In-network Allreduce with Multiple Spanning Trees on PolarFly

Kartik Lakhotia
Intel Labs
Santa Clara, USA
kartik.lakhotia@intel.com

Kelly Isham
Colgate University
Hamilton, USA
kisham@colgate.edu

Laura Monroe
Los Alamos National Laboratory
Los Alamos, USA
lmonroe@lanl.gov

Maciej Besta
ETH Zürich
Zurich, Switzerland
maciej.besta@inf.ethz.ch

Torsten Hoefler
ETH Zürich
Zurich, Switzerland
torsten.hoefler@inf.ethz.ch

Fabrizio Petrini
Intel Labs
Santa Clara, USA
fabrizio.petrini@intel.com

ABSTRACT

Allreduce is a fundamental collective used in parallel computing and distributed training of machine learning models, and can become a performance bottleneck on large systems. In-network computing improves Allreduce performance by reducing packets on the fly using network routers. However, the throughput of current in-network solutions is limited to a single link bandwidth.

We develop, compare and contrast two different sets of Allreduce spanning trees embedded into PolarFly, a high-performance diameter-2 network topology. Both of our solutions offer theoretically guaranteed near-optimal performance, boosting Allreduce bandwidth by a factor equal to *half the network radix* of nodes. While our first set offers low-latency with trees of *depth-3*, the second set offers congestion-free implementation which reduces complexity and resource requirements of in-network computing units.

In doing so, we also distinguish PolarFly as a highly suitable network for distributed deep learning and other applications that employ throughput-bound large Allreductions.

CCS CONCEPTS

• **Theory of computation** → *Discrete optimization*; • **Computing methodologies** → *Distributed computing methodologies*; • **Networks** → **In-network processing**.

This research is, in part, based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0081. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, ARO, or the U.S. Government. This work was also supported by the U.S. Department of Energy through the Los Alamos National Laboratory, operated by Triad National Security, LLC, for the National Nuclear Security Administration of U.S. Department of Energy (Contract No. 89233218CNA000001), and by the U.S. DOE LDRD program at Los Alamos National Laboratory under project number 20230692ER. The U.S. Government retains an irrevocable, nonexclusive, royalty-free license to publish, translate, reproduce, use, or dispose of the published form of the work and to authorize others to do the same. This paper has been assigned the LANL identification number LA-UR-23-20414.

SPAA '23, June 17–19, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9545-8/23/06.
https://doi.org/10.1145/3558481.3591073

KEYWORDS

In-network Collectives, Allreduce, PolarFly, Erdos-Renyi Graphs, Spanning Trees, Hamiltonian Paths

ACM Reference Format:
Kartik Lakhotia, Kelly Isham, Laura Monroe, Maciej Besta, Torsten Hoefler, and Fabrizio Petrini. 2023. In-network Allreduce with Multiple Spanning Trees on PolarFly . In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3558481.3591073

1 INTRODUCTION

Allreduce is a fundamental collective, which aggregates P vectors maintained by P processes using an associative operation, such as summation, and stores the result at each process. It is widely used in High-Performance Computing (HPC) and Machine Learning (ML) applications [50, 55]. In HPC, vectors are usually small, so the operation is latency-bound. In ML, however, the vectors are usually large, making the operation bandwidth-bound and data-parallel, as seen, for example, in state-of-the-art language models like GPT-3 [8]. In this paper, we propose a method to greatly increase Allreduce bandwidth for ML and other applications that reduce large vectors.

1.1 In-network Computing

In-network computing is widely used to improve performance of collectives [11, 21, 32, 36–38, 40, 54]. Allreduce can be offloaded by embedding spanning trees on the network and aggregating input packets in-flight using specialized network devices. This significantly reduces network traffic and transfer protocol overheads.

Recent systems such as Intel PIUMA [36, 37] and Mellanox SHARP [21], and switch architectures such as Flare [11], SwitchML [54] and others [32, 40], utilize streaming aggregation for high bandwidth Allreduce. However, for most of these approaches, the maximum Allreduce bandwidth is limited to a single link bandwidth. Mellanox SHARP supports concurrent operations on multiple (up to two) Allreduce trees [21]. However, SHARP allows logically defined trees and without any theoretical guarantees on congestion, they can suffer from performance issues on non fat-tree topologies. The solutions developed in this paper exhibit little or no congestion, and can significantly benefit systems that support multiple trees.

1.2 Optimizing Allreduce Bandwidth

In indirect switched networks such as fat-trees, Allreduce is easily implemented by mapping a reduction tree into multiple levels of the network, mirroring the switch arity. Typically, each compute node has a single or a limited number of connections to the switches, determined by the number of IO ports. This architectural design limits the maximum achievable bandwidth for Allreduce. On the other hand, direct networks such as multi-dimensional grids [33] or HyperX [1] typically have all network links of a node accessible to the compute. In such networks, we can use the entire pool of links on a node and *build multiple spanning trees to execute an Allreduce instance in a data-parallel fashion* over subsets of input vectors.

Direct networks can deliver significantly larger Allreduce bandwidth by using concurrent spanning trees. However, the trees must be carefully embedded, or else *congestion* (overlapping links between the trees) can create bottleneck edges with high traffic load, nullifying the performance benefits of data-parallelism. To address this, we propose novel methods of embedding multiple trees that minimize congestion and achieve *near-optimal* aggregate Allreduce bandwidth for a given network radix. Our schemes also offer a trade-off between latency and congestion, with spanning trees of depth just 3 in one solution, and edge-disjoint trees in another.

Prior works on multiported Allreduce on direct tori networks have exploited data parallelism by launching multiple instances of Reduce-scatter and All-gather [25, 30, 53]. They communicate, store and compute large blocks of data, and are used in host-based implementations. However, their memory requirements can be prohibitive for in-network computation due to the limited buffers on network devices. This paper explores Allreduce computation on concurrent trees, which can be pipelined with a small memory footprint equal to latency-bandwidth product of the links, and are thus suitable for in-network computation.

1.3 The Target Topology and its Advantages

For the target network topology, we use PolarFly [35], a recently proposed high-performance diameter-2 network based on Erdős-Rényi polarity graphs (ER_q) [9, 13].

Although recent, PolarFly has been shown to outperform previous networks such as Slim Fly, Dragonfly, HyperX, and Fat Trees, in terms of scaling efficiency, bisection width, and performance per cost. It also exhibits flexibility in construction with many feasible network radixes and modular layouts for practical large-scale deployments. These desirable qualities make PolarFly an enticing candidate for development of in-network collectives.

Our solutions for high-bandwidth Allreduce demonstrate further advantages of PolarFly, and highlight it as a good candidate network to be used in Machine Learning applications.

1.4 Our Mathematical Approach

We use the rich mathematical structure of PolarFly to embed high-performance Allreduce spanning trees with provable performance guarantees. We develop two different Allreduce solutions for PolarFly – one uses a set of trees with congestion-2 and depth-3 which reduces latency, and the other uses edge-disjoint trees (no congestion) which reduces resource requirement of in-network computing devices. Both of our solutions achieve provably optimal or near-optimal aggregate bandwidth, thus improving Allreduce bandwidth by a factor proportional to the network radix.

PolarFly is one of a recent wave of mathematically designed low-diameter networks, including Slim Fly [5], Bundlefly [39] and others. The success here of a mathematical approach to Allreduce on PolarFly suggests that the mathematical structure of other networks may similarly be used to generate optimal Allreduce solutions.

2 CONTRIBUTIONS

- We observe that the problem of maximizing Allreduce bandwidth on a direct network can be reduced to one of finding *an optimal set of spanning trees* in the network with highest aggregate bandwidth. Practical considerations impose additional constraints on depth and congestion of these trees.
- We present a formal performance model for a given set of spanning trees embedded in a network, which takes into account the impact of congestion on the achievable bandwidth.
- Under this performance model, we define optimality for a given network, and demonstrate that a multi-tree embedding can boost Allreduce bandwidth proportionally to the network radix, compared to a single tree embedding.
- We present here two novel solutions for near-optimal bandwidth Allreduce computation on the state-of-the-art low-diameter network PolarFly [35]. Our first solution derives a set of very low latency depth-3 trees using the Polarfly layout. The second derives an optimal number of disjoint spanning trees in ER_q in the form of Hamiltonian paths, whose existence and construction are first shown in this paper.
- Thus, our work also makes a case for suitability of PolarFly for applications sensitive to Allreduce bandwidth, such as distributed ML training.

3 PROBLEM STATEMENT

The primary focus of this work is to use multiple embedded trees to exploit data parallelism in Allreduce. As we discuss in Section 4.4, this translates to the mathematical problem of finding *a set of multiple spanning trees* in the network topology that maximizes the aggregate bandwidth under the performance model of Section 5.2.

Of course, one can always find large sets of spanning trees. However, a solution usable in practice minimizes edge overlap between the trees, as congested links create performance bottlenecks and require network devices to maintain a proportional number of states. This increases resource requirements and can even impact feasibility of implementation. For high performance, low-latency is also desirable, which encourages low-depth trees.

