Scalable Asynchronous Aggregation for Many-to-Many Transfers

Naveen Namashivayam
University of Minnesota
Hewlett Packard Enterprise
USA

Nathan Wichmann
Hewlett Packard Enterprise
USA

Pen-Chung Yew
University of Minnesota
USA

Abstract

Many-to-many aggregation—where multiple senders concurrently deposit messages into a shared remote buffer—is a critical pattern in data-driven workloads such as distributed sorting and sparse analytics. Today's implementations of this *append* pattern require a remote atomic round-trip to coordinate placement at every destination, creating contention and latency that worsen with scale.

Data placement is receiver-managed state: if the destination NIC can allocate offsets locally while streaming incoming payloads, the atomic round-trip and its associated hot-spot are eliminated. This paper presents a formal append framework with managed buffer regions, receiver-side placement, and byte-level completion tracking, together with a minimal hardware contract—receiver-side offset management and progress accounting—that enables full NIC offload. This contract is already met by HPE Slingshot and is mandated by the emerging Ultra Ethernet specification. We implement the framework in OpenSHMEM via Libfabric and evaluate it using many-to-many microbenchmarks and application kernels including bucket sorting of 32 TiB across 512 nodes (32,768 processes). The offloaded path achieves up to 90× speedup over MPI and sustains 2× over the software fallback at 512 nodes.

CCS Concepts

• **Computing methodologies** → **Parallel computing methodologies**; • **Networks** → *Network adapters*; • **Computer systems organization** → Interconnection architectures; • **Theory of computation** → Sorting and searching.

Keywords

PGAS, OpenSHMEM, one-sided communication, RDMA, NIC offload, many-to-many aggregation, append, distributed sorting, collective communication, MoE inference

1 Introduction

Many-to-many communication, where multiple source processes aggregate messages into shared destination buffers (Figure 1), is a fundamental building block of workloads spanning HPC analytics and large-scale AI. Distributed bucket sort [4, 29] is the canonical HPC example [19, 20, 22]. The same pattern now dominates Mixture-of-Experts (MoE) inference: models such as DeepSeek-V3 [14] and Mixtral [25] route each token to a data-dependent subset of experts, producing an irregular AlltoAll across processes [1, 23, 28, 33]. Every process deposits variable-sized token batches into per-expert buffers on remote nodes.

At scale, this communication is a first-order bottleneck, conventionally handled by a synchronous size exchange before transfer. That exchange often cannot be avoided: bucket sort's worst case—all keys in one bucket—forbids any smaller buffer, keeping it on the critical path. Where it can be skipped, as in MoE dispatch, doing so wastes memory by sizing every receive buffer for the worst case. Addressing it requires fine-grained, asynchronous data movement across dynamically changing sets of processes [3, 5].

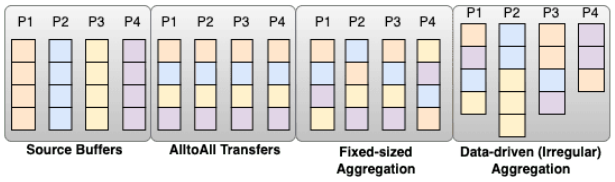


Figure 1: All-to-all, fixed-size, and data-driven many-to-many aggregation patterns across four processes.

The *append* pattern, supported across Partitioned Global Address Space (PGAS) languages and Remote Memory Access (RMA) libraries [7, 10–12], allows multiple sources to write concurrently to a shared destination buffer using one-sided operations. As illustrated in Figure 2, the sender first performs an atomic fetch-and-add (*FADD*) to reserve an offset in a pre-posted destination buffer (the *Table*) on the target, then issues a *Put* to write data at that offset. However, this round-trip atomic creates a fundamental bottleneck: contention at the shared offset pointer and network latency that worsen with scale [13, 18, 24].

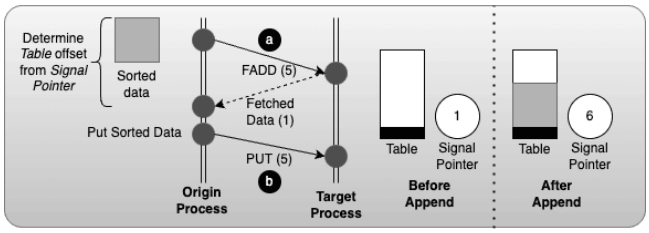


Figure 2: Append pattern: (a) FADD allocates offset in remote Table, (b) Put writes payload at computed offset.

The key insight observed in this work is that placement is receiver-managed state: the receiver (or its NIC) ultimately owns the offset allocation policy for the destination *Table*. If the destination NIC can manage offsets locally while streaming incoming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PACT '26, October 2026, TBD
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/26/10
https://doi.org/10.1145/nnnnnnn.nnnnnnn

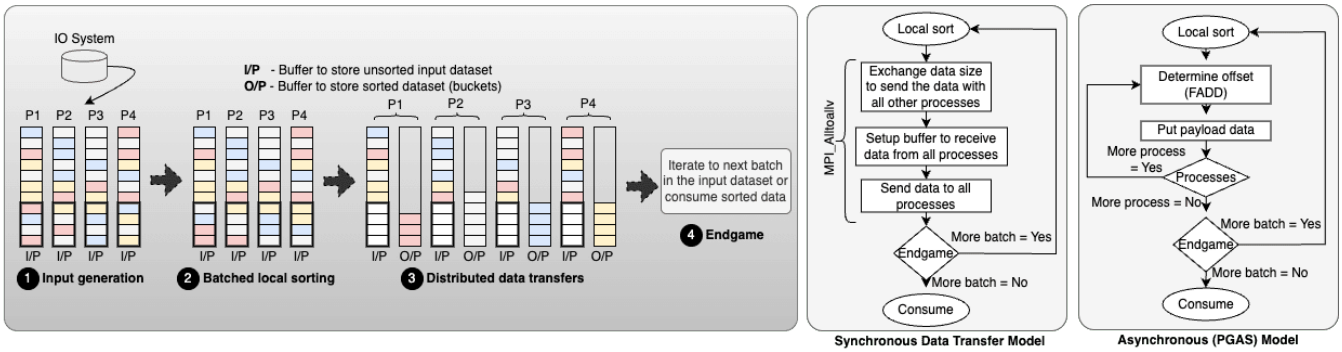


Figure 3: Various steps in distributed bucket sorting algorithm with synchronous and asynchronous data transfer model.

payloads into a pre-posted *Table*, we can eliminate the network round-trip for offset computation and avoid the atomic hot-spot at the target. This paper therefore presents an append communication *framework and primitive* designed to be offloaded to modern network hardware.

1.1 Contributions

The primary contributions of this work are:

(1) A formal *append framework and primitive* for many-to-many aggregation, with managed buffer regions, receiver-managed placement semantics, and byte-level completion tracking as a reusable programming-model abstraction independent of any single network technology;
(2) A minimal, portable *hardware contract*—receiver-side offset management and receive-side progress accounting—that enables full NIC offload of the append primitive. We show that this contract is already met by HPE Slingshot [13] and is mandated by the emerging Ultra Ethernet specification [31], demonstrating that the design is not tied to a single vendor or network architecture;
(3) A reference implementation in OpenSHMEM, mapped through Libfabric to concrete NIC primitives, alongside a software fallback path (FADD+Put) that preserves portability on networks without hardware support; and
(4) A systematic evaluation that quantifies the performance and scalability of the offloaded path—including many-to-many microbenchmarks and distributed bucket sorting of 32 TiB across 512 nodes (32,768 processes)—and characterizes the regimes where hardware offload is most beneficial versus where the software fallback remains competitive.

2 Background

We trace the append pattern through distributed bucket sorting. Figure 3 illustrates the four phases of distributed bucket sorting using four processes (*P1-P4*) to move data from locally sorted input buffers (*I/P*) to output buffers (*O/P*): 1 input generation, 2 batched local sorting, 3 distributed data transfers, and 4 endgame. Input generation involves reading or creating the dataset; each process sorts its local data, then distributes sorted data into buckets on other processes, repeating as needed. The endgame stage resolves bucket overflows and finalizes the output. This work focuses on optimizing the distributed data transfer stage, where the append pattern is central.

