Evaluating the Performance of One-sided Communication on CPUs and GPUs

Nan Ding, Muhammad Haseeb, Taylor Groves, Samuel Williams
Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
{nanding, mhaseeb, tgroves, swwilliams}@lbl.gov

***Abstract*—Effective programming models offer programmers the ability to harness the capabilities of the underlying platform. For decades, the two-sided Message Passing Interface (MPI) has become a de facto standard for communication among processes running on distributed memory systems. As high-performance GPU computing becomes the trend, GPU-initiated one-sided communication becomes a viable solution for multi-GPU scaling. It also highlights the use of one-sided communication on CPUs. However, the lack of deep understanding of one-sided communication performance and its impact on an application's performance becomes a hurdle. In this paper, we overcome this hurdle by proposing a Message Roofline model, which characterizes an application's sustained messaging performance (GB/s) as a function of its message size, number of messages per synchronization, peak network bandwidth, and network latency. We use three benchmarks to demonstrate the potentials of one-sided communication on CPUs and GPUs. These benchmarks include Stencils representing applications that follow the bulk synchronization programming model, Sparse Triangular Solve representing directed acyclic graph computations that perform asynchronous point-to-point communications, and Distributed HashTable performing atomic compare and swaps. Our evaluation provides insights into practically understanding the two-sided and one-sided communications in MPI applications, and can also guide hardware vendors with design principles lest the potential performance of one-sided communications being underutilized.**

***Index Terms*—two-sided communication, one-sided communication, distributed memory, multi-GPU communication**

I. Introduction

Over the last decade, accelerated computing architectures have become increasingly popular in High-Performance Computing (HPC) systems. A total of 289 systems on the 2023 TOP500 list use accelerators [1]. GPUs, the primary accelerator in modern HPC systems, contribute 80% of the compute capability of the entire machine [2]. This makes application developers spend tremendous efforts towards designing and implementing new algorithms to fully leverage GPU capabilities. The most common way of communicating on multiple GPU systems is to communicate via the host processor. This forces the developers to think differently from programming on distributed memory CPU systems, as they must think about host-initiated communications separately from GPU computations, instead of performing both computations and communications directly within the GPU kernels. This often results in increased algorithm complexity and decreased program productivity.

Effective programming models offer programmers the ability to utilize the capabilities of the underlying hardware. The two-sided Message Passing Interface (MPI) has become a de facto standard for communication among processes running on distributed memory systems. As high-performance GPU computing becomes ubiquitous, some numerical methods find host-initiated two-sided MPI and its CUDA-aware variant [3] to satisfy their requirements. It is because they have a relatively simple communication pattern, such as stencils, which adhere to the Bulk Synchronous Parallel (BSP) model [4]. Inter-processor communications follow the discipline of strict barrier synchronization. However, directed acyclic graph (DAG) computations, such as sparse triangular solve (SpTS), are hard to scale on multi-GPU platforms due to more complex communication patterns. Point-to-point communications can happen anytime between two processes with no strict barrier synchronization.

Fortunately, one-sided communication offers a friendly programming model for DAG-like computations. One-sided communication libraries, such as OpenSHMEM [5], NVSHMEM [6] and ROC_SHMEM [7] use the Partitioned Global Address Space (PGAS) [8] style programming. Senders can directly access the receivers' memory without requiring explicit communication calls from them. NVSHMEM and ROC_SHMEM further support initiating communication from within a GPU kernel. As such, one-sided communication provides a different execution paradigm from the standard host-initiated two-sided model, offering an attractive alternative for DAG applications. Several researchers have investigated one-sided communications performance on GPUs [9–14] using NVSHMEM and ROC_SHMEM. They found that the GPU-initiated one-sided communication makes scaling DAG-like computations more feasible. In addition, it provides better scaling performance for BSP-like applications compared to using host-initiated two-sided communications due to the lower network latency. Thus, one-sided communication is a potential and promising programming model for scalable GPU computing. The CPU one-sided communication performance has been explored in work [15–17] using an improved fast one-sided library that works on top of the standard one-sided MPI. Missing from past work is a lack of comparison between two-sided and one-sided communications in terms of performance, productivity, and portability.

In this paper, we propose a Message Roofline Model to provide a tighter upper bound and intuition on achieved communication bandwidth using the number of messages per synchronization. We evaluate the communication performance on CPUs and GPUs over various interconnects and topologies,

including standard two-sided and one-sided MPI on CPUs over Infiniband and Slingshot-11, NVSHMEM over NVLINK2 using a dual-island dumbbell shape topology and NVLINK3 fully connected on NVIDIA GPUs. We use three workloads, Stencil, Sparse Triangular Solve (SpTRSV), and Distributed HashTable, to understand and visualize the performance impact of one-sided and two-sided communications. Based on these results, we summarize our observations, challenges to address, and potential research topics regarding one-sided communications. Through this evaluation, application developers can gain deeper knowledge about multi-GPU communication, paving the way for building a more mature multi-GPU programming environment and code portability from two-sided to one-sided communications. We make the following contributions in this paper:

1) We create the Message Roofline Model, which extends the traditional bandwidth and message size plots to understand an application's sustained messaging performance (GB/s) as a function of its number of messages per synchronization, message size, empirical peak network bandwidth, and empirical network latency.
2) We create a realistic bound on the communication performance based on the number of messages per synchronization, instead of using a loose (practically unattainable) upper bound from the traditional flood send/put benchmark.
3) We observe that the CPU one-sided communications can potentially outperform the two-sided communication by introducing put-with-signal and receiver notification operations.
4) We observe that on GPUs, a message size larger than 131KB can achieve up 2.9× speedup by splitting into multiple smaller messages and simultaneously communicating them.
5) We provide helpful insights and guidance to communication software providers to better develop technologies for various architectures and applications.

II. Message Roofline Model

The Message Roofline Model follows the methodology of the traditional Roofline Model [18] but applies to bandwidth vs. message size plots adding ceilings and concurrency axes. We introduce and incorporate a new metric, the number of messages per synchronization (msg/sync), to better quantify the impact of different (one- and two-sided) communication libraries on an individual application. As such, the Message Roofline model provides a generalized framework to visualize and evaluate an application's sustained messaging performance (GB/s) as a function of its message concurrency, message size, the peak network bandwidth, and the network latency. It provides a quick visual comparison of an application's communication performance against the bounds set by the underlying hardware and software bounds.

Fig.1 shows an overview of the Message Roofline model on Frontier supercomputer. Frontier's detailed architecture can be found later in this section. The on-node communication data path on Frontier CPU partition is Infinity Fabric CPU-GPU (36GB/s) → PCIe4 ESM (50GB/s). Thus the ultimate bound is the Infinity Fabric CPU-GPU, at 36GB/s [19].

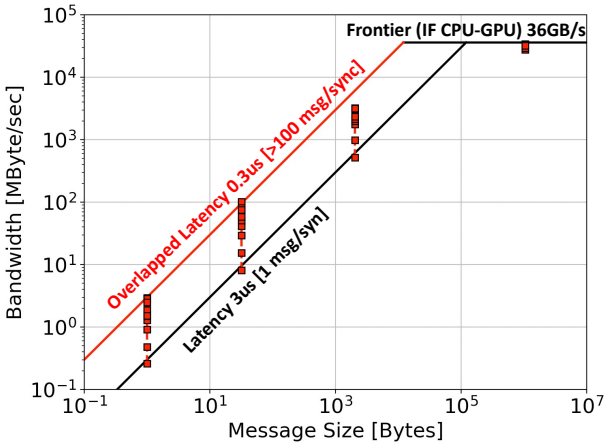


Fig. 1: An overview of the Message Roofline Model on Frontier.

