HORIZON 2020
TOPIC FETHPC-02-2017
Transition to Exascale Computing

Exascale Programming Models for Heterogeneous Systems
801039

D2.5
Update on Current Landscape for Novel Network Hardware and Programming Models

WP 2: Programming high-performance communication networks and fabrics in heterogeneous systems

Date of preparation (latest version): 30/11/2021

DOCUMENT INFORMATION

Deliverable Number	D2.5
Deliverable Name	Update on Current Landscape for Novel Network Hardware and Programming Models
Due Date	30/11/2021 (PM 39)
Deliverable lead	UEDIN
Authors	Valeria Bartsch (Fraunhofer) Oliver Thomson Brown (UEDIN) Tim Dykes (HPE) Arcesio Castaneda Medina (Fraunhofer) Harvey Richardson (HPE) Timo Schneider (ETH)
Responsible Author	Oliver Thomson Brown (UEDIN) e-mail: `o.brown@epcc.ed.ac.uk`
Keywords	MPI GPI Network
WP/Task	WP2 / Task 2.1
Nature	R
Dissemination Level	PU
Planned Date	30/11/2021
Final Version Date	30/11/2021
Reviewed by	Nick Johnson (UEDIN) Gilbert Netzer (KTH)
MGT Board Approval	

DOCUMENT HISTORY

Partner	**Date**	**Comment**	**Version**
UEDIN	06/07/2021	Initial version - modified template from D2.4	0.1
All Partners	29/10/2021	Internal review version	0.2
All Partners	26/11/2021	Final version	0.3

Executive Summary

This document is an update on the previous deliverable, D2.1 "Report on Current Landscape for Novel Network Hardware and Programming Models", prepared in January 2018.

In Section 3 we provide an update on advances in novel network hardware since the previous deliverable. In particular, we noticed that programmable network cards (SmartNICs) received a lot of attention in research since D2.1 was submitted. Thus we conducted a survey of SmartNIC technologies in Section 3.2. We identified a gap in the space of existing solutions, which is filled by PsPIN, a SmartNIC architecture based on RISC-V processors, described in Section 3.4. We also perform a survey of in-network compute solutions in Section 3.3, focusing on the ability to accelerate the allreduce collective operation because it is an important operation for ML applications which are part of EPiGRAM-HS. Again we identify a gap in the space of existing solutions, which we fill with our proposed solution Flare, described in Section 3.5. Note that hardware development is not the focus of EPiGRAM-HS, but hardware developments present features which could be exploited by software and programming models developed within the project, either now or in the future, and vice-versa, hardware developments need to be aware of needs and usage patterns of exascale applications. Thus EPiGRAM-HS is the perfect place to bring this work together.

In Section 4 we discuss advances in novel network programming models. In Section 4.1, we provide an update on standardisation efforts in MPI which should help application developers to take advantage of SmartNICs once library developers have updated to meet the latest standard. In particular, persistent collectives have been introduced in to the MPI-4.0 standard, and that standard is now ratified. In Section 4.2 we describe progress on a GPI-NIC. GPI-NIC is a project to develop a high-performance communication device that enables data transport on heterogeneous architectures where application processing units (APUs) and FPGA devices have to communicate at the same time to each other. In Section 4.3 we briefly discuss progress on a higher level abstraction which helps application developers to create heterogeneous application pipelines. This is done by combining the Mamba heterogeneous memory manager developed in WP3 with the UDJ (Universal Data Junction) library, allowing Mamba to support distributed layouts.

Contents

1 Introduction 6

2 Summary of D2.1 7

3 Update on novel network hardware 8
3.1 Network hardware components 8
3.2 Overview of SmartNICs 10
3.3 Overview of in-network reduction solutions 13
3.4 Our RISC-V based SmartNIC Architecture 16
3.5 Flexible in-network Allreduce 21

4 Update on novel network programming models 27
4.1 MPI persistent and planned collectives 27
4.2 GASPI 28
4.2.1 GPI support for interconnects 28
4.2.2 Change of GASPI specification to support Omnipath 28
4.2.3 GPI-NIC 28
4.3 UDJ 29

5 Conclusion 31

1 Introduction

The structure of this document is as follows.

In Section 2 we present a summary of the previous deliverable, followed by an update on advances in novel network hardware in Section 3. This includes a survey of SmartNIC technologies and of in-network compute solutions with a particular focus on the ability to accelerate the AllReduce collective operation because it is an important operation for ML applications which are part of EPiGRAM-HS.

In Section 4 we discuss advances in novel network programming models, including updates on MPI and GASPI.

Finally, we conclude in Section 5.

2 Summary of D2.1

Deliverable 2.1, “Report on Current Landscape for Novel Network Hardware and Programming Models”, emphasises that high bandwidth and low latency are always beneficial to the functioning of a network, but generally improves with time and so for the present report this not of particular interest. D2.1 identifies the following useful technologies that may be present in a HPC network:

- *OS bypass.* Sends and receives can be completed without needing to switch from user to kernel mode.
- *Zero-copy transfer.* Data is transferred from userspace to remote userspace without intermediate copies.
- *Application bypass.* The progress engine, or compute, is offloaded to the network hardware.
- *Cache injection.* Data is transferred directly to the cache of the remote host, on the grounds that the remote CPU is likely to need it soon.
- *RDMA.* Remote Direct Memory Access – data is transferred without involving the remote OS or CPU after an initial stage. Used to implement one-sided communication protocols.

Interconnect network systems were then discussed, including Cray Gemini, Aries, and Slingshot [1, 2, 3], though limited details on Slingshot were then available. Outside of Cray (now HPE), Mellanox (now NVIDIA) ConnectX series interconnects, NVIDIA NVLink [4], Tofu [5], and Atos BXI [6] were discussed, in particular highlighting where they made use of the technologies listed above.

Next, low-level network software APIs were considered. These APIs are designed to allow communication library developers to take advantage of the novel features a network may offer, and are designed to support offloading some of the processing effort involved in communications or collective operations from the CPU to other devices such as FPGAs or the NIC. As these are low-level libraries however, this does not necessarily translate in to support for these things at the application level.

Finally, D2.1 discussed the requirements of the GASPI specification, and the then-current MPI-3.0 standard, how the above technologies could be used to support them, as well as challenges to taking advantage of them given the restrictions of the two programming models. Potential solutions to these challenges involving extensions or improvements to the standardised APIs were suggested, including MPI support for persistent I/O and one-sided communications, and widening the range of communication semantics supported by GASPI.

3 Update on novel network hardware

As in D2.1 we focus on the unique capabilities of novel network hardware (such as those discussed in section 2), on the grounds that higher bandwidth and lower latency are always beneficial, and are generally just a proxy for how new the technology is. Higher bandwidth facilitates faster transmission with a greater impact on large messages, while reduced latency does the same but with a greater impact on small messages. The radix of a switch is the number of I/O ports available for connection.

3.1 Network hardware components

Since D2.1's report, progress has been made on the network hardware discussed there. In the following, interconnects superseding some of those networks, as well as new vendor fabrics and interconnect technologies, are briefly discussed.

- Slingshot.

  As the successor of Cray Aries, it was introduced as the underlying interconnect in the Cray Shasta system available since 2020. The Shasta system is now marketed under the HPE Cray Supercomputer branding. The actual Slingshot network follows the specifications discussed in D2.1 when it was advertised. Built on a customized version of Ethernet optimized for high-performance computing, it solves some of the shortcomings found in the standard Ethernet protocol (e.g., big headers and large packet sizes) to scale to large HPC workloads, and supports direct connection to third-party Ethernet-based storage devices and/or networks. With link speeds of 200 Gb/s, the HPE Slingshot and the high-radix 64-port switch (Dragonfly topology) delivers an aggregated bandwidth of 12.8 Tb/s (per direction) and scales up to 250000 endpoints with a maximum of three switch-to-switch hops. Other important features include a Quality of Service (QoS) system based on traffic classes which allows, e.g., assign the network resources required by high priority jobs; adaptive routing of packets according to load information gathered by the hardware; and an automatic congestion management system to identify sources of congestion and limit their impact on production traffic and system services.

- NVIDIA (Mellanox) ConnectX series.

  The ConnectX-6 has now been superseded by the NVIDIA ConnectX-7 NDR for InfiniBand and the NVIDIA ConnectX-7 SmartNIC for Ethernet, both of which support up to four ports and reach up to 400 Gb/s of bandwidth per port.

