Parameterized Algorithms and Parameter Selection for Fast GPU-GPU Collective Communication

Peizhi Liu
Northwestern University
peizhi.liu@u.northwestern.edu

Sean Rhee
Northwestern University
seanrhee2024@u.northwestern.edu

Michael Wilkins
Argonne National Laboratory
mjwilkins18@gmail.com

Peter Dinda
Northwestern University
pdinda@northwestern.edu

***Abstract*—High-performance collective communication among GPUs in modern supercomputers is crucial for enabling many applications. Complex hierarchical interconnects between GPU devices necessitate collective algorithms that can effectively leverage the underlying network topology. We present parameterized algorithms for two GPU-to-GPU collectives, *Allgather* and *Allreduce*, as well as an optimized permutation kernel used to further enhance GPU collective communication. By employing a LogGP-based model calibrated with real machine measurements, we can efficiently simulate various parameter choices to identify optimal settings for specific device allocations and message sizes. Our comprehensive evaluation on NCSA Delta and Argonne Polaris supercomputers demonstrates that our parameterized algorithms can achieve, on average, a 20% speedup over their non-parameterized counterparts, with our parameter selection process capturing 98% of the potential speedup.**

***Index Terms*—high performance computing, graphics processing units, collective communication.**

I. Introduction

State-of-the-art supercomputers in high-performance computing (HPC) have adopted graphics processing units (GPUs) as their main computing hardware, accelerating diverse scientific applications like artificial intelligence [10]. These accelerators have led to the development of low-latency, high-throughput interconnects within hierarchical networks (§II-A).

To effectively utilize these networks, collective communication operations, or *collectives*, are the most common primitives. Collectives, which abstract point-to-point communication, are crucial, yet bottleneck-prone operations in HPC (§II-B). The algorithms that implement collective operations are therefore essential for minimizing communication time and are the focus of this paper. Existing optimizations for GPU networks lack scalable, performant solutions for HPC (§II-C).

One recent study introduced parameterized algorithms for optimized, scalable communication schedules (§II-D). These algorithms include a user-adjustable parameter that, when properly configured, can lead to significant speedups. However, the study concentrated on CPU-based communications, leaving the effects on GPU communication unexamined.

We demonstrate that simple, yet non-obvious parameterizations effectively optimize GPU-to-GPU collective communication. We develop parameterized *ring*, *recursive doubling*, and *distance halving, recursive doubling* algorithms for *Allgather* and *Allreduce* on GPU clusters (§III). We furthermore harness the distinctive features of GPUs to implement a device-specific optimization, notably an in-place permutation kernel, for the *distance halving, recursive doubling*, *Allgather* algorithm and its parameterization (§III-C).

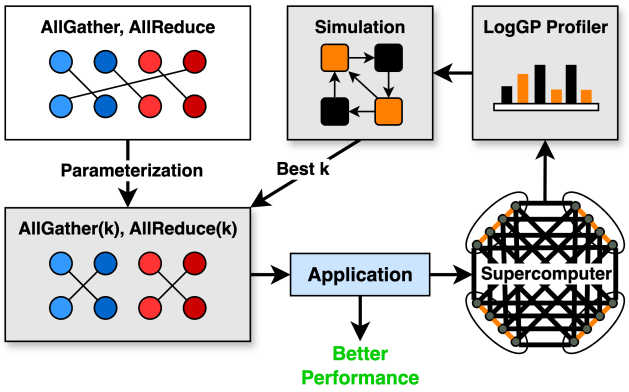


Fig. 1: Paper overview. New parameterized algorithms, accurately deployed thanks to simulation, result in significant speedups on large-scale GPU-based supercomputers.

To fully harness these parameterized algorithms, we must select the correct parameter values. Using the well-known LogGP model [5], we design a novel approach to simulate collective algorithms (§IV). To improve the accuracy of the model, we introduce the notions of device and node-level network contention. As a result, the simulator can quickly determine the best parameter value for a given algorithm/scale/message size, allowing users to easily achieve the maximum speedup from our approach.

We conduct a thorough evaluation on two large-scale supercomputers (NCSA Delta and ANL Polaris) to validate the usefulness of our algorithms and accuracy of our parameter selection process (§V). Our results showcase how GPU-based parameterized algorithms achieve speedups of 22% on average over the baseline, non-parameterized algorithms similar to those currently implemented in state-of-the-art libraries. Fig. 1 illustrates the narrative of this paper.

Our specific contributions are:

- We propose six parameterized collective algorithms for the *Allgather* and *Allreduce* collectives and show that collective algorithm parameterization is well-suited to optimize GPU-to-GPU communication (§III).
- We introduce an innovative GPU-specific optimization, an in-place permutation kernel, enhancing parameterized algorithms specifically for GPUs (§III-C).

- We develop a novel simulation-based approach for algorithmic parameter selection, achieving 98% of the performance compared to optimal selection (§IV).
- We conduct a comprehensive evaluation on two supercomputers, NCSA Delta and Argonne Polaris, demonstrating the effectiveness of algorithmic parameterization in GPU-to-GPU communication across varying scales and device allocations (§V).

Our code is available at github.com/PICCL-Project/PICCL.

II. Background and Related Work

A. Leveraging Modern GPU and Node Networks

GPU hardware has become increasingly ubiquitous in HPC systems, including the world's first three exascale supercomputers [2]–[4]. GPUs within a node are connected via high-bandwidth, low-latency connections such as NVIDIA's NVLink or AMD's InfinityFabric. These intranode GPU links add an additional layer to multi-port, multi-level hierarchical networks. For example, ANL Polaris, one of this paper's evaluation systems, uses 600 Gbps NVLink to fully connect the four GPUs within each node. The nodes are connected with two 200 Gbps Slingshot network cards into a dragonfly topology [19]. The performance gap between GPU-GPU direct connections and the internode network necessitates new collective algorithms to optimize the use of these links.