The diagonal ceilings (Latencies) are fitted from the empirical data. Each vertically dotted line represents a certain message size. Along the dotted line, as it moves from bottom to top, it sends more and more messages per synchronization, i.e., ranges from one message per synchronization to one million messages per synchronization. One can immediately observe the implications of overlapped latency by sending more messages per synchronization. According to the LogGP model [20], the parameters that denote the communication cost are:

- **L**: network latency cost, processor independent (PI)
- **o**: sender/receiver sequential overhead, processor dependent
- **g**: 1/bandwidth (time between sends or receives) (PI)
- **G**: message size/bandwidth (time per byte) (PI)
- **P**: number of processors

Thus, the processor-independent times, **L**, **g**, and **G**, can be overlapped with computation. Meanwhile, the **L** and **G** can further be overlapped by sending more messages. On the other hand, the g can not be overlapped with more messages. This is because if a processor wants to send more than one message in a row, it has to wait for g cycles after the first message goes out before it can push the first byte of the second message into the network.

The plots in Fig. 1 show a sharp Message Roofline model, which is calculated by $\frac{B}{\max(o,L,B\cdot G)}$, where **B** is the number of bytes. The junction between the diagonal and horizontal ceiling is an ideal region one can never practically reach. In reality, the achieved bandwidth is $\frac{B}{o+\max(L,B\cdot G)}$, which denoted a rounded Message Roofline model as the empirical dots show. The latter one infers that the overhead o dominates the total messaging time.

Eventually, the Message Roofline model tells the potential of overlapping messages compared to serialized messages. Fig.1 suggests that at maximum, you can get 10× improvement by sending one hundred (or more) messages per sync

TABLE I: Evaluation Platforms

Machine	GPUs per node	GPU Interconnect	GPU Runtime	GPU-CPU Interconnect	CPUs	CPU-CPU Interconnect	CPU Runtime	CPU-NIC Interconnect
Summit	6×V100	NVLINK2	CUDA v11.0.3 NVSHMEM v2.8.0	NVLINK2	2×IBM POWER9	X-Bus	IBM Spectrum	PCIe4.0
Perlmutter GPU	4×A100	NVLINK3	cudatoolkit v11.7 NVSHMEM v2.8.0	PCIe4	1×AMD EPYC 7763	-	-	PCIe4.0
Perlmutter CPU	-	-	-	-	2×AMD EPYC 7763	Infinity Fabric	CrayMPI	PCIe4.0
Frontier CPU	-	-	-	-	1×AMD EPYC 7A53	Infinity Fabric	CrayMPI	Infinity Fabric and PCIe4.0 ESM

when $L >> G$. However, when G dominates, the benefit you can get from message overlapping is limited since the bandwidth bounds are already reached.

Table I lists the machine details evaluated in this paper. Fig. 2 plots the node architecture of the three machines.

Perlmutter CPUs: Fig. 2a shows the Perlmutter CPU node configuration [21]. Each CPU node has two Milan CPUs connected by Infinity Fabric (IF) CPU-CPU at 4× 32 GB/s/direction. The NIC is connected to one of the CPUs via PCIe4.0 at 25GB/s/direction.

Frontier CPUs: Fig. 2b describe the Frontier node configuration [19]. Each node has one Milan CPU and four MI250X GPUs via Infinity Fabric CPU-GPU at 36GB/s/direction. The four NICs are connected to the four GPUs via PCIe4 ESM at 50GB/s/direction.

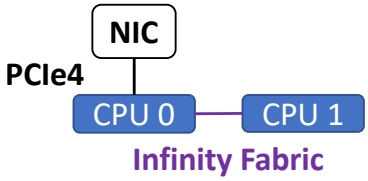
Summit CPUs: it has two Power9 CPU sockets [22] as Fig. 2c shows. The two CPUs are connected via X-Bus at 64GB/s/direction. Each CPU is connected to the NIC via PCIe at 16 GB/s/direction.

Summit GPUs: the six V100 GPUs on one node are organized in a dual-island dumbbell shape topology [22] as showed in Fig. 2c, each with three GPUs and one Power9 CPU socket. The three GPUs in each subset are fully connected using NVLINK2 at 50 GB/s/direction.

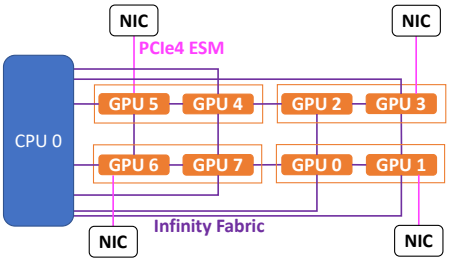
Perlmutter GPUs: Fig. 2d presents the Perlmutter GPU node configuration [23]. The four A100 GPUs on one node are fully connected using NVLINK3 at 300GB/s/direction. Since the twelve NVLink ports are divided into three groups for connection to the other GPUs in the node, the peak between two GPUs is 100GB/s/direction. All GPUs are connected to the Milan CPU via PCIe4.0.

Note, the Frontier GPU partition is not considered in this paper due to the lack of support of `wait_until_any` in ROC_SHMEM [7]. Additionally, the backend implementation is not in the scope of this study. We demonstrate the achieved performance of using the existing one-sided and two-sided communication libraries. However, based on the presented performance data, we can motivate their backend optimization.

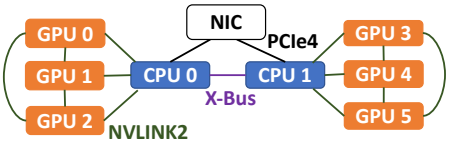
Fig. 3 plots the sustained network bandwidth using two-sided and one-sided MPI on Perlmutter, Frontier, and Summit CPUs. On Perlmutter CPUs, the on-node communication transfers the message via IF. As such, the achieved bandwidth in Fig. 3a is close to the IF peak of 32GB/s. The Frointer CPU cores also communicate via IF, which provides 36GB/s, as Fig. 3b shows. Even though the X_Bus provides 64GB/s/direction between the two Summit CPUs within one node, the achieved bandwidth is relatively low, approximately 25GB/s in Fig 3c.



