Generalized Collective Algorithms for the Exascale Era

Michael Wilkins
Northwestern University
wilkins@u.northwestern.edu

Hanming Wang
Northwestern University
hanmingwang2022@u.northwestern.edu

Peizhi Liu
Northwestern University
peizhiliu2023@u.northwestern.edu

Bangyen Pham
Northwestern University
bangyen@u.northwestern.edu

Yanfei Guo
Argonne National Laboratory
yguo@anl.gov

Rajeev Thakur
Argonne National Laboratory
thakur@anl.gov

Peter Dinda
Northwestern University
pdinda@northwestern.edu

Nikos Hardavellas
Northwestern University
nikos@northwestern.edu

***Abstract*—Exascale supercomputers have renewed the exigence of improving distributed communication, specifically MPI collectives. Previous works accelerated collectives for *specific* scenarios by changing the radix of the collective algorithms. However, these approaches fail to explore the interplay between modern hardware features, such as multi-port networks, and software features, such as message size. In this paper, we present a novel approach that uses system-agnostic, generalized (i.e., variable-radix) algorithms to capture relevant features and provide broad speedups for upcoming exascale-class supercomputers.**

We identify hardware commonalities found on announced exascale systems and three omnipresent communication kernels (binomial tree, ring, and recursive doubling) that can be generalized to better leverage these features, creating 10 total implementations. For each kernel, we develop analytical models to intuit algorithm performance with varying radix values.

Experiments on the world's first exascale supercomputer (Frontier at ORNL) and a pre-exascale system (Polaris at ANL) show that our generalized algorithms outperform the baseline open-source and proprietary vendor MPI implementations by a significant margin, up to over 4.5x. We empirically determine optimal algorithms and parameter values, identifying where the analytical models are accurate and where hardware features directly determine performance. Most notably, we show how a *single, system-agnostic* implementation of a generalized algorithm can optimize for *multiple* hardware/software features across *multiple* systems.

***Keywords*-Exascale computing, collective communication, MPI**

I. Introduction

As the HPC community enters the "exascale" era, communication performance is more critical than ever. Frontier [3], the first exascale system on the TOP500 list, hightlights the need for improved communication as each multi-GPU/CPU node requires a steady flow of data. In this work, we focus on Message Passing Interface (MPI) collective operations, which are among the most popular communication primitives. Collectives consume 25–50% (or more) of the overall execution time of current/future applications [10], [26], [25], [35].

TABLE I: Our 10 Collective Algorithms

Base Kernel	Generalized Kernel	Collective Operations
Binomial	K-nomial	*MPI_Reduce*, *MPI_Bcast*, *MPI_Allgather*, *MPI_Allreduce*
Recursive Doubling	Recursive Multiplying	*MPI_Bcast*, *MPI_Allgather*, *MPI_Allreduce*
Ring	K-Ring	*MPI_Bcast*, *MPI_Allgather*, *MPI_Allreduce*

Both the main open-source implementations of the MPI standard, MPICH [1] (the focus of our work) and Open MPI [15], implement a limited set of algorithms per collective. Previous works have developed generalized collective algorithms (see §II and §VII), which enable users to optimize the radix of the algorithm. However, these efforts are specialized for specific scenarios (e.g., a specific network topology [34], small message size allreduce [32], or intranode broadcast [33]), and the broader efficacy of the strategy is unknown. As a result, generalized (i.e., variable-radix) algorithms appear very sparsely in current implementations of MPI, sacrificing performance on modern HPC systems.

In this work, we study the usefulness of generalized algorithms for exascale systems. We first identify hardware features shared among the upcoming exascale supercomputers and algorithm generalizations that could better leverage them (§II). Then, we design system-agnostic generalizations of three major communication patterns (i.e., kernels): binomial (§III), recursive doubling (§IV), and ring (§V) inspired by our identified generalizations [33], [32], [17]. We use our kernels to implement 10 algorithms for the four most common collective operations (see Table I). For each generalized algorithm, we create analytic models and compare with the non-generalized version, creating intuition regarding the optimal radix values.

Then, we marry intuition with empirical analysis. For our evaluation, we integrate our new algorithms into MPICH and experiment on Frontier and Polaris [4] (a pre-exascale system at ANL). Our collective algorithms improve performance by

1–4.5x (§VI). We compare our results with our analytical models. We identify performance variations across parameters, algorithms, scales and machines. We specify where the models are accurate, and we pinpoint production/system realities that overtake our theory in other cases. Finally, we also create a new algorithm selection configuration for MPICH, so that generalized algorithms may be leveraged automatically on exascale systems.

We summarize our contributions as follows:

- We develop generalized communication kernels that optimize for the commonalities of exascale systems, creating 10 collective algorithms;
- We create system-agnostic analytical models for each new algorithm to understand their theoretical performance;
- We achieve 1-4.5x speedups on Frontier, the world's first exascale system, and Polaris, a pre-exascale system, and create a new configuration to automate these speedups;
- We highlight how, surprisingly, a *single, system-agnostic* implementation of a generalized algorithm can optimize for *multiple* hardware features across *multiple* systems.

II. Background

Here we describe MPI collective operations and hardware/-software features of exascale systems. We identify algorithm generalization techniques that can better leverage them.

A. MPI Collective Operations

MPI is the de facto standard communication interface for HPC applications, and collective operations are MPI's most popular primitive. Collectives abstract away point-to-point messages in favor of program-wide communication patterns.

Collectives consume a large percentage of the overall runtime of HPC applications. Chunduri et al. found that MPI collectives alone account for 20%-50% of application runtime on two production supercomputers in 2018 [10]. A profile of the Exascale Computing Project's (ECP) Proxy Application Suite 4.0 found the expected workload of future exascale systems spend 40% or more of their overall runtime on collectives [35]. These studies indicate that optimizing MPI collectives will accelerate important applications.

B. Collective Performance on Exascale Hardware

Frontier and the next expected exascale supercomputers (e.g., Aurora, El Capitan) share multiple features that impact collective performance. Here we list the relevant features and pinpoint algorithm generalization techniques to leverage them.

1) Network Topology: Exascale networks (including Frontier) use the dragonfly topology [24], which is a two-layer topology design that was introduced relatively recently. Its fully connected groups and hierarchical design minimizes the latency between nodes at large scale. One advantage of dragonfly is that it uses high radix virtual switches and global adaptive routing to ensure that there exists a shortest path between any two nodes. Therefore, topology-aware generalized algorithms for traditional HPC interconnects (e.g., torus, hypercube) that use non-minimal routing [34] will not be effective. Instead, we design minimal communication algorithms that leverage other hardware features (discussed below).

2) Multi-Port Nodes & Message Buffering: In exascale supercomputers, high network bandwidth is necessary to support the computational power of multi-CPU/GPU nodes. To meet this need, exascale networks assign subsets of GPUs to dedicated network links. For example, each node on Frontier includes four 200 Gb/s links (one per 2 GPUs). Furthermore, non-blocking send/receive primitives in software enable the buffering of multiple messages simultaneously beyond the number of physical ports. Message buffering is critical to overlap the message submission latency of smaller messages.

Collective algorithms must employ multi-port functionality and message buffering to fully utilize exascale systems. However, in popular communication patterns such as binomial tree and recursive doubling, each process only communicates with one other process at a time, thus they only buffer a single message. To solve this challenge, we propose two generalized algorithms, k-nomial and recursive multiplying.

Previous works have presented generalizations for limited circumstances, namely intranode broadcast [33] and small message (<4kB) allreduce [32] We develop new generalized algorithms inspired by these strategies to exploit multi-port networks and message buffering. We use the variable radix of these algorithms to elegantly capture the interplay between hardware and software to optimize for exascale systems.

