Design and Implementation of MPI-Native GPU-Initiated MPI Partitioned Communication

Yıltan Hassan Temuçin[1], Whit Schonbein[2], Scott Levy[2], Amirhossein Sojoodi[1], Ryan E. Grant[1], and Ahmad Afsahi[1]

[1]ECE Department, Queen's University, Kingston, ON, Canada
[2]Sandia National Laboratories, Albuquerque, New Mexico, USA

ExaMPI Workshop 2024

November 17th, 2024

Outline

Introduction

- The Message Passing Interface (MPI) is a popular parallel programming model in HPC and AI

Introduction

- The Message Passing Interface (MPI) is a popular parallel programming model in HPC and AI
- Graphics Processing Units (GPUs) are commly used to offload computationally intensive application code

Introduction

- The Message Passing Interface (MPI) is a popular parallel programming model in HPC and AI
- Graphics Processing Units (GPUs) are commly used to offload computationally intensive application code
  - 9 of the top 10 supercomputers from the Top500 list utilize GPUs

Introduction

- The Message Passing Interface (MPI) is a popular parallel programming model in HPC and AI
- Graphics Processing Units (GPUs) are commly used to offload computationally intensive application code
  - 9 of the top 10 supercomputers from the Top500 list utilize GPUs
- A recent survey with developers involved in the U.S Department of Energy's Exascale Computing Project showed:
  - 80% plan to use GPUs
  - 43% would like to use the MPI within GPU kernels

Introduction

- The Message Passing Interface (MPI) is a popular parallel programming model in HPC and AI
- Graphics Processing Units (GPUs) are commly used to offload computationally intensive application code
  - 9 of the top 10 supercomputers from the Top500 list utilize GPUs
- A recent survey with developers involved in the U.S Department of Energy's Exascale Computing Project showed:
  - 80% plan to use GPUs
  - 43% would like to use the MPI within GPU kernels
- MPI Partitioned Point-to-Point Communication was added to the MPI-4.0 in June 2021

Introduction

- The Message Passing Interface (MPI) is a popular parallel programming model in HPC and AI
- Graphics Processing Units (GPUs) are commly used to offload computationally intensive application code
  - 9 of the top 10 supercomputers from the Top500 list utilize GPUs
- A recent survey with developers involved in the U.S Department of Energy's Exascale Computing Project showed:
  - 80% plan to use GPUs
  - 43% would like to use the MPI within GPU kernels
- MPI Partitioned Point-to-Point Communication was added to the MPI-4.0 in June 2021
  - Better supports hybrid programming models
  - Viable path to obtaining good GPU performance with MPI

Graphics Processing Units (GPUs)

- CUDA is used to program Nvidia GPUs

Graphics Processing Units (GPUs)

- CUDA is used to program Nvidia GPUs
- Nvidia GPUs consist of many streaming multiprocessors (SMs)
  - Kernels are launched on the GPU and executed on SMs

Graphics Processing Units (GPUs)

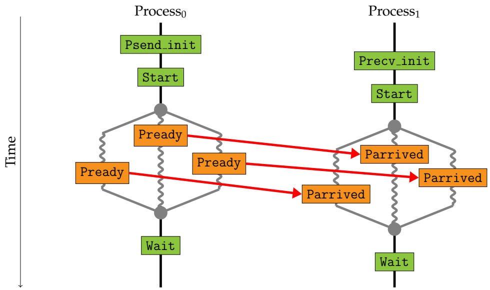
- CUDA is used to program Nvidia GPUs
- Nvidia GPUs consist of many streaming multiprocessors (SMs)
  - Kernels are launched on the GPU and executed on SMs
- Each kernel is placed on a CUDA stream
  - A stream can be thought as a First-In First-Out (FIFO) queue of operations that will be executed
  - Streams are in order with respect to themselves but work that is placed on multiple streams will not be in order relative to each other

Graphics Processing Units (GPUs)

- CUDA is used to program Nvidia GPUs
- Nvidia GPUs consist of many streaming multiprocessors (SMs)
  - Kernels are launched on the GPU and executed on SMs
- Each kernel is placed on a CUDA stream
  - A stream can be thought as a First-In First-Out (FIFO) queue of operations that will be executed
  - Streams are in order with respect to themselves but work that is placed on multiple streams will not be in order relative to each other
- MPI buffers used for GPU communication are located in GPU Memory

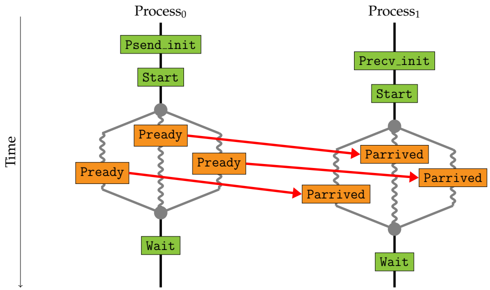
MPI Partitioned Communication

- `MPI_Psend_init`/`MPI_Precv_init` is used to initialize communication between processes



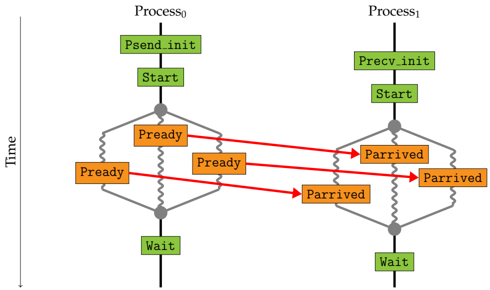
MPI Partitioned Communication

- `MPI_Psend_init/MPI_Precv_init` is used to initialize communication between processes
- `MPI_Start` is called to start communication



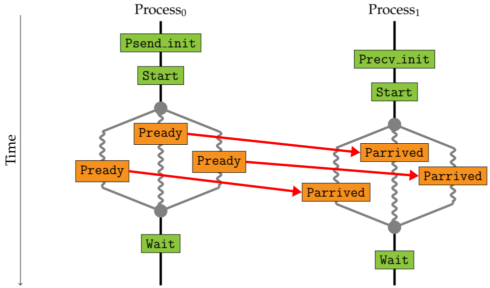
MPI Partitioned Communication

- `MPI_Psend_init/MPI_Precv_init` is used to initialize communication between processes
- `MPI_Start` is called to start communication
- A parallel region begins



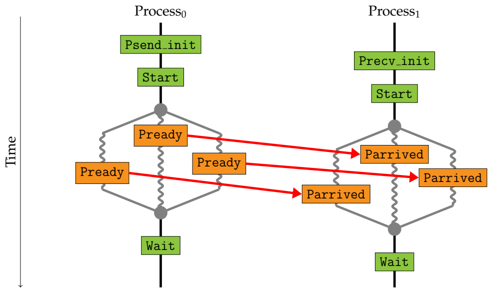
MPI Partitioned Communication

- `MPI_Psend_init`/`MPI_Precv_init` is used to initialize communication between processes
- `MPI_Start` is called to start communication
- A parallel region begins



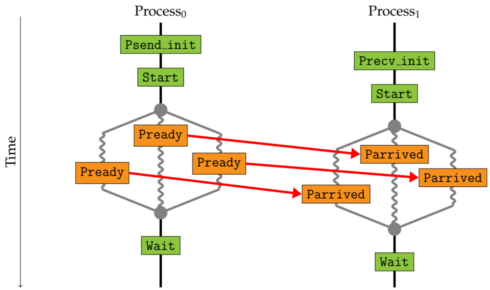
MPI Partitioned Communication

- `MPI_Psend_init/MPI_Precv_init` is used to initialize communication between processes
- `MPI_Start` is called to start communication
- A parallel region begins
  - Work is Computed



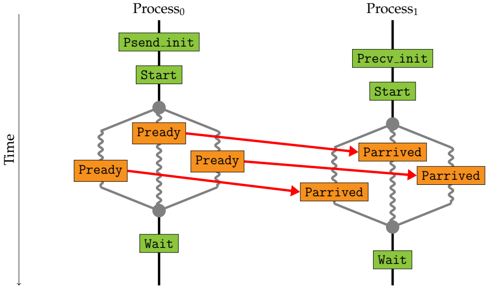
MPI Partitioned Communication

- `MPI_Psend_init`/`MPI_Precv_init` is used to initialize communication between processes
- `MPI_Start` is called to start communication
- A parallel region begins
  - Work is Computed
  - Once data is ready, `MPI_Pready` is called



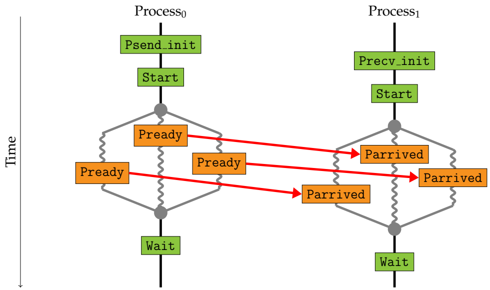
MPI Partitioned Communication

- `MPI_Psend_init/MPI_Precv_init` is used to initialize communication between processes
- `MPI_Start` is called to start communication
- A parallel region begins
  - Work is Computed
  - Once data is ready, `MPI_Pready` is called
  - Optionally, `MPI_Parrived` to check if incoming data has arrived



MPI Partitioned Communication

- `MPI_Psend_init`/`MPI_Precv_init` is used to initialize communication between processes
- `MPI_Start` is called to start communication
- A parallel region begins
  - Work is Computed
  - Once data is ready, `MPI_Pready` is called
  - Optionally, `MPI_Parrived` to check if incoming data has arrived
- `MPI_Wait` is called to complete communication



GPU Support for MPI Communication

- How does the current MPI+CUDA model communicate?

