**In-Network Computing for Collective Offload, Streaming Processing in the Network** Paper Report

Mahmoud Abdelhadi

School of Computing

Queen’s University

Kingston, ON, Canada

m.abdelhadi@queensu.ca

*Abstract*—The increasing of the system size and greater reliance on that systems opened new opportunities to utilize the current systems parallelism to achieve computational needs, achieving that new level requires a better architectures and new solutions. As a step towards a new data processing technique that involves the network itself to be the bigger part and letting the old processing ways to do more useful thing. In this paper I will introduce In-Network Computing for Collective Offload and Streaming processing, and how other companies applied it in their applications like Scalable Hierarchical Aggregation Reduction Protocol (SHArP) [1], High-performance streaming Processing in the Network (sPIN) [2] and In-Network Compute Assistance (INCA) [3], explaining each solution and showing some benchmarks.

Keywords; HPC; Collectives; Computing; Processing; Optimization.

# Introduction

We all know that the world is a huge network that has endless connections established between millions of devices, these connections carry a stream of data exchanged between those devices, going throw many switches and routers until it arrives to its destination. This data needs to be processed before it can be used, especially in the applications of High-Performance Computing (HPC) as it is not using only a point-to-point communication but also a high volume of collective communications which involves sending the data to more than one device, that requires more processing which lead to stealing more computational time from the CPU and preventing it from doing another more useful things. Sometimes the increase of processing this data in the devices will cause a bottleneck in the network, by feeding the nodes of this network with more packets and more not-processed data which will cause the network and the cluster to not perform as expected and it will increase the latencies in the network while the network itself is empty.

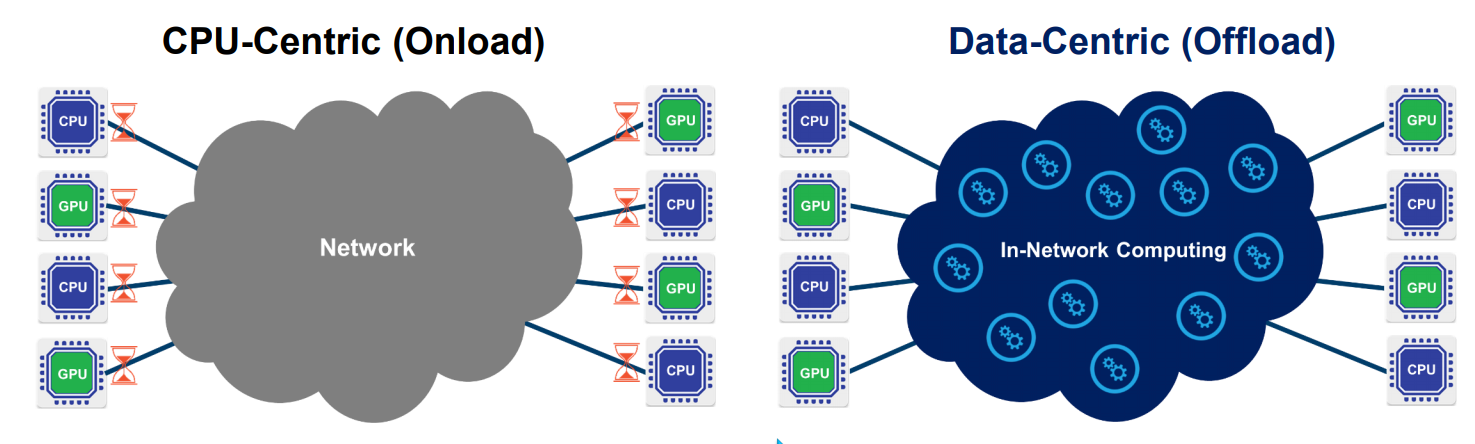
A solution to not fully-rely on the CPU and its computational speed proposed, specialized data processors offer remote direct memory access (RDMA) functions and enable tens of gigabytes per second transmission rates at sub-microsecond latencies in modern network interface cards (NICs).

However, RDMA only transports data between (virtual) memories of processes on different network endpoints. Different RDMA interfaces, such as OFED [4], uGNI/DMAPP [5], Portals 4 [6], or FlexNIC [7] provide varying levels of support for steering the data at the receiver. Yet, with upcoming terabits-per-second networks [8], we foresee a new bottleneck when it comes to processing the delivered data: A modern CPU requires 10-15ns to access L3 cache (Haswell: 34 cycles, Skylake: 44 cycles [9, 10]). However, a 400 Gib/s NIC can deliver a 64-Byte message each 1.2ns.