3) Intranode Links: Applications on multi-GPU node-sytems may assign a separate MPI process to each GPU. For example, applications on Frontier commonly use 8 MPI processes per node. These processes communicate via dedicated higher-bandwidth hardware links (e.g., NVLink, InfinityFabric) that provide higher performance compared to the internode network. However, the predominant communication kernel for large (i.e., bandwidth-bound) message sizes, the ring kernel, does not differentiate between link types, slowing the entire algorithm to match the slower connections.

To address this issue, we propose a generalized "k-ring" kernel. Previous work on a new reduction algorithm showed how a hierarchical strategy can better saturate a heterogeneous network structure [17]. Combining this idea with the ring kernel, we better utilize the high-bandwidth intranode links found on exascale systems.

In summary, by analysing the major factors for algorithm performance, we identify three promising generalizations (k-nomial, recursive multiplying, and k-ring), which all expose their radix as a tunable parameter. We proceed to explain our generalized algorithms and use analytical models to predict how changing the parameter values will affect performance.

III. Binomial Tree and K-nomial Tree Algorithms

The first communication kernel we generalize is the binomial tree algorithm. Binomial is typically the optimal choice for small message sizes (<16KB), where the limiting factor for performance is point-to-point latency. Binomial tree minimizes the effect of latency by overlapping communications.

A. The Binomial Tree Algorithm

A binomial tree is a recursive tree structure where the sub-tree at each non-leaf node is the same as the sub-tree at that node's first child. Figure 1 shows an example binomial tree communication for *MPI_Gather* on 6 processes and the prospective placement of two more processes. The identical sub-trees enable parallelism, and the first "round" of overlapped communications is highlighted in green.

B. Binomial Tree Algorithm Cost

We now define the cost model of the binomial algorithm given a specified number of processes p. We use the common (α, β) model [18]. In this model, the execution time of a point-to-point communication is $\tau = \alpha + \beta * n$. α (latency) represents the startup cost, β (bandwidth) is the per-byte cost, and n is message size (bytes). Intuitively, α determines performance for small messages, while $\beta * n$ controls larger messages. Collectives are composed of point-to-point messages between p processes, so we model them by scaling the equation by (p). For example, a naïve broadcast, where the root sends to every process sequentially, is thus $\tau = p(\alpha + \beta * n)$. For the reduction collectives, we also include γ (per-byte computation cost). Collective algorithms overlap point-to-point communications to reduce the impact of α or β. Note that we use the same symbols/model in all analyses.

The costs of binomial tree algorithms for the simpler collectives are shown in (1).

$$T(n,p) = \begin{cases} \log_2(p)\alpha + n\log_2(p)\beta & \text{Bcast} \\ \log_2(p)\alpha + n\log_2(p)\beta & \\ \quad + n\log_2(p)\gamma & \text{Reduce} \\ \log_2(p)\alpha + n\frac{p-1}{p}\beta & \text{Gather} \end{cases} \quad (1)$$

Allgather and allreduce are implemented using a gather or reduce followed by a bcast, as shown in (2).

$$T(n,p) = \begin{cases} \log_2(p)\alpha + n(\log_2(p) + \frac{p-1}{p})\beta & \text{Allgather} \\ \log_2(p)\alpha + n(\log_2(p) + \frac{p-1}{p})\beta & \\ \quad + n\log_2(p)\gamma & \text{Allreduce} \end{cases} \quad (2)$$

In these models, the recursive tree structure causes the latency overhead α to scale *logarithmically* with the number of processes, p. Hence, the algorithm performs well for latency-bound, small message operations.

C. The K-nomial Tree Algorithm

The goal of the k-nomial generalization is to further reduce the latency penalty for small message collectives. Thanks to overlapping communications, the latency cost of a basic binomial tree is the latency of a point-to-point communication times the depth of the tree. Therefore, decreasing the depth of the tree can reduce the overall communication latency.

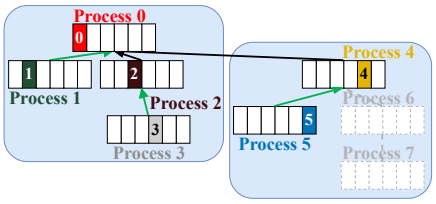


Fig. 1: Binomial tree algorithm for gather. The recursive structure allows all sub-trees to be processed in parallel.

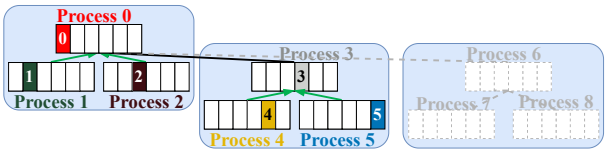


Fig. 2: Trinomial tree algorithm (k=3) for gather. Decreased tree depth increases the parallelism per subtree.

Binomial trees have an assumed radix of 2, which we generalize to create the k-nomial tree algorithm. Figure 2 shows a trinomial tree $(k = 3)$, still with 6 processes. In a full k-nomial tree, the sub-trees rooted at a non-leaf node is identical to the sub-tree at that node's first $(k-1)$ children. Nodes 6, 7, and 8 are, again, placeholders showing the structure of a complete trinomial tree. Notice how in Figure 1, adding a potential 8^{th} node would increase the depth of the tree. However, Figure 2 shows how a trinomial tree can hold up 9 nodes without increasing the depth.

Increasing the radix of k-nomial tree flattens the structure by overlapping communications within a given level of the tree. In Figure 2, the messages from processes 1 and 2 to 0 (and 4+5 to 3) are executed simultaneously as highlighted by the green arrows. To overlap these messages, we leverage multi-port/message buffering (§II-B2), so that a single endpoint can send/receive multiple messages simultaneously.

D. K-nomial Tree Algorithm Cost

K-nomial tree algorithms change constants in the cost model, allowing the user to tune the impact of each term. The costs of the k-nomial tree algorithms are shown in (3).

$$T(n,p,k) = \begin{cases} \log_k(p)\alpha + (k-1)n\log_k(p)\beta & \text{Bcast} \\ \log_k(p)\alpha + (k-1)n\log_k(p)\beta & \\ \quad + (k-1)n\log_k(p)\gamma & \text{Reduce} \\ \log_k(p)\alpha & \\ \quad + (k-1)n(\log_k(p) + \frac{p-1}{p})\beta & \text{Allgather} \\ \log_k(p)\alpha & \\ \quad + (k-1)n(\log_k(p) + \frac{p-1}{p})\beta & \\ \quad + (k-1)n\log_k(p)\gamma & \text{Allreduce} \end{cases} \quad (3)$$

Larger k values decrease the effect of latency (α) and increase the effect of the bandwidth (β). For very small message sizes, bandwidth is a non-factor, so we expect increasing k to improve performance. Smaller k values should perform better for larger messages when bandwidth is the limiting factor.

In these models, we assume multiport/message buffering enable perfect overlapping of messages with shared endpoints. The optimal k value and the performance gain from the k-nomial algorithm are dependent on this assumption. An ideal overlapping would result in an optimal k value for very small messages at or near p. However, it is possible that the physical number of network ports caps the number of overlapping communications per endpoint, lowering the optimal k.

IV. Recursive Doubling and Recursive Multiplying Algorithms

We now consider the recursive doubling algorithm, which performs best for small-to-medium message sizes (1B-512kB). For these sizes, latency is again the dominant bottleneck. In current MPI implementations, recursive doubling is commonly used for small-to-medium message sizes because it minimizes the number of sequential communication rounds.