2.1 Message Passing-style Synchronous Model

In message-passing implementations, each send must match a receive, and processes serve as both senders and receivers. Due to variable data sizes per bucket, a synchronous AlltoAll operation exchanges size information so processes can allocate appropriate receive buffers before point-to-point transfers. Apart from being synchronous, this approach, similar to *MPI_Alltoallv* [26], can introduce high overhead for small messages and limits the ability to overlap communication with computation [30].

2.2 PGAS-style Asynchronous Model

In PGAS-style implementations, each process allocates a receive buffer after input generation and independently calculates write offsets for asynchronous remote writes (*Put*) in the distributed transfer phase. Specifically, for small message payloads, this one-sided, append pattern enables concurrent data movement and reduces synchronization overhead compared to message passing.

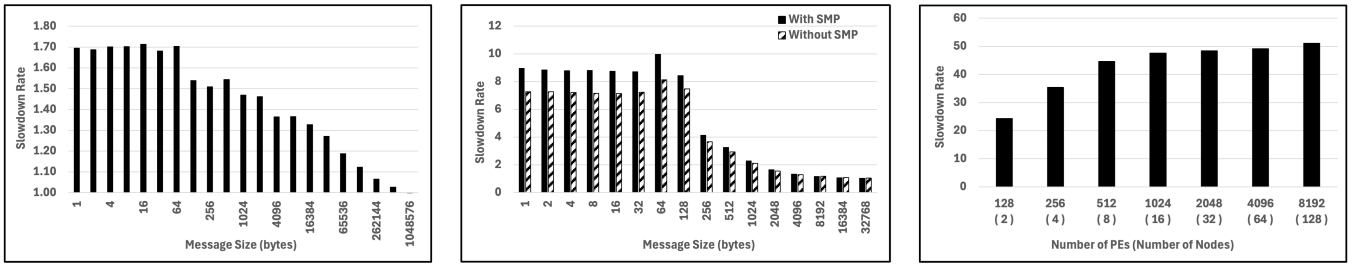
While the append pattern offers benefits over the synchronous model described in Section 2.1, it is not without limitations. The offset calculation and buffer contention can impact scalability due to network round-trips and congestion (see Figure 4a-4c). Section 3 analyzes these limitations in detail.

3 Motivation

This section examines the limitations of the traditional implementation of the append communication pattern, which relies on one-sided atomic *FADD* and *Put* operations, as illustrated in Figure 2. The analysis is conducted on a test system powered by compute nodes based on AMD EPYC7763 processors (codenamed AMD Milan) and HPE Slingshot interconnect [13].

3.1 Cost of Network Roundtrip

Calculating an offset within a buffer on the target process—prior to issuing a *Put* operation as seen in the append pattern—introduces additional performance overhead when compared to direct remote writes to a predetermined memory location. This offset computation



(a) Network Roundtrip Cost. **(b) Message Size Impact.** **(c) *FADD* Congestion Cost.**

Figure 4: Illustrating the limitations of implementing the append communication pattern using *FADD* and *Put*.

necessitates a network roundtrip involving an atomic operation initiated by the source process on the target.

Figure 4a illustrates the latency overhead associated with this operation. A modified OSU OpenSHMEM latency benchmark [32] is used to assess the impact. Specifically, Figure 4a compares the latency of a *FADD* followed by a *Put* operation against the latency of a basic *Put* operation, which transfers data from the origin to fixed addresses on the target. The *FADD* operation is employed to dynamically compute the offset for each data transfer.

As shown, the overhead of the *FADD* is significant for small message sizes, leading to a 1.7×–1.4× slowdown in overall data transfer compared to direct transfers, particularly for payload sizes between 8 B and 8 KiB. However, as the payload size increases, the relative impact of the additional *FADD* operation diminishes.

3.2 Impact of Message Size

Beyond single-operation latency, the append pattern also degrades network bandwidth when many *FADD* operations compete with data transfers. Figure 4b presents results from a modified OSU OpenSHMEM AlltoAll collective communication benchmark, executed on a system with 8 compute nodes, each hosting 128 processes. The benchmark is implemented using two distinct algorithms.

The baseline algorithm utilizes simple *Put* operations to directly transfer data to predetermined memory locations in the target processes. In contrast, the append-based variant employs a combined *FADD* + *Put* communication pattern to realize the AlltoAll exchange. Figure 4b highlights the cost associated with transmitting multiple offset-determining operations within the AlltoAll exchange. Across message sizes ranging from 8 B to 16 KiB, the use of *FADD* for dynamic offset computation introduces substantial performance overhead (1.5×–10×) and results in significantly increased memory bandwidth consumption.

This behavior is consistent regardless of the use of shared memory (SMP) optimization techniques typically employed by implementations of parallel programming models. The SMP optimization allows the implementation to dynamically select the most efficient communication path, either inter-node or intra-node, for message delivery while preserving the semantics of the programming model. However, the additional bandwidth consumed by offset-determining operations reduces the resources available for payload transfers, degrading overall communication efficiency.

3.3 Cost of Memory Buffer Contention

A third cost arises when multiple origin processes contend for the same target buffer: their *FADD* operations must be serialized, increasing latency.

Figure 4c illustrates the overhead of offset determination when varying numbers of source processes concurrently issue *FADD* operations to a single target process. This contention analysis uses a modified OSU OpenSHMEM atomic communication benchmark. As the number of source processes increases (scale-out), the latency of offset computation rises significantly, demonstrating performance degradation caused by memory buffer contention in the append pattern. The results in Figure 4c are obtained by running the microbenchmark across 128 to 8192 processes distributed across 2 to 128 compute nodes.

Overall, Sections 3.1 through 3.3 highlight the limitations of employing *FADD*+*Put* for implementing append. Although this approach outperforms message-passing for small payloads by eliminating synchronization overhead, it still falls short of optimal network utilization due to round-trip latency and contention. Section 4 describes the proposed framework that addresses these challenges.

4 Append Proposal

While the core primitives for implementing append pattern—*FADD* and *Put*—enable atomic offset calculation and data transfer, simply combining them does not yield a fully efficient or robust append mechanism. A comprehensive framework is required to coordinate buffer management, data movement, and synchronization across distributed processes.

This section presents a dedicated communication framework for the append pattern. The framework formalizes semantics for buffer management, distinguishes operational modes for data movement and consumption, and defines protocols for orderly termination. It also addresses coordination challenges during the termination (endgame) phase to ensure correctness and resource cleanup.

4.1 Framework Overview

The proposed framework organizes append operations into four categories: memory management, data movement, data consumption, and termination. At its core is the concept of an *append session*, spanning from buffer creation (*Create*) to destruction (*Destroy*). Upon session creation, a partitioned, globally addressable memory

table is established and shared by all processes as the aggregation point for incoming data.

Data movement (*Transfers*)—principally, the append operation using *FADD+Put*—occurs during the session. Status operations allow processes to query the buffer's state for message consumption. The interval between transfers and status updates defines an *append epoch*; a session may comprise multiple epochs. Upon session termination, all outstanding data transfers are finalized and the buffer is released. Figure 5 depicts the operation categories and execution model. Sections 4.3–4.5 describe each category in detail.

4.2 Semantics and Guarantees

This section summarizes the semantics of the append primitive provided by the framework. We use *primitive* to mean a semantic communication operation with explicitly defined guarantees, independent of how it is realized (such as in software using conventional one-sided operations or through NIC offload).

- **Aggregation and placement.** Each process exposes a local Table that serves as a byte-addressable aggregation region for incoming appends. Append operations place payload bytes contiguously into the destination Table. The framework does not impose a global ordering across senders.
- **Message boundaries.** The framework aggregates bytes and does not track per-message boundaries. Applications requiring delineation can use fixed-size records or embed headers.
- **Sender-side completion.** Append transfers are non-blocking. A sender ensures local completion via the underlying model's ordering primitive (e.g., *shmem_quiet* or *MPI_Win_flush*). Epoch boundaries typically require collective synchronization for a consistent global view.
- **Receiver-side querying.** *shmem_append_query* returns a cumulative byte count for the local Table, which may grow while concurrent transfers are in flight. To consume a stable prefix for an epoch, applications close the epoch with collective synchronization, then query the byte count.
- **Table sizing and overflow.** The Table must be provisioned with sufficient space for each epoch. If insufficient space is provided, delivered bytes may be less than requested. Dynamic resizing is possible but not included in this work; Tables are sized to avoid overflow in our evaluation.