GPU Support for MPI Communication

- How does the current MPI+CUDA model communicate?

```
kernel_A<<<stream>>>(sbuf);
cudaStreamSynchronize(stream);
MPI_Send(sbuf);
```

GPU Support for MPI Communication

- How does the current MPI+CUDA model communicate?
- Can be expensive as we must explicitly syncronize before communication

```
kernel_A<<<stream>>>(sbuf);
cudaStreamSynchronize(stream);
MPI_Send(sbuf);
```

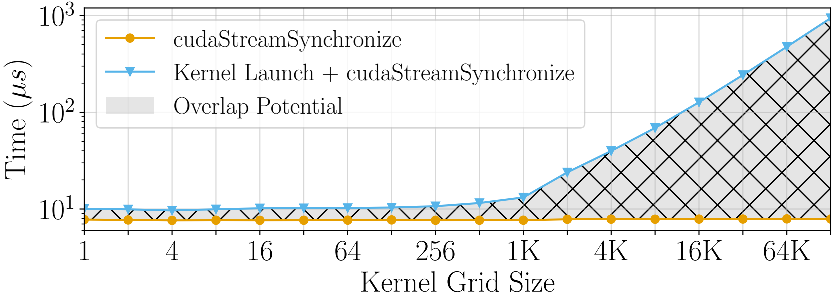
GPU Support for MPI Communication

- How does the current MPI+CUDA model communicate?
- Can be expensive as we must explicitly syncronize before communication
- How can we avoid `cudaStreamSynchronize`?

```
kernel_A<<<stream>>>(sbuf);
cudaStreamSynchronize(stream);
MPI_Send(sbuf);
```

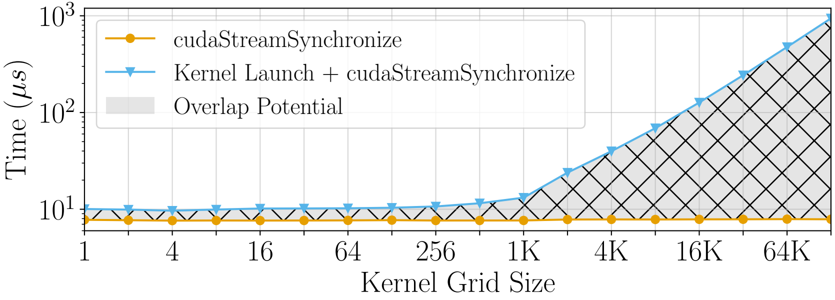
GPU Synchronization Cost

- We evaluate the cost of syncronization for a vector addition ($C = A + B$)



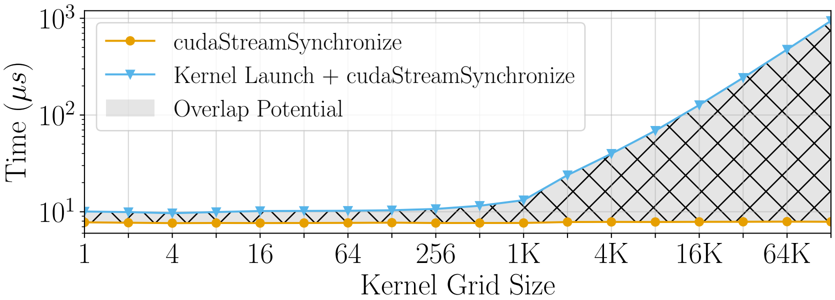
GPU Synchronization Cost

- We evaluate the cost of syncronization for a vector addition ($C = A + B$)
- For smaller kernels, the synchronization cost is anywhere between 71.6-78.9% of the total time to execute a kernel.



GPU Synchronization Cost

- We evaluate the cost of syncronization for a vector addition ($C = A + B$)
- For smaller kernels, the synchronization cost is anywhere between 71.6-78.9% of the total time to execute a kernel.
- For large kernels, the host is idle for 99.2% of its execution



GPU-Support MPI Partitioned Communication

- MPI Hybrid Working Group has proposed the following calls for MPI Partitioned:

GPU-Support MPI Partitioned Communication

- MPI Hybrid Working Group has proposed the following calls for MPI Partitioned:
- `MPIX_Pbuf_Prepare(MPI_Request request)`

GPU-Support MPI Partitioned Communication

- MPI Hybrid Working Group has proposed the following calls for MPI Partitioned:
- `MPIX_Pbuf_Prepare(MPI_Request request)`
  - Synchronize processes associated with the request

GPU-Support MPI Partitioned Communication

- MPI Hybrid Working Group has proposed the following calls for MPI Partitioned:
- `MPIX_Pbuf_Prepare(MPI_Request request)`
  - Synchronize processes associated with the request
  - Provides remote buffer guarantees

GPU-Support MPI Partitioned Communication

- MPI Hybrid Working Group has proposed the following calls for MPI Partitioned:
- `MPIX_Pbuf_Prepare(MPI_Request request)`
  - Synchronize processes associated with the request
  - Provides remote buffer guarantees
- `MPIX_Prequest_create(MPI_Prequest preq, MPI_Request req)`

GPU-Support MPI Partitioned Communication

- MPI Hybrid Working Group has proposed the following calls for MPI Partitioned:
- `MPIX_Pbuf_Prepare(MPI_Request request)`
  - Synchronize processes associated with the request
  - Provides remote buffer guarantees
- `MPIX_Prequest_create(MPI_Prequest preq, MPI_Request req)`
  - Exports a request object that can be accessed by a device

GPU-Support MPI Partitioned Communication

- MPI Hybrid Working Group has proposed the following calls for MPI Partitioned:
- `MPIX_Pbuf_Prepare(MPI_Request request)`
  - Synchronize processes associated with the request
  - Provides remote buffer guarantees
- `MPIX_Prequest_create(MPI_Prequest preq, MPI_Request req)`
  - Exports a request object that can be accessed by a device
- `MPIX_Pready(int partition, MPI_Prequest preq)`

GPU-Support MPI Partitioned Communication

- MPI Hybrid Working Group has proposed the following calls for MPI Partitioned:
- `MPIX_Pbuf_Prepare(MPI_Request request)`
  - Synchronize processes associated with the request
  - Provides remote buffer guarantees
- `MPIX_Prequest_create(MPI_Prequest preq, MPI_Request req)`
  - Exports a request object that can be accessed by a device
- `MPIX_Pready(int partition, MPI_Prequest preq)`
  - A device callable `MPI_Pready`

GPU-Support MPI Partitioned Communication

- MPI Hybrid Working Group has proposed the following calls for MPI Partitioned:
- `MPIX_Pbuf_Prepare(MPI_Request request)`
  - Synchronize processes associated with the request
  - Provides remote buffer guarantees
- `MPIX_Prequest_create(MPI_Prequest preq, MPI_Request req)`
  - Exports a request object that can be accessed by a device
- `MPIX_Pready(int partition, MPI_Prequest preq)`
  - A device callable `MPI_Pready`
- How do we use these API calls?

GPU-Support MPI Partitioned Communication

- We start our communication

```
__host__ int host_function(MPI_Request req,
                           double *sbuf)
{
  MPI_Start(req)
  MPIX_Pbuf_Prepare(req);
  MPIX_Prequest_create(preq, req);

  kernel_B<<<stream>>>(preq, sbuf);
  /* Do work on host */
  MPI_Wait(req);
}

__global__ int kernel_B(MPIX_Prequest preq,
                        double *sbuf)
{
  /* Do Work */
  MPIX_Pready(idx, preq);
}
```

GPU-Support MPI Partitioned Communication

- We start our communication
- Wait for the remote buffer to be ready

```
__host__ int host_function(MPI_Request req,
                           double *sbuf)
{
  MPI_Start(req)
  MPIX_Pbuf_Prepare(req);
  MPIX_Prequest_create(preq, req);

  kernel_B<<<stream>>>(preq, sbuf);
  /* Do work on host */
  MPI_Wait(req);
}

__global__ int kernel_B(MPIX_Prequest preq,
                        double *sbuf)
{
  /* Do Work */
  MPIX_Pready(idx, preq);
}
```