The multi-port capabilities of these networks also require careful consideration. Multiple network ports allow multiple simultaneous transfers, which can reduce communication bottlenecks and improve overall throughput.

The SIMD (Single Instruction, Multiple Data) nature of GPUs presents unique opportunities for developing novel communication algorithms. GPUs are inherently designed to perform parallel computations, executing the same operation across multiple data points simultaneously. This parallelism can be harnessed to accelerate communication tasks.

B. Importance of Collective Communication

Collective communication, often referred to as collective operations or collectives, serves as an abstraction layer over the message-passing model in distributed systems. Rather than designating specific senders and receivers for each individual message, programmers can employ collectives to define communication patterns for entire groups of processes.

Collectives often represent a significant bottleneck in HPC applications. A recent survey revealed that collective operations can consume up to 90% of the execution time in distributed AI training workloads [11]. Some scientific applications spend as much as 80% of their total execution time in communication [18]. Clearly, optimizing collective communication algorithms can have far-reaching effects.

We focus here on two prevalent collective communication operations: *Allgather* and *Allreduce*. Both operations facilitate many-to-many communications among processes. The *Allgather* operation collects data from all participating processes and distributes it to every process in the group. In contrast, the *Allreduce* operation aggregates data from all processes, typically using an arithmetic operation, and then disseminates the result to each process. These collectives are essential for key applications, such as distributed AI [23], [28].

C. Optimizing Collective Communication for GPUs

Many previous works have offered new algorithms for collectives. However, these algorithms are either older, generic works [8], [15], [21], [25], [26], designed for specific internode network topologies [13], [22], or only consider specific collectives (e.g., *MPI_Alltoall* [12]). To specifically optimize GPU-GPU collective communication, NVIDIA, AMD, and Intel have all developed their own vendor-specific collective libraries: NCCL [20], RCCL [6], and oneCCL [17] respectively. However, the algorithms implemented in these libraries do not generalize well to arbitrary heterogeneous network topologies, leaving performance on the table.

To overcome the performance issues with current vendor "CCLs", recent work synthesized collective algorithms using SAT solvers [7], [24]. This approach can provide optimal performance for fixed cluster topologies. However, it is intractable on large-scale HPC systems. The SAT solver may take hours to complete or fail to converge, especially for larger process counts. This is exacerbated by the fact that schedulers on space-shared supercomputers will assign different nodes (with different effective topologies) for each job, demanding per-job synthesis. Therefore, this approach has not been adopted by the HPC community, seeing only select use by cloud providers.

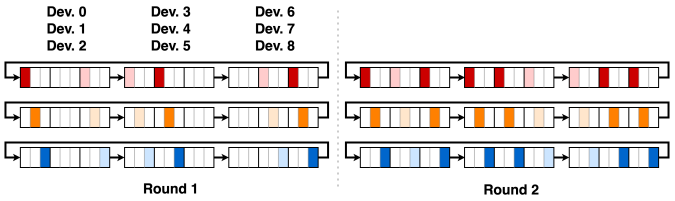
D. Generalized Collective Algorithms

A recent addition to this corpus showed how "generalized" algorithms are particularly effective for modern, hierarchical networks [27]. Generalized algorithms expose a parameter, often referred to as the "parameter," "radix" or symbolically as "k" or "k-value," that can be tuned to optimize the communication pattern. This work found that recursive multiplying, a generalization of the well-known recursive doubling algorithm, is particularly effective for balanced (i.e., many-to-many) collectives. On the other hand, k-ring, a generalization of the ring algorithm, provided no practical benefit.

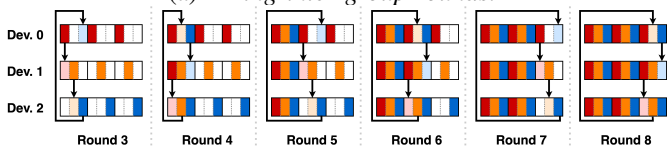
While showcasing promising results, this previous work concentrated on generic CPU-CPU shared memory communication. This paper reconsiders and enhances parameterized algorithms in the new context of GPU-GPU communication.

E. LogGP Performance Model

In this paper, we develop a novel simulation technique for collective algorithms based on the LogGP model. LogGP, first proposed by Alexandrov et al. [5] as an extension to the LogP model [9], is an analytical model used for analyzing parallel computation and communication. The model is comprised of five separate parameters, each representing some aspect of the communication. L is the latency of the communication between endpoints. o, which can be separated into o_s and o_r, corresponds to the per-message processor send and receive overheads, respectively. g is the time between messages, which can be thought of as the bandwidth for small messages. G is



(a) *K-ring inter-group rounds.*



(b) *K-ring intra-group rounds, only devices 0-2 shown.*

Fig. 2: *Allgather* k-ring on 9 devices, *k*=3. K-Ring is comprised of (a) two inter-group rounds and (b) six intra-group rounds.

the time per-byte for large message transfers, which corresponds to the network bandwidth. Finally, P is the number of total processes in the network.

As an example, a single message of size m takes $s = o_s + G*(m-1)$ time to send and $r = L + o_r$ time to receive. The total time for one message is $T = o_s + G*(m-1) + L + o_r$. After the initial message, the time at which the next message can be started from the same node is determined by $T_{next} = o_s + G*(m-1) + \max(0, g - o_s)$, depending on whether the send overhead o_s of the next message can completely overlap the per-message network gap g. In our analytical models, we make the assumption that $o_s \leq g$. We found this assumption to be true for both Delta and Polaris.