4 BACKGROUND

4.1 Interconnection Network

A network may be represented as an undirected graph $G = (V, E)$: V is the set of nodes, E is the set of links and $|V| = N$ is the number of nodes. There is a network device associated with each node, which coordinates routing and computation on data packets for in-network computing, which we refer to as a *router*. Each node is incident to at most d bidirectional links, where d is the network radix, and can simultaneously communicate on all of them.

4.2 Allreduce Collective

We study a global Allreduce that takes an associative binary operator $\bigoplus$ and an input x_i from each node $v_i \in V$, and distributes the reduction $y = \bigoplus_{j=0}^{N-1}\{x_j\}$ to all nodes. In the generalized vector

Allreduce, each $\boldsymbol{x_i}$ is a vector of m elements $\{x_{i,0}, ...x_{i,m-1}\}$, and the output $\boldsymbol{y} = \{y_0, ...y_{m-1}\}$ is an element-wise reduction of all input vectors, i.e., $y_k = \bigoplus_{j=0}^{N-1}\{x_{j,k}\}$. Therefore, Allreduce computation can be *parallelized over vector elements*.

Host-based Allreduce algorithms require compute nodes to coordinate computation and communication. Latency-optimal algorithms such as Recursive Doubling [18, 22] minimize the number of point-to-point communication rounds. Bandwidth-optimal algorithms such as Recursive Halving and Ring-Allreduce [46, 51] minimize communication volume per node and enable pipelining. However, these algorithms incur multiple rounds of communication and data transfers from process memory to network, and large traffic volume per compute node. Therefore, a host-based Allreduce may suffer from high latency and poor performance scalability.

4.3 In-network Computing and Allreduce

In-network computing offloads computation to specialized network devices, referred to as routers in this paper, that can reduce data packets in-flight. This enables Allreduce computation with a single data transfer from application memory to network, and simultaneously reduces network traffic.

Offloading collectives like Allreduce onto the network requires embedding a dataflow graph into the network. Logically, Allreduce can be computed as a reduction operation followed by an output broadcast, both of which can be executed on a spanning tree topology [19–21, 36, 37]. Inputs move up the tree towards the root, getting reduced at the nodes. The final reduction output computed at the root can be broadcasted to all nodes by traversing down the same tree. In a network embedding, the vertices and edges are mapped to routers and physical paths in the network, respectively. Allreduce can also be computed on a Hypercube but we found the tree-based approach to be the most effective, giving strong theoretical guarantees on PolarFly.

The performance of an embedded Allreduce tree is primarily characterized by two parameters:

(1) *Latency* – a function of the *depth* of the embedded tree, and
(2) *Bandwidth* – upper bounded by *link bandwidth B*, for one tree.

In a direct network, Allreduce bandwidth can be boosted by *embedding multiple spanning trees* in the network. Each node, with access to all of the associated router's links, can concurrently insert disjoint sub-vectors in different trees, exploiting data parallelism.

4.4 In-network Computing: Router Architecture

We define an abstract router architecture inspired by real world designs that support high-bandwidth collective embeddings, primarily Intel PIUMA [36, 37] and Mellanox SHARP [21].

Each router has a reduction engine that can aggregate data packets in-flight. We assume that data movement in network embeddings (including reduction engines) is pipelined for high throughput computation. Flow-control mechanisms such as credit tracking, are often used to operate the pipeline at link bandwidth [21, 37].

We assume that each router features a mechanism to configure connectivity between its I/O-ports and reduction engine, allowing us to map a dataflow graph onto the network [36, 37]. This mechanism provides explicit control over the embedded paths and deterministic routing. Thus, an Allreduce can be computed on a spanning tree over the physical network topology itself. The problem of embedding Allreduce trees now translates to *finding spanning trees in the physical network topology.*

Some routers allow embeddings to be logically defined by configuring the children and parent(s) of each router [20, 21]. The physical routing paths are decided by the routing algorithm at runtime and can be variable. Such mechanisms can incur path conflicts and are difficult to analytically reason about in terms of performance.

5 CONGESTION AND BANDWIDTH

5.1 Congestion Model

Congestion $C(e)$ on an edge $e = (u, v)$ is defined as the number of dataflow graph edges mapped to a physical link in the network. The bandwidth on a congested link is shared between the logical edges and can bottleneck the performance of the collective embedding.

When the spanning trees are defined over the physical network topology itself, there is no congestion within a tree [36, 37]. However, congestion can happen when multiple spanning trees with *overlapping edges* are embedded. In this scenario, congestion on a link is equal to the number of trees containing the link.

We assume that the arithmetic unit on a router can compute multiple reductions at link rate. Thus, overlapping reduction vertices in trees do not affect Allreduce bandwidth. This can be realized by adding more compute resources in routers, whereas link bandwidth sharing under congestion is a fundamental limitation.

Operating under congestion requires maintaining unique states for each overlapping tree on a link. One way to do so is to tag the packets with a tree identifier and store them in a common buffer [20, 21]. This requires a sophisticated engine that can track and retrieve packets of all overlapping trees that use the same router port. Another option is to use a number of Virtual Channels (VCs) equivalent to *worst-case link congestion* [36, 37]. VCs have disjoint resources (buffers, configuration), which enables multiple logical datapaths on each link to identify the state.

However, maintaining multiple tree IDs or VCs increases logic and buffering requirements. This can increase router area and power, especially for large-scale networks with high radix [34, 52]. These practical constraints make it desirable to minimize congestion in the design of the set of spanning trees.

5.2 Performance Under Congestion

In a multi-tree embedding with congestion, the bandwidth for each tree is computed using Algorithm 1. The bottleneck edge with the lowest ratio of available bandwidth to congestion constrains the throughput of all trees containing this edge. Its bandwidth is equally divided amongst these trees that consume equivalent bandwidth on their other links as well. The algorithm iterates and distributes the remaining bandwidth on the links among rest of the trees. We note that for all trees, the bandwidth computed by Algorithm 1 is independent of the order in which edges are selected when multiple choices for the bottleneck edge are available.

Theorem 5.1. *Given a network $G(V, E)$ with link bandwidth B and a set of embedded Allreduce trees $F = \{T_0, T_1, \ldots, T_r\}$ executing concurrently, if Algorithm 1 computes bandwidth B_i for tree T_i, then the maximum achievable Allreduce bandwidth is $\sum_{i=0}^{r} B_i$.*

Algorithm 1 Performance under Congestion

```
Input: Network Topology $G(V, E)$, link bandwidth $B$
Input: set of Allreduce Trees $F$
Output: Bandwidth $B_i$ for each Tree $T_i \in F$
1: for each link $e \in E$                                ▷ Initialization
2:     Available link bandwidth $L(e) = B$
3:     Congestion $C(e)$ = # trees in $F$ containing $e$
4: while $F$ not empty do
5:     $e_{\min} \leftarrow \arg\min_e \frac{L(e)}{C(e)}$
6:     for each Tree $T_i \in F$ containing $e_{\min}$
7:         $B_i = \frac{L(e_{\min})}{C(e_{\min})}$                ▷ Assign Tree Bandwidth
8:         for each edge $e$ in $T_i$
9:             $L(e) \leftarrow L(e) - B_i$          ▷ remaining link bandwidth
10:            $C(e) \leftarrow C(e) - 1$            ▷ update congestion on link
11:        Remove $T_i$ from $F$
12:    Remove $e_{\min}$ from $E$
```

Proof. Assume that the Allreduce collective is computed on a vector of size m, and each tree T_i computes Allreduce on a subvector of size m_i. We compute the optimal distribution of m_i to maximize aggregate Allreduce bandwidth. Assuming constant latency L for each tree, the execution time of T_i is

$$t_i = L + \frac{m_i}{B_i}.$$

Since all trees are executing concurrently, overall Allreduce time is

$$t_{ar} = \max_{0 \le i \le r} t_i.$$

Given $\sum_{i=0}^{r} m_i = m$, the optimal distribution of sub-vector sizes that minimizes Allreduce time achieves equal execution times for individual trees. Mathematically speaking,

$$t_0 = t_1 = t_2 = \cdots = t_r \tag{1}$$

$$\implies m_i = m \cdot \frac{B_i}{\sum_{i=0}^{r} B_i} \text{ for all } i \in [0, r] \tag{2}$$

For this distribution of sub-vectors, the allreduce time is given by

$$t_{ar} = t_i = \frac{m_i}{B_i} = \frac{m}{\sum_{i=0}^{r} B_i} \tag{3}$$

From Equation (3), Allreduce bandwidth is $\sum_{i=0}^{r} B_i$. □

In Corollary 7.1, we apply Theorem 5.1 to our exemplar network, obtaining an optimal bandwidth proportional to the network radix.

6 POLARFLY

In this paper, we choose as a representative network the high-performance diameter-2 PolarFly [35], a network topology based on the Erdős-Rényi polarity graphs ER_q. (Note that this is not the same as the commonly known Erdős-Rényi random graphs [6, 17]).

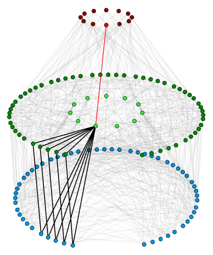
ER_q graphs exist for every radix $q + 1$, where a is a positive integer, p is prime, and $q = p^a$ is a prime power [9, 13, 35]. These are derived from Galois finite fields, which have order q.