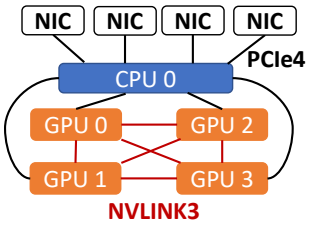
(a) Perlmutter CPU: NIC attached to CPU



(b) Frontier: NIC attached to GPU



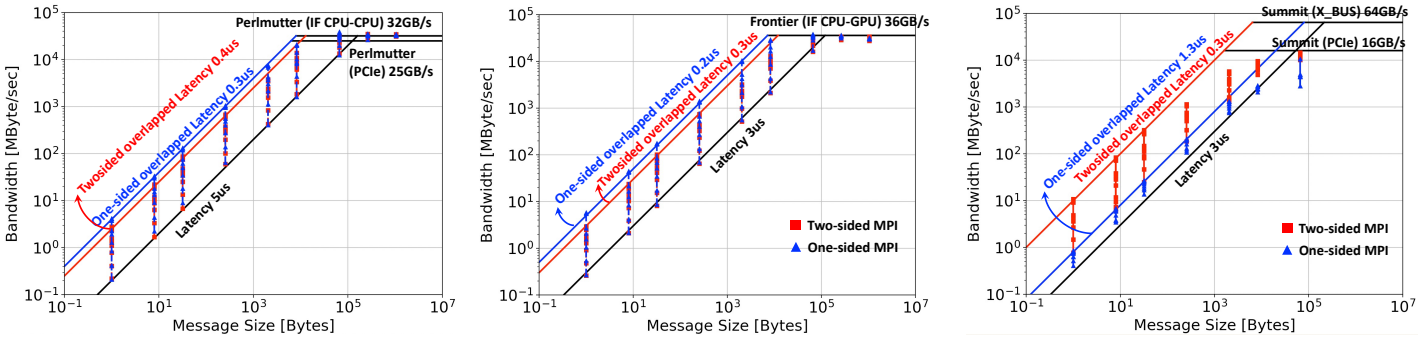
(c) Summit: dual-island dumbbell shape topology



(d) Perlmutter GPU: fully connected

Fig. 2: Node architecture of Perlmutter CPU, Frontier CPU, Summit and Perlmutter GPU.

As the number of messages per synchronization increases, one-sided MPI achieves higher bandwidth and lower latency than the two-sided MPI, as depicted by Fig. 3. Note, each message requires four one-sided MPI operations to complete but only two two-sided MPI operations. However, even with double the number of operations, one-sided can still achieve a comparable bandwidth as the two-sided. As such, one-sided has the potential to provide lower latency and higher bandwidth with the support of *put-with-signal*, which can reduce the number of one-sided MPI operations per message



(a) Perlmutter CPU (b) Frontier CPU (c) Summit CPU

Fig. 3: One-sided MPI (four MPI operations per message) has the potential to outperform the two-sided (two MPI operations) on Perlmutter and Frontier. The bandwidth ceiling (horizontal) is theoretical, and the diagonal ceilings (latency lines) are inferred based the empirical data.

to two. Fig. 3c shows a different behavior for Spectrum MPI communication performance on Summit. Here, we observe that the one-sided Spectrum MPI communication performance is consistently lower than the two-sided Spectrum MPI.

Fig. 4 plots the sustained NVSHMEM GPU-initiated put-with-signal bandwidth on Perlmutter and Summit GPUs. The GPU partition node architectures are presented in Fig. 2d and Fig. 2c. Perlmutter GPUs transfer messages via NVLINK3 at 100 GB/s/direction, while Summit GPUs transfer messages via NVINK2 at 50 GB/s/direction or 32 GB/s/direction (inter CPU sockets). Figures 4a and 4b show that similar to CPU-initiated communications, the achieved bandwidth for GPU-initiated NVSHMEM communications increases with the number of messages per synchronization. The latency for Perlmutter GPUs was observed from 4us to 0.5us, which is similar to the latency of 5us to 0.3us on Perlmutter CPUs. However, the observed bandwidth for Perlmutter GPU was much higher than that of Perlmutter CPU. As such, using GPU-initiated communications can improve the programming productivity as one can do everything inside GPU kernels, similar to programming distributed memory CPU codes.

III. Results

We consider three workloads: Stencil, Sparse Triangular Solve (SpTRSV), and HashTable (HB), in this paper. These three workloads represent three different communication patterns, as listed in Table II:

1) Stencil: it adheres to the BSP model [4]. Communications follow the discipline of strict barrier synchronization. The sender and receiver pairs and message sizes are all fixed.
2) SpTRSV: it's a DAG-like computation. Point-to-point communications can happen anytime between any two processes, depending on the sparsity pattern and the process decomposition, with no strict barrier synchronization. Both message size and sender-receiver pair could be different among messages.
3) HashTable: it represents data analytics applications that often require random access in distributed structures. Unlike SpTRSV and Stencil, it's a true sender's control communication paradigm in which senders perform inserts atomically on the remote memory. The message size is fixed, but the sender-receiver pair varies.

We then detail our one-sided and two-sided designs and achieved performances of each workload in the rest of this section.

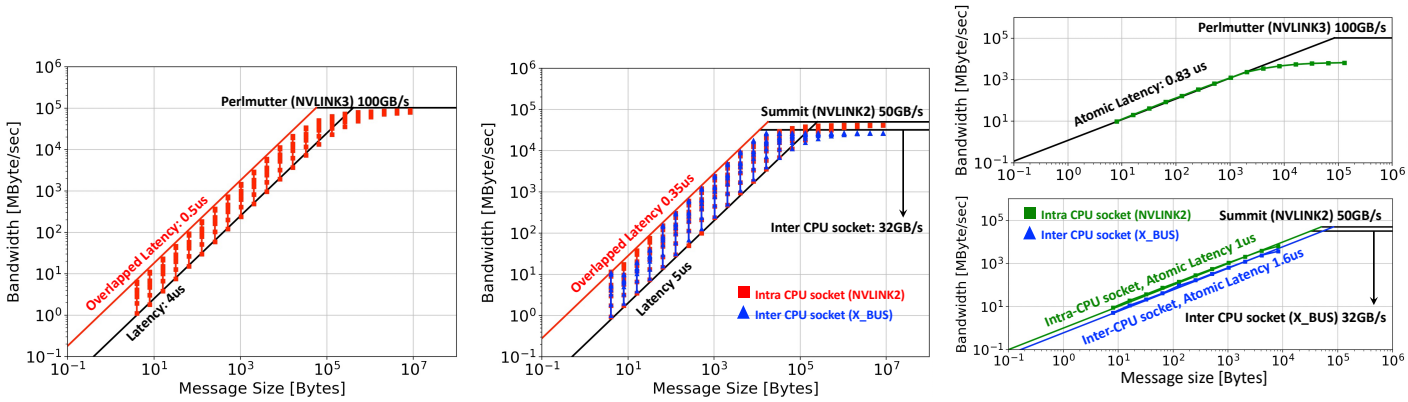
A. Stencil

In two-sided implementations of two-dimensional stencils, each process performs four sets of `MPI_Isend` and `MPI_Irecv` to send and receive from its neighbors, and then performs `MPI_Waitall` for synchronization. After `MPI_Waitall`, the received data can be used for local computation.

In one-sided implementations of two-dimensional stencils, each process performs four `MPI_Put` within a pair of `MPI_Win_fence`. The `MPI_Put` write the data directly to its neighbors. The `MPI_Win_fence` avoid the data hazard and the data can be used for local computations as it returns.

The GPU design follows the idea of put-with-signal. We use NVSHMEM for NVIDIA GPU clusters. The senders sends data using `nvshmem_double_put_signal_nbi`. The NVSHMEM library handles the memory order of the data and signal. The receiver uses `nvshmem_uint64_wait_until_all` to wait for the signal. Once it returns, the data can be used for local computation on the receiver side.

Porting the stencil code among the three implementations is relatively straightforward due to the simplicity of the BSP model. The two-sided CPU, one-sided CPU, and one-sided GPU share the same communication code design. Specifically, the one-sided and two-sided implementations have the same message concurrency (number of neighbors) and message size (halo size).



