Collectives Are Network Programs

Seyyidahmed Lahmer
IMEC

Kalyan Pullela
Oklahoma State University

H.B. Acharya
Oklahoma State University

Abstract

Collective communication has a clean abstraction at the application boundary, but how its execution is expressed inside the network depends on the offload mechanism. With the rise of in-network computation, collective execution is increasingly spread across the network. Yet deploying an offload often requires a matched stack of runtime plugins, NICs, and switches tied to vendor-specific interfaces. Even when the logic is the same, moving an operation between these stacks can require a different packet format and recovery mechanism.

We argue that collective execution should be represented as *network programs*: packet-carried instruction sequences in which each instruction names a bounded operation and the device that executes it. Endpoints encode these sequences, while network devices execute the named operations and maintain the required local state. Separating operations from their placement and device-specific implementation provides a common way to express execution across different devices. We reuse the existing IPv6 Segment Routing Header to carry these programs, adding collective behaviors to SRv6's network programming model. Our simulations demonstrate that the operation sequence is preserved across different placements, while our prototype shows that streaming reduction is feasible on a programmable switch.

CCS Concepts

• **Networks → In-network processing**; **Programmable networks**; **Network design principles**; **Programming interfaces**.

Keywords

Network programming, in-network computing

ACM Reference Format:
Seyyidahmed Lahmer, Kalyan Pullela, and H.B. Acharya. 2026. Collectives Are Network Programs. In *The 25th ACM Workshop on Hot Topics in Networks (HotNets '26), November 16-17, 2026, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3847290.3859934



This work is licensed under a Creative Commons Attribution 4.0 International License.
HotNets '26, Salt Lake City, UT, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859934

1 Introduction

Collective communication has a clear interface at the application boundary. MPI lets applications request operations such as broadcast, reduce, and all-reduce [19]; NCCL provides these operations for GPU clusters [23]. The application specifies the operation, and the runtime decides how to execute it. Systems such as SCCL and TACCL synthesize collective schedules [5, 25]. MSCCL lets developers describe data movements and reductions over buffers associated with GPU ranks and compile these schedules for its runtime [7].

Collective execution increasingly extends into the network. SwitchML and ATP aggregate gradients inside programmable switches [17, 24], and SHARP provides aggregation in InfiniBand fabrics [13]. Yet deploying an offload often requires a matched stack of runtime plugins, NICs, and switches tied to vendor-specific interfaces [22]. The same collective operation is expressed using different packet formats across these systems. The execution plan also depends on configuration installed in the network. EPIC, for example, pursues a common interface for in-network collectives while implementing its trees through forwarding rules and reduction state installed in switches [31].

Recent work on MRC and SRv6 illustrates a different division of responsibility for packet routing [1]. Endpoint NICs select packet paths and avoid failed paths, while switches follow static forwarding rules. This places adaptation at the endpoints and simplifies network control. We ask whether collective execution can follow a similar principle: endpoints define the collective tree and specify where each operation executes, while network devices carry out the corresponding local operations.

We propose *network collective programming* (NCP): endpoints encode the network's part of an execution plan as a packet-carried sequence of instructions. Each instruction names an operation, its execution location, and arguments identifying the required local state. The runtime chooses the aggregation tree and assigns operations to devices; the source NICs encode the corresponding instructions in packets. Participating network devices execute their assigned operations using resources allocated and state configured during tree initialization. The endpoints thus determine how

network operations are combined into a collective, while each device implements the operations it supports.

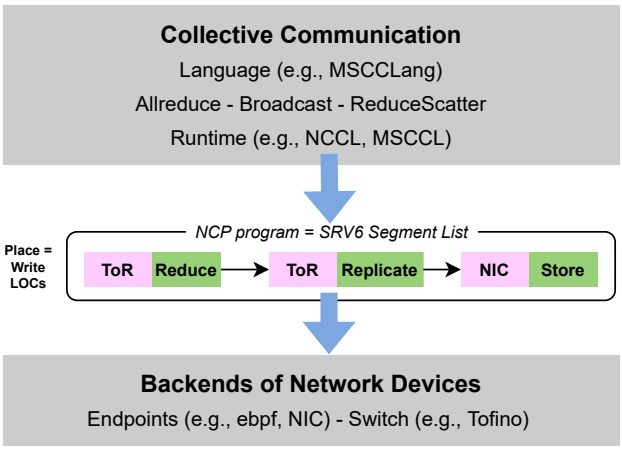


Figure 1: NCP as a common interface for collective execution. Runtimes encode network operations as packet-carried `LOC:FUNC:ARG` instructions implemented by participating network devices.

NCP is independent of the language used to describe the collective. Figure 1 shows how NCP connects collective descriptions to the devices that execute their network operations. MSCCL organizes execution around GPU ranks while NCP expresses the operations assigned to network devices. A compiler targeting NCP would translate the supported parts of a collective description into network instructions and retain the remaining work at endpoints. Depending on the operations and network device capabilities, the network program may implement a complete collective or only part of it. NCP therefore provides an execution interface that collective languages and compilers can target, rather than another language for defining collective algorithms.

