Argonne NATIONAL LABORATORY

ANL-20/XXXX

Evaluation of PETSc on a Heterogeneous Architecture the OLCF Summit System Part II: Basic Communication Performance

Mathematics and Computer Science Division

ANL-20/XXXX

Evaluation of PETSc on a Heterogeneous Architecture the OLCF Summit System Part II: Basic Communication Performance

Prepared by
Junchao Zhang, Richard Tran Mills, and Barry Smith
Mathematics and Computer Science Division, Argonne National Laboratory

September 2020
This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

Evaluation of PETSc on a Heterogeneous Architecture the OLCF Summit System Part II: Basic Parallel Communication Performance

Junchao Zhang, Richard Tran Mills, Barry Smith
Mathematics and Computer Science Division
Argonne National Laboratory

Abstract

Nearest neighbor communication is at the heart of many parallel high performance computing computations. We report on the performance of such communication on the Oak Ridge Leadership Computing Facility system Summit in the context of the communication module in PETSc. The analysis in this report includes basic ping-pong style point to point communication, regular and irregular nearest neighbor communication.

1 Introduction

We report on the performance of the Portable, Extensible Toolkit for Scientific Computation (PETSc) [3, 4] communication infrastructure using basic ping-pong style point to point communication, regular and irregular nearest neighbor communication on the IBM/NVIDIA Summit computing system [2] at the Oak Ridge Leadership Computing Facility (OLCF). Using the organization of the PETSc library, many PETSc solvers and preconditioners are able to run with GPU vector and matrix implementations. This report is a continuation of the previous report: Part I [7] that introduces the Summit architecture and analyses the on-node performance characteristics. The Part III report [] continues the analysis in this report for unstructured mesh communication for partial differential equations. This report builds on the analysis of the previous report and thus will not repeat the detailed material in that report.

The planned United States Department of Energy exascale computing systems [11] have designs similar to that of Summit. Thus, it is important to have a well-developed understanding of Summit in preparation for these systems. This document is not intended to provide a strict benchmarking of the Summit system; rather it is to develop an understanding of systems similar to Summit, in order to guide PETSc development.

2 The Summit System and Experimental Setup

Figure 1 shows the basic communication pathway of a Summit compute node. Each node has two CPU sockets and each socket contains one IBM POWER9 CPU, accompanied by three NVIDIA Volta V100 GPUs. The CPU and GPUs are connected by NVIDIA's NVLink interconnect, which has a bi-directional bandwidth of 50GB/s. Communication between the two CPUs are provided by IBM's X-Bus, with a bi-directional bandwidth of 64GB/s. Each CPU also connects to a single Mellanox InfiniBand ConnectX-5 (EDR IB) network interface card (NIC) through a PCIe Gen4 x8 bus with a bi-directional bandwidth of 16GB/s. The NIC has an injection bandwidth of 25GB/s.

PETSc uses MPI for communication between processes. When data is in GPU memory, PETSc is able to copy the data to CPU memory and perform the communication with regular MPI on CPUs and then copy the data to GPUs. The preferred approach, however, is to use CUDA-aware MPI, with which PETSc can pass device pointers directly to MPI routines. In this report, we focus on this preferred approach as it provides better performance. A quality CUDA-aware MPI implementation would use NVIDIA's GPUDirect Point-to-Point(P2P) and remote direct memory access (RDMA) technologies. With GPUDirect P2P, data can be directly copied between the memories of two GPUs within a node. With GPUDirect RMDA, GPUs can

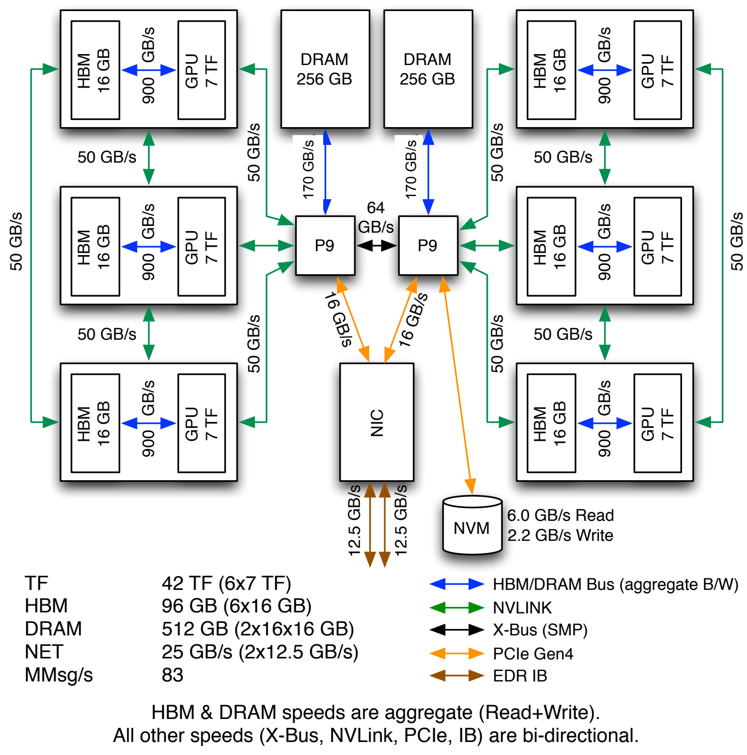


Figure 1: Diagram of a Summit node with communication pathway [8]

communicate directly to the NIC and send or receive data without staging in CPU memory. Obviously, the former is useful in MPI intra-node communication and the latter is useful in MPI inter-node communication.

The NIC that connects the node to the parallel network is connected to a programmable “local network” that connects it to the CPU memory as well as the GPU memory. This means the parallel communication latency and bandwidth (see Report I) are limited by the NIC, the local network, the NVLinks from the CPU to the local network and the GPUs memory, but not the CPU memory. However, CUDA-aware MPI calls (send, receive, and waits) must currently be called by code running on the CPU cores. There is ongoing research in triggering the MPI communication from within CUDA kernels to avoid the extra CPU to GPU operations but these are not currently available. The total communication time is a combination of the physical/software latencies and bandwidths of the various hardware components plus the latencies and bandwidths induced by the software stack.

3 MPI Point-to-Point Latency on Summit

In [9], the authors evaluated MPI point to point latency and bandwidth on a GPU-enabled OpenPower system similar to Summit, using MPI implementations including MVAPICH2-GDR, OpenMPI and IBM Spectrum MPI. In this section, we repeat their latency experiments on Summit. We only use Spectrum MPI since it is the only supported MPI on the machine; the others are difficult to install and use. Measuring MPI performance on Summit is not the purpose of this report. What we want to know is what communication performance PETSc can provide, since PETSc users and PETSc code itself usually do not directly call MPI, instead they do it through PETSc application programming interfaces (APIs). If an MPI implementation has better performance, PETSc surely can ride on that.

We used osu_latency from the OSU Microbenchmarks 5.6.2 [10], which can measure latency with CPU buffers or GPU buffers. We focus on the GPU case in this report. This test is also known as the MPI ping-poing test. Shown in Figure 2, it uses two MPI ranks and allocates a send buffer (`sbuf`) and a recieve buffer (`rbuf`) on each rank. The buffers are long enough (e.g., 4MB bytes). Rank 0 `MPI_Sends` a message of a certain size from its send buffer to rank 1’s receive buffer. Once rank 1 `MPI_Recvs` the message, rank

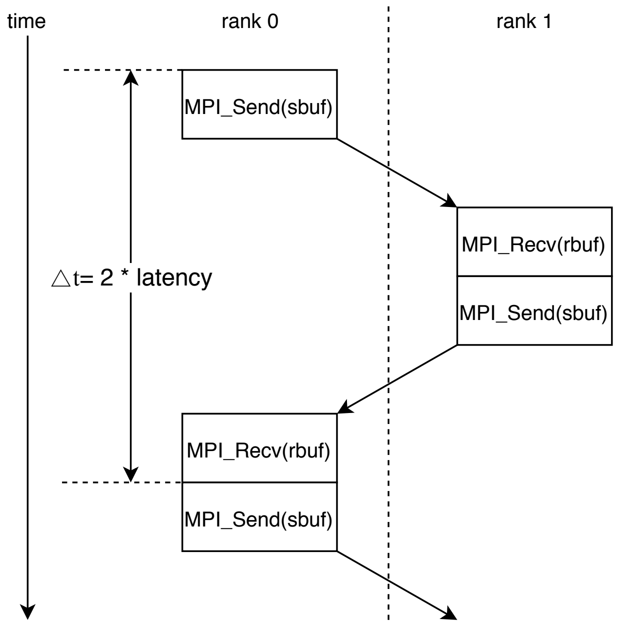


Figure 2: OSU Microbenchmarks latency test [10]

1 replies a message of the same size from its send buffer to rank 0's receive buffer. After rank 0 gets the reply, it finishes a round-trip from rank 0 to rank 1. The round-trip is repeated many times (10,000 times for messages $\leq$ 8KB and 1,000 times otherwise). The latency is calculated as the average time of a one-way trip. Looking at Figure 2, one might find rank 1 does not reply with the message it got from rank 0 (i.e., what in its `rbuf`). Instead, it sends data in its `sbuf`. This design is used to minimize cache effect, though that is not very important on GPUs as we later found. The microbenchmark uses `MPI_Wtime` for timing and assumes send buffers are ready for MPI, so there are no any kind of CUDA synchronizations involved.

We placed the two MPI ranks on the same GPU, on two GPUs attached to the same CPU, on two GPUs attached to different CPUs within a node, and on two GPUs across nodes and got latency results for them in Table 1, which we call intra-GPU, intra-socket, inter-socket and inter-node latency respectively. Though the microbenchmark can test message sizes starting from 0, we omitted results for messages smaller than 8 bytes for brevity. The intra-GPU results are better than those reported in Figure 6 of [9]. The remaining results largely match with those in Figures 4, 10, 12 of [9]. We can regard these performance numbers as an upper bound that a similar PETSc benchmark could achieve.

For a message of size s, its MPI ping-pong latency l can be modeled as $l = \alpha + \beta s$, where α is the start-up cost and β is reciprocal of the MPI send/recv bandwidth. Taking latency at 8 bytes as α, and applying the formula to messages at size 4MB, we can then get the bandwidth. The intra-GPU, intra-socket, inter-socket and inter-node MPI send/recv bandwidths are 364.7GB/s, 47.2GB/s, 34.5GB/s and 9.7GB/s respectively. The intra-GPU bandwidth reaches 81.0% of half of the GPU memory bandwidth at 900GB/s (note we both read and write the same GPU memory in this case). The intra-socket, inter-socket bandwidths reach 94.5%, 69.0% of the NVLink bandwidth at 50GB/s respectively, while the inter-node one only reaches 38.8% of the EDR IB bandwidth at 25GB/s. Since GPU virtualization on Summit comes with some cost, up to 20%, it is highly recommended that one uses one MPI rank per physical GPU. In the following studies in this report we follow this convention.