  Coupled to the 64-ports Quantum-2-based switches, the aggregated bandwith doubles that of Slingshot (25.6 Tb/s/per direction). The Infiniband adapter supports RDMA semantics, atomic operations and a hardware-based congestion system and reliable transport. Important enhanced features include the extended reliable connected (XRC) protocol, the dynamical connected transport (DCT) and out-of-order RDMA with adaptive routing, as well as, support for GPUDirect RDMA and GPUDirect Storage, and NVMe over fabrics (NVMeoF). The in-network compute features allow collective (and vector collective) operations offload, MPI tag matching and All-to-All offloads.

- NVIDIA Bluefield SmartNICs.

  This family of NVIDIA data processor units (DPU) are systems-on-a-Chip (SoC) which allow offloading from a host server CPU of tasks such as interfacing to storage devices, security checking, and network data transmission. Supporting Ethernet and Infiniband network interfaces they can reach up to 400 Gb/s and include networking features as those of the ConnectX-7 series. For computation they are equipped with up to 16 Arm A78 Hercules cores, 16 GB on-board DDR5 and can execute up to 1.5 TOPS (according to the SPECrate 2017 Integer benchmarks).

  The Ethernet SmartNICs, on the other hand, do not dispose of DPU hardware, but have features as Accelerated Switching and Packet Processing (ASAP2) for CPU bypass, advanced RoCE protocols, GPUDirect RDMA and Storage, NVMeoF and an All-to-All engine. Coupled to the Spectrum family of switches it is possible to attain up to 12.8 Tb/s of aggregated bandwidth.

- Atos Bull BXI

  The latest Bull interconnect (BXI-V2) has a hardware communication management system which offloads this task completely from the host CPUs. The NIC architecture is based on the portal P4 communication library and allows for optimization of the MPI and PGAS primitives, OS and application bypass, and offloaded collectives. Enhanced QoS and adaptive routing are also part of BXI, as well as, resilience and reliability mechanisms. The switch ASIC has low latency and features 48 ports at 100 Gb/s, for an aggregate bandwidth of 6 TB/s/per direction.

- AWS Elastic Fabric Adapter (EFA)

  EFA is an extension of the Elastic Network Adapter (ENA) attaining up to 100 Gb/s of bandwidth; it is offered as an additional service provided by Nitro Virtual Private Cloud (VPC) cards on AWS high-performance servers. The enhanced networking used in Nitro VPC cards rely on (Ethernet) PCIe NICs equipped with SR-IOV technology which allows exposure of the dedicated PCIe devices to EC2 instances with higher I/O performance, lower latency and lower CPU utilization (in comparison to traditional para-virtualized network interfaces). EFA, on the other hand, is designed according to a scalable reliable datagram (SRD) protocol, which provides reliable but out-of-order delivery, and leaves order restoration to the layers on top of it. Equal-Cost Multi-Path (ECMP) routing is used in the Ethernet switches in conjunction with a dynamic rate limit approach to the congestion control. At the user level, EFA interface resembles InfiniBand verbs, although the SRD semantics are quite different from the standard InfiniBand transport types. The user-space driver exposes the native EFA's reliable out-of-order delivery, while the libfabric provider implements packet re-ordering as a part of message segmentation and MPI tag matching support.

3.2 Overview of SmartNICs

The field of programmable NICs (SmartNICs) has received a lot of attention since the preparation of D2.1. Therefore in this deliverable we want to devote special attention to those. We first give a comparative overview of SmartNIC capabilities. We include novel products as well as older ones, to allow the reader to understand the directions in which the field is moving. As it became evident over the course of the project that SmartNICs will likely be a part of any future heterogeneous exascale system, ETH also devoted some of its research efforts to this field, which we will report on in Sections 3.4, and 3.5 respectively. Since we identified significant shortcomings both in existing in-network compute solutions in general but also in-network reduction solutions, we published those findings together with our proposed solutions at high-impact conferences, i.e., we propose PsPIN [7] and Flare [8]. In the following we give excerpts of those publications as applicable to the context of this deliverable document. The projects PsPIN and Flare are partially funded by EPIGRAM-HS and partially by the RED-SEA project — while the work within EPIGRAM-HS focused on identifying gaps in the landscape of SmartNICs and in-network compute solutions, our work in RED-SEA is focused on developing our proposed solutions into prototypes available to other partners.

Today's cloud and high-performance datacenters form a crucial pillar of compute infrastructures and are growing at unprecedented speeds: The number of cloud instances has risen by 550% in the period of 2010 to 2018, the IP traffic handled by datacenteres has increased ten-fold [9]. At the core, they are a collection of machines connected by a fast network carrying petabits per second of internal and external traffic. The importance of services such as video communication, streaming, and online collaboration tools for our society is evident since the COVID-19 pandemic — they can be regarded as a pillar of modern economy — in datacenters they increase the incoming and outgoing traffic volume. Furthermore, the growing deployment of specialized accelerators such as AccelNet devices [10] used by Microsoft Azure and general trends towards disaggregation exacerbates the quickly growing network load. Packet processing capabilities are a top performance target for datacenters.

These requirements have led to a wave of modernization in datacenter networks: not only are high-bandwidth technologies going up to 200 Gbit/s gaining wide adoption but endpoints must also be tuned to reduce packet processing overheads. Specifically, remote direct memory access (RDMA) networks move much of the packet and protocol processing to fixed-function hardware units in the network card and directly access data into user-space memory. Even though this greatly reduces packet processing overheads on the CPU, the incoming data must still be processed. A flurry of specialized technologies exists to move additional parts of this processing into network cards, e.g., FPGAs virtualization support [10], P4 simple rewriting rules [11], or triggered operations [12].

Streaming processing in the network (sPIN) [29] defines a unified programming model and architecture for network acceleration beyond simple RDMA. It provides a user-level interface, similar to CUDA for compute acceleration, considering the specialties and constraints of low-latency line-rate packet processing. It defines a flexible and programmable network instruction set architecture (NISA) that not only lowers the barrier of entry but also supports a large set of use-cases [29]. For example, Di Girolamo et al. demonstrate up to 10x speedups for serialization and deserialization (marshalling) of non-consecutive

Solution	L	P	G	U	Notes
Azure AccellNet [10]	👍	🖓	🖒	🖓	FPGA-based NICs; Flow-steering.
P4 [11]*, FlexNIC [13]	👍	🖒	🖒	🖓	Packet steering and rewriting. FlexNIC adds memory support. *Runs on NICs and switches.B.3
Mellanox SHARP [14]	👍	🖓	🖒	👍	Data aggregation and reduction. Runs on switches.
Portals 4 [12], INCA [15]	🖒	🖒	🖓	👍	Sequences of predefined actions can be expressed with triggered operations. Both target NICs.
Mellanox CORE-Direct [16]	👍	🖒	🖓	👍	Sequence of predefined actions can be chained. Targets switches.
Cray Aries Reduction Engine [2]	👍	🖓	🖒	👍	Data reductions (up to 64 bytes). Runs on switches.
Quadrics [17], Myrinet [18]	👍	👍	🖒	🖓	Users define threads to run on the NIC / NIC is re-programmable by users.
SmartNICs [19, 20]	🖒	👍	👍	🖓	Runs full linux stack on the NIC; Offloading of new code requires flashing.
FPGA packet parsing pipeline [21]	👍	🖒	🖒	🖓	Only packet parsing, read-only packets, might require FPGA reconfiguration. Target NICs.
eBPF (host) [22]	🖓	🖒	🖒	🖓	Runs user-defined code (eBPF code) in virtual machine in the OS kernel.
eBPF (Netronome) [23], hXDP [24]	👍	🖒	🖒	🖓	eBPF programs can be offloaded to NIC.
DPDK [25]	🖓	👍	🖒	🖓	Runs in user space. Applications can poll for new raw packets from the NIC.
StRoM [26]	👍	👍	🖓	🖓	Handlers for DMA streams are implemented on FPGA NIC.
NICA [27]	👍	👍	🖓	🖓	Bind kernels running on on-NIC accelerators to user sockets.
PANIC [28]	👍	👍	👍	🖓	Applications compose execution of pre-installed compute units. Targets NICs.
sPIN [29]	👍	👍	👍	👍	Applications define C/C++ packet handlers to map to different messages/flows. Targets NICs.

Table 1: **L**: Location (👍 on NICs or switches; 🖒 on NICs but outside the packet pipeline; 🖓 on host CPU). **P**: Programmability (👍 fully programmable; 🖒 limited programmability; 🖓 predefined functions). **G**: Granularity (👍 message and packets; 🖒 only packets; 🖓 only messages). **U**: Usability (👍 usable by applications and privileged users; 🖓 only by privileged users).

data [30].