Overall, the LogGP model is a simple yet precise framework for high-performance communication. Given its established use for modelling collective communication [5], [16], LogGP is the preferred foundation for our GPU-to-GPU collective communication simulator.

III. Parameterized Collectives

Parameterized collective algorithms can adapt to a heterogeneous network hierarchy while maintaining portability and scalability. In this paper, we explore two classes of parameterized algorithms, *k-ring* and *recursive multiplying*, for the *Allgather* and *Allreduce* collectives and derive simplistic analytical expressions for parameter values (k). We also develop a distance-halving variant of recursive multiplying, *permuted recursive multiplying*, and show how an in-place GPU permutation kernel outperforms other implementations.

A. K-Ring

K-Ring is a parameterized version of the standard ring algorithm, where each process has two adjacent neighbors. In a standard *ring* algorithm, for p processes, each process sends data to its next neighbor and receives data from its previous neighbor, forwarding the previous round's data until the collective completes in $p-1$ rounds. The ring algorithm generally performs well for large message sizes because it avoids network congestion by communicating with the same adjacent partners each round. However, each round depends on the previous round, meaning that in hierarchical networks, faster connections must wait for slower ones, effectively throttling the entire algorithm to the speed of the slowest link.

To overcome the limitations of the ring algorithm on modern networks, k-ring introduces the concept of groups, where the value k is the group size. For p processes, each process first completes $\frac{p}{k} - 1$ rounds with a ring comprised of one process from each group (the inter-group ring), then completes $p - \frac{p}{k}$ rounds within its group (the intra-group ring). Note that $p - \frac{p}{k} + \frac{p}{k} - 1$ simplifies to $p-1$, the same as the ring algorithm. By splitting the collective into two different communication domains, k-ring can decouple different link types. Fig. 2(a) illustrates the inter-group rounds and Fig. 2(b) illustrates the intra-group rounds for k-ring, *Allgather* with *p*=9 processes and *k*=3. In total, there are eight rounds of communication.

Using the LogGP model, we derive analytic performance estimates for k-ring. Subscripted I denotes inter-group communication, subscripted i denotes intra-group communication, and AG is shorthand for *Allgather*.

$$T_I = o_{s,I} + (m-1)G_I + \max(L_I + o_{r,I},\ g_I - o_{s,I})$$

$$T_i = o_{s,i} + (m-1)G_i + \max(L_i + o_{r,i},\ g_i - o_{s,i})$$

$$T_{final} = o_{s,i} + (m-1)G_i + L_i + o_{r,i}$$

$$T_{AG} = \left(\frac{p}{k} - 1\right)T_I + \left(\frac{p}{k}(k-1) - 1\right)T_i + T_{final}$$

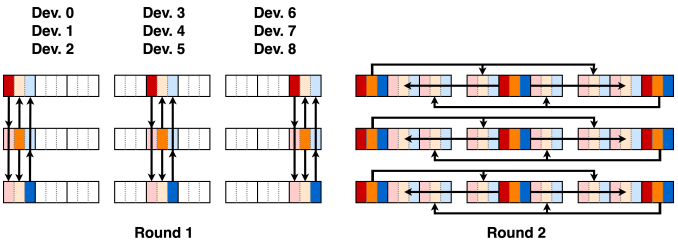
The analytical model for *Allreduce* is similar, with the addition of a per-byte computation cost γ. We assume a naive implementation that serializes computation with communication. Note that AR is shorthand for *Allreduce*.

$$T_{AR} = T_{AG} + (p-1)m\gamma$$

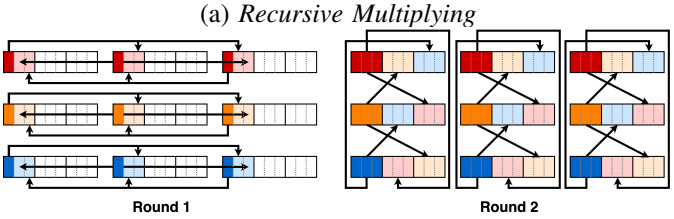
The cost models presented above are implementation-dependent. For example, in *Allreduce*, communication and computation stages can be overlapped. If $\gamma \leq G$, a pipelined *Allreduce* implementation would overlap all but the last byte of computation. Thus, we might expect that the $(p-1)m\gamma$ term in the expression above could be lowered to $(p-1)\gamma$. Additionally, in cases where inter and intra-group rings utilize different interconnects (e.g. k is selected such that inter-group communication traverses a NIC while intra-group communication uses NVLink), it is possible to overlap intra-group rounds with inter-group rounds, thereby giving:

$$T_{AG,overlap} = \left(\frac{p}{k} - 1\right)\max\left(T_I,\ (k-1)T_i\right) + (k-2)T_i + T_{final}$$

We leave implementation-specific optimizations as future work. The focus of this paper is to showcase the inherent benefit of these parameterized algorithms, external to implementation optimization. In the case of k-ring, the choice of a k parameter both decouples interconnect types and lowers communication volume over inter-group links, which is a clear algorithmic improvement for nonuniform clusters.



(a) *Recursive Multiplying*



(b) *Permuted Recursive Multiplying*

Fig. 3: *Allgather* recursive multiplying (a) and permuted recursive multiplying (b) algorithms on 9 devices, *k*=3. Permuted recursive multiplying results in a final buffer that is permuted.

