Stony Brook University

Stream Support in MPI without the Churn

Joseph Schuchart, Institute for Advanced Computational Science, Stony Brook University

Joseph.Schuchart@stonybrook.edu

Edgar Gabriel, AMD

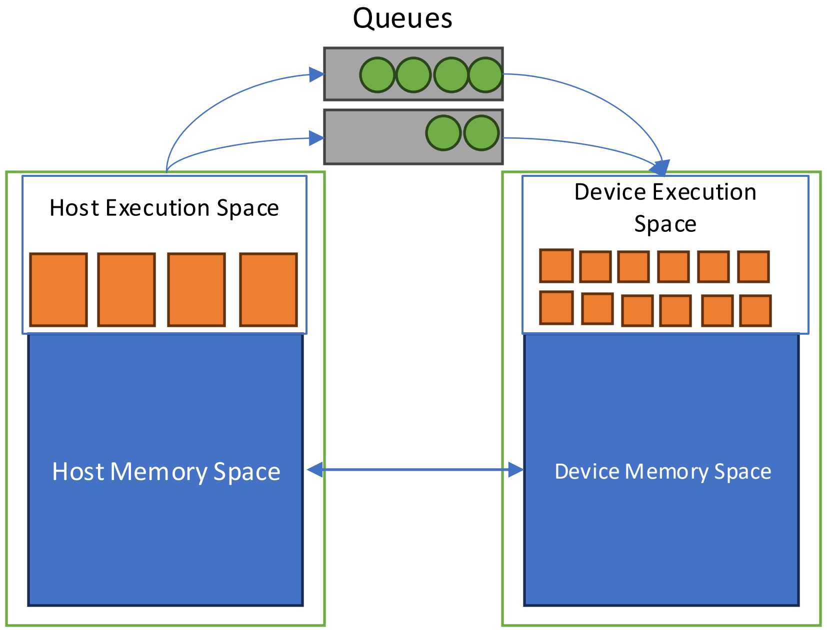
FAR
BEYOND

iACS INSTITUTE FOR ADVANCED COMPUTATIONAL SCIENCE

Motivation

- Accelerators provide separate **memory space** and **execution space**
- Host controls device execution space through queues/streams
- Data produced by device becomes eventually available for MPI to consume
- Data received by MPI will be consumed by device kernels
- **Memory spaces** exposed through new info keys

MPI is blissfully unaware of execution spaces so full synchronization is required before calling MPI.



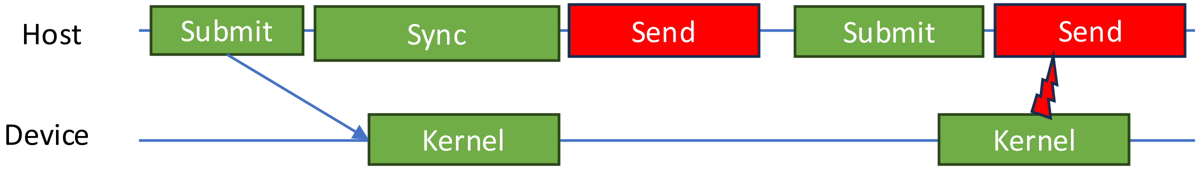
Execution Spaces in MPI Today

MPI exclusively interacts with the **host execution space**

Blocking operations block the calling thread

Nonblocking (and **persistent**) operations are ordered with operations on the calling thread prior to the starting MPI call

Applications must **synchronize device streams** producing data before calling MPI



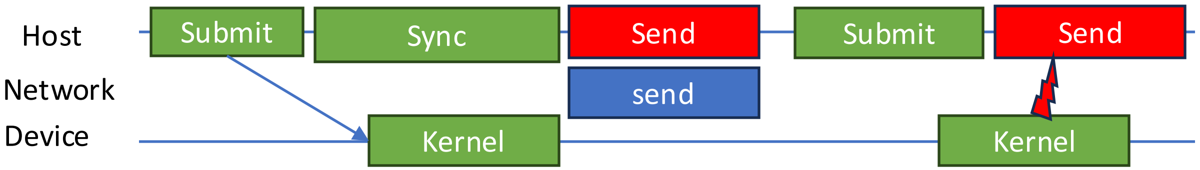
Why We Want Stream-Awareness

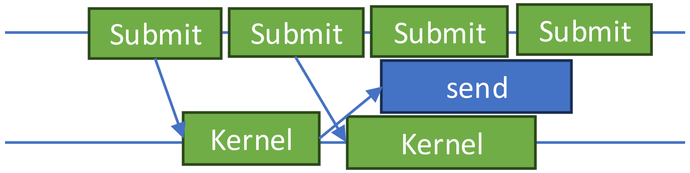
Correctness

- Exposes the device execution space
- Without proper synchronization MPI sees inconsistent data
- Source of errors in applications
- Allow applications to order kernel submission with MPI operations