In-network compute is the capability of an interconnection network to process, steer, and produce data according to a set of programmable actions. The exact definition of *action* depends on the specific in-network-compute solution: it can vary from pre-defined actions (e.g., pass or drop a packet according to a set of rules) to fully programmable packet or message handlers (e.g., sPIN handlers).

There are several advantages of computing in the network: (1) *More overlap.* Applications can define actions to execute on incoming data. Letting the network execute them allows applications to overlap these tasks with other useful work; (2) *Lower latency.* The network can promptly react to incoming data (cf. Portals 4 triggered operations [12], virtual functions [10], sPIN handlers), immediately executing actions depending on it. Doing the same on the host requires applications to poll for new data, check for dependent actions, and then execute them. (3) *Higher throughput.* Some in-network-compute solutions enable stream processing of the incoming data. For example, sPIN can run packet handlers on each incoming packet, potentially improving the overall throughput. (4) *Less resource contention.* Moving computational tasks into the network can reduce the volume of data moved through the PCIe bus and the memory hierarchy. This implies fewer data movements, less memory contention and cache pollution, potentially improving the performance of host CPU tasks.

Table 1 surveys existing in-network-compute solutions. This classification focuses on the high-level characteristics of these solutions, comparing them by the location where policies are run, the level of programmability, the granularity at which the actions are applied, and their usability.

(L) Location. Tasks can be executed at different points along the path from the endpoint sending the data to the endpoint receiving it. We classify in-network-compute solutions as: 👍 running in network devices (e.g., on NICs or switches); 🖒 running in network devices but not on the packet pipeline (e.g., SmartNICs act as close-to-network endpoints, running full Linux stack); 👎 if they run on the host CPUs.

(P) Programmability. It defines the expressiveness of the actions. Network solutions enabling fully programmable actions that can access the message/packet header and payload, access the NIC and host memory, and issue new network operations (e.g., RDMA put or gets) are marked with 👍. Solutions that provide a predefined set of actions that can be composed among themselves (e.g., P4 match-actions or Portals 4 triggered operations) are marked with 🖒. Solutions providing only predefined functions are marked with 👎.

(G) Granularity. Actions can be applied to full messages (👎), requiring to first fully receive the message, or to single packets, as they are received (🖒). Solutions enabling both types of actions are marked with 👍.

(U) Usability. It defines which entities can install actions into the network. In-network-compute solutions enabling user applications and libraries (even in multi-tenant settings) to install actions are marked with 👍. Solutions that require elevated privileges, service disruption, and/or device memory flashing to install new actions are marked with 👎.

Among the solutions of Table 1, sPIN is the only one letting user-space applications define per-message or per-packet tasks (called handlers) that are executed in the NIC. sPIN handlers can access and modify packet data, share NIC memory, and issue NIC and DMA commands. Applications can benefit from this in many ways, i.e., an MPI implementation can perform the processing of derived datatypes directly on the NIC (transparent to the user), custom reduction operators written in C or Fortran could be compiled for and executed on the NIC by MPI (assuming they do not make use of libraries not available on the NIC), or an application developer could write a part of the applications logic as a packet-handler. All such use-cases have been demonstrated [30]. In the future we see even more opportunities for offloading parts of an application when applications are expressed in a data-centric way which lends itself to be compiled for different accelerators, such as DaCe [31]. Such a framework could potentially offload parts of the application logic to the NIC automatically, making decisions about which parts to offload based on I/O analysis [32], while the offloading would be fully transparent to the user.

While the above comparison focuses on general capabilities, we can also compare in-network computing solutions from the angle of use-cases. The Allreduce collective for example could benefit from in-network compute, since it allows to reduce the total volume of bytes transferred between main memories of different nodes (since the reduction could happen already at the fabric level). To enable this, switches themselves need to be "smart". In the following we give an overview on smart switches. Performing this survey has led us to the conclusion that this area leaves room for improvement and we developed FLARE [8] to address some of the issues noted.

3.3 Overview of in-network reduction solutions

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.
Fixed-function switches Solutions targeting fixed-function switches [33, 34, 2, 5, 35, 36, 37, 38] 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 [33, 34] can be used on Mellanox's networks, and similar solutions are provided on Tofu [5], and Cray's Aries networks [2]. 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 [5, 2].

FPGAs Another set of solutions targets *Field-Programmable Gate Arrays* (FPGAs) [39, 40], trading performance for some flexibility. Some of these solutions [39] work by placing the FPGA between the hosts and the switch. Other solutions instead connect the FPGA only to the switch [40], 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 [39] to six 100Gbps ports [40].

Programmable switches Last, some solutions target programmable switches [41, 42, 43], 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) [44, 45, 46, 47], and can be configured with the P4 programming language [48, 49, 50]. In such architecture, each packet traverses a pipeline of $10 - 20$ stages [51], 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 [52, 51], 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. Some existing hardware can also only perform 32 operations per packet [42, 41], 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 [42].

We identified the following *flexibility limitations* in existing in-network allreduce solutions, that we summarize in Table 2, 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 [33], that cannot be customized nor extended (Table 2, 👎). In-network aggregation solutions for programmable switches can be customized, but they are still limited by the hardware (👍). For example, existing programmable switches do not have floating-point units, and do not support multiplication/division for integer data [45, 42, 41]. 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 [53, 54, 55, 56] 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.

F2 - Sparse data Many applications need to reduce sparse data [53, 57, 43, 58], 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 [43]. 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 [43]. 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.

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 [59, 60]. However, some aggregation functions (e.g., floating-point summation [61, 62, 63, 64, 65]) 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 [33, 39]. 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 (👍).

	Fixed-Function Switches							FPGAs		Progr. Switches			
	[33]	[34]	[2]	[5]	[35]	[37]	[38]	[39]	[40]	[41]	[42]	[43]	**Flare**
F1	👎	👎	👎	👎	👎	👎	👎	👎	👎	🖒	🖒	🖒	👍
F2	👎	👎	👎	👎	👎	👎	👎	👎	👎	👎	👎	🖒	👍
F3	?	👍	?	?	👍	?	👎	👍	👍	👎	👎	👎	👍

Table 2: Comparison between existing in-network allreduce solutions. F1: Custom operators and data types, F2: Sparse data, F3: Reproducibility. 👍: provided, 🖒: partially provided, 👎: not provided, ?: unknown.

3.4 Our RISC-V based SmartNIC Architecture

In this section we give a summary of the PsPIN Architecture, which we developed to address the shortcomings of existing SmartNICs pointed out above. We want to emphasize that our solution is based European technology, namely on the RISC-V architecture, and is fully open, i.e., a sythesizable hardware model can be downloaded by any interested party.

PsPIN builds on top of the PULP (parallel ultra-low power) platform [66], a silicon-proven [67] and open [68] architectural template for scalable and energy-efficient processing. PULP implements the RISC-V ISA [69] and organizes the processing elements in clusters: each cluster has a fixed number of cores (32-bit, single-issue, in-order) and single-cycle-accessible scratchpad memory. The system can be scaled by adding or removing clusters. *We have implemented all hardware components of PsPIN in synthesizable hardware description language (HDL) code.*

PsPIN has a modular architecture, where user-defined packet handlers are executed on handler processing units (HPUs). The HPUs are grouped into processing clusters. The HPUs are implemented as RISC-V cores, and each cluster is equipped with a single-cycle access scratchpad memory called L1 memory. All clusters are interconnected to each other (i.e., HPUs can access data in remote L1s) and to three off-cluster memories (L2): the packet buffer, the handler memory, and the program memory. Figure 1 shows an overview of how PsPIN integrates in a generic NIC model and its architecture. We adopt a generic NIC model to identify the general building blocks of a NIC architecture.