Later, we will create and contrast two sets of high-bandwidth Allreduce spanning trees that offer a trade-off between latency and congestion. To do so, we look at two very different ways to construct the Erdős-Rényi polarity graphs ER_q that underlie PolarFly. We then show how the layout of PolarFly derives from these.

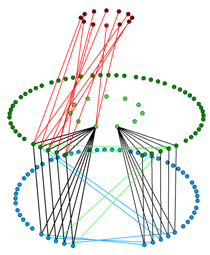
- The projective geometry construction of ER_q [9, 13] is the basis of our first construction of low-depth spanning trees, giving a *low-latency* high-bandwidth Allreduce.
- The Singer difference set construction of ER_q [4, 7, 56] is the basis of our second construction of disjoint spanning trees, giving a *zero-congestion* high-bandwidth Allreduce.

The graph constructions are isomorphic: they provide conceptually different methods of constructing the same ER_q graphs [14] (and thus the PolarFly topology) and these conceptually different methods are used in the design of the two types of spanning trees.

6.1 Projective Geometry Construction of Erdős-Rényi graphs



(a) A non-quadric cluster



(b) Edges between clusters

Figure 1: The PolarFly layout for $q = 11$. The vertex coloring distinguishes quadrics $W(11)$ (red), $V_1(11)$ vertices (green), and $V_2(11)$ vertices (blue). The starter quadric from Algorithm 2 is in bright red and the centers of each cluster are in bright green. The image of the layout on the left depicts a single non-quadric cluster. The one on the right shows the connections between two non-quadric clusters. The number of edges between clusters and within a cluster match up with Properties 1-3.

Erdős-Rényi (ER) graphs express dot-product orthogonality between vectors, using arithmetic defined over finite fields $\mathbb{F}_q$.

Since multiples of vectors retain their original orthogonality relationship, ER_q vertices are represented by left-normalized (the leftmost non-zero coordinate is 1) 3-dimensional vectors: $[x, y, z] \in \mathbb{F}_q^3$. The existence of multiplicative inverses in $\mathbb{F}_q$ ensures that every non-zero vector in $\mathbb{F}_q^3$ has a left-normalized representative, obtained by multiplying the vector by the inverse of the leading non-0 coefficient. Thus, the vertex set of ER_q is

$$\{[1, y, z] : y, z \in \mathbb{F}_q\} \cup \{[0, 1, z] : z \in \mathbb{F}_q\} \cup \{[0, 0, 1]\}.$$

From this construction, we see that the order of ER_q is $N = q^2 + q + 1$. An edge (u, v) exists in ER_q if and only if vectors $[u_1, u_2, u_3]$, $[v_1, v_2, v_3] \in \mathbb{F}_q^3$ are orthogonal, i.e., their dot-product $u \cdot v = u_1v_1 + u_2v_2 + u_3v_3 = 0$ in $\mathbb{F}_q$. When q is prime, this simply means checking whether $u \cdot v \equiv 0 \pmod{q}$, since the multiplication is modular when q is prime, but more complicated when q is not prime, as in [41, 42]. A detailed illustration of this construction can be found in Lakhotia et al. [35].

The use of finite field arithmetic in ER graphs gives rise to self-orthogonal vertices called *quadrics*, i.e. vectors $[u_1, u_2, u_3] \in \mathbb{F}_q$ so that $u_1^2 + u_2^2 + u_3^2 = 0$ in $\mathbb{F}_q$. The self-orthogonality of quadrics is represented by a self-loop incident on them. The PolarFly network ignores the self-loops for practical reasons.

Theorem 6.1. *[3, Proposition 2.2] ER_q is a diameter-2 graph for every prime power q. There is at most one path of length 2 between any pair of distinct vertices in ER_q.*

The quadrics lead to a natural categorization of ER_q vertices into three subsets [37, 45], as shown in Figure 1:

(1) Quadrics $W(q)$ that are self-orthogonal.
(2) Vertices $V_1(q)$ that are adjacent to quadrics.
(3) Vertices $V_2(q)$ that are not adjacent to quadrics.

Table 1 specifies the cardinality of these subsets in the entire graph as well as the neighborhood of individual vertices.

		$W(q)$	$V_1(q)$	$V_2(q)$
# of vertices in ER_q		$q+1$	$\frac{q(q+1)}{2}$	$\frac{q(q-1)}{2}$
# of neighbors of v, if:	$v \in W(q)$	0	q	0
	$v \in V_1(q)$	2	$\frac{q-1}{2}$	$\frac{q-1}{2}$
	$v \in V_2(q)$	0	$\frac{q+1}{2}$	$\frac{q+1}{2}$

Table 1: Vertex count of each type in the entire graph ER_q, and in the neighborhood of a single vertex (ignoring self-loops on quadrics).

6.1.1 PolarFly Layout.

Lakhotia et al. [37] proposed a modular layout for PolarFly that divides ER_q vertices into smaller clusters. For brevity, we restrict the discussion to odd prime powers q, which covers most design points of PolarFly. Note that we have a conceptually similar layout and an Allreduce solution for even q.

Algorithm 2 computes the PolarFly layout for ER_q. Lakhotia et al. [37] showed that Algorithm 2 adds every vertex to exactly one cluster, and proved several properties of the subgraphs induced by these clusters. The properties relevant to the derivation of Allreduce trees are listed below.

Property 1. *Contents of the Clusters:*

(1) Quadric cluster W has $q+1$ vertices and every non-quadric cluster C_i has q vertices, for a total of $N = q^2+q+1$ vertices.
(2) There are no edges between any pair of quadrics.
(3) The center v_i in a non-quadric cluster C_i is adjacent to all other vertices in C_i.

Property 2. *Connectivity between the quadric cluster W and a non-quadric cluster C_i:*

(1) There are $q+1$ edges between W and C_i.
(2) Every quadric is adjacent to exactly one vertex in C_i.
(3) Every $V_1(q)$ vertex in C_i is adjacent to exactly two quadrics.

Property 3. *Connectivity between distinct non-quadric clusters C_i and C_j:*

(1) There are $q-2$ edges between C_i and C_j.
(2) The center vertex v_j and one non-center vertex $u \in C_j$, are not adjacent to C_i.
(3) There is a non-starter quadric w' adjacent to both u and v_i (the center of C_i).

6.2 Singer Difference Set Graph Construction

The Singer difference set was introduced by Singer in 1938 [56]. Singer difference sets of order $q+1$ always exist if q is a prime power. None are known to exist for other orders, and their existence is a well-known open problem in mathematics [14].

Algorithm 2 PolarFly layout [37]

```
   Input: $ER_q$ graph of max degree $q+1$
   Output: Clusters $W, C_0 \dots, C_{q-1}$
1: Add all quadrics $W(q)$ to a cluster $W$
2: Select an arbitrary starter quadric $w \in W(q)$
3: for each vertex $v_i$ adjacent to $w$
4:     Add $v_i$ to an empty cluster $C_i$. $v_i$ is the center of $C_i$
5:     Add all non-quadric neighbors of center $v_i$ to $C_i$
```

The Symsig network used Singer difference sets in its construction in 2017 [7]. Erskine, Fratrič, and Širáň noted the isomorphism between the Singer graph and the Erdős-Rényi polarity graph ER_q in 2019 [14]. These papers provide much detail on the structure and construction of Singer difference sets and graphs. We outline here relevant details of their construction.

*Definition 6.2 (**Singer difference set**).* [7, 56] A *Singer difference set D* is a set of $q+1$ elements of $\mathbb{Z}_N$ such that the set $\{(d_i - d_j) \mod N \mid d_i, d_j \in D \text{ and } d_i \neq d_j\}$ is the set of all integers from 1 to $N-1 = q^2+q$, with no repetition.

Singer difference sets for $q = 3$ and 4 are shown in Figure 2. We use the following construction from [57] of Singer difference sets.

(1) Construct the finite Galois field $\mathbb{F}_{q^3}$ using a degree-3 primitive polynomial $f(x)$ over $\mathbb{F}_q$ with root ζ.
(2) List the q^3-1 powers of ζ in $\mathbb{F}_q^3$.
(3) Reduce each of these powers of ζ to its form $(i\cdot\zeta^2 + j\cdot\zeta + k)$ with $i, j, k \in \mathbb{F}_q$, using the fact that $f(\zeta) = 0$ in $\mathbb{F}_{q^3}$.
(4) Select those ζ^ℓ of the form $\zeta + k$. The difference set D is $\{\ell \mid \zeta^\ell = \zeta + k \mod f(\zeta), \text{ where } k \in \mathbb{F}_q\}$.
(5) Reduce all elements of D mod N.

Different primitive polynomials or roots may give different difference sets. For reproduction of our results and graphs, we used the lexicographically smallest degree-3 irreducible $f(x)$ over $\mathbb{F}_q$.

*Definition 6.3 (**Singer graph**).* [7] Let D be a Singer difference set D over $\mathbb{Z}_N$. The *Singer graph* $S_q = \{V, E\}$ has vertices $V = \{i \mid 0 \le i < N\}$, and has edges $E = \{(i,j) \mid (i+j) \mod N \in D\}$.

*Definition 6.4 (**Edge sum**).* Let $e = (i,j)$ be an edge of S_q. The sum $(i+j) \bmod N$ is called the *edge sum* of e.