Performance

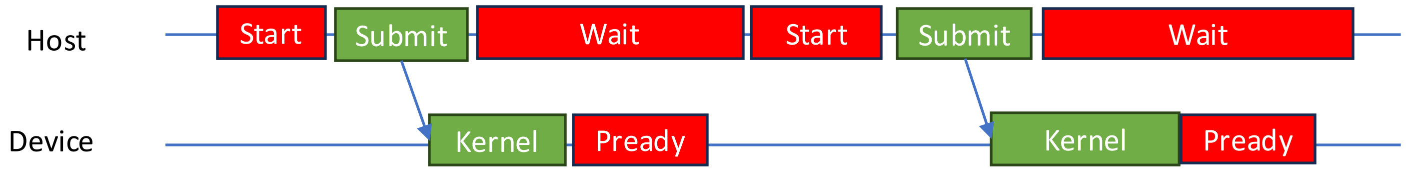
- Synchronization of queues blocks the host-thread and drains the device
- Integration increases potential for overlapping kernel submission and execution
- Enables MPI to interact with streams, e.g., to enqueue memory transfers or reduction operations





Orthogonal: Device-Side Partitioned Operations

- Allow kernels to start parts of communication inside a kernel
- Enables fine-grain data transfers
- Still requires completion & start from the host
- Stream integration complements

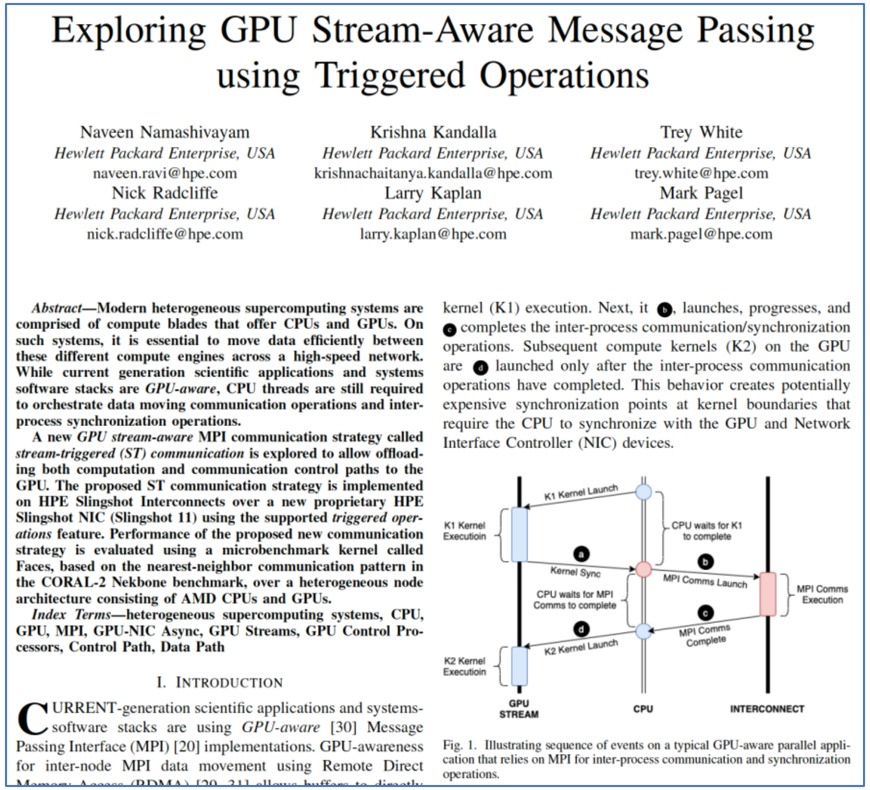


Alternative: Device Bindings for MPI

- Unlikely to offload all MPI functionality to devices
- Vendor libraries offload few operations supported by hardware, with constraints
- Significant burden on implementors
- Challenges: request management, stream-blocking, message matching, …