The main problem is that the packets are inserted into main memory, without considering the contents of the message itself. After that a lot of applications analyze the received messages and may change the structures of them to the application structure in the host memory (e.g., halo exchanges, parallel graph algorithms, database updates) even though this step can logically be seen as part of the data routing. This poses a barrier, very similar to pre-RDMA packet processing: CPU cores are inefficient message processors because their microarchitecture is optimized for computation. They require thread activation, scheduling, and incoming data potentially pollutes the caches for the main computation.

To address these limitations, researchers provided a new concept called “In-Network Computing” (Figure 1), which refers to the use of all the devices in a linked network (e.g., switches) to carry some of the huge load suppled to the CPUs and do the low-computational operations without sending it to the CPU, which will lead to offloading the data from the Processing Unit and give it some extra free time to do another high-computational operations.

In this paper, I am going to talk about In-Network Computing (Section two), address its capabilities and how some researchers applied this concept in Collective Communications and Stream processing in the network, followed by some applications (Section three) that used this concept, like sPIN [2], SHArP [1] and INCA [3], and ending this paper with a Conclusion (Section four).



**Figure 1: Illustrating the difference between processing the data in the network and processing it on the CPU [11]**

# In-network computing

As shown in Figure 1, the idea behind In-Network Computing is to make effective use of data where it is present in the system, instead of depending on all computing being done at the endpoints itself or bringing the data to the endpoints to be able to process it, In-Network Computing introduced a new ways that will allow the data to be manipulated in the fly and be processed before it reaches the end points and as it moves. There are several advantages of doing that, one is decentralizing the computation so computation can be more efficient when it is not done only by one part, and the other thing is data can be processed as it is moving throw the network, you can setup a pipeline when you are waiting the data being transferred from endpoint to the other endpoint and do something with it but you are processing it is moving in the network.

The challenge here is to find the sort of motifs that are usable and frequently used by apps and can really benefit from these optimizations, there are some that are very natural for HPC that depend on using communication libraries like MPI [12] or SHMEM [13] so they are very easy targets for optimization and applying this new technology in it.

The overall goal here is to reduce the latency and to get operations done faster, not just to process the data on the fly for the purpose of processing it there but to get overall speed-up in terms of latency and communication, so while doing computational operations based only at the end points, latency tend to take more overall time to do a certain product operation, the most expensive part in this operation is the computational part so it will take much fewer time if we can take advantage of computing this product operation in the network as it is moving, so collective operations are the best things that can take the benefit from processing the data in the network and there one can easily reduce the latency for this operations, this will cause a huge speedup overall application performance, and the main reason that will happen is that the CPU cores are inefficient message processors because their microarchitecture is optimized for computation, so doing this will leave processing the messages and the low-computational operations in the network and let the CPUs do the high computational operations, and even if the message has high computational operations using In-Network Computing will allow overlapping it with processing the message especially in case of multi packets messages.

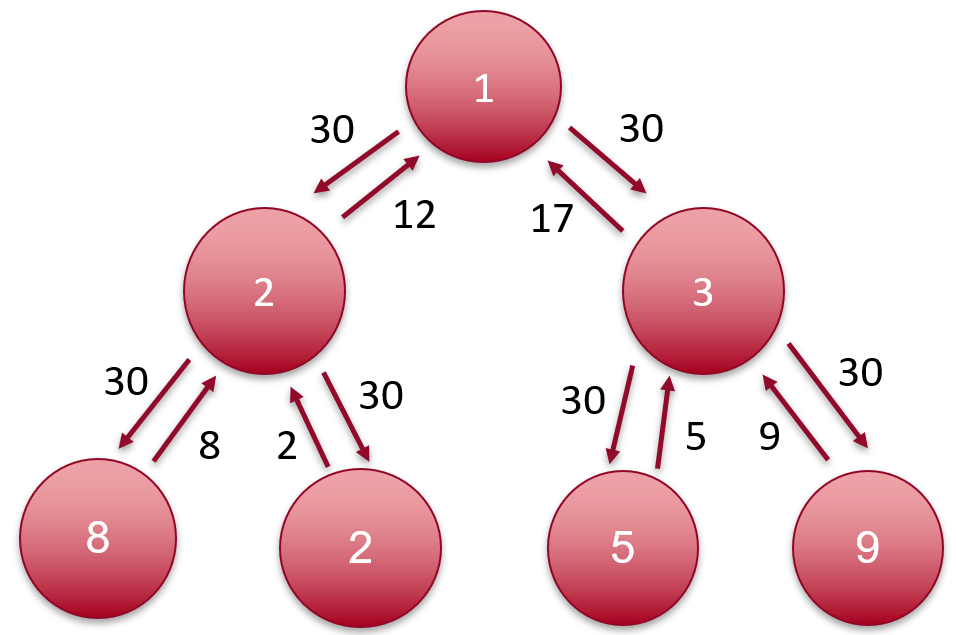
In the following section, you will see some applications that applied the In-Network Computing concept in different ways, I will explain how sPIN [2] NIC and its Handler Processing Units (HPUs) extended the success of RDMA and receiver-based matching to simple processing tasks that are dominated by data-movement, and how they enabled sPIN in existing NIC microarchitectures with typically very small but fast memories without obstructing line-rate packet processing. Also, you will see how SHArP [1] obtained large performance enhancements, with an improvement of a factor of 2.1 for an eight-byte MPI\_Allreduce() operation on 128 hosts, going from 6.01 to 2.83 microseconds. In the third application, I will show INCA can provide up to a 11% speedup for applications with minimal code modifications and between 25% to 37% when applications are optimized for INCA.