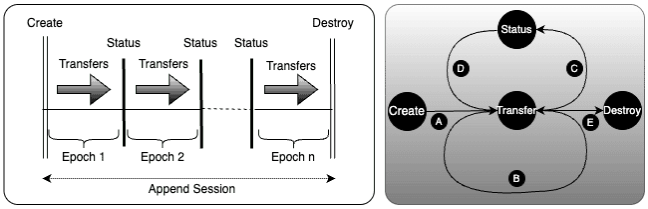


Figure 5: Append session lifecycle: Create → Transfer epochs → Status queries → Destroy.

4.3 Table Creation

An append session A starts by registering a user-supplied memory buffer (the Table) for RDMA-based, one-sided communication. Table size can differ across participants. Session creation is a collective operation, ensuring all processes have access to a buffer for aggregating transferred messages. Once created, each process can independently initiate data movement operations.

4.4 Data Movement

The framework supports a non-blocking append primitive. In a conventional one-sided implementation, append is realized using *FADD* (for offset allocation) followed by *Put* (for data transfer). The primitive is asynchronous; the source buffer must not be reused or modified until completion, which is managed through memory ordering primitives provided by the underlying programming model, such as *shmem_quiet* (OpenSHMEM) or *MPI_Win_flush* (MPI-RMA), that supports the proposed framework.

By integrating offset determination and payload delivery into a single operation, the framework eliminates the network round-trip discussed in Section 3, improving communication efficiency.

Append operations can be B repeated within an epoch, after which received buffers are available for consumption via status queries (C and D).

4.5 Table Status Query

Multiple processes may asynchronously update the Table. To determine the total data received, a local query operation is provided. This operation reports an accumulated byte count, but does not enforce ordering; concurrent updates may still be in progress. Applications must use collective synchronization (such as *shmem_sync*) for consistent views before querying.

4.6 Transfer and Session Completions

Append operations complete when previously issued transfers are locally completed by the origin (e.g., using *shmem_quiet*) and the application closes the epoch/session as needed (e.g., via collective synchronization). At session end, the Table is E detached and returned for non-append operations.

4.7 Example OpenSHMEM Support

TABLE 1 summarizes the OpenSHMEM API for the proposed append framework. Figure 6 compares two implementations of an alltoall-like algorithm: Figure 6a, a baseline approach using standard OpenSHMEM operations and Figure 6b, the proposed framework that exposes the append pattern as a single, higher-level operation.

In the baseline (Figure 6a), each process computes offsets via *shmem_atomic_fetch_add* and issues *shmem_put_nbi* per target, requiring explicit coordination every iteration. The proposed framework (Figure 6b) encapsulates offset allocation and data movement into *shmem_append_nbi*, with session management via *create*/*destroy* and progress via *query*.

This abstraction isolates the append operation, making it amenable to network offload by consolidating offset computation and data transfer into a single semantic operation. Section 5 describes a network-offloaded implementation on HPE Slingshot.

5 Implementation

This section details the acceleration of the append communication scheme using the proposed framework on HPE Slingshot [13], a

Phase	Operation Syntax	Collective	Blocking
Create	void shmem_append_create(void *table, size_t tsize, shmem_append_t *hndl)	✓	✓
Transfer	void shmem_append_nbi(shmem_append_t hndl, const void *src, size_t size, int pe)	×	×
Status	uint64_t shmem_append_query(shmem_append_t hndl)	×	✓
Destroy	void shmem_append_destroy(shmem_append_t hndl)	✓	✓

Table 1: Append framework exposed as OpenSHMEM operations.

```
for (n = 0 -> niter) {
  for (tpe = 0 -> npes) {
    offset = shmem_atomic_fetch_add(sig, size, tpe);
    shmem_put_nbi(&table[offset], src, size, tpe);
  }
  shmem_barrier_all();
}
```

(a) Baseline program with Append pattern.

```
shmem_append_t hndl;
shmem_append_create(&table, tsize, &hndl);
for (n = 0 -> niter) {
  for (tpe = 0 -> npes) {
    shmem_append_nbi(hndl, src, size, tpe);
  }
  shmem_barrier_all();
  long esize = n * size * npes;
  assert(shmem_append_query(hndl) <= esize);
}
shmem_append_destroy(hndl);
```

(b) Program using proposed framework.

Figure 6: Code comparison: manual *vs.* proposed framework.

Portals-class interconnect. Section 5.1 discusses the network hardware primitives underpinning the framework, while Section 5.2 describes the middleware network library interface that connects these hardware features to the OpenSHMEM runtime for efficient append implementation.

5.1 Hardware Foundations

Although the proposed framework is adaptable to different high-speed networks, our prototype targets the HPE Slingshot NIC (Cassini), which exposes Portals [6]-style primitives (e.g., *PtlPut*, *PtlGet*, and *PtlAtomic*) and supports receive-side placement and progress tracking features needed for offloaded append. Figure 7 illustrates the minimal components required for implementing the proposed append framework.

More generally, the offloaded append design relies on a small hardware contract that is independent of any single vendor implementation. Specifically, it assumes (1) *receiver-managed placement*, where the destination NIC can autonomously place each incoming payload into a pre-posted region using a locally managed offset (LMO or an equivalent mechanism), and (2) *receive-side progress accounting*, where the receiver can query a monotonically increasing byte count (or equivalent completion metric) for the posted region without requiring per-message acknowledgments. The emerging Ultra Ethernet specification [31], whose transport layer draws on Portals concepts, mandates both capabilities, indicating that the proposed framework is portable beyond Slingshot to future Ultra Ethernet fabrics.

On interconnects lacking these features, the framework falls back to software-managed append using conventional one-sided operations (FADD+Put); the SHMEM baseline in Section 6 quantifies this fallback path and demonstrates that the hardware-offloaded implementation delivers 4–14× improvement over the portable software realization.

5.1.1 Memory Descriptors. Portals requires explicit memory registration for remote operations: Memory Descriptors (MDs) define source buffers and Match Entries (MEs) define target buffers with matching criteria. In the append framework, each process creates an ME during session initialization to expose its Table for incoming data; origin processes then issue *PtlPut* operations against their local MD to deliver append payloads to the target's ME.

5.1.2 Locally Managed Offset (LMO). The *receiver-managed placement* requirement identified in Section 5.1 is realized on Slingshot through the Portals locally managed offset (LMO) mechanism—a capability available in Portals-class networks since the mid-1990s [8, 9]. Without LMO, the origin must specify the remote offset for each *PtlPut*, requiring an FADD round-trip to avoid data races.

With LMO enabled, the target NIC autonomously manages placement: upon receiving a message, it identifies the appropriate ME via portal table index and match bits, appends the data at the next available offset, and increments an internal counter. In our framework, each *shmem_append_nbi* maps to a single *PtlPut* with LMO enabled, so the NIC handles offset allocation that would otherwise require a separate FADD round-trip. The contribution is not LMO itself, but the systematic identification of this mechanism—together with byte counters (Section 5.1.3)—as the minimal contract needed to offload the complete append lifecycle.

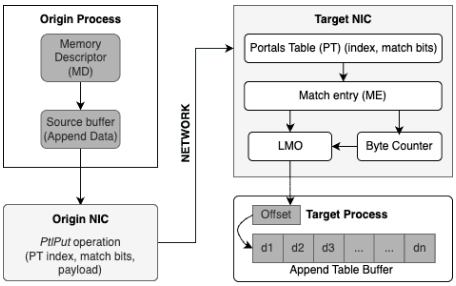


Figure 7: Portals components used for offloaded append: Memory Descriptor (MD), Match Entry (ME), and Locally Managed Offset (LMO).