We encode this model using IPv6 Segment Routing [10–12]. Its existing Segment Routing Header carries an ordered list of instructions, each naming a function at a network location. NCP adds collective behaviors to this network programming model without defining a separate collective header. The instruction model specifies what each operation does, SRv6 encodes the sequence and execution locations, and each device supplies its implementation of the supported operations.

Separating operations from their placement and device-specific implementation lets endpoints express different execution plans using the same instruction format. For a fixed collective tree, assigning an operation to another compatible device changes its location in the program. Adding a reduction level extends the instruction sequence. Section 5.1 evaluates these changes in simulation, showing that the operation sequence is preserved across placements and that an additional reduction level can be expressed within the same format. Section 4.3 demonstrates that streaming reduction is feasible on a programmable switch. Section 6 examines the implications for device capability discovery, resource sharing, and runtime placement.

Table 1: The abstract NCP instruction set. Each instruction is a tuple `LOC:FUNC:ARG`; `ARG` keys index local node state.

Instruction	ARG	Effect	Node State
`Reduce`	`q_id, p_id`	Accumulate	Slot, mask, ptrs
`Replicate`	`gid`	Multicast	gid → children
`Fetch`	`mem_id, off`	NIC dma read	`mem_id` → base
`Store`	`mem_id, off`	NIC dma write	`mem_id` → base

2 Collectives Are Network Programs

NCP represents collective execution as a packet-carried program: a sequence of instructions, each naming an *operation*, a *location* (the executing device), and *arguments*. Operation and location are specified independently. A packet carries an execution pointer identifying the current instruction. At the named location, the device performs the operation using the referenced local state. When the operation completes, execution continues with the next instruction.

Each instruction is a bounded network operation. We consider four instructions (Table 1): `Reduce`, `Replicate`, `Fetch`, and `Store`. `Reduce` accumulates contributions across packet arrivals. The other instructions do not accumulate partial results, but can still require configured state, such as multicast groups or memory mappings.

These instructions express tree reductions and broadcasts. A tree all-reduce consists of an upward sequence of `Reduce` operations, followed by a downward sequence of `Replicate` operations and a `Store` at each destination. A broadcast uses `Replicate` to distribute the data, and `Store` to place the copies in destination memory.

The runtime continues to choose the collective algorithm and the devices that execute it. Each device implements the instructions according to its capabilities. A device need not support the full instruction set: the runtime assigns an operation only to a device that supports it. Devices that only forward packets need not implement collective operations. Section 3 describes the SRv6 encoding; Section 4 describes our implementation of `Reduce`.

2.1 Packets, Segments, and State

Consider an AllReduce that combines gradient chunks from four GPUs during distributed training (Figure 2). Each GPU

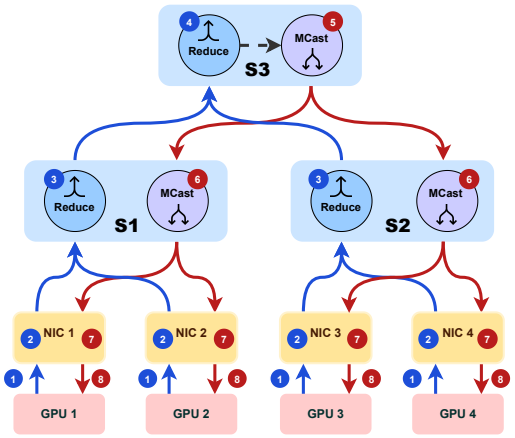


Figure 2: AllReduce over four GPUs. S1 and S2 produce partial results, S3 combines them, and replication distributes the result back to the GPUs.

passes its contribution through its NIC, which attaches the corresponding program 1 and 2. Switches S1 and S2 each combine two contributions 3, and S3 combines their partial results 4. The result is replicated at S3 and then at the leaves 5 and 6. The NICs store the result in GPU memory 7 and 8, so all four GPUs receive the same reduced chunk.

An NCP packet carries both data, such as a tensor chunk or partial result, and the instructions that direct its execution. Instruction arguments refer to local state; the packet need not carry the full configuration of each participating device.

Before execution, tree initialization allocates the resources needed at each participating device. The source NICs receive the identifiers used to reference these resources. Packets carry the execution sequence: they specify the order of operations, the devices that execute them, and the local resources each operation uses.

In our streaming implementation, we assume the same credit allocation on every edge of the collective tree. This allocation limits how many contributions a receiving device can hold outstanding from each producer at a time. At an intermediate reduction level, a producer is an upstream reducer supplying partial results. During execution, progress is driven by packet arrivals, including data and credit packets; it does not depend on timers or other external triggers.

