Flare: Flexible In-Network Allreduce

Daniele De Sensi
daniele.desensi@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Salvatore Di Girolamo
salvatore.digirolamo@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Saleh Ashkboos
saleh.ashkboos@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Shigang Li
shigang.li@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

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

ABSTRACT

The *allreduce* operation is one of the most commonly used communication routines in distributed applications. To improve its bandwidth and to reduce network traffic, this operation can be accelerated by offloading it to network switches, that aggregate the data received from the hosts, and send them back the aggregated result. However, existing solutions provide limited customization opportunities and might provide suboptimal performance when dealing with custom operators and data types, with sparse data, or when reproducibility of the aggregation is a concern. To deal with these problems, in this work we design a flexible programmable switch by using as a building block PsPIN, a RISC-V architecture implementing the sPIN programming model. We then design, model, and analyze different algorithms for executing the aggregation on this architecture, showing performance improvements compared to state-of-the-art approaches.

CCS CONCEPTS

• **Networks** → **In-network processing**; • **Hardware** → **Networking hardware**; • **Computer systems organization** → **Distributed architectures**.

KEYWORDS

In-Network Computing; Programmable Switch; Allreduce

ACM Reference Format:
Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. 2021. Flare: Flexible In-Network Allreduce. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458817.3476178

1 INTRODUCTION

Allreduce is a commonly used collective operation where P vectors, one for each host participating in the operation, are aggregated together. If each vector contains Z elements, the allreduce operation aggregates the P vectors element-wise and returns to each host a vector of Z aggregated elements. Common aggregation functions include the sum of the elements, the computation of their minimum or maximum, and others [1]. Allreduce is widely used in many applications, including scientific applications [2, 3], deep learning [4], graph processing [5, 6], big data analytics [7], and others, and recent studies [2] show that "`MPI_Allreduce` *is the most significant collective in terms of usage and time*".

The simplest bandwidth-optimal allreduce algorithm is the Rabenseifner algorithm (also known as *ring allreduce*) [8]. This algorithm is composed of two phases: a *scatter-reduce*, and an *allgather* phase. P hosts are arranged into a logical ring, and in each of these two phases, each host sends to its neighbor $(P-1)$ messages, each of size $\frac{Z}{P}$ (where Z is the number elements to be reduced). The total amount of data sent by each host is then $2(P-1)\frac{Z}{P} \approx 2Z$. To reduce the amount of transmitted data, and thus increase the performance, hosts can exploit *in-network* compute, i.e., they can offload the allreduce operation to the switches in the network.

To outline the advantages of performing an in-network allreduce, we describe the general idea underlying most existing in-network reduction approaches [9–11]. We first suppose to have the P hosts connected through a single switch. Each of the hosts sends its data to the switch, that aggregates together the vectors coming from all the hosts, and then sends them back the aggregated vector. Differently from the host-based optimal allreduce, in the in-network allreduce each host only sends Z elements, thus leading to a 2x reduction in the amount of transmitted data. If the switches can aggregate the received data at line rate, this leads to a **2x bandwidth improvement** compared to a host-based allreduce. Besides improvements in the bandwidth, in-network allreduce also reduces the network traffic. Because the interconnection network consumes a large fraction of the overall system power (from 15% to 50% depending on the system load [12]), any reduction in the network traffic would also help in reducing the power consumption and thus the running cost of the system.

If the hosts participating in the reduction span across multiple switches, the aggregation can be done recursively, as shown in Figure 1. Let us suppose the hosts *H0*, *H1*, and *H3* need to perform an allreduce on their data, denoted through geometric figures (A). First, they build a *reduction tree*, where the leaves are the hosts and the intermediate nodes are a subset of the switches (*S0*, *S2*, and *S3* – B). After a switch aggregates all the data coming from its children, if it is an intermediate switch in the tree, it sends the

SC '21, November 14–19, 2021, St. Louis, MO, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476178

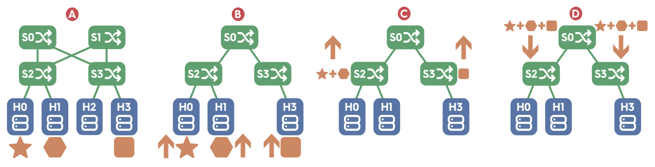

Figure 1: Example of an in-network allreduce.

aggregated data to its parent (Ⓒ). Otherwise, if it is the root of the tree, it broadcasts the data down the tree, and the fully reduced data will eventually reach all the hosts (Ⓓ).

Although many in-network allreduce solutions have been designed in the years, especially for HPC networks, all of them lack flexibility, in terms of supported data types, data formats, and operators, as we show in Section 2. We show in this work that this limits the applicability of in-network reduction (for example, none of them can deal with sparse data) and can lead to performance degradation. To overcome these limitations, we propose FLARE (*Flexible In-Network Allreduce*). FLARE includes a programmable switch based on PsPIN [13], and a set of aggregation algorithms for exploiting the switch architecture at best. PsPIN is an open-source multi-cluster RISC-V architecture [14], implementing the sPIN programming model [15], and allowing the programmer to specify *packet handlers* (i.e., the code to be executed for each packet), as plain C functions. The packet handlers can be programmed and loaded by the system administrator after the switch has been deployed. This extends the range of applicability and eases the programmer's task compared to existing programmable switches.

In this work, we answer the following research questions:

- **Q1** What are the limitations of existing in-network data reduction solutions and why there is the need for flexible in-network reductions? (Section 2)
- **Q2** How to design a switch architecture for flexible in-network data reduction? (Section 3)
- **Q3** How to exploit such architecture to efficiently execute allreduce operations? (Sections 4-7)
- **Q4** What performance advantages can we expect compared to existing in-network and host-based solutions? (Sections 6.4 and 7.1)

2 STATE OF THE ART

Existing solutions for in-network reductions can be divided into three broad categories: those implemented on fixed-function (i.e., non programmable) switches, those relying on *Field-Programmable Gate Arrays* (FPGAs), and those targeting programmable switches.

2.1 Fixed-function switches

Solutions targeting fixed-function switches [9, 10, 16–21] are characterized by high performance but tied to a specific implementation, and do not provide any customization opportunity. Many HPC networks provide some kind of support for in-network allreduce. For example, SHARP [9, 16] can be used on Mellanox's networks, and similar solutions are provided on Tofu [18], and Cray's Aries networks [17]. Usually, these solutions support the most commonly used MPI reduction operators, on both integer and floating point data, but some of them might only support latency-sensitive small data reductions [17, 18].

2.2 FPGAs

Another set of solutions targets *Field-Programmable Gate Arrays* (FPGAs) [22, 23], trading performance for some flexibility. Some of these solutions [22] work by placing the FPGA between the hosts and the switch. Other solutions instead connect the FPGA only to the switch [23], and configure the switch to route the packets that need to be aggregated to the FPGA. After the FPGA aggregates the packet, it sends the result back to the switch, that will then send it to the next hop. However, even if more flexible than fixed-function switches, extending an FPGA-based in-network allreduce solution might still require a hardware re-design. Moreover, the FPGAs considered by these solutions have limited network bandwidth, ranging from four 10Gbps ports [22] to six 100Gbps ports [23].

2.3 Programmable switches

Last, some solutions target programmable switches [11, 24, 25], that provide more flexibility by allowing the programmer to specify the type of processing to be executed on the packets with high-level programming languages. Programmable switches are often implemented through *Reconfigurable Match-Action Tables* (RMTs) [26–29], and can be configured with the P4 programming language [30–32]. In such architecture, each packet traverses a pipeline of 10 – 20 stages [33], each applying a longest-prefix match of one or more packet fields against a set of rules, to select an action to be executed on that field. Possible actions include simple *Arithmetic Logic Unit* (ALU) operations, memory read/write, and modifications of packet fields and the packet destination. Both the matching rules and the actions to be executed can be customized by the programmer.

However, the existing programmable switches architecture is rigid and has several limitations [33, 34], including the lack of loops, and a limited set of operations. Operations with dependencies must be placed on subsequent pipeline stages and non-trivial applications cannot be mapped on such an architecture. Moreover, state management is limited, and everything needs to be expressed as a statically-sized array. Existing hardware can also only perform 32 operations per packet [11, 24], and each packet can only access each memory location once. To overcome these limitations, packets can be *"recirculated"*, i.e., sent back to the switch on a loopback port. This however reduces the bandwidth of the switch proportionally to the number of times each packet is recirculated. For example, to process the data sent by the hosts at 100Gbps, existing allreduce implementations for programmable switches only allow 16 ports to be used on a 64-port switch [11].

2.4 Limitations

We identified the following *flexibility limitations* in existing in-network allreduce solutions, that we summarize in Table 1, and that we overcome in this work:

F1 - Custom operators and data types. Solutions for fixed-function switches and FPGAs support a predefined set of operators and data types [9], that cannot be customized nor extended (Table 1, 👎). In-network aggregation solutions for programmable switches can be customized, but they are still limited by the hardware (🤚).