5.1.3 Hardware Managed Byte Counters. The Slingshot NIC provides hardware-managed byte counters per ME to track cumulative

	NNODES	NPPN	NSOCKETS	NIC	NNICS	CPU	GPU	OS
System:1	16	128	2	HPE Slingshot 200	2	AMD EPYC 7763		SLES 15 SP5
System:2	1002	128	2	HPE Slingshot 200	1	AMD EPYC 7763		SLES 15 SP6
System:3	16	64	4	HPE Slingshot 200	4	AMD Trento	AMD MI250X	SLES 15 SP6

Table 2: Append test platforms overview showing NIC count, process density, and memory type.

data delivered to Portals tables. These counters enable efficient, in-network status queries for appended data, in contrast to traditional counters that only monitor operation counts.

Collectively, these hardware features—Match Entries, Locally Managed Offsets, and byte counters—enable the Slingshot NIC to offload core aspects of the proposed append framework. Section 5.2 describes the necessary Libfabric API support to expose these capabilities within OpenSHMEM.

5.2 Middleware Bridging Layer

Slingshot NIC capabilities are exposed to user-level libraries through the CXI Libfabric [2] provider. Hardware features such as MEs, LMOs, and byte counters must be mapped to appropriate middleware APIs to ensure high-level libraries, including OpenSHMEM and MPI, can utilize them. This section outlines the Libfabric APIs used to interface with these hardware features.

5.2.1 Multireceive. Libfabric provides a multi-receive mode, *FI_MULTI_RECV*, in which one large receive buffer accepts many incoming messages without re-posting, matching ME/LMO semantics. During session creation, each process posts an *fi_recvmsg* with *FI_MULTI_RECV* to register its Table. Later *shmem_append_nbi* calls issue *fi_sendmsg* to match the posted receives, and the NIC places data via LMO.

5.2.2 Byte Counters. Each *fi_recvmsg* is associated with a Libfabric endpoint mapped to a Portals table, with a hardware byte counter (*FI_CNT_BYTE*) tracking total bytes received. The design restricts each endpoint to a single outstanding *fi_recvmsg*, ensuring a one-to-one EP-to-ME mapping so the counter accurately reflects data received.

By utilizing standard Libfabric interfaces, the implementation exposes Slingshot NIC capabilities for a fully offloaded append framework. Section 6 presents the performance analysis.

6 Performance Analysis

This section evaluates the proposed append framework on HPE Slingshot, focusing on three areas: (1) process scaling per NIC, (2) memory type effects, and (3) overall scalability. Microbenchmarks simulating many-to-many communication are used throughout. Section 7 further analyzes the framework's use in application kernels.

6.1 Methodology and Baselines

This section summarizes the methodology used for the microbenchmarks and clarifies the baselines.

Metric. *Effective throughput* is reported as total payload bytes exchanged divided by the elapsed time for an iteration (or the total time across iterations for the non-blocking variant), and results are also discussed relative to the per-node injection bandwidth of each system.

Process placement. Unless otherwise noted, experiments are executed with one process per CPU core (CPU runs) or one process per GPU sub-device (GPU runs). For multi-NIC nodes, processes are distributed evenly across NICs.

Repetitions and Baselines. Each configuration is run for 10,000 iterations; the mean is reported. Three implementations of the many-to-many microbenchmark are compared:

(1) **MPI (synchronous exchange)**: uses *MPI_Alltoallv* to exchange variable-sized payloads in a single collective operation;
(2) **SHMEM (manual append)**: OpenSHMEM implementation that realizes append using *FADD* for offset allocation followed by *shmem_put_nbi* for payload placement; and
(3) **Append (offloaded)**: the proposed *shmem_append_nbi* primitive, with receiver-managed placement and byte-based status queries.

6.2 Test Bed Overview

Table 2 summarizes the three system configurations. *System:1* has dual-socket AMD EPYC 7763 nodes with two Slingshot NICs (50 GiB/s per node, up to 128 processes). *System:2* uses the same processor but a single NIC (25 GiB/s), isolating the impact of NIC count. *System:3* is a heterogeneous architecture with four Slingshot NICs and AMD MI250X GPUs per node (100 GiB/s), enabling CPU *vs.* GPU memory comparisons.

6.3 Test Kernel Overview

The primary analysis uses an all-to-all microbenchmark kernel (see Figure 6) in two variants: *blocking* (global barrier after each iteration) and *non-blocking* (local completion per iteration, global synchronization deferred). Sufficient Table memory is provisioned for all incoming messages. The three baselines from Section 6.1 are compared across a range of message sizes with configurable CPU- or GPU-attached buffers.

6.4 Impact of Process Density per NIC

This analysis uses *System:1* across 16 nodes with varying process densities: (1) 128 processes per node (2,048 total), (2) 16 processes per node (one per core complex die, CCD; 256 total), and (3) 8 processes per node (one per NUMA domain; 128 total). Performance results for these configurations are shown in Figure 8.

As discussed in Section 3, the overhead of FADD with payload transfer is most significant for small to medium payloads (less than 32 KiB). For larger payloads, the FADD cost is amortized. Accordingly, Figure 8 reports results for small payload sizes.

Overall, Append achieves 20% to 5× speedup over MPI and manual OpenSHMEM. At 128 processes/node (Figure 8a), Append delivers 50% speedup over MPI and 41% over manual SHMEM. At lower densities (Figure 8b,c), the MPI gap narrows as collective overhead becomes small, but Append still achieves 4–5× over manual

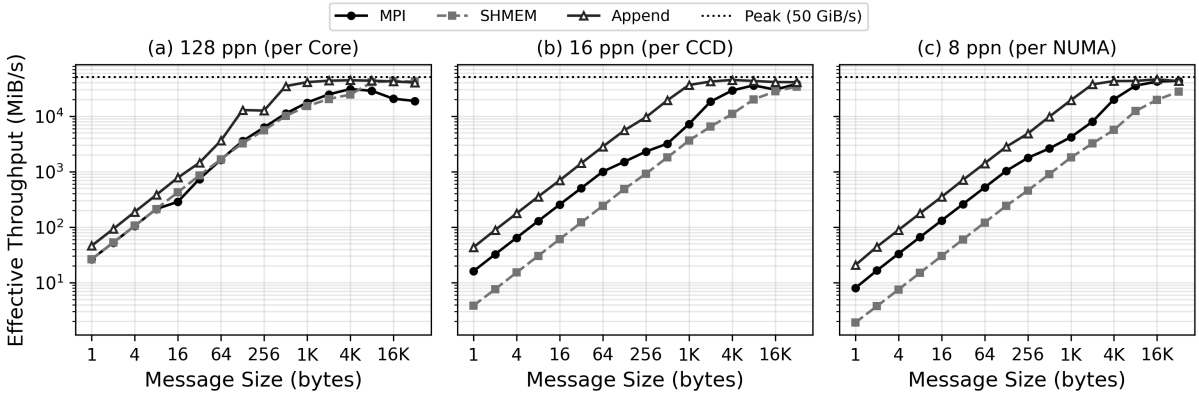


Figure 8: Many-to-many throughput *vs.* message size at varying process densities (*System:1*, 16 nodes).

SHMEM for small to medium payloads. The SHMEM gap is larger at low density because contention is reduced and FADD round-trip latency becomes the dominant cost; eliminating that round-trip via receiver-managed placement yields the biggest relative win.

Key observations: (1) gains are greatest for payloads <32 KiB and high process density per NIC, where offset coordination costs dominate transfer time; (2) at small scales (128–256 processes), Append shows limited benefit over MPI because *MPI_Alltoallv*'s metadata exchange is cheap at low process counts—this crossover is expected and is not the target regime for the append primitive; (3) advantages taper for large messages as both approaches become bandwidth-limited.