A packet need not survive from source to destination. A reduction consumes several contributions and emits one aggregate, which continues with the remaining instructions in the packet-carried program. Replication produces copies that continue toward their respective destinations. The program describes how the data is processed and distributed, even when the packets carrying it change.

NCP applies the idea of packet-directed execution from Active Networks [28, 29] to collective communication. Packets invoke a defined set of operations implemented by participating devices. Collectives provide a concrete use for this model: endpoints compose reduction, replication, and memory operations into an execution plan, while devices implement the individual operations.

2.2 Hierarchical Composition

The same instructions compose across levels of a tree. In the example, S1 and S2 receive contributions from their attached GPUs and each emit one partial result to S3. From S3's perspective, S1 and S2 are its two producers. The reduction operation has the same meaning at both levels; the producer sets and associated state differ.

Each additional reduction level adds a `Reduce` instruction to the upward sequence. The downward sequence uses `Replicate` at the branching nodes. Changing the tree, therefore, involves updating the instruction sequences while retaining the same instruction definitions and packet format. Section 5.1 evaluates different placements and tree structures within this representation.

3 Encoding with SRv6

We encode the instruction model of Section 2 using IPv6 Segment Routing. Algorithm selection remains with the runtime; this section describes how packets specify the operations to execute and their locations. The existing IPv6 Segment Routing Header (SRH) carries a list of 128-bit segment identifiers (SIDs) [10, 11]. Each SID identifies a function at a network location and can include arguments interpreted by that device. An NCP instruction maps to one SID, and an instruction sequence maps to a segment list.

We use the `LOC:FUNC:ARG` structure: `LOC` identifies the execution location, `FUNC` identifies the operation, and `ARG` references the local resources it uses. For example, a reduction instruction identifies both the device that performs the reduction and the reduction context allocated there (i.e., queue id + producer id). The packet carries the sequence of operations; initialization allocates the resources (i.e., reduction queue) referenced by those operations.

Consider the upward phase of the running example in Figure 2. Packets from the GPUs attached to S1 first carry a segment identifying `Reduce` at S1, followed by a segment identifying `Reduce` at S3. Packets from the other branch first identify S2 and then S3. The arguments select the corresponding reduction resources at each switch. After S1 or S2 combines its local contributions, the resulting packet continues to the reduction at S3 specified by its segment list.

Separating location from operation allows the runtime to place an operation on any device that supports it. Adding a reduction level inserts another reduction segment into the

upward sequence, together with the resource allocation and changes to the downward sequence required by the new tree.

The encoding uses SRv6's existing representation, but collective execution requires additional endpoint behaviors.

Stateful behaviors. A reduction combines data across several packet arrivals. Its local state records the partial result and which contributions have arrived. This differs from processing each packet independently: the arrival of one contribution updates the state, while the arrival that completes the reduction also causes an aggregate to be emitted. Section 4 describes our streaming implementation, where packet arrivals drive these actions without timers or other external triggers. The queue organization and credit allocation belong to that implementation and are independent of the SRv6 instruction format.

Consuming semantics. Reduction consumes several contribution packets and produces one aggregate packet. The aggregate continues with the instructions following the reduction in the packet-carried program. Network devices need not know the complete collective tree: participating endpoints encode its execution in the packets, rather than install it as a program when reduction resources are allocated. Each device uses its local resources to execute the current instruction, while the remaining instructions specify how the result continues.

NCP thus reuses the SRH format and the SRv6 model of functions identified at network locations, while adding behaviors for collective operations. Participating devices must implement these behaviors; SRv6 support alone does not provide reduction. The instruction model does not depend on SRv6, but SRv6 provides an existing encoding that separates location, function, and arguments without introducing a separate collective header.

4 Reduce: Streams or Memory?

Of the five instructions, `End.Replicate` resembles multicast, `End.Forward` resembles routing, and `End.Fetch`/`End.Store` perform endpoint DMA. Reduction, however, combines arithmetic with state that persists across packet arrivals (Section 3). How a device maintains this state affects resource use, composition, and hardware support. While the abstraction specifies the operation and its completion condition, it leaves state management to the implementation. We now consider two implementation models.

(1) *Memory-mapped* model: exposes addressable memory in switch. Endpoints issue store-and-reduce operations (a write to address *a* atomically adds into what is stored there), and a switch may initiate RDMA reads to pull operands. e.g. NVSwitch multicast-mem [21] and, in spirit, SHARP [13].

(2) *Stream* model: exposes a pool of reduction queues to which endpoints send contribution packets. Each arrival drives a state transition: the switch combines the contribution with the running partial result, checks the fan-in, and emits an aggregate when the slot completes.

We adopt the stream model, (2), for two reasons. First, it simplifies the switch: without timers, RDMA read engines, or threads, its behavior is driven entirely by packet arrivals. Second, it simplifies failure recovery. A memory-mapped switch loses both accumulated aggregates and in-flight pull requests when it fails; a stream switch loses only accumulated state, and endpoints can recover using their knowledge of what they sent.