GPU-Support MPI Partitioned Communication

- We start our communication
- Wait for the remote buffer to be ready
- Export request if a device request does not exist

```
__host__ int host_function(MPI_Request req,
                           double *sbuf)
{
  MPI_Start(req)
  MPIX_Pbuf_Prepare(req);
  MPIX_Prequest_create(preq, req);

  kernel_B<<<stream>>>(preq, sbuf);
  /* Do work on host */
  MPI_Wait(req);
}

__global__ int kernel_B(MPIX_Prequest preq,
                        double *sbuf)
{
  /* Do Work */
  MPIX_Pready(idx, preq);
}
```

GPU-Support MPI Partitioned Communication

- We start our communication
- Wait for the remote buffer to be ready
- Export request if a device request does not exist
- Launch kernel

```
__host__ int host_function(MPI_Request req,
                           double *sbuf)
{
  MPI_Start(req)
  MPIX_Pbuf_Prepare(req);
  MPIX_Prequest_create(preq, req);

  kernel_B<<<stream>>>(preq, sbuf);
  /* Do work on host */
  MPI_Wait(req);
}

__global__ int kernel_B(MPIX_Prequest preq,
                        double *sbuf)
{
  /* Do Work */
  MPIX_Pready(idx, preq);
}
```

GPU-Support MPI Partitioned Communication

- We start our communication
- Wait for the remote buffer to be ready
- Export request if a device request does not exist
- Launch kernel
- Communicate within kernel

```
__host__ int host_function(MPI_Request req,
                           double *sbuf)
{
  MPI_Start(req)
  MPIX_Pbuf_Prepare(req);
  MPIX_Prequest_create(preq, req);

  kernel_B<<<stream>>>(preq, sbuf);
  /* Do work on host */
  MPI_Wait(req);
}

__global__ int kernel_B(MPIX_Prequest preq,
                        double *sbuf)
{
  /* Do Work */
  MPIX_Pready(idx, preq);
}
```

GPU-Support MPI Partitioned Communication

- We start our communication
- Wait for the remote buffer to be ready
- Export request if a device request does not exist
- Launch kernel
- Communicate within kernel
- Wait for communication to complete

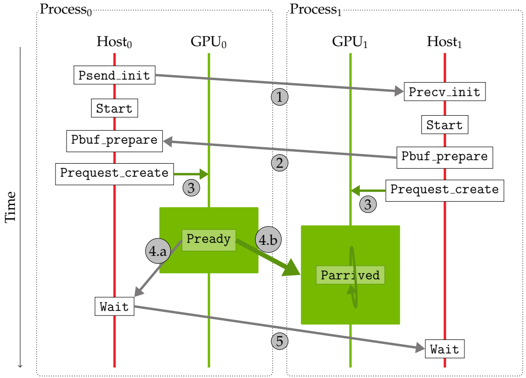
```
__host__ int host_function(MPI_Request req,
                           double *sbuf)
{
  MPI_Start(req)
  MPIX_Pbuf_Prepare(req);
  MPIX_Prequest_create(preq, req);

  kernel_B<<<stream>>>(preq, sbuf);
  /* Do work on host */
  MPI_Wait(req);
}

__global__ int kernel_B(MPIX_Prequest preq,
                        double *sbuf)
{
  /* Do Work */
  MPIX_Pready(idx, preq);
}
```

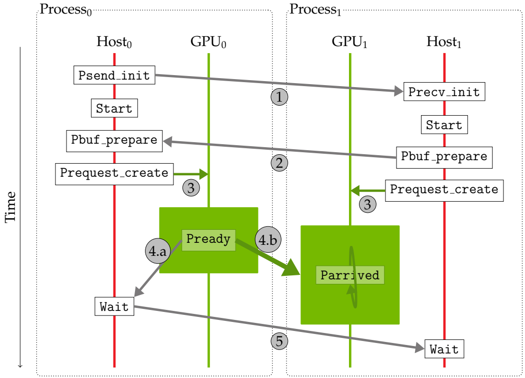
GPU-Initiated MPI Partitioned Communication Design

1. We initialize our UCX context, endpoints and workers
   - Exchange their addresses



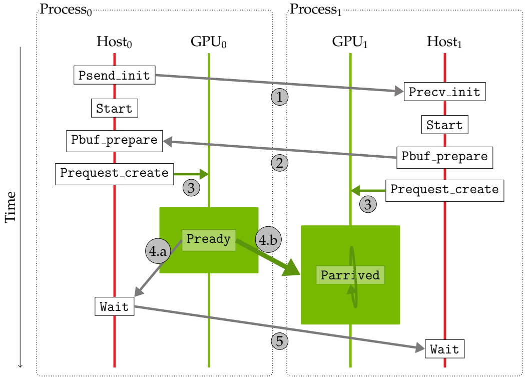
GPU-Initiated MPI Partitioned Communication Design

1. We initialize our UCX context, endpoints and workers
   - Exchange their addresses
2. We send an acknowledgement to P_0 once the remote buffer is ready



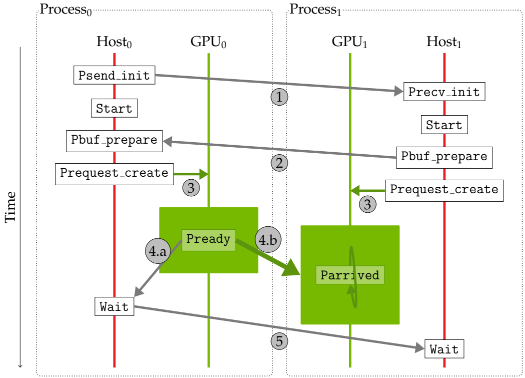
GPU-Initiated MPI Partitioned Communication Design

1. We initialize our UCX context, endpoints and workers
   - Exchange their addresses
2. We send an acknowledgement to P_0 once the remote buffer is ready
3. Allocate `MPI_Prequest` object in GPU memory
   - Copy type
   - Aggregation counters
   - Pointers to flags in host memory



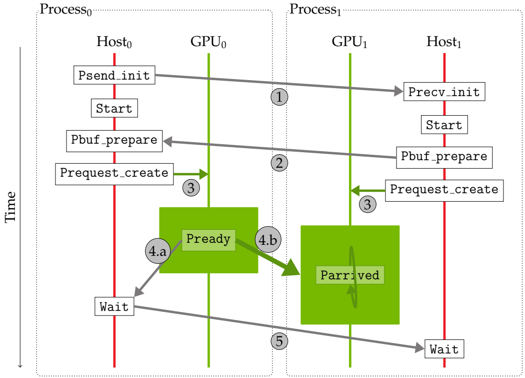
GPU-Initiated MPI Partitioned Communication Design

1. We initialize our UCX context, endpoints and workers
   - Exchange their addresses
2. We send an acknowledgement to P_0 once the remote buffer is ready
3. Allocate `MPI_Prequest` object in GPU memory
   - Copy type
   - Aggregation counters
   - Pointers to flags in host memory
4. `MPIX_Pready` increments counters in GPU memory
   a. Write to host flags if threshold met
   b. Moves data, using a kernel copy or MPI progression engine



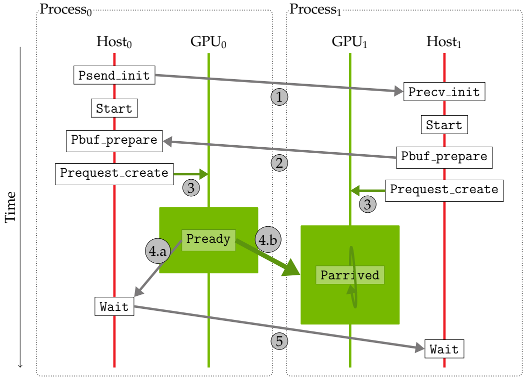
GPU-Initiated MPI Partitioned Communication Design

1. We initialize our UCX context, endpoints and workers
   - Exchange their addresses
2. We send an acknowledgement to P_0 once the remote buffer is ready
3. Allocate `MPI_Prequest` object in GPU memory
   - Copy type
   - Aggregation counters
   - Pointers to flags in host memory
4. `MPIX_Pready` increments counters in GPU memory
   a. Write to host flags if threshold met
   b. Moves data, using a kernel copy or MPI progression engine



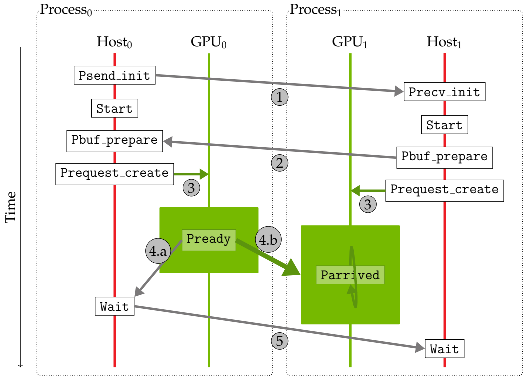
GPU-Initiated MPI Partitioned Communication Design

1. We initialize our UCX context, endpoints and workers
   - Exchange their addresses
2. We send an acknowledgement to P_0 once the remote buffer is ready
3. Allocate `MPI_Prequest` object in GPU memory
   - Copy type
   - Aggregation counters
   - Pointers to flags in host memory
4. `MPIX_Pready` increments counters in GPU memory
   a. Write to host flags if threshold met
   b. Moves data, using a kernel copy or MPI progression engine