A. The Recursive Doubling Algorithm

Recursive doubling is a pairwise exchange algorithm where during every round, each process is assigned a peer with which to exchange information. As an example, consider Figure 3, which depicts a recursive doubling allgather with 4 processes. In total, there are two communication rounds. In the first round, processes exchange data with peers that are $2^0 = 1$ apart. Peers are formed from groups of size $2^0 = 1$ between odd and even-numbered groups. In the second round, processes in odd and even groups of size $2^1 = 2$ exchange their accumulated data with peers that are $2^1 = 2$ apart. The amount of data exchanged doubles every round.

B. Recursive Doubling Algorithm Cost

The cost models for the recursive doubling algorithm assuming a power-of-two number of nodes are shown in (4).

$$T(n,p) = \begin{cases} \alpha \log_2 p + \beta n \frac{p-1}{p} & \text{Allgather, Bcast} \\ (\log_2 p)\,(\alpha + (\beta+\gamma)\,n) & \text{Allreduce} \end{cases} \tag{4}$$

The cost of round i is given by (5).

$$T_i(n,p) = \begin{cases} \alpha + \beta n \frac{2^{i-1}}{p} & \text{Allgather, Bcast} \\ \alpha + (\beta+\gamma)n & \text{Allreduce} \end{cases} \tag{5}$$

Like binomial, recursive doubling scales logarithmically with latency, making it a good choice for smaller messages.

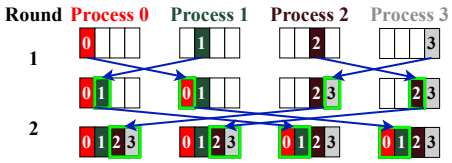


Fig. 3: Recursive doubling for allgather with 4 processes.

C. The Recursive Multiplying Algorithm

Recursive doubling, by only doubling the amount of data sent in each round, can induce unnecessary latency with many rounds of small message sizes. Our recursive multiplying algorithm balances the latency and bandwidth trade-off through the number/size of rounds.

Recursive multiplying introduces parameter k that controls the number of communication partners each round. For each round i, every process exchanges data between $k-1$ other processes spaced a multiple of k^{i-1} apart, with the specific pairings chosen by dividing p processes into k^i groups.

Figure 4 shows an example recursive multiplying implementation of allgather with $p = 9$ processes and $k = 3$. Despite the added processes, the allgather still completes in just 2 rounds. Sending more messages per round decreases the number of rounds, improving performance for small-medium messages.

D. Recursive Multiplying Algorithm Cost

The cost model of the recursive multiplying algorithm in (6) is similar to (4) except for the recursive base k.

$$T(n,p,k) = \begin{cases} \alpha \log_k p + \beta n \frac{p-1}{p} & \text{Allgather, Bcast} \\ (\log_k p) & \\ \cdot\,(\alpha + (\beta+\gamma)\,(k-1)n) & \text{Allreduce} \end{cases} \tag{6}$$

The parameter k increases the per-round cost of the recursive multiplying algorithm. The cost for the i^{th} round is now (7).

$$T_i(n,p,k) = \begin{cases} \alpha + \beta n \frac{(k-1)k^{i-1}}{p} & \text{Allgather, Bcast} \\ \alpha + (\beta+\gamma)(k-1)n & \text{Allreduce} \end{cases} \tag{7}$$

The per-round bandwidth and computation costs increase to accommodate multiple messages per round. The validity of this strategy once more depends on the overlapping capabilities of the multiport network and message buffering (§II-B2).

V. Ring and K-Ring Algorithms

Lastly, we consider the ring algorithm. Ring is used for larger messages, where the communication bottleneck shifts to bandwidth. The ring algorithm provides a bandwidth-optimal implementation by using neighbor-only communication.

A. The Ring Algorithm

In the ring algorithm, processes only communicate with their two neighboring processes. Each round, every process receives new data from its left neighbor and forwards the received data from the previous round to its right neighbor

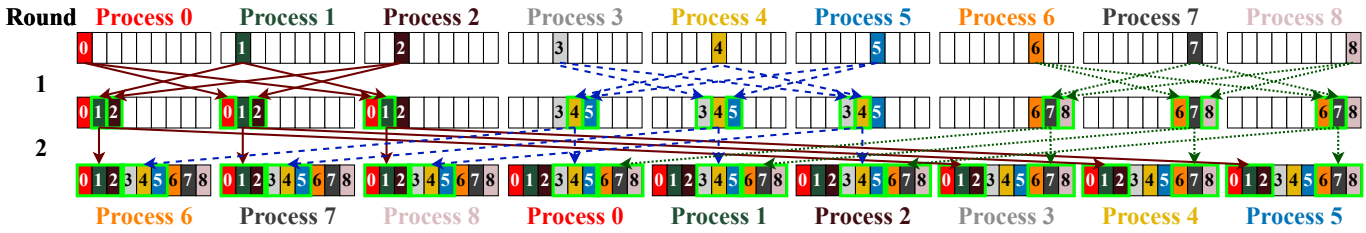


Fig. 4: Recursive multiplying for allgather. Each round, processes exchange with two other nodes using a power-of-3 offset.

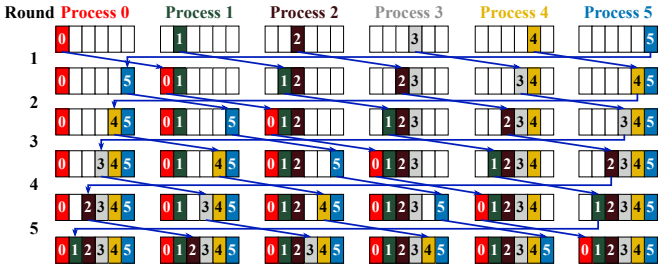


Fig. 5: Ring algorithm for allgather. In each round, processes forward data to their cyclic adjacent neighbors.

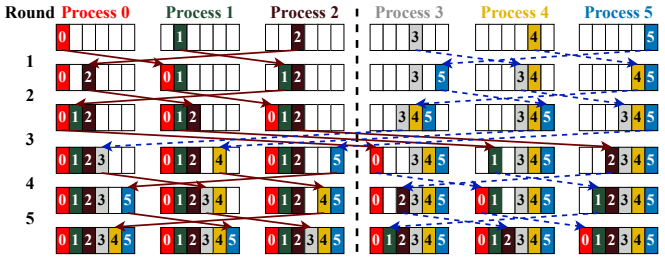


Fig. 6: K-ring algorithm for allgather. Two faster intranode rounds alternate with a slower internode round.

in a ring-like fashion. Figure 5 gives an example of the ring algorithm used for the allgather collective with 6 processes.

B. Ring Algorithm Cost

We define the cost model for the ring algorithm for allgather, allreduce, and part of bcast in (8).

$$T(n,p) = (p-1)T_i \tag{8}$$

Given p processes, the ring algorithm will have $(p-1)$ rounds of communication, where the single-round cost T_i is (9).

$$T_i(n,p) = \begin{cases} \alpha + \beta\frac{n}{p} & \text{Allgather, Bcast} \\ \\ \alpha + \beta\frac{n}{p} + \gamma\frac{n}{p} & \text{Allreduce} \end{cases} \tag{9}$$

When compared to the recursive doubling algorithm, ring has a worse latency term (log $\rightarrow$ linear) and equivalent bandwidth term (both linear). Nonetheless, ring is preferred for large messages in practice because the neighbor communication minimizes network hops and congestion, which would limit bandwidth.