*Definition 6.5 (**Reflection point**).* [7] Let D be a Singer difference set. An element i of $\mathbb{Z}_N$ is called a *reflection point* if $i+i \in D$, i.e. i has a self-loop.

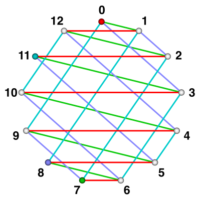
The use of the Singer difference set construction permits an easy edge-coloring of ER_q, where colors correspond to edge sums, which are elements of the difference set, by the definition of the edges. We later use that coloring to construct paths of two alternating colors, many of which are Hamiltonian, and collect Hamiltonian paths of different colors to form a large set of edge-disjoint spanning trees.

Example Singer difference sets and graphs for $q = 3$ and 4, with their edge-colorings, are shown in Figure 2, and corresponding collections of edge-disjoint Hamiltonian paths are shown in Figure 4.

6.3 Relationship of the Singer Graph to PolarFly

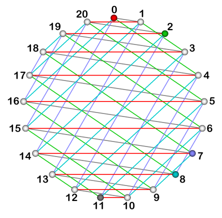
Erskine et al. showed the isomorphism of ER_q and S_q, in the following theorem [14]. This is also an immediate consequence of the

-	0	1	3	9
0	0	12	10	4
1	1	0	11	5
3	3	2	0	7
9	9	8	6	0



(a) A Singer difference set and graph for $q = 3$. The order of the graph N is $3^2 + 3 + 1 = 13$. This Singer difference set is $\{0, 1, 3, 9\}$. Reflection points (quadrics) are $\{0, 7, 8, 11\}$, and the corresponding vertices have self-loops.

-	0	1	4	14	16
0	0	20	17	7	5
1	1	0	18	8	6
4	4	3	0	11	9
14	14	13	10	0	19
16	16	15	12	2	0



(b) A Singer difference set and graph for $q = 4$. The order of the graph N is $4^2 + 4 + 1 = 21$. This Singer difference set is $\{0, 1, 4, 14, 16\}$. Reflection points (quadrics) are $\{0, 2, 7, 8, 11\}$, and the corresponding vertices have self-loops.

Figure 2: Two Singer difference sets and graphs. The tables show the difference sets in grey cells, and the differences generated in white cells. Each difference element corresponds to a color. All integers from 1 to q^2+q appear exactly once in each table. In the graphs, edges are colored according to their difference-set edge sums (sums mod N of the vertices they connect). Reflection points are colored by their self-loop color. The layout of the integers exhibits a striking parallelism between edges with the same edge sum.

relationship between finite projective planes and Singer difference sets shown much earlier in Berman [4]. The theorem implies that all results on Singer graphs discussed here apply to the family of Erdős-Rényi polarity graphs as well, and thus to PolarFly.

Theorem 6.6. *[14] The Singer graph S_q is isomorphic to the Erdős-Rényi polarity graph ER_q.*

This implies that the quadrics, V_1, and V_2 nodes in PolarFly correspond with vertices in the Singer graph, and may be discussed in terms of Singer difference sets.

First, we require the following lemma, which is true since $N = q^2 + q + 1$ is always odd.

Lemma 6.7. *In the ring $\mathbb{Z}_N$, 2^{-1} exists and is $\frac{q^2+q+2}{2} = \frac{N+1}{2}$.*

Corollary 6.8. *The quadrics of PolarFly are the elements w of S_q of the form $w = 2^{-1} \cdot d_i$, for some $d_i \in D$. Each quadric in ER_q is thus a reflection point in S_q and has a self-loop corresponding to a unique element of the difference set.*

Proof. Let w be a quadric. Then (w, w) is an edge, so $w + w = 2 \cdot w \mod N = d_i \in D$. So $w = d_i \cdot 2^{-1}$, by Lemma 6.7. Conversely, if $d \in D$, $d = 2 \cdot w_j$, for some quadric w_j. □

Corollary 6.9. *The V_1 nodes of PolarFly are the elements of S_q of the form $d_i - 2^{-1} \cdot d_j$, where $d_i, d_j \in D$ with $d_i \neq d_j$. The V_2 nodes of PolarFly are the elements of S_q that are neither quadrics nor V_1 nodes.*

Proof. The V_1 vertices are the neighbors of the quadrics. The corollary follows from Definition 6.3 and Corollary 6.8. □

We summarize these identifications between the two graphs:

- Corollary 6.8 identifies quadrics in ER_q with reflection points in S_q. Quadrics have self-loops, and are shown as colored vertices in the graphs in Figure 2.
- Corollary 6.9 identifies V_1 elements of PolarFly with neighbors of reflection points in S_q. V_1 elements are shown as neighbors of self-loop vertices in the graphs in Figure 2.

Together, these two corollaries show that the PolarFly layout can also be derived from Singer graphs, as this only relies on the classification of vertices into quadrics, V_1 and V_2 sets (Algorithm 2). Thus, our Allreduce solution based on PolarFly layout (discussed later in Section 7.1) can be derived for either construction of PolarFly.

7 IN-NETWORK ALLREDUCE ON POLARFLY

As discussed in Section 3, Allreduce performance reduces to an optimization problem: finding a large set of spanning trees in the network with low congestion and latency. Sections 4.3 and 5.1 show that the number of occurences of an edge in the set of spanning trees determines congestion, and the depth of the trees, latency.

We provide two Allreduce solutions on PolarFly. The first solution is based on the layout of PolarFly as shown by Lakhotia et al. [35], and provides a set of trees with congestion of 2 and depth at most 3. In this solution, there is some congestion as some of the edges in PolarFly appear in two different trees. The second is based on Singer difference sets [56], and provides a set of edge-disjoint trees, offering no congestion at the cost of higher depth.

We start with the following corollary of Theorem 5.1, giving the optimal bandwidth of in-network Allreduce in PolarFly.

Corollary 7.1. *For a PolarFly topology ER_q, the optimal bidirectional bandwidth of in-network Allreduce is $\frac{(q+1)B}{2}$.*

Proof. ER_q has $q + 1$ quadrics of radix q each (ignoring self-loops), and q^2 non-quadrics of radix $q + 1$ each, so the number of edges is $\frac{q(q+1)^2}{2}$. There are $q^2 + q + 1$ vertices, so each spanning tree uses $q^2 + q$ edges. The corollary then follows from Theorem 5.1. □

7.1 Low Depth Allreduce Trees

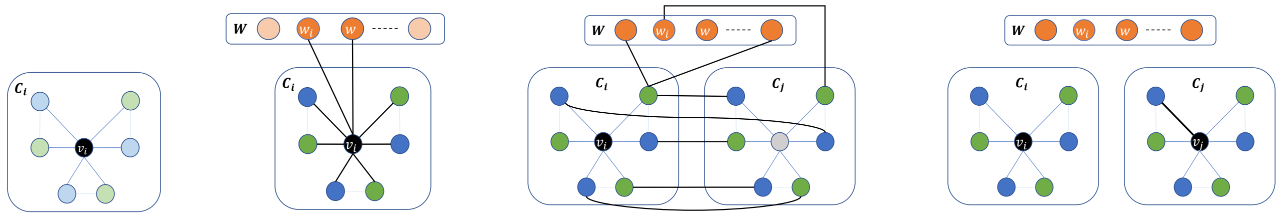
In this section, we create a set of high-bandwidth Allreduce trees with low depth for low latency. First, we analyze the connectivity between quadrics and cluster centers v_i in the following lemma.

Lemma 7.2. *Consider a layout (Algorithm 2) of the PolarFly topology $G(V, E)$ with starter quadric w. Given any pair of distinct non-quadric clusters C_i and C_j, the quadric neighbors of their centers v_i and v_j are $\{w, w_i\}$ and $\{w, w_j\}$, respectively, for some $\{w_i, w_j\} \in W(q)$ such that $w_i \neq w_j$.*

Proof. From Table 1, centers v_i and v_j have exactly two quadric neighbors. From the layout construction (Algorithm 2), one of these is the starter w which creates a 2-hop path (v_i, w, v_j). If the other quadric neighbors w_i and w_j are identical, there will be another 2-hop path between v_i and v_j, contradicting Theorem 6.1. □

Corollary 7.3. *Every non-starter quadric w_i is adjacent to exactly one unique cluster center v_i.*

Proof. Follows directly from Lemma 7.2 and the fact that there are q clusters and q non-starter quadrics (Table 1). □



(a) Root (level 0) (b) Vertices at level 1 (c) Vertices at level 2 (d) Vertices at level 3

Figure 3: Construction of depth-3 Allreduce Spanning Tree T_i. Level x represents vertices at distance x from the root. Edges used to span each level of T_i are highlighted in black. Vertices covered at or before the corresponding level of each subfigure are shown in a darker shade compared to uncovered vertices. (a) The root of T_i is the center v_i of cluster C_i (line 3, Alg. 3). (b) Vertices adjacent to v_i lie in level 1 of T_i. This comprises all non-center vertices in C_i, starter quadric w, and w_i – the non-starter quadric neighbor of v_i (Corollary 7.3 and line 5, Alg. 3). (c) The remaining quadrics and non-center vertices of other clusters $C_{j|j\neq i}$ are covered in level 2 of T_i (line 8, Alg. 3). For clarity, only one other non-quadric cluster is shown. (d) The centers of other clusters lie in level 3 of T_i (line 10, Alg. 3).

