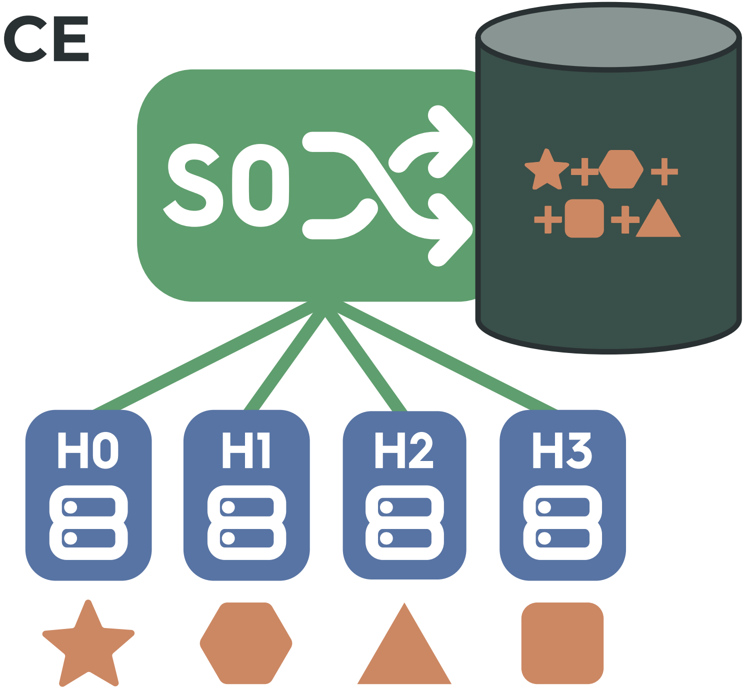
Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, Torsten Hoefler

Flare: Flexible In-Network Allreduce

SC21
St. Louis, MO | science & beyond.