Given sufficiently large message sizes, the cost of the ring algorithm reduces roughly to (10), which is independent of latency and the number of processes.

$$T(n) = \begin{cases} \beta n & \text{Allgather, Bcast} \\ \\ \beta n + \gamma n & \text{Allreduce} \end{cases} \tag{10}$$

C. The K-Ring Algorithm

The classic ring algorithm is optimized for bandwidth, but it does not consider the extremely high-bandwidth intranode links on modern supercomputers (§II-B3). When applications use one MPI process per GPU on exascale systems, the more powerful links create a discrepancy in communication cost between processes in the ring algorithm. This heterogeneity is unfavorable for the ring algorithm which has an implicit barrier between rounds, so processes with intranode neighbors are starved for data by the slower internode links.

To reduce the impact this bottleneck, our generalized k-ring algorithm breaks the communication into multiple, smaller "rings." k determines the size of the smaller ring groups; for p total processes, there are $\frac{p}{k}$ groups. Every process has two pairs of left (receive) and right (send) neighbors, one within its group and one with another group. For the communication pattern, the k-ring algorithm is carried out in a series of alternating intra-group communication rounds and a single inter-group round using the two ring structures.

An example of the k-ring algorithm for *MPI_Allgather* with 6 processes and group size $k = 3$ is shown in Figure 6. In the first 2 rounds, processes within groups communicate in rings of size k. Therefore, the third round is inter-group communication with processes passing data to their inter-group neighbors. Following the third round, another two rounds of intra-group communication complete the allgather.

The goal of the k-ring algorithm is to prevent bottleneck links from slowing the entire communication. Within the smaller rings, communication is faster and more consistent because processes are likely to be physically closer (e.g., within the same node) in the system topology. Inter-ring communication rounds are slower, but they are far less frequent (e.g., in Figure 6, there are 4 intra-ring rounds to only 1 inter-ring round).

D. K-Ring Algorithm Cost

With the per-round cost T_i, the k-ring algorithm cost model is split into $g(k-1)$ intra-group and $(g-1)$ inter-group communication rounds, where the number of groups is $g = \frac{p}{k}$. The intra-group and inter-group costs are shown in (11).

$$\begin{cases} T_{\text{intra}}(n,p,k) = g(k-1)T_i \\ \\ T_{\text{inter}}(n,p,k) = (g-1)T_i \end{cases} \tag{11}$$

Hence, the total cost is (12).

$$\begin{aligned} T(n,p,k) &= T_{\text{intra}} + T_{\text{inter}} \\ &= (p-1)T_i \end{aligned} \tag{12}$$

The advantage of k-ring is the reduction of data exchanged between groups. In the example shown in Figure 6, given each partition of size ϕ, the total inter-group data sent and received by Group 0 is 6ϕ. For the ring algorithm, the total inter-group data sent and received would be 10ϕ. To generalize this idea for p nodes and an intra-ring group size of k, the amount of data sent/received by a group for the k-ring algorithm is (13).

$$D_{\text{k-ring}}(n,p,k) = 2n\frac{p-k}{p} \tag{13}$$

This formula reduced to the classic ring algorithm (*k=1*) is (14).

$$D_{\text{ring}}(n,p) = 2n\frac{p-1}{p} \tag{14}$$

Should there be bandwidth bottlenecks between groups of processes, the generalized ring algorithm can be chosen with an appropriate group size k to reduce its effect.

Implementation of the k-ring algorithm for allgather and bcast are identical since bcast uses a "scatter-*allgather*" algorithm. The implementation of allreduce is slightly different as the partitions are offset by 1.

Given the difference between intranode and internode links on exscale systems, we expect the best radix value for k-ring to be the number of processes per node. However, there may be additional bottlenecks between nodes that are farther apart. The k-ring algorithm gives us the flexibility to explore the realities of the intranode and internode topologies.

VI. Evaluation

We now describe how we integrated our new algorithms into MPICH, detail our experimental methodology to test their performance, and analyze the results.

A. MPI Library Integration

We implemented each new algorithm in the MPICH source code based on the non-generalized version if it exists. Implementations range from 100–400 lines of code, with *MPI_Bcast* being the longest because of the multi-phase communication for recursive multiplying and k-ring. The largest burden was ensuring correctness for the many corner cases induced by our generalizations (e.g., non-uniform group sizes for the recursive multiplying and k-ring algorithms).

B. Experimental Methodology

Our performance evaluation uses Frontier, the world's first exascale supercomputer, at Oak Ridge National Laboratory. Frontier contains 9,408 compute nodes, each equipped with one 64-core AMD EPYC 7A53 CPU, four AMD MI250X (eight logical GPUs), and 512 GB of DDR4 memory. The GPUs within each node are connected to each other via Infinity Fabric and the network via 4x200 Gb/s links.

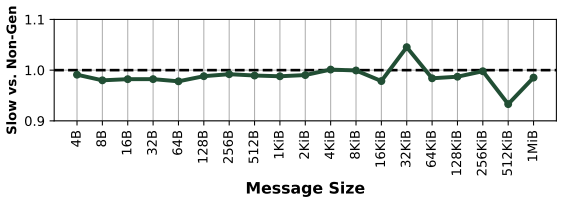
We compare our work against MPICH [1] and Cray MPI using the OSU microbenchmark suite [2]. For each experiment, we fixed MPICH's algorithm selection to the non-generalized version of the comparative algorithm, meaning we test our k-nomial implementation against MPICH's binomial implementation, etc. This practice isolates the improvement gained by generalization. For Cray MPI, we compare against its default selections to highlight the total speedup from the optimal generalized algorithm and parameter value. Cray MPI is the vendor-supported, state-of-the-art MPI implementation on Frontier. Therefore, these values represent the speedup a user could experience due to the adoption of our contributions.

We ran all our experiments on Frontier in both 32-node and 128-node configurations. We tested with both common programming models for the machine: 1 MPI process per node (PPN) (1 MPI process per GPU) and 8 MPI PPN (MPI + X). We performed each experiment multiple times to account for runtime variance and selected representative trials for visualization and analysis. Overall, we found our results to be very similar for both 32 nodes and 128 nodes and with 1 MPI process per node and 8 processes per node. We proceed to focus on our 128 node with 1-PPN results and highlight the scenarios where the results diverge.

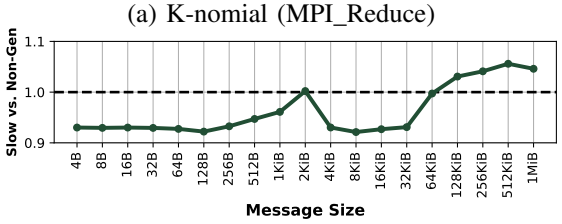
Beyond our core set of results, we also include experiments using 1024 nodes on Frontier to measure how the performance improvements from generalization scale to leadership-class applications. Due to limited resources and job length, we are only able to test our most promising configurations identified at smaller scale. Finally, we test how the improvements translate to other exascale hardware by using another system, Polaris. Polaris is a pre-exascale system for Aurora, the next anticipated exascale supercomputer at Argonne National Laboratory. Polaris contains 560 multi-GPU nodes connected by a dragonfly network, each with one 32-physical-core AMD EPYC Milan 7543P, four NVIDIA A100 GPUs fully connected with 600GB/s NVLink, 512GB of DDR4 memory, and two Slingshot network ports via 64 GB/s PCIe Gen4 for internode communication.

C. Frontier Results

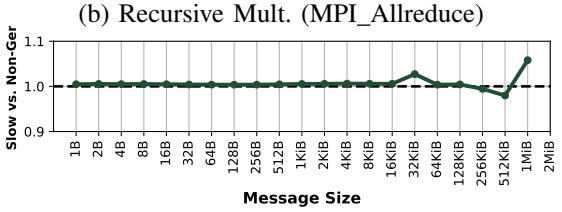
To summarize our experiments, we selected representative operations for each generalized kernel. For k-nomial, we chose *MPI_Reduce* because k-nomial is the only new generalized algorithm for *MPI_Reduce*. For recursive multiplying, we show results for *MPI_Allreduce* because it is the most popular collective for exascale applications [35]. For the k-ring algorithms, we selected *MPI_Bcast* because it is utilizes the



(a) K-nomial (MPI_Reduce)



(b) Recursive Mult. (MPI_Allreduce)



(c) K-ring (MPI_Bcast), 8 PPN

Fig. 7: Message Size vs. Slowdown (Lower is Better), 128 Nodes w/ 1 or 8 Process(es) Per Node on Frontier. Generalization does not result in slowdown.