(a) Perlmutter GPU Put (b) Summit GPU Put (c) Summit GPU Atomic

Fig. 4: NVSHMEM GPU-initiated put and atomic compare and swap on Perlmutter and Summit. The plotted dots are measured. The bandwidth ceiling (horizontal) is theoretical, and the diagonal ceilings (latency lines) are inferred based the empirical data.

TABLE II: Evaluated workloads characterizations. P=number of processes.

Workloads	Patterns	Notify Receiver	Operation	P2P pair	#Msg/sync	Words/Msg
Stencil	BSP sync	Yes	two-sided: non-blocking send and receive with waitall	deterministic & fixed	4	problem size / P
			one-sided: non-blocking put with fence			
SpTRSV	DAG async	Yes	two-sided: non-blocking send and receive with waitall	deterministic & variable	1	avg. 100
			one-sided: nonblocking put with flush (data and signal, respectively) user implemented receiver notification			
Hashtable	Random async	No	two-sided: non-blocking send and blocking receive	indeterministic	P	3
			one-sided: atomic compare and swap		1e6	1

For the test case in this paper, both two-sided and one-sided send four messages per synchronization, as listed in Table II. The grid size is 16384×16384. Thus, the message size is a mixture of 2^{16} to 2^{13} as we increase the number of processes from 4 to 128 using a 2D process grid. Fig. 6a shows the sustained two-sided and one-sided MPI performance on Perlmutter CPUs. The one-sided has lower latency than the two-sided, and the two communication models begin to converge at a message size of 2^{16}. The observed two-sided and one-sided implementations perform equally in Fig. 5 because stencil computations are inherently bandwidth-bound. As such, even though the one-sided has 20% lower latency than two-sided, applications like stencil won't see any benefit from it on CPUs.

However, GPU-like architectures provide massive parallelism and higher bandwidth than multi-core CPUs, making them an ideal acceleration platform for such stencils as shown by Fig. 5. One can achieve 30GB/s bandwidth on Perlmutter GPUs while only getting 20GB/s on CPUs. Thus, the speedups on GPUs are mainly gained from higher achieved bandwidth and parallelism. One can scale to 16×8 MPI ranks on CPUs, and the work within each MPI rank is serial. However, each GPU can have eighty thread blocks scheduled simultaneously, and thus achieving 320 × parallelism on one node.

B. SpTRSV

SpTRSV has a more complex data dependency than stencils. Here we consider performing SpTRSV on the LU factored matrix via `SuperLU_DIST`. Let's consider a lower triangular matrix L to explain its communication pattern. A supernode is a set of consecutive columns of L with the triangular block just below the diagonal being full, and the same nonzero structure below the triangular block. After obtaining a supernode partition along the columns of L, it applies the same partition row-wise to obtain a 2D block partitioning. The nonzero block pattern defines the supernodal DAG. After partitioning the matrix L among multiple processes using a 2D block-cyclic layout, each process is responsible for a subset of solution subvectors. A DAG can precisely express this data dependency. The solution of those subvectors and partial summation results require communication. The message size equals the size of the subvectors or the partial summation results. Ultimately, the message size varies from a few bytes to hundreds of bytes depending on each supernode size.

In the two-sided SpTRSV, each process uses `MPI_Isend` to send messages and uses `MPI_Recv` in a loop to wait for the messages. The loop size equals the number of expected messages.

In the one-sided SpTRSV, we use four MPI operations to send the data and signal: `MPI_PUT(data)`, `MPI_Win_flush`, `MPI_PUT(signal)` and `MPI_Win_flush`. Among the four operations, `MPI_PUT` is non-blocking, and `MPI_Win_flush` ensures the signal arrives after the data. After sending the signal, we use a second `MPI_Win_flush` to avoid a delayed signal in the network. This is because the standard one-sided library lacks support for signal waiting and notification operations at

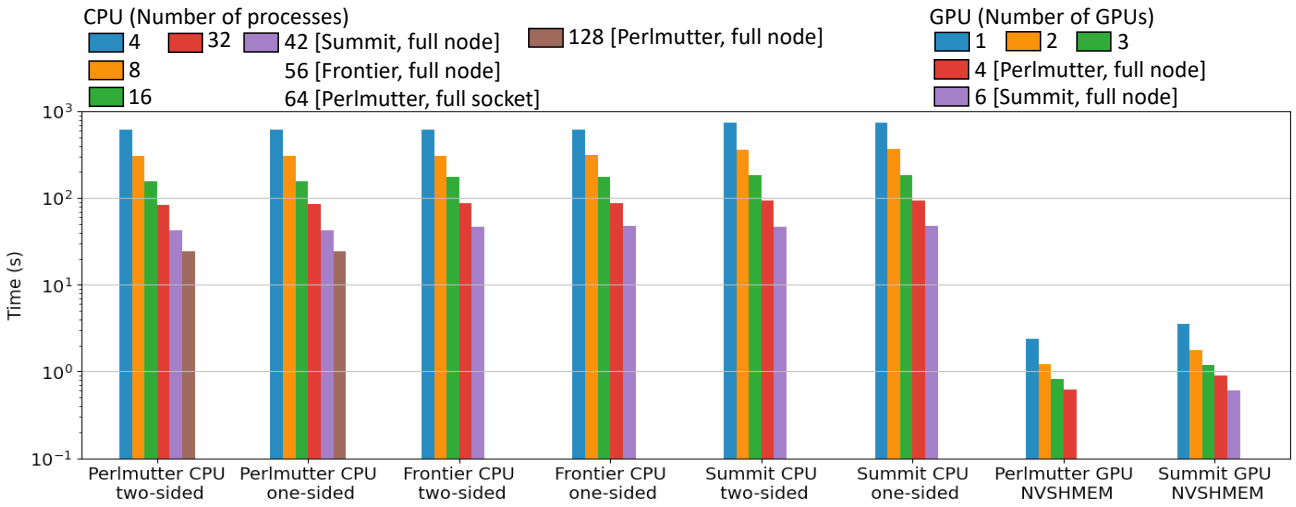
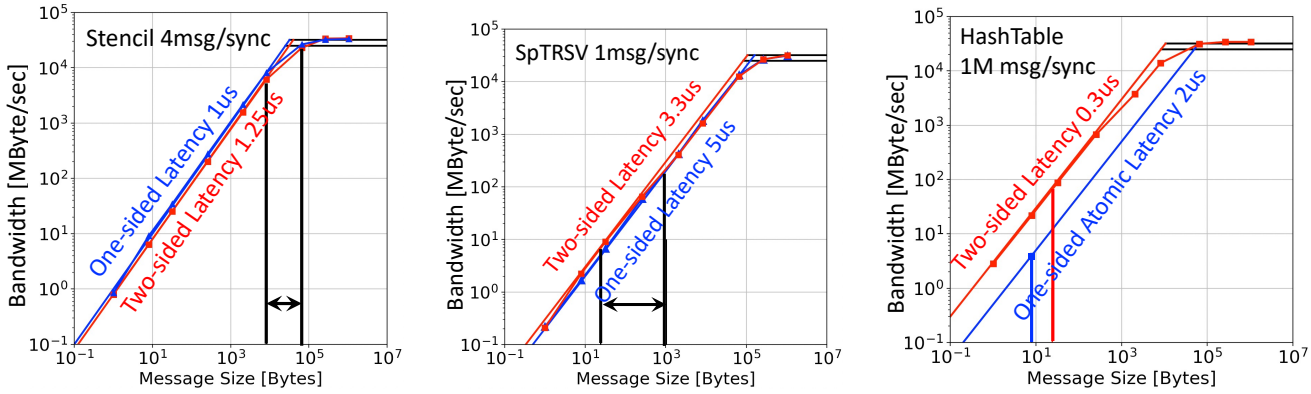


Fig. 5: Stencil time on CPUs and GPUs using two-sided and one-sided communications. Increased parallelism, as evidenced by GPUs, leads to larger improvements with the number of messages per synchronization.