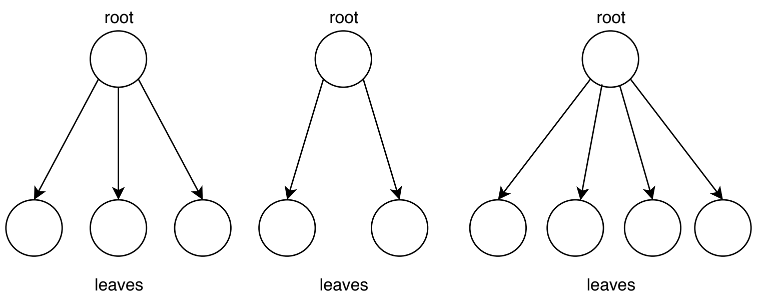


Figure 3: A star-forest example

Message size (bytes)	Latency (µs)			
	Intra-GPU	Intra-socket	Inter-socket	Inter-node
8	20.1	17.8	19.3	6.0
16	20.1	17.8	19.4	6.0
32	20.1	17.8	19.4	6.8
64	20.1	17.8	19.5	6.0
128	20.1	17.8	19.5	6.1
256	20.1	17.8	19.4	6.2
512	20.1	17.8	19.5	6.2
1K	20.1	17.8	19.4	6.3
2K	20.0	17.8	19.4	6.8
4K	20.1	17.8	19.4	7.2
8K	20.1	17.8	19.5	8.2
16K	20.1	17.8	19.5	9.3
32K	20.0	17.8	19.4	11.4
64K	20.1	18.5	20.1	14.1
128K	20.1	20.0	21.6	19.9
256K	20.1	22.6	24.6	30.5
512K	20.4	28.2	30.9	51.8
1M	20.7	39.4	43.2	98.2
2M	25.6	61.7	68.2	191.2
4M	31.6	106.6	140.9	436.7

Table 1: MPI ping-pong latency[1] measured by osu_latency from the OSU Microbenchmarks [10]

4 The Communication Module in PETSc

4.1 Introduction

PetscSF is PETSc's communication module. It is heavily used by other PETSc modules internally. Applications can also call it directly. VecScatter, a public interface for communications on PETSc vectors, is also implemented in PetscSF. PetscSF abstracts nearest neighbor communications into a star-forest (SF) graph. An SF is a forest containing multiple star-shaped trees, where each tree has a height of one, with one root and multiple leaves. See Figure 3 for an example.

To build a PetscSF, users need to provide on each MPI processes two integer-indexed spaces: the leaf space and the root space. Leaves in the leaf space can be dense (i.e., contiguous) or sparse, and must be local to the process such that an integer can identify a leaf. Roots must be dense. Roots might be remote, in that case one uses (`rank`, `index`) pairs to specify roots the local leaves connect to, where `rank` is the MPI rank a root resides in, and `index` is the *local* index of the root on that MPI rank.

[1]It's worth noting we observed big variations in the inter-node big messages tests (e.g., 4MB), which could be 20% higher than what reported here. We thought that was due to location of the two nodes allocated by the job system.

PetscSF provides split-phase communication routines to communicate between roots and leaves of an SF. For example, `PetscSFReduceBegin/End` reduces leaves to their connected roots with a given MPI reduction operation. `PetscSFBcastBegin/End` broadcasts roots to their connected leaves. Users are expected to put computation in between `PetscSFXxxBegin/End` so that communication and computation could be overlapped. In addition, one can interleave communications on the same SF with different leaf data or root data.

4.2 PetscSF Implementation

On each MPI process, PetscSF internally computes the process's neighbors (a list of destination ranks and source ranks) with which the process will communicate, i.e., send data to or receive data from. For each destination, it computes indices of local data (leaves or roots depending on the context) which it needs to send. For each source, it computes indices of local data where it should deposit the received data. When a neighbor is the process itself, we call the communication *self* or *local* communication; Otherwise we call it *remote* communication. We separate local and remote communications since for the local one we can bypass MPI and enjoy unique optimization opportunities.

For remote communication, PetscSF in general allocates on each MPI process a send buffer and a receive buffer. Let's use `PetscSFReduceBegin(sf,unit,leafdata,rootdata,op)` as an example. A process packs selected entries of `leafdata` into the send buffer and then sends them out. After it receives data it needs in the receive buffer, it unpacks entries from the buffer and deposits them back to rootdata. Each remote neighbor takes its own chunk from the send or receive buffer. PetscSF's pack/unpack routines are overloaded according to location of the root/leafdata. When data is in CPU memory, the routines are CPU functions; when data is in GPU memory, the routines are CUDA kernels, where each CUDA thread works on a leaf/root. PetscSF will use atomic instructions in unpack CUDA kernels when there are data race chances.

PetscSF employs index analysis to set up optimizations to lower packing cost. The analysis is done in PetscSF setup phase, with a low cost that could also be amortized by multiple calls to PetscSF. For instance, in `PetscSFReduce`, when leaf indices used in packing happen to be contiguous, PetscSF disguises leafdata as the send buffer and completely avoids packing. Still with `PetscSFReduce`, when root indices for unpacking are contiguous, can it disguise rootdata as the receive buffer and avoid unpacking? That depends on the reduction argument `op`. If op is `MPI_REPLACE` (similar to `INSERT_VALUES` in VecScatter), it can; Otherwise, it can't and has to allocate a receive buffer and launches an unpack kernel performing the reduction such as `MPI_SUM`. Even in this case, it takes advantage of the fact that root indices are contiguous. It avoids copying root indices to GPUs and uses simpler expressions in the unpack kernel. Note that PetscSF employs persistent `MPI_Isend/Irecv` for communication. With this data and buffer disguising, that means in an SF's lifetime it may encounter different send/receive buffers. PetscSF handles this complexity and makes them work with MPI persistent requests. PetscSF does buffer allocation and MPI persistent request initialization on-demand, in the sense that it only uses resources when needed.

For local communication, PetscSF abstracts it as a scatter operation: `x[idx[i]]` $\rightarrow$ `y[idy[i]]`, for `i` $\in$ [0, n). The scatter is a GPU kernel when data is on GPU. It uses simpler expressions like `x[startx+i]` $\rightarrow$ `y[idy[i]]` when it knows indices in `idx[]` are contiguous and `startx` is the first. There are other variants, such as the scatter is simply a memory copy, or even a no-op when it finds out it is a memory copy with the destination and the source having the same address. PetscSF exploits these opportunities to simplify local communication.

In `PetscSFXxxBegin()`, it first checks memory types of the input rootdata and leafdata, to know whether they point to CPU or GPU memory. It needs this info to set up data structures such as pack routines. Then it posts `MPI_Irecv` requests through `MPI_Startall`, calls a pack routine to pack source data into the send buffer, and posts `MPI_Isend` requests. After that, it calls a scatter routine to do local communication. In `PetscSFXxxEnd()`, it waits for the requests it has posted with `MPI_Waitall`. At the end, it calls an unpack routine to unpack data from the receive buffer. The pack/unpack is skipped sometimes as discussed above.

Although PetscSF is able to communicate data on GPUs without GPU-aware MPI support, we focus exclusively in this report on code path using GPU-aware MPI since it avoids back-and-forth buffer copying between CPUs and GPUs and has superior performance.

CUDA kernels are executed asynchronously with respect to CPUs. When a PetscSF routine is called, the leaf/root data might be being computed by some CUDA kernels on CUDA streams which are unknown to PetscSF, therefore in theory PetscSF has to call `cudaDeviceSynchronize()` to wait for the data to be ready. PetscSF could launch pack/unpack kernels on its own stream. On the sender side,

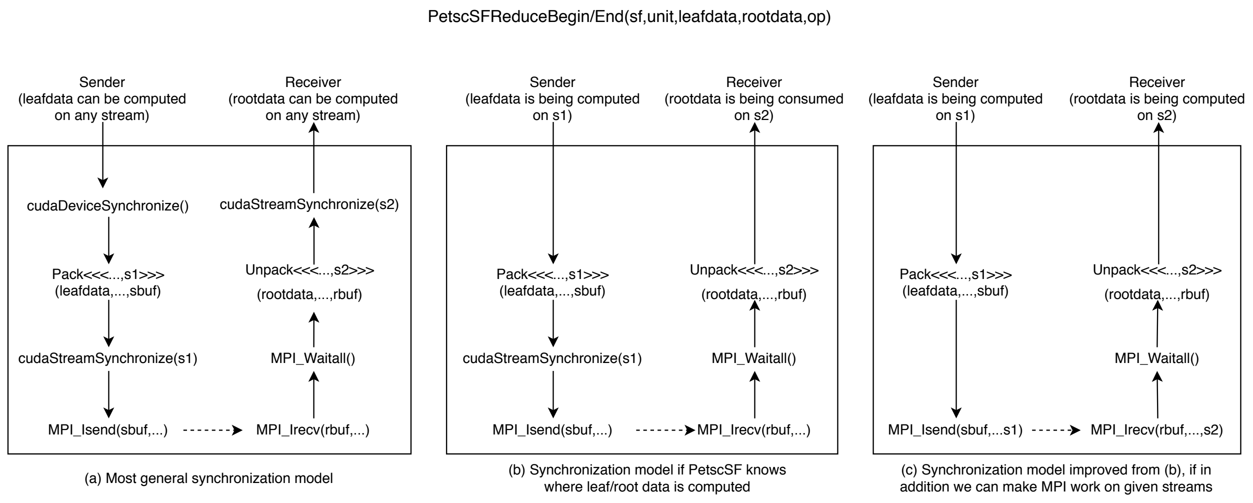

PetscSFReduceBegin/End(sf,unit,leafdata,rootdata,op)
Sender (leafdata can be computed on any stream)
Receiver (rootdata can be computed on any stream)
cudaDeviceSynchronize()
cudaStreamSynchronize(s2)
Pack<<<...,s1>>> (leafdata,...,sbuf)
Unpack<<<...,s2>>> (rootdata,...,rbuf)
cudaStreamSynchronize(s1)
MPI_Waitall()
MPI_Isend(sbuf,...)
MPI_Irecv(rbuf,...)
(a) Most general synchronization model
Sender (leafdata is being computed on s1)
Receiver (rootdata is being consumed on s2)
Pack<<<...,s1>>> (leafdata,...,sbuf)
Unpack<<<...,s2>>> (rootdata,...,rbuf)
cudaStreamSynchronize(s1)
MPI_Waitall()
MPI_Isend(sbuf,...)
MPI_Irecv(rbuf,...)
(b) Synchronization model if PetscSF knows where leaf/root data is computed
Sender (leafdata is being computed on s1)
Receiver (rootdata is being computed on s2)
Pack<<<...,s1>>> (leafdata,...,sbuf)
Unpack<<<...,s2>>> (rootdata,...,rbuf)
MPI_Waitall()
MPI_Isend(sbuf,...s1)
MPI_Irecv(rbuf,...,s2)
(c) Synchronization model improved from (b), if in addition we can make MPI work on given streams

Figure 4: Different synchronization models in PetscSF

PetscSF calls `cudaStreamSynchronize()` on the stream before `MPI_Isend`. On the receiver side, after `MPI_Waitall`, PetscSF is assured that data is received. It launches an unpack kernel and then calls `cudaStreamSynchronize()` again to assure the data is ready for PetscSF clients (either applications or other modules of PETSc). This procedure is demonstrated in Figure 4(a) using `PetscSFReduce` as an example. One can see that there are many synchronizations involved. If PetscSF could know the streams where leaf/rootdata is produced or consumed, it could save the synchronizations before `Pack` and after `Unpack` as shown in Figure 4(b). Further, if MPI routines are CUDA-stream aware, e.g., by taking a stream argument or other means, and work more like a kernel launch, we then could remove the synchronization before `MPI_Isend` as shown in Figure 4(c). This requires support from MPI that is currently not available. One can refer to the MPI and CUDA semantic mismatch discussion in [6].