	Fixed-Function Switches							FPGAs		Progr. Switches			
	[9]	[16]	[17]	[18]	[19]	[21]	[10]	[22]	[23]	[24]	[11]	[25]	**Flare**
F1	👎	👎	👎	👎	👎	👎	👎	👎	👎	👈	👈	👈	👍
F2	👎	👎	👎	👎	👎	👎	👎	👎	👎	👎	👎	👈	👍
F3	?	👍	?	?	👍	?	👎	👍	👍	👎	👎	👎	👍

Table 1: Comparison between existing in-network allreduce solutions. F1: Custom operators and data types, F2: Sparse data, F3: Reproducibility. 👍: provided, 👈: partially provided, 👎: not provided, ?: unknown.

For example, existing programmable switches do not have floating-point units, and do not support multiplication/division for integer data [11, 24, 27]. Moreover, due to the limited number of elements that can be processed per packet (independently from the element size), aggregating sub-byte elements, as common in deep learning applications [35–38] would not improve application performance. On the other hand, Flare provides full customizability of operators and data types (👍), allowing the user to specify arbitrary aggregation functions as sPIN handlers (Section 3).

F2 - Sparse data. Many applications need to reduce sparse data [5, 25, 35, 39], i.e., data containing mostly null values. To save bandwidth and improve performance, an application might only transmit and reduce the non-null values. However, to the best of our knowledge, none of the solutions targeting fixed-function switches provide explicit support for sparse data (👎). Among the programmable switches solutions, only one partially targets in-network sparse data reduction [25]. However, it forces the application to sparsify the data per-block, i.e., to send sparse blocks of dense data (👈). This is however not possible for any application and, even when possible (for example for deep learning models training), this negatively affects the convergence of the training [25]. On the other hand, with Flare we design the first in-network sparse allreduce algorithm, that does not make any assumption on the data sparsity, and can process the data as generated by the hosts (👍), improving application performance compared to existing solutions (Section 7.1).

F3 - Reproducibility. Many scientific applications require the computed results to be reproducible across different runs and for different allocations. For example, in weather and climate modeling, a small difference in computation on the level of a rounding error could lead to a completely different weather pattern evolution [40, 41]. However, some aggregation functions (e.g., floating-point summation [42–46]) might depend on the order in which the elements are aggregated, and the final result might change if, in subsequent runs, packets arrive at the switch in a different order. Many solutions for fixed-function switches ensure reproducibility, in most cases by storing all the packets and aggregate them in a pre-defined order only when they are all present [9, 22]. This however increases the memory occupancy of the switch, even when this is not required by the application, for example for integer data or for an application that might tolerate different results across different runs (👍). Differently from existing solutions, Flare guarantees reproducibility only when explicitly requested by the user, by organizing the aggregation in the switch so that associativity of the operator is never used. Moreover, Flare reproducible allreduce does not require storing all the packets before aggregating them (👍).

3 SWITCH ARCHITECTURE

Figure 2 illustrates the high-level architecture of a Flare PsPIN-based programmable switch. After a packet is received from any of the switch ports, its headers are processed by a *parser* [47, 48] that, based on configurable *matching rules*, decides if the packet must be processed by a *processing unit* (or sent directly to the routing tables unit[1]), and which function must be executed on the packet. The processing unit can modify the content of each packet, and decide its destination. We assume that the network administrator configures the matching rules in the parser through the control plane [49], specifying the functions to execute for each packet, based on the values of specific packet fields (e.g., *EtherType* in the Ethernet header [50], or IP optional headers). In principle, the code to be executed could be specified by users of the system, but this would open security and accountability concerns that are outside the scope of this paper.

Differently from existing programmable switches [27, 28, 30, 51, 52], that implement the processing unit through *Reconfigurable Match Action* (RMT) tables [26, 29], we consider the processing unit to be implemented as a PsPIN unit [13], highlighted in the right part of Figure 2. PsPIN is a clustered RISC-V built on top of the PULP [53] platform. If a packet needs to be processed, the parser copies the packet in a 4MiB *L2 packet memory* and sends a request to a packet scheduler[2]. The packet scheduler forwards the request to one of multiple clusters (four in the example). A cluster-local scheduler (CSCHED) then selects a *Handler Processing Units* (HPU, eight in the example, denoted with H) where the packet will be processed, and starts a DMA copy of the packets from the L2 packet memory to a single-cycle scratchpad 1MiB memory (L1 TCDM - *Tightly-Coupled Data Memory*). The cluster scheduler also loads the code to be executed from the 32KiB *L2 program memory* to a cluster-local 4KiB instruction cache (not shown in the figure). Each HPU is a RISC-V RI5CY core [54] that can execute sPIN *handlers* [15] i.e., C functions defining how to process the content of the packet. The handlers can use the single-cycle L1 memory also to store and load data, that is preserved after the processing of the packet is terminated, and until the handler is uninstalled from the control plane. Each cluster also has a DMA engine that can be used to access a globally shared 4MiB memory (*L2 handler memory*).

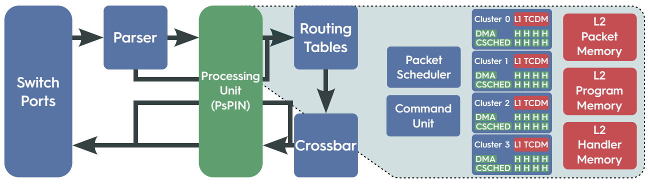
Switch Ports
Parser
Processing Unit (PsPIN)
Routing Tables
Crossbar
Packet Scheduler
Command Unit
Cluster 0 L1 TCDM DMA CSCHED H H H H H H H H
Cluster 1 L1 TCDM DMA CSCHED H H H H H H H H
Cluster 2 L1 TCDM DMA CSCHED H H H H H H H H
Cluster 3 L1 TCDM DMA CSCHED H H H H H H H H
L2 Packet Memory
L2 Program Memory
L2 Handler Memory

Figure 2: PsPIN switch high-level architecture.

[1]By doing so, packets that do not need additional processing are not further delayed.
[2]If the packet memory is full, the packet is dropped or congestion is notified before filling the buffer, depending on the specific network where the switch is integrated.

After processing a packet, each HPU can set its destination and send it to the routing tables unit through a command unit. After getting the destination port, the packet is sent to a crossbar unit, that also implements queueing and quality of service (QoS) functionalities. This unit is also known as *traffic manager* [27], and we assume that the implementation and functionalities of both the routing tables unit and the crossbar are similar to those of any other switch. When the packet is ready to be sent, it might be processed again by the processing unit[3]. After this additional and optional processing step, the packet can be either dropped or forwarded.

The PsPIN processing unit is clocked at 1 GHz, and a PsPIN unit with four clusters with eight cores each (as the one shown in the example) occupies 18.5 mm^2 [13] in a 22nm FDSOI process. Half of the area in the PsPIN unit is occupied by the L2 memory, and each cluster occupies on average 2mm^2 [13], most of which is L1 memory (1.64mm^2). We also add an FP32/FP16 *Floating Point Unit* (FPU) [55] to each core, increasing the area of the cluster to 2.29mm^2. The processing unit in programmable switches is estimated to occupy up to 140mm^2 in a 28nm FDSOI process [56]. By scaling the area, we set a 180mm^2 target area for our PsPIN unit, and we then assume we can fit ~64 clusters and the L2 memory in our processing unit area budget. We assume that more clusters can be fit in the unit by organizing them hierarchically (in principle, by having separate units with a scheduler on the front). Moreover, it is worth remarking that our estimation of the available area budget is conservative because existing switches are manufactured in 16nm [57] to 7nm [58] processes.

4 FLARE ALLREDUCE GENERAL DESIGN

In Flare, before starting the aggregation, the application sends a request to a network manager node [9, 16], that computes a reduction tree, and installs the handlers on all the switches of the tree through the control plane of the network. For each switch of the reduction tree, the network manager also sets the number of ports from which it will receive the packets to be aggregated (i.e., its children in the reduction tree), and the port to be used to forward the aggregated data (i.e., the port connected to the parent in the reduction tree). Because the structure of the reduction tree depends on the location of the hosts participating in the reduction, this setup phase must be done once for each subset of hosts executing an allreduce. For example, in the case of MPI, this must be done once for each communicator, similarly to other existing approaches [9, 16].

Each switch can participate simultaneously in different allreduces (issued by the same user/application or by different ones), and the network manager assigns a unique identifier to each allreduce, so that only packets belonging to the same allreduce are aggregated together. Because each allreduce consumes some memory resources on the switch, we assume that the memory is statically partitioned across a predefined maximum number of allreduces, as done by most in-network reduction approaches [9, 11, 16–18, 25]. If any switch on the reduction tree is already managing the maximum possible number of allreduces, the network manager can try to recompute a different reduction tree excluding that switch or just reject the request issued by the application, which then needs to rely on host-based allreduce [9, 16]. Because the memory is partitioned and the compute resources are dynamically assigned, to simplify the analysis in the following we focus on describing what happens for a single allreduce.

[3]For example, this might be useful when performing telemetry, to insert in the packet information about queue occupancy or queueing time.