# In-Network Computing applicatios

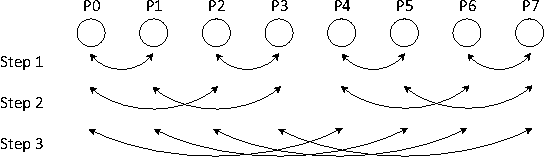
## SHArP

Scalable Hierarchical Aggregation Reduction Protocol (SHArP) introduced by Mellanox [14] in 2016. SHArP was firstly built for HPC and it was done to accelerate MPI operations, it is a part of the MPI flame communication frameworks focused on MPI to reduce the time of the operations by removing the CPU or GPU from the operations so the CPU become more free to do other things, also to enable overlap with that. It was firstly built for MPI\_Allreduce() operations.

There are multiple ways to implement Allreduce operations, and the most common implementation for it is the tree based way (Figure 2 a), sending the data within a group into a single node which will reduce the data and then doing that in hierarchies so we don’t have too large number of sending operations, doing that in a tree based will give us a multi stage implementation moving the data multiple times to do Allreduce it, and then broadcasting it with the same tree or using another implementation with a hardware multicast capabilities. Another implementation is using Recursive Doubling (Figure 2 b) which is sort of a binary tree, it’s a logarithmic operation with moving the data multiple times.



(**a) Allreduce operation using tree-based way**



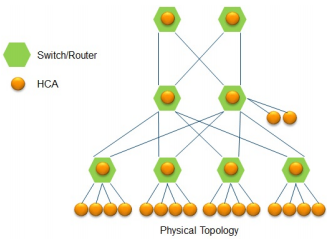
**(b) Allreduce operation using recursive doubling**

**Figure 2: Allreduce operation**

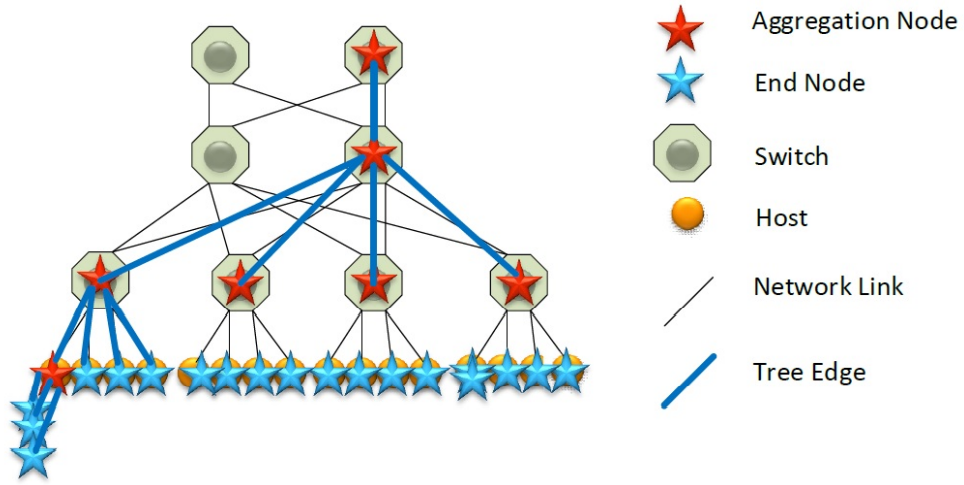
SHArP came to accelerate this process, while we are sending the data into the switches to reach the end nodes, the switches see all the data anyway because it will pass throw it, all we can do is just reducing the data within the switches because the switches is the only component in the network that can absorb data from multiple sources in an efficient way, and it is already connected to all the end nodes so it makes more since to reduce the data while it traverse the switch and move less data . In a case that switches do not directly connected to the end nodes as shown in (Figure 3) where switches connected to end nodes and the switches connected physically to have large numbers of nodes like what we have in the Fat-Tree topology, then what they did is building a hierarchical tree between those switches and do the reduction in the network, each switch will be responsible for a subset of the nodes and all of them will be working in parallel on the same layer, then next layer will aggregate data from multiple switches. After reducing the data, it will enable broadcasting the result to all the nodes using the same way the data aggregated in.

