The PetscSF Scalable Communication Layer

Junchao Zhang*, Jed Brown†, Satish Balay*, Jacob Faibussowitsch‡, Matthew Knepley§, Oana Marin*, Richard Tran Mills*, Todd Munson*, Barry F. Smith¶, Stefano Zampini‖
*Argonne National Laboratory, Lemont, IL 60439 USA
†University of Colorado Boulder, Boulder, CO 80302 USA
‡University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA
§University at Buffalo, Buffalo, NY 14260 USA
¶Argonne Associate of Global Empire, LLC, Argonne National Laboratory, Lemont, IL 60439 USA
‖King Abdullah University of Science and Technology, Thuwal, Saudi Arabia

Abstract—PetscSF, the communication component of the Portable, Extensible Toolkit for Scientific Computation (PETSc), is designed to provide PETSc's communication infrastructure suitable for exascale computers that utilize GPUs and other accelerators. PetscSF provides a simple application programming interface (API) for managing common communication patterns in scientific computations by using a star-forest graph representation. PetscSF supports several implementations based on MPI and NVSHMEM, whose selection is based on the characteristics of the application or the target architecture. An efficient and portable model for network and intra-node communication is essential for implementing large-scale applications. The Message Passing Interface, which has been the de facto standard for distributed memory systems, has developed into a large complex API that does not yet provide high performance on the emerging heterogeneous CPU-GPU-based exascale systems. In this paper, we discuss the design of PetscSF, how it can overcome some difficulties of working directly with MPI on GPUs, and we demonstrate its performance, scalability, and novel features.

Index Terms—Communication, GPU, extreme-scale, MPI, PETSc

1 Introduction

Distributed memory computation is practical and scalable; all high-end supercomputers today are distributed memory systems. Orchestrating the communication between processes with separate memory spaces is an essential part of programming such systems. Programmers need interprocess communication to coordinate the processes' work, distribute data, manage data dependencies, and balance workloads. The Message Passing Interface (MPI) is the de facto standard for communication on distributed memory systems. Parallel applications and libraries in scientific computing are predominantly programmed in MPI. However, writing communication code directly with MPI primitives, especially in applications with irregular communication patterns, is difficult, time-consuming, and not the primary interest of application developers. Also, MPI has not yet fully adjusted to heterogeneous CPU-GPU-based systems, where avoiding expensive CPU-GPU synchronizations is crucial to achieving high performance.

Higher-level communication libraries tailored for specific families of applications can significantly reduce the programming burden from the direct use of MPI. Requirements for such libraries include an easy-to-understand communication model, scalability, and efficient implementations while supporting a wide range of communication scenarios. When programming directly with MPI primitives, one faces a vast number of options, for example, MPI two-sided vs. one-sided communication, individual sends and receives, neighborhood operations, persistent or non-persistent interfaces, and more.

In [1] we discuss the plans and progress in adapting the Portable, Extensible Toolkit for Scientific Computation and Toolkit for Advanced Optimization [2] (PETSc) to CPU-GPU systems. This paper focuses specifically on the plans for managing the network and intra-node communication within PETSc. PETSc has traditionally used MPI calls directly in the source code. PetscSF is the communication component we are implementing to gradually remove these direct MPI calls. Like all PETSc components, PetscSF is designed to be used both within the PETSc libraries and also by PETSc users. It is not a drop-in replacement for MPI. Though we focus our discussion on GPUs in this paper, PetscSF also supports CPU-based systems with high performance. Most of our experimental work has been done on the OLCF IBM/NVIDIA Summit system that serves as a surrogate for the future exascale systems; however, as discussed in [1], the PetscSF design and work is focused on the emerging exascale systems.

PetscSF can do communication on any MPI datatype and supports a rich set of operations. Recently, we consolidated VecScatter (a PETSc module for communication on vectors) and PetscSF by implementing the scatters using PetscSF. We now have a single communication component in PETSc, thus making code maintenance easier. We added various optimizations to PetscSF, provided multiple implementations, and, most importantly, added GPU support. Notably, we added support of NVIDIA NVSHMEM [3] to provide an MPI alternative for communication on CUDA devices. With this, we can avoid device synchronizations needed by MPI and accomplish a distributed fully asynchronous computation, which is key to unleashing the power of

exascale machines where heterogeneous architecture will be the norm.

The paper is organized as follows. Section 2 discusses related work on abstracting communications on distributed memory systems. Section 3 introduces PetscSF. In Section 4, we describe several examples of usage of PetscSF within PETSc itself. In Section 5, we detail the PetscSF implementations and optimizations. Section 6 gives some experimental results and shows PetscSF's performance and scalability. Section 7 concludes the paper with a summary and look to future work.

2 Related Work

Because communication is such an integral part of distributed-memory programming, many regard the communication model *as* the programming model. We compare alternative distributed memory programming models against the predominantly used MPI.

In its original (and most commonly used) form, MPI uses a two-sided communication model, where both the sender and the receiver are explicitly involved in the communication. Programmers directly using MPI must manage these relationships, in addition to allocating staging buffers, determining which data to send, and finally packing and/or unpacking data as needed. These tasks can be difficult when information about which data is shared with which processes is not explicitly known. Significant effort has been made to overcome this drawback with mixed results.

The High Performance Fortran (HPF) [4] project allowed users to write data distribution directives for their arrays and then planned for the compilers to determine the needed communication. HPF failed because compilers, even today, are not powerful enough to do this with indirectly indexed arrays. Several Partitioned Global Address Space (PGAS) languages were developed, such as UPC [5], Co-array Fortran [6], Chapel [7], and OpenSHMEM [8]. They provide users an illusion of shared memory. Users can dereference global pointers, access global arrays, or put/get remote data without the remote side's explicit participation. Motivated by these ideas, MPI added one-sided communication in MPI-2.0 and further enhanced it in MPI-3.0. However, the PGAS model had limited success. Codes using such models are error-prone, since shared-memory programming easily leads to data race conditions and deadlocks. These are difficult to detect, debug, and fix. Without great care, PGAS applications are prone to low performance since programmers can easily write fine-grained communication code, which severely hurts performance on distributed memory systems. Because of this, writing a correct and efficient code that requires communication using PGAS languages is not fundamentally easier than programming in MPI.

While MPI has excellent support for writing libraries and many MPI-based libraries target specific domains, few communication libraries are built using MPI. We surmise the reason for this is that the data to be communicated is usually embedded in the user's data structure, and there is no agreed-upon interface to describe the user's data connectivity graph. Zoltan [9] is one of the few such libraries that is built on MPI. Zoltan is a collection of data management services for parallel, unstructured, adaptive, and dynamic applications. Its unstructured communication service and data migration tools are close to those of PetscSF. Users need to provide pack and unpack callback functions to Zoltan. Not only does PetscSF pack and unpack automatically, it is capable of performing reductions when unpacking. PetscSF supports composing communication graphs, see Section 3.3. The gs library [10] used by Nek5000 [11] gathers and scatters vector entries. It is similar to PETSc's VecScatter, but it is slightly more general and supports communicating multiple data types. DIY [12] is a block-parallel communication library for implementing scalable algorithms that can execute both in-core and out-of-core. Using a block parallelism abstraction, it combines distributed-memory message passing with shared-memory thread parallelism. Complex communication patterns can be built on a graph of blocks, usually coarse-grained, whereas in PetscSF, graph vertices can be as small as a single floating-point number. We are unaware that any of these libraries support GPUs.

Since distributed-memory communication systems are often conflated with programming models, we should clarify that tools such as Kokkos [13] and Raja [14] provide functionality orthogonal to communication systems. For example, in [1], we provide a prototype code that uses PetscSF for the communication and Kokkos as the portable programming model. SYCL [15] automatically manages needed communication between GPU and CPU but does not have general communication abilities. Thus, it also requires a distributed memory communication system.

3 The PetscSF Interface

PetscSF has a simple conceptual model with a small, but powerful, interface. We begin by introducing how to create PetscSF objects and their primary use cases.