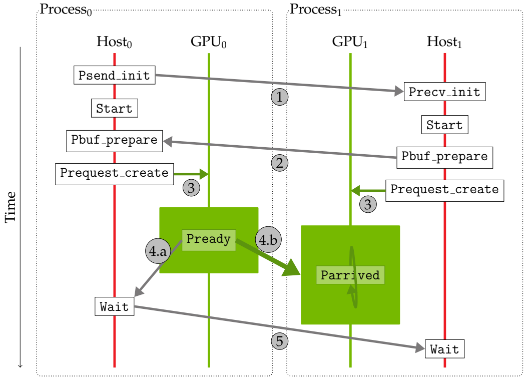
GPU-Initiated MPI Partitioned Communication Design

1. We initialize our UCX context, endpoints and workers
   - Exchange their addresses
2. We send an acknowledgement to P_0 once the remote buffer is ready
3. Allocate `MPI_Prequest` object in GPU memory
   - Copy type
   - Aggregation counters
   - Pointers to flags in host memory
4. `MPIX_Pready` increments counters in GPU memory
   a. Write to host flags if threshold met
   b. Moves data, using a kernel copy or MPI progression engine



GPU-Initiated MPI Partitioned Communication Design

1. We initialize our UCX context, endpoints and workers
   - Exchange their addresses
2. We send an acknowledgement to P_0 once the remote buffer is ready
3. Allocate MPI_Prequest object in GPU memory
   - Copy type
   - Aggregation counters
   - Pointers to flags in host memory
4. MPIX_Pready increments counters in GPU memory
   a. Write to host flags if threshold met
   b. Moves data, using a kernel copy or MPI progression engine
5. MPI_Wait is called to complete communication



GPU-Initiated MPI Partitioned Communication Design

1. We have two copy types which we evaluate

GPU-Initiated MPI Partitioned Communication Design

1. We have two copy types which we evaluate
2. Kernel Copy
   - We implemented `ucp_rkey_ptr` call for CUDA UCT

GPU-Initiated MPI Partitioned Communication Design

1. We have two copy types which we evaluate
2. Kernel Copy
   - We implemented `ucp_rkey_ptr` call for CUDA UCT
   - This API call provides a pointer to a memory location allocated by the remote process

GPU-Initiated MPI Partitioned Communication Design

1. We have two copy types which we evaluate
2. Kernel Copy
   - We implemented `ucp_rkey_ptr` call for CUDA UCT
   - This API call provides a pointer to a memory location allocated by the remote process
   - We store data in the remote process inside the CUDA kernel

GPU-Initiated MPI Partitioned Communication Design

1. We have two copy types which we evaluate
2. Kernel Copy
   - We implemented `ucp_rkey_ptr` call for CUDA UCT
   - This API call provides a pointer to a memory location allocated by the remote process
   - We store data in the remote process inside the CUDA kernel
3. Progression Engine Copy

GPU-Initiated MPI Partitioned Communication Design

1. We have two copy types which we evaluate
2. Kernel Copy
   - We implemented `ucp_rkey_ptr` call for CUDA UCT
   - This API call provides a pointer to a memory location allocated by the remote process
   - We store data in the remote process inside the CUDA kernel
3. Progression Engine Copy
   - The GPU writes to a flag in host memory

GPU-Initiated MPI Partitioned Communication Design

1. We have two copy types which we evaluate
2. Kernel Copy
   - We implemented `ucp_rkey_ptr` call for CUDA UCT
   - This API call provides a pointer to a memory location allocated by the remote process
   - We store data in the remote process inside the CUDA kernel
3. Progression Engine Copy
   - The GPU writes to a flag in host memory
   - The MPI Progression polls the flags and issues a `ucp_put_nbx` when serviced

MPI Partitioned Collectives

- Collectives are the natural extension to MPI Partitioned Point-to-Point

MPI Partitioned Collectives

- Collectives are the natural extension to MPI Partitioned Point-to-Point
  - Similar semantics

MPI Partitioned Collectives

- Collectives are the natural extension to MPI Partitioned Point-to-Point
  - Similar semantics
- Initialization calls for each collective

MPI Partitioned Collectives

- Collectives are the natural extension to MPI Partitioned Point-to-Point
  - Similar semantics
- Initialization calls for each collective
  - `MPIX_Pbcast_init`
  - `MPIX_Pallreduce_init`
  - ect.

MPI Partitioned Collectives

- Collectives are the natural extension to MPI Partitioned Point-to-Point
  - Similar semantics
- Initialization calls for each collective
  - `MPIX_Pbcast_init`
  - `MPIX_Pallreduce_init`
  - ect.
- Ideally we could generalize initialization for the 21 proposed MPI Partitioned Collective calls

MPI Partitioned Collectives

- Collectives are the natural extension to MPI Partitioned Point-to-Point
  - Similar semantics
- Initialization calls for each collective
  - `MPIX_Pbcast_init`
  - `MPIX_Pallreduce_init`
  - ect.
- Ideally we could generalize initialization for the 21 proposed MPI Partitioned Collective calls
- `MPI_Pbuf_prepare` now gaurantees readiness for all processes in communicator

MPI Partitioned Collectives Design

- We first create a schedule our partitioned collective will run

---

Algorithm 1: `MPIX_Pallreduce_init` schedule creation for a Ring-Based RSA algorithm

---

for $i \leftarrow 0$ **to** $2(P-1)$ **do**
- $I \leftarrow (\text{rank} - 1) \mod P$
- $O \leftarrow (\text{rank} + 1) \mod P$
- $R \leftarrow (\text{rank} + 2P - i) \mod P$
- $A \leftarrow (\text{rank} + 2P - i - 1) \mod P$
- **if** $i < (P-1)$ **then**
  - $\oplus \leftarrow MPI\_Op$
- **else**
  - $\oplus \leftarrow NOP$
- $S_i \leftarrow (I, R, \oplus, O, A)$
- $\mathbb{S} \leftarrow S_i$

MPI Partitioned Collectives Design

- We first create a schedule our partitioned collective will run
- We list our incomming neighbours (I)

Algorithm 2: `MPIX_Pallreduce_init` schedule creation for a Ring-Based RSA algorithm

for $i \leftarrow 0$ **to** $2(P-1)$ **do**
    $I \leftarrow (\text{rank} - 1) \mod P$
    $O \leftarrow (\text{rank} + 1) \mod P$
    $R \leftarrow (\text{rank} + 2P - i) \mod P$
    $A \leftarrow (\text{rank} + 2P - i - 1) \mod P$
    if $i < (P-1)$ **then**
        $\oplus \leftarrow MPI\_Op$
    else
        $\oplus \leftarrow NOP$
    $S_i \leftarrow (I, R, \oplus, O, A)$
    $\mathbb{S} \leftarrow S_i$

MPI Partitioned Collectives Design

- We first create a schedule our partitioned collective will run
- We list our incomming neighbours (I)
- We list our outgoing neighbours (O)

Algorithm 3: `MPIX_Pallreduce_init` schedule creation for a Ring-Based RSA algorithm

for $i \leftarrow 0$ **to** $2(P-1)$ **do**

  $I \leftarrow (\text{rank} - 1) \mod P$

  $O \leftarrow (\text{rank} + 1) \mod P$

  $R \leftarrow (\text{rank} + 2P - i) \mod P$

  $A \leftarrow (\text{rank} + 2P - i - 1) \mod P$

  if $i < (P-1)$ **then**

    $\oplus \leftarrow MPI\_Op$

  else

    $\oplus \leftarrow NOP$

  $S_i \leftarrow (I, R, \oplus, O, A)$

  $\mathbb{S} \leftarrow S_i$

MPI Partitioned Collectives Design

- We first create a schedule our partitioned collective will run
- We list our incomming neighbours (I)
- We list our outgoing neighbours (O)
- We list our offsets (R, A)

Algorithm 4: `MPIX_Pallreduce_init` schedule creation for a Ring-Based RSA algorithm

for $i \leftarrow 0$ **to** $2(P-1)$ **do**
  $I \leftarrow (\text{rank}-1) \mod P$
  $O \leftarrow (\text{rank}+1) \mod P$
  $R \leftarrow (\text{rank}+2P-i) \mod P$
  $A \leftarrow (\text{rank}+2P-i-1) \mod P$
  if $i < (P-1)$ **then**
    $\oplus \leftarrow MPI\_Op$
  else
    $\oplus \leftarrow NOP$
  $S_i \leftarrow (I, R, \oplus, O, A)$
  $\mathbb{S} \leftarrow S_i$

MPI Partitioned Collectives Design

- We first create a schedule our partitioned collective will run
- We list our incomming neighbours (I)
- We list our outgoing neighbours (O)
- We list our offsets (R, A)
  - Certain algorithms move data as a part of its execution

Algorithm 5: `MPIX_Pallreduce_init` schedule creation for a Ring-Based RSA algorithm

for $i \leftarrow 0$ **to** $2(P-1)$ **do**
$\quad I \leftarrow (\text{rank} - 1) \mod P$
$\quad O \leftarrow (\text{rank} + 1) \mod P$
$\quad R \leftarrow (\text{rank} + 2P - i) \mod P$
$\quad A \leftarrow (\text{rank} + 2P - i - 1) \mod P$
$\quad$ **if** $i < (P-1)$ **then**
$\quad\quad \oplus \leftarrow MPI\_Op$
$\quad$ **else**
$\quad\quad \oplus \leftarrow NOP$
$\quad S_i \leftarrow (I, R, \oplus, O, A)$
$\quad \mathbb{S} \leftarrow S_i$

MPI Partitioned Collectives Design

- We first create a schedule our partitioned collective will run
- We list our incomming neighbours (I)
- We list our outgoing neighbours (O)
- We list our offsets (R, A)
  - Certain algorithms move data as a part of its execution
- Some collectives a reduce component (allreduce, reducescatter, scan, etc.)