Pulls remain available through `End.Fetch`, but the switch itself only reacts to packet arrivals: the request triggers the fetch on the endpoint side, and the arriving contributions on the switch trigger reduction. Whether a mature system should support both models remains an open question (see §6).

4.1 Asynchronous Reduction Queues

We realize the stream model as circular reduction queues on a programmable switch (Figure 3). Each queue is one active reduction stream with several reusable slots; a slot accumulates one contribution per producer and emits when the configured fan-in is reached. The design goal is to avoid lockstep: *independent producer pointers* let a fast producer run ahead, while a slow one delays completion of the slots to which it has not yet contributed. A contribution identifies

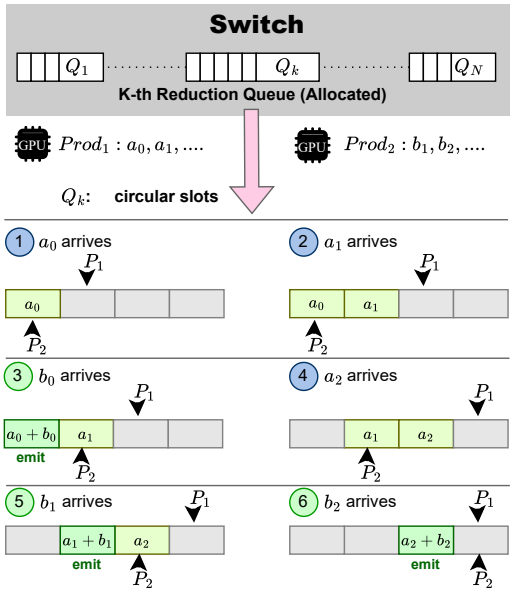


Figure 3: Asynchronous reduction in an allocated switch queue.

its reduction queue and producer through (*queue*, *producer*).

Within each queue, the switch maintains a pointer for each producer that selects the slot for its next contribution. Figure 3 illustrates this process with two producers and four slots. Producer 1 contributes a_0 and a_1 to successive slots (steps 1–2). Producer 2 then contributes b_0, completing the first slot and triggering emission of a_0+b_0 (step 3). Producer 1 advances again with a_2 (step 4), before producer 2 completes the next two slots with b_1 and b_2 (steps 5–6).

The runtime can divide a collective into channels, with endpoints encoding each channel's aggregation tree, reduction queues, and contribution paths. We assume in-order delivery from each producer along these paths, allowing its local pointer to map successive contributions to successive slots.

4.2 Receiver-Driven Credits

In the stream model, the receiver controls when a sender injects packets into the network. NCP does this with credits: a reducer grants each producer one credit per free slot, and a producer sends only while it holds credits. This is the natural congestion boundary for a reduction tree: each sender is rate-limited hop-by-hop by the next reducer, not by an end-to-end rate loop [33]. A ToR grants credits to its attached producers; a spine grants credits to the ToR reducers; the rule composes unchanged across tiers. The scheme is receiver-driven in the spirit of credit-based transports [6, 20], but applied per-reducer along the tree rather than end-to-end, so back-pressure forms exactly where fan-in concentrates. Our prototype assumes a lossless fabric, so credits sit above priority-based flow control rather than replace it; whether per-reducer credits can safely coexist with DCQCN-governed RDMA on shared links, especially across overlapping reduction trees, is an open question (Section 6).

4.3 Feasibility on Hardware

We implemented the stream reducer on a Tofino-1 ASIC (SDE 9.13.1) with sixteen 32-bit lanes per vector (a 64-byte payload). To emulate multiple producers, we developed a custom DPDK traffic generator running on a single server. Each queue uses disjoint register state, so concurrent collectives do not serialize. Across 64 concurrent queues, we sustain 43.4 million vector reductions / sec, and every run reproduces the exact sum (no drops, no corruption). The reduction is integer/fixed-point, so the sum is bit-exact despite the nondeterministic combination order of §4.1 (unlike floating-point addition, integer addition is associative, so changing arrival order does not change the result). This does inherit SwitchML's [24] limitation of requiring endpoint quantization for gradients; improving on this is future work.

We note that the reduction itself is a single stateful-ALU update folded into the match-action pipeline; it consumes no extra pipeline pass and so does not lower the switch's forwarding rate. 43.4 M/s is a *generator* bound, not a switch bound: it is the rate a single server driving the switch over one 100G link with a DPDK generator can produce, throttled by its packet rate and the credit window ($43.4\,\mathrm{M} \times 64\,\mathrm{B} \approx 22$ Gbps, well under link capacity). Finding the ASIC's true ceiling with a multi-server generator is future work.

5 Experimental Evaluation

This section examines whether the same instruction sequence can execute at different placements (Section 5.1) and how the resulting collective scales (Section 5.2).