It was designed as flexible as possible to do this reduction, also it was built with the vision of HPC so it is very much similar to MPI API, it supports the data types important in the MPI and for its operations (e.g., min, min-loc, max-loc, XOR etc.). Later they added a 16-bit floating point and other capabilities when they saw an opportunity to support Artificial Intelligence and Machine Learning, with that they introduced a full solution for doing reduction in the network.

Figure 1 shows an example of a SHArP tree allocation over the physical network topology. Figure 1a shows an example of physical network fat tree topology, and Figure 1b is the SHArP tree allocation over this topology. Generally, since the SHArP tree is logical, it can be created over any topology.



**(a) Physical Network Topology**



**(b) Logical SHArP Tree. Note that in the SHArP abstraction an Aggregation Node may be hosted by an end-node.**

**Figure 3: SHArP Tree example.**

To show some result, a low-level verbs test, and the OSU collective latency test version 5.2 [15] for MPI\_Barrier() and MPI\_Allreduce() were used. All tests were run with a single process per host. The basic verbs-level tests consisted of a tight loop around individual collective operations, such as a SHArP-level barrier. This consists of two steps: 1) posting the SHArP barrier message to the HCA, and 2) polling the SHArP complete on queue for completion of the reduction operation. In a pipelined6Authorized licensed use limited to: Queen's University. Downloaded on November 22,2020 at 22:35:06 UTC from IEEE Xplore. Restrictions apply. algorithm, a single MPI-level message was fragmented into multiple SHArP messages which were simultaneously in flight. In addition, the impact of using the SHArP based aggregation support on the OpenFOAM [16] application is studied.

Some micro benchmarking has been done on the Native SHArP operations, Table 1 present the result of SHArP-based latency measurements for barrier and allreduce operations. In all of these measurements the hosts are equally distributed across eight leaf switches configured in a two-level fat tree topology with 100Gbps links.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Number of Hosts** | | |
| **Message Size [B]** | 32 | 64 | 128 |
| 0 | 2.59 | 2.57 | 2.63 |
| 8 | 2.66 | 2.68 | 2.79 |
| 16 | 2.72 | 2.70 | 2.86 |
| 32 | 2.72 | 2.85 | 2.92 |
| 64 | 2.89 | 2.92 | 3.04 |
| 128 | 3.06 | 3.10 | 3.25 |
| 256 | 3.89 | 3.99 | 4.21 |

**Table 1: Native SHArP Allreduce() average latency (μ seconds). Optimized system with turbo-mode off.**

Table 2 reports the latencies of MPI\_Allreduce() and MPI\_Barrier() compared with native SHArP data. The MPI integration increases latency by two to four tenths of a μ-second.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Number of Hosts** | | | | | |
| Size [B] | 32 | | 64 | | 128 | |
| MPI | Native | MPI | Native | MPI | Native |
| 0 | 2.83 | 2.59 (0.24) | 2.83 | 2.57 (0.26) | 2.88 | 2.63 (0.25) |
| 8 | 2.95 | 2.66 (0.29) | 3.10 | 2.68 (0.42) | 3.15 | 2.79 (0.36) |
| 16 | 2.89 | 2.72 (0.17) | 3.04 | 2.70 (0.34) | 3.03 | 2.86 (0.17) |
| 32 | 3.01 | 2.72 (0.29) | 3.17 | 2.85 (0.32) | 3.24 | 2.92 (0.32) |
| 64 | 3.06 | 2.89 (0.17) | 3.24 | 2.92 (0.32) | 3.24 | 3.04 (0.20) |
| 128 | 3.24 | 3.06 (0.18) | 3.44 | 3.10 (0.34) | 3.57 | 3.25 (0.32) |