Model (a) is the most general model. Since PETSc currently only uses the CUDA default stream, we provide an option `-sf_use_default_stream` to let PetscSF skip the `cudaDeviceSynchronize()` before `Pack` and the `cudaStreamSynchronize()` after `Unpack`. This option turns Model (a) into Model (b) in Figure 4 (with `s1 = s2 = NULL`). For experiments, we also provide an option `-sf_use_stream_aware_mpi` pretending the underlying MPI knows where the send/receive data is being produced/consumed, so that it can get rid of the `cudaStreamSynchronize()` after `Pack` and turns Model (b) into Model (c).

5 Experimental Results

5.1 PetscSF without pack/unpack

We wrote a ping-pong test using PetscSF, which had the same parameters as those in the OSU ping-pong test used in Section 3. Suppose we want to measure latency for a message of size 8n. We build an SF in which rank 0 has n roots and zero leaves, while rank 1 has 0 roots and n leaves, as shown in Figure 5. Rank 1's leaves are one-on-one sequentially connected to rank 0's roots. With this SF, `PetscSFBcast` will be a send from rank 0 to rank 1, while `PetscSFReduce` will be a send from rank 1 to rank 0. We used double-precision and PETSc's `MPIU_SCALAR` (same as `MPI_DOUBLE`) as the MPI datatype for roots and leaves. In other words, a root or leaf is eight bytes. We built different SFs for different message sizes. The following loop shows a ping-pong test for a given message size. Note that `sbuf` and `rbuf` in the code work as a pair of rootdata on rank 0, and as a pair of leafdata on rank 1, which is intended to mimic the behavior in the OSU test.

Since in this test root/leaf indices are contiguous and we do not actually do reduction on roots, PetscSF has optimizations that directly use `sbuf` or `rbuf` as MPI's send/receive buffers and avoid packing/unpacking kernels. In other words, we get a simplified code path like the one in Figure 6(a). To get rid of the `cudaDeviceSynchronize()` before `MPI_Isend`, we use option `-sf_use_default_stream` indicating root/leaf data is good to use on the default stream, and get the code path in Figure 6(b). The

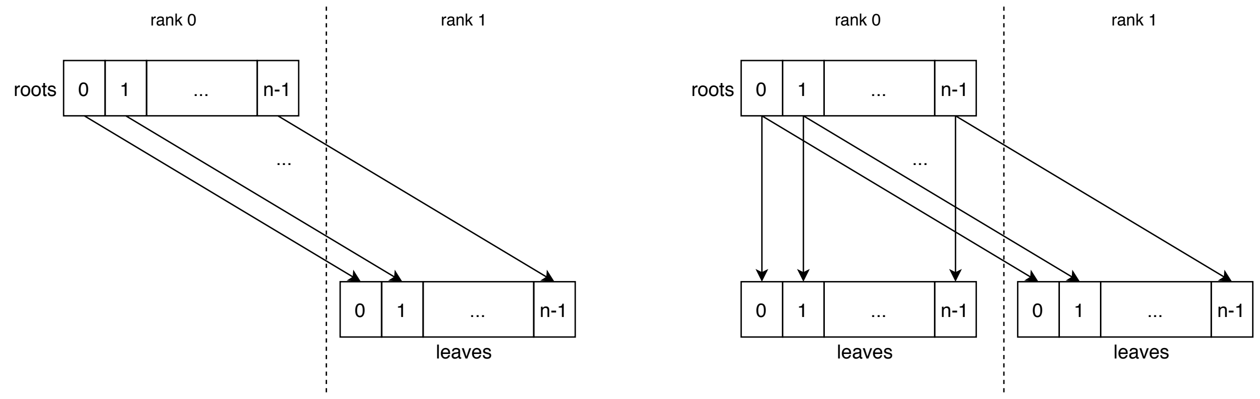


Figure 5: Star-forests in the PetscSF Ping-pong/Unpack tests (left), and in the PetscSF Scatter test (right)

```
for (i=0; i<niter; i++) {
  ierr = PetscSFBcastBegin(sf,MPIU_SCALAR,sbuf,rbuf);CHKERRQ(ierr);
  ierr = PetscSFBcastEnd(sf,MPIU_SCALAR,sbuf,rbuf);CHKERRQ(ierr);
  ierr = PetscSFReduceBegin(sf,MPIU_SCALAR,sbuf,rbuf,MPIU_REPLACE);CHKERRQ(ierr);
  ierr = PetscSFReduceEnd(sf,MPIU_SCALAR,sbuf,rbuf,MPIU_REPLACE);CHKERRQ(ierr);
}
```

Listing 1: sf_pingpong benchmark loop

`cudaStreamSynchronize(NULL)` is there because the condition that leaf data is on the default stream does not necessarily mean it is ready for MPI to send. To get rid of it, we use option `-sf_use_stream_aware_mpi` indicating MPI knows which streams to get input data or put output data. Though IBM Spectrum can not do that, it does not matter in this simple test since the input data is always ready and we do not use the output data. This gives us the code path in Figure 6(c).

We measured intra-socket GPU to GPU latency for the three variants. The results are show in columns Opt-A/B/C respectively. Comparing intra-socket columns Opt-A and Opt-B, we can see `cudaDeviceSynchronize()` has a slightly higher cost (about 1.5µs) than `cudaStreamSynchronize()`. Comparing intra-socket columns Opt-B and Opt-C, we know cost of a `cudaStreamSynchronize()` call is about 4µs, since Opt-C does not have synchronizations at all. We profiled the code with Opt-C and found a notable routine was a CUDA driver call `cuPointerGetAttribute()`, which was called twice in `PetscSFXxxBegin()` to test pointer attributes of the arguments rootdata and leafdata. Since we knew in this test they were GPU pointers, we manually modified PetscSF code and bypassed the CUDA driver call. The results are in column Opt-D. Comparing it with the intra-socket column in Table 1, we can see the minimal overhead of PetcSF is around 1µs over pure MPI, which is quite satisfying. Overall, PetscSF ping-pong latency is about 6µs longer than pure MPI. For completeness, Table 2 also shows inter-socket and inter-node latency with Opt-B, which is PETSc's default model and we will use it for remaining tests in this report. Comparing the most general synchronization model in Figure 4(a) and PETSc's default model in Figure 4(b), the former has one `cudaDeviceSynchronize()` and one `cudaStreamSynchronize()`, whose cost is about 9µs in total, based on above analysis.

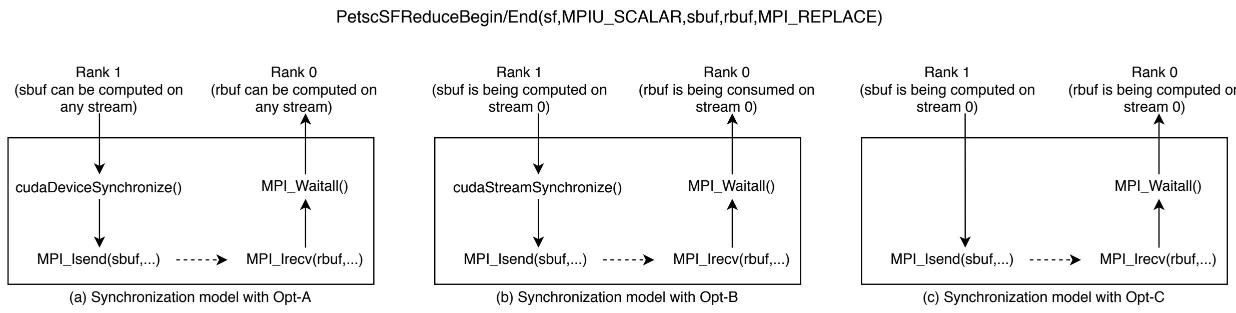

PetscSFReduceBegin/End(sf,MPIU_SCALAR,sbuf,rbuf,MPI_REPLACE)
Rank 1 (sbuf can be computed on any stream)
Rank 0 (rbuf can be computed on any stream)
cudaDeviceSynchronize()
MPI_Waitall()
MPI_Isend(sbuf,...)
MPI_Irecv(rbuf,...)
(a) Synchronization model with Opt-A
Rank 1 (sbuf is being computed on stream 0)
Rank 0 (rbuf is being consumed on stream 0)
cudaStreamSynchronize()
MPI_Waitall()
MPI_Isend(sbuf,...)
MPI_Irecv(rbuf,...)
(b) Synchronization model with Opt-B
Rank 1 (sbuf is being computed on stream 0)
Rank 0 (rbuf is being computed on stream 0)
MPI_Waitall()
MPI_Isend(sbuf,...)
MPI_Irecv(rbuf,...)
(c) Synchronization model with Opt-C

Figure 6: Code paths in the sf_pingpong test with different synchronization models

Message size (bytes)	Intra-socket latency (µs)				Latency (µs) with Opt-B	
	Opt-A	Opt-B	Opt-C	Opt-D	Inter-socket	Inter-node
8	25.3	23.8	19.9	19.0	25.4	12.0
16	25.2	23.7	19.7	19.0	25.4	11.6
32	25.2	23.6	19.7	18.9	25.3	11.6
64	25.2	23.7	19.7	19.0	25.3	11.6
128	25.2	23.6	19.8	19.0	25.3	11.9
256	25.2	23.6	19.8	19.0	25.4	11.8
512	25.2	23.6	19.8	19.0	25.3	11.8
1K	25.2	23.5	19.8	19.0	25.3	11.9
2K	25.2	23.6	19.8	19.0	25.3	12.5
4K	25.1	23.6	19.8	19.0	25.3	12.9
8K	25.0	23.5	19.6	18.9	25.3	13.9
16K	25.3	23.5	19.8	18.9	25.3	15.1
32K	25.3	23.5	19.8	19.0	25.4	17.2
64K	25.7	24.3	20.5	19.7	25.9	19.8
128K	27.3	25.5	21.7	20.9	27.5	25.7
256K	30.0	28.3	24.5	23.6	30.5	36.2
512K	35.5	34.0	30.1	29.3	36.8	58.8
1M	46.8	45.1	41.3	40.5	49.2	104.3
2M	68.9	67.3	63.6	62.8	74.3	197.0
4M	113.9	112.5	108.6	107.9	147.2	441.2

Table 2: sf_pingpong latency. Options used: Opt-A = `-use_gpu_aware_mpi`; Opt-B = Opt-A + `-sf_use_default_stream`; Opt-C = Opt-B + `-sf_use_stream_aware_mpi`; Opt-D = Opt-C + manually set types of root/leafdata as GPU pointers. PETSc's default is Opt-B.