3.1 PetscSF Creation

PetscSF supports both structured and irregular communication graphs, with the latter being the focus of this paper. A star forest is used to describe the communication graphs. Stars are simple trees consisting of one root vertex connected to zero or more leaves. The number of leaves is called the *degree* of the root. We also allow leaves without connected roots. These isolated leaves represent holes in the user's data structure that do not participate in the communication. Fig. 1 shows some example stars. A union of disjoint stars is called a star forest (SF). PetscSFs are typically partitioned across multiple processes. In graph terminology, an PetscSF object can be regarded as being similar to a quotient graph [16], which embeds the closely connected subpartitions of a larger graph, which is the abstraction of a domain topological representation.

Following PETSc's object creation convention, one creates a PetscSF with `PetscSFCreate(MPI_Comm comm,PetscSF *sf)`, where `comm` specifies the MPI communicator the PetscSF lives on. One then describes the graph by calling `PetscSFSetGraph()` on each process.

```
typedef struct {PetscInt rank,offset;} PetscSFNode;

PetscErrorCode PetscSFSetGraph(PetscSF sf, PetscInt nroots,
  PetscInt nleaves, const PetscInt *local,
  PetscCopyMode localmode, const PetscSFNode *remote,
  PetscCopyMode remotemode);
```

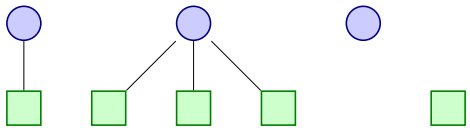

Fig. 1: Examples of stars, the union of which forms a star forest. Root vertices are identified with circles and leaves with rectangles. Note that roots with no leaves are allowed as well as leaves with no root.

This call indicates that this process owns `nroots` roots and `nleaves` *connected* leaves. The roots are numbered from 0 to `nroots-1`. `local[0..nleaves]` contains local indices of those connected leaves. If `NULL` is passed for `local`, the leaf space is contiguous. Addresses of the roots for each leaf are specified by `remote[0..nleaves]`. Leaves can connect to local or remote roots; therefore we use tuples `(rank,offset)` to represent root addresses, where `rank` is the MPI rank of the owner process of a root and `offset` is the index of the root on that process. Fig. 2 shows an PetscSF

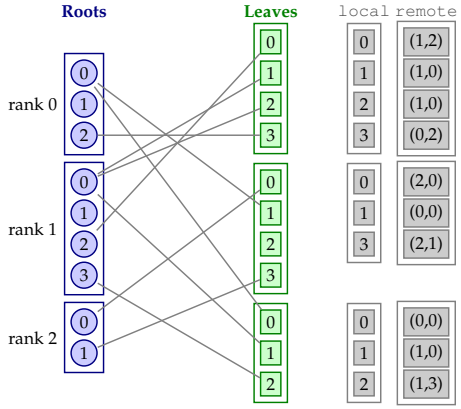

Fig. 2: Distributed star forest partitioned across three processes, with the specification arrays at right. Roots (leaves) on each rank are numbered top-down, with local indices starting from zero. Note the isolated vertices on rank 1.

with data denoted at vertices, together with the `local` and `remote` arguments passed to `PetscSFSetGraph()`. The edges of an PetscSF are specified by the process that owns the leaves; therefore a given root vertex is merely a candidate for *incoming* edges. This one-sided specification is important to encode graphs containing roots with a very high degree, such as globally coupled constraints, in a scalable way. `PetscSFSetUp()` sets up internal data structures for its implementations and checks for all possible optimizations. The setup cost is usually amortized by multiple operations performed on the PetscSF. A PetscSF provides only a template for communication. Once a PetscSF is created, one can instantiate simultaneous communication on it with different data. All PetscSF routines return an error code; we will omit it in their prototypes for brevity.

3.2 PetscSF Communication Operations

In the following, we demonstrate operations that update data on vertices. All operations are split into matching *begin* and *end* phases, taking the same arguments. Users can insert independent computations between the calls to overlap computation and communication. Data buffers must not be altered between the two phases, and the content of the result buffer can be accessed only after the matching End operation has been posted.

(1) Broadcast roots to leaves or reduce leaves to roots

```
PetscSFBcastBegin/End(PetscSF sf, MPI_Datatype unit,
  const void *rootdata, void *leafdata, MPI_Op op);
PetscSFReduceBegin/End(PetscSF sf, MPI_Datatype unit,
  const void *leafdata, void *rootdata, MPI_Op op);
```

These operations are the most used PetscSF operations (see concrete examples in Section 4.1). `unit` is the data type of the tree vertices. Any MPI data type can be used. The pointers `rootdata` and `leafdata` reference the user's data structure organized as arrays of `units`. The term `op` designates an MPI reduction, such as MPI_SUM, which tells PetcSF to add root values to their leaves' value (`SFBcast`) or vice versa (`SFReduce`). When `op` is MPI_REPLACE, the operations overwrite destination with source values.

(2) Gather leaves at roots and scatter back

One may want to gather (in contrast to reduce) leaf values to roots. For that purpose, one needs a new SF by splitting roots in the old SF into new roots as many as their degree and then connecting leaves to the new roots one by one. To build that SF, a difficulty is for leaves to know their new roots' offset at the remote side. We provide a fetch-and-add operation to help this task.

```
PetscSFFetchAndOpBegin/End(PetscSF sf,MPI_Datatype unit,
  void *rootdata,const void *leafdata,void *leafupdate,
  MPI_Op op);
```

With that, one can first do fetch-and-add on integers (`unit` = MPI_INT) with the old SF, with roots initialized to their old offset value and leaves to 1. The operation internally does these: For a root has n leaves, $leaf_{0..n-1}$, whose values are added (`op` = MPI_SUM) to the root in n reductions. When updating using $leaf_i$, it first fetches the current, partially reduced value of the root before adding $leaf_i$'s value. The fetched value, which is exactly the offset of the new root $leaf_i$ is going to be connected to in the new SF, is stored at $leaf_i$'s position in array `leafupdate`. After that, constructing the new SF is straightforward. This set of operations are so useful that we give the new SF a name, *multi-SF*, and provide more user-friendly routines:

```
PetscSFGetMultiSF(PetscSF sf, PetscSF *multi);
PetscSFGatherBegin/End(PetscSF sf,MPI_Datatype unit,
  const void *leafdata,void *multirootdata);
PetscSFScatterBegin/End(PetscSF sf,MPI_Datatype unit,
  const void *multirootdata,void *leafdata);
```

The first routine returns the multi-SF of `sf` while the other two make use of the internal multi-SF representation of `sf`. `SFGather` gathers leaf values from `leafdata` and stores them at `multirootdata`, which is an array containing m `units`, where m is the number of roots of the multi-sf. `SFScatter` reverses the operation of `SFGather`. A typical use of `SFGather/Scatter` in distributed computing is for owner points, working as an arbitrator, to make some non-trivial decision based on data gathered from ghost points and then scatter the decision back.

3.3 PetscSF Composition Operations

One can make new PetscSFs from existing ones, using

(1) Concatenation: Suppose there are two PetscSFs, `A` and `B`, and `A`'s leaves are overlapped with `B`'s roots. One call

```
PetscSFCompose(PetscSF A,PetscSF B,PetscSF *AB);
```

to compose a new PetscSF `AB`, whose roots are `A`'s roots and leaves are `B`'s leaves. A root and a leaf of `AB` are connected if there is a leaf in `A` and a root in `B` that bridge them. Similarly, if `A`'s leaves are overlapped with `B`'s leaves and `B`'s roots all have a degree at most one, then the result of the following composition is also well defined.

```
PetscSFComposeInverse(PetscSF A,PetscSF B,PetscSF* AB);
```

`AB` is a new PetscSF with `A`'s roots and leaves being `B`'s roots and edges built upon a reachability definition similar to that in PetscSFCompose. Such SF concatenations enable users to redistribute or re-order data from existing distributions or orderings.

(2) Embedding: PetscSF allows removal of existing vertices and their associated edges, a common use case in scientific computations, for example, in field segregation in multi-physics application and subgraph extraction. The API

```
PetscSFCreateEmbeddedRootSF(PetscSF sf, PetscInt n,
  const PetscInt selected_roots[], PetscSF *esf);
PetscSFCreateEmbeddedLeafSF(PetscSF sf, PetscInt n,
  const PetscInt selected_leaves[], PetscSF *esf)
```

removes edges from all but the selected roots/leaves without remapping indices and returns a new PetscSF that can be used to perform the subcommunication using the original root/leaf data buffers.