MPI_Allgather ring algorithm, so we encapsulate both collectives and cover all four operations across these selections.

1) Generalized vs. Non-Generalized: In Figure 7, we plot the slowdown of the generalized implementation of each algorithm using the default parameter value ($k = 2$, $k = 2$, and $k = 1$) versus the fixed-radix implementation in MPICH. In this configuration, both algorithms are logically identical. These experiments ensure that generalization does not sacrifice performance from MPICH's low-level software tricks (e.g., bitwise operations) that require a fixed radix. In these graphs, a value of 1.0 means that the generalized and non-generalized algorithms have identical performance. Above 1.0 means that the generalized version is slower, and below 1.0 means that the generalized version is faster.

Across all three plots, we see that generalization has a minimal effect on the default radix performance. Slowdowns do not exceed 1.06x, and the generalized algorithms slightly outperform the fixed-radix version on average, thanks to our careful implementations and noise on the machine. With our implementation proven effective, we use our algorithms with the default radix as a baseline in future figures to further eliminate potential sources of variation.

2) Sensitivity to Parameter Value: Now, we describe how varying the parameter value affects performance. Figure 8 plots the parameter values on the x-axis and the latency on the y-axis in microseconds. We plot various message sizes as separate lines on the graph.

We begin with k-nomial in Figure 8(a). For the k-nomial algorithm, software message buffering is the dominant performance feature, and the latency/bandwidth trade-off of the message size determines the optimal parameter value. For very small messages (<128 bytes), large parameter values (k=128, i.e., k = the number of processes) greatly outperform small parameter values. As the message size increases, the optimal parameter value decreases. This smooth trend follows our analytical model. Message buffering and multiport functionality allow us to overlap many communications and garner significant speedups, which we will quantify in Figure 9.

These results reveal that the network's physical limitation of 4 network ports does not hamper performance. We believe this hardware restriction does not matter because the amount of shared-endpoint communications is relatively small for k-nomial. The one-sided nature of the algorithm means that each thread is only sending *or* receiving *k* messages per round.

Among the other collectives, *MPI_Bcast* had a pattern similar to *MPI_Reduce*. For *MPI_Allgather* and *MPI_Allreduce*, other algorithms outperformed k-nomial for all message sizes, rending their trends irrelevant. This result is logical because the k-nomial kernel maps more naturally to unbalanced collectives where there is a single node sending/receiving the data.

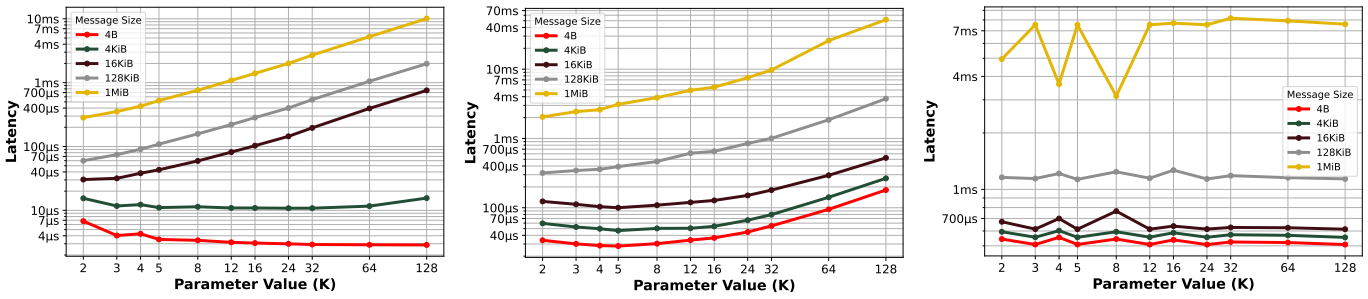
The next algorithm is recursive multiplying in Figure 8(b). For the recursive multiplying algorithm, the number of network ports is the dominant performance feature, and the number of ports per node determines the optimal k-value. For all message sizes regardless of magnitude, *k* values at or near 4 (the number of ports per node) are the best-performing choice for *MPI_Allreduce*. This result contradicts our expectations based on the analytical models. We believe it occurs because compared to k-nomial, the number of simultaneous messages increases much faster with larger k values. In recursive multiplying, each process sends *and* receives k messages per round, further stressing the multiport network.

MPI_Allgather and *MPI_Bcast* favor *k=4* or a multiple of *k=4*. They do not require receiver-side computation like a reduction, which creates less predictable performance.

The last algorithm is k-ring in Figure 8(c), which we test with 8 processes per node for the 1-MPI-process-per-GPU programming model. For the k-ring algorithm, the intranode links are the dominant performance feature, and the number of processes per node determines the optimal k-value. For larger message sizes, $k = 8$ is the most performant parameter value. Despite the analytic model not presenting a clear benefit of k-ring, we measure a significant improvement. When $k = 8$, the "intragroup" and "intergroup" communication steps are effectively "intranode" and "internode" steps, allowing much of the communication to leverage the superior intranode interconnect without implicit synchronization with internode messages.

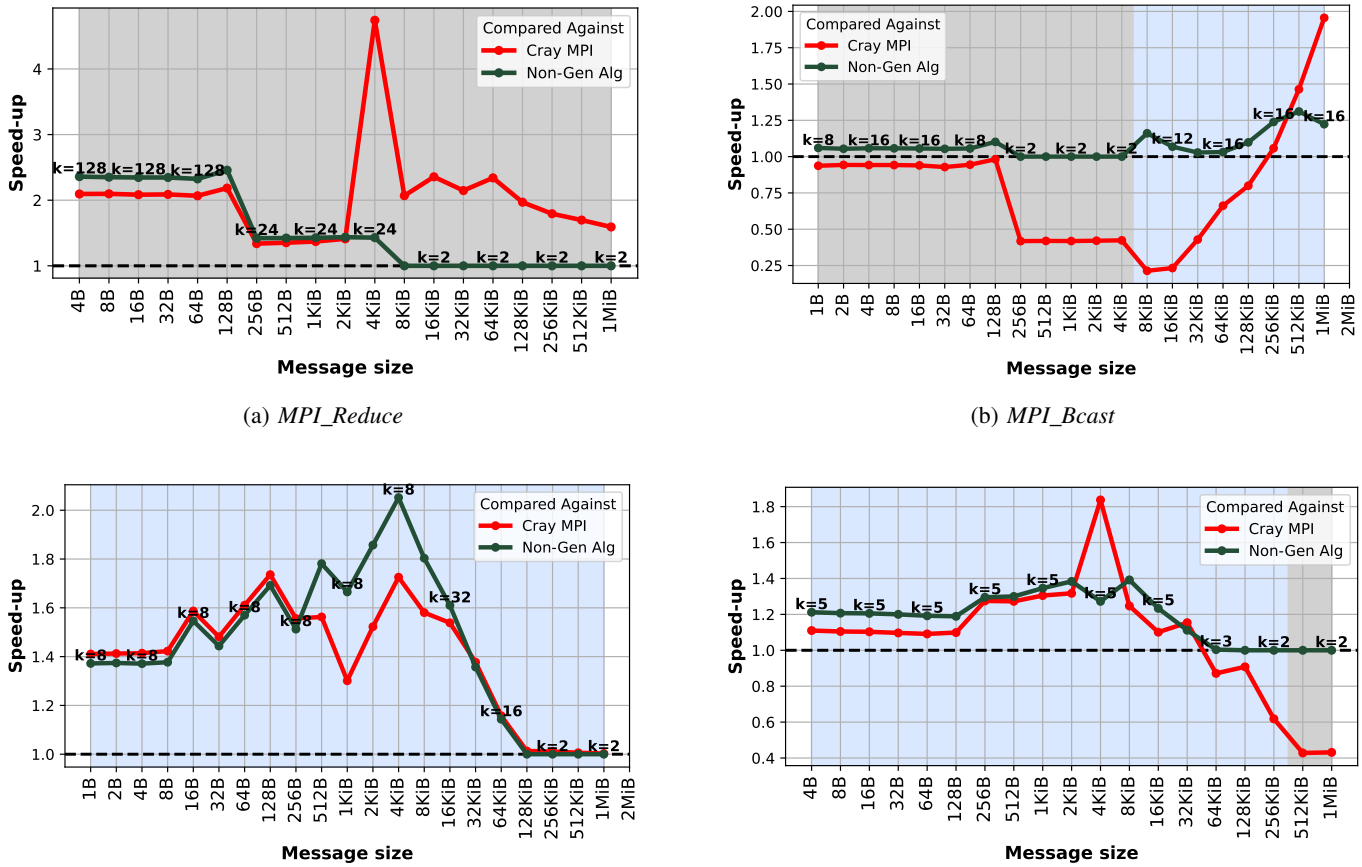
As part of the *MPI_Bcast* algorithm, *MPI_Allgather* naturally sees similar benefit from k-ring. For *MPI_Allreduce*, the reduce-scatter-allgather algorithm (which can also leverage the *MPI_Allgather* k-ring algorithm) generally outperforms ring for large message allreduces.