IN-NETWORK ALLREDUCE



2x traffic reduction compared to host-based allreduce

2x bandwidth improvement

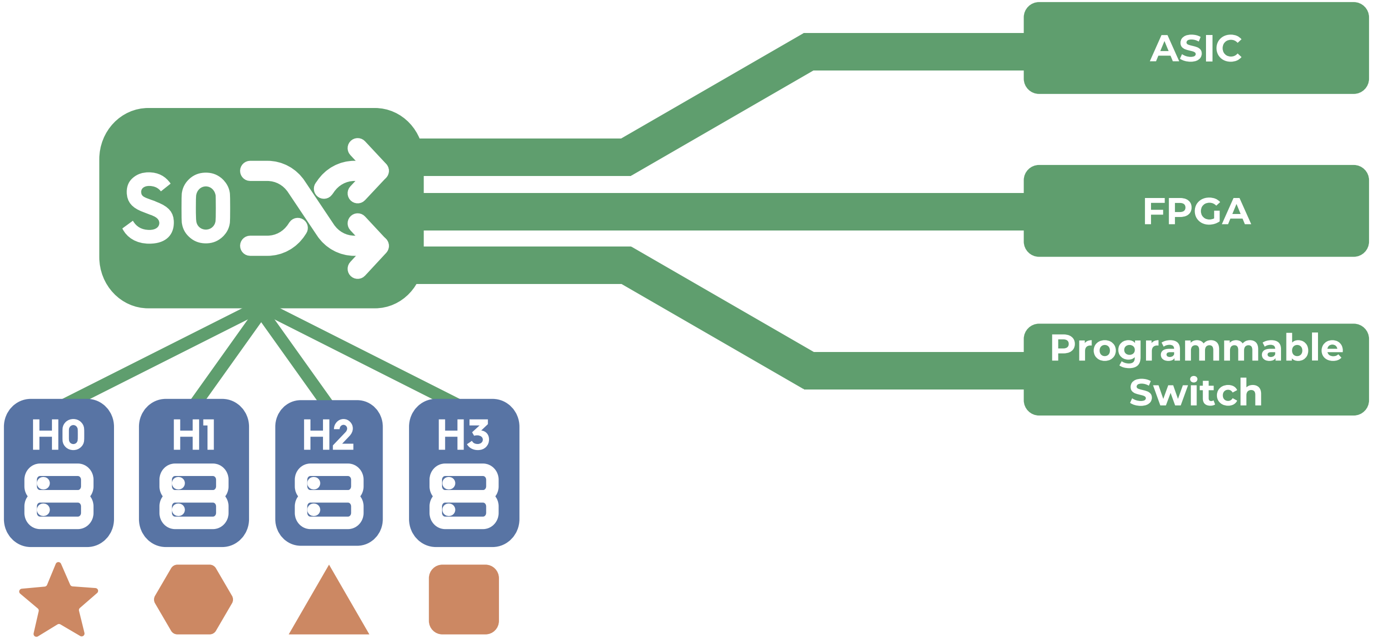
MISSING FEATURES

Custom **operators** and **datatypes**

Support for **sparse data**

Reproducibility

EXISTING SWITCHES ARCHITECTURES



FLARE

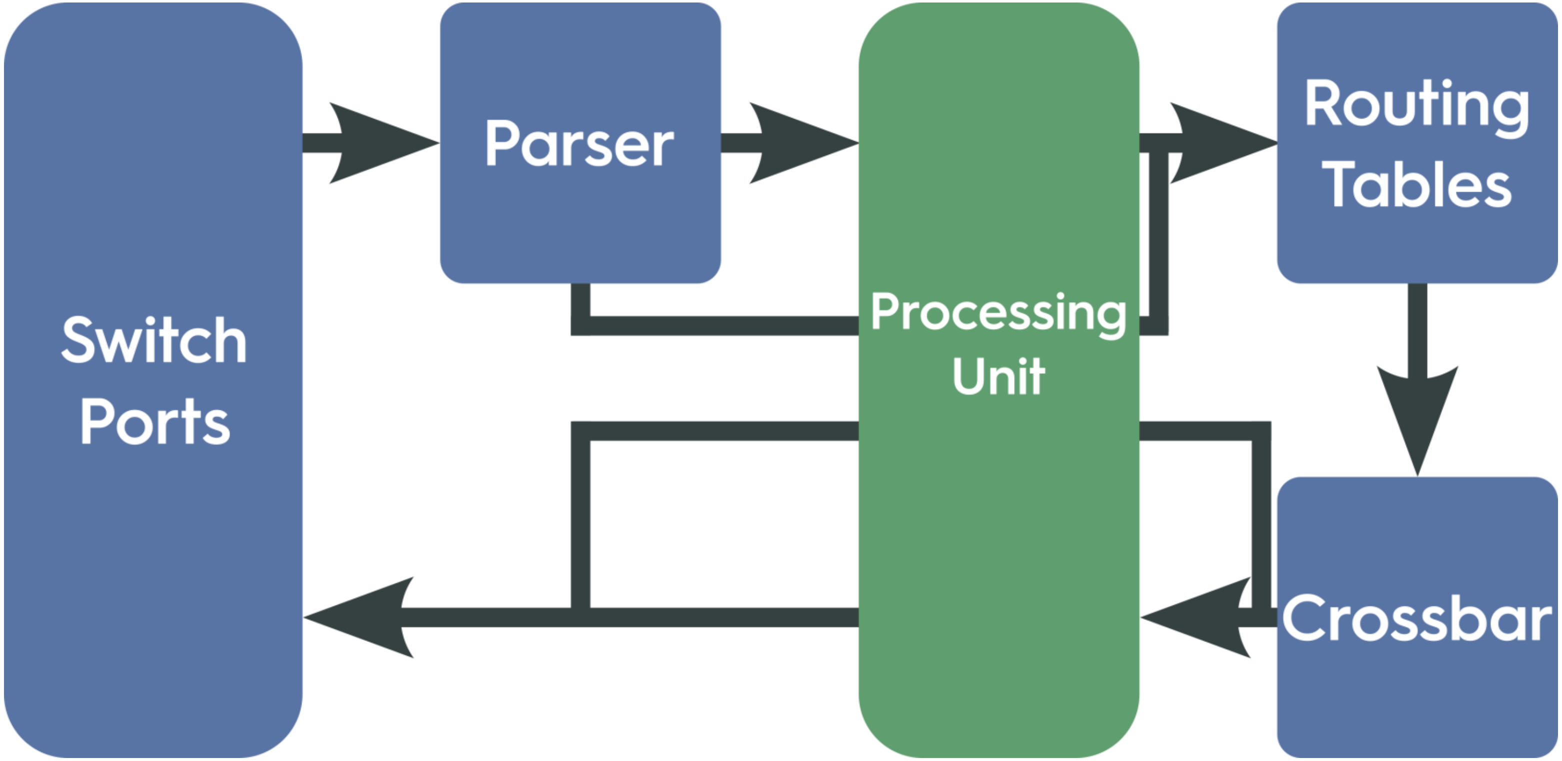
Programmable switch **architecture**

Set of **algorithms**

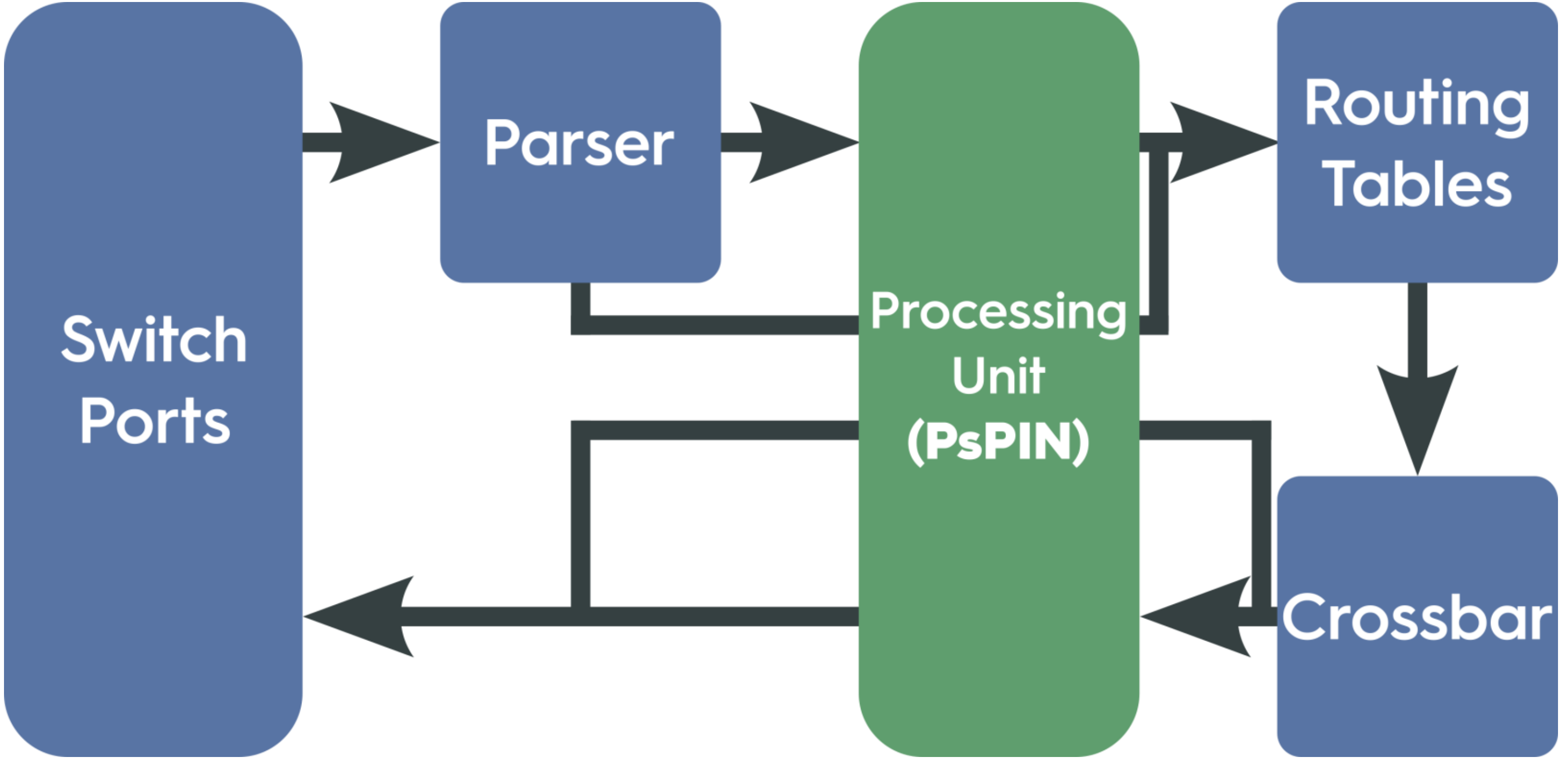
Performance and memory occupancy **models**

Programmable switch **architecture**

SWITCH ARCHITECTURE



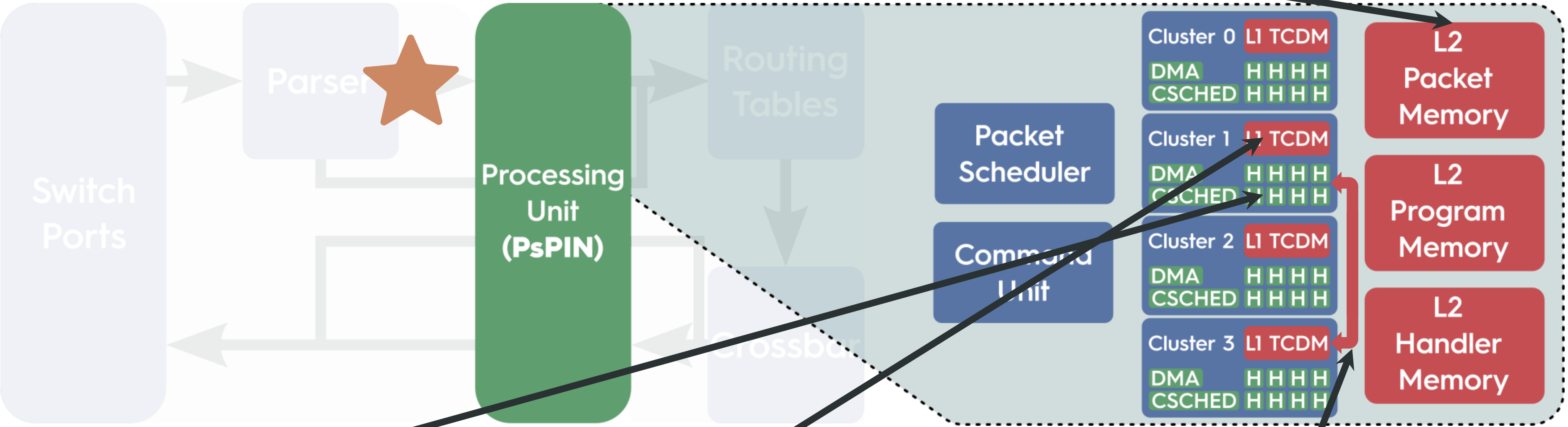
SWITCH ARCHITECTURE



PsPIN: A high-performance low-power architecture for flexible in-network compute - S. Di Girolamo et al. - ISCA'21
sPIN: High-performance streaming Processing in the Network – T. Hoefler et al. – SC'17