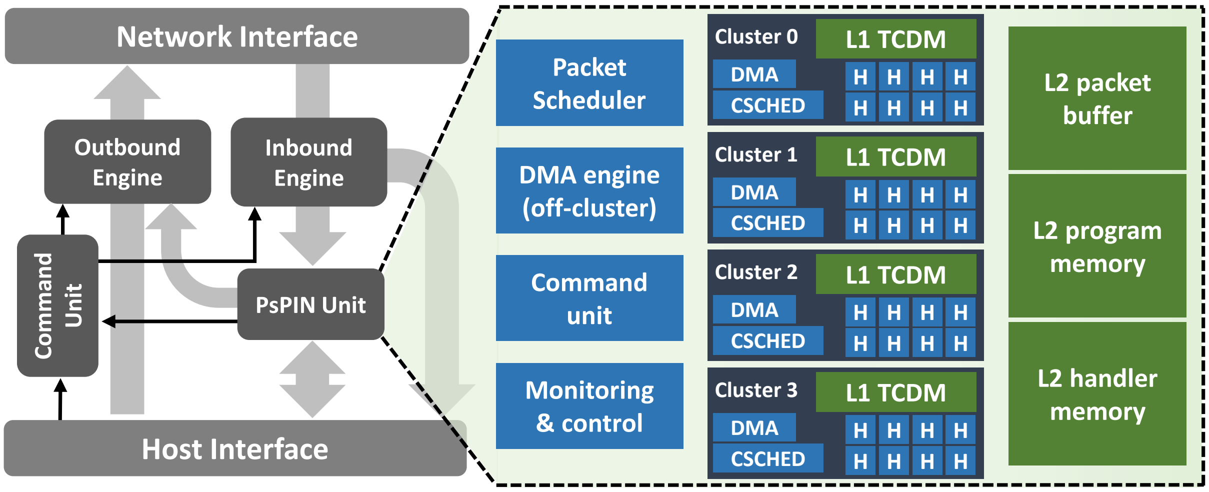

Figure 1: NIC model and PsPIN architecture overview.

Host applications access program and handler memories to offload handlers code and data, respectively. The management of these memory regions is left to the NIC driver, which is in charge of exposing an interface to the applications in order to move code and data. The toolchain and the NIC driver extensions to offload handlers code and data are out of the scope of this work. Once both code and data for the handlers are offloaded, the host builds an execution context, which contains: pointers to the handler functions (header, payload, and completion handlers), a pointer to the allocated handler memory, and information on how to match packets that need to be processed according to this execution context. The execution context is offloaded to the NIC and it is used by the NIC inbound engine to forward packets to PsPIN.

Receiving data. Data is received by the NIC inbound engine, which is normally interfaced with the host for copying it to host memory. In a PsPIN-NIC, the inbound engine is also interfaced to the PsPIN unit. The inbound engine must be able to distinguish packets that need to be processed by PsPIN from the ones taking the classical non-processing path. To make this distinction, it matches packets to PsPIN execution contexts and, if a match is found, it forwards the packet to the PsPIN unit. Otherwise, the packet is copied to the host as normal. While some networks already have the concept of packet matching (e.g., RDMA NICs match packets to queue pairs), in others this concept is missing and needs to be introduced to enable packet-level processing.

Packets to be processed on the NIC are copied to the L2 packet buffer. Once the copy is complete, the NIC inbound sends a Handler Execution Request (HER) to PsPIN's packet scheduler. The HER contains all information necessary to schedule a handler to process the packet, which are a pointer to the packet in the L2 packet buffer and an execution context. If the packet buffer is full, the NIC inbound engine can either back pressure the senders [70], send explicit congestion notifications [71], drop packets, or kill connections [12]. The exact policy to adopt depends on the network in which PsPIN is integrated and the choice is similar to the case where the host cannot consume incoming packets fast enough.

The packet scheduler selects the processing cluster that processes the new packet. The cluster-local scheduler (CSCHED) is in charge of starting a DMA copy of the packets from the L2 packet buffer to the L1 Tightly-Coupled Data Memory (TCDM) and selecting an idle HPU (H) where to run handlers for packets that are available in L1. Once the packet processing completes, a notification is sent back to the NIC to let it update its view of the packet buffer (e.g., move the head pointer in case the packet buffer is managed as a ring buffer).

Sending Data. Packet handlers, in addition to processing the packet data, can send data over the network or move data to/from host memory. To send data directly from the NIC, the sPIN API provides an RDMA-*put* operation: When a handler issues this operation, the PsPIN runtime translates it into a NIC command, which is sent to the NIC outbound engine. If the NIC outbound engine cannot receive new commands, the handler blocks waiting for it to become available again. The NIC outbound can send data from either the L2 packet memory, the L2 handler memory, or L1 memories, or it can specify a host memory address as data source, behaving as a host-issued command. To move data to/from the host, the handlers can issue DMA operations: These operations translate to commands that are forwarded to the off-cluster DMA engine, which writes data to host memory through PCIe.

We have evaluated both the area and power requirements as well as the latency and throughput of the proposed architecture on a variety of benchmarks, we will summarize our findings below.

We synthesized PsPIN in GlobalFoundries' 22 nm fully depleted silicon on insulator (FDSOI) technology using Synopsys DesignCompiler 2019.12, and we were able to close the timing of the system at 1 GHz.

Area and power measurements are summarized in Table 3. Including memories, the entire accelerator has a complexity on the order of 95 MGE.[1] Of the overall area, the

[1]One gate equivalent (GE) equals 0.199 μm^2 in GF 22 nm FDSOI.

four clusters (including their L1 memory and the intra-cluster scheduler) occupy 43 %, the L2 memory 51 %, the inter-cluster scheduler 3 %, and the inter-cluster interconnect and L2 memory controllers another 3 %. The L2 memory macros occupy a total area of 9.48 mm^2. Depending on the NIC architecture where PsPIN is integrated into, the L2 packet buffer could be mapped to the NIC packet buffer, saving memory area. The area of the clusters is dominated by the L1 memory macros, which take 1.65 mm^2 per cluster. The instruction cache and the cluster interconnect have a complexity of ca. 700 kGE per cluster, which corresponds to ca. 0.2 mm^2 at 70 % placement density. Each core has a complexity of ca. 50 kGE, which corresponds to ca. 0.014 mm^2. The total cluster area is ca. 1.99 mm^2. The total area of our architecture is ca. 18.5 mm^2. For comparison, from [72, 73] it can be inferred that a Mellanox BlueField SoC, scaled to 16 ARM A72 cores (22 nm), would occupy ca. 51 mm^2.

Die area is an factor for economical viability of a design, the cost of a die roughly scales with die area raised to the fourth power [74], since a smaller die area allows to place more dies on a fixed sized/priced wafer. A smaller die size also improves the yield, assuming defects are randomly distributed across the wafer area.

Component	**Area** (mm^2)			**Power** (W)		
	Unit	Total	Perc.	Unit	Total	Perc.
PsPIN	18.47	18.47	100.0%	6.08	6.08	100.0%
↳ **L2 memories** (×1)	9.48	9.48	51.3%	1.09	1.10	18.1%
↳ **Interconnect** (×1)	0.57	0.57	3.0%	0.71	0.71	11.7%
↳ **Cluster** (×4)	1.99	7.95	43.0%	0.94	3.77	62.0%
↳ **L1** (×1)	1.65	1.65	82.9%	0.52	0.52	55.3%
↳ **Core** (×8)	0.01	0.08	4.0%	0.02	0.14	15.3%
↳ **Instr. cache** (×1)	0.08	0.08	4.0%	0.14	0.14	15.1%
↳ **Interconnect** (×1)	0.06	0.06	3.0%	0.11	0.11	11.3%

Table 3: Area and energy of main PsPIN components. Percentages are relative to the parent component.

We derive a worst-case upper bound for the power consumption of our architecture by assuming 100 % toggle rate on all logic cells and 50/50 % read/write activity at each memory macro. The overall power envelope is 6.1 W, 99.8 % of which is dynamic power (**S6**). The four clusters consume 62 % of the total power, ca. 3.8 W. Within each cluster, the L1 memory consumes ca. 55 % of the power. The L2 memory consumes 18 % of the total power, ca. 1.1 W. The inter-cluster scheduler consumes 8 % of the total power, ca. 0.5 W. The inter-cluster interconnect and L2 memory controllers consume 11.7 %, ca. 0.7 W. As our architecture offers 32 HPUs, the power normalized to the number of HPUs is 190 mW.

Because the interconnection network consumes a large fraction of the overall system power (from 15% to 50% depending on the system load [75]), any reduction in the power consumption of NICs would also help in reducing the power consumption and thus the running cost of the system. Our power envelope of 6.1 W is a 12-fold reduction from the 75 W minimum power requirement given by NVIDIA for the Bluefield 2P Series NICs [76].