4 Use Cases

Although PetscSF has many uses, we describe here only a small subset of applications of PetscSF to other PETSc operations, namely, parallel matrix and unstructured mesh operations.

4.1 Parallel Matrix Operations

Sparse matrix-vector products (SpMV) The standard parallel sparse matrix implementation in PETSc distributes a matrix by rows. On each MPI rank, a block of consecutive rows makes up a local matrix. The parallel vectors that can be multiplied by the matrix are distributed by rows in a conforming manner. On each rank, PETSc *splits* the local matrix into two blocks: a *diagonal* block A, whose columns match with the vector's rows on the current rank, and an *off-diagonal* block B for the remaining columns. See Fig. 3. A and B are separately encoded in the compressed sparse row (CSR) format with *reduced local* column indices. Then, the SpMV in PETSc is decomposed into $y = Ax + Bx$. Ax depends solely on local vector data, while Bx requires access to remote entries of x that correspond to the nonzero columns of B. PETSc uses a sequential vector *lvec* to hold these needed ghost points entries. The length of *lvec* is equal to the number of nonzero columns of B, and its entries are ordered in their corresponding column order. See Fig. 3.

We build an PetscSF for the communication needed in Bx as described below. On each rank, there are n roots,

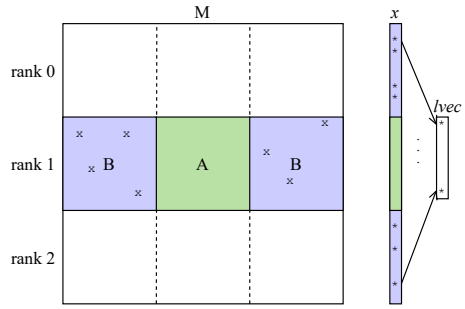

Fig. 3: A PETSc matrix `M` on three ranks. On rank 1 the diagonal block `A` is in green and the off-diagonal block `B` is in blue. The x in `B` denotes nonzeros, while the * in x denotes remote vector entries needed to compute Bx.

where n is the local size of x, and m leaves, where m is the size of vector *lvec*. The leaves have contiguous indices running from 0 to $m-1$, each connected to a root representing a global column index. With the matrix layout info available on each rank, we calculate the owner rank and local index of a global column on its owner and determine the PetscSFNode argument of SFSetGraph() introduced in Section 3.1. The SpMV $y = Mx$ can be implemented in the listing below, which overlaps the local computation $y = Ax$ with the communication.

```
// x_d, lvec_d are data arrays of x, lvec respectively
PetscSFBcastBegin(sf,MPI_DOUBLE,x_d,lvec_d,MPI_REPLACE);
y  = A*x;
PetscSFBcastEnd(sf,MPI_DOUBLE,x_d,lvec_d,MPI_REPLACE);
y += B*lvec;
```

The transpose multiply $y = M^T x$ is implemented by

```
y    = A^T*x;
lvec = B^T*x;
PetscSFReduceBegin(sf,MPI_DOUBLE,lvec_d,y_d,MPI_SUM);
PetscSFReduceEnd(sf,MPI_DOUBLE,lvec_d,y_d,MPI_SUM);
```

Note that $lvec = B^T x$ computes the local contributions to some remote entries of y. These must be added back to their owner rank.

Extracting a submatrix from a sparse matrix The routine `MatCreateSubMatrix(M,ir,ic,c,P)` extracts a parallel submatrix `P` on the same set of ranks as the original matrix `M`. Parallel global index sets, `ir` and `ic`, provide the rows the local rank should obtain and columns that should be in its diagonal block of the submatrix on this rank, respectively. The difficulty is to determine, for each rank, which columns are needed (by any ranks) from the owned block of rows.

Given two SFs, `sfA`, which maps reduced local column indices (leaves) to global columns (roots) of the original matrix `M`, and `sfB`, which maps "owned" columns (leaves) of the submatrix `P` to global columns (roots) of `M`, we determine the retained local columns using the following:

1) SFReduce using `sfB`, results in a distributed array containing the columns (in the form of global indices) of `P` that columns of `M` will be replicated into. Unneeded columns are tagged by a negative index.
2) SFBcast using `sfA` to get these values in the reduced local column space.

The algorithm prepares the requested rows of `M` in *non-split* form and distributes them to their new owners, which can be done using a row-based PetscSF.

4.2 Unstructured Meshes

PetscSF was originally introduced in `DMPlex`, the PETSc class that manages unstructured meshes. PetscSF is one of its central data structures and is heavily used there. Interested readers are referred to [17]–[19]. Here we only provide a glimpse of how we use DMPlex and PetscSF together to *mechanically* build complex, distributed data distributions and communication from simple combination operations.

DMPlex provides unstructured mesh management using a topological abstraction known as a Hasse diagram [20], a directed acyclic graph (DAG) representation of a partially ordered set. The abstract mesh representation consists of *points*, corresponding to vertices, edges, faces, and cells, which are connected by *arrows* indicating the topological adjacency relation. Points from different co-dimensions are indexed uniformly, therefore this model is *dimension-independent*; it can represent meshes with different cell types, including cells of different dimensions.

Parallel topology is defined by a mesh point PetscSF, which connects ghost points (leaves) to owned points (roots). Additionally, PetscSFs are used to represent the communication patterns needed for common mesh operations, such as partitioning and redistribution, ghost exchange, and assembly of discretized functions and operators. To generate these PetscSFs, we use `PetscSection` objects, which maps points to sizes of data of interest related to the points, assuming the data is packed. For example, with an initial mesh point PetscSF, applying a PetscSection mapping mesh points to degrees-of-freedom (dofs) on points generates a new dof-PetscSF that relates dofs on different processes. Another example, with the dof-PetscSF just generated, applying a PetscSection mapping dofs to adjacent dofs produces the PetscSF for the Jacobian, described in detail in [21].

Mesh distribution, in response to a partition, is handled by creating several PetscSF objects representing the steps needed to perform the distribution. Based on the partition, we make a PetscSF whose roots are the original mesh points and whose leaves are the redistributed mesh points so that SFBcast would migrate the points. Next, a PetscSection describing topology and the original PetscSF are used to build a PetscSF to distribute the topological relation. Then, the PetscSections describing the data layout for coordinates and other mesh fields are used to build PetscSFs that redistribute data over the mesh.

Ghost communication and assembly is also managed by PetscSFs. PETSc routines `DMGlobalToLocal()` and `DMLocalToGlobal()` use the aforementioned dof-PetscSF to communicate data between a global vector and a local vector with ghost points. Moreover, this communication pattern is reused to perform assembly as part of finite element and finite volume discretizations. The `PetscFE` and `PetscFV` objects can be used to automatically create the PetscSection objects for these data layouts, which together with the mesh point PetscSF automatically create the assembly PetscSF. This style of programming, with the emphasis on the declarative specification of data layout and communication patterns, *automatically* generating the actual communication and data manipulation routines, has resulted in more robust, extensible, and performant code.

5 Implementations

PetscSF has multiple implementations, including ones that utilize MPI one-sided or two-sided communication. We focus on the default implementation, which uses persistent MPI sends and receives for two-sided communication. In addition, we emphasize the implementations for GPUs.

5.1 Building Two-Sided Information

From the input arguments of PetscSFSetGraph, each MPI rank knows which ranks (i.e., root ranks) have roots that the current rank's leaves are connected to. In PetscSFSetUp, we compute the reverse information: which ranks (i.e., leaf ranks) have leaves of the current rank's roots. A typical algorithm uses MPI_Allreduce on an integer array of the MPI communicator size, which is robust but non-scalable. A more scalable algorithm uses MPI_Ibarrier, [22], which is PETSc's default with large communicators. In the end, on each rank, we compile the following information:

1) A list of root ranks, connected by leaves on this rank;
2) For each root rank, a list of leaf indices representing leaves on this rank that connects to the root rank;
3) A list of leaf ranks, connected by roots on this rank;
4) For each leaf rank, a list of root indices, representing roots on this rank that connect to the leaf rank.

These data structures facilitate message coalescing, which is crucial for performance on distributed memory. Note that processes usually play double roles: they can be both a root rank and a leaf rank.