PsPIN ARCHITECTURE

Shared memory (~20 cycles accesses)



RISC-V Cores (HPUs)

Scratchpad (1 cycle accesses)

Remote memory accesses (~20 cycles)

PsPIN: A high-performance low-power architecture for flexible in-network compute - S. Di Girolamo et al. - ISCA'21

ADVANTAGES

Processing functions specified as **C kernels**

Coverage of **more use cases**

We can fit **512 cores + memory**

Algorithms

KEY FEATURES

Where to store the data

How to access the data

How to manage **sparse data**

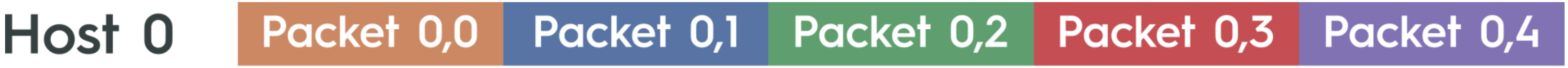
KEY FEATURES

Where to store the data

How to access the data

How to manage **sparse data**

DATA TRANSMISSION

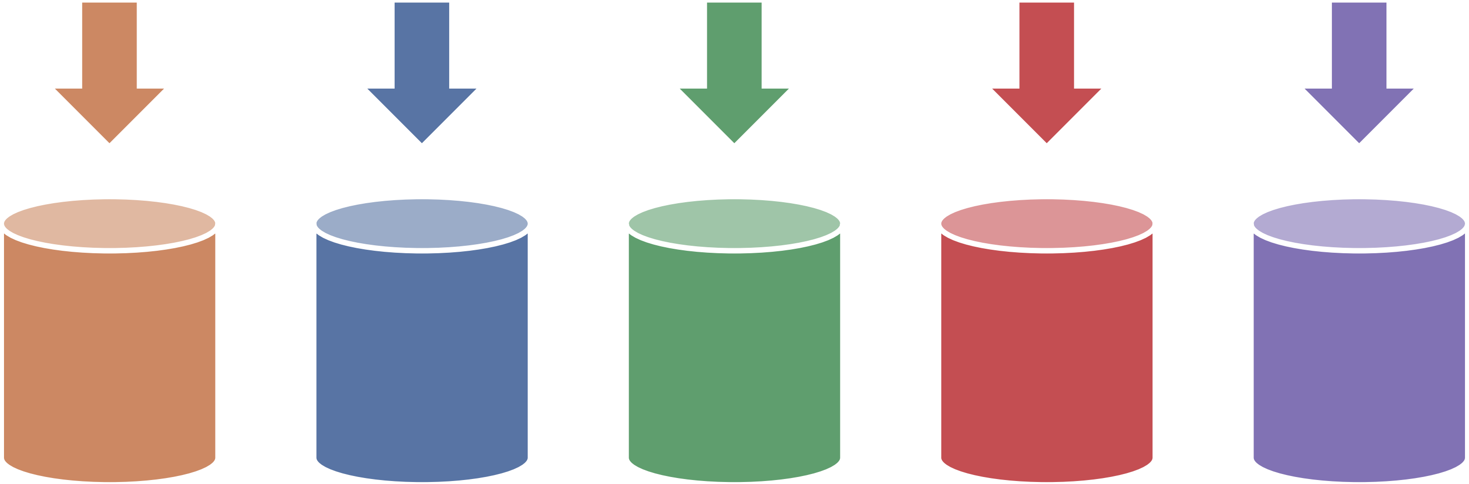


Host 0

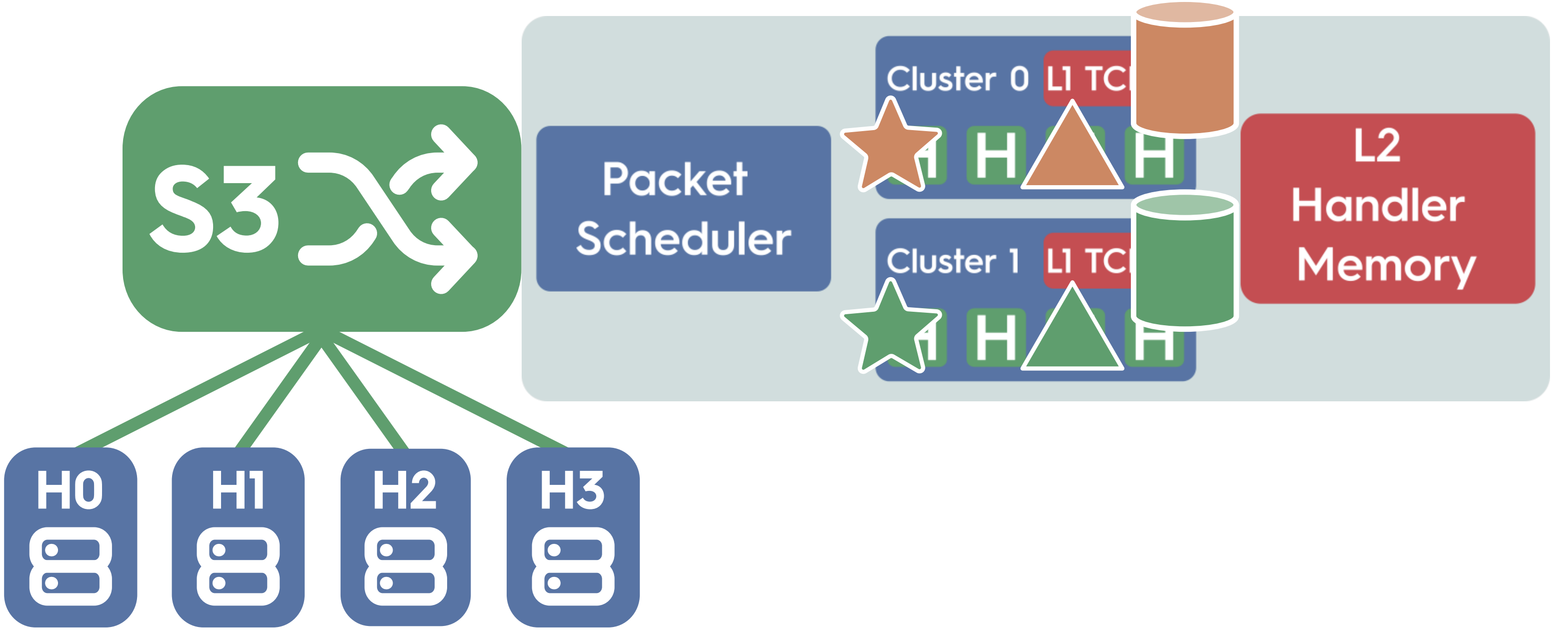
Packet 0,0 | Packet 0,1 | Packet 0,2 | Packet 0,3 | Packet 0,4

DATA AGGREGATION

Host 0	Packet 0,0	Packet 0,1	Packet 0,2	Packet 0,3	Packet 0,4
Host 1	Packet 1,0	Packet 1,1	Packet 1,2	Packet 1,3	Packet 1,4
Host 2	Packet 2,0	Packet 2,1	Packet 2,2	Packet 2,3	Packet 2,4