Once the setup is complete, the hosts can start sending the data to be reduced. We assume that Z elements must be aggregated and that each host splits its data in $\frac{Z}{N}$ packets of N elements each. In Figure 3 we show an example where we have three hosts, and each of them splits the data into five packets. To simplify the exposition we denote each packet with two indexes: the index of the host that generated the packet, and the position of the packet within the host data. For example, *Packet 0,1* is the second packet sent by *Host 0*. The size of the packets is smaller than the *Maximum Transmission Unit* (MTU) and, besides the payload, the hosts also add a small header containing the identifier of the allreduce (not shown) and of the packet within that allreduce. We refer to a set of packets to be aggregated together as a *reduction block* (*block* for brevity). In our example, the set of packets *Packet x,2* is a block, and the switch aggregates them together. There are three main resources that the switch needs to manage: the cores (HPUs), the input buffer memory (i.e., the memory where the received packets are stored while being processed), and the working memory (i.e., the memory where the partially aggregated data is stored). The working memory is located in the L1 memory of the clusters, and the input buffers in the L2 packet memory (and copied in the L1 memory before starting the handler). We show how these resources interact in Figure 4, assuming a switch with 4 cores, processing the data shown in Figure 3.

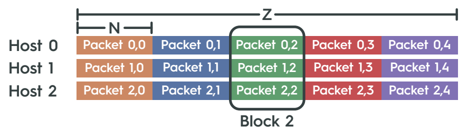

Figure 3: Partitioning of the data into packets and *aggregation blocks.*

4.1 Cores

When a matching packet arrives at the processing unit of the switch, it is scheduled to one of the cores (we discuss scheduling more in detail in Section 5). For example, in Figure 4, *Packet 2,0* is the first to arrive and it is scheduled to *Core 0*. Because it is the first packet received for *Block 0*, its content is simply copied in an available *aggregation buffer* in the L1 memory (Figure 4, A). All the subsequent packets received for the same block are aggregated in the same buffer. For each block, the handler keeps a *children* counter, incremented each time a packet of a block is received. The block can be considered fully aggregated when the counter is equal to the number of children in the reduction tree (specified when the handler was installed). In the example in Figure 4, this happens at B for *Block 0*. The content of the aggregation buffer is then multicast to the children (or if the switch is not the root of the tree, sent to its parent), and the aggregation buffer is released, so that it could be used later by a subsequent block (in the example, by *Block 3*).

We assume that if a packet is lost, a timeout is triggered in the host, that retransmits the packet. To manage retransmissions, FLARE can use a bitmap (with one bit per port) rather than a counter. When a packet of a block is received on a port, FLARE checks the corresponding bit. If the bit is not set, the packet is aggregated and the bit is flipped. If the bit was already set, then the packet is a retransmitted packet, its content was already aggregated and it is not aggregated again. To simplify the exposition, in the following we assume refer to a *children* counter rather than a bitmap.

The aggregation of the packets belonging to the same block can be done in different ways and has a direct impact on the maximum bandwidth that can be achieved by the switch when aggregating data. We denote with τ the service time of a core (i.e., the number of cycles it needs to aggregate a packet). Because we have K cores (4 in the example), if the workload is evenly distributed across the cores (and we will show this is the case), the maximum bandwidth achievable by the switch is expressed as $\frac{K}{\tau}$. We assume the switch receives a packet every δ cycles and we can then express the bandwidth of the switch (in packets processed per cycle) as $\mathscr{B} = \min\left(\frac{K}{\tau}, \frac{1}{\delta}\right)$. K, τ, and δ are determined by the specific switch and network design, and we show in Section 6 how to properly organize the computation so to minimize τ.

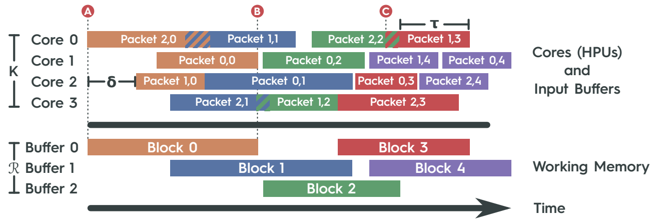

Figure 4: Utilization of input buffers, cores, and working memory during an in-network allreduce on a switch with 4 cores, processing the data shown in Figure 3.

4.2 Input Buffers Memory

While being processed, the packets occupy input buffers memory, for as long as the handler duration. Additionally, if there are no cores available for scheduling the packet, the packet waits in the input buffers memory until a core becomes available. In the example in Figure 4, we denote this situation with a striped pattern in point C, to indicate that the packet is sitting on a queue waiting to be scheduled. The input buffer occupancy is thus equal to the time the packet spends in the queue plus the time it spends while being processed. We denote the maximum size of these queues with Q. For example, *Core 0* has $Q = 0$ at B, and $Q = 1$ at C. We model and analyze this in detail in Section 5.

4.3 Working Memory

We mentioned that the memory is partitioned among multiple allreduces, and in the example, we assume three buffers have been allocated to this specific reduction. For simplicity, we also assume each block is aggregated on a single buffer. However, multiple concurrent buffers per block could be used, for example, to reduce contention, and in Section 6 we describe different possibilities for organizing the working memory. To avoid running out of memory, each host can have a number of *"in-flight"* blocks not larger than the number of aggregation buffers assigned to that allreduce. In our example, the hosts send the fourth block only after the first block has been fully reduced and the buffer has been released. We can use Little's Law [59] to determine how many aggregation buffers should be allocated to each allreduce. Because we have P packets per block (three in this case, equal to the number of hosts), the target bandwidth (in blocks per cycle), can be simply defined as $\mathscr{B}/P$. We define with $\mathscr{L}$ the latency (in cycles) to process a block (in our example, the time between A and B in Figure 4), and with M the number of buffers needed to aggregate a block. Then, each allreduce needs a working memory (in number of buffers) equal to $\mathscr{R} = M\frac{\mathscr{B}}{P}\mathscr{L}$.

5 PACKETS SCHEDULING AND INPUT BUFFERS OCCUPANCY

By default, packets are scheduled to the cores with a *First Come First Serve* (FCFS) policy, so that they are evenly distributed across the cores. To simplify the exposition, we also assume that we size the system so that the *interarrival time* to the processing unit (i.e., the time between the reception of two subsequent packets) is larger or equal than its *service time* (i.e., the time between the sending of two subsequent packets). Under these conditions, on average packets will never be enqueued because they will always find an available core. In general, however, packets might be enqueued, waiting for a core to become available. When the queue is full, the packet is dropped or congestion is notified before filling the queue, depending on the specific network where the switch is integrated.

In PsPIN the L1 memory of the switch is partitioned across multiple clusters of cores (Figure 2). This means that each of the aggregation buffers shown in Figure 4 is allocated on the L1 memory of a specific cluster. For example, if we assume to have two cores per cluster, and that *Buffer 0* is allocated on the cluster of *Cores 0* and *1*, then the handler running on *Core 2* would need to access a remote L1 memory. By doing so, it incurs a higher latency (up to 25x higher [13]) compared to accessing its local L1 memory.

To only have local L1 accesses and improve performance, we restrict the processing of packets belonging to the same block to a subset of cores (located on the same cluster). To keep the workload balanced among all the available cores, thus still guaranteeing line-rate processing, we adopt hierarchical FCFS scheduling. We assign packets belonging to the same block with an FCFS policy to the same subset of cores, and different blocks to different subsets[4]. Even if in the long run this evenly distributes the packets to the cores, it might generate short bursts of packets directed to the same core(s), that need to be enqueued, thus increasing the occupancy of the packets buffers memory. We analyze this more in detail in Section 5.

To understand the impact of scheduling decisions on the input buffer occupancy, we illustrate in Figure 5 three different scenarios (A, B, and C), and we report in Table 2 some variables we use in our analysis. In scenario A packets are received from four ports (we suppose there is a host attached to each of them), and packets with the same color belong to the same block and must be aggregated together. The number inside each packet represents the time when it is received by the switch. Because each packet might spend some

[4]The identifier of the block can be carried in the packet as an IP optional header, processed by the *parser* and communicated to the packet scheduler.

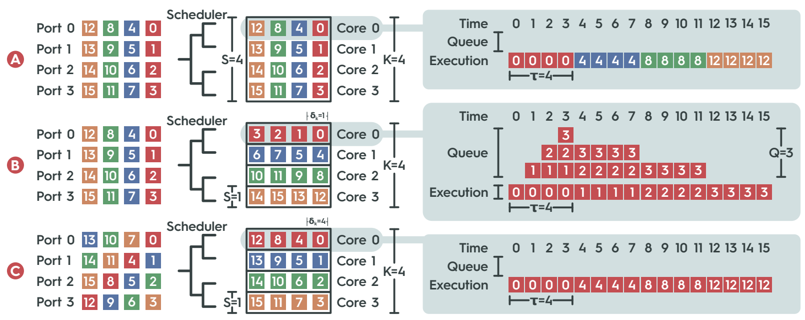

Figure 5: Impact of intra-block interarrival time and hierarchical FCFS scheduling on packets memory occupancy.

Par.	Description
K	Number of cores in the switch.
S	Number of cores in each scheduling subset.
P	Number of packets received for each block (this is equal to the number of children of the switch in the reduction tree).
$\mathbb{Q}$	Maximum number of packets in the switch.
δ	Average interarrival time of packets.
δ_c	Average interarrival time of packets belonging to the same block.
δ_k	Average interarrival time of packets to a core (during a burst).
τ	Average service time of a core.
M	Memory occupied by a block (number of elements).