5.2 PetscSF with unpack and local communication

We now turn to unpack kernels and local communications. We slightly modified the sf_pingpong test and created a new test called *sf_unpack*. For easy understanding, in sf_unpack we used only one set of root data on rank 0 and one set of leaf data on rank 1. We added roots to leaves with `PetscSFBcastAndOp` and leaves to roots with `PetscSFReduce` using code in Listing 2. Because of `MPI_SUM`, we need a receive buffer at the destination and an unpack kernel performing the addition. With PETSc's default option, we got a code path shown in Figure 7. Comparing it with Figure 6(b), we paid an extra cost for calling `Unpack`, including kernel launch time and kerenl execution time.

To add local communication, we created another test called *sf_scatter* by simply changing the SFs used in sf_unpack. We added leaves on rank 0 and made them connected to its roots one-on-one. An example SF is shown in the left of Figure 7. With the new SFs and the same code in Listing 2, `PetscSFBcastAndOp` will add roots on rank 0 to both local and remote leaves; and `PetscSFReduce` will add both local and remote leaves to roots. The code path for `PetscSFReduce` is shown in the right of Figure 7. On rank 0, the local

```
for (i=0; i<niter; i++) {
  ierr = PetscSFBcastAndOpBegin(sf,MPIU_SCALAR,rootdata,leafdata,MPI_SUM);CHKERRQ(ierr);
  ierr = PetscSFBcastAndOpEnd(sf,MPIU_SCALAR,rootdata,leafdata,MPI_SUM);CHKERRQ(ierr);
  ierr = PetscSFReduceBegin(sf,MPIU_SCALAR,leafdata,rootdata,MPI_SUM);CHKERRQ(ierr);
  ierr = PetscSFReduceEnd(sf,MPIU_SCALAR,leafdata,rootdata,MPI_SUM);CHKERRQ(ierr);
}
```

Listing 2: sf_unpack benchmark loop

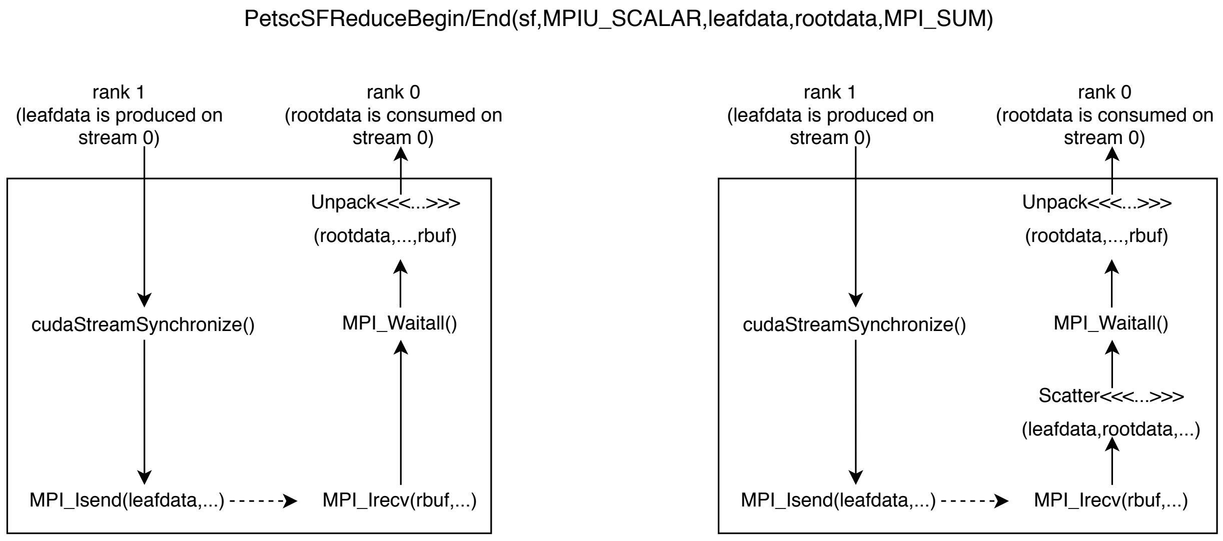

PetscSFReduceBegin/End(sf,MPIU_SCALAR,leafdata,rootdata,MPI_SUM)
rank 1 (leafdata is produced on stream 0)
rank 0 (rootdata is consumed on stream 0)
Unpack<<<...>>> (rootdata,...,rbuf)
cudaStreamSynchronize()
MPI_Waitall()
MPI_Isend(leafdata,...)
MPI_Irecv(rbuf,...)
rank 1 (leafdata is produced on stream 0)
rank 0 (rootdata is consumed on stream 0)
Unpack<<<...>>> (rootdata,...,rbuf)
cudaStreamSynchronize()
MPI_Waitall()
Scatter<<<...>>> (leafdata,rootdata,...)
MPI_Isend(leafdata,...)
MPI_Irecv(rbuf,...)

Figure 7: Code paths of `PetscSFReduce` in tests sf_unpack (left) and sf_scatter (right)

communication is done through the `Scatter` kernel, which directly works on rootdata and leafdata. The remote communcation is done through the `Unpack` kernel, which works on rootdata and the receive buffer `rbuf`. The two kernels are executed in the default stream one after another so we are not concerned with data-race in reduction. Also note that `Scatter` is called between `MPI_Irecv` and `MPI_Waitall`, so that local communication could be overlapped with remote communication.

For fair comparision, we modified sf_pingpong to let it use one set of root/leaf data (the code is equal to replacing `MPI_SUM` in Listing 2 with `MPI_REPLACE`) and called it *sf_newpingpong*. We tested sf_newpingpong, sf_unpack and sf_scatter and have their results in Table 3. We have these observations:

1. Comparing the results of sf_pingpoing in Table 2 (columns labeled with Opt-B) and the results of sf_newpingpong in Table 3, we can see they are very close except for the inter-socket and inter-node tests with large messages. For example, in the inter-node 4MB message size tests, sf_newpingpong is about 13% faster than sf_pingpoing. This implies caching did take a role in these cases. Further investigation is out of scope of this report.

2. In these tests roots and leaves are dense such that the `Unpack` and `Scatter` kernels are basically a vector addition. Using the GPU memory bandwidth 900GB/s given in Figure 1, a rough estimation of kernels `Unpack` and `Scatter`'s execution time with 4MB messages size is 4MB*2÷900GB/s = 9.3µs, including both read and write. Let's denote sf_newpingpong's latency as l, and kernel launch time and execution time for kernel K as $T_l(K)$ and $T_e(K)$ respectively. Then sf_unpack's latency $l_{unpack} = l + T_l(Unpack) + T_e(Unpack)$. If we deem $T_e(Unpack) = 0$ at 8 bytes (i.e, one double), then we can easily get kernel launch time $T_l(Unpack) = l_{unpack} - l = 12\mu s$. Since $T_e(Scatter) < l$ in all cases of Table 3, local communication should be fully overlapped with remote communication, such that sf_scatter's latency $l_{scatter} = l_{unpack} = l + T_l(Unpack) + T_e(Unpack)$. We can clearly observe $l_{scatter} = l_{unpack}$ for messages from 8B to 2MB. Data for message size 4MB is an outlier. We guess that is because the local communication (i.e., the Scatter kernel) and the remote communication interfere at the memory system, which makes $l_{scatter}$ longer than l_{unpack}. Figure 8 shows the timeline of sf_scatter on rank 0 with message size 4MB using the Nvidia profiling tool nvprof. We can clearly see execution of the

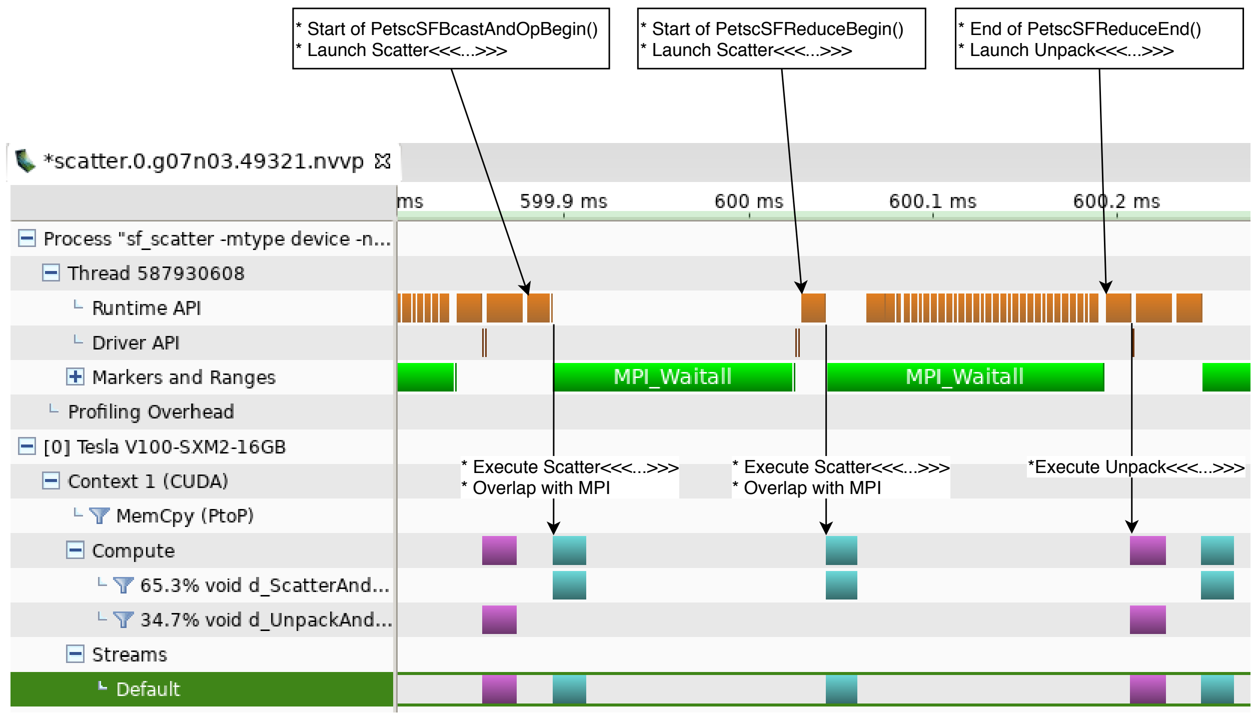

* Start of PetscSFBcastAndOpBegin()
* Launch Scatter<<<...>>>
* Start of PetscSFReduceBegin()
* Launch Scatter<<<...>>>
* End of PetscSFReduceEnd()
* Launch Unpack<<<...>>>
*scatter.0.g07n03.49321.nvvp
ms
599.9 ms
600 ms
600.1 ms
600.2 ms
Process "sf_scatter -mtype device -n...
Thread 587930608
Runtime API
Driver API
Markers and Ranges
MPI_Waitall
MPI_Waitall
Profiling Overhead
[0] Tesla V100-SXM2-16GB
Context 1 (CUDA)
* Execute Scatter<<<...>>>
* Overlap with MPI
* Execute Scatter<<<...>>>
* Overlap with MPI
*Execute Unpack<<<...>>>
MemCpy (PtoP)
Compute
65.3% void d_ScatterAnd...
34.7% void d_UnpackAnd...
Streams
Default