MPI & Streams: Prior Work

- Two similar proposals that wrap compute streams
  - MPIX_Streams [1]
  - MPIX_Queue [2]
- MPI Operations are enqueued into a stream
- Dedicated stream/queue object
- API duplication
- Relying on strong progress



Exploring GPU Stream-Aware Message Passing using Triggered Operations

Naveen Namashivayam
Hewlett Packard Enterprise, USA
naveen.ravi@hpe.com

Krishna Kandalla
Hewlett Packard Enterprise, USA
krishnachaitanya.kandalla@hpe.com

Trey White
Hewlett Packard Enterprise, USA
trey.white@hpe.com

Nick Radcliffe
Hewlett Packard Enterprise, USA
nick.radcliffe@hpe.com

Larry Kaplan
Hewlett Packard Enterprise, USA
larry.kaplan@hpe.com

Mark Pagel
Hewlett Packard Enterprise, USA
mark.pagel@hpe.com

***Abstract*—Modern heterogeneous supercomputing systems are comprised of compute blades that offer CPUs and GPUs. On such systems, it is essential to move data efficiently between these different compute engines across a high-speed network. While current generation scientific applications and systems software stacks are *GPU-aware*, CPU threads are still required to orchestrate data moving communication operations and inter-process synchronization operations.**

A new *GPU stream-aware* MPI communication strategy called *stream-triggered (ST) communication* is explored to allow offloading both computation and communication control paths to the GPU. The proposed ST communication strategy is implemented on HPE Slingshot Interconnects over a new proprietary HPE Slingshot NIC (Slingshot 11) using the supported *triggered operations* feature. Performance of the proposed new communication strategy is evaluated using a microbenchmark kernel called Faces, based on the nearest-neighbor communication pattern in the CORAL-2 Nekbone benchmark, over a heterogeneous node architecture consisting of AMD CPUs and GPUs.

***Index Terms*—heterogeneous supercomputing systems, CPU, GPU, MPI, GPU-NIC Async, GPU Streams, GPU Control Processors, Control Path, Data Path**

I. INTRODUCTION

CURRENT-generation scientific applications and systems-software stacks are using *GPU-aware* [30] Message Passing Interface (MPI) [20] implementations. GPU-awareness for inter-node MPI data movement using Remote Direct

kernel (K1) execution. Next, it ⓑ, launches, progresses, and ⓒ completes the inter-process communication/synchronization operations. Subsequent compute kernels (K2) on the GPU are ⓓ launched only after the inter-process communication operations have completed. This behavior creates potentially expensive synchronization points at kernel boundaries that require the CPU to synchronize with the GPU and Network Interface Controller (NIC) devices.

Fig. 1. Illustrating sequence of events on a typical GPU-aware parallel application that relies on MPI for inter-process communication and synchronization operations.

---

[1] Zhou, H., Raffenetti, K., Guo, Y., Thakur, R.: MPIX stream: an explicit solution to hybrid MPI+X programming. In: Proceedings of the 29th European MPI Users' Group Meeting, EuroMPI/USA 2022

[2] Namashivayam, N., Kandalla, K., White, T., Radcliffe, N., Kaplan, L., Pagel, M.: Exploring GPU stream-aware message passing using triggered operations. (2022)

Related: MPIX_Streams

1. Create a stream from a ninfo object with device stream hex-encoded
2. Create a stream-comm from that stream
3. Explicit enqueue functions for blocking & nonblocking operations & wait

MPIX_Info_set_hex()
MPIX_Stream_create()
MPIX_Stream_comm_create()
MPIX_Send_enqueue()
MPIX_Isend_enqueue()
MPIX_Wait_enqueue()