B. Recursive Multiplying

Recursive multiplying is a parameterized version of the standard *recursive doubling* algorithm first presented in [27]. In recursive doubling, for every round r, a pair of nodes of distance 2^{r-1} apart exchange messages. Recursive doubling is best for small to medium messages since it only has $log_2 p$ rounds of communication.

Recursive multiplying generalizes recursive doubling by introducing a parameter k, which specifies the number of partners a process communicates with in each round. In round r, each process exchanges k^{r-1} messages with $k-1$ partners spaced k^{r-1} apart. As a result, recursive multiplying completes in $\log_k p$ rounds. Fig. 3(a) illustrates an *Allgather* with *p*=9 processes and *k*=3. In total, there are two rounds of communication.

For $k > 2$, and assuming the hardware supports concurrent send/receive operations, recursive multiplying reduces the number of rounds relative to recursive doubling. This reduction lowers overhead and latency, which are dominant costs for small to medium message sizes. Given these properties, the characteristics of GPU interconnects (§II-A) make them a natural fit for recursive multiplying.

Using the LogGP model, we present a simple analytical expression for *recursive multiplying Allgather* in a uniform multi-radix topology. The cost of a single round r for *recursive multiplying* is:

$$T_i(r) = o_s + (m * k^{r-1} - 1)G + max(L + o_r, g - o_s)$$

$$T_{final} = o_s + (m * k^{log_k p - 1} - 1)G + L + o_r$$

The total cost of the *Allgather* is therefore as follows:

$$T_{AG} = T_{final} + \sum_{r=1}^{log_k p - 1} T_i(r)$$

For *Allreduce*, every round has the same amount of data sent and received. The single-round cost for *Allreduce* is adjusted to be the following:

$$T_i = o_s + (m-1)G + max(L + o_r, g - o_s)$$

$$T_{final} = o_s + (m-1)G + L + o_r$$

The final cost of *Allreduce* is thus:

$$T_{AR} = (log_k p - 1)\, T_i + T_{final} + log_k p * m\gamma$$

C. Permuted Recursive Multiplying

In recursive multiplying *Allgather*, later rounds involve larger messages sent over longer communication distances, which reduces efficiency. To better exploit modern network topologies, larger messages should be exchanged with nearby processes. The distance *halving* recursive doubling algorithm addresses this by halving the communication distance each round [22]. However, since processes do not begin with adjacent partners, each message in subsequent rounds contains noncontiguous data.

As shown in [22], this limitation can be mitigated by an initial exchange that "permutes" the data so the first exchange involves adjacent elements. Their approach, however, requires an additional non-minimal communication round to correct the permuted buffers, adding overhead. We instead develop a parameterized version of the distance halving, recursive doubling algorithm, called the *permuted recursive multiplying* algorithm, that leverages an in-place permutation kernel on the GPU to preserve minimal communication and achieve better performance.

Fig. 3(b) illustrates the data movement for permuted recursive multiplying. Each process first copies its datum to the beginning of the receive buffer. Nodes in round r then communicate with partners spaced p/k^r apart, storing the incoming data contiguously, not at its original index. For example, in Fig. 3(b), all processes initially receive data into indices 1 and 2, even though this collectively represents the full dataset.

At the end of communication, each buffer contains the correct data in the wrong order. To restore order efficiently, we exploit GPU architectures' high-throughput on-device data movement with a GPU permutation kernel. This kernel works by first detecting permutation cycles in the buffer. A permutation cycle is a sequence of indices where the next adjacent index denotes the permuted location of the buffer element located at the current index. Permutation cycles are independent, meaning that each index will only appear in one cycle. Therefore, GPU threads can independently traverse and swap elements within each cycle in parallel, yielding an ordered buffer.

Our kernel performs the permutation in place, avoiding extra device memory, and eliminates race conditions by ensuring each element is accessed by only one thread. The approach outperforms both minimal and non-minimal communication schemes for small to medium message sizes, and because the additional data movement is purely local, it scales efficiently to large process counts.

IV. Fast Parameter Selection Through Simulation

Selecting the right parameter values is essential for optimizing the performance of parameterized algorithms. In this paper, we introduce a simulation method for choosing the algorithmic parameters. Our simulation predicts the relative performance of different parameters to find the best selection.

To simulate collective communication, we decompose each collective into individual send/receive messages and use the LogGP model to characterize them. Our schedule generator creates a message schedule for a given algorithm and scale.

To simulate our schedules, we model the machine as an abstract topology that groups nodes with similar link characteristics. LogGP parameters for each group are derived by profiling the real system. We then perform a discrete-event simulation of the collective schedule using the LogGP model to evaluate relative performance across different algorithms and parameter values and identify the optimal configuration. To more accurately capture performance for large message sizes, we extend the LogGP model with contention models.

A. Collective Schedule

A collective schedule expresses a collective algorithm with a set of device-to-device messages and dependencies between messages that receive and forward the same data. Each message in the collective schedule contains information about its source and destination devices, as well as the amount of data being transmitted. Together, the individual messages in the collective schedule are interpreted by the discrete event simulator in dependency order.

To enable simulation of larger-scale collectives, we develop collective schedule generators for each of our algorithms. User-specified inputs to a schedule generator include the number of devices, total message size, and parameter value. The generated message dependency graph is then encoded as an XML file, where it is later consumed by the discrete event simulator. Simulations of the same algorithm across different systems and topologies can reuse the generated schedule.

B. GPU Topology and LogGP Parameters

We model the point-to-point messages in collective schedules using the LogGP model. To model a modern HPC network, we assume symmetry at each level, meaning all NVLink connections or all links within a dragonfly group share the same LogGP parameters. We obtain these parameters using the low-overhead method described in [14]. Current systems, typically with three to four network levels, require only a few hours on a pair of nodes to obtain parameters per level. Since this measurement is performed once per target machine, the amortized cost of collecting parameters is negligible.