5.2 Reducing Packing Overhead

With the two-sided information, we could have a simple implementation, using `PetscSFBcast(sf, MPI_DOUBLE, rootdata, leafdata, op)` as an example (Section 3.2). Each rank, as a sender, allocates a root buffer, `rootbuf`, packs the needed root data entries according to the root indices (`rootidx`) obtained above into the buffer, in a form such as `rootbuf[i] = rootdata[rootidx[i]]`, and then sends data in `rootbuf` to leaf ranks. Similarly, the receiver rank allocates a leaf buffer `leafbuf` as the receive buffer. Once it has received data, it unpacks data from `leafbuf` and deposits into the destination leafdata entries according to the leaf indices (`leafidx`) obtained above, in a form such as `leafdata[leafidx[i]]` $\oplus$= `leafbuf[i]`. Here $\oplus$= represents `op`.

However, PetscSF has several optimizations to lower or eliminate the packing (unpacking) overhead. First, we separate local (i.e., self-to-self) and remote communications. If on a process the PetscSF has local edges, then the process will show up in its leaf and root rank lists. We rearrange the lists to move self to the head if that is the case. By skipping MPI for local communication, we save intermediate send and receive buffers and pack and unpack calls. Local communication takes the form `leafdata[leafidx[i]]` $\oplus$= `rootdata[rootidx[i]]`. We call this a *scatter* operation.

This optimization is important in mesh repartitioning since most cells tend to stay on their current owner and hence local communication volume is large.

Second, we analyze the vertex indices and discover patterns that can be used to construct pack/unpack routines with fewer indirections. The most straightforward pattern is just contiguous indices. In that case, we can use the user-supplied rootdata/leafdata as the MPI buffers without any packing or unpacking. An important application of this optimization is in SpMV and its transpose introduced in Section 4.1, where the leaves, the entries in `lvec`, are contiguous. Thus, `lvec`'s data array can be directly used as the MPI receive buffers in `PetscSFBcast` of PETSc's SpMV or as MPI send buffers in `PetscSFReduce` of its transpose product. Note that, in general, we also need to consider the `MPI_Op` involved. If it is not an assignment, then we must allocate a receive buffer before *adding* it to the final destination. Note that allocated buffers are reused for repeated PetscSF operations.

Another pattern is represented by multi-strided indices, inspired by halo exchange in stencil computation on regular domains. In that case, ghost points living on faces of (sub)domains are either locally strided or contiguous. Suppose we have a three-dimensional domain of size `[X,Y,Z]` with points sequentially numbered in the *x*, *y*, *z* order. Also, suppose that within the domain there is a subdomain of size `[dx,dy,dz]` with the index of the first point being `start`. Then, indices of points in the subdomain can be enumerated with expression `start+X*Y*k+X*j+i`, for `(i,j,k)` in `(0≤i<dx,0≤j<dy,0≤k<dz)`. With this utility, faces or even the interior parts of a regular domain are all such qualified subdomains. Carrying several such parameters is enough for us to know all indices of a subdomain and for more efficient packs and unpacks on GPUs.

5.3 GPU-Aware MPI Support

Accelerator computation, represented by NVIDIA CUDA GPUs, brings new challenges to MPI. Users want to communicate data on the device while MPI runs on the host, but the MPI specification does not have a concept of device data or host data. In this paper's remainder, we use CUDA as an example, but the concept applies to other GPUs. With a non-CUDA-aware MPI implementation, programmers have to copy data back and forth between device and host to do computation on the device and communication on the host. This is a burden for programmers. CUDA-aware MPI can ease this problem, allowing programmers to directly pass device buffers to MPI with the same API. This is convenient, but there is still an MPI/CUDA semantic mismatch [23]. CUDA kernels are executed asynchronously on CUDA streams, but MPI has no concept of streams. Hence, MPI has no way to queue its operations to streams while maintaining correct data dependence. In practice, before sending data, users must synchronize the device to ensure that the data in the send buffer is ready for MPI to access at the moment of calling MPI send routines (e.g., MPI_Send). After receiving (e.g., MPI_Recv), MPI synchronizes the device again to ensure that the data in the receive buffer on the GPU is ready for users to access on *any* stream. These excessive synchronizations can impair pipelining of kernel launches. We address this issue later in the paper.

Since rootdata/leafdata is on the device, pack/unpack routines also have to be implemented as kernels, with associated vertex indices moved to the device. The packing optimizations discussed in the preceding paragraph are more useful on GPUs because we could either remove these kernels or save device memory otherwise allocated to store indices with patterns. Since we use CUDA threads to unpack data from the receive buffer in *parallel*, we distinguish the case of having duplicate indices, which may lead to data races. This is the case, for example, in the PetscSFReduce for `MatMultTranspose` (see Section 4.1): A single entry of the result vector `y` will likely receive contributions from multiple leaf ranks. In this case, we use CUDA atomics, whereas we use regular CUDA instructions when no duplicated indices are present.

PetscSF APIs introduced in Section 3 do not have stream or memory type arguments. Internally we call cudaPointerGetAttributes() to distinguish memory spaces. However, since this operation is expensive (around 0.3 µs per call from our experience), we extended PetscSF APIs to ones such as the following:

```
PetscSFBcastWithMemTypeBegin/End(PetscSF sf, MPI_Datatype
  unit, PetscMemType rootmtype, const void *rootdata,
  PetscMemType leafmtype, void *leafdata, MPI_Op op);
```

The extra `PetscMemType` arguments tell PetscSF where the data is. PETSc vectors have such built-in info, so that PETSc vector scatters internally use these extended APIs.

We could further extend the APIs to include stream arguments. However, since stream arguments, like C/C++ constantness, are so intrusive, we might have to extend many APIs to pass around the stream. We thus take another approach. PETSc has a default stream named *PetscDefaultCudaStream*. Almost all PETSc kernels work on this stream. When PETSc calls libraries such as cuBLAS and cuSPARSE, it sets their work stream to this default one. Although PetscSF assumes that data is on the default stream, it does provide options for users to indicate that data is on other streams so that PETSc will take stricter synchronizations.

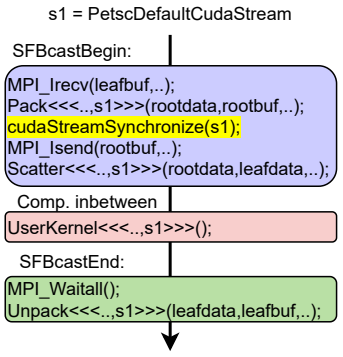

Fig. 4: CUDA-aware MPI support in PetscSFBcast. *Scatter* denotes the local communication. Note the cudaStreamSynchronize() before MPI_Isend.

Fig. 4 shows a diagram of the CUDA-aware MPI support in PetscSF using PetscSFBcast as an example. The code sequence is similar to what we have for host communication except that pack, unpack, and scatter operations are CUDA kernels, and there is a stream synchronization before

MPI_Isend, for reasons discussed above. If input data is not on PETSc's default stream, we call cudaDeviceSynchronize() in the beginning phase to sync the whole device and cudaStreamSynchronize() in the end phase to sync the default stream so that output data is ready to be accessed afterward.

5.4 Stream-Aware NVSHMEM Support

CUDA kernel launches have a cost of around 10 µs, which is not negligible considering that many kernels in practice have a smaller execution time. An important optimization with GPU asynchronous computation is to overlap kernel launches with kernel executions on the GPU, so that the launch cost is effectively hidden. However, the mandatory CUDA synchronization brought by MPI could jeopardize this optimization since it blocks the otherwise nonblocking kernel launches on the host. See Fig. 5 for an example. Suppose a kernel launch takes 10 µs and there are three kernels A, B, and C that take 20, 5, and 5 µs to execute, respectively. If the kernels are launched in a nonblocking way (Fig. 5(L)), the total cost to run them is 40 µs. Launch costs of B and C are completely hidden by the execution of A. However, if there is a synchronization after A (Fig. 5(R)), the total cost will be 55 µs. Scientific codes usually have many MPI calls, implying that their kernel launches will be frequently blocked by CUDA synchronizations. While the MPI community is exploring adding stream support in the MPI standard, we recently tried NVSHMEM for a remedy.