We simulate a Clos network with 400 Gbps links and a 4 KB MTU in ASTRA-sim 2.0 and ns-3.42 (DCQCN, PFC, ECN). NCP injection is governed by the per-hop receiver credits of Section 4.2. We compare NCP against Ring and double-binary Tree, as baseline *algorithms* from ASTRA-sim (the endpoint-based schedules a runtime uses today).

5.1 Placement Invariance

First, we show the same instruction sequence executed at different devices. Placement invariance requires a fixed aggregation-tree structure: the tree determines the instruction sequence, while the `LOC` fields specify where each instruction executes. We evaluate this with a single flat NCP AllReduce program (one `Reduce` followed by one `Replicate`), executed with the reducer at the host NIC (NIC-only), the rack ToR (Leaf-only), and the spine (Spine-only). Across all three, the instruction sequence is unchanged; the execution locations differ. Each placement requires the corresponding local resources to be allocated at the participating devices.

Our representation also supports changes to the tree structure. Spine+Leaf implements a two-level tree, where each ToR aggregates its rack before a spine combines the partials, by inserting one `Reduce` segment into the upward sequence and updating replication in the downward sequence. This is a different program, not a different placement of the same one; but it is a change to the packet-carried program within the same representation, sharing the packet format, instruction set, and data-plane behaviors. NCP programs thus admit two granularities of change: *re-placement* (rewrite `LOC`, structure unchanged) and *restructuring* (insert or remove segments), while retaining the same packet format and instruction definitions. Both require appropriate resources at the selected devices, including reduction queues and credits (Section 6).

We sweep message sizes from 64 KB to 16 MB for deployments within a single rack ($N = 8, 16$) and across two racks ($N = 32$). Every placement is functionally correct: all configurations complete the AllReduce and produce identical results. Figure 4 reports their completion times.

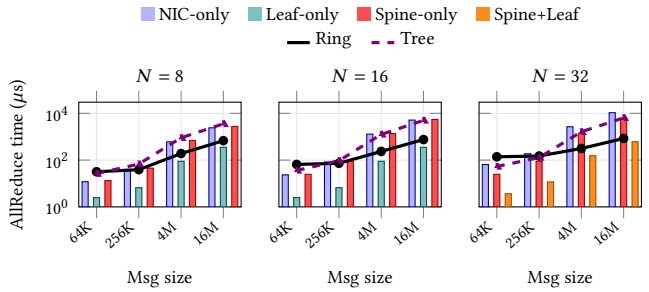


Figure 4: AllReduce performance with GPU count (N = 8, 16: one rack; N = 32: two racks of 16; log scale). Bars: NCP placements; lines: endpoint baselines.

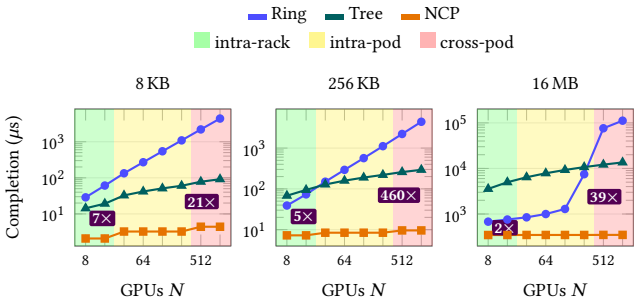


Figure 5: AllReduce completion as GPU count grows, at different message sizes. Bands: placement tier; annotations: NCP speedup over Ring/Tree at extremes.

In a single rack (N = 16), *Leaf-only* is consistently the fastest because the ToR connects directly to all GPUs and avoids an incast bottleneck. At 16 MB, it outperforms Ring by 2.1× (353 μs vs. 758 μs) and Tree by 14× (4.98 ms). At 64 KB, it is 26× faster than Ring. In contrast, *NIC-only* and *Spine-only* concentrate traffic at a single root behind an oversubscribed uplink, making them approximately six to seven times slower than Ring at 16 MB.

Across two racks (N = 32), a single root remains incast-bound, whereas *Spine+Leaf* first reduces traffic within each rack before combining the partial results at the spine. This reduction in traffic between racks makes *Spine+Leaf* the fastest configuration, outperforming Ring by 1.4× and Tree by 10× at 16 MB, and Ring by 38× at 64 KB.

No single configuration wins: in one rack the one-level Leaf-only program is fastest; across racks the restructured Spine+Leaf program takes over, while any single root is incast-bound. *Well-placed* in-network reduction thus delivers the gains in-network aggregation promises [13, 24], but good placement depends on scale and topology, and can change with operating conditions (congestion, multi-tenancy, failures...). NCP expresses these changes through updates to execution locations and instruction sequences within the same packet format. It therefore establishes a foundation for future work in dynamic placement selection (§6).