To evaluate the performance of PsPIN for realistic packet handlers, we select a set of use cases ranging from packet steering to full message processing. We first show the throughput they can achieve on PsPIN, then we measure the per-core throughput achievable on PsPIN RISC-V and compare it to the one achieved on more complex and powerful, but bigger, architectures such as x86 and ARM. We simulate a network with zero inter-packet-delay, in order to not make our results network-bound and show the

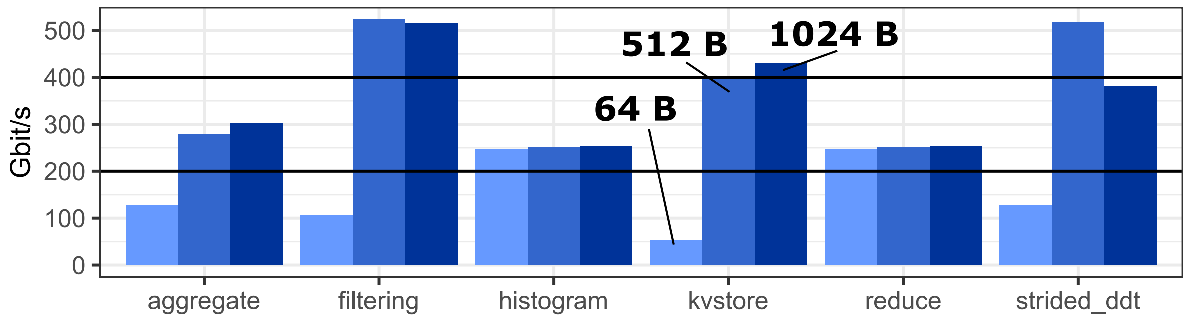

Figure 2: Handler throughput on PsPIN for different packet sizes.

maximum achievable throughput. The considered use cases are:

Data reduction. Reducing data of multiple messages is a core operation of collective reductions [77] and one-sided accumulations [78]. Given n messages, each carrying m data items of type t, this operation computes an array of m entries of type t where entry i is the reduction of the i-th data item across the n messages. We benchmark an instance of this use case (named *reduce*) with 512 packet, each carrying 512 32-bit integers. Payload handlers accumulate data in L1 using the sum operator. The completion handler informs the host that the result is available with a a direct host write command.

Data aggregation. Utilized in, e.g., data-mining applications [79], this operation consists in accumulating the data items carried by a message. This benchmark (*aggregate*) uses a 1 MiB message of 32-bit integers that are summed up in L1. The completion handler copies the aggregate to host memory.

Packet filtering/rewriting. Typical of intrusion-detection, traffic monitoring, and packet sniffing systems [80]. For each packet, it queries an application-defined hash table (in L2) by using the source IP address (32-bit) as key. If a match is found, the UDP destination port is overwritten with the matched value and written to host memory. This benchmark, named *filtering*, uses 512 messages and a hash table of 65,536 entries.

Key-Value cache. A key-value store (*kvstore*) cache on the NIC. The cache is stored in L2 and is implemented as a set-associative cache to limit the L2 accesses needed to maintain the cache (e.g., eviction victims are chosen within a row). We generate a YCSB [81] workload of 1,000 requests (50/50 read/write ratio, θ=1.1). The cache associativity is set to 4 and the total number of entries is set to 500. The set is determined as the key (32-bit integer) modulo the number of sets.

Scatter. This use case (*strided_ddt*) models data transfers that are copied to the destination memory according to a receiver-specified memory layout [77, 30]. This benchmark sends a 1 MiB message that is copied to host memory in blocks of 256 bytes and with a stride of 512 bytes. The layout description (i.e., block size and stride) is stored in L2.

Histogram. Given a set of messages, it summarizes data items by value. This application is common in distributed join algorithms [82]. In our instance, we receive 512 messages, each carrying 512 integers randomly generated in the $[0, 1024]$ interval. The handlers count how many data items per value have been received and finally copy the histogram to the host.

Figure 2 shows the throughput achieved by the considered handlers on PsPIN for different packet sizes. We observe that PsPIN achieves 400 Gbit/s for *filtering*, *kvstore*, and *strided_ddt* already for 512 B packets. In the other cases, handlers are compute-intensive, and they operate on every 32-bit word of each received packet. Nonetheless, PsPIN achieves more than 200 Gbit/s, which the state-of-the-art network speed, from 512 B packets. Thanks to the modularity of this architecture, a scenario where 400 Gbit/s must be sustained also for this type of workload can be satisfied by doubling the number of processing clusters.

3.5 Flexible in-network Allreduce

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 [83]. Allreduce is widely used in many applications, including scientific applications [84, 85], deep learning [86], graph processing [58, 87], big data analytics [88], and others, and recent studies [84] 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*) [89]. 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 [33, 38, 42]. 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 [75]), any reduction in the network traffic would also help in reducing the power consumption and thus the running cost of the system.

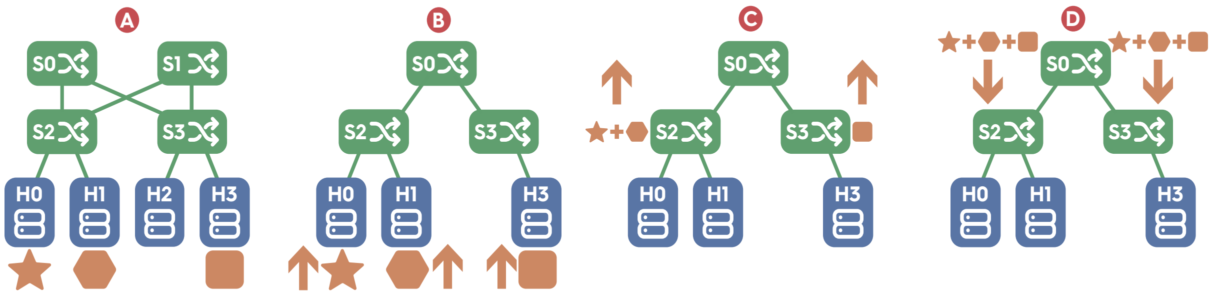

Figure 3: Example of an in-network allreduce.

If the hosts participating in the reduction span across multiple switches, the aggregation can be done recursively, as shown in Figure 3. 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* – Ⓑ). After a switch aggregates all the data coming from its children, if it is an intermediate switch in the tree, it sends the 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 3.3. 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 [7], and a set of aggregation algorithms for exploiting the switch architecture at best. PsPIN is an open-source multi-cluster RISC-V architecture [90], implementing the sPIN programming model [29], 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, e.g, the programmer is not required to write FPGA code, or constrained to a special purpose language like P4.

Figure 4 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* [91, 92] 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[2]), 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 [93], specifying the functions to execute for each packet, based on the values of specific packet fields (e.g., *EtherType* in the Ethernet header [94], or IP optional headers).

Differently from existing programmable switches [48, 45, 46, 95, 96], that implement the processing unit through *Reconfigurable Match Action* (RMT) tables [44, 47], we consider the processing unit to be implemented as a PsPIN unit [7], highlighted in the right part of Figure 4. PsPIN is a clustered RISC-V built on top of the PULP [66] 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[3]. 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 Ⓗ) 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 [97] that can execute sPIN *handlers* [29] 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. Each cluster also has

[2]By doing so, packets that do not need additional processing are not further delayed.

[3]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.

a DMA engine that can be used to access a globally shared 4MiB memory (*L2 handler memory*).

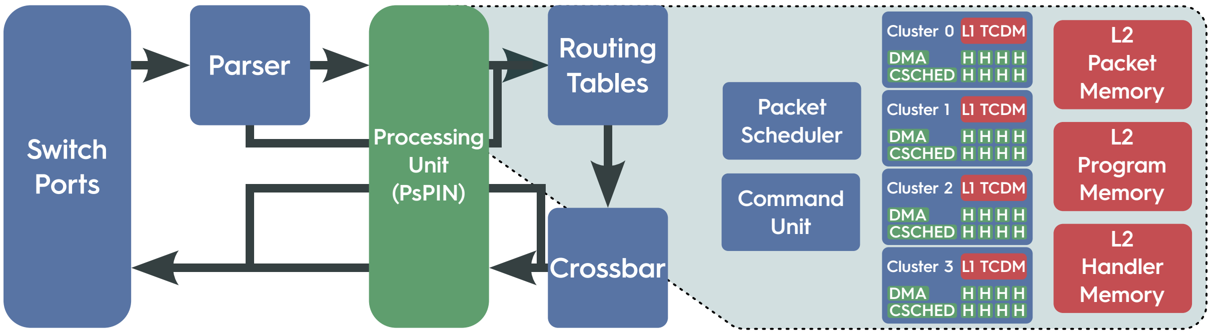
Switch Ports
Parser
Processing Unit (PsPIN)
Routing Tables
Crossbar
Packet Scheduler
Command Unit
Cluster 0 L1 TCDM DMA CSCHED H H H H H H H H
Cluster 1 L1 TCDM DMA CSCHED H H H H H H H H
Cluster 2 L1 TCDM DMA CSCHED H H H H H H H H
Cluster 3 L1 TCDM DMA CSCHED H H H H H H H H
L2 Packet Memory
L2 Program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

Figure 4: PsPIN switch high-level architecture.