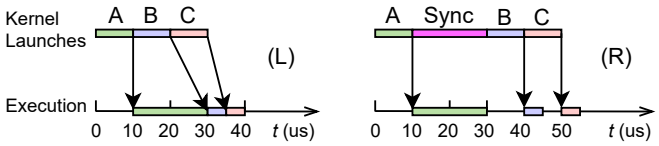

Fig. 5: Pipelined kernel launches (L) vs. interrupted kernel launches (R).

NVSHMEM [3] is NVIDIA's implementation of the OpenSHMEM [8] specification on CUDA devices. It supports point-to-point and collective communications between GPUs within a node or over networks. Communication can be initiated either on the host or on the device. Host side APIs take a stream argument. NVSHMEM is a PGAS library that provides one-sided *put*/*get* APIs for processes to access remote data. While using get is possible, we focus on put-based communication. In NVSHMEM, a process is called a processing element (PE). A set of PEs is a *team*. The team containing all PEs is called `NVSHMEM_TEAM_WORLD`. PEs can query their rank in a team and the team's size. One PE can use only one GPU. These concepts are analogous to ranks, communicators, and `MPI_COMM_WORLD` in MPI. NVSHMEM can be used with MPI. It is natural to map one MPI rank to one PE. We use PEs and MPI ranks interchangeably. With this approach, we are poised to bypass MPI to do communication on GPUs while keeping the rest of the PETSc code unchanged.

For a PE to access remote data, NVSHMEM uses the *symmetric data object* concept. At NVSHMEM initialization, all PEs allocate a block of CUDA memory, which is called a symmetric heap. Afterwards, every remotely accessible object has to be *collectively* allocated/freed by *all* PEs in `NVSHMEM_TEAM_WORLD`, with the *same* size, so that such an object always appears symmetrically on all PEs, at the same offset in their symmetric heap. PEs access remote data by referencing a symmetric address and the rank of the remote PE. A symmetric address is the address of a symmetric object on the local PE, plus an offset if needed. The code below allocates two symmetric double arrays `src[1]` and `dst[2]`, and every PE puts a double from its `src[0]` to the next PE's `dst[1]`.

```
double *src = nvshmem_malloc(sizeof(double));
double *dst = nvshmem_malloc(sizeof(double)*2);
int    pe   = nvshmem_team_my_pe(team); // get my rank
int    size = nvshmem_team_n_pes(team), next = (pe+1)%
nvshmemx_double_put_on_stream(&dst[1],src,1,next,stream);
```

For PEs to know the arrival of data put by other PEs and then read it, they can call a collective nvshmem_barrier(team) to separate the put and the read, or senders can send signals to receivers for checking. Signals are merely symmetric objects of type uint64_t. We prefer the latter approach since collectives are unfit for sparse neighborhood communications that are important to PETSc. Because of the collective memory allocation constraint, we support NVSHMEM only on PetscSFs built on `PETSC_COMM_WORLD`, which is the MPI communicator we used to initialize PETSc and NVSHMEM.

Fig. 6 gives a skeleton of the NVSHMEM support in PetscSF, again using PetscSFBcast as an example. We create a new stream `RemoteCommStream` (s2) to take charge of remote communication, such that communication and user's computation, denoted by `UserKernel`, could be overlapped. First, on `PetscDefaultCudaStream` (s1), we record a CUDA event `SbufReady` right after the pack kernel to indicate data in the send buffer is ready for send. Before sending, stream s2 waits for the event so that the send-after-pack dependence is enforced. Then PEs put data and *end-of-put* signals to destination PEs. To ensure that signals are delivered *after* data, we do an NVSHMEM memory fence at the local PE before putting signals. In the end phase, PEs wait until end-of-put signals targeting them have arrived (e.g., through `nvshmem_uint64_wait_until_all()`). Then they record an event `CommEnd` indicating end of communication on s2. PEs just need to wait for that event on s1 before unpacking data from the receive buffer. Note that all function calls in Fig. 6 are asynchronous.

NVSHMEM provides users a mechanism to distinguish locally accessible and remotely accessible PEs. One can roughly think of the former as intranode PEs and the latter as internode PEs. We take advantage of this and use different NVSHMEM APIs when putting data. We call the host API `nvshmemx_putmem_nbi_on_stream()` for each local PE, and the device API `nvshmem_putmem_nbi()` on CUDA threads, with each targeting a remote PE. For local PEs, the host API uses the efficient CUDA device copy engines to do GPUDirect peer-to-peer memory copy, while for remote PEs, it uses slower GPUDirect RDMA.

We now detail how we set up send and receive buffers for NVSHMEM. In CUDA-aware MPI support of PetscSF, we generally need to allocate on each rank two CUDA buffers, `rootbuf` and `leafbuf`, to function as MPI send or receive buffers. Now we must allocate them symmetrically

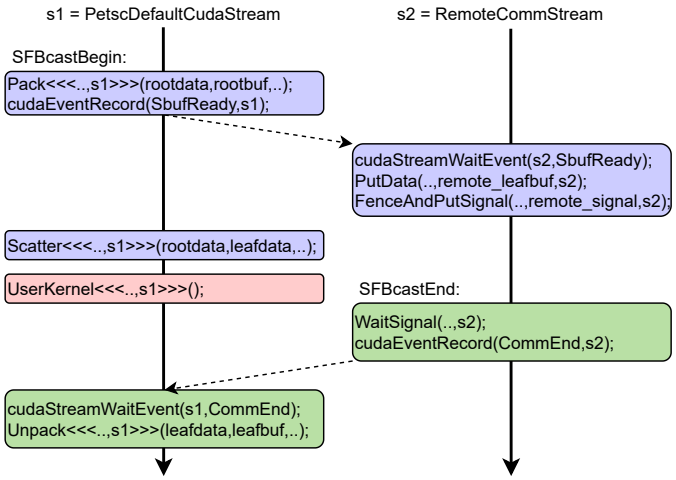

Fig. 6: Stream-aware NVSHMEM support in PetscSFBcast. Blue boxes are in the beginning phase, and green boxes are in the end phase. The red box in between is the user code. Dashed lines represent data dependence between streams. Functions are ordered vertically and called asynchronously.

to make them accessible to NVSHMEM. To that end, we call an MPI_Allreduce to get their maximal size over the communicator and use the result in `nvshmem_malloc()`. As usual, `leafbuf` is logically split into chunks of various sizes (see Fig. 7). Each chunk in an PetscSFBcast operation is used as a receive buffer for an associated root rank. Besides `leafbuf`, we allocate a symmetric object `leafRecvSig[]`, which is an array of the *end-of-put* signals with each entry associated with a root rank. In one-sided programming, a root rank has to know the associated chunk's offset in `leafbuf` to put the data and also the associated signal's offset in `leafRecvSig[]` to set the signal. The preceding explanations apply to `rootbuf` and leaf ranks similarly. In PetscSFSetUp, we use MPI two-sided to assemble the

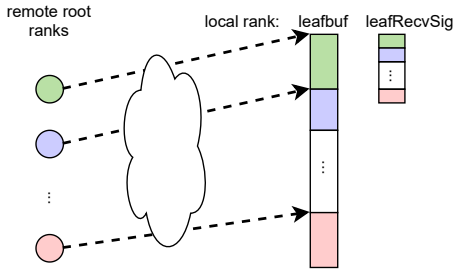

Fig. 7: Right: two symmetric objects on a local rank. Left: remote root ranks putting data to the objects. Root ranks need to know offsets at the remote side. The cloud indicates these are remote accesses.

information needed for NVSHMEM one-sided. At the end of the setup, on each rank, we have the following new data structures in addition to those introduced in Section 5.1:

1) A list of offsets, each associated with a leaf rank, showing where the local rank should send (put) its `rootbuf` data to these leaf ranks' `leafbuf`
2) A list of offsets, each associated with a leaf rank, showing where the local rank should set signals on these leaf ranks
3) A list of offsets, each associated with a root rank, showing where the local rank should send (put) its `leafbuf` data to these root ranks' `rootbuf`
4) A list of offsets, each associated with a root rank, showing where the local rank should set signals on these root ranks

With these, we are almost ready to implement PetscSF with NVSHMEM. But there is a new complexity coming with one-sided communication. Suppose we do communication in a loop. When receivers are still using their receive buffer, senders could move into the next iteration and put new data into the receivers' buffer and corrupt it. To avoid this situation, we designed a protocol shown in Fig. 8, between a pair of PEs (sender and receiver). Besides the end-of-put signal (RecvSig) on the receiver side, we allocate an *ok-to-put* signal (SendSig) on the sender side. The sender must wait until the variable is 0 to begin putting in the data. Once the receiver has unpacked data from its receive buffer, it sends 0 to the sender's SendSig to give it permission to put the next collection of data.