Table 2: Parameters used in the scheduling modeling.

time in the queue this is, in general, different from the time when a core starts to process it. Due to congestion and unbalance, arrival times are not so nicely aligned, and there might be gaps between the reception of subsequent packets. However, for illustrative purposes, we assume the simplest case where all packets are received at a steady and constant rate of one packet per second. We assume that the switch has K cores (4 in this case), P ports, that it receives one packet every δ seconds (1 in this case), and that each core has a service time τ (4 seconds in the example). Because the service time of the switch is $\frac{\tau}{K} = \delta$, if packets are evenly distributed the switch can process the packets at line rate. We also define δ_c as the interarrival time of the packets *within a block* ($\delta_c = 1$ on scenario Ⓐ). On the right part of the figure we show a detail of what happens in *Core 0*. Because it receives a packet every 4 seconds, and its service time is 4 seconds, packets are never enqueued.

On scenario Ⓑ of Figure 5, packets belonging to the same block, are instead assigned to a subset of cores (each containing one core, i.e., $S = 1$). Each core now does not receive a steady flow of packets, but bursts of packets at regular intervals. Whereas on scenario Ⓐ each core receives one packet every 4 seconds, on scenario Ⓑ each core receives 4 packets in 4 seconds, and then nothing for 12 seconds. Even if in both Ⓐ and Ⓑ each core receives on average 4 packets every 16 seconds, on scenario Ⓑ the bursts build up queues in front of each core. On the right part of the figure, we show in detail what happens in *Core 0*. The core receives 4 packets in 4 second, thus building a queue, that will eventually be completely absorbed before the beginning of the next burst (because we assumed to size the system to process packets, on average, at line rate). However, these queues increased the time a packet spent in the switch and, thus, the input buffer occupancy.

Moreover, the intensity of the bursts (and thus the queue length) does not only depend on the size of the subsets S, but also on δ_c. For example, in scenario Ⓒ of Figure 5 we show the same scenario as Ⓑ, but now with $\delta_c = 4$, that implies that packets belonging to the same block will arrive four seconds apart one from on another. By comparing Ⓒ with Ⓑ we can observe that although packets arrive at the same rate (one packet per second in both cases), and the scheduler assigns the packets to the same subset of cores, on scenario Ⓒ packets of the same block arrive at each core at a slower rate and are never enqueued, as we can observe in the detail of *Core 0*. In this scenario, we obtain the same locality as in scenario Ⓑ (packets belonging to the same block are assigned to the same subset of cores), and the minimal input buffer occupancy as in scenario Ⓐ.

δ_c depends on non-controllable factors such as application imbalance [60, 61], network noise [62–66], OS noise [67, 68], and network contention [69–74], but also on some controllable factors, such as the order the packets are sent by the hosts. In this work, we propose a solution called *staggered sending*, that consists in having each host sending the packets in a different order so that, on average, packets belonging to the same block can be scheduled to a specific subset of cores, while not increasing the size of the queues and, thus, the input buffer occupancy. Moreover, as we show in Section 6.1, *staggered sending* is also helpful in reducing contention on the shared aggregation buffer. In general, the maximum δ_c we can induce with staggered sending depends on the number of blocks to be sent. In the example in scenario Ⓒ, if we would have only 2 blocks, the δ_c would be half of that we have when having 4 blocks. In general, we have $\delta \leq \delta_c \leq \delta \frac{Z}{N}$.

Input buffer occupancy. Because we are considering bursty arrival times at the cores, we can't use Little's Law [59] to compute the average number of packets in the switch, because it would consider an average case, and would not capture differences between the three scenarios. Instead, we first compute the interarrival time of the packets in a burst to a specific core, that we indicate with δ_k. Packets arrive to a subset of S cores with an interarrival time δ_c, and thus to each of the cores in the subset with interarrival $\delta_k = S\delta_c$. Because we on the long run packers are evenly distributed across the cores, this can never be higher than $K\delta$, and thus we have $\delta_k = \min(S\delta_c, K\delta)$. Because P packets are received for each block, a burst can contain up to $\frac{P}{S}$ packets, and it takes $\delta_k \frac{P}{S}$ cycles to completely receive the burst. By that time, $\frac{\delta_k P}{\tau S}$ packets have been processed by the core and removed from the queue. Accordingly, we can express the maximum queue length as $Q = \frac{\delta_k P}{\delta_k S} - \frac{\delta_k P}{\tau S} = \frac{P}{S}\left(1 - \frac{\delta_k}{\tau}\right)$, and the maximum number of packets in the switch (including those being currently processed by each core) as:

$$\mathbb{Q} = (Q + 1)K = \frac{PK}{S}\left(1 - \frac{\delta_k}{\tau}\right) + K \qquad (1)$$

This equation shows the relationship between the scheduling decision and the input buffer occupancy (e.g., the smaller S, the higher the input buffer occupancy). It can also be used to compute the latency $\mathscr{L}$ to process a block and, thus, the working memory occupancy (Section 4.3). Indeed, the latency can be computed as the time the switch waits for all the packets of the block to be received ($(P-1)\delta_c$), plus the time needed for processing the last packet. This includes both the time spent aggregating the packet and the time spent in the queue. In the worst case, a packet spends $Q\tau$ cycles in the queue, and the latency is equal to $\mathscr{L} = (P-1)\delta_c + (Q+1)\tau$. We use the queueing time and the latency models for estimating the input buffer and working memory occupancy in Section 6.

6 PARALLELISM AND MEMORY ORGANIZATION

We now describe the design of different aggregation approaches that can be used to reduce a block of P packets in Flare. We consider different designs: aggregation on a single memory buffer shared by all the packets of a block (Section 6.1), aggregation on multiple buffers (Section 6.2), and asynchronous tree aggregation (Section 6.3). We then analyze the different tradeoffs between these alternatives in Section 6.4.

When analyzing the different aggregation approaches we model the service time of a core τ (Section 4.1). This can then be used to model the size of the working memory (Section 4.3); and the input buffer occupancy (Equation 1). To model the working memory size, we also need to model M, i.e., the number of buffers used for each block. In our modeling, we assume to have 1KiB packets containing 256 floating-point values. We measured the time required to aggregate a packet in the aggregation buffer by using the PsPIN cycle-accurate simulator. On average, a core of the PsPIN unit needs four cycles to sum two 4-bytes floating point values and to store the result back in the aggregation buffer. Because the unit is clocked at 1GHz (Section 3), the time required to process a packet is 1ns per byte circa, and we use this information to model τ.

6.1 Aggregation using a single buffer

The first approach we propose is the most straightforward one, where all the packets of the same block are accumulated in the same working memory buffer, as shown in Figure 6. We show on the left the packets processed by each core, and how the cores access the buffer. In this case, cores *C0*, *C2*, and *C3* just sum the content of their packets into the buffer. Cores *C1* also reads back the fully aggregated result, that will then send on a packet over the network. On the right, we show the timeline of these operations. For the cores, the boxes represent the duration of the handlers, whereas for the buffer we depict for how long the buffer has been used.

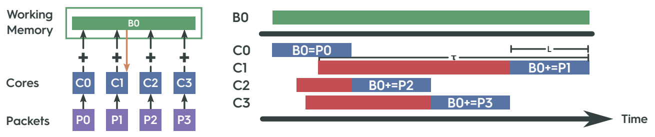

Figure 6: Single buffer aggregation.

We assume the most general case where the concurrent aggregation of multiple packets in a shared aggregation buffer is executed in a critical section. Indeed, even if in principle we could use atomic operations [75], the computation would still be affected by severe contention and/or performance overhead [76, 77]. Moreover, by assuming a critical section we cover the cases where the user function cannot be executed by using atomic operations, and the aggregation of sparse data that, as we will show in Section 7, requires more complex processing and in most cases needs to be executed anyhow in a mutually exclusive way.

To avoid expensive context switches, PsPIN handlers are never suspended and terminate only after the packet has been processed. Thus, if waiting to enter a critical section, they will actively consume compute cycles on one of the cores. For example, when a packet arrives at *C2*, the core waits for the buffer to be released by *C0* (we denote this waiting time with a red box), and only then it starts the aggregation. We define with L the number of cycles needed to aggregate a block after the handler enters the critical section (i.e., in general, $L \leq \tau$). The probability that two handlers need to access the same aggregation buffer concurrently depends on δ_c, i.e., the interarrival time of the packets belonging to the same block. If $\delta_c \geq L$, on average there is never more than one packet of the same block processed concurrently. This is also true if all the packets of a block are executed by the same core (i.e., $S = 1$). However, for $S = 1$ the input buffers occupancy would significantly increase, as discussed in Section 5.

To estimate the cost of contention, we assume to have C cores in each cluster. Because we schedule the packets belonging to the same block to a subset of C cores, in the worst case we have C concurrent packets of the same block that need to access the same aggregation buffer in the working memory. If a handler needs L cycles to aggregate the packet (excluding the time spent waiting to enter the critical section), in the worst case, the first handler to be executed needs L cycles to aggregate the packet, the second handler $2L$ cycles, the third handler $3L$ cycles, etc…In general, the average service time of a core can thus be expressed as $\tau = \frac{\sum_{i=1}^{C} iL}{C} = \frac{L(C-1)}{2}$.