C. Discrete-Event Simulator

We estimate our algorithms' performance using discrete event simulation. The simulator uses the collective schedule to determine when new message tasks can start, ensuring all dependencies are met and there is no device-level contention (§IV-D). Each initiated message schedules future LogGP events based on the chosen LogGP parameter profile and node-level contention(§IV-D). The LogGP parameter choice depends on the GPU topology and message size. The simulation results in the cost of executing the collective schedule on the given topology with the associated LogGP parameters. Schedules for collectives with different algorithmic parameters can be input into the simulator, and the resulting costs are compared to identify the parameter with the lowest cost.

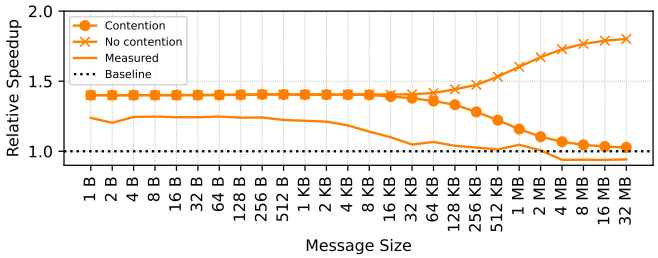


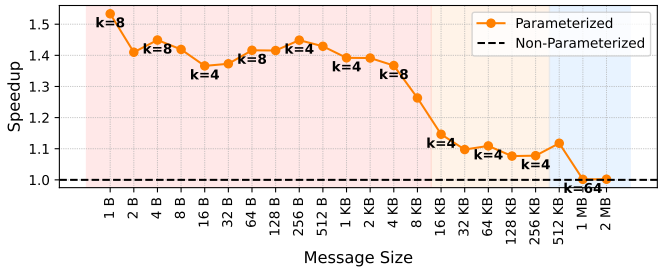
Fig. 4: Relative speedup predicted by the simulator with (o) and without (x) contention. With contention, the simulated speedup matches the behavior of the measured speedup (solid), converging to the baseline for large message sizes.

Our simulator implementation is relatively fast; it takes only 3 minutes to evaluate the cost of a 64-device *Allgather*, *ring* schedule for a 2 MB message size. We believe a multi-threaded simulator could significantly reduce this time. Simulation is preferred over exhaustive experimental tuning because it can be run offline on a single node, making it much faster and more resource-efficient, especially for larger allocations.

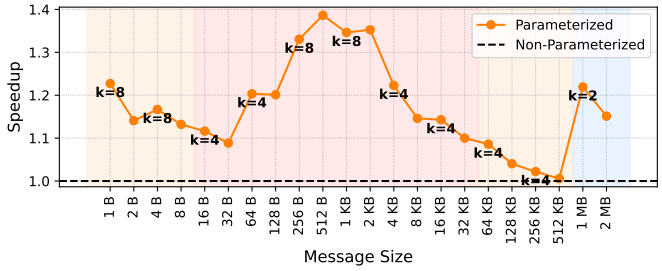
D. Device and Node Contention

In simulating collective communication on GPUs, we discovered the need to consider device and node-level contention. Device-level contention happens when multiple sends or receives on a single GPU device compete for limited hardware resources. While GPUs can handle multiple send/receive primitives concurrently, such as with a NCCL group, attempting too many at once increases the message gap g when the device hits its limit and must process tasks sequentially. We call this threshold the *device-radix*. Node-level contention, on the other hand, occurs when multiple devices on a single node compete for limited interconnect resources, such as a network interface controller (NIC). Node-level contention also serializes messages going in and out of the node once a certain threshold is reached. We call this threshold the *node-radix*. When simulating Delta, we set the device-radix to 8 and node-radix to 1, based on empirical observations.

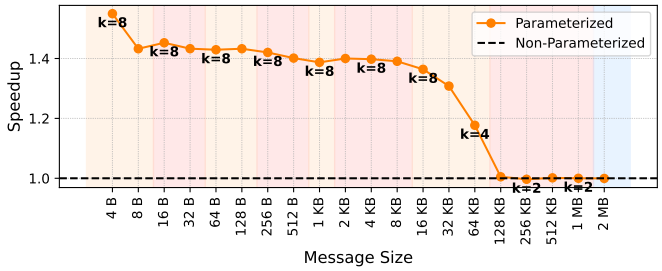
We model device-level contention by delaying new messages once the maximum number of outstanding sends is reached. For receives, we maintain a buffer of all message bytes to be received when the maximum number of outstanding receives is reached. Modeling node-level contention is also crucial for simulation accuracy. Node contention in the simulator is modeled using a node-level send/receive buffer, where messages sent to or received from a device outside the node must pass through this buffer. The node buffer arbitrates



(a) *Allgather (Delta)*



(b) *Allgather (Polaris)*



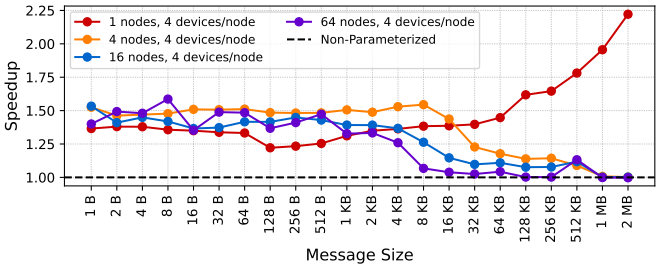
(c) *Allreduce (Delta)*

Fig. 5: Message Size vs. Speedup (Higher is Better), 16 Nodes on Delta/Polaris. Parameterized algorithms provide significant speedups across nearly all messages on both test platforms.