Figure 8: Timeline of one iteration of sf_scatter on rank 0 with 4MB messages. Local communication (i.e., the Scatter kernel[3]) is fully hidden by remote communication (i.e., `MPI_Waitall`).

Scatter kernel is overlapped with MPI communication.

Message	Intra-socket(µs)			Inter-socket(µs)			Inter-node(µs)		
size (bytes)	newpingpong	unpack	scatter	newpingpong	unpack	scatter	newpingpong	unpack	scatter
8	24.3	35.9	35.8	25.4	37.6	37.8	12.2	22.9	23.0
16	24.2	35.7	35.6	25.5	37.5	37.6	11.5	22.6	22.6
32	24.1	35.8	35.8	25.4	37.5	37.8	11.6	22.6	22.8
64	24.2	35.8	35.8	25.4	37.6	37.8	11.6	22.6	22.6
128	24.1	35.7	35.6	25.4	37.5	37.6	11.7	22.8	22.6
256	24.2	35.8	35.8	25.5	37.6	37.8	11.7	22.7	22.7
512	24.2	35.7	35.8	25.4	37.6	37.9	11.8	22.8	23.2
1K	24.2	35.7	35.6	25.4	37.6	37.7	11.9	23.0	22.9
2K	24.2	35.6	35.8	25.4	37.6	37.8	12.5	23.3	23.5
4K	24.1	35.7	35.8	25.4	37.6	37.7	12.9	24.0	23.9
8K	24.0	35.7	35.6	25.6	37.6	37.6	13.8	24.7	25.0
16K	24.0	35.7	35.8	25.6	37.6	37.8	15.0	25.9	25.9
32K	24.1	35.7	35.7	25.7	37.6	37.5	17.2	28.1	28.1
64K	24.7	36.3	36.2	26.3	37.9	38.1	19.8	31.1	31.1
128K	25.9	37.4	37.4	27.7	39.5	39.7	25.5	36.8	36.9
256K	28.5	40.3	40.4	30.7	42.7	42.9	36.2	47.5	47.5
512K	34.2	46.7	46.7	36.9	49.8	49.7	57.5	69.6	69.3
1M	45.3	58.0	58.1	49.3	62.4	62.5	106.5	115.9	115.9
2M	67.6	81.2	81.2	74.0	88.0	88.0	197.5	210.7	210.9
4M	112.2	138.8	140.5	123.5	153.4	160.8	382.7	415.7	427.1

Table 3: One-way latency for the three tests: sf_newpingpong, sf_unpack and sf_scatter

[3]The actual kernel names are `d_ScatterAndXxx`, `d_UnpackAndXxx` as shown by nvprof. For bravity, we just call them `Scatter`

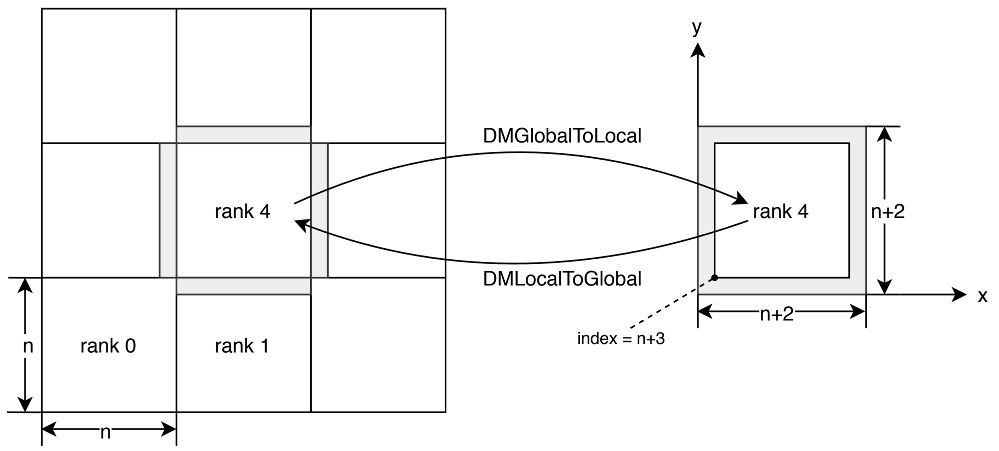


Figure 9: A DM created by DMDACreate2d on nine processors (left) and a local vector on rank 4 (right). Shadowed areas are ghost points.

5.3 PetscSF in regular neighborhood communication

In this section we evaluate PetscSF with a five-point stencil code featuring regular neighborhood communication. We leverage PETSc's `DMDACreate2d` to construct a two-dimensional grid (DM), and then do communication between global vectors and local vectors created with this DM. To be simple, the code creating the DM and the vectors is like this:

```
    bt   = DM_BOUNDARY_PERIODIC;
    ierr = DMDACreate2d(comm,bt,bt,DMDA_STENCIL_STAR,3*n,3*n,3,3,1,1,0,0,&da);CHKERRQ(ierr);
    ierr = DMCreateGlobalVector(da,&g);CHKERRQ(ierr);
    ierr = DMCreateLocalVector(da,&l);CHKERRQ(ierr);
```

Here, we create a 3×3 processor grid, set stencil type to `DMDA_STENCIL_STAR`, stencil width to 1, boundary type to `DM_BOUNDARY_PERIODIC` and let every process have a square subgrid of size `n`×`n`. The DM is shown in the left of Figure 9. With this setting, each MPI rank will have four neighbors and communicate with them with the same amount of data.

In PETSc, global vectors on this grid have a local size of n^2 and elements of the vectors are consecutively stored on each process. Local vectors have a size of $(n+2)^2$, including a halo region. `DMGlobalToLocal`, internally implemented by `PetscSFBcast`, copies local part of a global vector to the interior part of a local vector on each rank, and also copies ghost points received from neighbors to the halo region of the local vector, shown in the right of Figure 9. In each MPI rank's view, copying the interior region is the local communication, and send/receiving ghost points is the remote communication. Each process has to pack four faces of its subgrid into a send buffer and send out to its four neighbors, and finally unpack ghost points from its receive buffer. To copy local vectors to global vectors, one uses `DMLocalToGlobal`, which simply reverses the process above and is implemented by `PetscSFReduce`.

We can easily see local indices of global vectors are contiguously running from 0 to $n^2 - 1$. However, indices of ghost points as a whole, or indices of points in the interior region of a local vector, are not contiguous. Since no hints are given to PetscSF that these indices are incidental to a regular 2D grid, a naive implementation would copy the indices to GPU and resort to indirections like `buf[i] = x[idx[i]]` to do the copying. Instead, our optimized PetscSF uses index analysis to see if indices associated with a destination rank can be arranged in a 3D subgrid. Suppose we have a 3D gird of size `[X,Y,Z]` with nodes sequentially numbered in the *x, y, z* order, and within it there is a subgrid of size `[dx,dy,dz]` with index of the first node being `start`. Then indices of the subgrid can be enumerated with `start+X*Y*k+X*j+i`, for `(i,j,k)` in `(0≤i<dx,0≤j<dy,0≤k<dz)`. By this token, the interior region of a local vector on this DM can be described as a subgrid of size `[n,n,1]` in a grid of size `[n+2,n+2,1]` with a start index `n+3`. Each face of the halo region can also be described similarily. With this abstraction, we only need to copy these grid parameters to GPU and then be able to easily calculate indices there.

or `Unpack` in this report.

```
for (i=0; i<niter; i++) {
  ierr = DMGlobalToLocalBegin(da,g,INSERT_VALUES,l);CHKERRQ(ierr);
  ierr = DMGlobalToLocalEnd(da,g,INSERT_VALUES,l);CHKERRQ(ierr);
  ierr = DMLocalToGlobalBegin(da,l,ADD_VALUES,g);CHKERRQ(ierr);
  ierr = DMLocalToGlobalEnd(da,l,ADD_VALUES,g);CHKERRQ(ierr);
}
```

Listing 3: sf_dmda benchmark loop

Since indices of ghost points are not contiguous, PetscSF has to allocate separate send/recv buffers and call pack/unpack kernels, rendering a code path very similar to Figure 4(b), except in the current case a Scatter kernel is launched after `MPI_Irecv()` to do local communication. We perform back-and-forth communication between a global vector and a local vector using code in Listing 3.

Note that in `DMLocalToGlobal` we use `ADD_VALUES` instead of `INSERT_VALUES` since points along subgrid boundaries are reduced with ghost points received from their neighbors. Using `ADD_VALUES` makes more sense here. The consequence is PetscSF has to handle the potential data races in the Unpack kernel. We tested the code on Summit with two configurations. One had nine compute nodes and one MPI rank per node. Since there was only inter-node communication, ideally all ranks should run uniformly. The other had three compute nodes and three MPI ranks per node. MPI ranks were distributed in a packed manner such that ranks 0, 1, 2 were on node 0, ranks 3, 4, 5 were on node 1, and so on so forth. Even more, we placed each group of three ranks on one socket of a node. Looking at Figure 9, we know that every rank did intra-socket communication with its eastern/western neighbors, and did inter-node communication with its southern/northern neighbors. However, all ranks had even work and communication. Similar to the ping-pong test, we measured average one-way latency of the communication, which is shown in Table 4.

n	Message size (bytes)	Latency(µs)	
		Nine nodes	Three nodes
4	32	45.6	75.7
8	64	44.8	75.6
16	128	45.5	75.7
32	256	45.5	75.8
64	512	45.0	75.8
128	1K	46.0	75.9
256	2K	46.3	75.9
512	4K	47.1	76.0
1024	8K	57.1	83.0
2048	16K	139.9	139.0
4096	32K	499.9	498.3

Table 4: One-way latency for the sf_dmda test, where n is the subgrid size, and message size = 8n, which is the size of messages between two neighbors.