6.4.1 Speedup Attribution. The end-to-end speedup of offloaded Append over manual SHMEM bundles two distinct effects: removal of *counter contention* and removal of the *FADD round-trip*. To attribute the share of each, we add a contention-free control to the many-to-many microbenchmark—a FADD+Put variant that shards the offset counter across sources (one counter per source process, each owning a disjoint *Table* sub-region), eliminating atomic serialization while retaining the round-trip. This yields a three-rung ladder:

(1) **SHMEM (shared counter):** FADD to a single shared counter, then Put—both round-trip and contention;
(2) **SHMEM (sharded counters):** FADD to a per-source counter, then Put—round-trip only, contention removed;
(3) **Append (offloaded):** receiver-managed placement—neither cost.

The gap between (1) and (2) isolates the *contention share*; the gap between (2) and (3) isolates the *round-trip share*, realized jointly by receiver-managed placement and NIC offload—the two are not independently separable, since placement is realized by the offload. Figure 9 reports the attribution across process densities.

Figure 9 reports the decomposition at 512 B across three densities on 16 nodes (128–2,048 PEs), matching the process placements of Section 6.4. We run the ablation at a single, fixed message size, and 512 B is a deliberate choice. The round-trip we isolate is a fixed per-message cost, so its share of the time saved climbs as messages shrink: a much smaller payload would inflate append's apparent

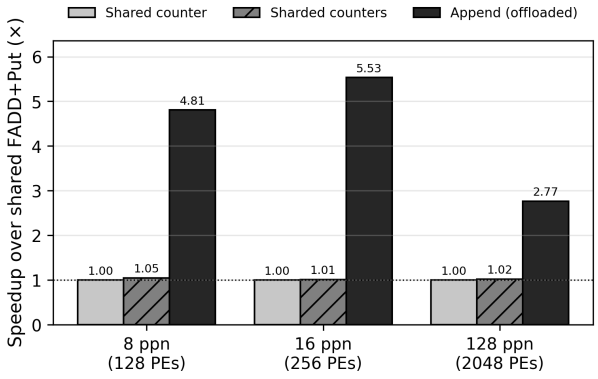


Figure 9: Speedup attribution at 512 B (16 nodes, 128–2,048 PEs), normalized to the shared-counter FADD+Put baseline. The shared→sharded gap isolates counter contention (2–6%); the sharded→Append gap isolates the FADD round-trip (94–98%).

speedup without changing what we learn, drive the already-small contention term into measurement noise, and—below about 64 B—leave all three variants bound by message rate rather than the round-trip. A much larger payload has the opposite problem, since past 32 KiB both variants become bandwidth-bound and append no longer helps, so there is no speedup left to attribute. At 512 B the round-trip is dominant yet both effects stay resolvable, and the size matches the operating point of the density study in Section 6.4. The round-trip is the dominant cost: removing it (sharded→Append) accounts for 94–98% of the time saved—a 2.8–5.5× speedup—whereas removing counter contention alone (shared→sharded) contributes only 2–6% at every density, with no clear upward trend. This is expected on Slingshot: its NIC-offloaded atomics keep shared-counter serialization cheap, so the decisive cost is the round-trip *dependency* itself—the sender stall plus the zero-payload FADD consuming injection bandwidth—which only offloaded Append removes. Counter

contention, though real (Section 3.3), is thus a secondary contributor at this scale; we expect its share to grow at the extreme process counts of Section 6.6, which we leave to future work.

6.5 Impact of Append Memory Type

Using *System:3* (8 processes per node, one per GPU sub-device, four NICs per node), Append achieves 9–13× improvement over manual SHMEM for small payloads (Figure 10). This configuration exercises the critical case of one process per GPU; since CPU-attached runs at the same density show equivalent behavior, additional GPU configurations are omitted for brevity. GPU-attached Table memory does not degrade offloaded Append performance: because the NIC manages placement autonomously, the host CPU never touches the Table during transfers, so the memory backing type is irrelevant to the offload path.

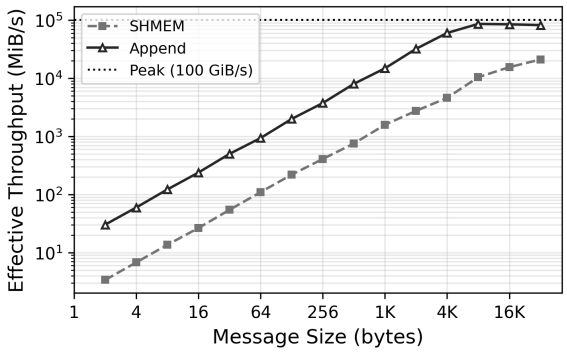


Figure 10: Throughput comparison with GPU-attached Tables (*System:3*, 16 nodes, 8 processes/node).

6.6 Scalability Analysis

Figure 11 presents the scalability results for the Append framework, evaluated on *System:2* with node counts ranging from 32 to 900 and 64 processes per node (2,048 to 57,600 processes). Although *System:2* comprises 1,002 nodes, 900 was the maximum available allocation due to scheduling constraints and nodes offline for maintenance. The analysis compares the accelerated Append implementation against the manual OpenSHMEM-based approach for the non-blocking microbenchmark described in Section 6.3.

For small to medium payloads (<32 KiB), Append consistently outperforms manual OpenSHMEM, exhibiting speedups from 2.5× at 32 nodes to 30% at 900 nodes. At scale, both sources of overhead compound: more processes increase the FADD round-trip stall (due to serialization at target atomics) and the fraction of link bandwidth consumed by zero-payload control traffic. This trend echoes results at smaller scales (Section 6.4), highlighting the framework's efficiency in exploiting network bandwidth at extreme scale.

Comparable behavior is observed for both blocking and non-blocking variants, though only non-blocking results are shown for brevity. As the system scales, the relative speedup of Append diminishes because both implementations converge toward the system's all-to-all ceiling on *System:2*, which is constrained by the tapered dragonfly topology and is lower than the per-node injection peak. At 900 nodes, both approaches achieve over 50% of peak bandwidth, with Append reaching 56% and a direct all-to-all test (without append) achieving 60%.

In other words, the gap narrows at large scale because the job is operating near the maximum achievable all-to-all throughput on this particular interconnect configuration; improvements primarily change *how quickly* that ceiling is reached, rather than increasing the ceiling itself. Nonetheless, Append remains effective and achieves near-optimal utilization even at the largest tested scales.

In summary, the accelerated Append framework delivers substantial performance benefits at scale, particularly for small to medium messages. Section 6.6.1 shows the results of the scalability study relative to the MPI-based implementation of the benchmark.

6.6.1 MPI Comparison At Scale. The append primitive targets the regime where MPI's synchronous metadata exchange becomes the bottleneck: high process counts and small payloads. At small scale (128–256 processes, Figure 8), *MPI_Alltoallv*'s coordination cost is low and it performs comparably to Append—this is expected, since the collective overhead that append eliminates has not yet become significant.

As scale increases, the picture changes. At 850 nodes and 64 processes per node (54,400 processes, Figure 12)—a separate allocation from the 900-node SHMEM sweep of Figure 11, both drawn from the 1,002-node *System:2*, Append outperforms MPI by more than 6×. At this scale, *MPI_Alltoallv* spends the majority of its time in size exchange and scheduling rather than data movement, while Append issues one-sided transfers with no prior coordination.

The crossover point—where Append overtakes MPI—moves to lower node counts as process density per node increases (Section 6.4), because higher density amplifies the coordination overhead that the append primitive avoids.

6.7 Key Takeaways

The performance results highlight four practical takeaways.

- **Mechanism.** Offloaded append recovers injection bandwidth wasted by manual FADD+Put in two ways. First, it eliminates the sender-side stall: in manual append, the sender's NIC sits idle during the FADD round-trip before it can issue the Put; offloaded append fires the payload immediately with no prior coordination. Second, it removes FADD packets from the wire: each FADD carries no useful payload but consumes NIC injection slots and link bandwidth; at high process density, $O(N^2)$ such control packets compete with data transfers for the same links. The first effect dominates at low process counts; the second compounds at high density and scale.
- **Regime.** The largest gains appear for small to medium payloads and for configurations with high process density per NIC, where coordination costs and offset contention would otherwise dominate; as payload size increases, the benefit naturally tapers as both approaches become bandwidth-limited.
- **Scale.** At hundreds of nodes, Append sustains high utilization and remains competitive even as the job approaches the system-specific all-to-all ceiling imposed by the interconnect topology.