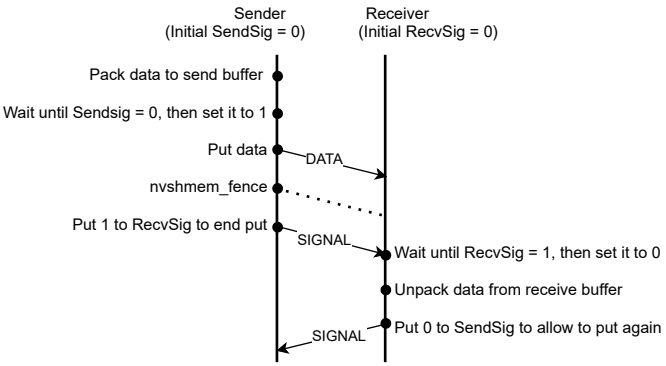

Fig. 8: Protocol of the PetscSF NVSHMEM put-based communication. The dotted line indicates that the signal below is observable only after the remote put is completed at the destination. We have rootSendSig, rootRecvSig, leafSendSig, and leafRecvSig symmetric objects on each PE, with an initial value 0.

6 Experimental Results

We evaluated PetscSF on the Oak Ridge Leadership Computing Facility (OLCF) Summit supercomputer as a surrogate for the upcoming exascale computers. Each node of Summit has two sockets, each with an IBM Power9 CPU accompanied by three NVIDIA Volta V100 GPUs. Each CPU and its three GPUs are connected by the NVLink interconnect at a bandwidth of 50 GB/s. Communication between the two CPUs is provided by IBM's X-Bus at a bandwidth of 64 GB/s. Each CPU also connects through a PCIe Gen4 x8 bus with a Mellanox InfiniBand network interface card (NIC) with a bandwidth of 16 GB/s. The NIC has an injection bandwidth of 25 GB/s. On the software side, we used gcc 6.4 and the CUDA-aware IBM Spectrum MPI 10.3. We

also used NVIDIA CUDA Toolkit 10.2.89, NVSHMEM 2.0.3, NCCL 2.7.8 (NVIDIA Collective Communication Library, which implements multi-GPU and multi-node collectives for NVIDIA GPUs), and GDRCopy 2.0 (a low-latency GPU memory copy library). NVSHMEM needs the latter two.

In [24], we provided a series of microbenchmarks for PetscSF and studied various part of it with CUDA-aware MPI. In this paper, we focus more on application-level evaluation on both CPUs and GPUs.

6.1 PetscSF Ping Pong Test

To determine the overhead of PetscSF, we wrote a ping pong test *sf_pingpong* in PetscSF, to compare PetscSF performance against raw MPI performance. The test uses two MPI ranks and a PetscSF with n roots on rank 0 and n leaves on rank 1. The leaves are connected to the roots consecutively. With this PetscSF and op = `MPI_REPLACE`, PetscSFBcast sends a message from rank 0 to rank 1, and a following SFReduce bounces a message back. Varying n, we can then measure the latency of various message sizes, mimicing the *osu_latency* test from the OSU microbenchmarks [25]. By comparing the performance attained by the two tests, we can determine the overhead of PetscSF.

We measured PetscSF MPI latency with either host-to-host messages (H-H) or device-to-device (D-D) messages. By device messages, we mean regular CUDA memory data, not NVSHMEM symmetric memory data. Table 1 shows the intra-socket results, where the two MPI ranks were bound to the same CPU and used two GPUs associated with that CPU. The roots and leaves in this PetscSF are contiguous, so PetscSF's optimization skips the pack/unpack operations. Thus this table compares a raw MPI code with a PetscSF code that embodies much richer semantics. Comparing the H-H latency, we see that PetscSF has an overhead from 0.6 to 1.0 µs, which is spent on checking input arguments and bookkeeping. The D-D latency is interesting. It shows that PetscSF has an overhead of about 5 µs over the OSU test. We verified this was because PetscSF calls cudaStreamSynchronize() before sending data, whereas the OSU test does not. We must have the synchronization in actual application codes, as discussed before. We also performed inter-socket and inter-node tests. Results, see [24], indicated a similar gap between the PetscSF test and the OSU test.

TABLE 1: Intra-socket host-to-host (H-H) latency and device-to-device (D-D) latency (µs) measured by osu_latency (OSU) and sf_pingpong (SF), with IBM Spectrum MPI.

Msg (Bytes)	1K	4K	16K	64K	256K	1M	4M
OSU H-H	0.8	1.3	3.5	4.7	12.2	36.3	152.4
SF H-H	1.5	1.9	4.2	5.5	12.9	37.3	151.8
OSU D-D	17.7	17.7	17.8	18.4	22.5	39.2	110.3
SF D-D	22.8	23.0	22.9	23.5	27.7	46.3	111.8

We used the same test and compared PetscSF MPI and PetscSF NVSHMEM, with results shown in Fig. 9. For small messages, NVSHMEM's latency is about 12 µs higher than MPI's in intra-socket and inter-socket cases and 40 µs more in the inter-node cases. For large inter-socket messages, the gap between the two is even larger (up to 60 µs at 4 MB). Possible reasons for the performance difference include: (1) In PetscSF NVSHMEM, there are extra memory copies of

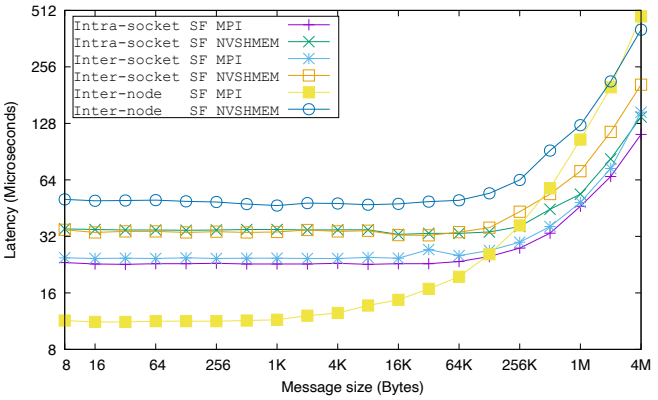

Fig. 9: Device to device (D-D) latency measured by sf_pingpong, using CUDA-aware IBM MPI or NVSHMEM.

data between the root/leafdata in CUDA memory and the root and leaf buffers in NVSHMEM memory; (2) The current NVSHMEM API has limitations. For example, we would like to use fewer kernels to implement the protocol in Fig. 8. Within a node, the NVSHMEM host API delivers much better performance than its device API, forcing us to do signal-wait through device API, but data-put through the host API. For another example, NVSHMEM provides a device API to do data-put and signal-put in one call, but there is no host counterpart. One has to use two kernel launches for this task using the host API for data-put. All these extra kernel launches increase the latency; (3) NVSHMEM is still a new NVIDIA product. There is much headroom for it to grow.

6.2 Asynchronous Conjugate Gradient on GPUs

To explore distributed asynchronous execution on GPUs enabled by NVSHMEM, we adapted CG, the Krylov conjugate gradient solver in PETSc, to a prototype asynchronous version CGAsync. Key differences between the two are as follows. (1) A handful of PETSc routines they call are different. There are two categories. The first includes routines with scalar output parameters, for example, vector dot product. CG calls `VecDot(Vec x,Vec y,double *a)` with `a` pointing to a host buffer, while CGAsync calls `VecDotAsync(Vec x, Vec y, double *a)` with `a` referencing a device buffer. In VecDot, each process calls cuBLAS routines to compute a partial dot product and then copies it back to the host, where it calls MPI_Allreduce to get the final dot product and stores it at the host buffer. Thus VecDot synchronizes the CPU and the GPU device. While in VecDotAsync, once the partial dot product from cuBLAS is computed, each process calls an NVSHMEM reduction operation on PETSc's default stream to compute the final result and stores it at the device buffer. The second category of differences includes routines with scalar input parameters, such as `VecAXPY(Vec y,double a, Vec x)`, which computes y += a*x. CG calls `VecAXPY` while CGAsync calls `VecAXPYAsync(Vec y,double *a, Vec x)` with `a` referencing device memory, so that `VecAXPYAsync` can be queued to a stream while `a` is computed on the device. (2) CG does scalar arithmetic (e.g., divide two scalars) on the CPU, while CGAsync does

them with tiny *scalar kernels* on the GPU. (3) CG checks convergence (by comparison) in every iteration on the CPU to determine whether it should exit the loop while CGAsync does not. Users need to specify maximal iterations. This could be improved by checking for convergence every few (e.g., 20) iterations. We leave this as future work.

