI-FPGA with the commercial ASIC-based version of compute-in-the-network from Mellanox. Unfortunately gaining access to SHARP-based systems is extremely challenging. As a proxy we have superimposed FPGA results onto SHARP results from [8]. SHARP performance improvement ranges from 17% for 32 ranks to 32% for 128 ranks. Since SHARP has a fanout of 36 versus 6 for the current MPI-FPGA fixture, and bandwidth of 100Gps versus 40Gps, we consider these results highly promising. This is even before considering FPGA advantages of cost and flexibility, and planned enhancements to MPI-FPGA described above.

VI. Conclusion

We present a new method for supporting MPI communicators and accelerating collectives in the network switch. We

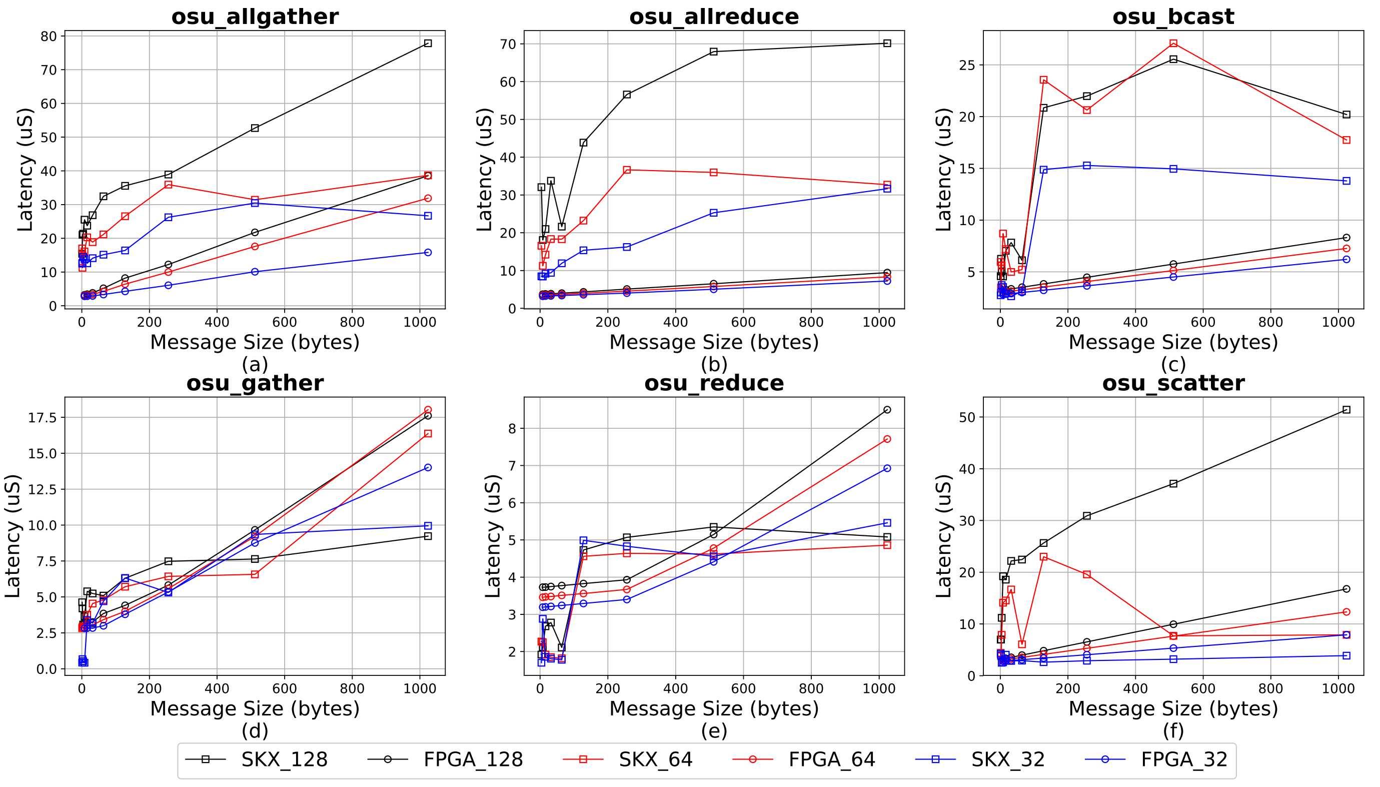

Fig. 7: MPI CPU cluster (SKX) vs MPI-FPGA execution time for 32, 64, and 128 nodes: (a) osu_allgather, (b) osu_allreduce, (c) osu_bcast, (d) osu_gather, (e) osu_reduce, and (f) osu_scatter.