3) Speedup: In Figure 9, we show the speedup of each collective operation by selecting the optimal algorithm for each message size using our complete results. We denote the algorithm using a color overlay (grey is k-nomial, blue is



(a) *K-nomial (MPI_Reduce)* (b) *Recursive Mult. (MPI_Allreduce)* (c) *K-ring (MPI_Allgather), 8 PPN*

Fig. 8: Parameter Value (K) vs. Latency (Lower is Better), 128 Nodes w/ 1 or 8 Process(es) Per Node on Frontier. For all algorithms, the parameter value has a significant impact on performance.



(a) *MPI_Reduce* (b) *MPI_Bcast*

(c) *MPI_Allgather* (d) *MPI_Allreduce*

Fig. 9: Message Size vs. Speedup (Higher is Better), 128 Nodes w/ 1 Process Per Node for on Frontier. Generalization provides varying speedups over both baselines in most cases.

recursive multiplying). We did not encounter situations where k-ring is optimal over recursive doubling, including additional experiments to study even larger message sizes. We believe the k-ring algorithm gets outperformed because jobs of smaller size are dispersed across the 9000+ nodes in the system, eliminating k-ring's neighbor communication advantage.

The X-axis is once again the message size, and the Y-axis is the speedup over the two baselines. The dark green line represents the speedup over the default radix of the algorithm to show the speedup from generalization alone. The red line represents the speedup over Cray MPI. The Cray MPI baseline showcases how a current production user stands to benefit from our contributions. Cray MPI occasionally outperforms our algorithms most likely due to other algorithms, which may be proprietary or hardware-accelerated. In these cases, generalization is orthogonal and may provide additional speedup despite our results. On the other hand, when the speedup vs. Cray MPI is much greater than vs. the other baseline, Cray MPI is likely using a sub-optimal algorithm.

Now we analyse the results shown in each figure. The first

one is *MPI_Reduce* in Figure 9(a), for which k-nomial is our only generalized algorithm. As expected from the previous section, the speedup starts out high (over 2.0x) and erodes as the message size increases for the default-parameter-value baseline. Surprisingly, the Cray MPI baseline matches the small-message speedup, meaning that it is also employing the binomial algorithm instead of the more competitive "linear" algorithm. Then, the speedup over Cray MPI soars to over 4.5x, where we believe it is incorrectly switching algorithms.

The second one is *MPI_Bcast* in Figure 9(b), which typically sees little speedup. For message sizes under 256KB, we observe small (1.05x-1.2x) speedups over binomial/recursive doubling and no speedup over Cray MPI. For large messages, recursive multiplying accomplishes its only significant performance improvements (up to nearly 2.0x over Cray MPI) with *k=16*. For *MPI_Bcast*, multiples of four are best for the recursive multiplying parameter value.

Third is *MPI_Allgather*, whose speedups are shown in Figure 9(c). For nearly all message sizes, we see significant (1.4x-2.0x) speedups over both baselines. Similar to *MPI_Bcast*, multiples of four are the best parameter value.

The last is *MPI_Allreduce* in Figure 9(d). As we saw in Figure 8(b), recursive multiplying prefers parameter values near 4, and it generates significant (1.2x-1.8x) speedups. While the optimal parameter value is *k=5*, *k=4* is less than 1% worse on average, meaning the slight win by *k=5* is likely noise. For the largest message sizes in our range, the performance improvement tails off as expected from our analytical models.

D. Large-Scale Frontier Results

With performance trends established with smaller node counts, we now study how the performance gains scale up to 1024 nodes. To keep our experiments tractable at this size, we could no longer perform a sweep of all parameter values. Instead, we specifically study how our most promising trends at smaller scale translate to larger scale. We seek to show that these parameter values provide turnkey performance improvements for leadership-class applications.

In Figure 10, we present three figures representing 1024 node performance for the best k-nomial and recursive multiplying scenarios from smaller scale. In these graphs, we plot the message size on the x-axis again and the latency in microseconds of the various configurations on the y-axis. We include our speedup baselines (Cray MPI and *k=2*) as lines on the graph for easy visual comparison.

In Figure 10(a), we see that our performance trends for *MPI_Reduce* k-nomial remain intact; larger parameter values provide significant speedup at smaller message sizes. Interestingly, the parameter value equal to the number of processes (1024) always performs worse than *k=128*. It appears that at large scale, the parameter value has an upper bound.

In Figures 10(b)-(c), we present larger scale performance for *MPI_Allgather* and *MPI_Allreduce*. These experiments created the most consistent speedups in our smaller-scale tests, and those trends are replicated here. While there is some noise in the *MPI_Allreduce* results (e.g., 512KB performs worse than some larger message sizes), we observe consistent speedup from *k=4* and *k=8* until large message sizes.

Overall, our large-scale experiments show how generalized algorithms can provide meaningful performance gains for leadership-class use cases.

E. Polaris Comparison

We conclude our evaluation by exploring how generalized algorithms perform on other pre-exascale hardware, specifically Polaris. For these plots, shown in Figure 11, we use the same style as Figure 8. We seek to determine if the same trends are present on a different pre-exascale system architecture.

For k-nomial and recursive multiplying (Figures 11(a)-(b)), the results match expectations. Just as on Frontier, the optimal k-nomial parameter value for very small messages is close to the number of processes and decreases as message sizes increases. Again matching Frontier, the optimal recursive multiplying parameter value is four or eight, which are the smallest multiples of the two ports per node on Polaris.

For k-ring, however, the parameter value shows minimal affect. Unlike Frontier, Polaris' nodes are fully connected with equal bandwidth between every pair of GPUs. This architecture is less compatible with the k-ring algorithm because many links within a node go underutilized.

Overall, both k-nomial and recursive multiplying show that our generalized algorithm findings translate from one exascale system to another.