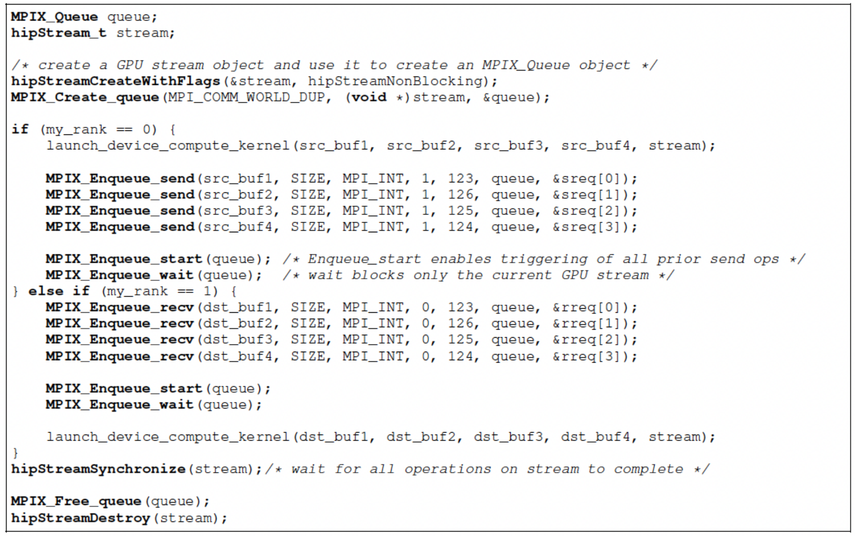
Proposed for broader use with multi-threading through multiplexing

[1] Zhou, H., Raffenetti, K., Guo, Y., Thakur, R.: MPIX stream: an explicit solution to hybrid MPI+X programming. In: Proceedings of the 29th European MPI Users' Group Meeting, EuroMPI/USA 2022

Related: MPIX_Enqueue

1. Create an MPIX_Queue object
2. Enqueue operations into the queue
3. Start the queue
4. Wait for the queue to complete

MPIX_Create_queue()
MPIX_Free_queue()
MPIX_Enqueue_send()
MPIX_Enqueue_start()
MPIX_Enqueue_wait()

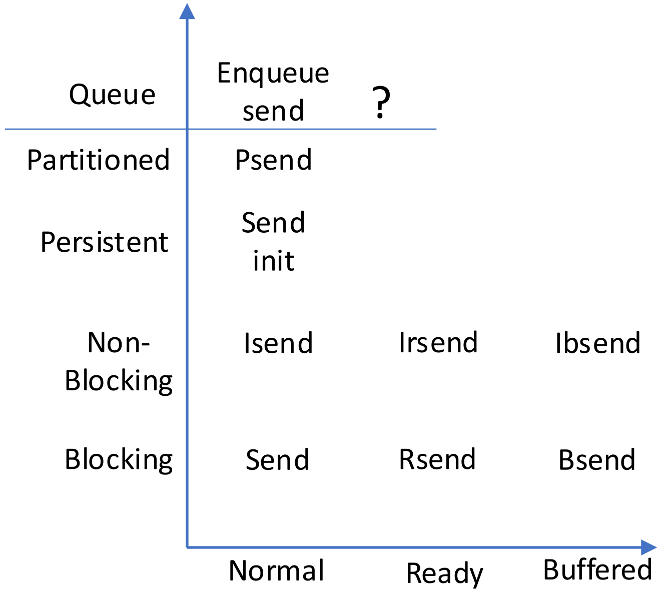


```
MPIX_Queue queue;
hipStream_t stream;

/* create a GPU stream object and use it to create an MPIX_Queue object */
hipStreamCreateWithFlags(&stream, hipStreamNonBlocking);
MPIX_Create_queue(MPI_COMM_WORLD_DUP, (void *)stream, &queue);

if (my_rank == 0) {
    launch_device_compute_kernel(src_buf1, src_buf2, src_buf3, src_buf4, stream);

    MPIX_Enqueue_send(src_buf1, SIZE, MPI_INT, 1, 123, queue, &sreq[0]);
    MPIX_Enqueue_send(src_buf2, SIZE, MPI_INT, 1, 126, queue, &sreq[1]);
    MPIX_Enqueue_send(src_buf3, SIZE, MPI_INT, 1, 125, queue, &sreq[2]);
    MPIX_Enqueue_send(src_buf4, SIZE, MPI_INT, 1, 124, queue, &sreq[3]);

    MPIX_Enqueue_start(queue); /* Enqueue_start enables triggering of all prior send ops */
    MPIX_Enqueue_wait(queue);  /* wait blocks only the current GPU stream */
} else if (my_rank == 1) {
    MPIX_Enqueue_recv(dst_buf1, SIZE, MPI_INT, 0, 123, queue, &rreq[0]);
    MPIX_Enqueue_recv(dst_buf2, SIZE, MPI_INT, 0, 126, queue, &rreq[1]);
    MPIX_Enqueue_recv(dst_buf3, SIZE, MPI_INT, 0, 125, queue, &rreq[2]);
    MPIX_Enqueue_recv(dst_buf4, SIZE, MPI_INT, 0, 124, queue, &rreq[3]);

    MPIX_Enqueue_start(queue);
    MPIX_Enqueue_wait(queue);

    launch_device_compute_kernel(dst_buf1, dst_buf2, dst_buf3, dst_buf4, stream);
}
hipStreamSynchronize(stream);/* wait for all operations on stream to complete */

MPIX_Free_queue(queue);
hipStreamDestroy(stream);
```