We then have:

$$\tau \leq \begin{cases} L, & S = 1 \text{ or } \delta_c \geq L. \\ \frac{L(C-1)}{2}, & \text{otherwise.} \end{cases} \tag{2}$$

We minimize τ for $S = 1$ or $\delta_c \geq L$. As discussed in Section 5, we can change S by restricting the execution of the packets of the same block to a subset of S cores. According to equation 1, by reducing τ we would decrease the input buffers occupancy. However, this increase might be canceled out because we are also decreasing S. On the other hand, by increasing δ_c we always decrease the input buffer occupancy, but we increase the latency required for processing a block and, thus, the working memory occupancy.

Before showing the effect of these decisions on both bandwidth and memory occupancy, we observe that we can increase δ_c by using *staggered sending*, as discussed in Section 5. In a nutshell, this means that to avoid contention there should never be two cores working on two packets of the same block. This can only happen if the data to be reduced is large enough to have, at any time, one block per core. For our switch, this can only be guaranteed if the hosts use staggered sending and if the size of the data to be reduced is larger than 512KiB. Because small-size allreduces are a significant fraction of the allreduce traffic [2], we propose in the next sections some alternative approaches to address such cases. Moreover, because we only need one buffer per block, we have $M = 1$.

6.1.1 Main insights. Figure 7 illustrates the modeled bandwidth, memory occupancy of the input buffers ($\mathcal{Q}$), and occupancy of the working memory ($\mathcal{R}$). On the x-axis, we have different values of S. We consider restricting the execution of packets belonging to the same block to 1 core, and C cores (i.e., all the cores of a cluster).

First, we observe that for small messages (where we can't sufficiently increase δ_c) there is a large performance drop when $S = C$. However, having $S = 1$ significantly increases the memory occupancy. For larger messages, we observe good performance also when scheduling the packets on C cores of the same cluster, thus decreasing the queue length and the occupancy of the input buffers memory. The occupancy of the working memory is negligible and around 512KiB. For large enough packets and $S = C$, the maximum total memory occupancy is 2MiB for a single reduction. In general, scheduling the packets belonging to the same reduction block to a single core significantly increases the memory occupancy and, for this reason, in the following we design alternative solutions that can achieve a higher bandwidth and a lower memory occupancy for smaller allreduces.

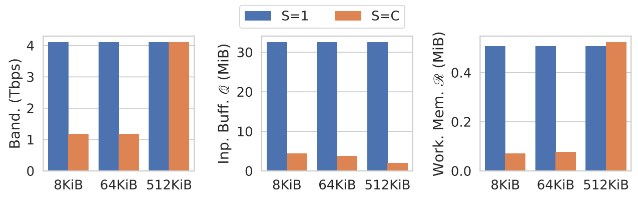

Figure 7: Bandwidth and memory occupancy of single buffer aggregation.

6.2 Aggregation on multiple buffers

An extension to the previous aggregation design consists in having B aggregation buffers for each block. In Figure 8 we show an example with two buffers. When a handler is executed, it takes whichever of those buffers is not currently used by any other handler. In the example, differently from the single buffer case, when *C2* receives the packet, it finds *B0* already being used by *C0*, and thus aggregates the packet on *B1*. If no buffer is available, the handler needs to wait to enter a critical section, as for the single buffer aggregation (for example, this is what happens to both *C1* and *C3*). Because the aggregated data is now distributed over B buffers, the last handler to be executed for the block needs to aggregate together the partially aggregated data contained in the remaining $B - 1$ buffers. In the example, the handler running on *C1* is the last one to be executed and, aggregates the content of its packet with the content of *B0*, and then of *B1*.

τ can be computed starting from Equation 2, by replacing δ_c with $B\delta_c$. Indeed, when using B buffers, the probability that two running handlers need to access the same buffer decreases proportionally with B. Compared to the single buffer aggregation, this solution is less affected by contention for shorter interarrival times (and, thus, for smaller data). Moreover, the last handler needs to aggregate the

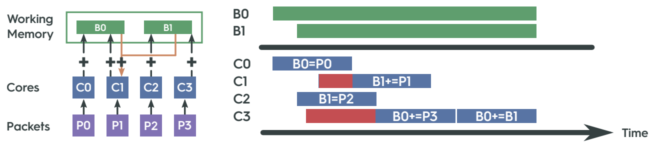

Figure 8: Multiple buffer aggregation.

data from the other $B - 1$ buffers, and this costs $(B - 1)L$ additional cycles. Because B buffers are stored for each block, we have $M = B$.

6.2.1 Main insights. This aggregation algorithm increases the concurrency and the bandwidth at the cost of higher memory occupancy. However, using too many buffers reduces the bandwidth, due to the extra cost required for sequentially aggregating the partially reduced data of each buffer. We analyze the tradeoffs between bandwidth and memory occupancy in Section 6.4.

6.3 Tree aggregation

The main drawback of the multiple buffer aggregation, is the sequential aggregation of the data contained in each of the aggregation buffers. To avoid that cost, we can organize the computation in a tree-like way. We illustrate this idea in Figure 9. When a core receives a packet, it allocates a buffer and copies the data into it. After copying the data, a core might also aggregate the data contained in some of the other buffers. The idea is to organize the aggregation of those buffers in a pre-defined tree-like way. For example, the data contained in *B0* can only be aggregated with *B1*, and the result stored back in *B1* (after this aggregation step, *B0* can be deallocated). Similarly, the data contained in *B2* can only be aggregated with *B3* and stored back in *B3* (and *B2* deallocated). When *B1* contains the result of *B0+B1*, and *B3* the result of *B2+B3*, the final aggregation of *B1+B3* can be computed.

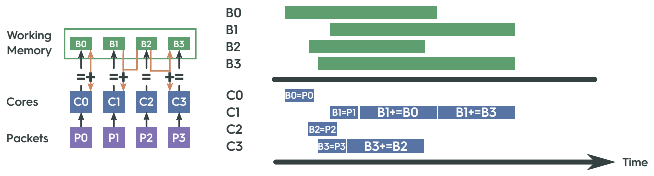

Figure 9: Tree aggregation.

To completely avoid contention on shared aggregation buffers (independently from δ_c and, thus, from the message size) the computation on the next level of the tree is carried only if a core finds available data in both buffers. For example, when *C0* receives the packet, it copies it in *B0*, and could aggregate *B0* with *B1*. However, there is no available data yet in *B1*, and the handler is terminated. Instead, when *C1* processes its packet, it finds the data in *B0* and carries on the computation of the next level of the tree. It also finds the result of *B2+B3* in *B3*, and thus also computes *B1+B3*, and sends the completely aggregated result on the network. To guarantee the reproducibility of floating-point summation, a packet coming from a port i is always stored in a buffer j. This forces the tree to always have the same structure and guarantees the reproducibility of floating-point summation across different runs.

To compute the service time, we assume that the cost of copying the data is negligible. For example, in PsPIN this can be done through the DMA engine, at the cost of 64 cycles instead of the 1024 cycles needed for the aggregation. A total of $P - 1$ aggregations are executed, and we can compute the average number of cycles per packet as $\tau = \frac{(P-1)L}{P}$. To compute the memory consumption, we observe that each time two buffers are aggregated, one of them is discarded. Because $P - 1$ aggregations are performed and they are arranged in $\log P$ levels in the tree, on average we have $M = \frac{P-1}{\log P}$ active buffers for each block.

6.3.1 Main insights. Differently from single and multiple buffers aggregation, in this case, the handlers never waste cycles waiting for entering a critical section, and this design achieves the optimal bandwidth independently from δ_c and, thus, from the data size. However, it requires more buffers per block, increasing the average memory occupancy. We analyze the bandwidth, latency, and memory occupancy of this design in Section 6.4.

6.4 Evaluation

In Figure 10 we show the modeled maximum bandwidth and the memory occupancy, for $S = C$ and different data sizes. We observe that the only best performing algorithm on data smaller than 128KiB is the tree aggregation. When increasing the data size, multi-buffers aggregation catches up, and the higher the number of buffers, the higher the bandwidth for smaller messages. Eventually, for data larger than 512KiB, single buffer aggregation catches up with the other solutions.

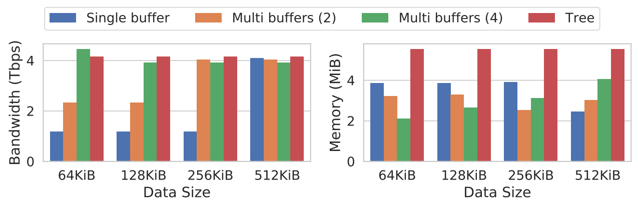

Figure 10: Modeled bandwidth and memory occupancy for $S = C$ and different data sizes.

Regarding the memory occupancy, it is worth remarking that, in some cases, using more buffers decreases the memory occupancy, because the performance is higher and those buffers are used for a shorter time. To optimize both compute and memory resources, Flare uses single buffer aggregation if the size of the data to be reduced is larger than 512KiB, multi buffers with 4 buffers if larger than 256KiB, with 2 buffers if larger than 128KiB, and tree aggregation otherwise. When reproducibility of floating-point summation is required, Flare always uses tree aggregation.