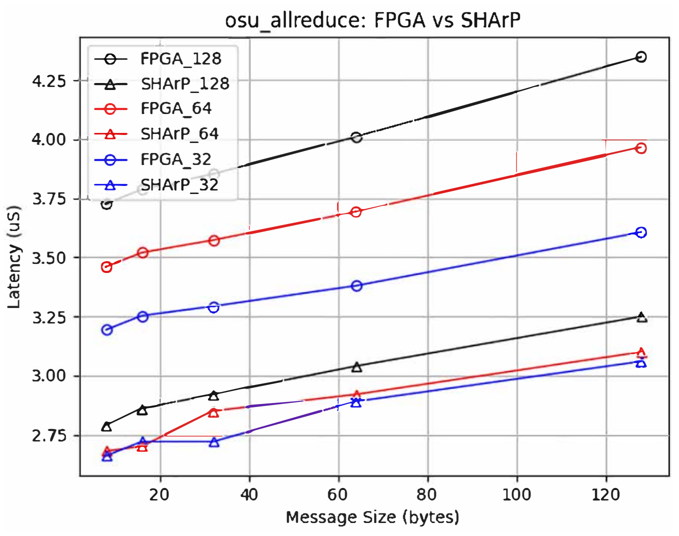

Fig. 8: Allreduce performance comparison of MPI-FPGA versus SHARP (from [8]).

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 a storing entire process groups in the network is not a scalable solution. We then introduce the Communicator Table (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. By supporting a full offload of six 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 wormhole router, but remains portable enough that it is independent of the type of router and and system infrastructure. We evaluate MPI-FPGA with respect to a CPU cluster and find that the in-switch accelerator achieves significant and scalable speedups. Much higher performance is easily attainable through the addition of ALUs, increased fanout, and higher bandwidth connections.

Acknowledgements

Support is acknowledged from the National Science Foundation under Grants Nos. CCF-1562659, CCF-1562306, CCF-1618303, CCF-1617690, CCF-1822191, CCF-1821431, CCF-1919130, and CNS-1925504; from the NIH through Award R44GM128533; from grants from Microsoft and Red Hat; and from Xilinx and by Intel through donated FPGAs, tools, and IP. 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] J. Stern, Q. Xiong, J. Sheng, A. Skjellum, and M. Herbordt, “Accelerating MPI_Reduce with FPGAs in the Network,” in *Workshop on Exascale MPI*, 2017.

[2] J. Stern, Q. Xiong, A. Skjellum, and M. Herbordt, "A Novel Approach to Supporting Communicators for In-Switch Processing of MPI Collectives," in *Workshop on Exascale MPI*, 2018.

[3] Q. Xiong, C. Yang, P. Haghi, A. Skjellum, and M. Herbordt, "Accelerating MPI Collectives with FPGAs in the Network and Novel Communicator Support," in *IEEE Symposium on Field Programmable Custom Computing Machines*, 2020.

[4] B. Klenk and H. Froening, "An Overview of MPI Characteristics of Exascale Proxy Applications," *International Supercomputing Conference*, vol. 10266, pp. 217–236, 2017.

[5] D. Bernholdt, S. Boehm, G. Bosilca, M. Venkata, R. Grant, T. Naughton, H. Pritchard, M. Schulz, and G. Vallee, "A Survey of MPI Usage in the US Exascale Computing Project," *Concurrency and Computation: Practice and Experience*, vol. Special Issue, pp. 1 – 16, 2018.

[6] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," in *19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[7] A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels, "MPI collective communications on the blue gene/P supercomputer: Algorithms and optimizations," *Proceedings - Symposium on the High Performance Interconnects, Hot Interconnects*, pp. 63–72, 2009.

[8] R.L. Graham, et al., "Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction," in *First International Workshop on Communication Optimizations in HPC (COMHPC)*, 2016.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, pp. 789 – 828, 1996, doi: 10.1016/0167-8191(96)00024-5.

[10] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[11] E. W. Chan, M. F. Heimlich, A. Purkayastha, and R. A. van de Geijn, "On optimizing collective communication," in *2004 IEEE International Conference on Cluster Computing*, Sep. 2004, pp. 145–155.

[12] K. Rafenetti, L. Oden, C. Archer, W. Bland, M. Blocksome, M. Si, P. Cofman, J. Jose, A. Sannikov, M. Chuvelev, P. Fischer, M. Otten, and M. Min, "Why Is MPI So Slow ? Analyzing the Fundamental Limits in Implementing MPI-3 . 1," *Sc*, 2017.

[13] O. Arap and M. Swany, "Offloading Collective Operations to Programmable Logic on a Zynq Cluster," in *IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI)*, 2016, pp. 76–83.

[14] A. G. Schmidt, W. V. Kritikos, S. Gao, and R. Sass, "An evaluation of an integrated on-chip/off-chip network for high-performance reconfigurable computing," *International Journal of Reconfigurable Computing*, vol. 2012, p. 5, 2012.

[15] Y. Peng, M. Saldana, and P. Chow, "Hardware support for broadcast and reduce in MPSOC," in *International Conference on Field Programmable Logic and Applications*, 2011, pp. 144–150.

[16] Mellanox, "Fabric Collective Accelerator (FCA)," https://www.mellanox.com/, 2019.