5.2 Scaling and Representation Cost

We extend the one- and two-rack placement experiments to a three-level fat-tree, increasing GPU count from within one rack to across pods (Figure 5). Each reduction level combines its incoming contributions into one outgoing stream. In the evaluated configurations, completion time depends mainly on tree depth and message size (Figure 5). In the latency regime (8 KB), completion rises in discrete steps of a few microseconds, one per tier added, and is flat in N within a tier. In the bandwidth regime (16 MB), NCP is flat near 345 μs from 8 to 1024 GPUs, as each tier forwards a single reduced stream; Ring climbs from 679 μs to 112 ms as cross-pod links oversubscribe, and Tree ships the full message at each of $2 \log N$ levels.

Header overhead. An SRH occupies $8 + 16S$ bytes for S full-length segments, without optional TLVs [11]. A flat tree uses three segments: `Reduce`, `Replicate`, and `Store`, requiring 56 bytes. Adding one `Reduce` segment increases this to 72 bytes. For a 4096-byte IP MTU, these headers occupy 1.37% and 1.76% of the packet, respectively. Including the 40-byte IPv6 header leaves payload fractions of 97.66% and 97.27%. These estimates assume full-sized packets without additional encapsulation and exclude transport headers, link-layer overhead, and credit traffic.

6 Discussion and Open Questions

Treating collective execution as a program of placeable instructions opens several research questions.

NCP lets endpoints compose and place network operations through a common interface, while devices execute bounded operations using local resources. Runtimes can exploit locality or resource availability when choosing placements (Section 5.1). For ML workloads, hierarchical reduction reduces communication overhead; in our scaling experiment, completion time remains nearly constant with GPU count at fixed aggregation depth. The same structure could support latency-sensitive barriers by aggregating arrival notifications and distributing a release, extending NCP to implement synchronization primitives for large GPU jobs.

Adoption and capability discovery. An NCP segment targets a device advertising that instruction; other components forward it without understanding its function. So adoption can be incremental (§2): a fabric manager tracks which devices advertise which instructions, and their available capacity, so the runtime can place, bypass, or re-place accordingly.

Standardizing this interface (and the cost model it exposes) are open questions for future work.

Heterogeneous composition. We chose the stream model (in §4), but the memory-mapped model also has strengths. NVSwitch multicast-mem [21] lets a single store atomically reduce across an NVLink domain (more natural for tightly coupled GPUs). A single program could do both: `End.Reduce` memory-mapped inside an NVLink domain, stream-based across an Ethernet fabric, and endpoint reduction as fallback. This extends our vision from one portable reduction mechanism to a choice of interchangeable mechanisms. Whether a switch should implement both models, and how a program declares which to use, are open questions.

Placement synthesis and failure. NCP makes placement a field the runtime can set. Collective languages and synthesizers such as MSCCLang, SCCL, and TACCL [5, 7, 25] could target NCP for supported network operations while retaining the remaining work at endpoints. Placement would require reasoning about device capabilities, available reduction resources, and link capacities jointly.

A reducer that fails loses only accumulated state (Section 4). Re-placement costs only a few locator edits, so recovery can rebind the affected subtree's `End.Reduce` to a sibling or to the endpoints and replay the epoch, rather than restarting the collective. Automatic recovery, and bounding its cost, is ongoing work.

Synthesis and verification. Compiling collective descriptions into NCP programs requires selecting devices and allocating resources while preserving collective semantics. An open question is how to combine collective synthesis with protocol verification tools such as VeriNC [2] to establish safety and liveness: contributions must be combined correctly, results delivered to the intended endpoints, and queue reuse and credit handling must ensure progress

Isolation and shared resources. Our implementation allocates separate reduction state and credits to each queue. Performance isolation also requires scheduling and bandwidth allocation across shared switch and link resources, together with access checks restricting queues to authorized producers. These are implementation concerns: how should devices schedule concurrent collectives while preserving their semantics and respecting resource limits [26]? How should per-reducer credits interact with transport congestion control [33] and loss recovery when extending NCP to lossy fabrics, particularly when overlapping trees share links?

7 Related Work

In-network aggregation. In-network reduce is provided by a decade's worth of systems: SHARP in InfiniBand fabrics [13], SwitchML and ATP on programmable Ethernet switches [17, 24], NetReduce transparently beneath RoCE [18], and OmniReduce for sparse data [9]. They establish the mechanism NCP builds on; it is the common representation under which each becomes a backend of `End.Reduce`.

Programmable offload. sPIN runs user handlers per packet on NICs [14], and Flare builds flexible allreduce on the PsPIN RISC-V core [8]. These make the *operator* programmable, on server-side or research silicon; NCP makes the *placement* portable: the same instruction at endpoint, NIC, or switch ASIC, composed across tiers.

Programs in packets, placed dataplanes. Active networks carried arbitrary code [28, 29] (§2); Tiny Packet Programs carry a small general-purpose instruction set that switches execute at line rate, for telemetry and control [16]. NCP narrows scope to a fixed, stateful, collective-specific ISA, and makes placement a rewritable field. Flightplan splits one P4 program across heterogeneous dataplanes [27], and ClickINC compiles modular INC programs onto devices [30, 32]; both place at compile or deploy time, at program granularity, through the control plane. Such toolchains are natural frontends that could emit NCP programs, embedding rebindable, per-instruction placement in the packet itself.