(a) Stencil (b) SpTRSV. (c) Distributed HashTable

Fig. 6: Communication upper bound of Distributed HashTable, Stencil, and SpTRSV on Perlmutter CPUs. Vertical lines represent the tested message sizes. Stencil and SpTRSV have a wide range of message sizes because increasing parallelism can decrease the message size in Stencil, while SpTRSV inherently has various message sizes according to the LU factorization, and the message size will not change by parallelism. The Distributed Hashtable has a fixed message size across all parallelism.

the receiver side. We also implement our acknowledgment for receivers as described in Listing 1. Inspired by the `nvshmem_wait_until_any`, we pre-compute a mask array ($validindex[]$) for receivers. The length of this mask array equals the number of messages that needs to be received ($msg\_num\_expected$). The receiver keeps looping over the mask array until all messages are received. In each loop, the receiver masks out (set value to -1) the corresponding element if the signal arrives. Otherwise, the receiver continues the loop. After receiving the signal, the receiver can use the received data in its local computation.

```
int recv_count=0;
while (recv_count < msg_num_expected) {
    for (int i = 0; i < msg_num_expected; i++) {
        if (validindex[i]==-1) continue;
        if (signal[i] == 1) {
            validindex[i] = -1;
            recv_count++;
        }
        continue;
    }
}
```

Listing 1: Receiver Acknowledgment Example

The one-sided waiting message loop is similar to the two-sided design. However, since the standard one-sided library lacks support for signal waiting and notification operations on the receiver side, users must create the receiver acknowledgment by themselves.

The one-sided GPU implementation follows a simi-

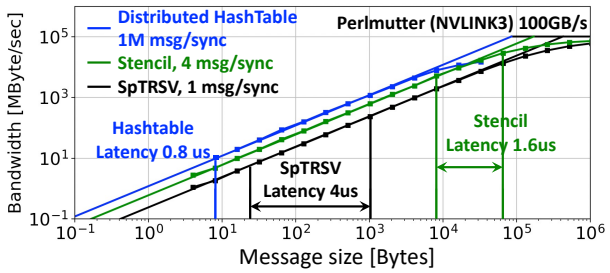


Fig. 7: More messages per synchronization help to overlap the latency. The GPU messaging latency of Hashtable (one million messages per synchronization) is the smallest, while the latency of SpTRSV is the largest due to one message per synchronization.

lar idea to CPU SpTRSV. Each process sends data using `nvshmem_double_put_signal_nbi`, and uses `nvshmem_wait_until_any` in a loop to wait for messages. The loop size equals the number of expected messages. Once the signal arrives, the sending data is already completed for local computation.

Both two-sided and one-sided SpTRSV send one message per synchronization. However, in addition to sending the data, one-sided also sends an extra signal. As such, the CPU one-sided communication needs four MPI operations per message and needs to implement the receiver notification. Therefore, two-sided performs one operation per synchronization (3.3us in Fig 6b) while one-sided performs four operations per synchronization (5us in Fig 6b).

Fig. 8 shows the SpTRSV time. The tested matrix has $1e+08$ non-zeros with a dimension of $126K \times 126K$. One could immediately observe that, unlike stencil, the one-sided SpTRSV is slower than the two-sided. Recall that SpTRSV has one message per synchronization and the message size ranges from 24 bytes to 1040 bytes. The low performance of the one-sided implementation is due to higher latencies introduced by $4\times$ as many MPI operations.

The second observation from Fig. 8 is that one-sided implementation stops scaling at higher parallelism. This is due to extra work to maintain data arrival in one-sided implementation. Recall that each process needs to loop all the remaining signal buffers to check the message arrival every time it expects one message.

The SpTRSV scales well on Perlmutter GPUs using NVSHMEM but not on Summit GPUs. One can immediately find the advantages of a faster GPU-to-GPU interconnection. Recall that the latency via NVLINK3 (4us) is 20% lower than NVLINK2 (5us), and bandwidth is $2\times$ higher than NVINK2. In addition, the observed single GPU times are equal on the two machines. As it scales to four GPUs, SpTRSV can achieve a $3.7\times$ speedup via NVLINK3 on Perlmutter compared to NVLINK2 on Summit. This emphasizes the benefits of a lower latency between GPUs for applications like SpTRSV.

In addition, the implication of high latency on Summit GPU can be immediately observed by comparing the Summit GPU and Summit CPU runs. First, the single GPU is the fastest due to the GPU power, and then the time increases as we scale to more GPUs due to the high latency on Summit GPUs (5us). SpTRSV can scale up to 32 Summit CPUs because Summit CPUs provide a lower latency of 3us, and the time increases using all 42 CPUs due to contention.

C. Distributed Hashtable

The distributed hashtable benchmark represents data analytics applications that often require random access in distributed structures. Each process manages a part of the hashtable and an additional overflow heap to store elements after collisions. The total number of messages each process expects is random. Only the sender knows the receiver when the inserted element is generated. Unlike stencil and SpTRSV, Hashtable is a sender's control benchmark. Thus, using one-sided communication is more straightforward than using two-sided ones.

The table and overflow list are placed in shared arrays in the one-sided implementation. Inserts are based on the atomic compare and swap (CAS, `MPI_Fetch_and_op`) operation. If a collision happens, the losing thread acquires a new element in the overflow list by atomically incrementing the next free pointer. It also updates the last pointer using a second CAS. `MPI_Win_flush_local` are used to ensure the memory consistency.

In the two-sided implementations, each process sends a triplet $(ID, elem, pos)$ to other processes using `MPI_Isend`, where ID is the receiver rank ID, $elem$ is the insert value, and pos represents the insert position. After sending, each process waits for $P-1$ messages using `MPI_Recv` with `MPI_ANY_SOURCE` and `MPI_ANY_TAG`. If the received ID equals my rank ID, the process starts to perform the local insert. Otherwise, the process continues waiting for the rest messages.