[2] Namashivayam, N., Kandalla, K., White, T., Radcliffe, N., Kaplan, L., Pagel, M.: Exploring GPU stream-aware message passing using triggered operations. (2022)

In This Work

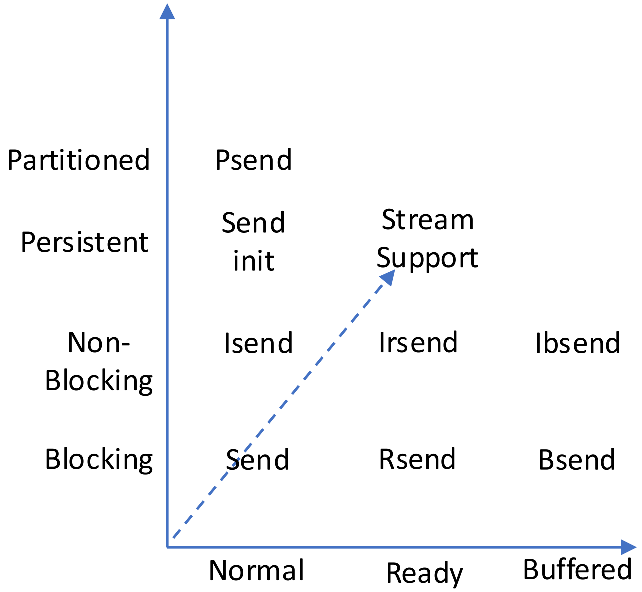
- Explore possible design of minimal extension for device stream integration in MPI
- Avoid significant expansion of MPI API
- Apply existing operation semantics to device streams



	Normal	Ready	Buffered
Queue	Enqueue send	?	
Partitioned	Psend		
Persistent	Send init		
Non-Blocking	Isend	Irsend	Ibsend
Blocking	Send	Rsend	Bsend

In This Work

- Explore possible design of minimal extension for device stream integration in MPI
- Avoid significant expansion of MPI API
- Apply existing operation semantics to device streams



	Normal	Ready	Buffered
Partitioned	Psend		
Persistent	Send init		
Non-Blocking	Isend	Irsend	Ibsend
Blocking	Send	Rsend	Bsend

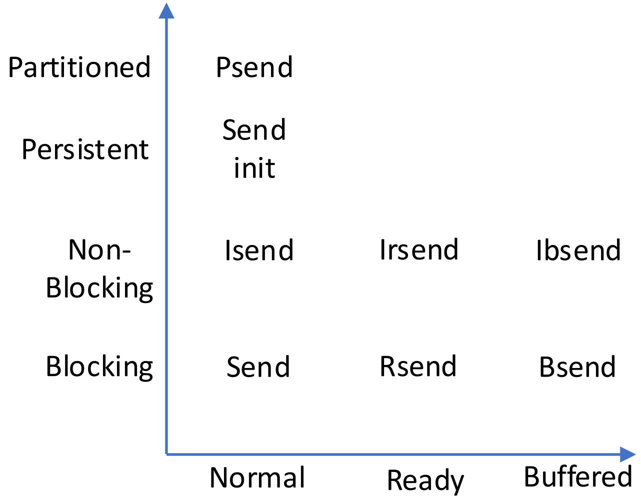
Integrating With Existing Objects & Semantics

Communication objects in MPI provide **context** for operations:

- **Communicators** provide process mapping & communication contexts
- **Windows** hold memory & contexts for RMA operations
- **Files** provide I/O context

Existing semantics cover all use-cases

- **Blocking**: FIFO start and completion
- **Nonblocking**: FIFO start, deferred completion
- **Persistent** & **Partitioned**: prior setup, FIFO start, deferred completion



	Normal	Ready	Buffered
Partitioned	Psend		
Persistent	Send init		
Non-Blocking	Isend	Irsend	Ibsend
Blocking	Send	Rsend	Bsend

Our Proposal

1. Associate stream with communicator/file/window.
2. Enqueue operations (blocking, nonblocking, start).
3. Enqueue wait if needed, potentially after enqueueing more work.
4. Synchronize stream (eventually).