Following the notation from Lemma 7.2 and Corollary 7.3, we use w_i to denote the non-starter quadric neighbor of center v_i for all $0 \leq i < q$. Algorithm 3 derives q spanning trees in PolarFly with worst-case congestion 2 and depth 3. The construction of these trees is graphically and textually illustrated in Figure 3.

Algorithm 3 Low-Latency Spanning Trees in PolarFly

```
    Input: PolarFly $G(V,E)$, layout with starter quadric $w$, quadric cluster $W$ and non-quadric clusters $C_{i|0\leq i<q}$
    Output: set of $q$ spanning trees $\{T_0,...,T_{q-1}\}$ of $G$
1:  Initialize available edge set $E_a = E$
2:  for each $i \in \{0,1...,q-1\}$                    ▷ Construct $T_i$
3:      Assign center $v_i$ of $C_i$ as the root of $T_i$
4:      for each neighbor $u$ of $v_i$
5:          Add $u$ and $(v_i,u)$ to $T_i$             ▷ Cover $C_i, w, w_i$
6:          if $u \neq w$ then
7:              for each neighbor $z$ of $u$
8:                  if $z \notin T_i$ then add $z$ and $(u,z)$ to $T_i$
9:      for each non-quadric cluster $C_j$ for $j \neq i$
10:         Select any edge $(u,v_j) \in E_a$ incident with $v_j$
11:         Add $v_j$ and $(u,v_j)$ to $T_i$
12:         Remove $(u,v_j)$ from $E_a$
```

Theorem 7.4. *Every output T_i of Algorithm 3 is a spanning tree in $G(V,E)$.*

Proof. Consider the tree T_i, with root the cluster center v_i. Since the diameter of G is 2 (Theorem 6.1), any vertex is at most 2-hops away from v_i. Lines 4-8 cover all vertices within 2-hops of v_i except neighbors of starter quadric w, the cluster centers v_j for $j \neq i$.

From Property 3, no two centers are adjacent to each other. Hence, for any center $v_{j|j\neq i}$, all its neighbors are covered. Clearly, T_i does not have a cycle (line 8) and is a spanning tree if E_a has at least one edge of v_j when constructing T_i. Initially (line 1), E_a has $q+1$ edges of v_j and in any tree, at most one of these edges is removed. Thus, for any $i < q$, E_a contains non-zero edges of v_j. □

Theorem 7.5. *The depth of trees from Algorithm 3 is at most 3.*

Proof. Follows directly from the construction. □

Theorem 7.6. *Given a PolarFly $G(V,E)$, any edge $(u,v) \in E$ lies in at most 2 trees computed by Algorithm 3.*

Proof. Let $\Delta_i(u)$ denote the distance of u from the root in T_i. Since T_i is a tree, an edge (u,v) can be in T_i only if $\Delta_i(u) \neq \Delta_i(v)$. From Theorem 7.5, $\Delta_i(u) \leq 3$ for any i and u, and from Algorithm 3, we can see that $\Delta_i(u) = 3$ only if $u = v_j$ is a cluster for some $j \neq i$.

An edge (u,v) can belong to either of the three categories:

(1) Either u or v is a cluster center – Without loss of generality, assume that $v = v_i$ is a center for some $0 \leq i < q$. Then, (u,v_i) lies in T_i and at most one other tree when popped from E_a (line 10, Algorithm 3).
(2) Either u or v is a quadric, and neither is a cluster center – Without loss of generality, assume that $v = w'$ is a quadric and u is not a cluster center. From Property 1, u is not a quadric and from Algorithm 2, $w' \neq w$ is a non-starter quadric. Let $w' = w_i$ be adjacent to center v_i (Corollary 7.3) and $u \neq v_i$ be a non-center vertex in cluster $C_{j|j\neq i}$. Then (u,w') can only lie in trees T_i and T_j because for any $\ell \neq i$ and $\ell \neq j$, $\Delta_\ell(w_i) = \Delta_\ell(u) = 2$.
(3) u and v are neither quadrics nor cluster centers – Assume $u \in C_i$ and $v \in C_j$. As in case 2, (u,v) can only exist in T_i and T_j, as $\Delta_\ell(w_i) = \Delta_\ell(u) = 2$ for $\ell \neq i$ and $\ell \neq j$. □

Corollary 7.7. *The aggregate bidirectional Allreduce bandwidth of spanning trees obtained from Algorithm 3 is at least $\frac{B\cdot q}{2}$.*

Proof. From Theorem 7.6, the worst-case congestion is 2 for q trees. To conclude, plug this into Algorithm 1. □

Corollaries 7.1 and 7.7 show that the spanning trees generated by Algorithm 3 achieve near-optimal bandwidth and approach optimality asymptotically.

As discussed in Section 4.3, in-network Allreduce computes a reduction on the embedded tree and then broadcasts the output using the same tree. The partial sums communicated during reduction computation comprise the *reduction traffic*, and the packets containing reduction output comprise the *broadcast traffic*. The following lemma analyzes the flow of traffic on congested links (Algorithm 3).

Lemma 7.8. *Consider a link (u,v) that lies in two distinct trees T_i and T_j, computed by Algorithm 3. If the reduction traffic in T_i flows from u to v, then the reduction traffic in T_j flows from v to u.*

Proof. In an Allreduce tree, reduction traffic moves towards the root vertex (Section 4.3). Without loss of generality, we assume that in T_i, reduction traffic flows from u to v. Therefore, $\Delta_i(v) < \Delta_i(u)$.

To complete the proof, it suffices to show that $\Delta_j(v) > \Delta_j(u)$. If (u, v) is used in T_i and T_j and $\Delta_i(v) < \Delta_i(u)$, either of the following is true (from the proof of Theorem 7.6):

(1) $v = v_i$ is a center of C_i – in T_j, v is a leaf vertex. Hence, $\Delta_j(v) > \Delta_j(u)$.
(2) u is a center of C_j and (u, v) is popped from E_a in T_i – in T_j, u is the root. Hence, $\Delta_j(v) > \Delta_j(u) = 0$.
(3) $v = w_i$ is a non-starter quadric adjacent to v_i, and $u \in C_j$ is not a center – in T_j, u is adjacent to the root v_j, but v is not. Hence, $\Delta_j(v) > \Delta_j(u) = 1$.
(4) $v \in C_i$ and $u \in C_j$ are neither quadrics, nor cluster-centers – in T_j, u is adjacent to the root v_j, but v is not. Hence, $\Delta_j(v) > \Delta_j(u) = 1$. ☐

Lemma 7.8 highlights a practical advantage of the Allreduce trees computed by Algorithm 3: any input port on a router participates in at most one reduction mapped to the router. Some routers implement a wide-radix arithmetic engine, that can simultaneously compute multiple reductions on disjoint input ports [21, 36]. Given such an architecture, these trees can be embedded using a single arithmetic engine per router. We also note that routers in Intel PIUMA provide separate Virtual Channels (VCs) for reduction and broadcast traffic [36]. For such a system, the proposed low-latency solution does not require any additional VCs to handle congestion.

7.2 Edge-disjoint Allreduce Trees

To obtain a full bandwidth disjoint set of spanning trees, we use the Singer difference set representation of ER_q discussed in Section 6.2. This representation allows us to compute a large set of edge-disjoint trees, in the form of Hamiltonian paths. The number of trees in the set hits the upper bound of $\lfloor \frac{q+1}{2} \rfloor$ in all cases where $q < 128$.

*Definition 7.9 (**Non-repeating path**).* A path in a graph G is called *non-repeating* if no vertex appears twice in it.

*Definition 7.10 (**Maximal path**).* A path of a given type in a graph G is called *maximal* of its type if it is not contained in any longer path of the same type.

*Definition 7.11 (**Alternating-sum path**).* A path $(b_1, b_2, \ldots, b_k)$ in the Singer graph S_q is called an *alternating-sum* path if there exist distinct $d_0, d_1 \in D$ such that the edge sums of (b_{i-1}, b_i) are d_0 for i even and d_1 for i odd.

Definition 7.11 will be used in this construction of maximal edge-disjoint paths, since clearly, if two alternating-sum paths have four distinct edge sums, they will be edge-disjoint.

All maximal alternating-sum paths may be constructed using the element pairs (d_i, d_j) of the difference set D. In particular, the constructed path is Hamiltonian if and only if $(d_i - d_j)$ is relatively prime to N. This fact is proven in this section, and allows us to construct Hamiltonian spanning trees.

The following lemmas will be used to construct Hamiltonian paths in S_q, which are of course spanning trees.

Lemma 7.12. *Let $(b_1, b_2, \ldots, b_k)$ be a maximal alternating-sum non-repeating path on S_q, with alternating sums d_0 and d_1. Then k is odd, and b_1 and b_k are reflection points with $b_1 = 2^{-1} \cdot d_1$ and $b_k = 2^{-1} \cdot d_0$.*

Proof. The alternating-sum path is maximal, so there is no b_0 such that $b_0 + b_1 = d_1$. So $b_1 + b_1 = d_1$ (or equivalently, $b_1 = 2^{-1} \cdot d_1$, by Lemma 6.7) and b_1 is a reflection point.