Eventually, the one-sided implementation performs the atomic operation, and there's no synchronization until ending the insert. The two-sided has P number of messages per synchronization (or insert), and the message size is also tripled. According to the Message Roofline Model in Fig. 6c, the two-sided MPI takes 0.3us per message when the number of messages per synchronization is a hundred. When using 128 processes, each process must send (or receive) 127 messages to other processes. Thus, for one insert, the two-sided message time should be $0.3 \times \log_2 128 = 2.1$ us, and also considering the extra overhead of the local insert, the GUPS [24] of the two-sided hashtable can never be higher than 476K GUPS (one CAS in 2.1us). On the other hand, the on-sided can provide 500K GUPS corresponding to one CAS in 2us. It suggests that the one-sided HashTable should be faster than the two-sided. This is corroborated by the results in Fig. 9 which shows that one-sided is $5\times$ faster than the two-sided using 128 processes. Note, the one-sided is slower than the two-sided in the case of 2 processes, as the two-sided message time for one insert should be 1.1×1=1.1 us as compared to 2 us for one-sided

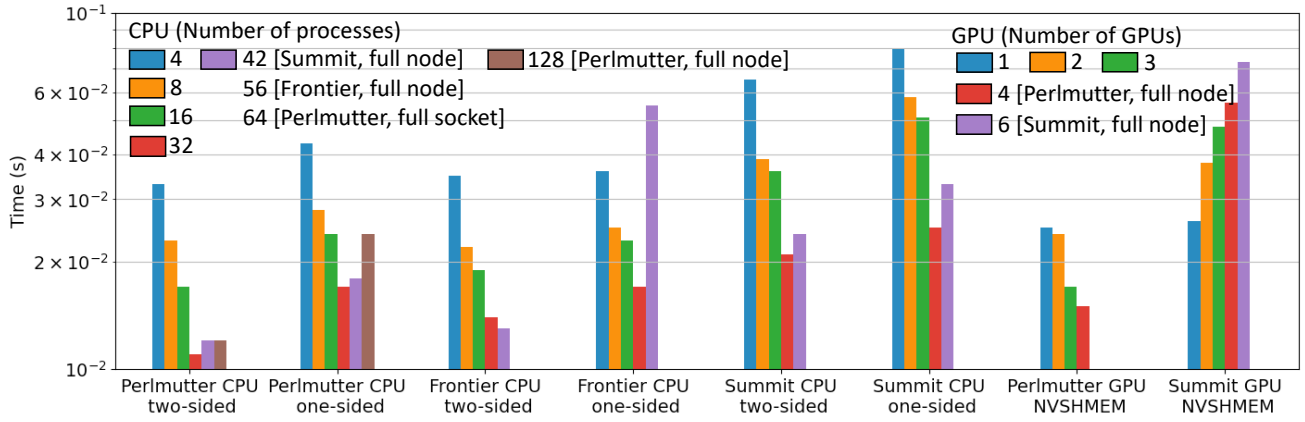


Fig. 8: SpTRSV time on CPUs and GPUs using two-sided and one-sided communications. The opposite performance scaling trends between Perlmutter GPUs and Summit GPUs indicates the network impact on scaling performance to multiple GPUs.

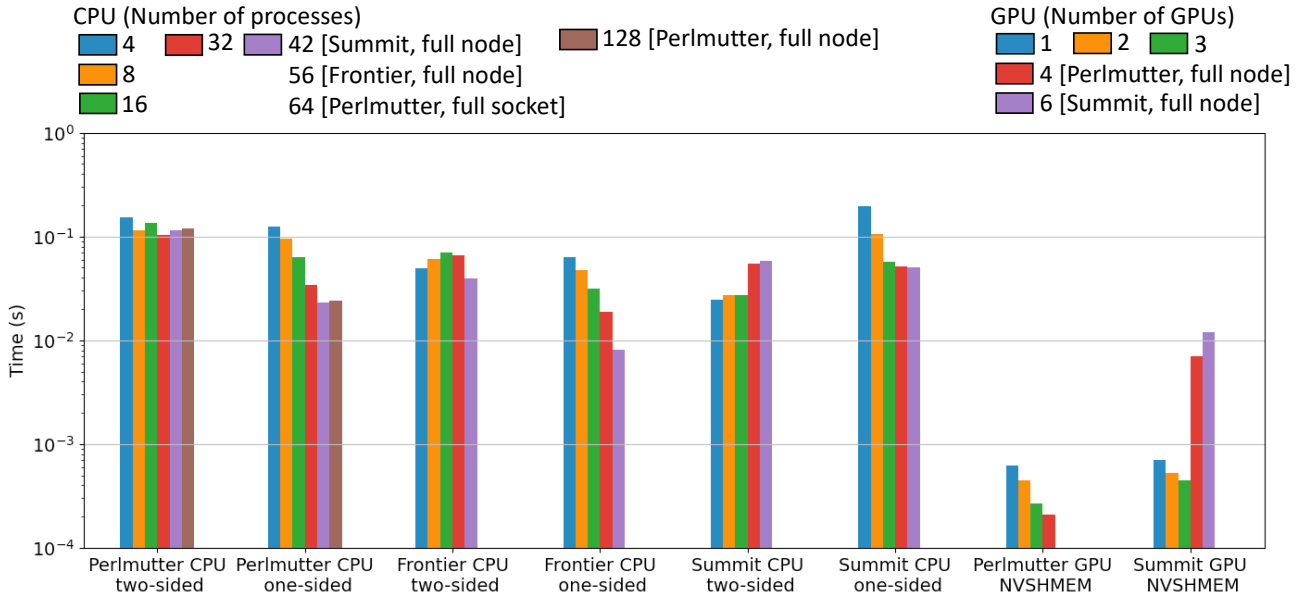


Fig. 9: Distributed hashtable time on CPUs and GPUs using two-sided and one-sided communications. The increased time using more than three GPUs on Summit shows the performance implication of on-node GPU topology.

CAS. These observations apply to Frontier and Summit CPUs as well.

Looking at the Summit GPU results in Fig. 9, one can immediately see the implication of on-node GPU topology. It stops scaling when using Summit GPUs across two CPU sockets because each atomic compare and swap takes 1.6us across sockets while it only takes 1us within a socket. If we further compare the Perlmutter GPU results to Summit GPU results within a CPU socket, we can find that the Perlmutter results are slightly better than Summit results. Again, it is because each atomic compare and swap takes 0.8us on Perlmutter GPUs, but it takes 1us on Summit GPUs within a CPU socket.

IV. Related Work

Li et al. [25] evaluated modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect NUMA effects in a multi-GPU node and provided optimization guidance. Groves et al. [26] studied the performance trade-offs between the host and device-initiated GPU communications. Spafford et al. [27] analyzed the NUMA effects in a multi-GPU node and provided optimization guidance. Sun et al. [28] evaluated the potential performance benefits and tradeoffs of AMD's Radeon Open Compute (ROC) platform for Heterogeneous System Architecture (HSA). All of the existing studies use the flood send (or put) or ping-pong to benchmark the communication performance. However, it provides a loose bound of the achieved performance because most practical applications can seldom reach the benchmarked performance. Dufek el al. [29] create an extended Roofline model which characterizes an application performance (GFLOP/s) as a function of flops per MPI byte. The proposed Message Roofline model, using the number of messages per synchronization, provides a tighter

bound of achieved performance, especially for latency-bound applications like SpTRSV and Distributed HashTable. It also characterizes the message overhead, which can never be overlapped. It also guides the software developers on how much optimization room they have by overlapping the messages and guides the MPI distribution developers about future software optimization.

V. Discussion and Conclusion

In this paper, we characterize and evaluate three types of communications namely CPU two-sided MPI, CPU one-sided MPI and GPU one-sided MPI, using three workloads over three architectures namely Perlmutter CPUs and GPUs, Summit CPUs and GPUs, and Frontier CPUs. We also present a Message Roofline model using the number of messages per synchronization to provide realistic performance bounds for different workloads. This study also helps motivate the HPC community to push forward the multi-GPU research, particularly the development of more mature multi-GPU programming, execution, and performance models.