Step 1: Associate Stream to Communicator

Analoguous for Files and Windows

```
MPIX_Comm_set_stream(MPI_Comm comm,
                     void* steam,
                     const char* kind,
                     MPI_Info info,
                     int* flag);
```

Stream passed via `void*` (e.g., hipStream_t*)

Stream type described as string (e.g., “hip”, “cuda”, “sycl”)

Flag returns 1 if MPI supports this type, 0 otherwise

Query stream (if previously associated)

```
MPIX_Comm_get_stream(MPI_Comm comm,
                     void* stream,
                     int* flag);
```

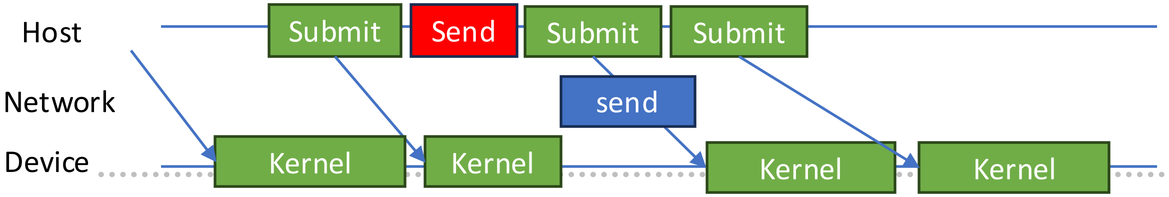
Step 2: Enqueue Operations

Blocking Operations

- Setup operation on stream (memory transfers / work descriptor / kernel launch)
- Operations will be pending on stream
- Prevent execution of subsequent operations

MPIX_Queue equivalence:

MPIX_Enqueue_send → MPIX_Enqueue_start → MPIX_Enqueue_wait

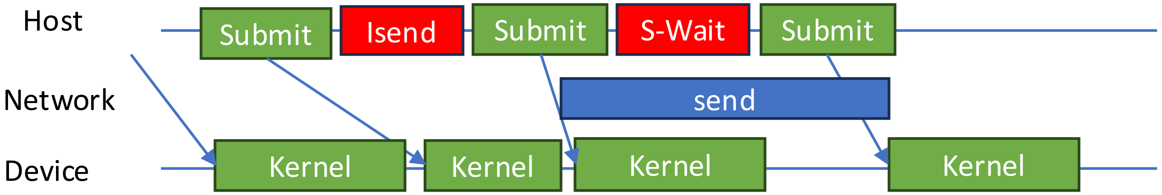


Switch/expand **Execution Space** of Communicator from Host to Device Stream

Nonblocking Operations

```
MPIX_Stream_wait(MPI_Request* request,
                 MPI_Status* status);
MPIX_Stream_waitall(int count,
                    MPI_Request request[],
                    MPI_Status status[]);
```

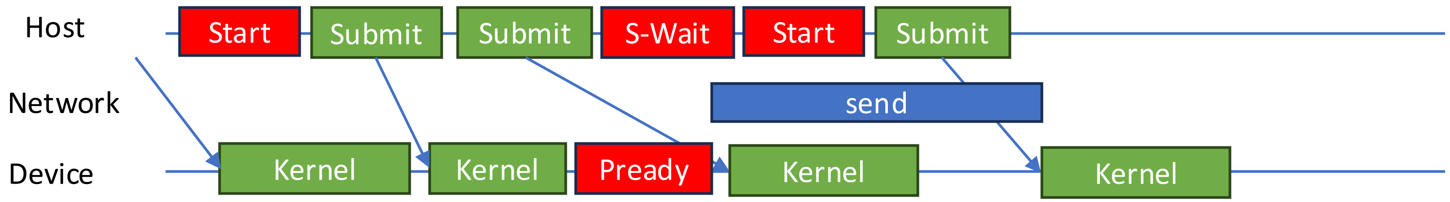
- Enqueue operation on stream and return immediately
- Request represents state of operation
- Stream associated with the resulting request
- Block stream to allow overlap or wait on the host for completion
- Stream-wait prevents execution of subsequent operations until completion