Algorithm 6: `MPIX_Pallreduce_init` schedule creation for a Ring-Based RSA algorithm

for $i \leftarrow 0$ **to** $2(P-1)$ **do**
- $I \leftarrow (\text{rank} - 1) \mod P$
- $O \leftarrow (\text{rank} + 1) \mod P$
- $R \leftarrow (\text{rank} + 2P - i) \mod P$
- $A \leftarrow (\text{rank} + 2P - i - 1) \mod P$
- **if** $i < (P-1)$ **then**
  - $\oplus \leftarrow MPI\_Op$
- **else**
  - $\oplus \leftarrow NOP$
- $S_i \leftarrow (I, R, \oplus, O, A)$
- $\mathbb{S} \leftarrow S_i$

MPI Partitioned Collectives Design

- We first create a schedule our partitioned collective will run
- We list our incomming neighbours (I)
- We list our outgoing neighbours (O)
- We list our offsets (R, A)
  - Certain algorithms move data as a part of its execution
- Some collectives a reduce component (allreduce, reducescatter, scan, etc.)
- These parameters populate our schedule $\mathbb{S}$

Algorithm 7: `MPIX_Pallreduce_init` schedule creation for a Ring-Based RSA algorithm

for $i \leftarrow 0$ **to** $2(P-1)$ **do**
    $I \leftarrow (\text{rank} - 1) \mod P$
    $O \leftarrow (\text{rank} + 1) \mod P$
    $R \leftarrow (\text{rank} + 2P - i) \mod P$
    $A \leftarrow (\text{rank} + 2P - i - 1) \mod P$
    if $i < (P-1)$ **then**
        $\oplus \leftarrow MPI\_Op$
    else
        $\oplus \leftarrow NOP$
    $S_i \leftarrow (I, R, \oplus, O, A)$
    $\mathbb{S} \leftarrow S_i$

MPI Partitioned Collectives Design

- We check if all partitions for the algorithm step have completed

Algorithm 8: `MPI_Wait` Progression of a Partitioned Collective Schedule

for *part* $\leftarrow$ 0 **to** num_partitions **do**
    state = states[part]
    $S \leftarrow$ state.step
    if $S > S_k$ **then** continue ;
    if state.parrived_complete $\neq |I|$ **then**
        for $I_x \in I$ **do**
            `MPI_Parrived(flag)`
            if flag = true **then**
                state.parrived_complete++
                if $\oplus \neq$ NOP **then** reduceData() ;
    if state.parrived_complete $= |I|$ **and** state.pready_complete $= |O|$ **then**
        $S \leftarrow S_{(i+1)}$
        state.parrived_complete $\leftarrow 0$
        state.pready_complete $\leftarrow 0$
    if $S \neq S_0$ **and** $S! = S_k$ **and** state.pready_complete $= 0$ **then**
        for $O_x \in O$ **do**
            `MPI_Pready(...)`
            state.pready_complete++

MPI Partitioned Collectives Design

- We check if all partitions for the algorithm step have completed
  - If not, we check with `MPI_Parrived`

Algorithm 9: `MPI_Wait` Progression of a Partitioned Collective Schedule

for *part* $\leftarrow$ 0 **to** num_partitions **do**
    state = states[part]
    $S \leftarrow$ state.step
    if $S > S_k$ **then** continue ;
    if state.parrived_complete $\neq |I|$ **then**
        for $I_x \in I$ **do**
            `MPI_Parrived(flag)`
            if flag = true **then**
                state.parrived_complete++
                if $\oplus \neq$ NOP **then** reduceData() ;
    if state.parrived_complete $= |I|$ **and** state.pready_complete $= |O|$ **then**
        $S \leftarrow S_{(i+1)}$
        state.parrived_complete $\leftarrow 0$
        state.pready_complete $\leftarrow 0$
    if $S \neq S_0$ **and** $S! = S_k$ **and** state.pready_complete $= 0$ **then**
        for $O_x \in O$ **do**
            `MPI_Pready(...)`
            state.pready_complete++

MPI Partitioned Collectives Design

- We check if all partitions for the algorithm step have completed
  - If not, we check with `MPI_Parrived`
  - If it has arrived, and there is a reduce operation $\oplus$, we reduce the data.

Algorithm 10: `MPI_Wait` Progression of a Partitioned Collective Schedule

for *part* $\leftarrow$ 0 **to** num_partitions **do**
  state = states[part]
  $S \leftarrow$ state.step
  if $S > S_k$ **then** continue ;
  if state.parrived_complete $\neq |I|$ **then**
    for $I_x \in I$ **do**
      `MPI_Parrived(flag)`
      if flag = true **then**
        state.parrived_complete++
        if $\oplus \neq$ NOP **then** reduceData() ;
  if state.parrived_complete $= |I|$ **and**
    state.pready_complete $= |O|$
  then
    $S \leftarrow S_{(i+1)}$
    state.parrived_complete $\leftarrow 0$
    state.pready_complete $\leftarrow 0$
  if $S \neq S_0$ **and** $S! = S_k$ **and**
    state.pready_complete $= 0$
  then
    for $O_x \in O$ **do**
      `MPI_Pready(...)`
      state.pready_complete++

MPI Partitioned Collectives Design

- We check if all partitions for the algorithm step have completed
  - If not, we check with `MPI_Parrived`
  - If it has arrived, and there is a reduce operation $\oplus$, we reduce the data.
- If all communication is complete, we move to the next step in the schedule

Algorithm 11: `MPI_Wait` Progression of a Partitioned Collective Schedule

```
for part ← 0 to num_partitions do
    state = states[part]
    S ← state.step
    if S > S_k then continue ;
    if state.parrived_complete ≠ |I| then
        for I_x ∈ I do
            MPI_Parrived(flag)
            if flag = true then
                state.parrived_complete++
                if ⊕ ≠ NOP then
                    reduceData() ;
    if state.parrived_complete = |I| and
        state.pready_complete = |O|
    then
        S ← S_(i+1)
        state.parrived_complete ← 0
        state.pready_complete ← 0
    if S ≠ S_0 and S! = S_k and
        state.pready_complete = 0
    then
        for O_x ∈ O do
            MPI_Pready(...)
            state.pready_complete++
```

MPI Partitioned Collectives Design

- We check if all partitions for the algorithm step have completed
  - If not, we check with `MPI_Parrived`
  - If it has arrived, and there is a reduce operation $\oplus$, we reduce the data.
- If all communication is complete, we move to the next step in the schedule
- Then we call `MPI_Pready` for our outgoing neighbors

Algorithm 12: `MPI_Wait` Progression of a Partitioned Collective Schedule

for *part* $\leftarrow 0$ **to** num_partitions **do**
  state = states[part]
  $S \leftarrow$ state.step
  if $S > S_k$ **then** continue ;
  if state.parrived_complete $\neq |I|$ **then**
    for $I_x \in I$ **do**
      `MPI_Parrived(flag)`
      if flag = true **then**
        state.parrived_complete++
        if $\oplus \neq$ NOP **then** reduceData() ;
  if state.parrived_complete $= |I|$ **and** state.pready_complete $= |O|$ **then**
    $S \leftarrow S_{(i+1)}$
    state.parrived_complete $\leftarrow 0$
    state.pready_complete $\leftarrow 0$
  if $S \neq S_0$ **and** $S! = S_k$ **and** state.pready_complete $= 0$ **then**
    for $O_x \in O$ **do**
      `MPI_Pready(...)`
      state.pready_complete++

Experimental Setup

- We used a two-node NVIDIA GH200 Grace Hopper Superchip testbed*

*We would like to thank Nvidia for these resources

Experimental Setup

- We used a two-node NVIDIA GH200 Grace Hopper Superchip testbed[*]
  - NVIDIA Grace CPU with 72 ARM Neoverse V2 CPU cores with 120GB of LPDDR5X memory