Performance. We show that the applications similar to Stencils are not sensitive to on-node GPU topology as they scale well using six GPUs on Summit. These applications achieve better speedups from the massive parallelism and high bandwidth of GPUs as compared to the CPUS due to their inherent bandwidth-bound nature. However, the latency-bound applications, similar to SpTRSV, prefer a lower-latency interconnect. For example, running SpTRSV using 4 GPUs on Perlmutter is 3.7× faster than running it on Summit. Liu et al. [15] showed that a low latency one-sided foMPI [16] on CPU can improve SpTRSV performance by 1.5× compared to the standard two-sided MPI. Applications similar to the Distributed HashTable also show sensitivity to the on-node GPU topology due to their nature.

We demonstrate that sending more messages can help to overlap the latency by increasing the number of messages while keeping the message size the same for each message. This gives a tight bound of achievable communication performance. If one can send more messages per synchronization while the message size of each message remains the same, the dot on the Message Roofline plot also moves upwards.

Another insight from the Message Roofline model is whether the communication performance can be improved by splitting the message into several smaller ones. Fig. 10 is a Message Roofline variant, showing the performance achieved by splitting one message into four smaller ones on Perlmutter GPUs. Instead of using message size on the x-axis, Fig. 10 uses message volume (equal to the number of messages × message size) on the x-axis. Therefore, as we send more messages, the size of each message decreases accordingly. It gives more optimization guides on leveraging *messages per synchronization* to optimize communication performance. Message sizes larger than 131K can get 2.9× speedups by splitting one big message into four smaller ones.

Portability. We discuss three kinds of portability. The first one is design portability. One-sided communications preserve

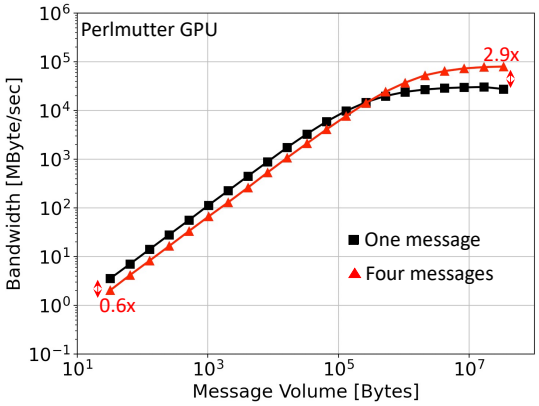


Fig. 10: Large message sizes can get up to 2.9× speedups by splitting one big message into four smaller ones on GPUs.

design portability within the SHMEM model across architectures. NVSHMEM, ROC_SHMEM, and one-sided MPI follow the SHMEM model. As such, the communication design can be shared with CPUs and GPUs. Two-sided is different from them. The second is code portability within the same processor type. For example, porting NVSHMEM code to ROC_SHMEM on devices is relatively straightforward but requires some user efforts to maintain a proper interface mapping. Conversely, porting two-sided MPI to one-sided MPI on CPUs needs a re-design and many coding efforts on data placement and receiver notification. The third is code portability across processor types. For example, porting CPU one-sided MPI to GPU NVSHMEM. Even though the design can be shared, it still requires many user efforts in engineering because the control flow and interfaces are different. It definitely requires more effort to re-design the communication scheme when porting CPU two-sided MPI to NVSHMEM.

Productivity. Effective programming models offer programmers the ability to utilize the capabilities of the underlying hardware and improve productivity. Applications like stencils that follow the BSP model may find two-sided MPI can satisfy the performance and productivity on both CPUs and GPUs. Conversely, applications with asynchronous communication patterns prefer one-sided communications across architectures.

In addition, We demonstrate that one-sided communications have the potential to provide lower latency and higher bandwidth than two-sided communication. On Perlmutter and Frontier CPUs, the one-sided communication depicts the same latency as the two-sided communications, even with a doubled number of MPI operations per send. Therefore, it can be intuitively inferred that the one-sided MPI can easily outperform the two-sided MPI with hardware-level support for put-with-signal.

Our future efforts include extending our Message Roofline Model to AMD GPUs using ROC_SHMEM and assessing other communication patterns, system interconnects, and libraries. e.g., AI applications using NCCL [30], RCCL [31], and HCCL [32]. We believe our work will be instrumental in

obtaining insights into the messaging performance of complex HPC applications running on existing and emerging hardware accelerator architectures.

ACKNOWLEDGEMENTS

This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231 and used resources of the National Energy Research Scientific Computing Center (NERSC) which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research also used resources of the Oak Ridge Leadership Facility which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This work was completed in part at the NERSC Open Hackathon, part of the Open Hackathons program. The authors would like to acknowledge OpenACC-Standard.org for their support.

REFERENCES

[1] "Top500 Highlights - November 2020." [Online]. Available: https://www.top500.org/lists/top500/2020/11/highs/

[2] "Perlmutter Interconnect," https://docs.nersc.gov/systems/perlmutter/architecture/#interconnect.

[3] "An Introduction to CUDA-Aware MPI." [Online]. Available: https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/

[4] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *Journal of parallel and distributed computing*, vol. 22, no. 2, pp. 251–267, 1994.

[5] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, pp. 1–3.

[6] "NVIDIA NVSHMEM Documentation." [Online]. Available: https://docs.nvidia.com/hpc-sdk/nvshmem/index.html

[7] "ROC_SHMEM," https://github.com/ROCm-Developer-Tools/ROC_SHMEM, 2020.

[8] G. Almasi, "Pgas (partitioned global address space) languages." 2011.

[9] S. Potluri, N. Luehr, and N. Sakharnykh, "Simplifying multi-gpu communication with nvshmem," in *GPU Technology Conference*. NVIDIA, 2016.

[10] C.-H. Hsu, N. Imam, A. Langer, S. Potluri, and C. J. Newburn, "An initial assessment of nvshmem for high performance computing," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 1–10.

[11] K. Hamidouche and M. LeBeane, "Gpu initiated openshmem: correct and efficient intra-kernel networking for dgpus," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 336–347.

[12] C. Xie, J. Chen, J. S. Firoz, J. Li, S. L. Song, K. Barker, M. Raugas, and A. Li, "Fast and scalable sparse triangular solver for multi-gpu based hpc architectures," *arXiv preprint arXiv:2012.06959*, 2020.

[13] N. Ding, Y. Liu, S. Williams, and X. S. Li, "A message-driven, multi-gpu parallel sparse triangular solver," in *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM, 2021, pp. 147–159.

[14] V. Cardellini, A. Fanfarillo, S. Filippone *et al.*, "Overlapping communication with computation in mpi applications," *Università degli Studi di Roma Tor Vergata Technical Reports*, 2016.

[15] N. Ding, S. Williams, Y. Liu, and X. S. Li, "Leveraging one-sided communication for sparse triangular solvers," in *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 2020, pp. 93–105.

[16] R. Belli and T. Hoefler, "Notified access: Extending remote memory access programming models for producer-consumer synchronization," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 871–881.

[17] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling highly-scalable remote memory access programming with mpi-3 one sided," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[18] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[19] "Frontier User Guide." [Online]. Available: https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#frontier-compute-nodes

[20] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation," in *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, 1995, pp. 95–105.