PACKET SCHEDULING



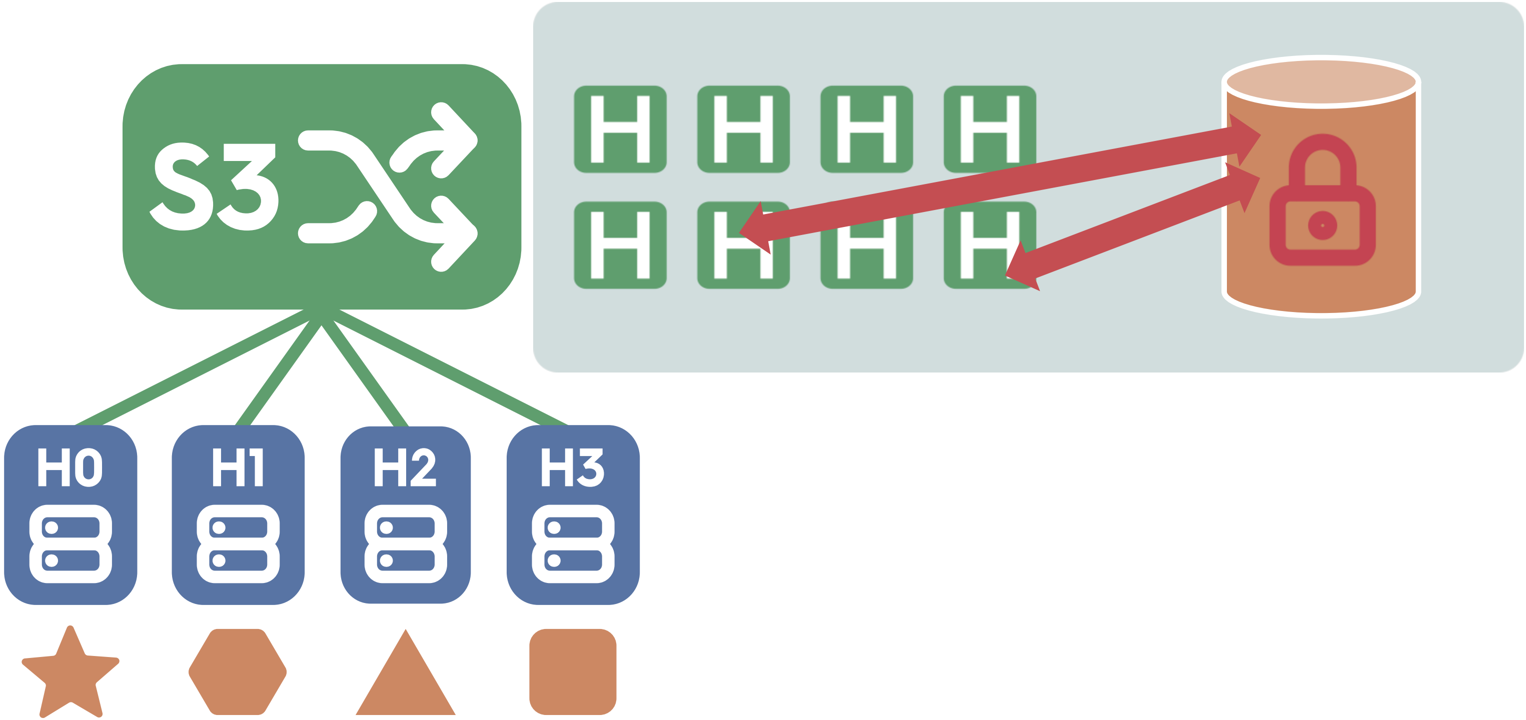
KEY FEATURES

Where to store the data

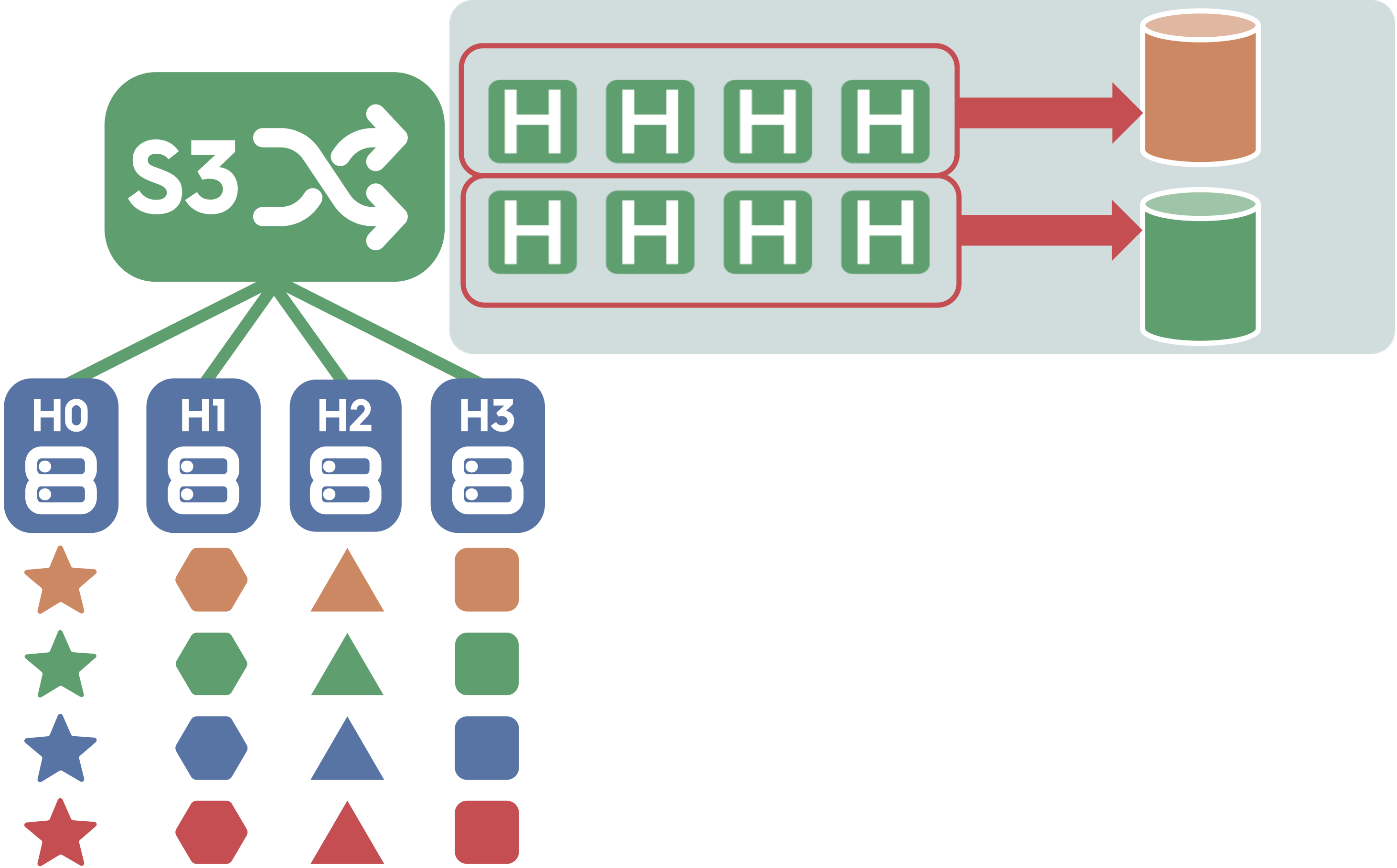
How to access the data

How to manage **sparse data**

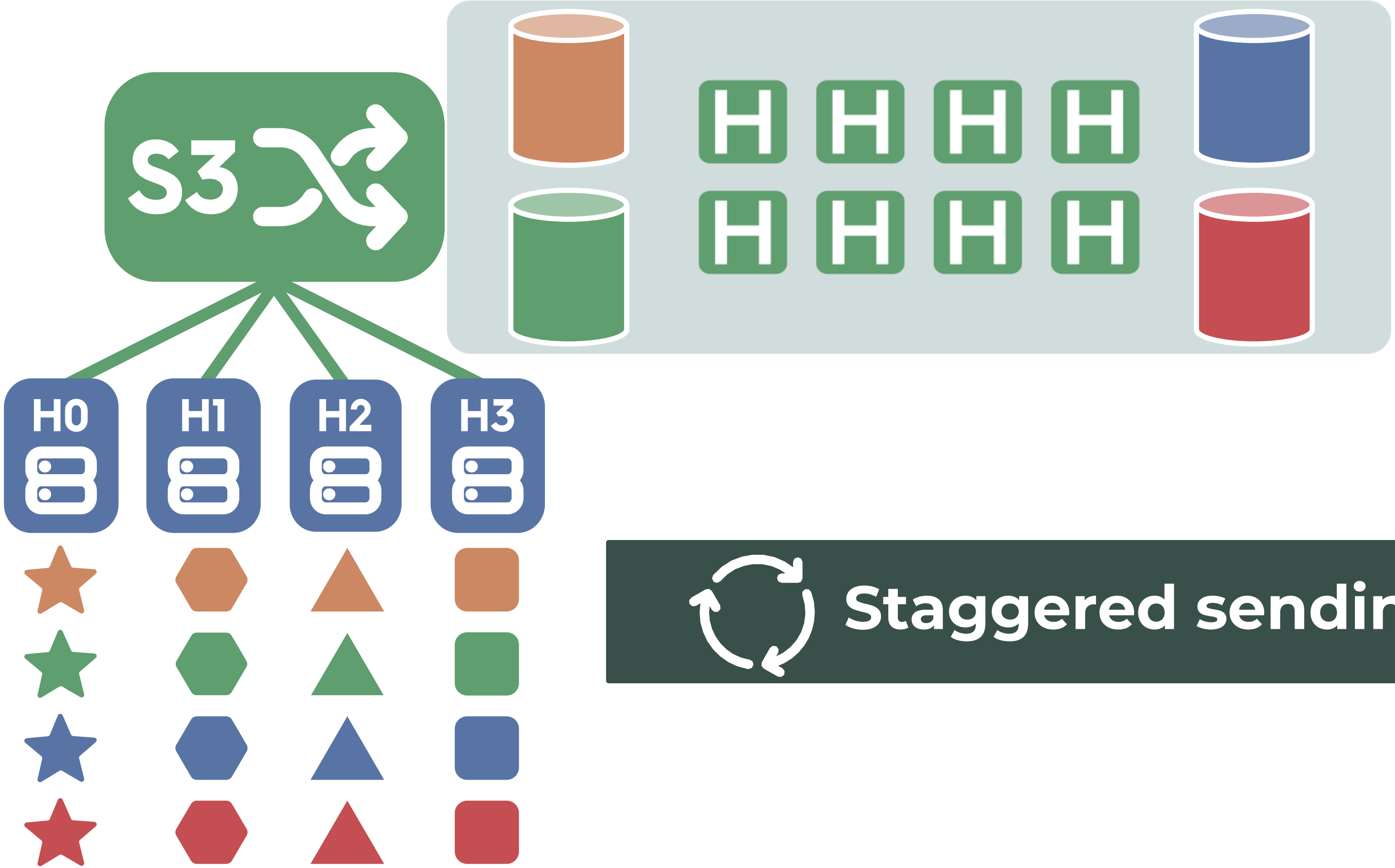
SHARED BUFFER CONTENTION



SHARED BUFFER CONTENTION



SHARED BUFFER CONTENTION



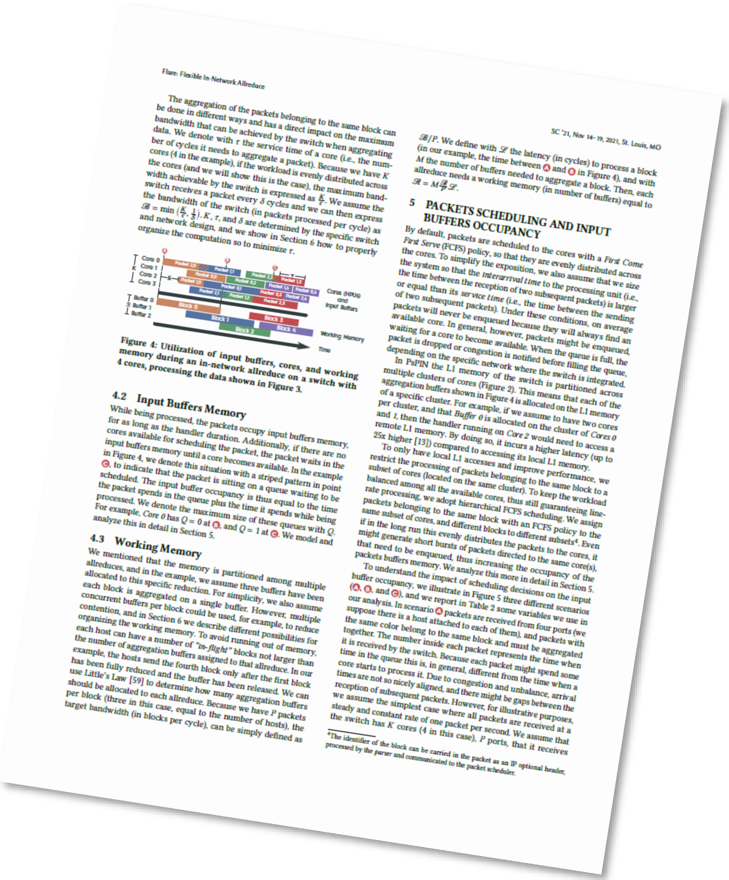
Staggered sending

HOW TO REDUCE CONTENTION

Staggered sending

Single buffer	Multiple buffers	Tree
👍 Minimal memory occupancy	👉 Higher memory occupancy	👎 Highest memory occupancy
👎 High contention	👉 Lower contention	👍 No contention
		👍 Reproducible

PERFORMANCE AND MEMORY MODELLING



Flare: Flexible In-Network Allreduce

SC '21, Nov 14–19, 2021, St. Louis, MO

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 $\mathcal{B} = \min\left(\frac{K}{\tau}, \frac{1}{\delta}\right)$. 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 Ⓒ, 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 Ⓑ, and $Q = 1$ at Ⓒ. 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 $\mathcal{B}/P$. We define with $\mathcal{L}$ the latency (in cycles) to process a block (in our example, the time between Ⓐ and Ⓑ 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 $\mathcal{R} = M\frac{\mathcal{B}}{P}\mathcal{L}$.