[17] Q. Xiong, C. Yang, R. Patel, T. Geng, A. Skjellum, and M. Herbordt, "GhostSZ: A Transparent SZ Lossy Compression Framework with FPGAs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 258–266, doi: 10.1109/ FCCM.2019.00042.

[18] D. Eppstein and Z. Galil, "Parallel algorithmic techniques for combinatorial computing," *Annual Review of Computer Science*, vol. 3, pp. 233–283, 1988.

[19] G. Almàsi, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng, "Optimization of MPI collective communication on BlueGene/L systems," in *19th International Conference on Supercomputing*, 2005, pp. 253–262.

[20] Arista Networks, Inc., 2013, http://www.aristanetworks.com/en/products/7100series/7124fx/, accessed 10/2013.

[21] R. Sass, et al., "Reconfigurable computing cluster (RCC) project: Investigating the feasibility of FPGA-based petascale computing," in *IEEE Symposium on Field Programmable Custom Computing Machines*, 2007, pp. 127–138.

[22] S. Moore, P. Fox, A. Markettos, and A. Majumdar, "Bluehive–A Field Programmable Custom Computing Machine for Extreme-Scale Real-Time Neural Network Simulation," in *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, 2012.

[23] A. Putnam, et al., "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *International Symposium on Computer Architecture*, 2014, pp. 13–24, doi: 10.1109/ISCA.2014.6853195.

[24] J. Sheng, C. Yang, and M. Herbordt, "Towards Low-Latency Communication on FPGA Clusters with 3D FFT Case Study," in *International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, 2015.

[25] A. George, M. Herbordt, H. Lam, A. Lawande, J. Sheng, and C. Yang, "Novo-G#: A Community Resource for Exploring Large-Scale Reconfigurable Computing Through Direct and Programmable Interconnects," in *2016 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA*, 2016, pp. 1–7, doi: 10.1109/ HPEC.2016.7761639.

[26] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *49th IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–13.

[27] J. Sheng, C. Yang, A. Caulfield, M. Papamichael, and M. Herbordt, "HPC on FPGA Clouds: 3D FFTs and Implications for Molecular Dynamics," in *27th International Conference on Field Programmable Logic and Applications*, 2017, doi: 10.23919/ FPL.2017.8056853.

[28] J. Sheng, C. Yang, and M. Herbordt, "High Performance Dynamic Communication on Reconfigurable Clusters," in *28th International Conference on Field Programmable Logic and Applications*, 2018, doi: 10.1109/ FPL.2018.00044.

[29] C. Plessl, "Bringing FPGAs to HPC Production Systems and Codes," in *H2RC'18 workshop at Supercomputing (SC'18)*, 2018, doi: 10.13140/RG.2.2.34327.42407.

[30] T. Miyajima, T. Ueno, A. Koshiba, J. Huthmann, K. Sano, and M. Sato, "High-Performance Custom Computing with FPGA Cluster as an Off-loading Engine," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018.

[31] H. Kamal, S. M. Mirtaheri, and A. Wagner, "Scalability of communicators and groups in MPI," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 264–275.

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

[33] M. Saldaña, A. Patel, C. Madill, D. Nunes, D. Wang, P. Chow, R. Wittig, H. Styles, and A. Putnam, "MPI as a programming model for high-performance reconfigurable computers," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, no. 4, p. 22, 2010.

[34] J. Sheng, C. Yang, and M. Herbordt, "Application-Aware Collective Communication on FPGA Clusters," in *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, doi: 10.1109/ FCCM.2016.55.

[35] J. Sheng, Q. Xiong, C. Yang, and M. Herbordt, "Collective Communication on FPGA Clusters with Static Scheduling," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, 2016, doi: 10.1145/ 3039902.3039904.

[36] Q. Xiong, A. Skjellum, and M. Herbordt, "Accelerating MPI Message Matching Through FPGA Offload," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 191–1914, doi: 10.1109/ FPL.2018.00039.

[37] Q. Xiong, P. Bangalore, A. Skjellum, and M. Herbordt, "MPI Derived Datatypes: Performance and Portability Issues," in *25th European MPI Users' Group Meeting*, 2018, doi: 10.1145/ 3236367.3236378.

[38] A. Wachtel, *Boosting Scalability of InfiniBand-based HPC Clusters*, 2010.

[39] S. Kumar, A. Mamidala, P. Heidelberger, D. Chen, and D. Faraj, "Optimization of MPI collective operations on the IBM Blue Gene/Q supercomputer," *The International Journal of High Performance Computing Applications*, vol. 28, no. 4, pp. 450–464, 2014.

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

[41] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Elsevier, 2004.

[42] "OSU Micro-benchmarks." [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/

[43] D. Stanzione, B. Barth, N. Gaffney, K. Gaither, C. Hempel, T. Minyard, S. Mehringer, E. Wernert, H. Tufo, D. Panda, and P. Teller, "Stampede 2: The Evolution of an XSEDE Supercomputer," in *Practice and Experience in Advanced Research Computing on Sustainability, Success and Impact*, 2017.