Synthesis and substrate. MSCCLang supports collective descriptions, while SCCL, TACCL, and CoCoNet synthesize collective schedules for endpoints [5, 7, 15, 25]; these tools could target NCP for supported network operations (§6).; targeting NCP, they would instead choose the tree and place each instruction on a device (§6). RMT, P4, and packet transactions supply the programmable substrate [3, 4, 26]; SRv6 supplies the encoding (§3).

8 Conclusion

Collective execution inside the network remains tied to offload-specific interfaces. We argue that it should be expressed through packet-carried programs that separate operations from their placement and device-specific implementation. In NCP, endpoints encode the execution sequence, while network devices perform bounded operations using allocated local resources. SRv6 provides an existing header format for these programs, extended with collective behaviors. Our Tofino prototype implements asynchronous reduction, and our simulations show that the same operation sequence can execute at different placements. Changes to the tree structure use the same instruction set and packet format. This common execution interface provides a basis for collective runtimes to exploit network resources across different devices. Determining how to select placements, share resources, and coordinate concurrent collectives remains an open research direction.

References

[1] Araujo, J., Chow, A., Handley, M., Lewis, R., Paasch, C., Padhye, J., Papamichael, M., Steinbrecher, G., Tootoonchian, A., Yuan, L., Anantharamu, S., Dosi, A., Garg, M., Ghazi, M., Hoefler, T., Jayasinghe, D., Jose, J., Kabbani, A., Lu, G., Wang, Y., Doddapaneni, K., Garimella, M., Jain, V., Le, Y., Nagulapalli, H., Narayanan, S., Pan, R., Sabesan, R., Sivaramu, R., Sohan, R., Davis, E., Dumitrescu, D., Kalkunte, M., Mitra, B., Morandin, G., Popa, A., Raiciu, C., Spada, E., Spillane, J., Vaidya, N., Barnea, A., Burstein, I., Cohen, E., Friedman, Y., Katz, N., Moshref, M., Shpigelman, Y., Shuler, S., Shyman, S., and Sur, S. Resilient AI supercomputer networking using MRC and SRv6. arXiv preprint arXiv:2605.04333, 2026.

[2] Bai, T., Zhang, X., Wang, H., Zhang, Y., and Wu, W. VeriNC: Finding design risks of in-network computing systems. In *2026 IEEE 34th International Conference on Network Protocols (ICNP)* (2026), IEEE.

[3] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (2014), 87–95.

[4] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2013), Association for Computing Machinery, pp. 99–110.

[5] Cai, Z., Liu, Z., Maleki, S., Musuvathi, M., Mytkowicz, T., Nelson, J., and Saarikivi, O. Synthesizing Optimal Collective Algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (New York, NY, USA, 2021), Association for Computing Machinery, pp. 62–75.

[6] Cho, I., Jang, K., and Han, D. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2017), Association for Computing Machinery, pp. 239–252.

[7] Cowan, M., Maleki, S., Musuvathi, M., Saarikivi, O., and Xiong, Y. MSCCLang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (2023), ASPLOS 2023, Association for Computing Machinery, pp. 502–514.

[8] De Sensi, D., Di Girolamo, S., Ashkboos, S., Li, S., and Hoefler, T. Flare: Flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (New York, NY, USA, 2021), Association for Computing Machinery.

[9] Fei, J., Ho, C.-Y., Sahu, A. N., Canini, M., and Sapio, A. Efficient Sparse Collective Communication and Its Application to Accelerate Distributed Deep Learning. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2021), Association for Computing Machinery, pp. 676–691.

[10] Filsfils, C., Camarillo, P., Leddy, J., Voyer, D., Matsushima, S., and Li, Z. Segment Routing over IPv6 (SRv6) Network Programming. RFC 8986, Internet Engineering Task Force, Feb. 2021.

[11] Filsfils, C., Dukes, D., Previdi, S., Leddy, J., Matsushima, S., and Voyer, D. IPv6 Segment Routing Header (SRH). RFC 8754, Internet Engineering Task Force, Mar. 2020.

[12] Filsfils, C., Previdi, S., Ginsberg, L., Decraene, B., Litkowski, S., and Shakir, R. Segment Routing Architecture. RFC 8402, Internet Engineering Task Force, July 2018.

[13] Graham, R. L., Levi, L., Burredy, D., Bloch, G., Shainer, G., Cho, D., Elias, G., Klein, D., Ladd, J., Maor, O., Marelli, A., Petrov, V., Romlet, E., Qin, Y., and Zemah, I. Scalable Hierarchical Aggregation and Reduction Protocol (SHARP): Streaming-Aggregation Hardware Design and Evaluation. In *High Performance Computing: 35th International Conference, ISC High Performance 2020* (2020), vol. 12151 of *LNCS*, Springer, pp. 41–59.