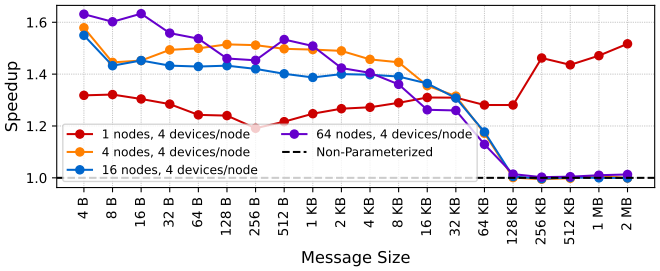
messages in and out of the node at a maximum rate that is a multiple of G, based on the node-radix from the profile. In the case of k-ring, node-level contention increases with the parameter value k, becoming more significant at larger message sizes. Fig. 4 illustrates simulation results with and without node-level contention for k-ring (k=2) on a 2-node, 4 devices per node machine. At small to medium message sizes (<64KB), simulations with and without contention are relatively consistent. However, at sizes beyond 64KB, the models diverge, with the contention model converging with the baseline and matching our measured results. Enhancing our LogGP-based event simulator with contention allows it to accurately predict the relative performance of different algorithms.

V. Evaluation

We now describe our algorithm implementations, evaluation mechanisms, and experimental results. We analyze the performance of our algorithms at different scales and machines, compare the effectiveness of a GPU-specific optimization for collective algorithms, as well as show the effectiveness of simulation-based parameter selection.



(a) *Allgather*



(b) *Allreduce*

Fig. 6: Message Size vs. Speedup, 1-64 nodes on Delta. Parameterized algorithms achieve speedups at various scales.

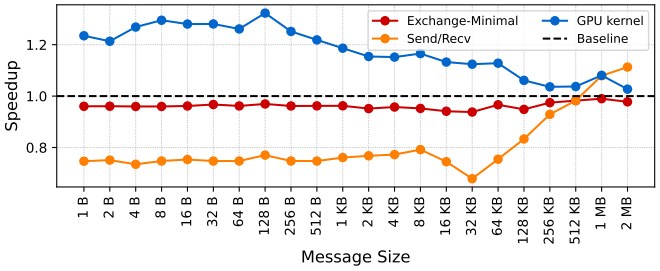


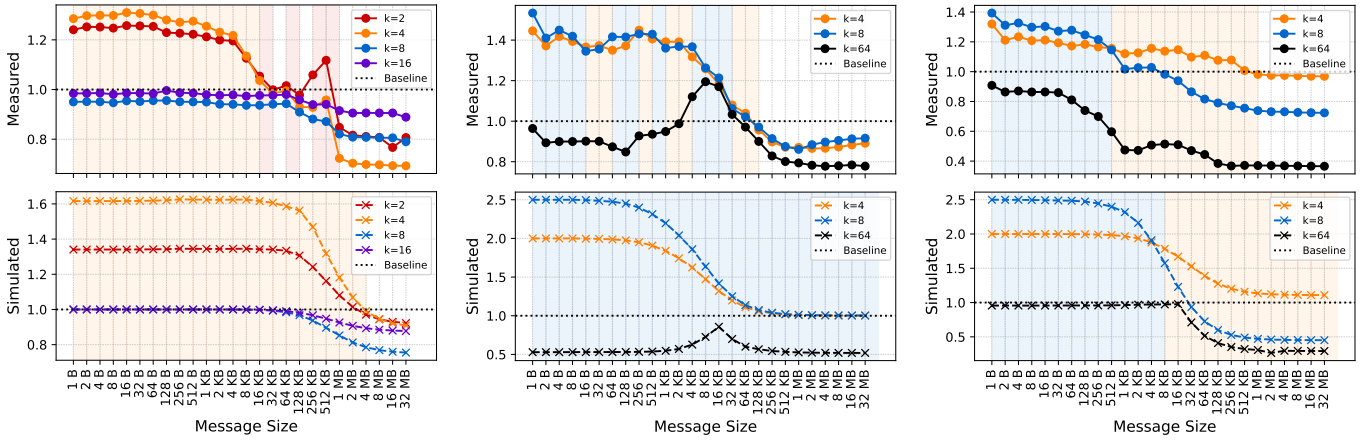
Fig. 7: Message Size vs. Speedup, 4 Nodes on Delta/Polaris. Our GPU permutation kernel (blue) outperforms other minimal and non-minimal communication techniques for distance halving, recursive doubling *Allgather*.

A. Implementation

We implement both parameterized and non-parameterized versions of *k-ring*, *recursive multiplying*, and *permuted recursive multiplying* as an LD_PRELOAD library for NCCL *Allgather* and *Allreduce* collective functions. The library is built on top of NCCL send/receive primitives. We use NCCL groups to overlap independent message primitives, such as in *recursive multiplying*. We implement the LogGP profiler using a customized version of the NCCL latency test from the OSU microbenchmark suite [1]. We implement the discrete event simulation framework and schedule generator in Python.

B. Evaluation Systems

We use NCSA Delta and Polaris at ALCF for our empirical evaluation. Both machines are equipped with a single socket AMD CPU with 4 NVIDIA A100 GPUs connected via NVLink rated at 600Gbps. Note that all experiments in this paper use 4 processes per node, one per GPU. Both machines also use an HPE/Cray Slingshot 11, 200Gbps interconnect



(a) *K-Ring* (b) *Recursive Mult.* (c) *Per. Recursive Mult.*

Fig. 8: Message Size vs. Measured or Simulated Speedup (Higher is Better), 16-node Allgather on Delta. The simulated speedups align with the measured relative parameter ordering and curve features for each algorithm.