Persistent & Partitioned Operations

```
MPIX_Stream_wait(MPI_Request* request,
                 MPI_Status* status);
MPIX_Stream_waitall(int count,
                    MPI_Request request[],
                    MPI_Status status[]);
```

- Initialization binds operation to stream that is set on communicator
- MPI_Start enqueues operation start on stream
- Useful with partitioned operations to manage starting and completion



Step 3: Stream-Wait

```
MPIX_Stream_wait(MPI_Request* request,
                 MPI_Status* status);
MPIX_Stream_waitall(int count,
                    MPI_Request request[],
                    MPI_Status status[]);
```

Return ownership of non-persistent requests

Status(es) set before subsequent operations start

- Potentially in device memory (i.e., MPI implementation enqueues transfer)

Ensures that no subsequent operations on the associated streams execute before respective operations are complete

Does not block calling thread

Step 4: Ensuring Fair Progress for All

`MPIX_Comm_sync_stream(MPI_Comm comm);`

Synchronizing a stream (e.g., via `hipStreamSynchronize`()) may not provide sufficient progress for MPI operations

We may not have a request to poll on for progress in MPI

We do not want to force strong progress onto implementations

→ Need combined progress for device and MPI

MPIX_Comm_sync_stream blocks until stream is synchronized and all operations have completed

Example: Allocate, Compute, Send, Copy, Wait

Enqueue allocate

Enqueue compute

Associate stream

Enqueue send

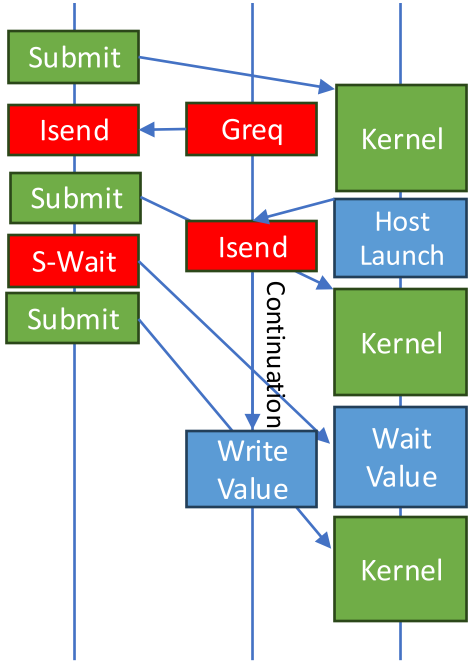
Enqueue copy

Enqueue free

Wait for completion

```
if (rank == 0) {
    hipMallocAsync((void**)&buf_dev, SIZE*sizeof(buf_dev[0]), stream);
    fill<<<dim3(SIZE/blockSize), dim3(blockSize), 0, stream>>>(buf_dev, SIZE);
    MPIX_Comm_set_stream(MPI_COMM_WORLD, "hip", &stream, MPI_INFO_NULL, &flag);
    if (!flag) {
        std::cout << "MPIX_Comm_set_stream failed to set the stream!" << std::endl;
        std::abort();
    }
    MPI_Isend(buf_dev, SIZE, MPI_INT, recv_rank, 0, MPI_COMM_WORLD, &req);
    hipMemcpyAsync(buf_host, buf_dev, SIZE*sizeof(buf_dev[0]), stream);
    MPIX_Stream_wait(&req, MPI_STATUS_IGNORE);
    hipFreeAsync(buf_dev, stream);
    MPIX_Comm_sync_stream(MPI_COMM_WORLD);
}
```

Implementation



Using PMPI interception for [Send|Recv]_init, Start, Isend, Irecv

Based on Open MPI branch with Continuations*

Generalized requests for user-facing requests

No kernel launch, HIP-based triggers

- Using `launchHostFunc` to start operations from the host
- Events polled from the host

Using `hipWriteValue` & `hipWaitValue` to facilitate stream synchronization

Optional progress thread support

Graph capturing support (if stream is capturing)