[14] Hoefler, T., Di Girolamo, S., Taranov, K., Grant, R. E., and Brightwell, R. sPIN: High-Performance Streaming Processing In the Network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (New York, NY, USA, 2017), Association for Computing Machinery.

[15] Jangda, A., Huang, J., Liu, G., Nodehi Sabet, A. H., Maleki, S., Miao, Y., Musuvathi, M., Mytkowicz, T., and Saarikivi, O. Breaking the Computation and Communication Abstraction Barrier in Distributed Machine Learning Workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (New York, NY, USA, 2022), Association for Computing Machinery, pp. 402–416.

[16] Jeyakumar, V., Alizadeh, M., Kim, C., and Mazières, D. Tiny Packet Programs for Low-Latency Network Control and Monitoring. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks (HotNets-XII)* (New York, NY, USA, 2013), Association for Computing Machinery.

[17] Lao, C., Le, Y., Mahajan, K., Chen, Y., Wu, W., Akella, A., and Swift, M. M. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2021), USENIX Association, pp. 741–761.

[18] Liu, S., Wang, Q., Zhang, J., Wu, W., Lin, Q., Liu, Y., Xu, M., Canini, M., Cheung, R. C. C., and He, J. In-Network Aggregation with Transport Transparency for Distributed Training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS)* (New York, NY, USA, 2023), Association for Computing Machinery, pp. 376–391.

[19] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Standard specification, 2023.

[20] Montazeri, B., Li, Y., Alizadeh, M., and Ousterhout, J. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2018), Association for Computing Machinery, pp. 221–235.

[21] NVIDIA. NVIDIA NVSwitch and NVLink: Multicast and Reduction Operations. Technical documentation, 2024.

[22] NVIDIA. NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) Documentation. Online documentation, 2025. Accessed 2026-06-05.

[23] NVIDIA. NVIDIA Collective Communications Library: Collective Operations. Online documentation, 2026. Accessed 2026-06-05.

[24] Sapio, A., Canini, M., Ho, C.-Y., Nelson, J., Kalnis, P., Kim, C., Krishnamurthy, A., Moshref, M., Ports, D. R. K., and Richtárik, P. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2021), USENIX Association, pp. 785–808.

[25] Shah, A., Chidambaram, V., Cowan, M., Maleki, S., Musuvathi, M., Mytkowicz, T., Nelson, J., Saarikivi, O., and Singh, R. TACCL: Guiding Collective Algorithm Synthesis Using Communication Sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2023), USENIX Association, pp. 593–612.

[26] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S. Packet Transactions. In *Proceedings of the ACM SIGCOMM Conference* (New York, NY, USA, 2016), Association for Computing Machinery, pp. 15–28.

[27] Sultana, N., Sonchack, J., Giesen, H., Pedisich, I., Han, Z., Shyamkumar, N., Burad, S., DeHon, A., and Loo, B. T. Flightplan: Dataplane

Disaggregation and Placement for P4 Programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Apr. 2021), USENIX Association, pp. 571–592.

[28] Tennenhouse, D. L., and Wetherall, D. J. Towards an Active Network Architecture. *ACM SIGCOMM Computer Communication Review 26*, 2 (1996), 5–17.

[29] Wetherall, D. J., Guttag, J. V., and Tennenhouse, D. L. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of the IEEE OPENARCH* (1998), pp. 117–129.

[30] Xu, W., Zhang, Z., Feng, Y., Song, H., Chen, Z., Wu, W., Liu, G., Zhang, Y., Liu, S., Tian, Z., and Liu, B. ClickINC: In-network computing as a service in heterogeneous programmable data-center networks. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA, 2023), Association for Computing Machinery, pp. 798–815.

[31] Yuan, Y., Nie, J., Bai, T., Zhou, R., Cao, S., Fan, X., Xu, Y., Chen, J., Zhao, C., Zhang, N., Fang, S., Chen, J., Chen, Y., Sun, J., Wang, Z., Xu, X., Zhang, Y., Liu, Y., Yang, X., Lin, J., Hu, X., Li, Y., Jiang, C., Xiao, L., Zhang, W., Wang, J., Cheng, W., Lan, Y., Dong, J., Fu, B., and Wu, W. EPIC: Abstraction and polymorphism of in-network collectives on Ethernet. In *Proceedings of the ACM SIGCOMM 2026 Conference* (2026), Association for Computing Machinery.

[32] Zheng, C., Tang, H., Zang, M., Hong, X., Feng, A., Tassiulas, L., and Zilberman, N. DINC: Toward distributed in-network computing. *Proceedings of the ACM on Networking 1*, CoNEXT3 (Nov. 2023).

[33] Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M. H., and Zhang, M. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM SIGCOMM Conference* (2015), pp. 523–536.