F. Evaluation Summary

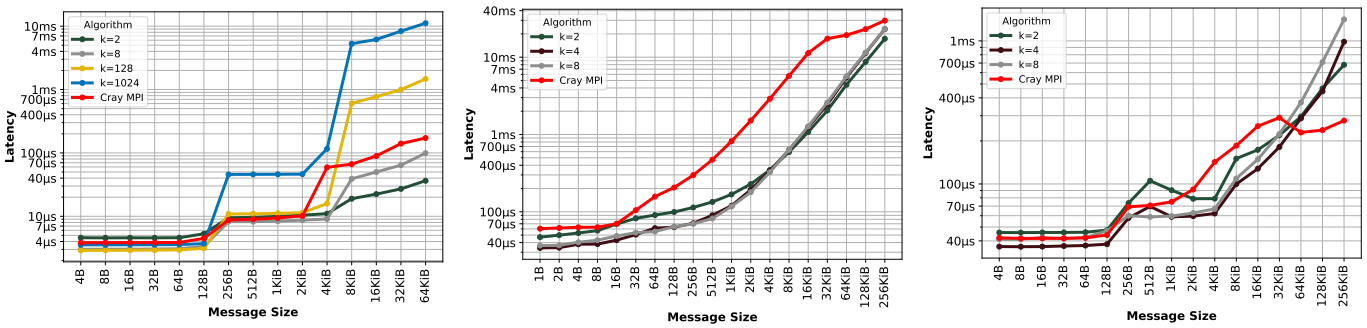
Overall, our experimental analysis generated many new findings. For k-nomial, we found the software features (message buffering) control performance, and that our analytical models are fairly accurate for optimizing the generalization. For recursive multiplying and k-ring, hardware features (multiport/intranode links) dominate performance, and empirical analysis contradicted our analytical intuition. We showed how these same trends also achieve significant speedups at larger scale and other exascale hardware.

G. Selection Configuration

When collecting our results, we exhaustively benchmarked every algorithm in MPICH to determine the optimal algorithms/parameters. Using this data and our analyses, we created a new algorithm/parameter selection configuration file that incorporates our generalized algorithms. Just by changing one environment variable to point to our new configuration, MPICH users can automatically and transparently leverage the speedups we uncover in this work.

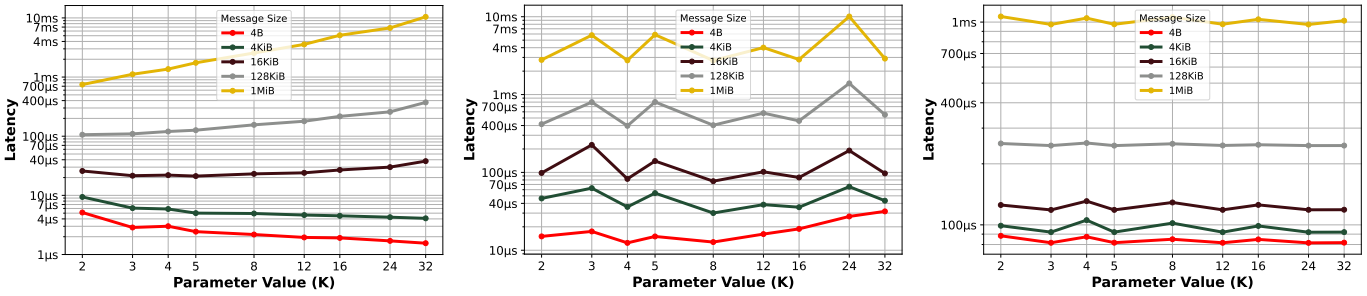
H. Considerations

To ensure the stability of our results, we re-executed each microbenchmark 4-10x (depending on execution length) within each trial. We included repetition at every level of our experimental methodology (within the microbenchmarks, re-running the microbenchmarks, and running multiple separate jobs on the systems). Still, when re-running experiments to



(a) *MPI_Reduce (K-nomial)* (b) *MPI_Allgather (Recursive Mult.)* (c) *MPI_Allreduce (Recursive Mult.)*

Fig. 10: Message Size vs. Latency (Lower is Better), 1024 Nodes w/ 1 Process Per Node for *MPI_Reduce*, *MPI_Allgather*, and *MPI_Allreduce* on Frontier. The speedups from generalization are maintained at large scale.



(a) *K-nomial (MPI_Reduce)* (b) *Recursive Mult. (MPI_Allgather)* (c) *K-ring (MPI_Bcast), 4 PPN*

Fig. 11: Parameter Value (K) vs. Latency (Lower is Better), 32 Nodes w/ 1 or 4 Process(es) Per Node on Polaris. The trends in how the parameter value affects performance are similar to those observed on Frontier (Figure 8).

select representative trials and develop our understanding, we encountered significant run-to-run variance, which changed the optimal algorithm selections and parameter values.

These effects are previously documented [21], but they are less well understood, particularly on exascale hardware. Therefore, it is best to view our conclusions as guidelines or heuristics. We found these guidelines produced consistent, significant speedups over both the fixed-radix algorithms and Cray MPI, but we do not claim to have analytically or empirically determined the optimal algorithms/parameters for all cases. Instead, autotuning tools show the ability to incorporate the effects of run-to-run variance [22], [40], [39], and we believe integrating generalized algorithms with these tools is an exciting future direction.

VII. Related Work

Many past works have optimized collective algorithms. Seminal work on collective algorithms by Thakur et al. [36], [37] laid the foundations for the standard set of collective implementations in MPICH [1]. Other important works include Bruck's algorithm [7], the n-way dissemination barrier by Hoefler et al. [19], and the ring-based all-reduce algorithm designed by Patarasuk et al. [29].

More similar to this paper, others have also created generalized collective algorithms for specific scenarios. Ruefencht et al. proposed a generalization of the recursive doubling algorithm for *MPI_Allreduce* for small message sizes [32]. Ruhela et al. first utilized the k-nomial algorithm to optimize intranode *MPI_Bcast* [33]. *MPI_Allreduce* k-nomial appears in Intel MPI, but its use case is unexplained. Hasanov et al. created a hierarchical structure across reduction algorithms [17]. Recently, Fan et al. generalized the Bruck's algorithm [7] by developing the padded Bruck and two-phase Bruck algorithms for nonuniform all-to-all communication [12]. Our work is inspired by and goes beyond these previous efforts by showing how a single generalized kernel can optimize for multiple collectives on multiple systems. Additionally, we identify how the hardware/software features of exascale machines pair with specific generalization techniques, and we perform new modelling and empirical analysis to explain how these features are responsible for the observed speedups.

To further improve collective performance on new and emerging hardware, many works focus on the development of new, topology-aware collective algorithms. Bienz et al. designed a locality-aware Bruck allgather [6]. Gong et al. proposed a set of network-aware algorithms for MPI bcast, reduce, gather, and scatter on cloud platforms [14]. Most recently, Feng et al. simulated collective algorithms specially designed for the standard dragonfly topology [13]. Our approach is more general because we do not incorporate the topology directly into our algorithms; instead, we design system-agnostic algorithms that consider both the topology and other features (e.g., multi-port, etc.).

The rising popularity of machine learning has motivated GPU-centric research. Cai and Liu et al. proposed the Synthesized Collective Communication Library (SCCL) to synthesize

optimal algorithms on specific GPU topologies [8]. Leveraging new, collective-specific hardware, Haghi et al. offloaded collective operations to in-switch hardware accelerators with two additional modules: a Collective Control Module and a Reduction Unit [16]. Awan et al. pipelined bcast operations during deep learning workloads on GPU clusters [5]. Also for distributed deep learning, Cho et al. [9] decomposed allreduce operations into parallel reduce-scatter and allgather operations. These network/application-specific designs are restricted to the contexts for which they were created. By contrast, our algorithms are transparent to both applications and hardware, meaning they can impact a wider variety of systems and users.