Likewise, there is no b_{k+1} where $b_k + b_{k+1} = d_i$, where $d_i = d_0$ if k is odd and $d_i = d_1$ if k is even. So $b_k + b_k = d_0$ or d_1, depending on whether k is odd or even, and b_k must be a reflection point.

If k is even, then $b_k + b_k = d_1$. By Definition 6.5, $b_1 = b_k$. This makes $(b_1, b_2, \ldots b_k)$ a repeating path, contrary to assumption, so k is odd. This means that $b_k + b_k = d_0$ (or equivalently, $b_k = 2^{-1} \cdot d_0$). ☐

Theorem 7.13. *Let d_0 and d_1 be distinct alternating sums from the Singer graph S_q of order N. A unique maximal alternating-sum non-repeating path $(b_1, b_2, \ldots, b_k)$ in S_q with alternating sums d_0 and d_1 exists if and only if*

$$k = \frac{N}{\gcd(d_0 - d_1, N)}.$$

Proof. Let $(b_1, b_2, \ldots, b_k)$ be a maximum alternating-sum non-repeating path in the Singer graph. By Lemma 7.12, $b_1 = 2^{-1} \cdot d_1$ and $b_k = 2^{-1} \cdot d_0$. So

$$b_k - b_1 = 2^{-1} \cdot (d_0 - d_1). \tag{4}$$

The path is alternating-sum, so the element b_i for $1 < i \leq k$ is $d_1 - b_{i-1}$ if i is odd, and $d_0 - b_{i-1}$ if i is even. So $b_k = d_1 - b_{k-1}$ by Lemma 7.12. Substituting b_{i-1} in the equations for b_i, we get $b_k = 2^{-1} \cdot (k-1) \cdot (d_1 - d_0) + b_1$, and thus

$$b_k - b_1 = -2^{-1} \cdot (k-1) \cdot (d_0 - d_1). \tag{5}$$

Setting the right side of equations (4) and (5) equal gives

$$k \cdot (d_0 - d_1) \equiv 0 \mod (q^2 + q + 1).$$

The path is maximum with no repetition, so k must be

$$k = \frac{N}{\gcd(d_0 - d_1, N)}.$$

Conversely, assume there exist distinct d_0 and $d_1 \in D$ so that $k \cdot (d_0 - d_1) = 0 \mod N$. For any such d_0 and d_1, k is odd, since $N = q^2 + q + 1$ is odd. Consider the path defined as follows: b_1 is the reflection point $2^{-1} \cdot d_1$, b_k is the reflection point $2^{-1} \cdot d_0$, and b_i for $1 < i \leq k$ is $d_0 - b_{i-1}$ if i is even, and is $d_1 - b_{i-1}$ if i is odd. This is clearly an alternating-sum path. It is maximal: if there were a preceding b_0 part of the alternating-sum path, then $b_0 + b_1 = d_1$. But $b_1 + b_1 = d_1$, so $b_0 = b_1$. Likewise, $b_k = b_{k+1}$. It is non-repeating, since k is the smallest integer for which $k \cdot (d_0 - d_1) \equiv 0 \mod N$. ☐

Corollary 7.14. *A maximal alternating-sum non-repeating path in the Singer graph exists for every pair of distinct d_i and d_j in the difference set D, and each of these is unique (considering reversed paths as distinct).*

These paths need not be Hamiltonian. When the path is not Hamiltonian, other alternating-sum paths using the same (d_i, d_j) also exist, but are part of a cycle (thus either repeating or non-maximal), and do not span the vertex set.

The following corollary gives the construction of all maximal alternating-sum non-repeating paths in S_q. The example in Table 2 shows the non-Hamiltonian paths in S_4. Two examples of maximal sets of edge-disjoint Hamiltonians are shown in Figure 4.

Corollary 7.15. *Let d_0 and d_1 be a pair of alternating sums in the Singer graph S_q, and let*

$$k = \frac{N}{gcd(d_0 - d_1, N)}.$$

Then there is a unique maximal alternating-sum non-repeating path $(b_1, b_2, \ldots, b_k)$ in the Singer graph S_q where

(1) the path source b_1 is the reflection point $2^{-1} \cdot d_1$,
(2) the path sink b_k is the reflection point $2^{-1} \cdot d_0$,
(3) the path element b_i for $1 < i \leq k$ is $d_0 - b_{i-1}$ if i is even, and is $d_1 - b_{i-1}$ if i is odd,
(4) the path length is $k - 1$ and the number of vertices is k, and
(5) the path is Hamiltonian if and only if $d_0 - d_1$ is relatively prime to N.

This construction accounts for all maximal alternating-sum non-repeating paths in S_q.

Proof. Follows from Lemma 7.12, and Theorem 7.13. □

We know from Corollary 7.15 that the Hamiltonian alternating-sum non-repeating paths in the Singer graph S_q are exactly those generated by pairs (d_0, d_1) from the Singer difference set D, where $d_0 - d_1$ is relatively prime to N. Thus, it is easy to generate a Hamiltonian path in S_q by simply choosing two such elements d_0 and d_1 from D. Non-Hamiltonian maximal alternating-sum non-repeating paths exist when N is not prime, as seen in Table 2, but will not be the focus of this paper.

diff. set elts.		$gcd(d_0 - d_1)$	*# vertices*	*end points*	
d_0	d_1		k	b_0	b_1
0	14	7	3	7	0
1	4	3	7	2	11
1	16	3	7	8	11
4	16	3	7	8	2

Table 2: This table shows all non-Hamiltonian maximal alternating-sum non-repeating paths $(b_1, \ldots, b_k)$ over S_4 using the difference set $\{0, 1, 4, 14, 16\}$ from Figure 2b. These non-Hamiltonian paths come from those (d_0, d_1) whose difference $(d_0 - d_1)$ is not relatively prime to $|S_4| = 21$. This excludes the path reversals, which are generated by reversing d_0 and d_1. Each such path may be generated by starting with b_1, then appending $d_0 - b_{i-1}$ if i is even and $d_1 - b_{i-1}$ if not. The end points are reflection points.

To maintain zero congestion in Allreduce, we want to use edge-disjoint spanning trees, and to maximize bandwidth, we require as many edge-disjoint spanning trees as possible. Therefore we restrict our focus to the Hamiltonian paths.

Corollary 7.16. *Let d_0 and d_1 be two distinct elements of the Singer difference set D, and let $(b_1, b_2, \ldots, b_k)$ be the unique maximal alternating-sum non-repeating path in the Singer graph S_q that they generate. Then*

$$b_i = \begin{cases} \frac{i}{2} \cdot (d_1 - d_0) + b_1 & \text{if } i \text{ is even} \\ \frac{i+1}{2} \cdot d_0 - \frac{i-1}{2} \cdot d_1 - b_1 & \text{if } i \text{ is odd.} \end{cases}$$

This corollary leads to a description of the depth of a Hamiltonian path (which is a spanning tree) of S_q.

Lemma 7.17. *The optimal depth of an alternating-sum Hamiltonian path $(b_1, b_2, \ldots, b_N)$ is $\frac{N-1}{2}$, and the root mod N is*

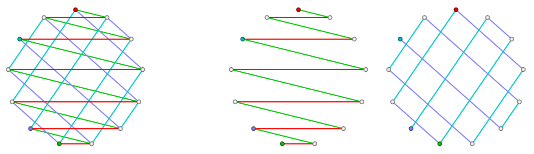
$$b_{(N+1)/2} = b_{2^{-1}} = \begin{cases} 4^{-1} \cdot (d_1 - d_0) + b_1 & \text{if } i \text{ is even} \\ 4^{-1} \cdot (3 \cdot d_0 + d_1) - b_1 & \text{if } i \text{ is odd.} \end{cases}$$

Proof. We choose the midpoint of the path to minimize the depth. This occurs at vertex $b_{(N+1)/2}$ and gives a depth of $\frac{N-1}{2}$. The formulation for the root is a consequence of Corollary 7.16. □

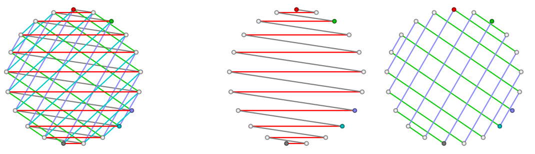
We will use sets of edge-disjoint Hamiltonian paths in Allreduce. In order to optimize bandwidth, we need to maximize the number of edge-disjoint Hamiltonian paths in S_q. Recalling that a Hamiltonian path is a spanning tree, we have the following upper bound.

Lemma 7.18. *The upper bound for the number of edge-disjoint Hamiltonian paths on S_q is $\frac{q+1}{2}$.*

Proof. There are $\frac{q \cdot (q+1)^2}{2}$ edges in S_q, and each Hamiltonian path has $q \cdot (q + 1)$ edges. □



(a) A maximal set of 2 edge-disjoint Hamiltonian paths for $q = 3$. One is colored with difference-set colors 0 (red) and 1 (green), and the other with colors 3 (blue) and 9 (cyan). All edges of S_3 are used in the two paths.



(b) A maximal set of 2 edge-disjoint Hamiltonian paths for $q = 4$. One is colored with difference-set colors 0 (red) and 1 (grey), and the other with colors 4 (green) and 14 (blue). The cyan edges from S_4 corresponding to difference set element 16 are not used in either of the two paths.