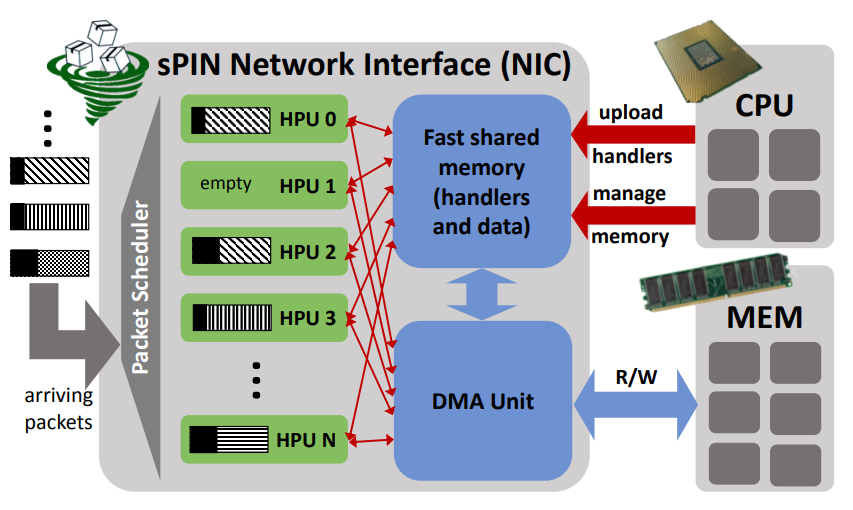
**Table 2: A comparison of MPI Allreduce() and Native Allreduce() average latencies (μ-seconds) on optimized system. The latency difference (Diff) is in units of μ-seconds.**

You can check the SHArP [1] paper for more benchmarking results.

## sPIN

High-performance streaming Processing in the Network or is they call it “beyond RDMA”, is firstly introduced by ETH Zürich in 2017. It was introduced to extend the success of RDMA and receiver-based matching, in particular, sPIN is a unified interface where programmers can specify kernels, similar to CUDA [17] and OpenCL [18] that is executed on the NIC. The main difference between sPIN and CUDA or OpenCL that kernels do not offload compute-heavy tasks but data-movement tasks, specifically tasks that can be performed on incoming messages and only require limited local state. Such tasks include starting communications with NIC-based collectives, advanced data steering with MPI datatypes, data processing such as network raid, compression, and database filters. Similarly to OpenCL, sPIN’s interface is device and vendor-independent and can be implemented on a wide variety of systems.

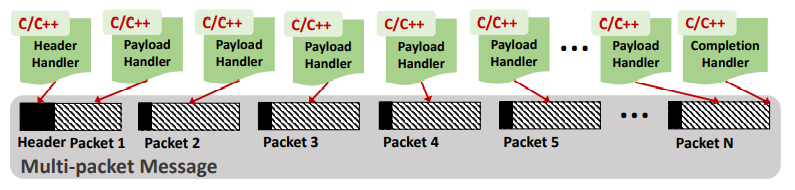
sPIN is enabled on existing NIC microarchitectures with very small but fast memories without obstructing line-rate packet processing. For this to be done, they designed sPIN around networking concepts (e.g., packetization, buffering, and packet steering). Packetization is the most important concept in sPIN because unlike other networking layers, packetization is exposed to the programmer. Programmers can define header, payload, and completion handlers (kernels) that are executed in a streaming way by handler processing units (HPUs) for the respective packets of each matching message. Handlers can access packets in fast local memory and they can communicate through shared memory. sPIN offers protection and isolation for user-level applications and can thus be implemented in any environment. Figure 3 shows sPIN’s architecture.



**Figure 3: sPIN Architecture [2]**

sPIN’s philosophy is to expose the highly specialized packet processors in modern NICs to process short user defined functions, this may seem like active messages (AM) [19] , they certainly both share some potential use case, but it is very different because it specifies an architecture for fast packet processing. The major difference is that AMs are invoked on full messages while sPIN is defined in a streaming manner on a per-packet basis. Also, sPIN enables to pipeline packet processing similarly to wormhole routing while AM would correspond to store and forward routing. Furthermore, AMs use host memory for buffering messages while sPIN stores packets in fast buffer memory in the NIC close to the processing unit for fastest access, it also enables accessing the host memory, but it minimized it as much as possible. sPIN handlers will be invoked on parts of the message while AM handlers is invoked after the message is delivered. This will make sPIN closer to packet processing system than AM.

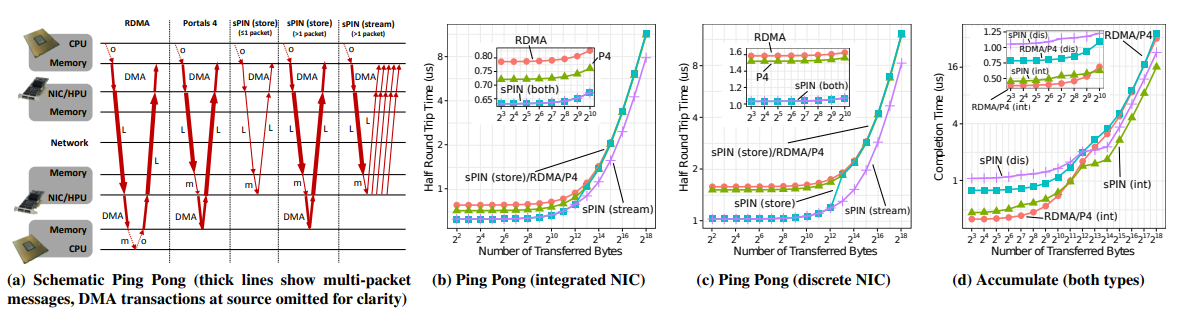
sPIN is relying on the fact that network devices split messages into packets, packets are easier to manage because it can be forwarded and buffered independently. In sPIN, the programmer is the one who define the handler function to be executed on a set of packets that form a message, this function is executed on one or multiple handler processing units (HPUs), each one of this HPUs owns shared memory that is persistent across the lifetime of a message, so handlers can use this memory to communicate easily.