We implemented the different aggregation algorithms in the PsPIN cycle-accurate simulator [13]. To simulate delays in the hosts sending the data and in the network, we generate packets with a random and exponentially distributed arrival rate. The actual PsPIN implementation [13] only simulates 4 clusters. Because the clusters are organized in a shared-nothing configuration, we scale the results linearly with the number of deployed clusters. First, we report in Figure 11 the maximum bandwidth that the switch can achieve for the aggregation of 32-bits integers vectors of different size Z. We compare Flare to two baselines: SwitchML [11] and SHARP [9, 16]. SwitchML runs on programmable Tofino switches [27], can only process integer elements, and achieves a maximum bandwidth of 1.6Tbps [11]. SHARP is a solution running on Mellanox's fixed-function switches [78], and can process floating-point elements. Switches supporting SHARP have 40 ports at 200Gbps. However, to the best of our knowledge, the best available known data for SHARP (for a single switch) shows a 3.2Tbps bandwidth [16] (32 ports at 100Gbps), and we use this as a reference.

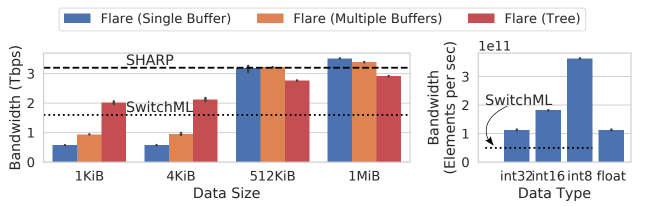

Figure 11: Bandwidth for different data size and data types.

We observe that, for small data, only tree aggregation provides higher bandwidth than SwitchML. Indeed, single and multi-buffers aggregation cannot exploit staggered sending for small data, and experience contention when accessing the aggregation buffers. Moreover, for small data, we are showing a *"cold start"* case, where the handlers were not loaded yet in the instruction cache. For larger data, single buffer aggregation efficiently exploits staggered sending, achieving a higher bandwidth compared to multi-buffer and tree aggregation. Indeed, both multi-buffer and tree aggregation have some additional overhead caused by the management of multiple buffers. We also observe that even Flare achieves a higher bandwidth compared to SwitchML and SHARP, while also capable of running arbitrary computations on network packets. As we show in Section 7.1 this leads, for example, to significant performance improvements when dealing with sparse data.

We also report in Figure 11 the bandwidth (in elements aggregated per second), for different data types, for the aggregation of 1*MiB* data. We consider 32-bits, 16-bits, and 8-bits integers, and floating-point elements (not supported by SwitchML and by existing programmable switches in general). Programmable switches can only process a fixed number of elements per packet. Processing more elements per packet would require more *recirculations*, thus decreasing the bandwidth accordingly. On the other hand, in Flare the HPUs of the PsPIN unit use vectorization and can aggregate, for example, two `int16` elements in a single cycle. This leads to an increase in the number of elements aggregated per second when sending elements with a smaller data type. However, differently from SHARP, Flare currently does not support the aggregation of double-precision floating-point elements.

7 SPARSE DATA

To reduce the network traffic and increase the performance, applications dealing with sparse data might send only the non-zero elements and their positions. In this case, each host splits the data into blocks so that, on average, each block contains a number of non-zero elements that would fit in a network packet. This implies that different hosts might have a different number of elements in each block, as shown in Figure 12. We show both the partitioning of the data in reduction blocks and the elements carried by each packet (packets also carry the position of each element inside the block, not shown in the figure for the sake of clarity). The number inside each box represents the value of that specific element. For simplicity, we assume that (due to the MTU) the hosts can send at most two elements per packet and that the data *density* (defined as the average percentage of non-zero elements in each reduction block) is 50%. Thus, on average, there will be 2 non-zero elements every 4 elements, and thus we set the span of the block to 4 elements.

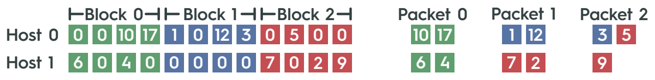

Figure 12: Packetization of sparse data. Packets also carry the position of the elements (not shown).

However, whereas this is true on average, this introduces the following additional challenges compared to the dense case:

Multiple blocks per packet. Elements that belongs to two different blocks would be sent in the same packet (like the elements 3 and 5 in Figure 12). This implies that the handler must check, for each element in the packet, the block it belongs to (to aggregate it in the correct aggregation buffer), for example by dividing its index by the number of blocks. This introduces some additional computation for each element. To avoid this additional overhead, we send the block identifier in the packet header (as for the dense case), and we force the hosts to never send multiple blocks in the same packet, and rather to send a packet even if it is not full (in our example, a packet with only 3).

Block split. Elements belonging to the same block could be split in two different packets (e.g., elements 7 2 and 9 in Figure 12). For dense reductions, it was sufficient to keep a *children* counter, incremented when a packet was received from a child, and consider the reduction of the block completed when the counter was equal to the number of children. In the sparse case, a node could receive multiple packets for the same block, and this approach is not sufficient anymore. To address this issue each host sends, in the last packet of a block, a counter representing the number of packets composing that block. For example, because node 1 splits block 2 in two packets, it sends the packet carrying element 9 with the counter set to 2. The handlers keep for each block an additional *shard* counter for each of the P hosts. The *shard* counter is incremented every time a packet for that block is received from that port. When the counter is equal to the value carried in the last packet of the block, then all the packets for that block and from that specific port have been received, and the *children* counter can be increased.

Empty blocks. In some cases, we could have all-zero blocks (for example, block 1 sent by node 1). In those cases, we still send a packet with no elements (and with only the header with the identifier of the block), so that the switch can increase the children counter nevertheless. If we assume uniformly distributed zero values, this however should rarely happen.

Another difference compared to the dense case is the design of the data structure that holds the partially aggregated data. Flare still aggregates separate blocks in different buffers. However, whereas for the dense case the aggregation buffer has size N, in the sparse case aggregating two buffers of N elements leads, in general, to more than N aggregated elements (because the non-zero elements in the two buffers might not perfectly overlap). For example, in Figure 12, aggregating the data in the packet 10 17 with the data in the packet 6 4 leads to 3 elements (6 14 17).

For sparse data, Flare stores the data and the indexes in a hash table. To avoid expensive collision resolution, when there is a collision, the colliding element is put in a *spill buffer*. When the spill buffer is full, the spilled data is immediately sent to the next switch (or to the hosts). For highly sparse data, this solution reduces memory consumption compared to storing it in a dense format. For denser data, this however increases both the memory occupancy (because indexes must be explicitly stored), the latency when aggregating the data, and the network traffic (because more data spills).

For denser data, Flare uses a contiguous memory buffer of the size of the block. From a computational perspective, this is the design with the lowest latency, because the handler simply needs to store the element in a specific position. However, when the reduction is completed, the buffer needs to be entirely scanned and only the non-zero elements inserted in the packet. Moreover, the memory consumption will be equal to that of the dense case, thus much higher than the optimal if the buffer contains many zeros.

In general, sparse data get denser after each aggregation and, when aggregating data on an in-network reduction tree, the data get denser while traveling from the hosts to the root of the tree. For this reason, Flare stores the data in hash tables in the leaves switches, and in an array in the root switch. The *"densification"* of sparse data depends on the number of non-overlapping indexes, and we analyze these effects at scale in Section 7.1.

We can distribute the computation to the cores using the same solutions we described in Section 6 for the dense case: single buffer, multiple buffers, and tree aggregation. We can also use the same models, with a few caveats. First, for sparse allreduce, the term P (i.e., the number of packets expected for a given block) represents the average case because, as described earlier, each block can be composed of more or less than P packets. Moreover, storing N elements might require more than N memory cells, depending on the data structure used to store the partially aggregated data.

7.1 Evaluation

First, we report the modeled bandwidth for both the hash and array storage in Figure 13, assuming a 10% density. We observe a lower bandwidth for the sparse allreduce compared to the dense one, due to the more complex operations that need to be executed by the handler to store both the indexes and the data. However, as we will show later, because less data is transmitted, this still allows to obtain a performance advantage compared to both host-based and in-network allreduce.

Then, we report the results for a 1MiB allreduce in Figure 14, analyzing the performance when using both the hash and the array storage, and for different data *density*. Deep learning applications

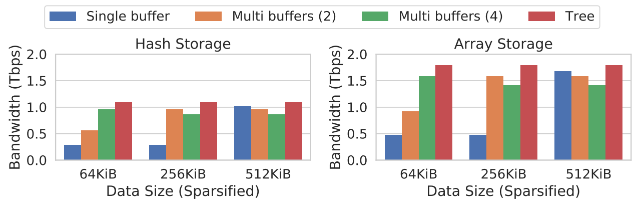

Figure 13: Modeled bandwidth for the FLARE sparse allreduce.

typically sparsify the data by only communicating the top 0.1% or 1% elements of the original data [39], whereas some graph processing algorithms might retain up to 20% of the data [5, 6]. We observe that, as expected, storing the data as a contiguous array increases the bandwidth and the memory occupied by each block compared to using a hash table. We do not report the data about array storage with 1% density because this requires a 600KiB array for each block (with, on average, only 6KiB non-zero elements) and all the concurrently processed blocks do not fit in FLARE memory. Hash table storage is characterized by a constant bandwidth and memory occupancy independently from the density. Indeed, when data is stored in a hash table FLARE always executes a number of instructions that only depend on the size of the packet. On the other hand, for lower density the array storage requires storing a larger array, thus increasing the number of cycles needed to flush it when the reduction is over.