We tested CG and CGAsync without preconditioning on a single Summit compute node with two sparse matrices from the SuiteSparse Matrix Collection [26]. The CG was run with PetscSF CUDA-aware MPI, and CGAsync was run with PetscSF NVSHMEM. The first matrix is Bump_2911 with about 3M rows and 128M nonzero entries. We ran both algorithms 10 iterations with 6 MPI ranks and one GPU per rank. Fig. 10 shows their timeline through the profiler NVIDIA NSight Systems. The kernel launches (label `CUDA API`) in CG were spread over the 10 iterations. The reason is that in each iteration, there are multiple MPI calls (mainly in MatMult as discussed in Section 4.1, and vector dot and norm operations), which constantly block the kernel launch pipeline. In CGAsync, however, while the device was executing the 8th iteration (with profiling), the host had launched *all* kernels for 10 iterations. The long red bar cudaMemcpyAsync indicates that after the kernel launches, the host was idle, waiting for the final result from the device.

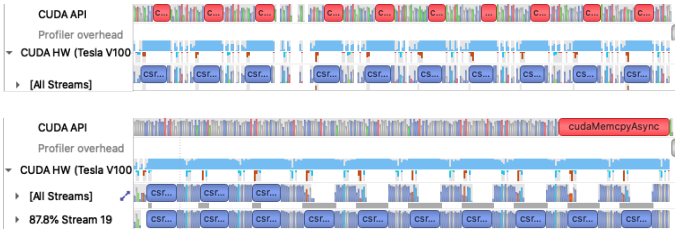

Fig. 10: Timeline of CG (top) and CGAsync (bottom) on rank 2. Each ran ten iterations. The blue *csr...* bars are csrMV (i.e., SpMV) kernels in cuSPARSE, and the red *c...* bars are cudaMemcpyAsync() copying data from device to host.

Test results show that the time per iteration for CG and CGAsync was about 690 and 676 µs, respectively. CGAsync gave merely a 2% improvement. This small improvement is because the matrix is huge, and computation is using the vast majority of the time. From profiling, we knew SpMV alone (excluding communication) took 420 µs. If one removes the computational time, the improvement in communication time is substantial. Unfortunately, because of bugs in the NVSHMEM library with multiple nodes, we could not scale the test to more compute nodes. Instead, we used a smaller matrix, Kuu, of about 7K rows and 340K nonzero entries to see how CGAsync would perform in a strong-scaling sense. We repeated the above tests. Time per iteration for CG and CGAsync was about 300 and 250 µs. CGAsync gave a 16.7% improvement. Note that this improvement was attained despite the higher ping pong latency of PetscSF NVSHMEM.

We believe CGAsync has considerable potential for improvement. As the NVSHMEM library matures, it should reach or surpass MPI's ping pong performance. We also note that there are many kernels in CGAsync; for the scalar kernels mentioned above, kernel launch times could not be entirely hidden with small matrices. We are investigating techniques like CUDA Graphs to automatically fuse kernels in the loop to further reduce the launch cost; with MPI, such fusion within an iteration is not possible due to the synchronizations that MPI mandates.

6.3 Mesh Distribution on CPU

This section reports on the robustness and efficiency of the PetscSF infrastructure as used in the mesh distribution process through DMPlex. In the first stage, the cell-face connectivity graph is constructed and partitioned, followed by the mesh data's actual migration (topology, labels associated with mesh points, and cell coordinates) and then the distributed mesh's final setup. We do not analyze the stage around graph partitioning and instead focus on the timings associated with the distribution of the needed mesh data followed by the final local setup.

We consider the migration induced by a graph partitioning algorithm on three different initial distributions of a fully periodic $128 \times 128 \times 128$ hexahedral mesh:

- Seq: the mesh is entirely stored on one process.
- Chunks: the mesh is stored in non-overlapping chunks obtained by a simple distribution of the lexicographically ordered cells.
- Rand: the mesh is stored randomly among processes.

The sequential case is common in scientific applications when the mesh is stored in a format that is not suitable for parallel reading. It features a one-to-all communication pattern. The Chunks and Rand cases represent different mesh distribution scenarios after parallel read, and a many-to-many communication pattern characterizes them. Ideally, the Chunks case would have a more favorable communication pattern than the Rand case, where potentially all processes need to send/receive data from all processes. Fig. 11 collects the mesh migration timing as a function of the number of processes used in the distribution. Timings remain essentially constant as the number of processes is increased from 420 to 16,800 due to increased communication times and a decrease in the subsequent local setup time, confirming the scalability of the overall implementation. The Chunks' timings had some irregularity, since initial mesh distributions in this case varied from less-optimal to almost-optimal, affecting the timed second redistribution.

6.4 Parallel Sparse Matrix-Matrix Multiplication (SpMM)

We recently developed a generic driver for parallel sparse matrix-matrix multiplications that takes advantage of the block-row distribution used in PETSc for their storage; see Fig. 3. Here we report preliminary results using cuSparse to store and manipulate the local matrices. In particular, given two parallel sparse matrices A and P, we consider the matrix product AP and the projection operation $P^T AP$, by splitting them into three different phases:

1) Collect rows of P corresponding to the nonzero columns of the off-diagonal portion of A.
2) Perform local matrix-matrix operations.
3) Assemble the final product, possibly involving off-process values insertion.

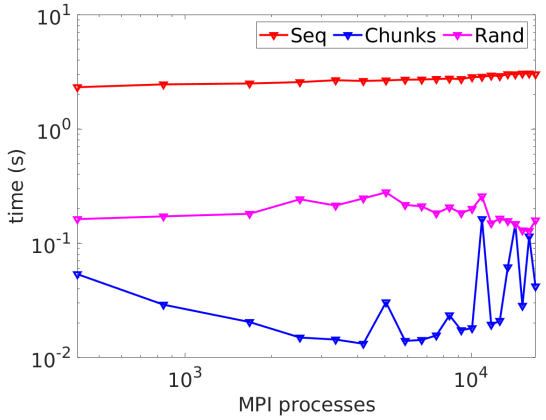

Fig. 11: DMPlex mesh migration timings as a function of the number of MPI processes for the Sequential, Chunks, and Random use cases, showing the robustness and scalability of the mesh operations using PetscSF.

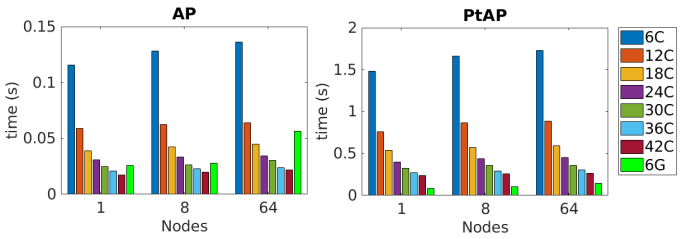

Fig. 12: Timings for parallel sparse matrix-matrix numerical products with constant workload per node. Left is AP; right is $P^T AP$. 6 GPUs per node (6G) are compared against an increasing number of cores per node (6C-42C) with different numbers of Summit nodes.

Step 1 is a submatrix extraction operation, while step 2 corresponds to a sequence of purely local matrix-matrix products that can execute on the device. We set up an intermediate PetscSF with leaf vertices represented by the row indices of the result matrix and root vertices given by its row distribution during the symbolic phase. The communication of the off-process values needed in step 3 is then performed using the `SFGather` operation on a temporary GPU buffer, followed by local GPU-resident assembly.