5 PACKETS SCHEDULING AND INPUT BUFFERS OCCUPANCY

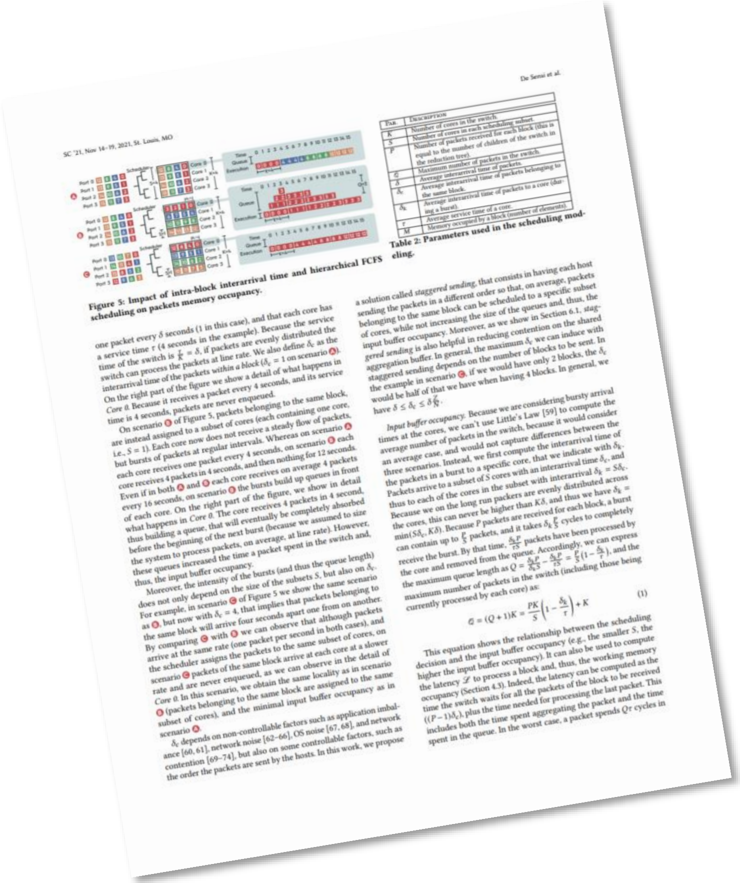
By default, packets are scheduled to the cores with a *First Come First Serve* (FCFS) policy, so that they are evenly distributed across the cores. To simplify the exposition, we also assume that we size the system so that the *interarrival time* to the processing unit (i.e., the time between the reception 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 (Ⓐ, Ⓑ, and Ⓒ), and we report in Table 2 some variables we use in our analysis. In scenario Ⓐ 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 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

[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.



SC '21, Nov 14–19, 2021, St. Louis, MO

De Sensi et al.

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).
$\mathcal{Q}$	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.

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 $\delta \le \delta_c \le \delta\frac{\tau}{K}$.

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:

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

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 $\mathcal{L}$ 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