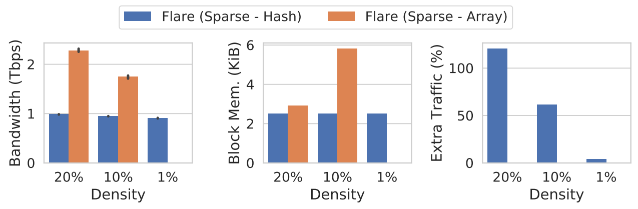

Figure 14: Simulated bandwidth, memory per block, and extra generated traffic for the FLARE sparse data allreduce, for different density and storage types.

However, to avoid costly collision management, with hash storage when there is a collision the element is put in a spill buffer, that when full is sent over the network. This leads to extra network traffic, because data is split between the hash table and the spill buffer, and the switch might not fully aggregate the data. We report the impact of spilling on the network traffic in Figure 14. For example, we observe that for 20% data density, spilling doubles the network traffic. On the other hand, array storage never generates extra traffic, and packets are only sent when the block has been fully reduced.

To analyze the performance of the FLARE sparse allreduce at scale, we extended the SST simulator [79] so that the switch can modify in-transit packets, and we implemented both dense and sparse FLARE in-network allreduce, and also the host-based ring allreduce. Moreover, we also implemented on top of SST the SparCML host-based sparse allreduce [39]. We then tuned the simulator parameters so that the bandwidth of the switches matches with that obtained through the cycle-accurate PsPIN simulator. Both the PsPIN-based simulator and the extended SST simulator are provided in the artifact.

As we discussed in Section 7, FLARE performance depends on the amount of overlapping indexes. To simulate a realistic scenario, we gathered the data exchanged during a Resnet50 [80] training iteration executed on 64 nodes using SparCML, and we send the same data in SST. Each host works on a 100MiB vector of floating point values. For sparse allreduces, the data is split in buckets of 512 values, and one single value is sent for each bucket (~0.2% density). We reproduce the same data on a simulated 2-level fat tree network [81] built with 8-port 100Gbps switches, connecting 64 nodes with 100Gbps network interfaces. We report in Figure 15 the results of our analysis, by showing the time the hosts need to complete the allreduce and the total number of bytes that traversed the network.

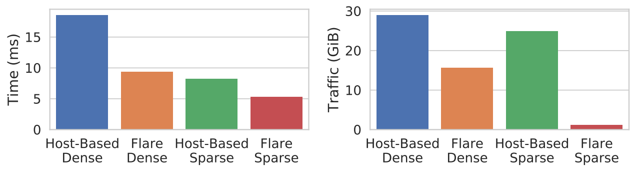

Figure 15: Execution time and network traffic for a 64 nodes allreduce executed on a simulated 2-levels fat tree network. The data to be reduced contains the gradients exchanged during a sparsified ResNet50 training iteration.

First, we observe that the in-network dense allreduce leads to more than 2x speedup compared to the host-based dense allreduce, and to a 2x reduction in the network traffic. We also observe that the host-based sparse allreduce prodives slightly better performance than the in-network dense allreduce. However, it also generates more traffic because, even if less data is sent by the hosts, that data traverses more hops. Eventually, we observe that the in-network FLARE sparse allreduce provides further advantages compared to both host-based sparse allreduce and in-network dense allreduce, both in terms of performance and network traffic. This leads to performance improvement up to 35% and traffic reduction up 20x compared to the SparCML host-based sparse allreduce, and to performance improvement up to 43% and traffic reduction up to 13x compared to the in-network FLARE dense allreduce.

8 DISCUSSION

Support for other collectives. Although we considered in this work the allreduce collective operation, other collectives like *reduce*, *broadcast*, and *barrier* can also be accelerated with FLARE. For example, a *barrier* can simply be implemented as an in-network *allreduce* with 0-bytes data. Moreover, in some frameworks like Horovod [82] each process might have multiple outstanding allreduce operations, and each rank might issue those operations in a different order, potentially leading to deadlock [83]. To address this problem, an additional synchronization step between the ranks is required, and this step could be offloaded to FLARE as well, together with the allreduce operation.

Limitations. As we shown in Section 7.1, the bandwidth achieved by the in-network sparse allreduce is lower than its dense counterpart, due to the higher latency required for processing and aggregating the sparse data. In our experiments, the lower bandwidth was compensated by the reduction in the amount of processed data, and the Flare in-network sparse allreduce outperformed the SparCML host-based sparse allreduce. However, we believe that there is still space for improvement, either by optimizing the handlers code or by introducing hardware support to optimize indirect memory accesses [84].

9 CONCLUSIONS

In this work, we introduced Flare, an architecture for flexible in-network data reduction. Flare is based on the open-source PsPIN RISC-V architecture, and allows running custom packet processing handlers on network packets. Flare also includes a set of aggregation algorithms with different performance and memory occupancy tradeoffs. We modeled and analyzed in detail each of these algorithms, in terms of bandwidth and memory occupancy. We then implemented the aggregation algorithms in the PsPIN cycle-accurate simulator, and analyzed their performance for different allreduce sizes and datatypes, comparing them with state-of-the-art in-network aggregation approaches such as SHARP and SwitchML. Moreover, we also designed and implemented the first (to the best of our knowledge) in-network sparse allreduce algorithm. We implemented the algorithm in the PsPIN and SST simulators, showing performance improvements and network traffic reduction compared to in-network dense allreduce and host-based sparse allreduce.

ACKNOWLEDGMENTS

This work has been partially funded by the EPIGRAM-HS project under grant agreement no. 801039, and the European Project RED-SEA under grant agreement no. 955776. Daniele De Sensi is supported by an ETH Postdoctoral Fellowship (19-2 FEL-50).

REFERENCES

[1] Message Passing Forum. Mpi: A message-passing interface standard. Technical report, USA, 1994.

[2] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. Characterization of mpi usage on a production supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[3] Steven Gottlieb, W. Liu, William D Toussaint, R. L. Renken, and R. L. Sugar. Hybrid-molecular-dynamics algorithms for the numerical simulation of quantum chromodynamics. *Physical review D: Particles and fields*, 35(8):2531–2542, 1987.

[4] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4), August 2019.

[5] H. Zhao and J. Canny. Kylix: A sparse allreduce for commodity clusters. In *2014 43rd International Conference on Parallel Processing*, pages 273–282, 2014.

[6] Huasha Zhao and John F. Canny. Sparse allreduce: Efficient scalable communication for power-law data. *CoRR*, abs/1312.3020, 2013.

[7] Torsten Hoefler, Andrew Lumsdaine, and Jack Dongarra. Towards efficient mapreduce using mpi. In Matti Ropo, Jan Westerholm, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 240–249, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[8] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117 – 124, 2009.

[9] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *Proceedings of COM-HPC 2016: 1st Workshop on Optimization of Communication in HPC Runtime Systems - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10. Institute of Electrical and Electronics Engineers Inc., jan 2017.

[10] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. An in-network architecture for accelerating shared-memory multiprocessor collectives. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 996–1009. IEEE Press, 2020.

[11] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[12] Dennis Abts, Michael R. Marty, Philip M. Wells, Peter Klausler, and Hong Liu. Energy proportional datacenter networks. *SIGARCH Comput. Archit. News*, 38(3):338–347, June 2010.

[13] Salvatore Di Girolamo, Andreas Kurth, Alexandru Calotoiu, Thomas Benz, Timo Schneider, Jakub Beranek, Luca Benini, and Torsten Hoefler. A risc-v in-network accelerator for flexible high-performance low-power packet processing. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[14] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[15] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E. Grant, and Ron Brightwell. Spin: High-performance streaming processing in the network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Richard L. Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, Ami Marelli, Valentin Petrov, Evyatar Romlet, Yong Qin, and Ido Zemah. Scalable Hierarchical Aggregation and Reduction Protocol (SHARP)TM Streaming-Aggregation Hardware Design and Evaluation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12151 LNCS, pages 41–59. Springer, jun 2020.

[17] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. Cray xc series network. *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[18] Y. Ajima, T. Kawashima, T. Okamoto, N. Shida, K. Hirai, T. Shimizu, S. Hiramoto, Y. Ikeda, T. Yoshikawa, K. Uchida, and T. Inoue. The tofu interconnect d. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 646–654, 2018.

[19] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony. The percs high-performance interconnect. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 75–82, 2010.

[20] B. Arimilli, Bernard C. Drerup, Paul F. Lecocq, and Hanhong Xue. Collective acceleration unit tree structure, U.S. Patent US8756270B2, 17/06/2014.