Figure 4: Two examples of maximal sets of $\lfloor (q + 1)/2 \rfloor$ edge-disjoint Hamitonian paths. S_q is shown on the left, and the Hamiltonian paths are shown to the right. Such (d_0, d_1)-colored paths start with the quadric having a self-loop of color d_1 and alternate edge colors until a self-loop of color d_0 is reached. All alternating-sum paths will be composed of two sets of joined parallel lines.

Theorem 7.19. *Let t denote the maximum number of edge-disjoint Hamiltonian paths on S_q. Then the aggregate bidirectional Allreduce bandwidth of these spanning trees is $B \cdot t$.*

If the optimal number of edge-disjoint Hamiltonian paths on S_q is achieved, then the aggregate Allreduce bandwidth is $\lfloor \frac{(q+1)}{2} \rfloor \cdot B$, which is the optimal bandwidth for PolarFly.

Proof. This follows from the fact that the spanning trees are edge-disjoint and the congestion model in Algorithm 1. □

In fact, we can often achieve the upper bound of $\frac{q+1}{2}$ for odd q and $\lfloor \frac{q+1}{2} \rfloor$ for even q, as illustrated by the examples in Figure 4.

- If N is prime, *all* maximal alternating-sum non-repeating paths in the Singer graph S_q are Hamiltonian.
- If N is composite, both non-Hamiltonian and Hamiltonian maximal alternating-sum non-repeating paths must exist. These are of cardinality k, where $\gcd(k, N) \neq 1$. All such paths are constructed as in Corollary 7.15.

It may seem like the existence of non-Hamiltonian paths when N is composite would imply that $\lfloor \frac{q+1}{2} \rfloor$ edge-disjoint spanning trees cannot be obtained. However, in Section 7.3, we show that a set of $\lfloor \frac{q+1}{2} \rfloor$ edge-disjoint Hamiltonian paths exists for all possible radixes of S_q in the range $[3, 129]$ (this is the upper bound $\frac{q+1}{2}$ when q is odd and is the maximal possible size when q is even).

To conclude this section, we provide a corollary to Theorem 7.13 counting exactly how many of these maximal alternating-sum non-repeating paths will be Hamiltonian.

Corollary 7.20. *The number of alternating-sum Hamiltonian paths in S_q is $\varphi(N)$, where φ is Euler's totient function.*

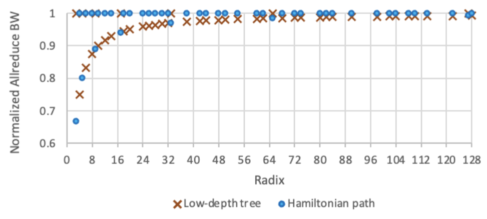
The totient function $\varphi(n)$ has generous bounds [43]:

- For prime n, the totient function is $n - 1$.
- For composite $n \neq 6$, $\sqrt{n} \leq \varphi(n) \leq n - \sqrt{n}$.

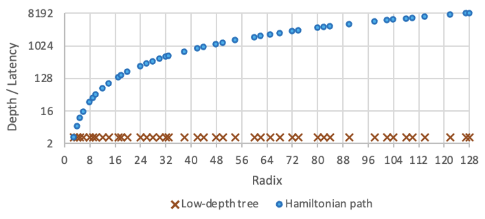
So even when N is composite, there are between $\frac{q+1}{2}$ and $\frac{q^2}{2}$ alternating-sum Hamiltonian paths to choose from.

7.3 Comparing the Two Solutions

We analyze the bandwidth achieved by our solutions for radixes in the range $[3, 129]$, which covers most of the design points of interest. Corollary 7.7 states that the bidirectional bandwidth for the low-depth solution is $\frac{qB}{2}$ if q is odd and is $\frac{(q+1)B}{2}$ if q is even (even q not covered in this paper for brevity). Recall that Theorem 5.1 shows that $\frac{(q+1)B}{2}$ is the optimal bandwidth for Allreduce in PolarFly.



(a) Comparison of Allreduce bandwidth, normalized against the optimal. Bandwidth of the Hamiltonian solution is optimal for all odd radixes (most of the feasible radixes). Bandwidth of the low-depth solution quickly approaches optimal for current- and next-generation high-radix routers.



(b) Comparison of tree depth, which is proportional to latency. Tree depth of the Hamiltonian solution is quadratic in terms of the radix, but tree depth of the low-depth solution is constant, giving consistent low latency.

Figure 5: Comparisons of Allreduce bandwidth and latency for the two approaches. For lower radixes, there is a tradeoff between bandwidth and latency, with large gaps in bandwidth at some radixes. For higher radixes, the low-depth solution provides both near-optimal bandwidth and constant low latency.

For our edge-disjoint Allreduce, the bandwidth is proportional to the maximum number of edge-disjoint Hamiltonian paths in the network as per Theorem 7.19, finding which can be expressed as an independent set problem as follows: Construct a graph $G_S(V_S, E_S)$, where (a) each vertex in V_S represents a unique pair of difference set elements whose maximal alternating-sum path is Hamiltonian, and (b) two vertices are connected if they have a common element. Clearly, the maximum independent vertex set of G_S gives a set of disjoint Hamiltonian paths of maximum cardinality in S_q.

Finding maximum independent sets is NP-hard in general. There are optimized algorithms to find a maximum independent set [24]. Here, we simply computed random maximal independent sets on difference sets (generated using galois.py [28], PARI [59] and the Online Encyclopedia of Integer Sequences [29]). We were able to find a maximum independent set in G_S for all radixes within 30 random instances, thus verifying that S_q contains a set of $\lfloor \frac{q+1}{2} \rfloor$ edge-disjoint Hamiltonian paths for all prime powers $q < 128$.

Figure 5 compares the Allreduce bandwidths and latencies for both solutions. Asymptotically, these solutions approach the optimal bandwidth. However, there is a clear trade-off between latency and bandwidth for small radix, as seen by comparing Figures 5a and 5b, and there are several data points for which there is a significant gap between the Hamiltonian and low-depth solutions.

8 RELATED WORK

Several works optimize host-based Allreduce using algorithmic [46, 51, 58], parameter tuning [16, 26, 60] and runtime [44, 47, 48] optimizations. A detailed overview of collective algorithms is provided in a survey by Moor and Hoefler [27]. However, all host-based implementations have fundamental limitations involving multiple communication rounds and network protocol overheads.

In-network computing is widely used to mitigate these challenges. Many networks provide hardware support for in-network reduction of short vectors [2, 15, 23, 33], which is a common operation in linear solvers and scientific computing applications [12, 49, 50].

Emerging AI/ML applications perform Allreduce on very large vectors [31, 54, 55], which tends to be bandwidth-bound and can bottleneck the performance for large systems and models. To address this, switches capable of link rate in-network aggregation have been proposed [19, 32, 40, 54], with demonstrable application-level benefits. Flare [11] uses general-purpose programmable switches for high-bandwidth Allreduce computation. General-purpose programmable devices can be used to implement our routines, but Flare is otherwise orthogonal. Intel PIUMA [1, 36] and Mellanox SHARP [21] deploy high bandwidth in-network Allreduce in real-world systems. BlueConnect uses parallel Reduce-scatter and Allgather computations to improve bandwidth utilization in hierarchical networks like Fat trees [10]. For flat networks with uniform link bandwidth and router radix, it is still gated by a single link bandwidth. This work advances the state of current practice by exploiting data parallelism in in-network Allreduce, boosting the bandwidth by more than an order of magnitude for high-radix networks.

9 CONCLUSION

The problem of optimizing Allreduce bandwidth can be mapped to that of finding multiple spanning trees in a network. We apply this to PolarFly, a diameter-2 topology whose rich mathematical structure affords sets of spanning trees with highly desirable properties.

In this paper, we provide two solutions that achieve near-optimal Allreduce bandwidth on PolarFly while minimizing congestion. In this first, we construct a large set of spanning trees in PolarFly with congestion just 2 and depth just 3. The second solution offers no congestion, albeit at the cost of high depth.

REFERENCES

[1] Sriram Aananthakrishnan, Nesreen K Ahmed, Vincent Cave, Marcelo Cintra, Yigit Demir, Kristof Du Bois, Stijn Eyerman, Joshua B Fryman, Ivan Ganev, Wim Heirman, et al. 2020. PIUMA: programmable integrated unified memory architecture.

[2] Baba Arimilli, Ravi Arimilli, Vicente Chung, Scott Clark, Wolfgang Denzel, Ben Drerup, Torsten Hoefler, Jody Joyner, Jerry Lewis, Jian Li, Nan Ni, and Ram Rajamony. 2010. The PERCS High-Performance Interconnect. In *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects (HOTI '10)*. IEEE Computer Society, USA, 75–82. https://doi.org/10.1109/HOTI.2010.16

[3] Martin Bachratý and Jozef Širáň. 2015. Polarity graphs revisited. *Ars Mathematica Contemporanea* 8 (01 2015), 55–67. https://doi.org/10.26493/1855-3974.527.74e

[4] Gerald Berman. 1953. Finite projective plane geometries and difference sets. *Trans. Amer. Math. Soc.* 74 (1953), 492–499.