**Figure 4: sPIN Message Handler Overview**

As shown in Figure 4, all necessary information to identify a message and steer it are in the first packet that it is called *header packet,* that is where the header handler is invoked. Then after the header handler completes its work, the payload handler will be invoked to process the message payload, and right after all instances of payload handler completed a completion handler will be invoked. The good thing about this model is that there is no requirement that packets arrive in a certain order.

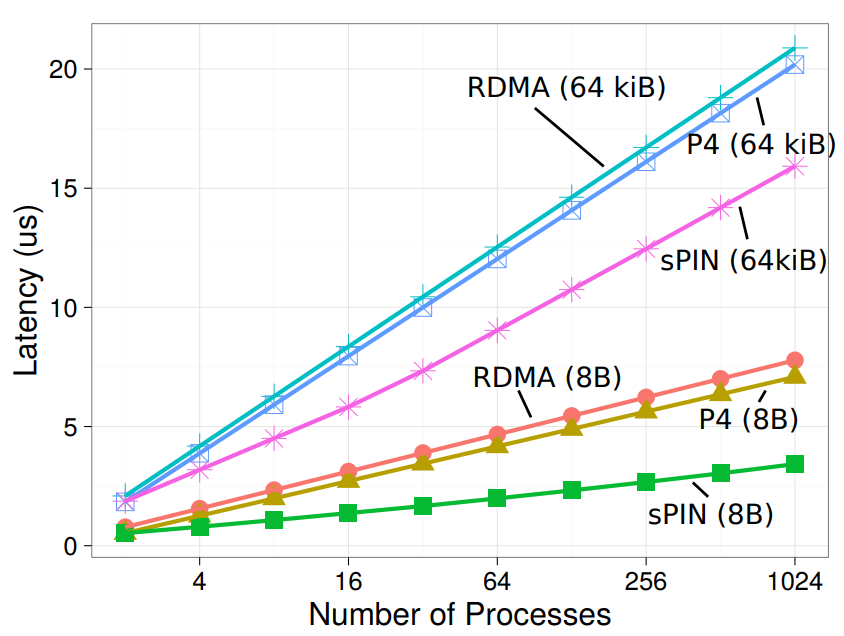
A good thing about sPIN is that it can be added to any RDMA network so demonstrating all its features was not a problem, but they used Portals 4 network interface because it offers advanced receiver-side steering, NIC resource management and many other things. To evaluate it in the best way they combined two open source simulators that have been vetted extensively by the research community, the two are 1) LogGOPSim [19] to simulate the network of parallel distributed memory applications and 2) gem5 [20] to simulate various CPU and HPU configurations.

For the benchmarking, they compared sPIN with standard RMDA and Portals 4 with simple ping pong benchmark, the reason for doing that is to illustrate the basic capabilities of processing messages on the NIC, Figure 5 a illustrates the time spent on the CPU, the host memory, the NIC, and its memory when the ping-point is executed. For RDMA, the pong is sent by the CPU, so the completion will only appear after the whole message has been placed in the host memory. For Portals 4, the pong message is pre-set up by the destination CPU and the reply is automatically triggered after the incoming message has been deposited into the host memory, but still the data is fetched via DMA from the CPU’s main memory. But in sPIN’s ping-pong, there are more than one option for generating the pong message: 1) (store) single pong message issued with a put from device, 2) (store) more than one packet message issued with put from host using the completion handler, 3) (stream) the payload handler could generate a pong put from device for each incoming packet.

**Figure 5: Ping pong and remote accumulate comparing RDMA, Portals 4, and various sPIN implementations**

The performance of ping pong for all configurations is shown for integrated sPIN implementations in Figure 5 b and for discrete implementations in Figure 5 c, and the accumulate result can be seen in Figure 5 d.

As another benchmark result in Figure 6, a broadcast operation has been done using a small message size (8 B) and large message size (64 KB) for varying number of processors. The integrated NIC has slightly lower differences but sPIN is still 7% and 5% faster than RDMA and Portals 4 at 1,024 processes, respectively.