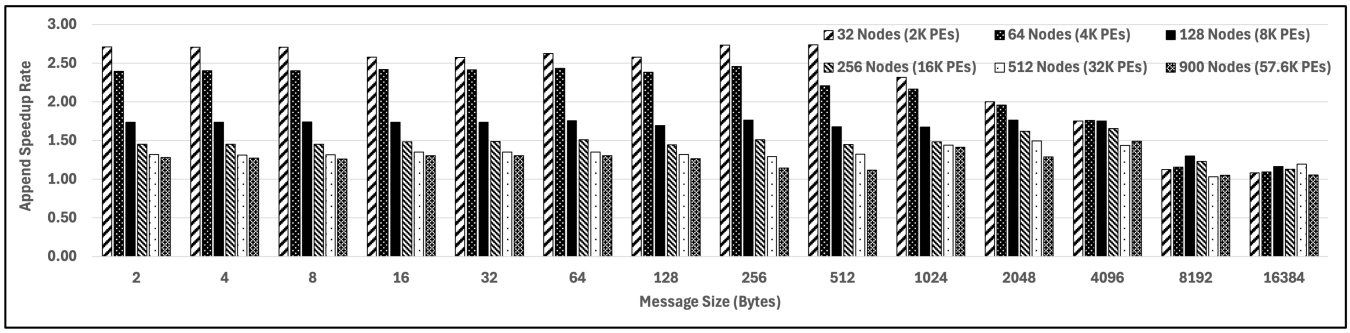


Figure 11: Scalability of offloaded Append *vs.* manual SHMEM (*System:2*, 32–900 nodes, 64 PPN, 2,048–57,600 processes; SHMEM scaling sweep).

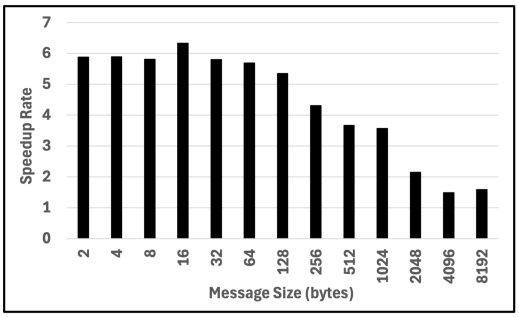


Figure 12: Append *vs.* MPI throughput at 850 nodes, 54,400 processes (*System:2*, 64 processes/node; separate MPI-comparison run).

- **Memory type.** Offloaded append performance remains robust when Tables are GPU-attached on heterogeneous nodes, indicating that the offload path does not depend on host memory type for efficiency.

Section 7 evaluates Append under more realistic application patterns with non-uniform message sizes and random destinations.

7 Application Kernels

This section evaluates the append framework using two application kernels with complementary goals: distributed bucket sorting demonstrates the *performance* gain from eliminating the FADD round-trip, whereas Mixture-of-Experts (MoE) expert-parallel dispatch demonstrates the *memory* savings from dense, receiver-managed placement. Unlike the controlled microbenchmarks in Section 6, these kernels exercise non-uniform message sizes, data-dependent destinations, and realistic process placement representing real-world scenarios.

7.1 Distributed Bucket Sorting

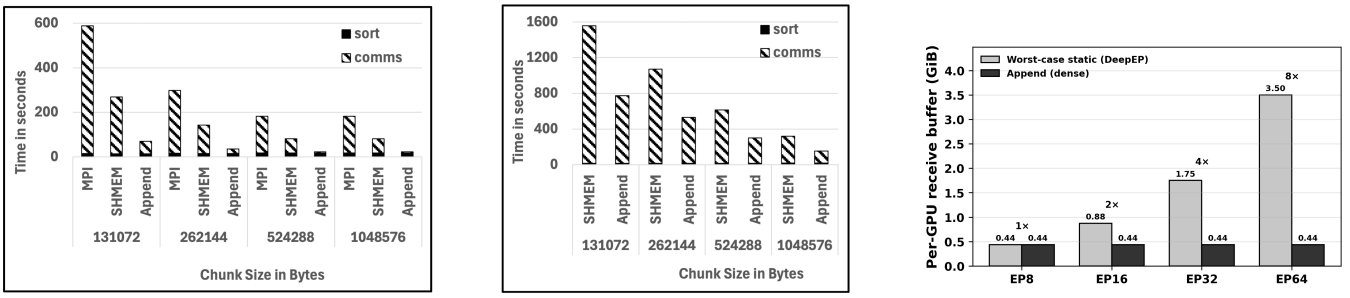
As described in Section 2, distributed bucket sorting exemplifies the many-to-many communication pattern. The algorithm proceeds in four phases: input generation, batched local sorting (*sort*), distributed data transfers (*comms*), and endgame processing (see Figure 3). Our benchmark focuses on the distributed data transfer phase, where each process distributes locally sorted data into destination buckets on remote processes. We compare three implementations:

(1) **MPI**: synchronous AlltoAll (*MPI_Alltoallv*) to exchange sizes, followed by point-to-point transfers.
(2) **SHMEM**: PGAS-style OpenSHMEM using atomic *FADD* to compute offsets before remote writes.
(3) **Append**: Proposed append framework performing remote appends without explicit offset calculations.

Figures 13a and 13b present the sorting performance across 64 and 512 nodes, respectively, using *System:2* from Table 2 with 64 processes per node. The input (unsorted table) size is fixed at 1 GiB per process, and the number of buckets matches the total process count. Results report total sorting time, with the distribution between sorting and communication phases highlighted. Since all three implementations use the same local sorting algorithm, the *sort* phase is identical across baselines; performance differences arise entirely from the *comms* phase.

All three implementations sort and transfer data in chunks. The chunk size determines how much data a process sorts locally before distributing it to remote buckets; smaller chunks allow earlier overlap of sorting and communication but produce more, smaller transfers and increase coordination cost. Because bucket occupancy is naturally skewed, per-chunk transfer sizes are non-uniform. The MPI baseline follows a synchronous two-phase strategy per chunk: global exchange of per-destination bucket sizes followed by point-to-point transfers, so its performance varies widely across chunk sizes.

64 Nodes: At 64 nodes (Figure 13a), 4 TiB of data is sorted. MPI exhibits the highest end-to-end time because the synchronous metadata exchange and coordination amplify the cost of the *comms* phase. SHMEM performs better by avoiding collective synchronization, but still incurs atomic overhead for offset allocation (FADD) before remote writes. Append outperforms both by allowing direct remote appends without explicit offset management, and shows consistent behavior across varying chunk sizes (i.e., different transfer granularities and non-uniform bucket sizes). This insensitivity arises because the receiver NIC handles placement regardless of chunk granularity, whereas SHMEM and MPI incur per-chunk coordination whose cost varies with the number and size of transfers.



(a) Sorting across 64 Nodes. (b) Sorting across 512 Nodes. (c) MoE dispatch receive-buffer footprint.

Figure 13: Application kernels: (a–b) distributed bucket-sort *performance*, (c) MoE dispatch per-GPU receive-buffer *footprint*.

Notably, at the right chunk size, Append enables effective overlap of the *sort* and *comms* phases: because append transfers are non-blocking and require no prior coordination, a process can begin distributing sorted chunks while local sorting of subsequent batches proceeds concurrently. Overall, Append achieves 10×–90× speedup over MPI and 5×–14× speedup over SHMEM depending on chunk size.

512 Nodes: At 512 nodes (Figure 13b), 32 TiB of data is sorted (32,768 processes). MPI results are omitted at this scale because the synchronous baseline requires an all-to-all size exchange across all processes and then executes a many-to-many payload transfer; at 32,768 ranks this coordination dominates the *comms* phase and overwhelms the end-to-end runtime, making the curve uninformative at the scale of the plot. SHMEM improves over MPI but still lags behind Append, which achieves up to 2× speedup over SHMEM across varying chunk sizes. The gap narrows from 5–14× at 64 nodes to 2× here because with 1 GiB per process and 32,768 destinations, both implementations approach the interconnect's all-to-all bandwidth ceiling. Append still wins because SHMEM continues to waste injection slots on FADD round-trips that append eliminates, recovering that bandwidth fraction for payload delivery.