[5] Maciej Besta and Torsten Hoefler. 2014. Slim Fly: A Cost Effective Low-Diameter Network Topology. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New Orleans, Louisana) *(SC '14)*. IEEE Press, 348–359. https://doi.org/10.1109/SC.2014.34

[6] Béla Bollobás. 2001. *Random Graphs* (2 ed.). Cambridge University Press. https://doi.org/10.1017/CBO9780511814068

[7] Dhananjay Brahme, Onkar Bhardwaj, and Vipin Chaudhary. 2013. SymSig: A low latency interconnection topology for HPC clusters. In *20th Annual International Conference on High Performance Computing*. 462–471. https://doi.org/10.1109/HiPC.2013.6799144

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[9] W. G. Brown. 1966. On Graphs that do not Contain a Thomsen Graph. *Can. Math. Bull.* 9, 3 (1966), 281–285. https://doi.org/10.4153/CMB-1966-036-2

[10] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. 2019. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *Proceedings of Machine Learning and Systems* 1 (2019), 241–251.

[11] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. 2021. Flare: Flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[12] Paul R Eller and William Gropp. 2016. Scalable non-blocking preconditioned conjugate gradient methods. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 204–215.

[13] Paul Erdős and Alfred Rényi. 1962. On a problem in the theory of graphs. *Publ. Math. Inst. Hungar. Acad. Sci.* 7A (1962), 623–641.

[14] Grahame Erskine, Peter Fratrič, and Jozef Širáň. 2021. Graphs derived from perfect difference sets. *Australas. J Comb.* 80 (2021), 48–56.

[15] Ahmad Faraj, Sameer Kumar, Brian Smith, Amith Mamidala, and John Gunnels. 2009. MPI collective communications on the Blue Gene/P supercomputer: Algorithms and optimizations. In *Symposium on High Performance Interconnects*. IEEE, 63–72.

[16] Ahmad Faraj, Xin Yuan, and David Lowenthal. 2006. STAR-MPI: self tuned adaptive routines for MPI collective operations. In *Proceedings of the 20th annual international conference on Supercomputing*. 199–208.

[17] Alan Frieze and Michał Karoński. 2015. *Introduction to Random Graphs*. Cambridge University Press. https://doi.org/10.1017/CBO9781316339831

[18] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 97–104.

[19] Nadeen Gebara, Manya Ghobadi, and Costa Paolo. 2021. In-network Aggregation for Shared Machine Learning Clusters. *Proceedings of Machine Learning and Systems* 3 (2021).

[20] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. 2016. Scalable hierarchical aggregation protocol (SHARP): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. IEEE, 1–10.

[21] Richard L Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, et al. 2020. Scalable hierarchical aggregation and reduction protocol (SHARP) streaming-aggregation hardware design and evaluation. In *International Conference on High Performance Computing*. Springer, 41–59.

[22] William Gropp. 2002. MPICH2: A new start for MPI implementations. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 7–7.

[23] JP Grossman, Brian Towles, Brian Greskamp, and David E Shaw. 2015. Filtering, reductions and synchronization in the Anton 2 network. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 860–870.

[24] Demian Hespe, Sebastian Lamm, and Christian Schorr. 2021. Targeted Branching for the Maximum Independent Set Problem. In *19th International Symposium on Experimental Algorithms (SEA 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[25] Torsten Hoefler, Tommaso Bonato, Daniele De Sensi, Salvatore Di Girolamo, Shigang Li, Marco Heddes, Jon Belk, Deepak Goel, Miguel Castro, and Steve Scott. 2022. HammingMesh: a network topology for large-scale deep learning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–18.

[26] Torsten Hoefler, Peter Gottschling, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Optimizing a conjugate gradient solver with non-blocking collective operations. *Parallel Comput.* 33, 9 (2007), 624–633.

[27] Torsten Hoefler and Dmitry Moor. 2014. Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Supercomputing frontiers and innovations* 1, 2 (2014), 58–75.

[28] Matt Hostetter. 2020. Galois: A performant NumPy extension for Galois fields. https://github.com/mhostetter/galois

[29] OEIS Foundation Inc. 2023. The On-Line Encyclopedia of Integer Sequences: Sequence A333852. https://oeis.org/A333852.

[30] Nikhil Jain and Yogish Sabharwal. 2010. Optimal bucket algorithms for large MPI collectives on torus interconnects. In *Proceedings of the 24th ACM International Conference on Supercomputing*. 27–36.

[31] Janis Keuper and Franz-Josef Preundt. 2016. Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*. IEEE, 19–26.

[32] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. 2020. An in-network architecture for accelerating shared-memory multiprocessor collectives. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 996–1009.

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

[34] Gyuyoung Kwauk, Seungkwan Kang, Hans Kasan, Hyojun Son, and John Kim. 2021. BoomGate: Deadlock Avoidance in Non-Minimal Routing for High-Radix Networks. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 696–708.

[35] Kartik Lakhotia, Maciej Besta, Laura Monroe, Kelly Isham, Patrick Iff, Torsten Hoefler, and Fabrizio Petrini. 2022. PolarFly: a cost-effective and flexible low-diameter topology. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–15.

[36] Kartik Lakhotia, Fabrizio Petrini, Rajgopal Kannan, and Viktor Prasanna. 2021. Accelerating Allreduce with in-network reduction on Intel PIUMA. *IEEE Micro* 42, 2 (2021), 44–52.

[37] Kartik Lakhotia, Fabrizio Petrini, Rajgopal Kannan, and Viktor Prasanna. 2021. In-network reductions on multi-dimensional HyperX. In *2021 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 1–8.

[38] Kartik Lakhotia, Fabrizio Petrini, Rajgopal Kannan, and Viktor Prasanna. 2022. Accelerating Prefix Scan with in-network computing on Intel PIUMA. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE.

[39] Fei Lei, Dezun Dong, Xiang-Ke Liao, and José Duato. 2020. Bundlefly: a low-diameter topology for multicore fiber. In *Proceedings of the 2020 International Conference on Supercomputing*.

[40] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating distributed reinforcement learning with in-switch computing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 279–291.

[41] Rudolf Lidl and Harald Niederreiter. 1997. Finite Fields: Encyclopedia of Mathematics and Its Applications. *Computers & Mathematics with Applications* 33, 7 (1997), 136–136.

[42] Robert J. McEliece. 1987. *Finite Fields for Computer Scientists and Engineers*. Springer, Boston, MA.

[43] D.S. Mitrinović, J. Sándor, and B. Crstici. 1995. *Handbook of Number Theory*. Kluwer Academic Publishers, Dordrecht. 9 pages.

[44] Emin Nuriyev and Alexey Lastovetsky. 2020. Accurate runtime selection of optimal MPI collective algorithms using analytical performance modelling. *arXiv preprint arXiv:2004.11062* (2020).

[45] T. D. Parsons. 1976. Graphs from projective planes. *Aequat. Math.* 14 (02 1976), 167–189. https://doi.org/10.1007/BF01836217

[46] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117–124.

[47] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E Fagg, Edgar Gabriel, and Jack J Dongarra. 2007. Performance analysis of MPI collective operations. *Cluster Computing* 10, 2 (2007), 127–143.

[48] Jelena Pješivac-Grbović, George Bosilca, Graham E Fagg, Thara Angskun, and Jack J Dongarra. 2007. Decision trees and MPI collective algorithm selection problem. In *European Conference on Parallel Processing*. Springer, 107–117.

[49] Karl E Prikopa, Wilfried N Gansterer, and Elias Wimmer. 2016. Parallel iterative refinement linear least squares solvers based on all-reduce operations. *Parallel Comput.* 57 (2016), 167–184.

[50] Rolf Rabenseifner. 2000. Automatic MPI counter profiling. In *42nd cug conference*. 396–405.

[51] Rolf Rabenseifner. 2004. Optimization of collective reduction operations. In *International Conference on Computational Science*. Springer, 1–9.

[52] Aniruddh Ramrakhyani, Paul V Gratz, and Tushar Krishna. 2018. Synchronized progress in interconnection networks (SPIN): A new theory for deadlock freedom. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 699–711.

[53] Paul Sack and William Gropp. 2015. Collective algorithms for multiported torus networks. *ACM Transactions on Parallel Computing (TOPC)* 1, 2 (2015), 1–33.

[54] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with {In-Network} Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 785–808.

[55] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799 http://arxiv.org/abs/1802.05799

[56] James Singer. 1938. A Theorem in Finite Projective Geometry and Some Applications to Number Theory. *Trans. Amer. Math. Soc.* 43, 3 (1938), 377–385.

[57] Douglas R. Stimson. 2004. *Combinatorial Designs: Constructions and Analysis*. Springer, New York, NY. 52–54 pages.

[58] Rajeev Thakur and William D Gropp. 2003. Improving the performance of collective operations in MPICH. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 257–267.

[59] The PARI Group 2022. *PARI/GP version* 2.13.4. The PARI Group, Univ. Bordeaux. available from http://pari.math.u-bordeaux.fr/.

[60] Udayanga Wickramasinghe and Andrew Lumsdaine. 2016. A survey of methods for collective communication optimization and tuning. *arXiv preprint arXiv:1611.06334* (2016).