We can see from the table for small messages ($n \leq 512$) the latency is almost the same, which indicates MPI latency and cuda runtime overhead dominate. Since intra-socket ping-pong latency is longer than the inter-node one, the three-node configuration has longer latency than the nine-node configuration. Figure 10 shows profiling result one rank 0 with the nine-node configuration. We can see MPI communication time is longer than the Scatter kernel execution time, and the Pack/Unpack kernel launch time is prominent. In contrast, with bigger n, kernel Scatter's execution time, which is proportional to n^2, out-weights all others such that three nodes have same execution time as nine nodes. We can easily see it from profiling result with n=4096 in Figure 11.

5.4 PetscSF in irregular neighborhood communication

We now turn attention to irregular communications. To study this problem, we use PETSc's sparse matrix-vector multiplication (SpMV) routine `MatMult(mat,x,y)`, which calculates `y=mat*x`. In PETSc, mat is distributed by row and vectors x and y are also distributed accordingly. On each process, the local matrix

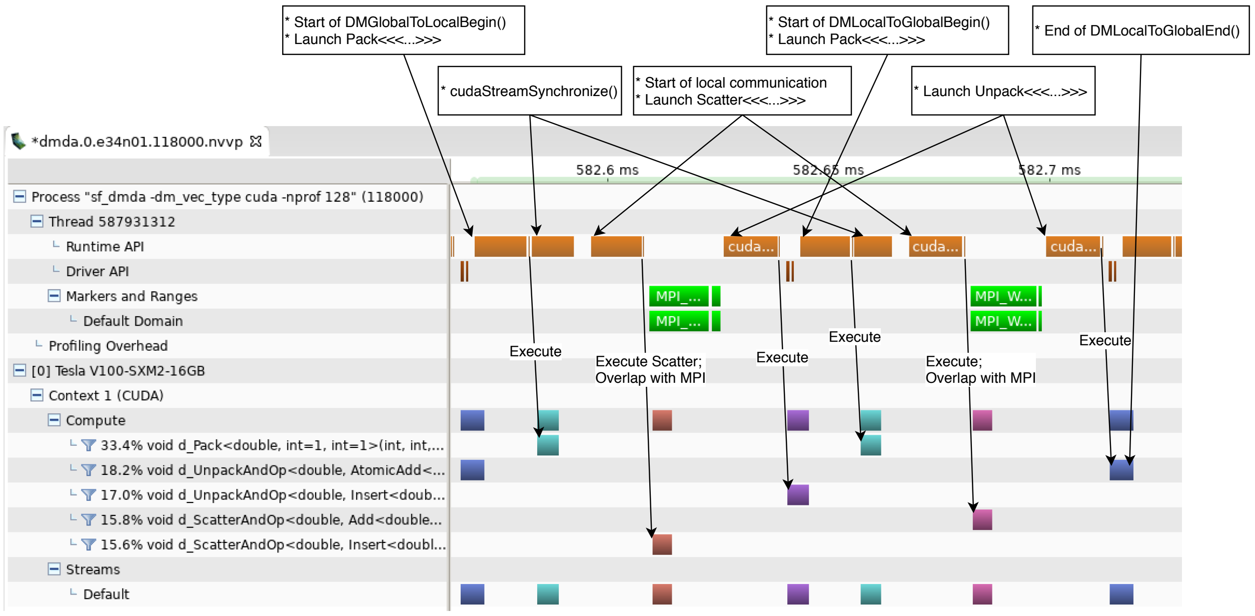

* Start of DMGlobalToLocalBegin()
* Launch Pack<<<...>>>
* Start of DMLocalToGlobalBegin()
* Launch Pack<<<...>>>
* End of DMLocalToGlobalEnd()
* cudaStreamSynchronize()
* Start of local communication
* Launch Scatter<<<...>>>
* Launch Unpack<<<...>>>
*dmda.0.e34n01.118000.nvvp
582.6 ms
582.65 ms
582.7 ms
Process "sf_dmda -dm_vec_type cuda -nprof 128" (118000)
Thread 587931312
Runtime API
Driver API
Markers and Ranges
Default Domain
Profiling Overhead
[0] Tesla V100-SXM2-16GB
Context 1 (CUDA)
Compute
33.4% void d_Pack<double, int=1, int=1>(int, int,...
18.2% void d_UnpackAndOp<double, AtomicAdd<...
17.0% void d_UnpackAndOp<double, Insert<doub...
15.8% void d_ScatterAndOp<double, Add<double...
15.6% void d_ScatterAndOp<double, Insert<doubl...
Streams
Default
cuda...
MPI_...
MPI_W...
Execute
Execute Scatter; Overlap with MPI
Execute; Overlap with MPI

Figure 10: Timeline of one iteration of sf_dmda on rank 0 with nine nodes and n=128

```
for (i=0; i<niter; i++) {
  ierr = VecScatterBegin(Mvctx,x,lvec,INSERT_VALUES,SCATTER_FORWARD);CHKERRQ(ierr);
  ierr = MatMult(A,x,y);CHKERRQ(ierr); /* overlapped computation: y = Ax */
  ierr = VecScatterEnd(Mvctx,x,lvec,INSERT_VALUES,SCATTER_FORWARD);CHKERRQ(ierr);
  ierr = MatMultAdd(B,lvec,y,y);CHKERRQ(ierr); /* y += B*lvec */
}
```

Listing 4: MatMult benchmark loop

is split into a diagonal submatrix `A` and an off-diagonal submatrix `B`. Multiplication `Ax` only needs to access local entries of x and does not need communication, while multiplication `Bx` needs to access remote entries of x and requires communication. The communication is done by VecScatter, implemented in PetscSFBcast. In MatMult implementation, PETSc allocates a local vector *lvec* working as SF leaves on each process to store remote entries of x. Without going to too many details, we have these statements: 1) The leaves are contiguous such that PetscSF can directly use leafdata (i.e., data array of `lvec`) as leaf buffer in `PetscSFBcast`, without resorting to an unpack kernel; 2) Since the matrix is sparse, each rank only needs to send out some entries of vector `x` (i.e., the roots). Therefore roots are generaly not contiguous and we need a pack kernel; 3) There is no local communication; 4) The local computation, i.e., `Ax`, could be overlapped with the communication. With that, we have this classical `MatMult(mat,x,y)` implementation in PETSc, shown as the loop body in Listing 4, whose diagram is shown in Figure 12(a).

Looking at Figure 12(a), we can see the `cudsStreamSynchronize()` in `VecScatterBegin()` is only to ensure `sbuf`, the output of kernel Pack, is ready for use in `MPI_Isend()`. However, it accidently blocks launch of `y = Ax`, which is done through a cuSPARSE kernel. In other words, the launch cost of `y = Ax` could not be hidden. A remedy is to use CUDA events and re-arrange `VecScatterBegin/End()` as shown in Figure 12(b). There we record a CUDA event right after Pack and move `MPI_Isend()` from `VecScatterBegin()` to `VecScatterEnd()`. The event is synchronized before `MPI_Isend()` so that MPI won't send out wrong data. Note that the `B*lvec` in Figure 12(b) only depends on the communication results and does not depend on `y = Ax`. However the algorithm forces `y += B*lvec` to be executed after `y = Ax`. We can decouple this dependancy with help of a temporary vector `z`. In Figure 12(c), We launch `z = B*lvec` on a new stream s, and then launch kernel `y += z` on the default stream to add the partial result to `y`. We use CUDA events to build the dependency between the two kernels on different streams. As long as the communication finishes before kernel `y = Ax`, kernel `z = B*lvec` has the potential to run concurrently with `y = Ax`. Since `y = Ax`

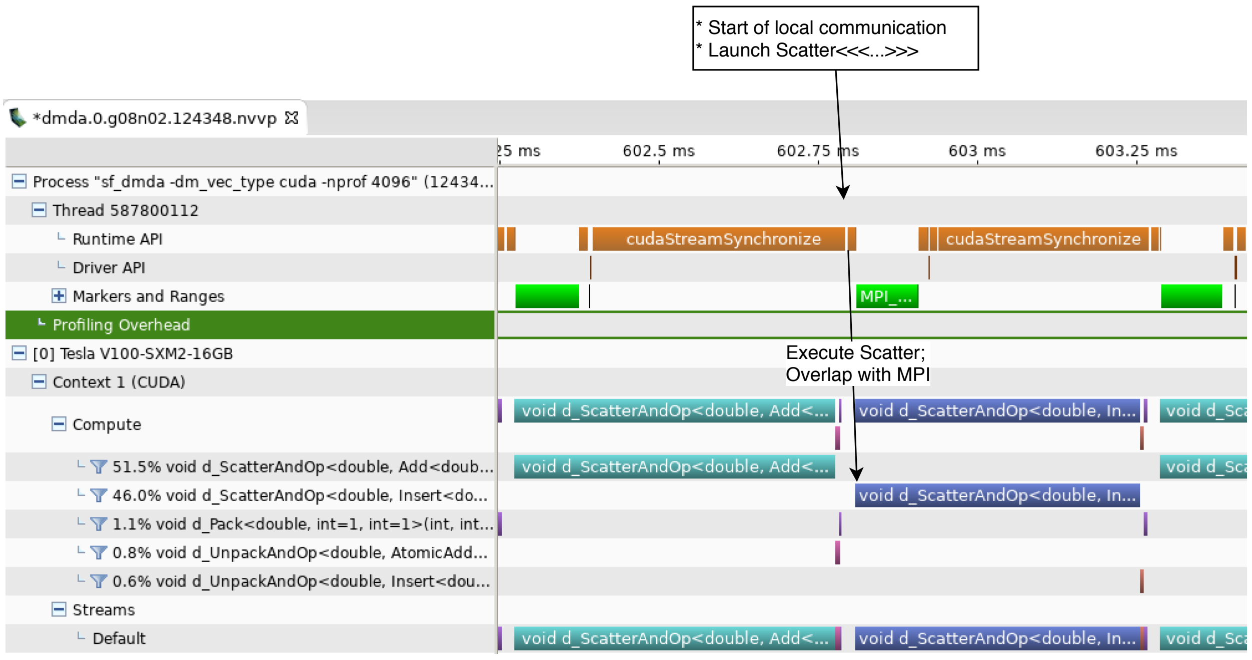

* Start of local communication
* Launch Scatter<<<...>>>
*dmda.0.g08n02.124348.nvvp
25 ms
602.5 ms
602.75 ms
603 ms
603.25 ms
Process "sf_dmda -dm_vec_type cuda -nprof 4096" (12434...
Thread 587800112
Runtime API
cudaStreamSynchronize
cudaStreamSynchronize
Driver API
Markers and Ranges
MPI_...
Profiling Overhead
[0] Tesla V100-SXM2-16GB
Execute Scatter;
Overlap with MPI
Context 1 (CUDA)
Compute
void d_ScatterAndOp<double, Add<...
void d_ScatterAndOp<double, In...
51.5% void d_ScatterAndOp<double, Add<doub...
46.0% void d_ScatterAndOp<double, Insert<do...
1.1% void d_Pack<double, int=1, int=1>(int, int...
0.8% void d_UnpackAndOp<double, AtomicAdd...
0.6% void d_UnpackAndOp<double, Insert<dou...
Streams
Default

Figure 11: Timeline of one iteration of sf_dmda on rank 0 with n=4096

and `y += z` are both lanuched on the default stream, their dependency is automatically maintained. Note both Figure 12(b) and (c) assume the computation sandwiched between `VecScatterBegin/End()` won't block the CPU thread so that `MPI_Isend()` can be posted as soon as possible. Therefore, without changes, they could not be directly applied to CPU codes. They are currently not in PETSc releases.