[*]We would like to thank Nvidia for these resources

Experimental Setup

- We used a two-node NVIDIA GH200 Grace Hopper Superchip testbed*
  - NVIDIA Grace CPU with 72 ARM Neoverse V2 CPU cores with 120GB of LPDDR5X memory
  - NVIDIA Hopper GPU that has 96GB of HBM3 memory

*We would like to thank Nvidia for these resources

Experimental Setup

- We used a two-node NVIDIA GH200 Grace Hopper Superchip testbed*
  - NVIDIA Grace CPU with 72 ARM Neoverse V2 CPU cores with 120GB of LPDDR5X memory
  - NVIDIA Hopper GPU that has 96GB of HBM3 memory
- Each node had four GH200s

*We would like to thank Nvidia for these resources

Experimental Setup

- We used a two-node NVIDIA GH200 Grace Hopper Superchip testbed*
  - NVIDIA Grace CPU with 72 ARM Neoverse V2 CPU cores with 120GB of LPDDR5X memory
  - NVIDIA Hopper GPU that has 96GB of HBM3 memory
- Each node had four GH200s
  - Connected by 18 NVLinks, four links per device

*We would like to thank Nvidia for these resources

Experimental Setup

- We used a two-node NVIDIA GH200 Grace Hopper Superchip testbed*
  - NVIDIA Grace CPU with 72 ARM Neoverse V2 CPU cores with 120GB of LPDDR5X memory
  - NVIDIA Hopper GPU that has 96GB of HBM3 memory
- Each node had four GH200s
  - Connected by 18 NVLinks, four links per device
  - Four Mellanox ConnectX-7 network cards

*We would like to thank Nvidia for these resources

Experimental Setup

- We used a two-node NVIDIA GH200 Grace Hopper Superchip testbed*
  - NVIDIA Grace CPU with 72 ARM Neoverse V2 CPU cores with 120GB of LPDDR5X memory
  - NVIDIA Hopper GPU that has 96GB of HBM3 memory
- Each node had four GH200s
  - Connected by 18 NVLinks, four links per device
  - Four Mellanox ConnectX-7 network cards
- Compiled with NVHPC 23.11

*We would like to thank Nvidia for these resources

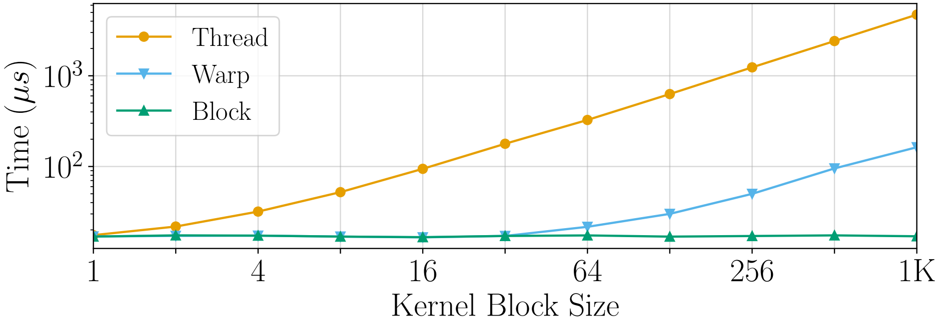
Experimental Setup

- We used a two-node NVIDIA GH200 Grace Hopper Superchip testbed*
  - NVIDIA Grace CPU with 72 ARM Neoverse V2 CPU cores with 120GB of LPDDR5X memory
  - NVIDIA Hopper GPU that has 96GB of HBM3 memory
- Each node had four GH200s
  - Connected by 18 NVLinks, four links per device
  - Four Mellanox ConnectX-7 network cards
- Compiled with NVHPC 23.11
- We implemented this with Open MPI v5.0.1rc1 and UCX (master:bc85b70e6)

*We would like to thank Nvidia for these resources

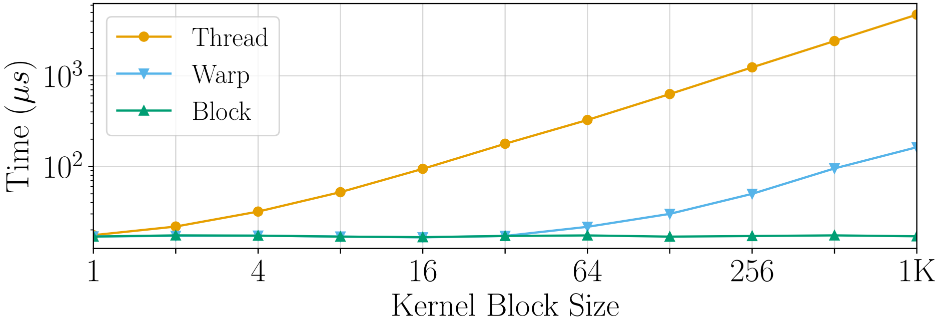
Message Aggregation Results

- The MPI Hybrid Group proposed a few granular bindings for `MPIX_Pready`



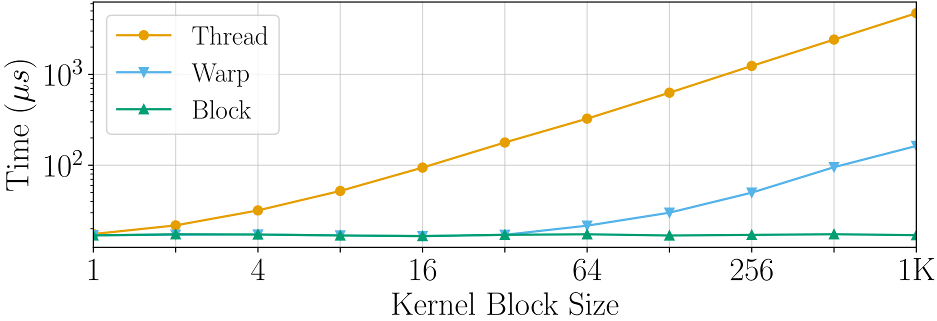
Message Aggregation Results

- The MPI Hybrid Group proposed a few granular bindings for `MPIX_Pready`
  - `MPIX_Pready_thread`
  - `MPIX_Pready_warp`
  - `MPIX_Pready_block`



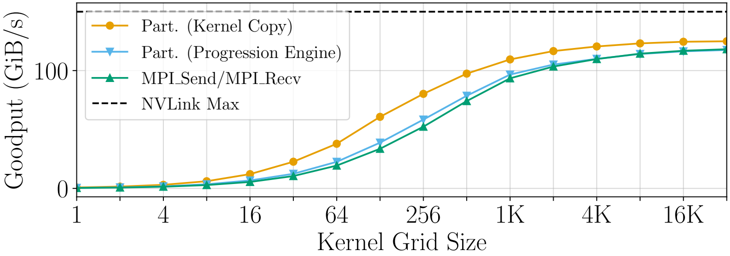
Message Aggregation Results

- The MPI Hybrid Group proposed a few granular bindings for `MPIX_Pready`
  - `MPIX_Pready_thread`
  - `MPIX_Pready_warp`
  - `MPIX_Pready_block`
- We found aggregating by blocks performed best



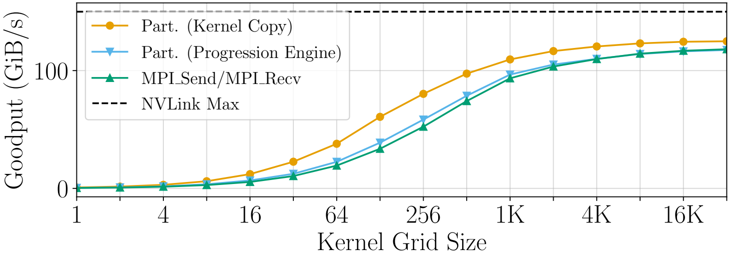
Intra-Node Point-to-Point Results

- We evaluated our intra-node point-to-point performance for our different copy mechanisms:



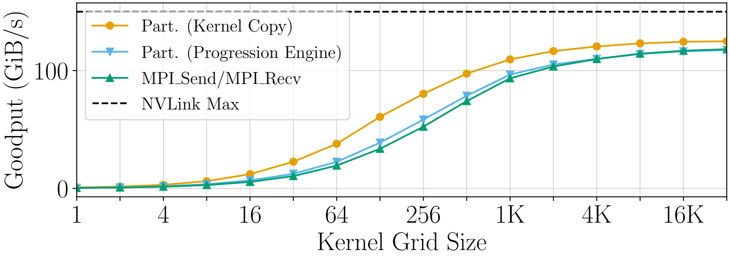
Intra-Node Point-to-Point Results

- We evaluated our intra-node point-to-point performance for our different copy mechanisms:
  - Kernel Copy
  - MPI Progession Engine Copy