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* [45], 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[4]. 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 [7] 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 [7], most of which is L1 memory (1.64mm^2). We also add an FP32/FP16 *Floating Point Unit* (FPU) [98] 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 [99]. 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 [3] to 7nm [100] processes.

In Flare, before starting the aggregation, the application sends a request to a network manager node [33, 34], 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

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

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 [33, 34].

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 [33, 34, 2, 5, 42, 43]. 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 [33, 34].

For the full discussion of the implementation of different allreduce algorithms and our evaluation methodology we refer the reader to our publication [8] on Flare. In this deliverable document we simply want to give an architecture overview and highlight key performance indicators in the following.

We implemented the different aggregation algorithms in the PsPIN cycle-accurate simulator [7]. 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 [7] 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 5 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 [42] and SHARP [33, 34]. SwitchML runs on programmable Tofino switches [45], can only process integer elements, and achieves a maximum bandwidth of 1.6Tbps [42]. SHARP is a solution running on Mellanox's fixed-function switches [101], 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 [34] (32 ports at 100Gbps), and we use this as a reference.

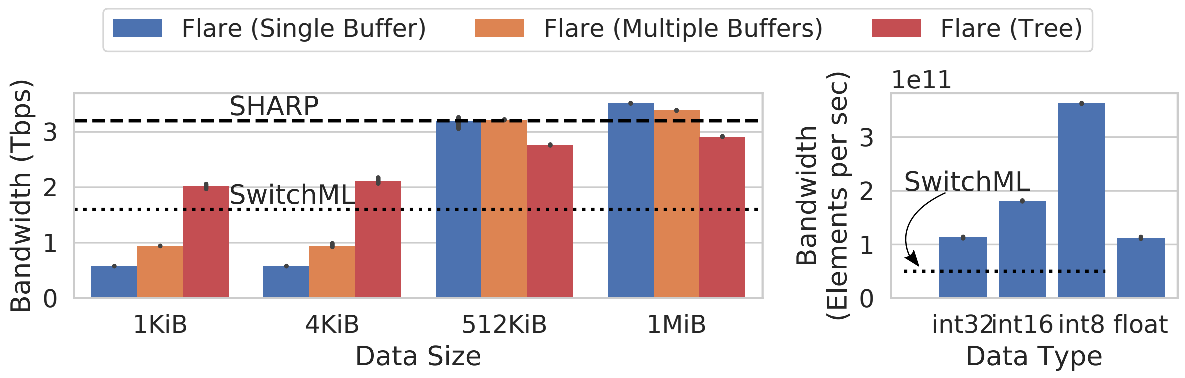

Figure 5: Bandwidth for different data size and data types.

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. This leads, for example, to significant performance improvements when dealing with sparse data.

We also report in Figure 5 the bandwidth (in elements aggregated per second), for different data types, for the aggregation of $1MiB$ 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.

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.

To analyze the performance of the Flare sparse allreduce at scale, we extended the SST simulator [102] 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 [57]. We then tuned the simulator parameters so that the bandwidth of the switches matches with that obtained through the cycle-accurate PsPIN simulator.

FLARE performance depends on the amount of overlapping indexes. To simulate a realistic scenario, we gathered the data exchanged during a Resnet50 [103] 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 ($\sim$0.2% density). We reproduce the same data on a simulated 2-level fat tree network [104] built with 8-port 100Gbps switches, connecting 64 nodes with 100Gbps network interfaces. We report in Figure 6 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.

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

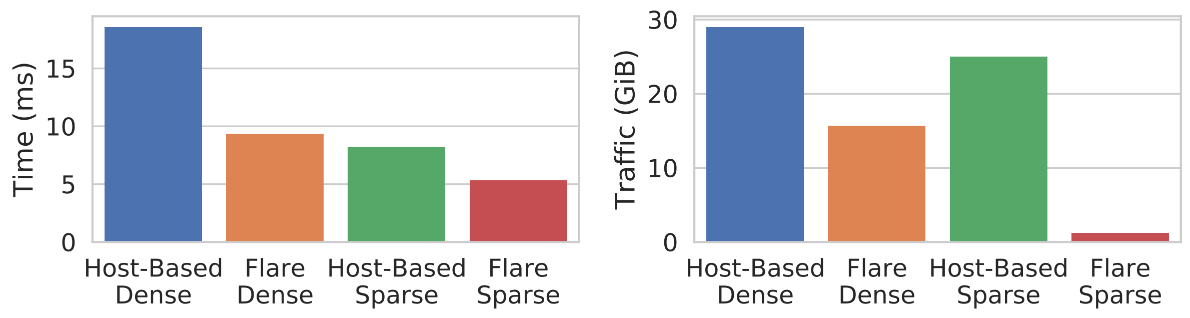

Figure 6: Execution time and network traffic for a 64 nodes allreduce executed on a simulated 2-levels fat tree network. The data to be reduced contains the gradients exchanged during a sparsified ResNet50 training iteration.

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.

4 Update on novel network programming models

In this section we present progress in enabling the use of novel hardware through MPI and GASPI.

4.1 MPI persistent and planned collectives

Since the completion of D2.1 the MPI-4.0 standard has been ratified by the MPI forum [105], including EPiGRAM-HS representatives Daniel Taylor and Oliver Thomson Brown (UEDIN). This new version of the standard includes 'persistent collectives', which were introduced to the standard right at the start of the project [106, 107]. The principle behind persistent collectives is that the application developer can amortise the cost of repeated collective communications, by performing an initialisation step which includes specifying the send and receive buffers, and processes involved. Furthermore this allows the library developer to potentially make more expensive optimisations on the grounds that the communication will be repeated many times. Initialisation is a blocking operation, but the communications themselves are non-blocking, allowing overlap of computation and communication. Many commonly used communication patterns can benefit from persistence. Collective operations benefit most from compute in the network, and persistence further bolsters this by giving the implementation advanced warning of what reduction operations (if any) will be being performed regularly. In particular, we note that persistent collectives could be sensibly prioritised for hardware acceleration by SmartNICs.

In order to enable persistent collectives to benefit from heterogeneous systems, UEDIN developed an implementation using only one-sided communication protocols (known as planned collectives). The principle behind this was that using one-sided communication allows the entire communication process to more easily be offloaded to simpler devices such as an FPGA, freeing up the host CPU for computational work. Such an approach is also recommended in D4.5 to help enable communication between FPGAs, where polling for messages is very expensive. This work was implemented within an experimental branch of OpenMPI, a widely-used open-source library implementing the MPI standard [108]. The advantage of this approach is that the code can be contributed back upstream to OpenMPI, allowing for rapid dissemination to the community. The implementation of the MPI_Alltoallv collective operation, which was identified as the most critical to the IFS dwarf application, could be used as a template to extend the approach to other collectives. Unfortunately, this decision has also added complexity to the implementation process. Although it is complete and works in test cases (see D2.4 "Report on final implementation of MPI and GPI extensions for heterogeneous systems with distributed GPUs and FPGAs"), we have encountered bugs when trying to use the implementation within larger applications such as the IFS code. We have identified that these are related to other areas of the OpenMPI library, and are working to put a detailed bug report together for the OpenMPI developers to tackle the issue, including reproducing the bug on multiple platforms. The crux is that the one-sided communication interface fails when using dynamic windows across multiple nodes or on more than two processes.We have been able to reproduce the bug in the latest release version of OpenMPI (4.1.1) using a standard simple test code, as well as with the Intel MPI benchmarks [109, 110], and verified that those codes do not fail when using other MPI libraries. It is hoped that once these issues are resolved we will be able to contribute the source code back to OpenMPI, as originally intended. While this will likely fall outwith the timeline of EPiGRAM-HS, it will form part of EPiGRAM-HS' legacy.

4.2 GASPI

4.2.1 GPI support for interconnects

The GASPI standard API leverages the remote completion and one-sided communication capabilities of RDMA. GPI-2, the actual implementation of GASPI, works with devices supporting Infiniband and RoCE, apart from the standard Ethernet/TCP fabric. The GPI-2 library is particularly well adapted to Mellanox Infiniband hardware, but other networks with RDMA capabilities can also be considered. Support for Cray/GNI and Omnipath networks is available as in-development branches. Particular technologies for more efficient collectives (as the Mellanox CORE-Direct) has been also considered as research work.