[21] J. P. Grossman, B. Towles, B. Greskamp, and D. E. Shaw. Filtering, reductions and synchronization in the anton 2 network. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 860–870, 2015.

[22] Nadeen Gebara, Paolo Costa, and Manya Ghobadi. Panama: In-network aggregation for shared machine learning clusters. In *Conference on Machine Learning and Systems (MLSys)*, April 2021.

[23] Shuo Liu, Qiaoling Wang, Junyi Zhang, Qinliang Lin, Yao Liu, Meng Xu, Ray C. C. Chueng, and Jianfei He. Netreduce: Rdma-compatible in-network reduction for distributed dnn training acceleration, 2020.

[24] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[25] Jiawei Fei, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Amedeo Sapio. Efficient Sparse Collective Communication and its application to Accelerate Distributed Deep Learning. In *Proceedings of SIGCOMM'21*, Aug 2021.

[26] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, August 2013.

[27] Intel. Intel Tofino Series. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html, mar 2021.

[28] Recep Ozdag Intel. Intel (R) Ethernet Switch FM6000 Series – Software Defined Networking. https://people.ucsc.edu/ warner/Bufs/ethernet-switch-fm6000-sdn-paper.pdf, mar 2021.

[29] R. Bifulco and G. Rétvári. A survey on the programmable data plane: Abstractions, architectures, and open problems. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–7, 2018.

[30] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.
[31] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with p4: Fundamentals, advances, and applied research, 2021.
[32] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends, 2021.
[33] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 209–215, New York, NY, USA, 2019. Association for Computing Machinery.
[34] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 67–82, Boston, MA, March 2017. USENIX Association.
[35] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks, 2021.
[36] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.
[37] Nikoli Dryden, Sam Ade Jacobs, Tim Moon, and Brian Van Essen. Communication quantization for data-parallel training of deep neural networks. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*, MLHPC '16, page 1–8. IEEE Press, 2016.
[38] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.
[39] Cedric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. Sparcml: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.
[40] Christoph Schär, Oliver Fuhrer, Andrea Arteaga, Nikolina Ban, Christophe Charpilloz, Salvatore Di Girolamo, Laureline Hentgen, Torsten Hoefler, Xavier Lapillonne, David Leutwyler, Katherine Osterried, Davide Panosetti, Stefan Rüdisühli, Linda Schlemmer, Thomas Schulthess, Michael Sprenger, Stefano Ubbiali, and Heini Wernli. Kilometer-scale climate models: Prospects and challenges. *Bulletin of the American Meteorological Society*, 100(12), Dec. 2019. Early Online Release.
[41] Edward N. Lorenz. The predictability of a flow which possesses many scales of motion. *Tellus*, 21(3):289–307, 1969.
[42] Beate Geyer, Thomas Ludwig, and Hans von Storch. Limits of reproducibility and hydrodynamic noise in atmospheric regional modelling. *Communications Earth & Environment*, 2(1):17, Jan 2021.
[43] Sylvain Collange, David Defour, Stef Graillat, and Roman Iakymchuk. Numerical reproducibility for the parallel reduction on multi- and many-core architectures. *Parallel Computing*, 49:83–97, 2015.
[44] P. Balaji and D. Kimpe. On the reproducibility of mpi reduction operations. In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 407–414, 2013.
[45] N. Revol and P. Theveny. Numerical reproducibility and parallel computations: Issues for interval algorithms. *IEEE Transactions on Computers*, 63(08):1915–1924, aug 2014.
[46] Oreste Villa, Vidhya Gurumoorthi, and Sriram Krishnamoorthy. Effects of floating-point nonassociativity on numerical computations on massively multithreaded systems. In *In CUG 2009 Proceedings*, pages 1–11, 2009.
[47] Hesam Zolfaghari, Davide Rossi, and Jari Nurmi. A custom processor for protocol-independent packet parsing. *Microprocessors and Microsystems*, 72:102910, 2020.
[48] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *Architectures for Networking and Communications Systems*, pages 13–24, 2013.
[49] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, April 2014.
[50] Internet Control Message Protocol. RFC 894, April 1984.
[51] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 15–28, New York, NY, USA, 2016. Association for Computing Machinery.
[52] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, February 2019. USENIX Association.
[53] Davide Rossi, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Alessandro Capotondi, Philippe Flatresse, and Luca Benini. PULP: A parallel ultra low power platform for next generation IoT applications. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–39. IEEE, 2015.
[54] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.
[55] S. Mach, F. Schuiki, F. Zaruba, and L. Benini. Fpnew: An open-source multi-format floating-point unit architecture for energy-proportional transprecision computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(04):774–787, apr 2021.
[56] H. Zolfaghari, D. Rossi, W. Cerroni, H. Okuhara, C. Raffaelli, and J. Nurmi. Flexible software-defined packet processing using low-area hardware. *IEEE Access*, 8:98929–98945, 2020.
[57] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. An in-depth analysis of the slingshot interconnect. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.
[58] Broadcom. Tomahawk4 BCM56990 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series, mar 2019.
[59] John D. C. Little. A Proof for the Queuing Formula: $L = \lambda W$. *Oper. Res.*, 9(3):383–387, June 1961.
[60] Shigang Li, Tal Ben-Nun, Salvatore Di Girolamo, Dan Alistarh, and Torsten Hoefler. Taming unbalanced training workloads in deep learning with partial collective operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 45–61, 2020.
[61] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan. A study of process arrival patterns for mpi collective operations. *Int. J. Parallel Program.*, 36(6):543–570, December 2008.
[62] Daniele De Sensi, Salvatore Di Girolamo, and Torsten Hoefler. Mitigating network noise on dragonfly networks through application-aware routing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.
[63] T. Groves, Y. Gu, and N. J. Wright. Understanding performance variability on the aries dragonfly network. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 809–813, Sept 2017.
[64] T. Hoefler, T. Schneider, and A. Lumsdaine. The impact of network noise at large-scale communication performance. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, May 2009.
[65] Sudheer Chunduri, Kevin Harms, Scott Parker, Vitali Morozov, Samuel Oshin, Naveen Cherukuri, and Kalyan Kumaran. Run-to-run variability on xeon phi based cray xc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 52:1–52:13, New York, NY, USA, 2017. ACM.
[66] D. Skinner and W. Kramer. Understanding the causes of performance variability in hpc workloads. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 137–149, Oct 2005.
[67] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.
[68] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero. A quantitative analysis of os noise. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 852–863, 2011.
[69] Bogdan Prisacari, German Rodriguez, Philip Heidelberger, Dong Chen, Cyriel Minkenberg, and Torsten Hoefler. Efficient task placement and routing of nearest neighbor exchanges in dragonfly networks. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 129–140, New York, NY, USA, 2014. ACM.
[70] Samuel D. Pollard, Nikhil Jain, Stephen Herbein, and Abhinav Bhatele. Evaluation of an interference-free node allocation policy on fat-tree clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 26:1–26:13, Piscataway, NJ, USA, 2018. IEEE Press.
[71] Abhinav Bhatele and Laxmikant V. Kalé. Quantifying network contention on large parallel machines. *Parallel Processing Letters*, 19(04):553–572, 2009.
[72] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan. Watch out for the bully! job interference study on dragonfly network. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 750–760, Nov 2016.

[73] X. Wang, M. Mubarak, X. Yang, R. B. Ross, and Z. Lan. Trade-off study of localizing communication and balancing network traffic on a dragonfly system. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1113–1122, May 2018.

[74] Staci A. Smith, Clara E. Cromey, David K. Lowenthal, Jens Domke, Nikhil Jain, Jayaraman J. Thiagarajan, and Abhinav Bhatele. Mitigating inter-job interference using adaptive flow-aware routing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '18, 2018.

[75] A. Kurth, S. Riedel, F. Zaruba, T. Hoefler, and L. Benini. Atuns: Modular and scalable support for atomic operations in a shared memory multiprocessor. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[76] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456, 2015.

[77] Maciej Besta and Torsten Hoefler. Accelerating irregular computations with hardware transactional memory and active messages. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 161–172, New York, NY, USA, 2015. Association for Computing Machinery.

[78] Mellanox. Mellanox Quantum Switches. https://www.mellanox.com/products/infiniband-switches-ic/quantum, mar 2019.

[79] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, March 2011.

[80] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[81] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, 1985.

[82] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[83] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[84] Paul Scheffler, Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. Indirection Stream Semantic Register Architecture for Efficient Sparse-Dense Linear Algebra. 2021.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

All the aggregation algorithms described in the paper have been implemented by using the PsPIN RTL open-source cycle-accurate simulator. For the sparse data results, we also extended the SST simulator (starting from v10.0.0), so that the switches can process in-transit packets. By using this extended SST simulator we implemented in-network dense and sparse allreduce, and host-based ring allreduce.

Author-Created or Modified Artifacts:

```
Persistent ID: 10.5281/zenodo.5497881
Artifact name: PsPIN simulator, handlers, and
↪  extended SST simulator.
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: PsPIN RTL simulator (provided inside the artifact)

Compilers and versions: Pulp RISC-V GNU Compiler Toolchain v1.0.16

Applications and versions: Verilator v4.108, SST simulator v10.0.0 (customized)

Key algorithms: allreduce