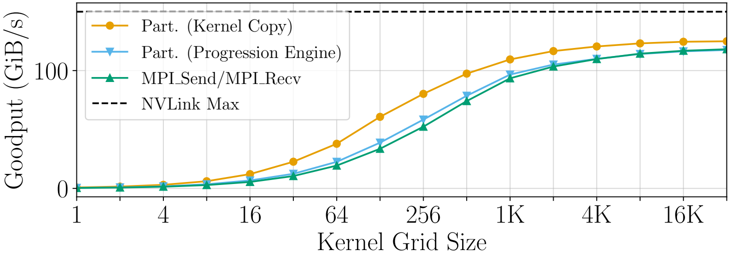
Intra-Node Point-to-Point Results

- We evaluated our intra-node point-to-point performance for our different copy mechanisms:
  - Kernel Copy
  - MPI Progession Engine Copy
- Goodput is the total data transfered divided by the time to launch the kernel, do work and complete communication
- Generally we see better performance with MPI Partitioned



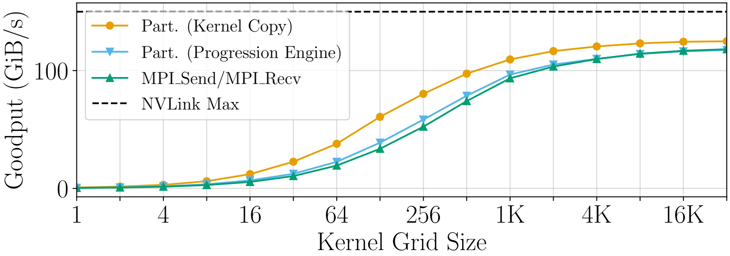
Intra-Node Point-to-Point Results

- We evaluated our intra-node point-to-point performance for our different copy mechanisms:
  - Kernel Copy
  - MPI Progession Engine Copy
- Goodput is the total data transfered divided by the time to launch the kernel, do work and complete communication
- Generally we see better performance with MPI Partitioned
  - Intra-Node Kernel Copy performs better than Progession Engine Copy



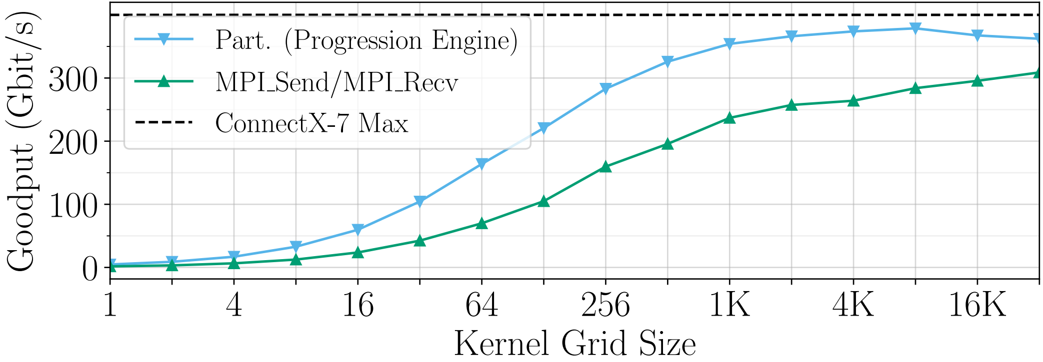
Intra-Node Point-to-Point Results

- We evaluated our intra-node point-to-point performance for our different copy mechanisms:
  - Kernel Copy
  - MPI Progession Engine Copy
- Goodput is the total data transfered divided by the time to launch the kernel, do work and complete communication
- Generally we see better performance with MPI Partitioned
  - Intra-Node Kernel Copy performs better than Progession Engine Copy
  - The Kernel Copy has the benefit of no host involement



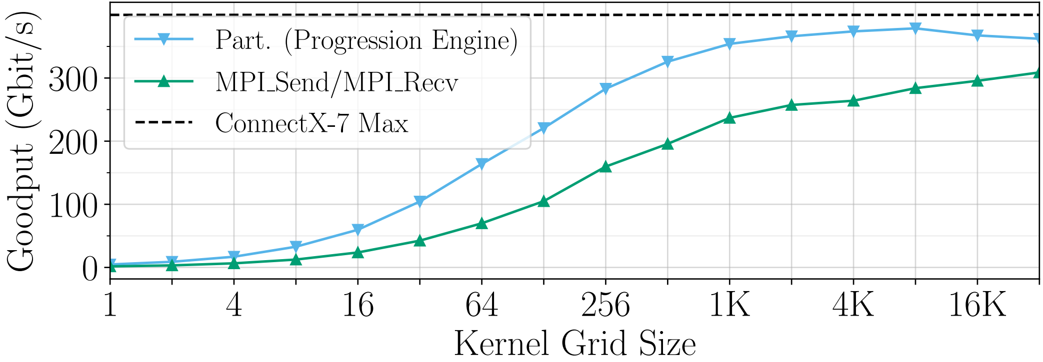
Inter-Node Point-to-Point Results

- We only evaluated the MPI Progression Copy for the inter-node case



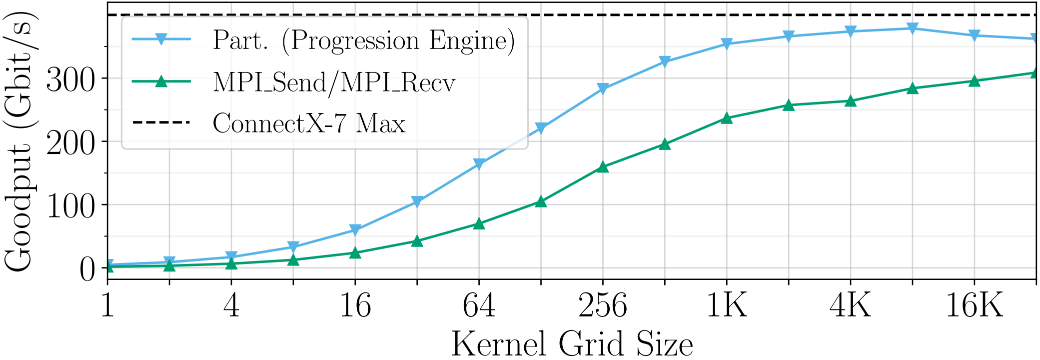
Inter-Node Point-to-Point Results

- We only evaluated the MPI Progression Copy for the inter-node case



Inter-Node Point-to-Point Results

- We only evaluated the MPI Progression Copy for the inter-node case
- This improved goodput performance for inter-node case



MPI_Pallreduce Results

- Significant improvement compared to `MPI_Allreduce`

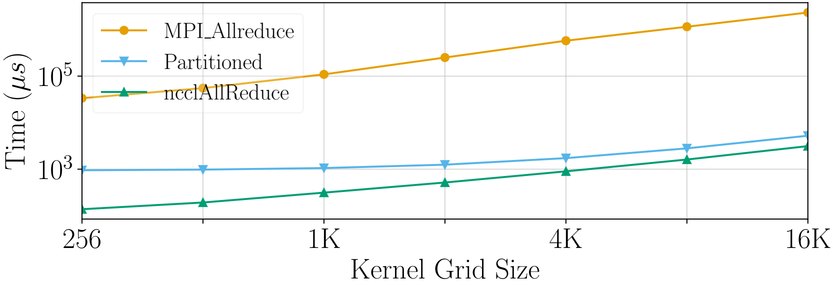


Figure: Eight-Node Allreduce Goodput Performance

MPI_Pallreduce Results

- Significant improvement compared to `MPI_Allreduce`
- Reduces the performance gap between Open MPI and NCCL

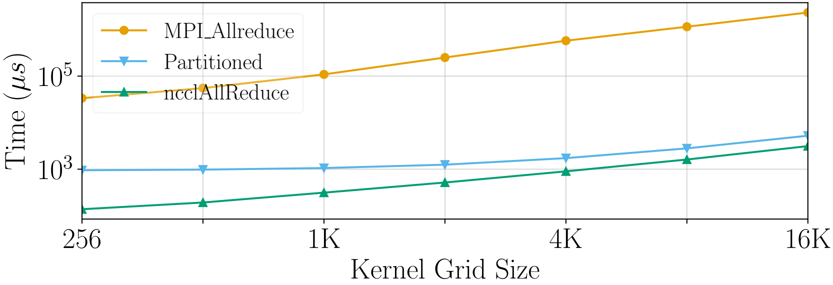


Figure: Eight-Node Allreduce Goodput Performance

Overheads

- The overheads of these new MPI calls must be considered

Table: Overheads for Different MPI Calls

MPI Call	Overhead
`MPI_PSend/Recv_init`	$17.2 \pm 10.2\mu s$
`MPIX_Pallreduce_init`	$62.3 \pm 6.2\mu s$
`MPIX_Prequest_create`	$110.7 \pm 37.8\mu s$
`MPIX_Pbuf_prepare`	$193.4\mu s$ first, $3.4 \pm 1.4\mu s$ avg.

Overheads

- The overheads of these new MPI calls must be considered
- Creating `MPIX_Prequest` can have a high overhead so it is important to minimize what must be copied to the device.