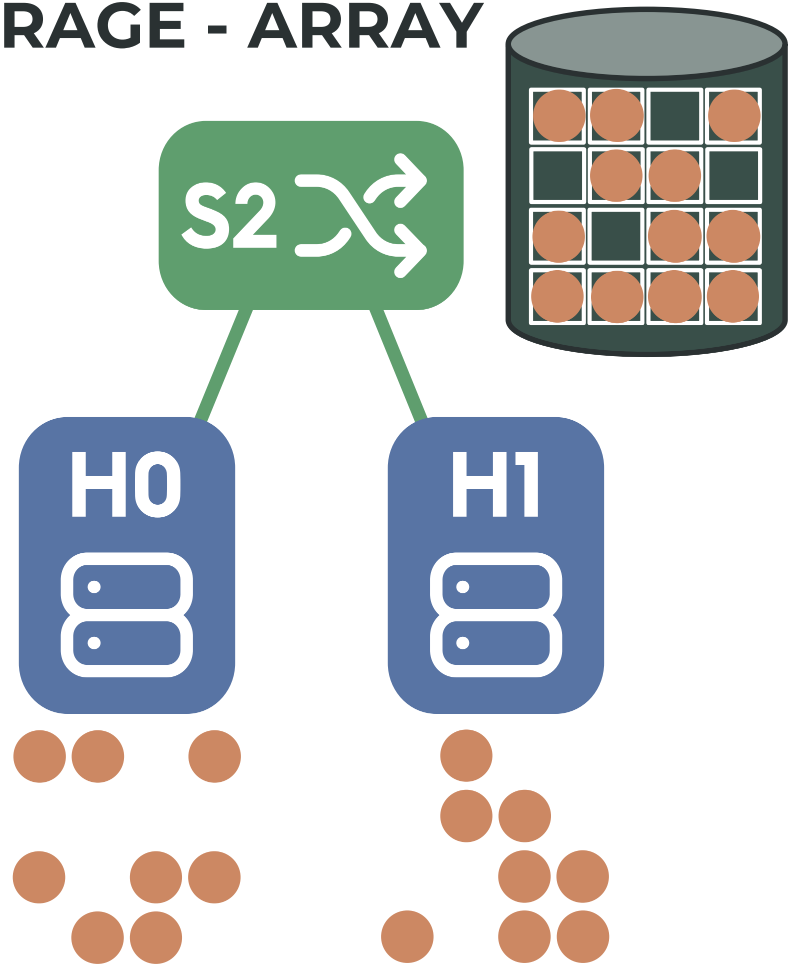
KEY FEATURES

Where to store the data

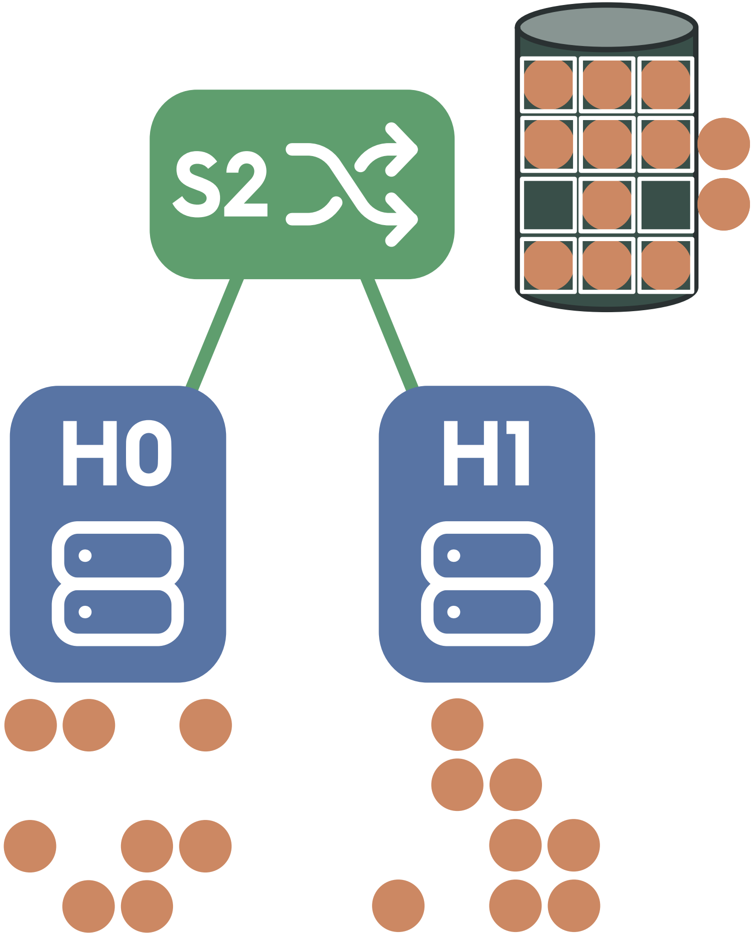
How to access the data

How to manage **sparse data**

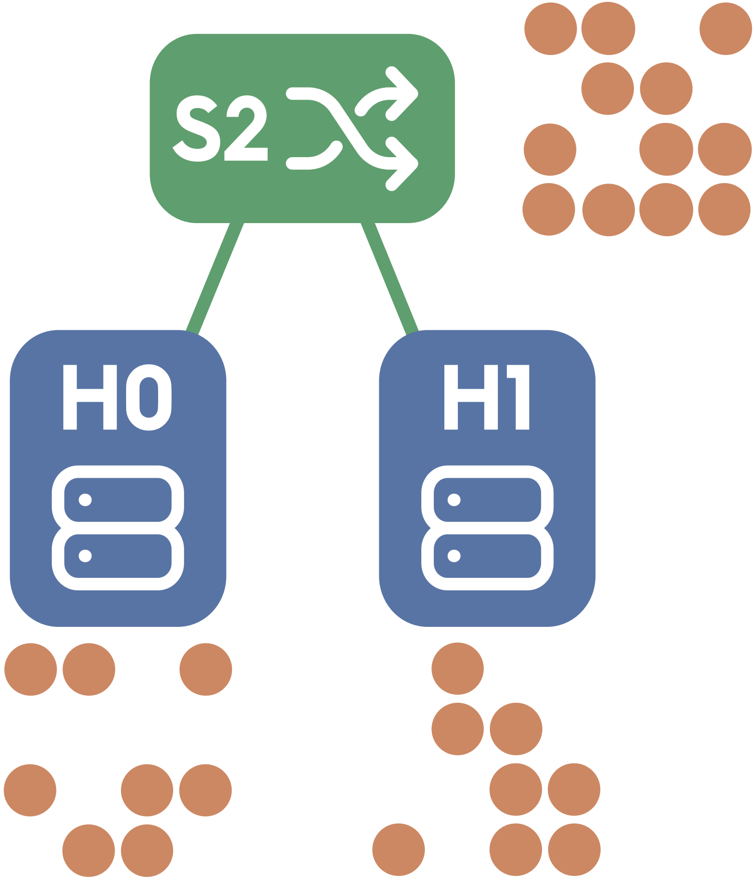
SPARSE DATA STORAGE - ARRAY



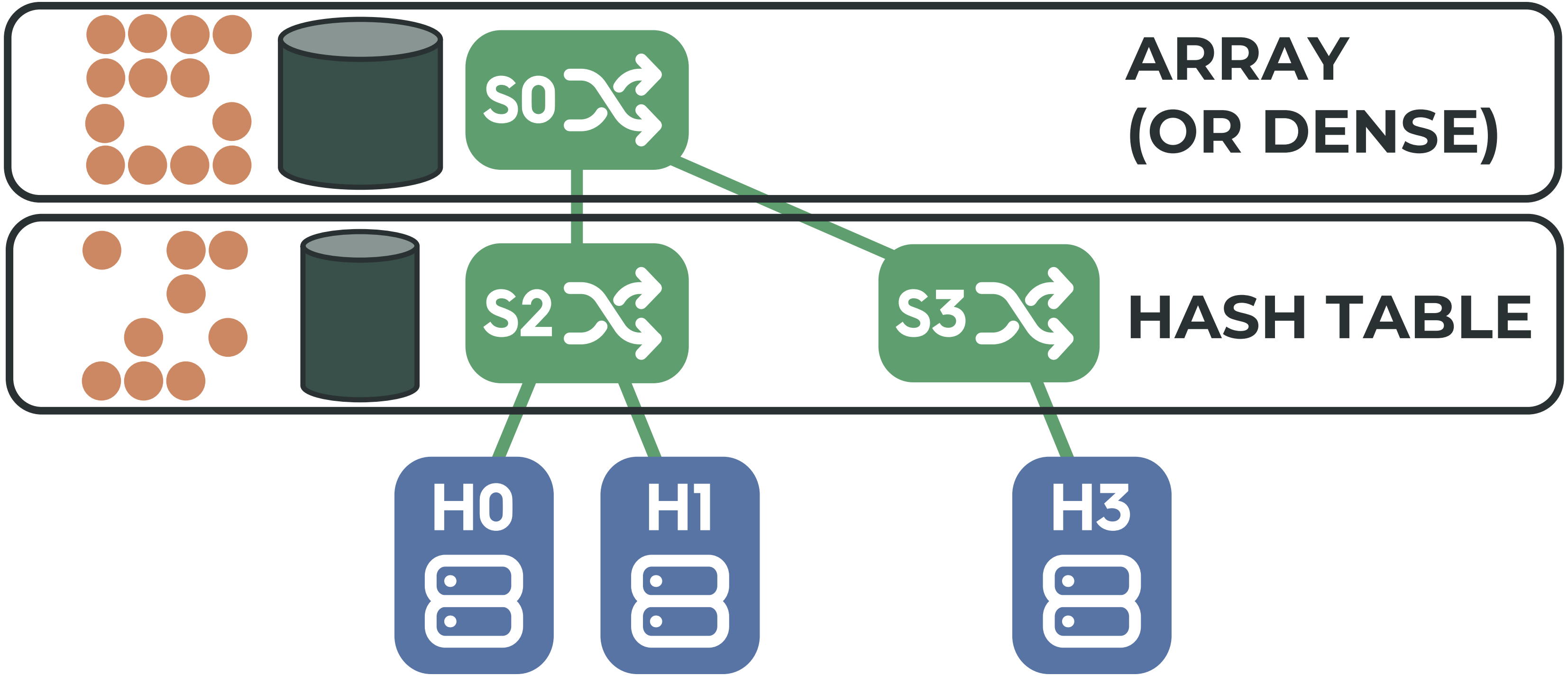
SPARSE DATA STORAGE – HASH TABLE



DATA FILL-IN

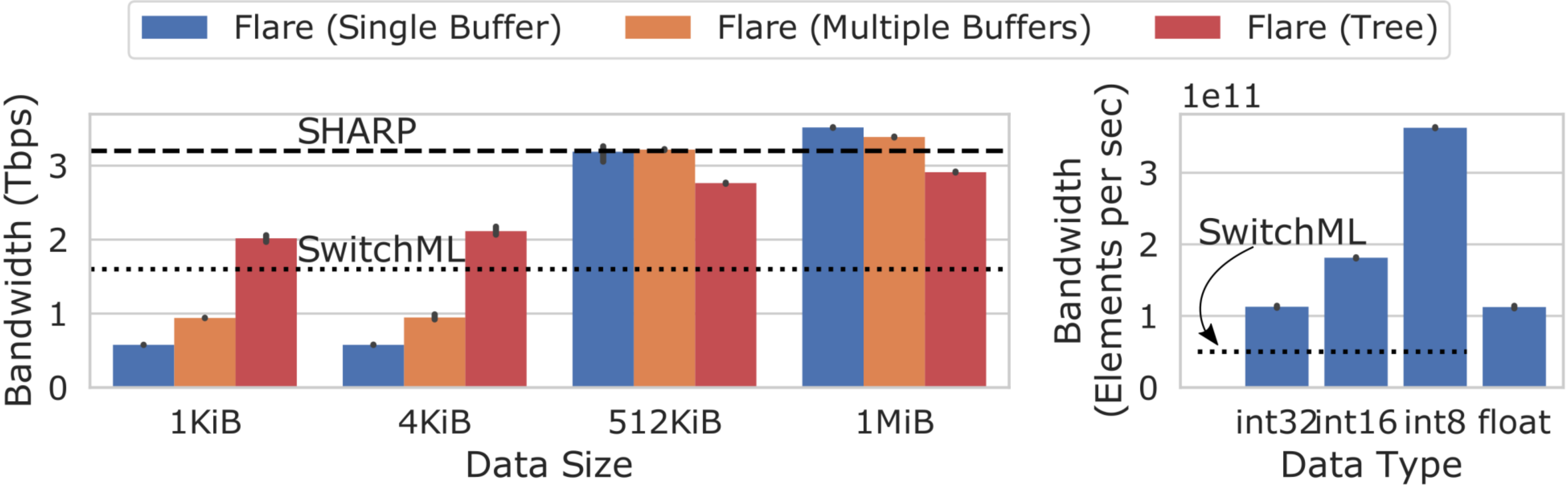


TWO-LEVEL APPROACH

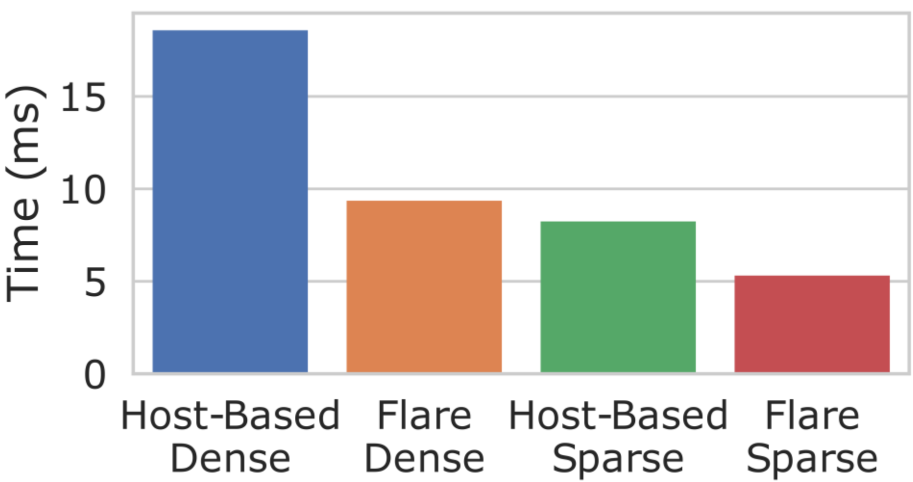


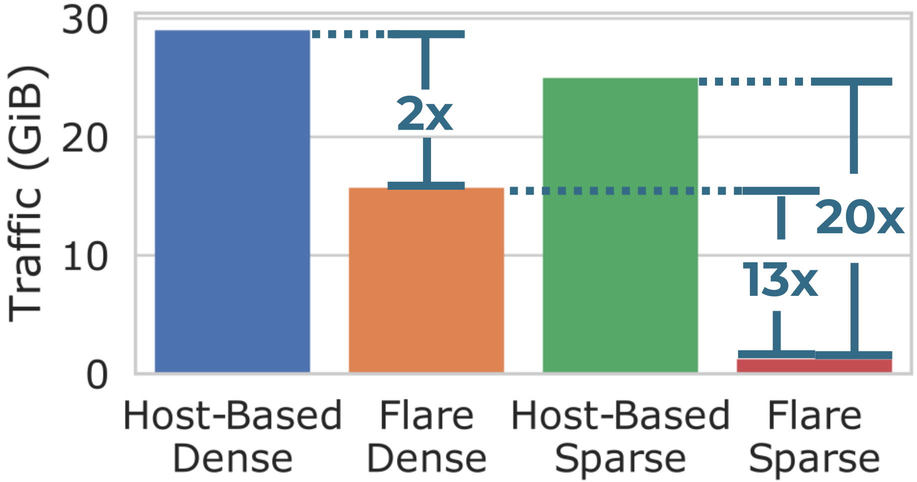
Evaluation

RESULTS – SINGLE SWITCH



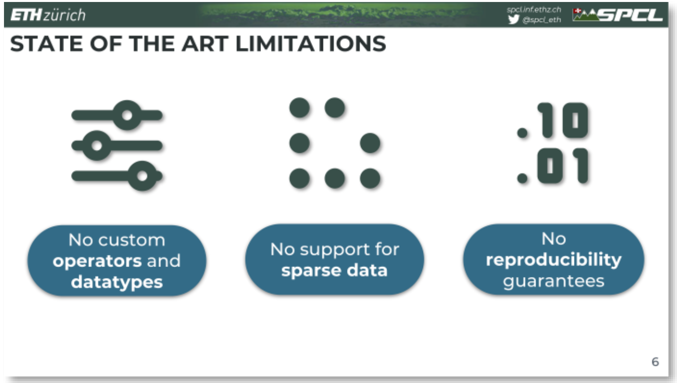
RESULTS - 64 NODES, 2-LEVELS FAT TREE





Communication time of a **ResNet50** iteration with **sparsified gradients** (0.2% density)

CONCLUSIONS