**Figure 6: Broadcast on binomial tree**

## INCA

In-Network Compute Assistance (INCA) is introduced by Sandia International Laboratories and Emory University in 2019. It is introduced to enhance the previous solutions as all of the proposed solutions for in-network data processing process the data while it is moving in the network whether in the switches or in the endpoint’s NIC. A disadvantage for this technique is that a computer resources must be available when data arrives, so these approaches provide deadline-based models of execution. As an example, for a data rate of 200Gb/s, 64B packets, and 32 2.5GHz cores with 1 IPC, a stream-based approach is limited to packet-processing kernels of less than 500 instructions. We can increase this instruction limit by adding more cores, or by introducing additional message processing latency through buffering, or we can increase the packet size (for example), but doing so will solve part of the problem but it does not remove the deadline.

INCA proposed as a complementary and alternative approach to enable general purpose on-NIC computing capabilities, it is built on top of a computational model that directly leverages the atomic and triggered operation capabilities. Also, the model is deadline-free in contrast to stream-based approach and can process data even when the network is idle, so under INCA, the NIC will be somewhat viewed as a co-processor because of the computing assistance they can provide to host processors.

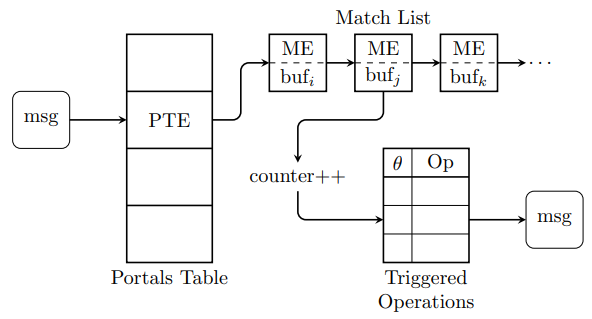
The packet processing engines (PPE) shown above (sPIN and SHArP) share a common strategy of processing the data, when there is no data flowing into the NIC, there will be no processing cause the network in this situation will be idle, but when data is offloaded to the network a computational resource must be available, working under that will create a deadline determined by the network speed, buffers, number of compute elements, etc. Failing to meet the deadline means the task has failed and may invalidate the data stream where the failure occurred, resulting in an unrecoverable, undefined data state, retransmission of data, or even termination of the application.

To avoid PPE deadline problems, the writers emphasized three such specialized offload compacities: 1) some systems provide the capacity to perform atomic operations (like summation for example) on a single operand to provide building blocks for offloading reductions. 2) providing hardware support to assist message matching process, and 3) hardware supporting triggered operations (like outgoing messages that are generated automatically in response to events caused by incoming traffic “like buffer updates”) provides primitives for constructing offloaded collectives and (potentially) rendezvous messaging [22, 23, 34].

To help INCA doing this, the writers presented triggered operation machine (TOM), which is a novel computational model that can do and handle a lot of operations in the NIC. TOM serves two purposes: 1) it demonstrates that when it is properly organized, these primitives are indeed capable of general purpose computing (are Turing complete), and 2) by specifying the basic capabilities required of any practical realization.

To proof their idea, they implemented the TOM model on top of the Portals network programming interface [4]. Portals is designed for implementation in hardware and includes all the core building blocks of TOM (atomic operations, message matching semantics, and triggered operations). However, several modifications are necessary to fully enable general-purpose computation under the TOM model.

Fully implementing TOM on top of Portals requires three modifications to the Portals specification. 1) a Portals triggered operation is discharged when its threshold is ≤ the counter value; it is modified to allow ‘strict’ indexing as required by the TOM model. 2) while match elements (MEs) can be used multiple times, triggered operations are by default ‘use once’. 3) while Portals provides an atomic for testing equality, it does not support modifying a counter value conditional on the outcome of such a test. To address this, they introduce a new atomic operation, (branch if ≤ zero). An ME specifying this atomic also provides an instruction index. If the value contained in the operand identified by the ME is zero or less, the counter is updated to this instruction index; otherwise, it is incremented by one.