4.2.2 Change of GASPI specification to support Omnipath

In order to be able to support Omnipath a change in the GASPI specification has been necessary. For OmniPath, the granularity of the cache coherency protocol used for write operations of the network interface is (at least can be) 2 Bytes. Using 4 Byte integer types as $gaspi\_notification\_t$ may produce a race between the $gaspi\_notify\_reset$ and the write operation for the notification value of the network interface.That means that the notification is not reset to zero whereas the semantics assures that. The next time the application will check on this notification by a $gaspi\_notify\_waitsome$, the application will come to the conclusion that a message associated with that notification id is locally visible which might not be the case. In order to avoid that race, the implementation needs to reduce the size of the integer type. However, this limits the range of allowed notification values. Therefore, one needs to have a possibility to inspect the maximum value which is allowed in order to provide portability.The change has been proposed in the last GASPI meeting in January 2020.

4.2.3 GPI-NIC

The GPI-NIC is a project to develop a high-performance communication device that enables data transport on heterogeneous architectures where application processing units (APUs) and FPGA devices have to communicate at the same time to each other. The GPI-NIC will directly implement GPI communication primitives in hardware and extend the remote direct memory access (RDMA) approach into a new self-programmable RDMA (SP-RDMA) model by using side-channel data. Recent work towards the GPI-NIC has been carried out and reported as part of the EuroEXA project (https://euroexa.eu/), which is a hardware-software co-design project funded by EU Horizon 2020.

Current development towards the GPI-NIC concentrates on a Xilinx embedded system with application processing unit (APU) and the FPGA programmable logic. The APU includes ARM CPUs as processing units. Software, firmware and hardware related topics need to be taken into account to implement the network interface card complemented by an efficient GPI memory model. The final design shall provide GPI hardware for single-sided board-to-board communication, concurrent access to GPI hardware by the APU and accelerated kernels (FPGA IP cores), as well as local communication of the APU with the FPGA.

As first step the local communication with the FPGA has been addressed, which includes the implementation of the GPI memory model. Currently, the GPI-NIC implements two different operation modes: i) Direct data transport, where payload and metadata are combined into one single package, and ii) a DMA mode, where DMA descriptors are used to describe the data transport of large memory blocks.

To evaluate our framework, we performed a set of latency benchmarks between two UltraScale+ ZU9 boards connected by an optical-fiber cable using the GTH-Transceivers and the Aurora 64b/66b protocol. We were able to achieve two-way latencies of around 548 ns and one-way latencies of 274 ns for 4-byte and 8-byte notification values by writing into GPI's notification vector. This allows for super-fast memory driven remote notifications initiated by APUs using one-sided communications. To our knowledge, these numbers are the fastest latencies ever reported for a GPI node to node communication. Compared to current InfiniBand NICs, these numbers are more than $3\times$ better in performance.

4.3 UDJ

The Universal Data Junction (UDJ) library [111] is a communication library developed by the Cray EMEA Research Lab for transporting distributed data between parallel high performance computing applications. UDJ is not a network programming model in the same context as MPI and GASPI, rather a higher-level network-focused data movement library that has been the focus of on-going research in this project that links WP2 with the memory-focused WP3. The intended use case of the UDJ library is to couple two parallel applications that are running simultaneously, but may be scheduled separately by one or more users on distinct resources connected via file system or high speed network. This might be, for example, a simulation code running on dense CPU-based compute nodes and a visualisation/analysis code running on GPU-enabled nodes. UDJ provides a common API for both applications to describe a parallel data distribution scheme and communicate data objects using that scheme via either a shared file system or via, for example, MPI. UDJ supports the use of MPI Dynamic Process Management (DPM) combined with credential support, such as LibDRC [112], for cross-job and/or cross-user communication on HPE Cray HPC systems. UDJ also has a built-in redistribution API that is able to redistribute data for asymmetric operations where each application has a different number of processes, or a different parallel data distribution scheme.

As part of the WP2 efforts, the Mamba heterogeneous memory library developed in WP3 has been extended to support distributed layouts, as detailed in Deliverable 2.4. This allows a parallel application to describe a set of distributed data consisting of a Mamba array on each process that is conceptually similar to a simple implementation of a partitioned global address space (PGAS). This parallel description, as well as the array data, can then be collectively passed to UDJ to transport, and possibly redistribute, over the network. Work is on-going, which will likely continue beyond the scope of EPiGRAM-HS, to further exploit this in application coupling scenarios.

Furthermore, the distributed Mamba layout is used to enable a similar distributed data object approach in the Maestro H2020 project, which is using distributed Mamba arrays combined with UDJ redistribution API to support the communication and redistribution of distributed data objects in the Maestro workflow management software.

5 Conclusion

This deliverable document describes updates in the space of novel network hardware and programming models, since the previous D2.1 report on the same topic. Much has changed in that time, in particular an increase in the use and availability of programmable network interface cards. This was covered in detail in Sections 3.2 to 3.5, including details of *PsPIN*, an architecture developed at ETH Zürich to address shortcomings in the available SmartNICs. We have also provided a brief update in Section 3.1 on the status of hardware lines introduced in D2.1. Taking advantage of the additional compute power provided by these compute-in-network devices is crucial for networks at the size required for exascale, especially for applications that make heavy use of collective operations.

In Section 4 we talk about the software side of things, with a focus on MPI and GASPI. In MPI, the big news is the ratification of the MPI-4.0 standard, voted for this year by the MPI forum including members of the EPiGRAM-HS consortium. In Section 4.1 we also provide an update on the status of the implementation of a persistent MPI_Alltoallv using one-sided communication, which was being developed at UEDIN. Unfortunately, it has been blocked by a bug in the OpenMPI libary's implementation of dynamic windows, but details are being gathered to raise a bug report with the library developers. It is hoped that once resolved, the persistent collective developed by UEDIN will be usable. The purpose of implementing such an operation exclusively on one-sided communication is that it is more amenable to offloading to devices such as SmartNICs, and by being built in to the MPI library application developers who already make use of collectives will be able to benefit with little or no additional effort required.

Work on GASPI, led by Fraunhofer ITWM and described in Section 4.2, has also focused on offloading communication work entirely from host CPUs to SmartNICs. Work on a GPI-NIC is described, a programmable NIC which directly implements GPI communication primitives.

Finally, in Section 4.3, an update is provided on HPE's UDJ library, which is a higher-level library for transporting distributed data between parallel applications. This work links WP2 and WP3, as UDJ provides a transport layer for the Mamba heterogeneous memory library. While MPI and GASPI aim to give the application developer full control over data movement across the network within an application, UDJ sits on top of these and aims to simplify the process of moving data between applications, and enable heterogeneous workflows.

Overall, we conclude that enabling applications to take advantage of compute-in-network devices such as SmartNICs has become increasingly important over the life of the EPiGRAM-HS project, and while support has improved, it could be improved further. This is of particular importance to codes that make heavy use of collective operations, as it is here that in-network compute offers the greatest advantage. Ideally these new in-network compute devices would allow applications to be accelerated with no changes to their code. This could be done by, as an example, a major MPI library implementation making use of Flare, presented in Section 3.5.

Further work suggested by our findings include continuing to expand the functionality of SmartNICs to allow them to support and accelerate a wider range of MPI operations, fixing the one-sided communication implementation Alltoallv, pushing it back upstream to OpenMPI and then expanding the approach to other collectives. This latter work is also suggested by the findings of D4.5. On the GASPI side there is an obvious route forward through the continued development and evaluation of the GPI-NIC.

References

[1] D. Roweth, R. Alverson, and L. Kaplan. The gemini system interconnect. In *High-Performance Interconnects, Symposium on*, pages 83–87, Los Alamitos, CA, USA, aug 2010. IEEE Computer Society.

[2] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. Cray XC series network. *Cray Inc., White Paper WP-Aries 01-1112*, 2012.

[3] HPE. HPE Slingshot: The Interconnect for the Exascale Era. Technical report, 2021.

[4] Denis Foley and John Danskin. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[5] Yuichiro Ajima, Shinji Sumimoto, and Toshiyuki Shimizu. Tofu: A 6d mesh/torus interconnect for exascale computers. *Computer*, 42(11):36–40, 2009.

[6] Saïd Derradji, Thibaut Palfer-Sollier, Jean-Pierre Panziera, Axel Poudes, and François Wellenreiter Atos. The bxi interconnect architecture. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 18–25, 2015.

[7] 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.