We tested these three MatMult implementations with a sparse matrix (HV15R) from the Florida sparse matrix collection [5]. Size of the matrix is 2,017,169 and it has 283,073,458 nonzeros. Tested on one node of Summit with six GPUs and six MPI ranks, the execution time was 918.9µs, 902.2µs and 904.6µs for the three MatMult implementations respectively. We can see MatMult(b) was 16.7µs faster than MatMult(a), which is close to a kernel lanuch time, indicating the launch time of `y = Ax` is effectively hidden in MatMult(b). However, MatMult(c) did not show advantage over MatMult(b). We profiled them and show their timeline on rank 3 in Figures 13 and 14. We can see SpMVs (i.e., csrMv_kernel) with the diagonal block and the off-diagonal block did overlap as we expected. But we also found with overlapping the kernel's execution time was a little longer than the non-overlapped one's, offsetting any gains gotten from overlapping. Further investigation reveals the reason. In CUDA, concurrent kernel execution have some requirements. Firstly, there must be enough resources to accommodate multiple kernels. None kernel can have enough resident thread blocks to fill up the GPU. Secondly, a streaming multiprocessor (SM) can only host thread blocks from the same kernel. In our test, kernel `y = Ax` had a grid of size (42025,1,1) and a thread block of size (16,8,1), while kernel `z = B*lvec` had a grid of size (10507,1,1) and a thread block of size (4,32,1) (note these kernel launch parameters were controlled by the cuSPARSE library). However a Nvidia V100 GPU has 80 SMs and each SM can only have maximal 32 resident thread blocks, giving total 2560 resident thread blocks per GPU. Therefore, we only saw overlap at the end of the first kernel, presumbly that was the time when some SMs were draining out from the first kernel and became available for the second one. Additionally, since SpMV is a bandwidth-bound kernel, running two SpMVs concurrently only limits bandwidth available to each and hurts their performance. We predict small compute-bound kernels would benefit from the design in Figure 12(c).

6 Discussion and Conclusion

Asynchronous computation on GPUs brings new challenges to MPI communication. In a communication module's view, it has to synchronize the device properly, and also provide efficient pack/unpack kernels. In this report we analized and evaluated PetscSF, the communication module in PETSc, on Summit GPUs.

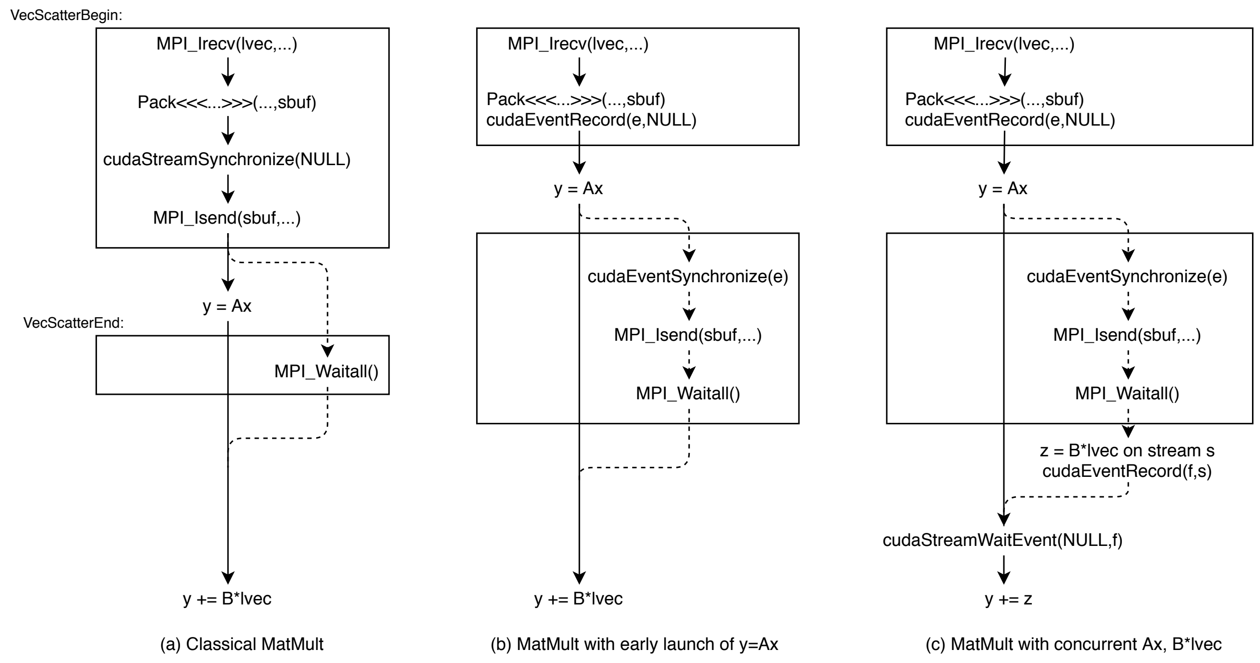


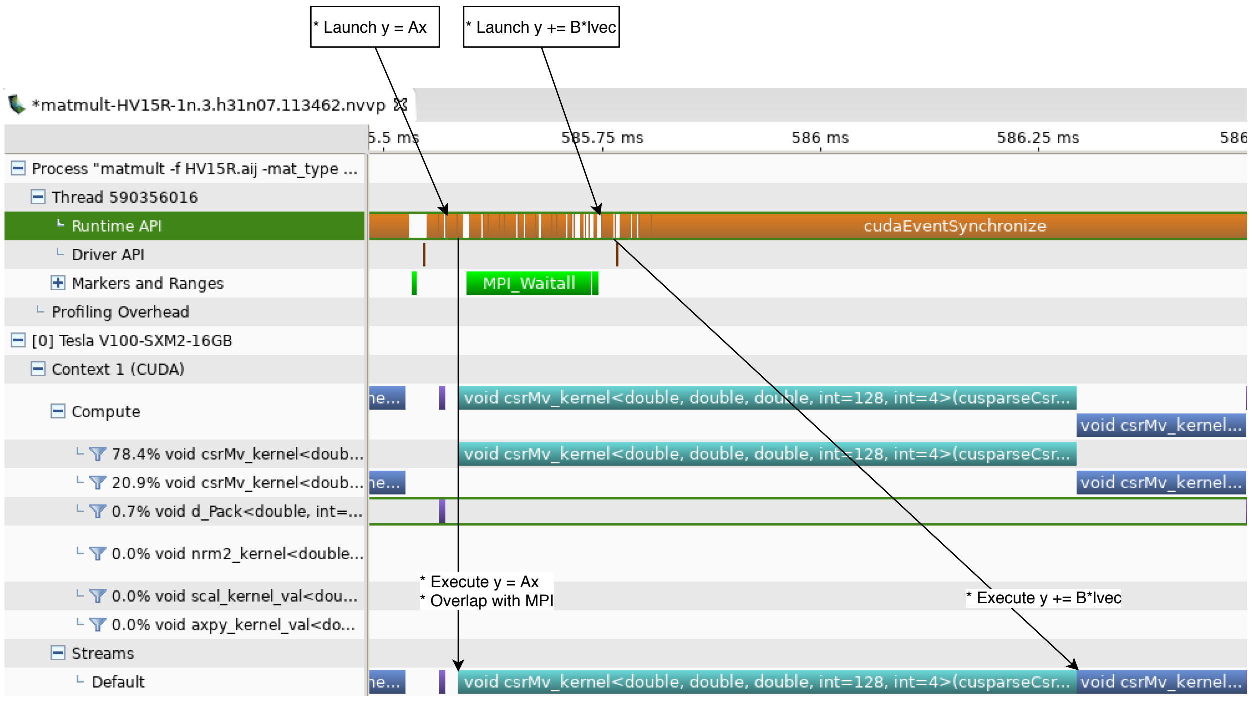
Figure 12: Various MatMult implementations. Boxes at the top are VecScatterBegin, at the bottom are VecScatterEnd. In each diagram, vertically parallel solid and dashed lines indicate overlapped computation and communication.

We first measured GPU communication latencies with an MPI ping-pong benchmark, which does not have any synchronizations or pack/pack kernels, and therefore whose performance can be seen as the upper bound for that of PetscSF. Then in Section 4, we analyzed three synchronization models in PetscSF under different assumptions. In Section 5.1 we evaluated a ping-pong test (sf_pingpong) written in PetscSF under those models. From the test results, we can know costs of various CUDA synchronizations. We also found the extra overhead brought by PetscSF can be as low as 1µs. In Section 5.2 we introduced two new bencharmks (sf_unpack and sf_scatter) that have unpacking and local communication. From the result we can get kernel launch cost and also see the effect of overlapped local communication and remote communication. In Section 5.3 we introduced index optimizations in Pack/Unpack kernels with regular neighborhood communication. Generally speaking, in this communiation pattern, with small (regular) domains, remote communication is the bottleneck, and with big domains, local communication is the bottleneck. Finally in Section 5.4 we evaluated PetscSF irregular neighborhood communication with a sparse matrix-vector multiplication kernel.