The performances of the numerical phase of these two matrix-matrix operations on GPUs are plotted in Fig. 12 for different numbers of nodes of Summit, and they are compared against our highly optimized CPU matrix-matrix operations directly calling MPI send and receive routines; the A operators are obtained from a second-order finite element approximation of the Laplacian, while the P matrices are generated by the native algebraic multigrid solver in PETSc. The finite element problem defined on a tetrahedral unstructured mesh is partitioned and weakly scaled among Summit nodes, from 1 to 64; the number of rows of A ranges from 1.3 million to 89.3 million. While the workload is kept fixed per node, a strong-scaling analysis is carried out within a node, and the timings needed by the numerical algorithm using 6 GPUs per node (label 6G) are compared against an increasing number of cores per node (from 6 to 42, labeled 6C and 42C, respectively). The Galerkin triple matrix product shows the most promising speedup, while the performances of the matrix product AP, cheaper to be computed, are more dependent on the number of nodes. We plan to perform further analysis and comparisons when our NVSHMEM backend for PetscSF supports multinode configurations and we have full support for asynchronous device operations using streams within the PETSc library.

7 Conclusion and Future Work

We introduced PetscSF, the communication component in PETSc, including its programming model, its API, and its implementations. We emphasized the implementation on GPUs since one of our primary goals is to provide highly efficient PetscSF implementations for the upcoming exascale computers. Our experiments demonstrated PetscSF's performance, overhead, scalabilty, and novel asynchronous features. We plan to continue to optimize PetscSF for exascale systems and to investigate asynchronous computation on GPUs enabled by PetscSF at large scale.

Acknowledgments

We thank Akhil Langer and Jim Dinan from the NVIDIA NVSHMEM team for their assistance. This work 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, and by the U.S. Department of Energy under Contract DE-AC02-06CH11357 and Office of Science Awards DE-SC0016140 and DE-AC02-0000011838. This research used resources of the Oak Ridge Leadership Computing Facilities, a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

References

[1] R. T. Mills, M. F. Adams, S. Balay, J. Brown, A. Dener, M. Knepley, S. E. Kruger, H. Morgan, T. Munson, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and J. Zhang, "Toward performance-portable PETSc for GPU-based exascale systems," *https://arxiv.org/abs/2011.00715*, 2020, submitted for publication.

[2] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.15.0, 2021, https://www.mcs.anl.gov/petsc.

[3] NVIDIA, "NVIDIA OpenSHMEM library (NVSHMEM) documentation," 2021, https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/introduction.html.

[4] D. B. Loveman, "High Performance Fortran," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, pp. 25–42, 1993.

[5] G. D. Bonachea and G. Funck, "UPC Language Specifications, Version 1.3," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-6623E, 2013, http://upc-lang.org.

[6] R. W. Numrich and J. Reid, "Co-Array Fortran for parallel programming," in *ACM Sigplan Fortran Forum*, vol. 17, no. 2, 1998, pp. 1–31.

[7] B. L. Chamberlain, S. Deitz, M. B. Hribar, and W. Wong, "Chapel," *Programming Models for Parallel Computing*, pp. 129–159, 2015.

[8] Open Source Software Solutions, Inc., "OpenSHMEM application programming interface v1.5," 2020, http://www.openshmem.org/.

[9] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine, "The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring," *Scientific Programming*, vol. 20, no. 2, pp. 129–150, 2012.

[10] "GSLIB," https://github.com/Nek5000/gslib, 2021.

[11] "nek5000," https://nek5000.mcs.anl.gov, 2021.

[12] D. Morozov and T. Peterka, "Block-parallel data analysis with DIY2," in *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE, 2016, pp. 29–36.

[13] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[14] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2019, pp. 71–81.

[15] The Khronos SYCL Working Group, "SYCL 2020 specification revision 3," 2020, https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

[16] Y. Saad, "Data structures and algorithms for domain decomposition and distributed sparse matrix computations," Department of Computer Science, University of Minnesota, Tech. Rep. 95-014, 1995.

[17] M. G. Knepley and D. A. Karpeev, "Mesh algorithms for PDE with Sieve I: Mesh distribution," *Scientific Programming*, vol. 17, no. 3, pp. 215–230, 2009.

[18] M. Lange, L. Mitchell, M. G. Knepley, and G. J. Gorman, "Efficient mesh management in Firedrake using PETSc-DMPlex," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S143–S155, 2016.

[19] V. Hapla, M. G. Knepley, M. Afanasiev, C. Boehm, M. van Driel, L. Krischer, and A. Fichtner, "Fully parallel mesh I/O using PETSc DMPlex with an application to waveform modeling," *SIAM Journal on Scientific Computing*, vol. 43, no. 2, pp. C127–C153, 2021.

[20] Wikipedia, "Hasse diagram," 2015, http://en.wikipedia.org/wiki/Hasse_diagram.

[21] M. G. Knepley, M. Lange, and G. J. Gorman, "Unstructured overlapping mesh distribution in parallel," https://arxiv.org/abs/1506.06194, 2017.

[22] T. Hoefler, C. Siebert, and A. Lumsdaine, "Scalable communication protocols for dynamic sparse data exchange," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 159–168, 2010.

[23] 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)*, Nov 2018, pp. 1–13.

[24] J. Zhang, R. T. Mills, and B. F. Smith, "Evaluation of PETSc on a heterogeneous architecture, the OLCF Summit system: Part II: Basic communication performance," Argonne National Laboratory, Tech. Rep. ANL-20/76, 2020.

[25] D. Panda *et al.*, "OSU micro-benchmarks v5.7," *http://mvapich.cse.ohio-state.edu/benchmarks/*, 2020.

[26] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

Junchao Zhang is a software engineer at Argonne National Laboratory. He received his Ph.D. in computer science from the Chinese Academy of Sciences, Beijing, China. He is a PETSc developer and works mainly on communication and GPU support in PETSc.

Jed Brown is an assistant professor of computer science at the University of Colorado Boulder. He received his Dr.Sc. from ETH Zürich and BS+MS from the University of Alaska Fairbanks. He is a maintainer of PETSc and leads a research group on fast algorithms and community software for physical prediction, inference, and design.

Satish Balay is a software engineer at Argonne National Laboratory. He received his M.S. in computer science from Old Dominion University. He is a developer of PETSc.

Jacob Faibussowitsch is a Ph.D. student in mechanical engineering and computational science and engineering at the University of Illinois at Urbana-Champaign, where he also received his B.S. His work focuses on high-performance scalable fracture mechanics at the Center for Exascale-Enabled Scramjet Design. He is a developer of PETSc.

Matthew Knepley is an associate professor in the University at Buffalo. He received his Ph.D. in computer science from Purdue University and his B.S. from Case Western Reserve University. His work focuses on computational science, particularly in geodynamics, subsurface flow, and molecular mechanics. He is a maintainer of PETSc.

Oana Marin is an assistant applied mathematics specialist at Argonne National Laboratory. She received her Ph.D. in theoretical numerical analysis at the Royal Institute of Technology, Sweden. She is an applications-oriented applied mathematician who works on numerical discretizations in computational fluid dynamics, mathematical modeling, and data processing.

Richard Tran Mills is a computational scientist at Argonne National Laboratory. His research spans high-performance scientific computing, machine learning, and the geosciences. He is a developer of PETSc and the hydrology code PFLOTRAN. He earned his Ph.D. in computer science at the College of William and Mary, supported by a U.S. Department of Energy Computational Science Graduate Fellowship.

Todd Munson is a senior computational scientist at Argonne National Laboratory and the Software Ecosystem and Delivery Control Account Manager for the U.S. DOE Exascale Computing Project. His interests range from numerical methods for nonlinear optimization and variational inequalities to workflow optimization for online data analysis and reduction. He is a developer of the Toolkit for Advanced Optimization.

Barry F. Smith is an Argonne National Laboratory Associate. He is one of the original developers of the PETSc numerical solvers library. He earned his Ph.D. in mathematics at the Courant Institute.

Stefano Zampini is a research scientist in the Extreme Computing Research Center of King Abdullah University for Science and Technology (KAUST), Saudi Arabia. He received his Ph.D. in applied mathematics from the University of Milano Statale. He is a developer of PETSc.