While bucket sorting highlights the *performance* gained by eliminating the FADD round-trip, the next kernel turns to a complementary benefit: the receive-buffer *memory* saved by dense, receiver-managed placement in Mixture-of-Experts dispatch.

7.2 MoE Expert-Parallel Dispatch

Mixture-of-Experts (MoE) dispatch is another many-to-many pattern: a gating function sends each token to a top-k subset of experts, and every process deposits variable-sized token batches into per-expert buffers on remote nodes. As in bucket sort, the per-expert counts are unknown until routing runs, and they change from step to step as the gating shifts.

What differs is the size of the worst case. A sort bucket can receive the entire dataset and so cannot be pre-sized, whereas an expert receives at most the tokens the ranks hold this step, a bounded amount that fits in memory. Libraries such as DeepEP [15] use this to avoid a size exchange: they pre-post a receive buffer large enough for every source to send all of its tokens to one local expert. The buffer is safe under any routing, but it wastes scarce GPU (HBM) memory, the more so as routing grows skewed.

In our prototype, each process issues `shmem_append_nbi` host-initiated operations that stream tokens directly into GPU-attached per-expert *Tables*; the receive-side byte counter then reports per-expert token counts without a size-exchange collective. We use a DeepSeek-V3-style configuration [14]: hidden size 7168 with `bf16` tokens (a 14,336 B per-token message) and top-8 routing, in the prefill-dispatch regime. We weak-scale expert parallelism across 2, 4, 8, and 16 nodes (EP8 through EP64, four processes per node) and control routing skew with a capacity-factor knob.

Transfer performance. At the 14 KiB per-token message size, append does not slow dispatch: it matches FADD+Put once the link saturates at high process density and runs faster at lower density (Section 6.4). Its value for MoE is therefore the memory it saves.

Receive-buffer footprint. Because the receiver manages placement and tracks arrivals with a byte counter, append sizes a single dense *Table* to the bytes it actually receives instead of to the worst case, which reserves num_experts/k times more. At EP64 the dense buffer holds 0.44 GiB against a 3.5 GiB worst-case reservation, an 8× reduction (Figure 13c). The factor grows with the expert count, from 1× at EP8 to 8× at EP64, since a token reaches a shrinking fraction of the experts; at the 256 experts of DeepSeek-V3 it would reach 32×. Routing skew does not change the total: every token still lands on k experts, so the received bytes are fixed and skew only shifts where they land. Even under heavy skew the busiest GPU stays under 0.7 GiB, well below the flat 3.5 GiB worst case.

The placement append provides is dense and needs no size exchange, matching the receiver-managed contract of Section 4. The memory it frees can be spent elsewhere, on a larger per-step batch or more experts per GPU, for an end-to-end throughput gain that depends on the workload and that we do not measure here. Driving the same primitive from GPU threads, inside a device-initiated MoE runtime, is ongoing work; here we validate the host-initiated mechanism.

7.3 Application Performance Summary

The application benchmarks demonstrate the versatility and performance benefits of the proposed append framework across both distributed sorting and MoE dispatch many-to-many communication patterns. In distributed bucket sorting, the append framework achieves up to 90× speedup over MPI and 14× over SHMEM at 64 nodes, while maintaining 2× speedup over SHMEM at 512 nodes. For MoE expert-parallel dispatch, append matches baseline dispatch

time at MoE message sizes while cutting the per-GPU receive-buffer footprint by up to 8× at EP64. These results validate that receiver-managed placement significantly improves performance, scalability, and memory efficiency for data-intensive distributed workloads.

8 Related Work

This section surveys prior work on communication aggregation and the append pattern, and clarifies how this work differs.

DeVinney and Maley introduced the *conveyor* abstraction [16] to address irregular many-to-many data exchanges common in graph and sparse computations. Their model buffers and coalesces tokens in pipelines, reducing per-message overhead. While effective, conveyors operate at the software level without exploiting network hardware for offset management.

Dinan and Flajslik proposed the *one-sided append primitive* [17], combining atomic fetch-and-add with one-sided *Put* operations to enable concurrent appends to a shared buffer. Their software-based approach demonstrated performance improvements for benchmarks such as NAS Integer Sort; the manual SHMEM baseline evaluated throughout this paper is essentially their technique. Our contribution builds on this foundation by formalizing the semantics and offloading the primitive to NIC hardware. Hammond et al. [21] and Zhou et al. [34] provided general techniques for optimized PGAS runtimes through improved one-sided communication and MPI-3 shared memory, respectively.

Paul et al. [27] proposed a scalable actor-based PGAS runtime that automatically batches fine-grained messages to reduce network congestion, relying on runtime heuristics for coalescing.

These approaches are closely related in motivation: they reduce overheads from fine-grained, irregular communication by buffering, batching, and avoiding unnecessary synchronization. However, their append and aggregation mechanisms ultimately rely on software-visible coordination (e.g., atomics for offset allocation, collective metadata exchange, or runtime heuristics), which becomes increasingly costly at extreme scale and high process density.

The works above primarily represent software-level approaches to aggregation. In contrast, this paper makes three distinct contributions beyond exploiting any single network feature.

First, it defines a *formal append framework*—a session-based lifecycle with well-defined semantics for buffer management, placement, progress queries, and termination—that is a programming-model abstraction independent of any specific interconnect. This framework can be realized entirely in software (the FADD+Put fallback path evaluated in Section 6) or accelerated by hardware.

Second, it identifies a *minimal, portable hardware contract*—receiver-side offset management and receive-side progress accounting—that enables full NIC offload of the append primitive. While individual mechanisms such as receiver-managed offsets have been available in Portals-class networks for decades [6, 8, 9], the contribution here is the systematic identification of which capabilities are necessary and sufficient to offload the complete append lifecycle, and the demonstration that this contract extends beyond Portals to the emerging Ultra Ethernet specification [31].

Third, it provides a *reference co-design path*: a concrete mapping from the high-level API through Libfabric middleware to NIC primitives (LMO, byte counters), together with a quantitative characterization of the regimes where hardware offload is most beneficial versus where the portable software path remains competitive. Higher-level abstractions such as conveyors [16] and actor-based runtimes [27] are complementary: they could adopt the proposed primitive as an underlying transport to preserve their programming model while benefiting from hardware-assisted placement and status tracking.

9 Conclusion

This work introduces a formal *append framework and primitive* for accelerating inter-process message aggregation, a recurring bottleneck in irregular many-to-many communication patterns. The framework defines managed buffer regions with receiver-side placement and byte-level status queries. These semantics form a programming-model abstraction independent of any single network technology. Crucially, it identifies a minimal hardware contract—receiver-side offset management and receive-side progress accounting—that enables full NIC offload, and shows that this contract is met by HPE Slingshot and mandated by the emerging Ultra Ethernet specification.

A reference implementation in OpenSHMEM demonstrates the concrete mapping through Libfabric to NIC primitives (LMO and hardware byte counters), while a software fallback path (FADD+Put) preserves portability on networks without hardware support.

Evaluation demonstrates that the approach improves both efficiency and scalability. The core mechanism is bandwidth recovery: offloaded append eliminates sender-side injection stalls and removes zero-payload control traffic from the wire, reclaiming that capacity for useful data movement. For distributed bucket sorting, Append achieves up to 90× speedup over MPI and 14× over SHMEM at 64 nodes (4 TiB), while maintaining up to 2× speedup over SHMEM when sorting 32 TiB across 512 nodes (32,768 processes). Benefits are most pronounced for small to medium messages and high process density per NIC, and remain competitive as jobs approach the system-specific all-to-all ceiling at large scale.

Future work includes broadening portability to more interconnects and runtimes, and extending the interface with dynamic resizing, stronger ordering, and fault-aware progress.

References

[1] Osayamen Jonathan Aimuyo, Byungsoo Oh, and Rachee Singh. 2025. FlashMoE: Fast Distributed MoE in a Single Kernel. In *Proceedings of a Major Machine Learning Conference.* Thirty-Ninth Annual Conference on Neural Information Processing Systems (NeurIPS 2025).