Selecting the best collective algorithm for different scenarios requires careful tuning. Many previous works attempt to accomplish this feat through analytical models [38], [11], [31], [30], [27], [28]. While these models can be very accurate, they would require additional arduous customization for new systems and algorithms. We believe that more recent work using machine learning (ML) to select collective algorithms is the most promising direction to incorporate our contributions [20], [40], [39], [23]. These approaches treat collective algorithms as a black box and rely on ML to learn their performance trends. These models could be augmented to predict both the best algorithm and parameter value for a given scenario, obviating any need for manual tuning.

VIII. Conclusion

In this work, we use a novel, comprehensive approach to create many new collective algorithms for exascale. By identifying algorithm generalizations that leverage the features of exascale systems, we provide a wide-sweeping speedup for multiple popular collective operations for two distinct exascale and pre-exascale systems (Frontier and Polaris).

As HPC expands to exascale and beyond, new machines will continue to increase in size and complexity. Generalized collective algorithms are a great fit for more complex systems because they expose easy-to-tune parameters, revealing the best solution for each system. In the future, we plan to further alleviate this burden by tying generalized algorithm tuning into autotuning frameworks.

IX. Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357, and by the U.S. National Science Foundation via award CCF-2119069.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research also used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] "MPICH." [Online]. Available: https://www.mpich.org

[2] "OSU micro-benchmarks." [Online]. Available: https://mvapich.cse.ohio-state.edu/benchmarks/

[3] "Frontier user guide," 2023. [Online]. Available: https://docs.olcf.ornl.gov/systems/frontier_user_guide.html

[4] "Polaris user guide," 2023. [Online]. Available: https://docs.alcf.anl.gov/polaris/getting-started/

[5] A. A. Awan, K. V. Manian, C.-H. Chu, H. Subramoni, and D. K. Panda, "Optimized large-message broadcast for deep learning workloads: MPI, MPI+ NCCL, or NCCL2?" *parallel computing*, vol. 85, pp. 141–152, 2019.

[6] A. Bienz, S. Gautam, and A. Kharel, "A locality-aware Bruck allgather," in *Proceedings of the 29th European MPI Users' Group Meeting*, 2022, pp. 18–26.

[7] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 11, pp. 1143–1156, 1997.

[8] Z. Cai, Z. Liu, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, and O. Saarikivi, "Synthesizing optimal collective algorithms," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 62–75. [Online]. Available: https://doi.org/10.1145/3437801.3441620

[9] M. Cho, U. Finkler, D. Kung, and H. Hunter, "Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 241–251, 2019.

[10] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI usage on a production supercomputer," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 386–400.

[11] G. E. Fagg, J. Pjesivac-Grbovic, G. Bosilca, T. Angskun, J. Dongarra, and E. Jeannot, "Flexible collective communication tuning architecture applied to Open MPI," in *Euro PVM/MPI*, 2006.

[12] K. Fan, T. Gilray, V. Pascucci, X. Huang, K. Micinski, and S. Kumar, "Optimizing the Bruck algorithm for non-uniform all-to-all communication," ser. HPDC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 172–184. [Online]. Available: https://doi.org/10.1145/3502181.3531468

[13] G. Feng, D. Dong, and Y. Lu, "Optimized MPI collective algorithms for dragonfly topology," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–11.

[14] Y. Gong, B. He, and J. Zhong, "Network performance aware MPI collective communication operations in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 11, pp. 3079–3089, 2015.

[15] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open MPI: A high-performance, heterogeneous MPI," in *2006 IEEE International Conference on Cluster Computing*. IEEE, 2006, pp. 1–9.

[16] P. Haghi, A. Guo, Q. Xiong, R. Patel, C. Yang, T. Geng, J. T. Broaddus, R. Marshall, A. Skjellum, and M. C. Herbordt, "FPGAs in the network and novel communicator support accelerate MPI collectives," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–10.

[17] K. Hasanov and A. Lastovetsky, "Hierarchical redesign of classic MPI reduction algorithms," *The Journal of Supercomputing*, vol. 73, no. 2, pp. 713–725, 2017.

[18] R. W. Hockney, "The communication challenge for MPP: Intel Paragon and Meiko CS-2," *Parallel computing*, vol. 20, no. 3, pp. 389–398, 1994.

[19] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm, "Fast barrier synchronization for InfiniBand," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, 2006, pp. 7 pp.–.

[20] S. Hunold, A. Bhatele, G. Bosilca, and P. Knees, "Predicting MPI collective communication performance using machine learning," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 259–269.

[21] S. Hunold and A. Carpen-Amarie, "Reproducible MPI benchmarking is still not as easy as you think," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3617–3630, 2016.

[22] ——, "Autotuning MPI collectives using performance guidelines," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 2018, pp. 64–74.
[23] S. Hunold and S. Steiner, "OMPICollTune: Autotuning MPI collectives by incremental online learning," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2022, pp. 123–128.
[24] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 77–88, 2008.
[25] B. Klenk and H. Fröning, "An overview of MPI characteristics of exascale proxy applications," in *International Supercomputing Conference*. Springer, 2017, pp. 217–236.
[26] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of MPI usage in open-source HPC applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.
[27] X. Luo, W. Wu, G. Bosilca, Y. Pei, Q. Cao, T. Patinyasakdikul, D. Zhong, and J. Dongarra, "HAN: A hierarchical autotuned collective communication framework," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 23–34.
[28] E. Nuriyev and A. Lastovetsky, "Accurate runtime selection of optimal MPI collective algorithms using analytical performance modelling," *arXiv preprint arXiv:2004.11062*, 2020.
[29] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.
[30] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, 2007.
[31] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "MPI collective algorithm selection and quadtree encoding," *Parallel Computing*, vol. 33, no. 9, pp. 613–623, 2007.
[32] M. Ruefenacht, M. Bull, and S. Booth, "Generalisation of recursive doubling for allreduce: Now with simulation," *Parallel Computing*, vol. 69, pp. 24–44, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819117301199
[33] A. Ruhela, B. Ramesh, S. Chakraborty, H. Subramoni, J. Hashmi, and D. Panda, "Leveraging network-level parallelism with multiple process-endpoints for mpi broadcast," in *2019 IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*. IEEE, 2019, pp. 34–41.
[34] P. Sack and W. Gropp, "Faster topology-aware collective algorithms through non-minimal communication," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 45–54, 2012.
[35] N. Sultana, M. Rüfenacht, A. Skjellum, P. Bangalore, I. Laguna, and K. Mohror, "Understanding the use of message passing interface in exascale proxy applications," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 14, p. e5901, 2021.
[36] R. Thakur and W. D. Gropp, "Improving the performance of collective operations in MPICH," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, J. Dongarra, D. Laforenza, and S. Orlando, Eds. Springer Berlin Heidelberg, 2003, pp. 257–267.
[37] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High-Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, Spring 2005.
[38] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically tuned collective communications," in *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 2000.
[39] M. Wilkins, Y. Guo, R. Thakur, P. Dinda, and N. Hardavellas, "ACCLAiM: Advancing the practicality of MPI collective communication autotuning using machine learning," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2022, pp. 161–171.
[40] M. Wilkins, Y. Guo, R. Thakur, N. Hardavellas, P. Dinda, and M. Si, "A FACT-based approach: Making machine learning collective autotuning feasible on exascale systems," in *2021 Workshop on Exascale MPI (ExaMPI)*. IEEE, 2021, pp. 36–45.