ETH zürich spcl.inf.ethz.ch @spcl_eth

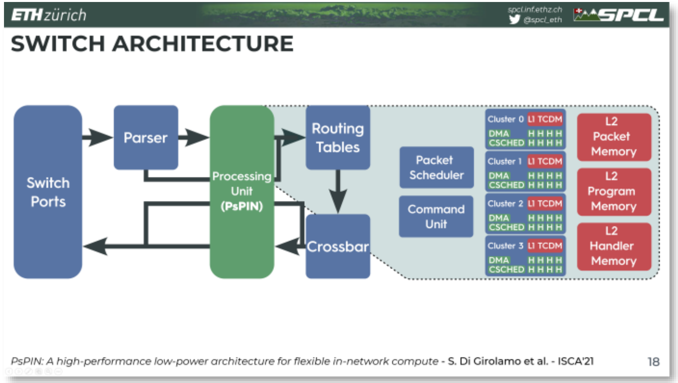
STATE OF THE ART LIMITATIONS

No custom **operators** and **datatypes**

No support for **sparse data**

No **reproducibility** guarantees

6

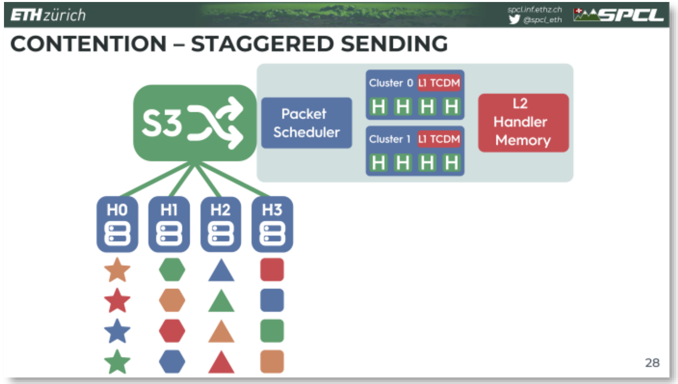


ETH zürich spcl.inf.ethz.ch @spcl_eth

SWITCH ARCHITECTURE

PsPIN: A high-performance low-power architecture for flexible in-network compute - S. Di Girolamo et al. - ISCA'21

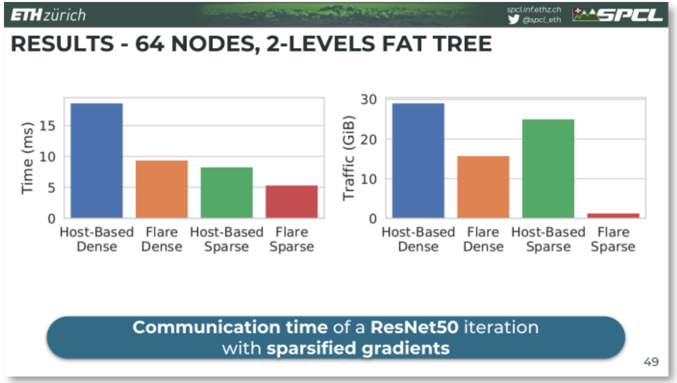
18



ETH zürich spcl.inf.ethz.ch @spcl_eth

CONTENTION – STAGGERED SENDING

28



ETH zürich spcl.inf.ethz.ch @spcl_eth

RESULTS - 64 NODES, 2-LEVELS FAT TREE

Communication time of a **ResNet50** iteration with **sparsified gradients**

49