---

* Generic implementation in the works

Github: https://github.com/devreal/mpix-streams/

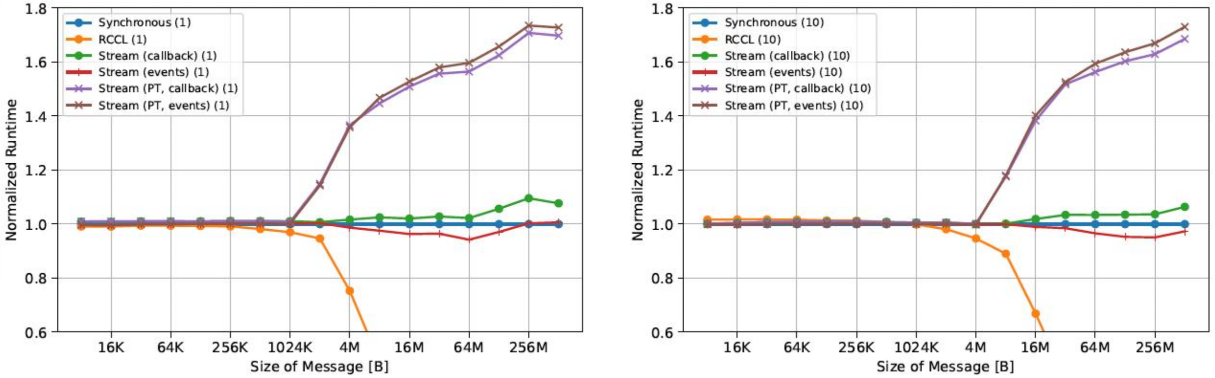
Results

Benchmark:
variable length kernel → variable size message → variable size kernel

Performance results are mixed bag, progress thread yields unsteady performance

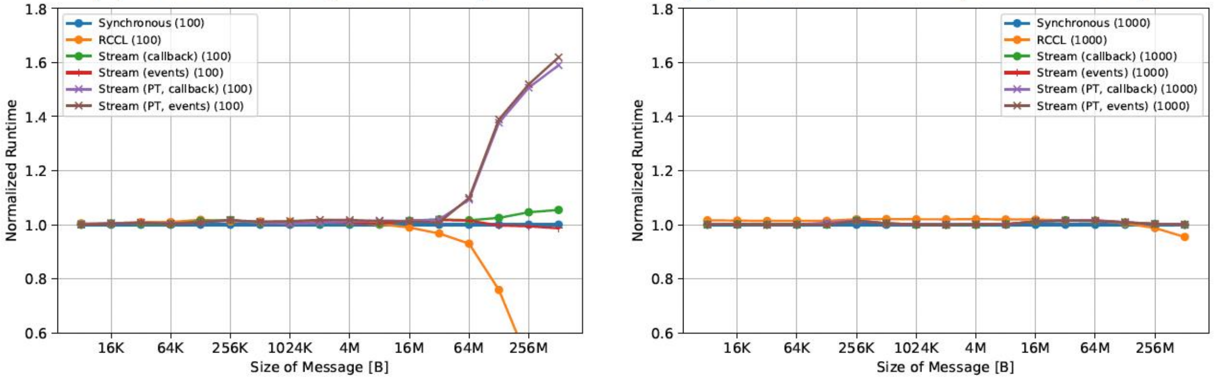
Focus on **functionality**, not optimized performance

RCCL benefits from communication kernel for larger messages



(a) Short kernel (1 iteration).

(b) Medium kernel (10 iterations).



(c) Medium kernel (100 iterations).

(d) Long kernel (1000 iterations).

Fig. 4: Normalized runtime of different implementations.

A benchmark suite with representative applications that enqueue communication on streams would help steer the design of stream integration in MPI.

Open Topics

1. May MPI operations synchronize two execution spaces at once?
   (i.e., may the calling thread block?)
2. Thread-specific binding of streams to communication objects
   (requesting thread-specific association)
3. Device-side triggering of operations inside a kernel scheduled on the stream
4. Stream-based communication benchmark suite (e.g., using KokkosComm)
5. Explicit graph API integration

Conclusions & Future Work

We can **reuse existing infrastructure** by associating streams with MPI objects

Extends existing semantics to compute streams (blocking, nonblocking, persistent, partitioned)

Requires **5 new MPI procedures** for stream association, stream-wait & stream-sync

Performance benefits are not clear but programmability benefits from integration

→ Benchmark suite for different stream integration approaches

Restart discussion in the Hybrid & Accelerator WG

Acknowledgements

This research was supported partly by NSF awards #1931347 and #1931384, and 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. We gratefully acknowledge the provision of computational resources by the Oak Ridge National Laboratory (ORNL) and the High-Performance Computing Center (HLRS) at the University of Stuttgart, Germany.

ECP EXASCALE COMPUTING PROJECT

NSF

OAK RIDGE National Laboratory

HLRS