**Figure 6: Portals message processing.**

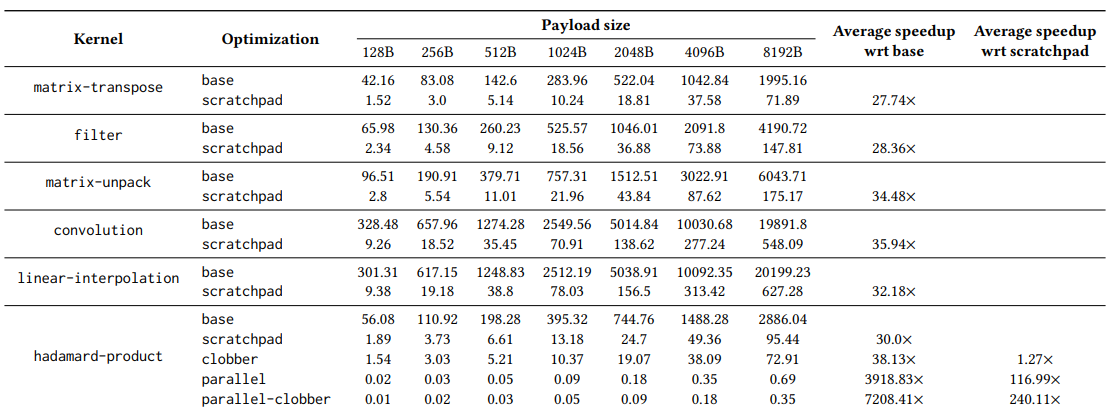
To evaluate this model, they determined instruction counts for various INCA kernels, and then based on these instruction counts, project runtimes for various INCA kernels under various configurations. Then using the LogP performance model, they first calculated INCA runtimes for the base and the scratchpad configurations and after that they calculated runtimes for a series of optimizations. These optimizations are presented incrementally so that benefits of each can be assessed independently, e.g., as regards potential implementation costs. Some of the results for the proposed configurations and optimizations are collected in Table 5.

As we can see in Table 5, INCA can provide up to a 11% speedup for applications with minimal code modifications and between 25% to 37% when applications are optimized for INCA (you can find the complete table of the LogP results in the paper [3]).

# Conclusion

Before you begin to format your paper, first write and save the content as a separate text file. Keep your text and graphic Reassign number of columns: Place your cursor to the

# References

1. Richard L. Graham, Devendar Bureddy, Devendar Bureddy, Hal Rosenstock et al., “Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction”, COM-HPC 2016.
2. Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ron Brightwell and Ryan E. Grant, “sPIN: High-performance streaming Processing In the Network”, SC 2017.
3. Whit Schonbein, Ryan E. Grant, Matthew G.F. Dosanjh and Dorian Arnold, “INCA: In-Network Compute Assistance”, SC 2019.
4. Shawn Hansen and Sujal Das, “Fabric-agnostic RDMA with OpenFabrics Enterprise Distribution: Promises, Challenges, and Future Direction”. SC 2006.
5. Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard, “Cray Cascade: A scalable HPC system based on a Dragonfly network”. In Proc. of the International Conference for High Performance Computing, Networking, 2012.
6. Brian W Barrett, Ronald Brightwell, Ryan E. Grant, Scott Hemmert, Kevin T Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B. Maccabe, and Trammell Hudson. “The Portals 4.1 network programming interface”. Technical Report, 2017, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States).
7. Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurth, “High Performance Packet Processing with FlexNIC”, 2016.
8. Ethernet Alliance. 2015. 2015 Ethernet Roadmap. (2015).
9. Daniel Molka, Daniel Hackenberg, and Robert Schöne, “Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer”, MSPC 2014.
10. Intel Corporation. 2016. Intel 64 and IA-32 Architectures Optimization Reference Manual. (July 2016).
11. Mellanox In-Network Computing For AI Presentation in May, 2017.
12. Message Passing Interface Forum 1994. MPI: A Message-Passing Interface Standard.
13. <http://www.openshmem.org>.
14. <https://www.mellanox.com>
15. <http://mvapich.cse.ohio-state.edu/benchmarks/>.
16. H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, “A tensorial approach to computational continuum mechanics using object-oriented techniques.”, 1998.
17. John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron, “Scalable Parallel Programming with CUDA”, 2008.
18. John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”, 2010.
19. Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser, “Active Messages: A Mechanism for Integrated Communication and Computation”, 1992.
20. T. Hoefler, T. Schneider, and A. Lumsdaine, “LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model”, 2010.
21. Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood, “The Gem5 Simulator”, 2011.
22. Brian W Barrett, Ron Brightwell, K Scott Hemmert, Kyle B Wheeler, and Keith D Underwood, “Using triggered operations to offload rendezvous messages”, 2011.

**Table 3: Kernel runtimes in µs. See main text for a description of the optimizations.**

1. Timo Schneider, Torsten Hoefler, Ryan E Grant, Brian W Barrett, and Ron Brightwell, “Protocols for fully offloaded collective operations on accelerated network adapters”, 2013.
2. K. D. Underwood, J. Coffman, R. Larsen, K. S. Hemmert, B. W. Barrett, R. Brightwell, and M. Levenhagen, “Enabling Flexible Collective Communication Offload with Triggered Operations”, 2011
3. R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press