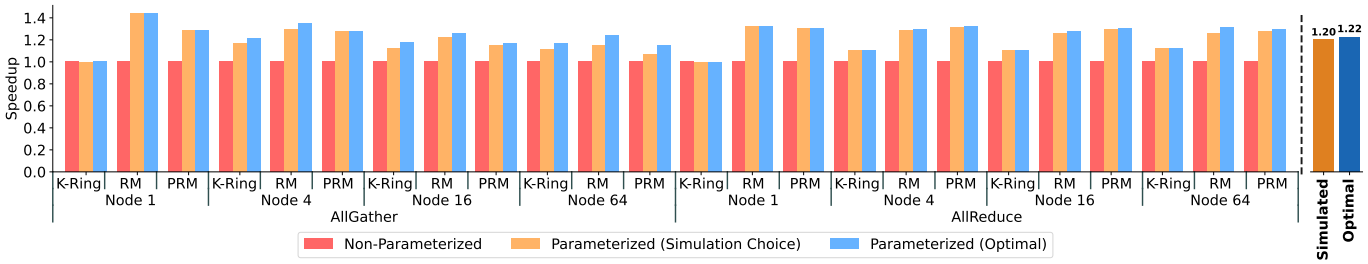


Fig. 9: Achievable performance with simulation-based parameter selection across different collectives, node counts, and algorithms on Delta. We are able to achieve, on average, 98% of the performance of the best parameter.

with dragonfly topology. Each Polaris node has two Slingshot network cards, while each Delta node only has one.

We do not compare directly against NCCL or other collective communication libraries. The goal of this paper is to specifically study the benefit of parameterized algorithms for GPU-GPU collective communication.

C. Speedups and Scaling

Fig. 5 illustrates that our parameterized algorithms achieve significant speedups on Delta for both *Allgather* and *Allreduce* operations (and *Allgather* on Polaris) across 16 nodes. These graphs plot the message size versus the maximum possible speedup from parameterization when compared to the non-parameterized implementation, across all algorithm classes. For these charts, we tested every non-trivial factor of 64, k-value ([2,4,8,16,32,64] for k-ring and [2,4,8] for permuted/non-permuted recursive multiplying) and plotted the highest-performing result from the fastest algorithm for each message size. The background color indicates the best-performing algorithm: red for *recursive multiplying*, light orange for *permuted recursive multiplying*, and blue for *k-ring*.

For *Allgather* on both machines, we achieve speedups of over 1.3x, especially with small message sizes, due to recursive multiplying which overlaps message latencies. On Delta, non-permuted *recursive multiplying* is slightly faster. On Polaris, recursive multiplying performed better than the permuted version for messages between 16B and 32KB. For small to medium-sized messages (32B-32KB), recursive multiplying generally exhibits better performance by avoiding permutation and associated kernel launch overheads. For medium-sized messages (32KB-1MB), permuted recursive multiplying provides the best performance by leveraging the improved bandwidth of intranode GPU-GPU links, offsetting permutation time. Both Delta and Polaris successfully leveraged multi-ported network resources by achieving the best performance with k=4 or k=8 in recursive multiplying algorithms for small to medium-sized messages. In particular, both parameter values achieve similar performance in recursive multiplying on Delta, while k=4 is better than k=8 in permuted recursive multiplying for medium-sized messages. For larger messages, k-ring offers modest speedups in some cases. Since results on Polaris were similar to those on Delta, we focus on the Delta results. Performance trends are alike unless indicated.

For *Allreduce*, we see a similar pattern. Recursive multiplying and permuted recursive multiplying have identical network loads, leading to minor performance differences (under 5%) for small to medium messages, likely due to run-to-run variance. For large messages, ring and k-ring are again the most effective algorithms, with k-ring providing occasional speedups for messages beyond 2MB.

Fig. 6 demonstrates that speedups from parameterization occur across scales from 1 to 64 nodes. The graphs plot

message size against speedup, with the 16-node line matching Fig. 5. We include lines for other scales to compare and contrast. In general, we see speedups for small messages increase with node count, while large messages benefit more at smaller scales. All scales show speedups of over 1.3x across message sizes.

We observe significant speedups with parameterized algorithms for both collectives across a variety of scales on a modern GPU-based supercomputer, validating the key contributions of this paper.

D. GPU-Specific Optimization

Fig. 7 shows that our GPU-specific in-place permutation kernel applied to the distance halving, recursive doubling algorithm outperforms other generic strategies. We plot speedups between different minimal-communication implementations of the distance halving, recursive doubling algorithm, with the non-minimal communication method presented in [22] as our baseline. We compare our GPU kernel approach with two other methods. The first method (Exchange-Minimal) is similar to the non-minimal exchange method, but keeps its original data after the initial message exchange. All sends and receives for the original data are then omitted to avoid redundant communication. The second method averts permutation by sending non-contiguous data as separate, smaller messages within a NCCL group (Send/Recv). Each message is then received separately into the correct buffer location.

Our GPU-optimized permutation kernel achieves over 1.3x speedup over the baseline for small message sizes. For large message sizes ($\geq$2MB), the Send/Recv method is superior due to its constant time latency, while the permutation kernel's cost scales linearly. However, for large messages, the ring and k-ring algorithms are the most efficient, making our GPU kernel approach optimal across all relevant scenarios.