[2] OpenFabrics Alliance. [n. d.]. libfabric: A High-Performance Fabric Interface. https://ofiwg.github.io/libfabric/. Accessed: 2025-10-22.

[3] Scott Atchley, Christopher Zimmer, John Lange, David Bernholdt, Veronica Melesse Vergara, Thomas Beck, Michael Brim, Reuben Budiardja, Sunita Chandrasekaran, Markus Eisenbach, Thomas Evans, Matthew Ezell, Nicholas Frontiere, Antigoni Georgiadou, Joe Glenski, Philipp Grete, Steven Hamilton, John Holmen, Axel Huebl, Daniel Jacobson, Wayne Joubert, Kim Mcmahon, Elia Merzari, Stan Moore, Andrew Myers, Stephen Nichols, Sarp Oral, Thomas Papatheodore, Danny Perez, David M. Rogers, Evan Schneider, Jean-Luc Vay, and P. K. Yeung. 2023. Frontier: Exploring Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.*

[4] David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Richard L. Carter, Lawrence Dagum, R. A. Fatoohi, P. O. Frederickson, Thomas A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications.* Accessed: 2025-05-01.

[5] Stephen J. Bailey, Deborah Bard, Surendra Byna, Amit Chourasia, Aashish Devarakonda, Lisa Gerhardt, Prabhat Ghosh, Sabine Hittmeir, Khaled Ibrahim, Shoaib Kamil, James Krueger, Thomas Lenz, John Lewis, Tobias Luft, Xiaoyi Ma, Peter Mendygral, Dong Mu, Prabhat, Mark Romanus, John Shalf, David Skinner, Jonas Strebel, Vamshi Suram, Jialin Tang, Michael Thomas, Craig Tull, Brian van Straalen, Samuel Williams, Lin Yang, and Andy Yoo. 2022. The NERSC Perlmutter Supercomputer. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC)* (Basel, Switzerland). Association for Computing Machinery, 1–12. https://doi.org/10.1145/3539781.3539788

[6] Brian W. Barrett, Ron Brightwell, Ryan E. Grant, William Schonbein, Kevin S. Hemmert, Kevin Pedretti, Keith D. Underwood, Rolf Riesen, Torsten Hoefler, Michael Barbe, Lucas H. Suraty Filho, Alexander Ratchov, and Arthur B. Maccabe. 2022. *The Portals 4.3 Network Programming Interface.* Technical Report SAND2022-8810. Sandia National Laboratories.

[7] D. Bonachea. 2002. GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages. http://gasnet.lbl.gov/pubs/258-paper.pdf.

[8] Ron Brightwell, Todd Hudson, Rolf E. Riesen, and Arthur B. Maccabe. 1999. *The Portals 3.0 Message Passing Interface Revision 1.0.* Technical Report SAND99-2959. Sandia National Laboratories. https://doi.org/10.2172/15154

[9] Ron Brightwell and Lance Shuler. 1996. Design and Implementation of MPI on Puma Portals. In *Proceedings of the Second MPI Developer's Conference.* IEEE Computer Society Press, 18–25. https://doi.org/10.1109/MPIDC.1996.534090

[10] D. Callahan, B.L. Chamberlain, and H.P. Zima. 2004. The cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.*

[11] William W. Carlson, Jesse M. Draper, and David E. Culler. 1996. S-246, 187 Introduction to UPC and Language Specification.

[12] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS '10).*

[13] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. 2020. An In-Depth Analysis of the Slingshot Interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.*

[14] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. *arXiv e-prints* (2024). arXiv:2412.19437.

[15] DeepSeek-AI. 2025. DeepEP: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP. Accessed: 2026-08-24.

[16] Jason G. DeVinney and Miller M. Maley. 2019. Conveyors for Streaming Many-to-Many Communication. In *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3).* 1–8. https://doi.org/10.1109/IA349570.2019.00007

[17] James Dinan and Mario Flajslik. 2014. One-Sided Append: A New Communication Paradigm for PGAS Models. In *Proc. 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14).* University of Oregon, Eugene, OR, USA. https://www.nic.uoregon.edu/pgas14/oug_submissions/oug2014_submission_2.pdf

[18] G. Faanes, A. Bataineh, B. Bergen, S. Blond, G. Bosilca, D. Chen, and et al. 2012. Cray Aries Interconnect. In *Proceedings of the Cray User Group (CUG) Conference.* Cray Inc. https://cug.org/proceedings/cug2012_proceedings/includes/pages/papers/pap117.pdf Accessed: 2025-05-01.

[19] Evangelos Georganas, Rob Egan, Steven Hofmeyr, Eugene Goltsman, Bill Arndt, Andrew Tritt, Aydin Buluç, Leonid Oliker, and Katherine Yelick. 2018. Extreme scale de novo metagenome assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18).*

[20] Evangelos Georganas, Marquita Ellis, Rob Egan, Steven Hofmeyr, Aydin Buluç, Brandon Cook, Leonid Oliker, and Katherine Yelick. 2017. MerBench: PGAS Benchmarks for High Performance Genome Assembly. In *Proceedings of the Second Annual PGAS Applications Workshop (PAW17).*

[21] Jeff Hammond, James Dinan, Pavan Balaji, Sreeram Potluri, and Vinod Tipparaju. 2013. OSPRI: An Optimized One-Sided Communication Runtime for Leadership-Class Machines. In *IEEE Cluster Conference.*

[22] Steven Hofmeyr, Rob Egan, Evangelos Georganas, Alex Copeland, Robert Riley, Alicia Clum, Emiley Eloe-Fadrosh, Simon Roux, Eugene Goltsman, Aydin Buluç, Daniel Rokhsar, Leonid Oliker, and Katherine Yelick. 2020. Terabase-scale metagenome coassembly with MetaHipMer. *Scientific Reports.*

[23] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, Joe Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. 2022. Tutel: Adaptive Mixture-of-Experts at Scale. *arXiv e-prints* (2022). arXiv:2206.03382.

[24] InfiniBand Trade Association. 2001. InfiniBand Architecture Specifications Release 1.0.a. http://www.infinibandta.org. Accessed: 2025-05-01.

[25] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. *arXiv e-prints* (2024). arXiv:2401.04088.

[26] MPI Forum. [n. d.]. MPI_Alltoallv. https://www.mpich.org/static/docs/v3.1/www3/MPI_Alltoallv.html. Accessed: 2025-10-22.

[27] Sri Raj Paul, Akihiro Hayashi, Kun Chen, and Vivek Sarkar. 2021. A Scalable Actor-based Programming System for PGAS Runtimes. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21).* https://arxiv.org/abs/2107.05516

[28] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. *arXiv e-prints* (2022). arXiv:2201.05596.

[29] William Saphir, Alex Woo, and Maurice Yarrow. 1996. *The NAS Parallel Benchmarks 2.1 Results.* Technical Memorandum NASA-TM-112038. NASA Ames Research Center. https://ntrs.nasa.gov/citations/19970009364 Accessed: 2025-10-22.

[30] Jesper Larsson Träff. 2004. Efficient Algorithms for All-to-All Personalized Communication and Synchronization. *Concurrency and Computation: Practice and Experience* 16, 7 (2004), 611–630. https://doi.org/10.1002/cpe.759

[31] Ultra Ethernet Consortium. 2024. Ultra Ethernet Specification, Version 1.0. https://ultraethernet.org/. Accessed: 2026-02-02.

[32] Ohio State University. [n. d.]. OSU Micro-Benchmarks. https://github.com/MPIP/OSU-IMB. Accessed: 2025-10-22.

[33] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, Quan Chen, and Xin Liu. 2025. Comet: Fine-grained Computation-communication Overlapping for Mixture-of-Experts. *arXiv e-prints* (2025). arXiv:2502.19811.

[34] Huan Zhou, Kamran Idrees, and José Gracia. 2016. Leveraging MPI-3 Shared-Memory Extensions for Efficient PGAS Runtime Systems. *Concurrency and Computation: Practice and Experience* 28, 12 (2016), 3459–3477.