[8] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. Flare: Flexible In-Network Allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC21)*. ACM, Nov. 2021.

[9] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020.

[10] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation NSDI 18)*, pages 51–66, 2018.

[11] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[12] B. W Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, T. Hoefler, A. B Maccabe, and T. Hudson. The Portals 4.2 network programming interface. *Sandia National Laboratories, November 2012, Technical Report SAND2012-10087*, 2018.

[13] Antoine Kaufmann, SImon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with FlexNIC. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 67–81. ACM, 2016.

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

[15] Whit Schonbein, Ryan E Grant, Matthew GF Dosanjh, and Dorian Arnold. INCA: in-network compute assistance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2019.

[16] Richard L Graham, Steve Poole, Pavel Shamis, Gil Bloch, Noam Bloch, Hillel Chapman, Michael Kagan, Ariel Shahar, Ishai Rabinovitz, and Gilad Shainer. ConnectX-2 InfiniBand management queues: First investigation of the new support for network offloaded collective operations. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 53–62. IEEE, 2010.

[17] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Hardware- and software-based collective communication on the Quadrics network. In *Proceedings IEEE International Symposium on Network Computing and Applications. NCA 2001*, pages 24–35. IEEE, 2001.

[18] Darius Buntinas, Dhabaleswar K Panda, and Ponnuswamy Sadayappan. Fast NIC-based barrier over Myrinet/GM. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 8–pp. IEEE, 2000.

[19] Mellanox BlueField SmartNIC. `https://www.mellanox.com/products/bluefield-overview`. Accessed: 2020-18-03.

[20] Broadcom Stingray SmartNIC. `https://www.broadcom.com/products/ethernet-connectivity/smartnic`. Accessed: 2020-18-03.

[21] Michael Attig and Gordon Brebner. 400 Gb/s programmable packet parsing on a single FPGA. In *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, pages 12–23. IEEE, 2011.

[22] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with eBPF: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2018.

[23] Jakub Kicinski and Nicolaas Viljoen. eBPF hardware offload to SmartNICs: clsbpf and XDP. `https://www.netronome.com/technology/ebpf/technical-papers/`.

[24] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. *arXiv preprint arXiv:2010.14145*, 2020.

[25] R Rajesh, Kannan Babu Ramia, and Muralidhar Kulkarni. Integration of LwIP stack over Intel DPDK for high throughput packet delivery to applications. In *2014 Fifth International Symposium on Electronic System Design*, pages 130–134. IEEE, 2014.

[26] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[27] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 345–362, 2019.

[28] Jiaxin Lin, Kiran Patel, Brent E Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259, 2020.

[29] 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.

[30] Salvatore Di Girolamo, Konstantin Taranov, Andreas Kurth, Michael Schaffner, Timo Schneider, Jakub Beránek, Maciej Besta, Luca Benini, Duncan Roweth, and Torsten Hoefler. Network-accelerated non-contiguous memory transfers. 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.

[31] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*, Nov. 2019.

[32] Grzegorz Kwasniewski, Marko Kabić, Tal Ben-Nun, Alexandros Nikolaos Ziogas, Jens Eirik Saethre, André Gaillard, Timo Schneider, Maciej Besta, Anton

Kozhevnikov, Joost VandeVondele, and Torsten Hoefler. On the Parallel I/O Optimality of Linear Algebra Kernels: Near-Optimal Matrix Factorizations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC21)*, Nov. 2021.

[33] 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., January 2017.

[34] 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, June 2020.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[55] 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.

[56] 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.

[57] 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.

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

[59] 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.

[60] Edward N. Lorenz. The predictability of a flow which possesses many scales of motion. *Tellus*, 21(3):289–307, 1969.

[61] 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.

[62] 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.

[63] 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.

[64] N. Revol and P. Theveny. Numerical reproducibility and parallel computations: Issues for interval algorithms. *IEEE Transactions on Computers*, 63(08):1915–1924, aug 2014.

[65] 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.

[66] 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.

[67] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca 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.

[68] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K Gurkaynak, and Luca Benini. PULPino: A small single-core RISC-V SoC. In *3rd RISCV Workshop*, 2016.

[69] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, and Volume I User level Isa. The RISC-V instruction set manual. *Volume I: User-Level ISA', version*, 2, 2014.

[70] InfiniBand Trade Association et al. InfiniBand architecture specification release 1.2. *http://www.infinibandta.org*, 2000.

[71] Kadangode Ramakrishnan, Sally Floyd, David Black, et al. The addition of explicit congestion notification (ECN) to IP. 2001.

[72] Hugh T Mair, Gordon Gammie, Alice Wang, Rolf Lagerquist, CJ Chung, Sumanth Gururajarao, Ping Kao, Anand Rajagopalan, Anirban Saha, Amit Jain, et al. A 20nm 2.5 GHz ultra-low-power tri-cluster CPU subsystem with adaptive power allocation for optimal mobile SoC performance. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 76–77. IEEE, 2016.

[73] Jungyul Pyo, Youngmin Shin, Hoi-Jin Lee, Sung-il Bae, Min-su Kim, Kwangil Kim, Ken Shin, Yohan Kwon, Heungchul Oh, Jaeyoung Lim, et al. A 20nm high-K metal-gate heterogeneous 64b quad-core CPUs and hexa-core GPU for high-performance

and energy-efficient mobile application processor. In *2015 IEEE International Solid-State Circuits Conference-(ISSCC) Digest of Technical Papers*, pages 1–3. IEEE, 2015.

[74] Randy H. Katz. Lecture 5: Cost, price, and price for performance, 1996.

[75] 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.

[76] NVIDIA BlueField-2 Ethernet DPU User Guide. `https://docs.mellanox.com/display/BlueField2DPUENUG/Specifications`, October 2021.

[77] Message Passing Interface Forum. MPI: A message-passing interface standard version 3.0, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[78] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. Remote memory access programming in MPI-3. *ACM Transactions on Parallel Computing (TOPC)*, 2(2):1–26, 2015.

[79] V Pradeep Kumar and RV Krishnaiah. Horizontal aggregations in SQL to prepare data sets for data mining analysis. *IOSR Journal of Computer Engineering (IOSRJCE)*, pages 2278–0661, 2012.

[80] Luca Deri. High-speed dynamic packet filtering. *Journal of Network and Systems Management*, 15(3):401–415, 2007.

[81] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[82] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. Distributed join algorithms on thousands of cores. *Proceedings of the VLDB Endowment*, 10(5), 2017.

[83] Message Passing Forum. MPI: A Message-Passing Interface Standard. Technical report, USA, 1994.

[84] 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.

[85] 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.

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

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

[88] 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.

[89] 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.

[90] 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.

[91] Hesam Zolfaghari, Davide Rossi, and Jari Nurmi. A custom processor for protocol-independent packet parsing. *Microprocessors and Microsystems*, 72:102910, 2020.

[92] 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.

[93] 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.

[94] Internet Control Message Protocol. `https://tools.ietf.org/html/rfc894`, April 1984.

[95] 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.

[96] 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.

[97] 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.

[98] S. Mach, F. Schuiki, F. Zaruba, and L. Benini. FPnew: An Open-Source Multiformat Floating-Point Unit Architecture for Energy-Proportional Transprecision Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(04):774–787, apr 2021.

[99] 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.

[100] Broadcom. Tomahawk4 BCM56990 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series, mar 2019.

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

[102] 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.

[103] 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.

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

[105] MPI Forum. Meeting agenda, June 2021. `https://www.mpi-forum.org/meetings/2021/06/agenda`.

[106] Daniel J. Holmes, Bradley Morgan, Anthony Skjellum, Purushotham V. Bangalore, and Srinivas Sridharan. Planning for performance: Enhancing achievable performance for mpi through persistent collective operations. *Parallel Computing*, 81:32–57, 2019.

[107] MPI Forum. Meeting agenda, September 2019. `https://www.mpi-forum.org/meetings/2018/09/agenda`.

[108] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[109] RookieHPC. MPI_Win_create_dynamic. `https://rookiehpc.com/mpi/docs/mpi_win_create_dynamic.php`. Accessed: 2021-10-29.

[110] Intel Corporation. Intel MPI Benchmarks. `https://github.com/intel/mpi-benchmarks`. Accessed: 2021-10-29.

[111] Aniello Esposito and Glendon Holst. In situ visualization of wrf data using universal data junction. In *Proceedings of the 5th Workshop on In Situ Visualization*, ISC '21, 06 2021.

[112] James Shimek and James Swaro. Dynamic RDMA credentials. In *In CUG 2016 Proceedings*, 2016.