E. Parameter Selection with Simulation

Fig. 8 shows how our simulations accurately capture trends in parameterized algorithm performance and identify optimal parameters. We plot the speedup of various k-values against the non-parameterized baseline for each *Allgather* algorithm on Delta (16-nodes). Shaded regions indicate the best k while regions that are not shaded indicate better baseline performance.

While simulations often show higher speedups than measured results, they align well in the relative ordering of parameters. For k-ring, the simulator correctly identifies 4 as the optimal parameter for small and medium message sizes. For permuted recursive multiplying, the simulator correctly chooses 8 for smaller messages and 4 for medium messages. When there is disagreement, the simulation typically selects parameters with near-optimal performance, such as k=4 versus k=8 for recursive multiplying.

The simulator replicates features observed in measured data, e.g., accurately reproducing the performance curve of large k-values (e.g., k=64) and predicting their performance drop-offs across all algorithms. These results confirm the simulator's validity and its potential for effective parameter selection.

Fig. 9 illustrates the speedups achievable with our methodology across various algorithms, scales, and collectives on Delta. The baselines (1.0 speedup) representing non-parameterized algorithms are given in red. Speedups achieved with parameters selected by the simulator (simulation-selected performance) are shown in orange. The maximum speedups, assuming the optimal set of parameters were selected, are depicted in blue. The speedup denoted by a single bar represents the geometric mean speedup across message sizes up to 2MB.

Altogether, the simulation-selected performance closely matches the measured optimal performance, achieving a geometric mean speedup of 20% and capturing 98% of the optimal performance.

F. Evaluation Summary

Overall, our experiments demonstrate that algorithmic parameterization significantly enhances collective performance on GPU-based supercomputers across different scales and machines. Furthermore, our exploration of an in-place permutation kernel exemplified the possibility of using GPU-specific characteristics to optimize collective algorithms. Finally, with a simulation-based parameter selection approach, we were able to choose performant parameter values, effectively realizing the benefits of collective algorithm parameterization.

VI. Conclusion

Efficient collective communication is vital for modern supercomputers with hierarchical networks and GPUs. We showed that parameterized algorithms for *Allgather* and *Allreduce* enhance adaptability and performance on supercomputers like NCSA Delta and Argonne Polaris. Using a LogGP-based simulation, we were able to achieve on average 20% speedup over non-parameterized algorithms while capturing 98% of the potential speedup automatically.

Acknowledgments

This effort is partially supported by the U.S. National Science Foundation (NSF) under awards CNS-2211315, CCF-2119069, and CNS-2211508 and is based upon work supported by the Graduate Research Fellowship Program under Grant No DGE-2234667. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This material is also supported by funding from the Laboratory Directed Research and from Argonne National Laboratory, provided by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.

This research used resources of the National Artificial Intelligence Research Resource (NAIRR) Pilot and the Delta advanced computing and data resource, which is supported by the National Science Foundation (award NSF-OAC 2005572). 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.

We would like to thank Griffin Dube for his early contributions and Suisui Xu for the insightful discussions.

REFERENCES

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

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

[3] "Aurora," 2025. [Online]. Available: https://docs.alcf.anl.gov/aurora/

[4] "El capitan," 2025. [Online]. Available: https://hpc.llnl.gov/hardware/compute-platforms/el-capitan

[5] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation," in *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, 1995, pp. 95–105.

[6] AMD, "RCCL," https://github.com/ROCm/rccl, 2025.

[7] 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*, 2021, pp. 62–75.

[8] E. Chan, M. Heimlich, A. Purkayastha, and R. Van De Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken, "LogP: Towards a realistic model of parallel computation," in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1993, pp. 1–12.

[10] W. J. Dally, S. W. Keckler, and D. B. Kirk, "Evolution of the graphics processing unit (gpu)," *IEEE Micro*, vol. 41, no. 6, pp. 42–51, 2021.

[11] J. Duan, S. Zhang, Z. Wang, L. Jiang, W. Qu, Q. Hu, G. Wang, Q. Weng, H. Yan, X. Zhang *et al.*, "Efficient training of large language models on distributed infrastructures: a survey," *arXiv preprint arXiv:2407.20018*, 2024.

[12] K. Fan, S. Petruzza, T. Gilray, and S. Kumar, "Configurable algorithms for all-to-all collectives," in *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*. Prometeus GmbH, 2024, pp. 1–12.

[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] T. Hoefler, A. Lichei, and W. Rehm, "Low-overhead LogGP parameter assessment for modern interconnection networks," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–8.

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

[16] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim: simulating large-scale applications in the LogGOPS model," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 597–604.

[17] Intel, "oneCCL," https://github.com/uxlfoundation/oneCCL, 2025.

[18] H. Khetawat, N. Jain, A. Bhatele, and F. Mueller, "Predicting GPUDirect benefits for hpc workloads," in *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2024, pp. 88–97.

[19] 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.

[20] NVIDIA, "NCCL," https://github.com/NVIDIA/nccl, 2025.

[21] 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.

[22] 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.

[23] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[24] A. Shah, V. Chidambaram, M. Cowan, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, O. Saarikivi, and R. Singh, "{TACCL}: Guiding collective algorithm synthesis using communication sketches," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 593–612.

[25] 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.

[26] 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.

[27] M. Wilkins, H. Wang, P. Liu, B. Pham, Y. Guo, R. Thakur, P. Dinda, and N. Hardavellas, "Generalized collective algorithms for the exascale era," in *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2023, pp. 60–71.

[28] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer *et al.*, "Pytorch fsdp: experiences on scaling fully sharded data parallel," *arXiv preprint arXiv:2304.11277*, 2023.