Table: Overheads for Different MPI Calls

MPI Call	Overhead
`MPI_PSend/Recv_init`	$17.2 \pm 10.2\mu s$
`MPIX_Pallreduce_init`	$62.3 \pm 6.2\mu s$
`MPIX_Prequest_create`	$110.7 \pm 37.8\mu s$
`MPIX_Pbuf_prepare`	$193.4\mu s$ first, $3.4 \pm 1.4\mu s$ avg.

Overheads

- The overheads of these new MPI calls must be considered
- Creating `MPIX_Prequest` can have a high overhead so it is important to minimize what must be copied to the device.
- `MPIX_Pbuf_prepare` has a high initial cost as it is required to complete initialization

Table: Overheads for Different MPI Calls

MPI Call	Overhead
`MPI_PSend/Recv_init`	$17.2 \pm 10.2\mu s$
`MPIX_Pallreduce_init`	$62.3 \pm 6.2\mu s$
`MPIX_Prequest_create`	$110.7 \pm 37.8\mu s$
`MPIX_Pbuf_prepare`	$193.4\mu s$ first, $3.4 \pm 1.4\mu s$ avg.

Overheads

- The overheads of these new MPI calls must be considered
- Creating `MPIX_Prequest` can have a high overhead so it is important to minimize what must be copied to the device.
- `MPIX_Pbuf_prepare` has a high initial cost as it is required to complete initialization
  - Subsequent calls are low cost

Table: Overheads for Different MPI Calls

MPI Call	Overhead
`MPI_PSend/Recv_init`	$17.2 \pm 10.2\mu s$
`MPIX_Pallreduce_init`	$62.3 \pm 6.2\mu s$
`MPIX_Prequest_create`	$110.7 \pm 37.8\mu s$
`MPIX_Pbuf_prepare`	$193.4\mu s$ first, $3.4 \pm 1.4\mu s$ avg.

Jacobi Solver Results

- Evaluated the CUDA-Aware MPI Jacobi Solver

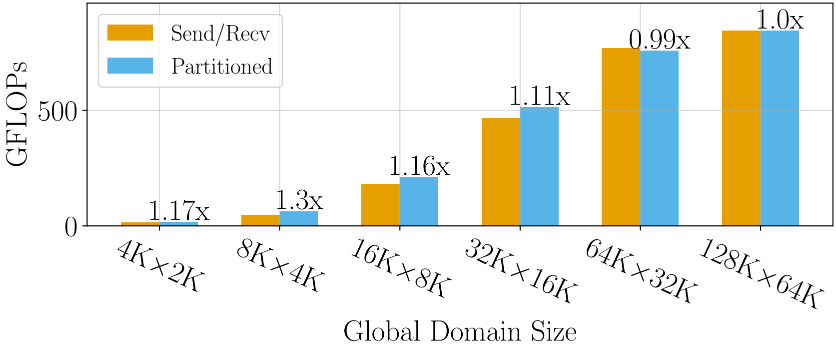


Figure: Eight-Node Jacobi Solver Performance

Jacobi Solver Results

- Evaluated the CUDA-Aware MPI Jacobi Solver
  - The solver uses a Point-to-Point Halo Exchange

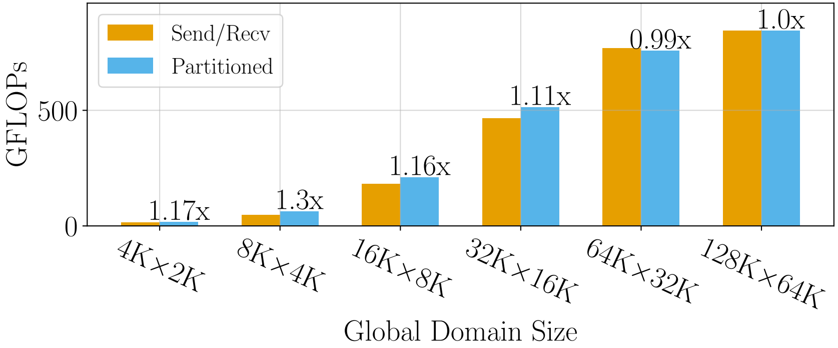


Figure: Eight-Node Jacobi Solver Performance

Jacobi Solver Results

- Evaluated the CUDA-Aware MPI Jacobi Solver
  - The solver uses a Point-to-Point Halo Exchange
  - Performs best for smaller kernel

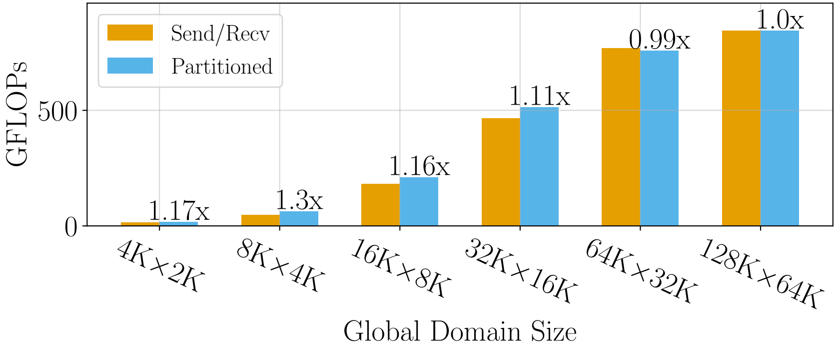


Figure: Eight-Node Jacobi Solver Performance

Deep Learning Kernel Results

- Evaluated a distributed Binary Cross Entropy Loss
  - Depends on Allreduce

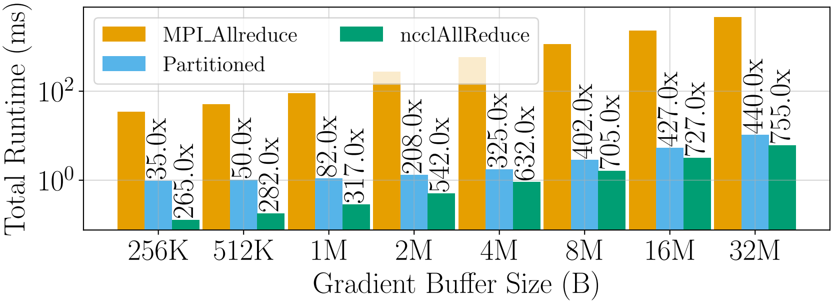


Figure: Eight-Node Deep Learning Kernel Performance

Deep Learning Kernel Results

- Evaluated a distributed Binary Cross Entropy Loss
  - Depends on Allreduce
- Significant improvement compared to `MPI_Allreduce`

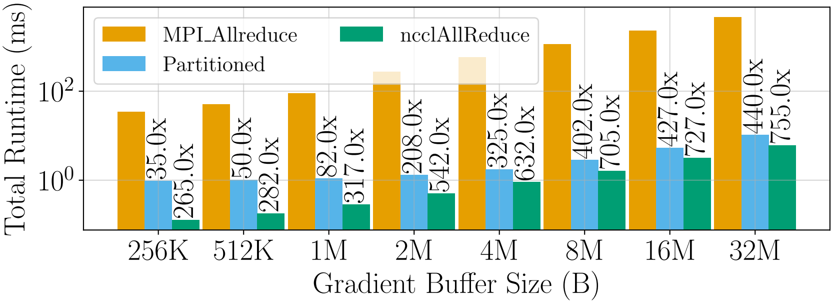


Figure: Eight-Node Deep Learning Kernel Performance

Conclusion

- GPU-Initiated MPI Partitioned allows MPI to stay relevent with vendor specific libraries
  - Such as NCCL/RCCL and NVSHMEM/ROC_SHMEM

Conclusion

- GPU-Initiated MPI Partitioned allows MPI to stay relevent with vendor specific libraries
  - Such as NCCL/RCCL and NVSHMEM/ROC_SHMEM
- Useful optimizations for GPU-Initiated MPI Partition are identified

Conclusion

- GPU-Initiated MPI Partitioned allows MPI to stay relevent with vendor specific libraries
  - Such as NCCL/RCCL and NVSHMEM/ROC_SHMEM
- Useful optimizations for GPU-Initiated MPI Partition are identified
  - Partition aggregation at the block-level provides the best performance

Conclusion

- GPU-Initiated MPI Partitioned allows MPI to stay relevent with vendor specific libraries
  - Such as NCCL/RCCL and NVSHMEM/ROC_SHMEM
- Useful optimizations for GPU-Initiated MPI Partition are identified
  - Partition aggregation at the block-level provides the best performance
  - Intra-node copies can be optimized using a kernel copy

Conclusion

- GPU-Initiated MPI Partitioned allows MPI to stay relevent with vendor specific libraries
  - Such as NCCL/RCCL and NVSHMEM/ROC_SHMEM
- Useful optimizations for GPU-Initiated MPI Partition are identified
  - Partition aggregation at the block-level provides the best performance
  - Intra-node copies can be optimized using a kernel copy
- Application results suggests that this could benefit HPC applications that use smaller kernels
  - Reduces the performance delta between MPI and vendor libaries in AI workloads

Acknowledgements

Questions

Thank You