[21] "Perlmutter User Guide - CPU Architecture." [Online]. Available: https://docs.nersc.gov/systems/perlmutter/architecture/#cpu-nodes

[22] "Summit User Guide." [Online]. Available: https://docs.olcf.ornl.gov/systems/summit_user_guide.html#summit-nodes

[23] "Perlmutter User Guide - GPU Architecture." [Online]. Available: https://docs.nersc.gov/systems/perlmutter/architecture/#gpu-nodes

[24] D. Chen, N. Eisley, P. Heidelberger, S. Kumar, A. Mamidala, F. Petrini, R. Senger, Y. Sugawara, R. Walkup, B. Steinmacher-Burow *et al.*, "Looking under the hood of the ibm blue gene/q network," in *SC'12: Proceedings of the International Conference on High Performance*

Computing, Networking, Storage and Analysis. IEEE, 2012, pp. 1–12.

[25] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2019.

[26] T. Groves, B. Brock, Y. Chen, K. Z. Ibrahim, L. Oliker, N. J. Wright, S. Williams, and K. Yelick, *Performance trade-offs in GPU communication: A study of host and device-initiated approaches*. IEEE, 2020.

[27] K. Spafford, J. S. Meredith, and J. S. Vetter, "Quantifying numa and contention effects in multi-gpu systems," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 1–7.

[28] Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, and D. Kaeli, "Evaluating performance tradeoffs on the radeon open compute platform," in *2018 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2018, pp. 209–218.

[29] A. S. Dufek, J. R. Deslippe, P. T. Lin, C. J. Yang, B. G. Cook, and J. Madsen, "An extended roofline performance model with pci-e and network ceilings," in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2021, pp. 30–39.

[30] S. Jeaugey, "Nccl 2.0," in *GPU Technology Conference (GTC)*, vol. 2, 2017.

[31] "RCCL 2.16.5 Documentation." [Online]. Available: https://rocm.docs.amd.com/projects/rccl/en/latest/

[32] "Accelerating Distributed Deep Learning using HCCL: Habana Collective Communication Library." [Online]. Available: http://nowlab.cse.ohio-state.edu/static/media/workshops/presentations/exacomm23/Habana_ExaComm_2023.pdf

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: [EVALUATING THE PERFORMANCE OF ONE-SIDED COMMUNICATION ON CPUS AND GPUS]

A. Abstract

The key contribution of this paper is the methodology of the Message Roofline Model for GPUs. The hardware and software environment used in this paper are all publicly available as described below.

B. Hardware

Results presented in this paper were obtained on Perlmutter at NERSC, Summit and Frontier at OLCF, as listed in Table III Perlmutter GPU partition is comprised of nodes with one AMD EPYC 7763 (Milan) CPU and four NVIDIA A100 GPUs. Perlmutter CPU partition has two AMD EPYC 7763 (Milan) CPUs. Each compute node on Summit contains two IBM POWER9 processors and six NVIDIA V100 accelerators, while each Frontier node consists of one 64-core AMD "Optimized 3rd Gen EPYC" CPU and four AMD MI250X GPUs.

C. Run-time environment

Table III presents the used run-time environments on all machines. We use NVSHMEM v2.8.0 with GDRcopy in all GPU experiments. We use cudatoolkit v11.7 on Perlmutter GPU partition and CUDA v11.0.3 on Summit GPUs. We use CrayMPI for two-sided and one-sided experiments on Perlmutter CPU partition and Frontier CPUs, while we use IBM Spectrum MPI on Summit. Note we did not experiment on Frontier AMD GPUs due to the lack of support of "wait_until_any" in ROC_SHMEM.

D. Benchmarks

The three benchmarks we evaluated in the paper are described below.

1) Sparse Triangular Solve (SpTRSV), the source code can be found https://github.com/xiaoyeli/superlu_dist/tree/new_multiGPU.
2) Stencil, the source code can be found https://web.cels.anl.gov/~thakur/sc16-mpi-tutorial/.
3) Distrbiuted HashTable, the source code can be found https://spcl.inf.ethz.ch/Research/Parallel_Programming/foMPI/.

E. Installation

The NVSHMEM installation script named "install_perlmutter.sh" and "install_summit.sh" on Perlmutter and Summit can be found https://github.com/nanding0701/nvshmem_test/tree/main.

The installation scripts of SpTRSV can be found https://github.com/xiaoyeli/superlu_dist/tree/gpu3d-batch/example_scripts. The installation script follows the naming convention by run_cmake_build_{machine}_{compiler}_nvshmem.sh for GPUs and run_cmake_build_{machine}_gcc_nogpu.sh for CPUs.

The Makefile of Stencil and Distributed HashTable are included in their source code.

F. Experiment workflow

We run SpTRSV with a matrix from M3D-C1, a fusion simulation code used for magnetohydrodynamics modeling of plasma. The matrices are first factorized via SuperLU_DIST with METIS ordering for fill-in reduction. The matrix dimension is 126K $\times$ 126K with $1e+08$ number of non-zeros after factorization.

We run Stencil using the grid size of 16384$\times$16384, and Distributed HashTable using one million inserts. There is no input for these two benchmarks. One can pass in the grid size and number of inserts via the command line.

TABLE III: Hardware and Run-time environments

Machine	GPUs per node	GPU Interconnect	GPU Runtime	GPU-CPU Interconnect	CPUs	CPU-CPU Interconnect	CPU Runtime	CPU-NIC Interconnect
Summit	6×V100	NVLINK2	CUDA v11.0.3 NVSHMEM v2.8.0	NVLINK2	2×IBM POWER9	X-Bus	IBM Spectrum	PCIe4.0
Perlmutter GPU	4×A100	NVLINK3	cudatoolkit v11.7 NVSHMEM v2.8.0	PCIe4	1×AMD EPYC 7763	-	-	PCIe4.0
Perlmutter CPU	-	-	-	-	2×AMD EPYC 7763	Infinity Fabric	CrayMPI	PCIe4.0
Frontier CPU	-	-	-	-	1×AMD EPYC 7A53	Infinity Fabric	CrayMPI	Infinity Fabric and PCIe4.0 ESM

G. Evaluation and expected result

The SpTRSV requires some run-time settings. The running script of SpTRSV can be found https://github.com/xiaoyeli/superlu_dist/tree/new_multiGPU/example_scripts. Similar to the installation scripts, it follows the naming convention by batch_script_mpi_runit_{machine}_{compiler}_nvshmem.sh for GPUs and batch_script_mpi_runit_{machine}_gcc_nogpu.sh for CPUs. One can refer to the SOLVE time reported at the end of each run.

The running command of Stencil is “srun -n4 -c32 –cpu-bind=cores -G 4 ./stencil 16384 1 1000 2 2 ” on Perlmutter GPUs, and “jsrun -n6 -c2 -a1 -g1 ./stencil 16384 1 1000 3 2 ” on Summit GPUs. The parameters represent the grid size, energy, number of iterations, and process decomposition in x- and y-axis. The timing will be printed at the end of each run.

The running command of Distributed Hashtable is “srun -n4 -c32 –cpu-bind=cores -G 4 ./hashtable 250000 ” on Perlmutter GPUs and “jsrun -n6 -c2 -a1 -g1 ./hashtable 166666” on Summit GPUs. Note that both commands are performing one million inserts in total. As more processes (GPUs) are used, the number of inserts of each GPU reduces accordingly. One must manually calculate that number and pass it in when running the benchmark. The timing will be printed at the end of each run.