PetscSF's default synchronization model assumes that the input and output data is on the default stream, so that we can avoid the `cudaDeviceSynchronize()` and `cudaStreamSynchronize()` calls before the Pack kernel and after the Unpack kernel, which translate into a savings of 9µs. The remaining synchronization is a `cudaStreamSynchronize()` call, which costs about 4µs and is denoted below by $T_{StreamSync}$. With that, we can model total time T of a general split-phase communication pattern `PetscSFXxxBegin(); UserKernel(); PetscSFXxxEnd()` as follows:

$$T = T(Pack) + T_{StreamSync} + max\begin{Bmatrix} l_{MPI} \\ T(Scatter) + T(UserKernel) \end{Bmatrix} + T(Unpack)$$

Here $T(K)$ represents the time of kernel K, including launch time and execution time. l_{MPI} is the MPI latency (i.e., time to communicate data). Pack, Unpack and Scatter only involve simple operations on elements (i.e., roots or leaves) and are usually bandwidth bound. One can easiy model their execution time as $\frac{Memory\ size}{Bandwidth}$, where memory size is total size of data a kernel accesses, including elements and their indices if elements are irregular. Bandwidth is the *effective* bandwidth, which depends on access patterns, such as contiguous access, stridded access or random access. One can write simple kernels to measure them. For point-to-point communication invovling only a pair of ranks, it is easy to model l_{MPI} as we did in Section 3. For communication invovling multiple senders and recievers sharing communication links, we do not have



* Launch y = Ax
* Launch y += B*lvec
*matmult-HV15R-1n.3.h31n07.113462.nvvp
5.5 ms
585.75 ms
586 ms
586.25 ms
586
Process "matmult -f HV15R.aij -mat_type ...
Thread 590356016
Runtime API
cudaEventSynchronize
Driver API
Markers and Ranges
MPI_Waitall
Profiling Overhead
[0] Tesla V100-SXM2-16GB
Context 1 (CUDA)
Compute
void csrMv_kernel<double, double, double, int=128, int=4>(cusparseCsr...
void csrMv_kernel...
78.4% void csrMv_kernel<doub...
20.9% void csrMv_kernel<doub...
0.7% void d_Pack<double, int=...
0.0% void nrm2_kernel<double...
0.0% void scal_kernel_val<dou...
0.0% void axpy_kernel_val<do...
* Execute y = Ax
* Overlap with MPI
* Execute y += B*lvec
Streams
Default

Figure 13: Timeline of MatMult with early launch of `y = Ax`. Note launch of kernel `y = Ax` does not need to wait for finish of the Pack kernel, but kernel `y += B*lvec` can not start until kernel `y = Ax` is completed.

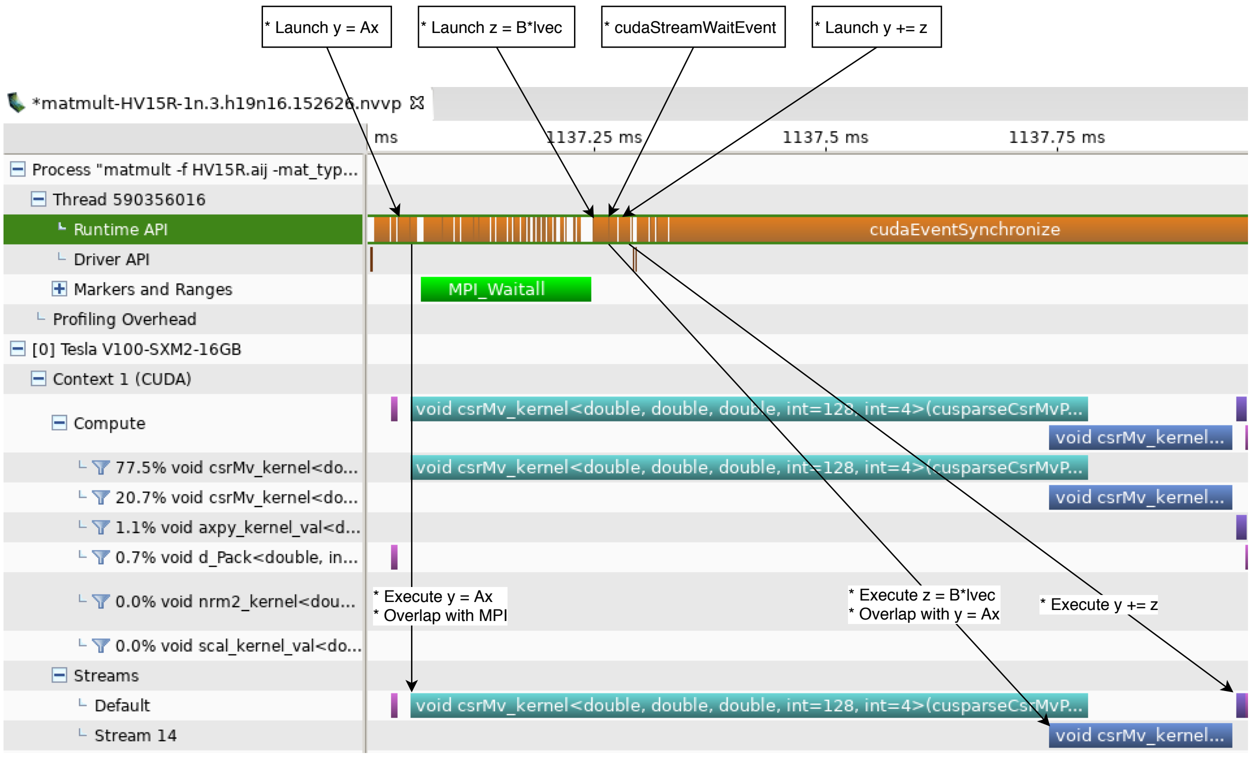
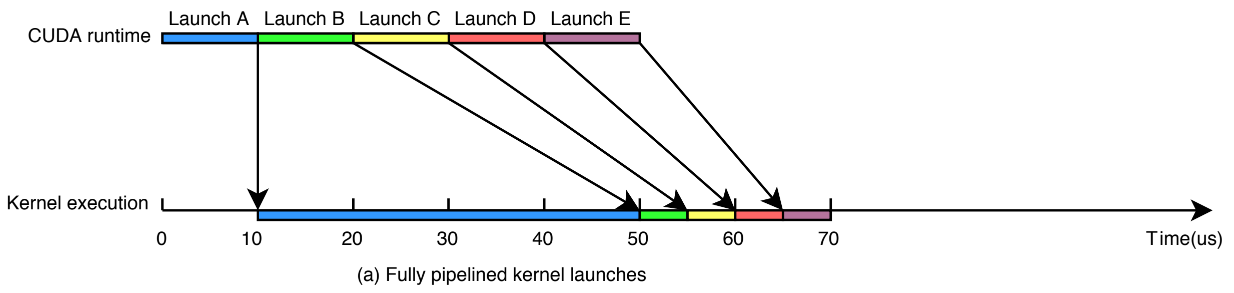


Figure 14: Timeline of MatMult with concurrent kernels `Ax` and `B*lvec`.



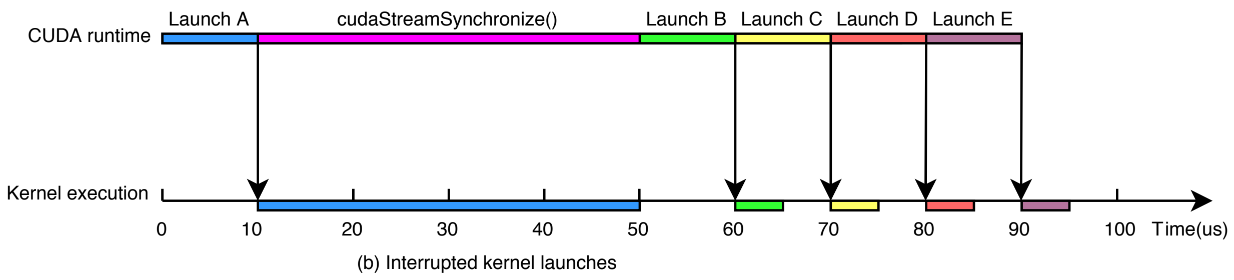


Figure 15: Effect of synchronization in kernel launches. Without synchronization, the five kernels from A to E take 70µs to finish. With `cudaStreamSynchronize()`, they take 95µs.

a reliable model. LogGP[1] might be an alternative, but we do not know how to validate it on Summit. We leave it as an open question.

$T_{StreamSync}$ of 4µs seems not high, however one should be aware that the synchronization may block further kernel launches in the pipeline, resulting in poor launch cost hiding, which could bring a cost much higher than `cudaStreamSynchronize()` itself. For example, let's assume we have five kernels A, B, C, D, E, and their execution time is 40µs, 5µs, 5µs, 5µs, 5µs respectively. Let's further assume a kernel launch costs 10µs. If kernel launches are fully pipelined, the total time for these five kernels is 70µs, as shown in Figure 15(a). However, if there is a `cudaStreamSynchronize()` after the first kernel launch, then the remaining launches will be stalled and the total time will be 95µs, as shown in 15(b).

In Section 5.4, we introduced an approach that uses CUDA events to avoid `cudaStreamSynchronize()`, but this approach requires operations in between `VecScatterBegin()` and `VecScetterEnd()` to be asynchronous, and there should not be too many operations since we need to issue `MPI_Isend()` as soon as possible. The ideal solution is to make MPI routines CUDA stream aware, such that a non-blocking MPI call works as an asynchronous kernel launch on a given stream, and an `MPI_Wait()` works as a `cudaEventSynchronize()`. In this way, MPI calls become a regular node in the dependance graph of a computation, instead of a barrier in it.

Acknowledgments

The authors were supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract DE-AC02-06CH11357.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the logp model for parallel computation. Journal of parallel and distributed computing, 44(1):71–79, 1997.

[2] Summit User Guide Website Authors. Summit User Guide. https://www.olcf.ornl.gov/for-users/system-user-guides/summit/summit-user-guide/. Accessed: 2019-08.

[3] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc users manual: Revision 3.12. Technical Report ANL-95/12 - Rev 3.12, Argonne National Laboratory, 2019.

[4] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc Web page, 2019.

[5] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS), 38(1):1–25, 2011.

[6] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. Van Essen. Aluminum: An asynchronous, gpu-aware communication library optimized for large-scale training of deep neural networks on hpc systems. In 2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC), pages 1–13, Nov 2018.

[7] Richard Trans Mills Hannah Morgan and Barry Smith. Evaluation of petsc on a heterogeneous architecture,the olcf summit system part i: Vector node performance. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 2019.

[8] Judy Hill. Summit at the oak ridge leadership computing facility, 2018.

[9] Kawthar Shafie Khorassani, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. Performance evaluation of mpi libraries on gpu-enabled openpower architectures: Early experiences. In International Conference on High Performance Computing, pages 361–378. Springer, 2019.

[10] DK Panda et al. Osu microbenchmarks v5.6.2. URL http://mvapich.cse.ohio-state.edu/benchmarks/, 2019.

[11] Exascale Support Team. Exascale web page, 2019.