Partitioned Communication to Support Hybrid Programming

PRESENTER: DR. RYAN E. GRANT

STUDENT CREDIT: YILTAN TEMUCIN, JORDAN ABT, ELIZABETH REID

Hybrid Programming

- More than just MPI+X in the traditional sense
  - Not just MPI+OpenMP
- Support for large amounts of concurrency
- Not necessarily on the CPU-only
  - Need solutions for accelerators as well
- Let's examine how we got here and what we need going forward and how Partitioned communication let's us get there….

Why Partitioned? Why Now?

MPI use cases continue to evolve

- MPI+X implies the use of threads, e.g. OpenMP

High Bandwidth Memory (HBM) plays a role

- Faster memory built on-die/in-package but you get less capacity than previously
- Buy DRAM for channels and get a lot of space you can use, not so with HBM
- MPI processes can take a lot of memory each

- Core counts skyrocketing
  - 192 cores in an AMD EPYC, 384 in a dual-socket node
  - Remember 56-core used to be a many-core architecture?

- Accelerators
  - Can’t rely on CPUs to move your messages anymore, different architectures, new challenges

What do we want in a solution?

Desirable/required features:

Low overhead

- Having many messages and ranks causes message matching/steering overhead

Similar semantics to existing concurrency solutions

- Ease of programmability

Decouple message passing setup/handling from data movement

- Don’t need to setup at the time of sending
- Useful for accelerators

Minimal locking/sycnhronization

Why not thread_multiple?

Threads introduce significant issues with concurrency in existing MPI implementations

- MPI_THREAD_MULTIPLE is hard and implementations don't do it well
- Difficult to support concurrency without encountering conflicts

Need to expose the parallelism but without locking/synchronization

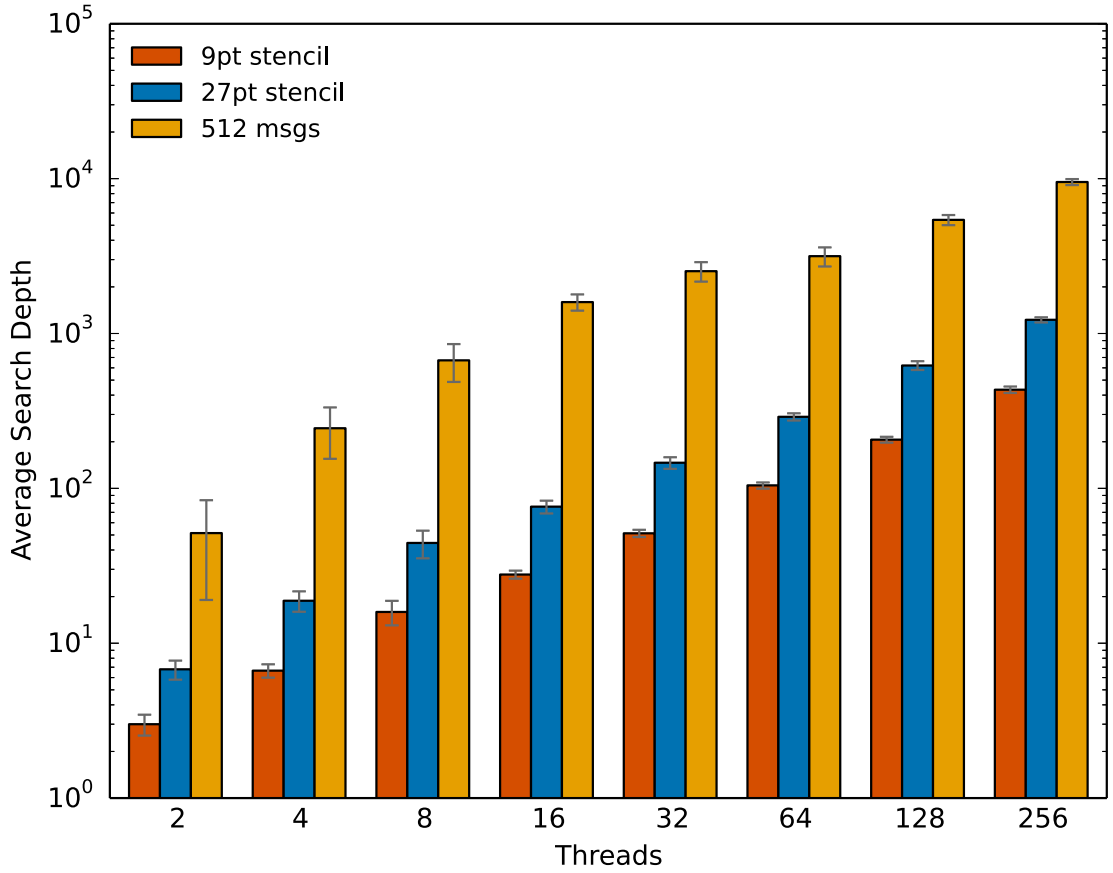
- Existing solutions already provide a way
- Build without conflicts between execution context (threads) and you don't need to worry about locking

Concurrency costs

Have known about matching list performance for sometime

Non-viable with many many messages, so you can either avoid many messages that need matching or reduce costs of matching

Partitioned avoids the matching outside of init and avoids wildcards to let other matching happen in efficient manners (e.g. hashing)



Data from Measuring Multithreaded Message Matching Misery, EuroPar 2018, Whit Schonbein et al.

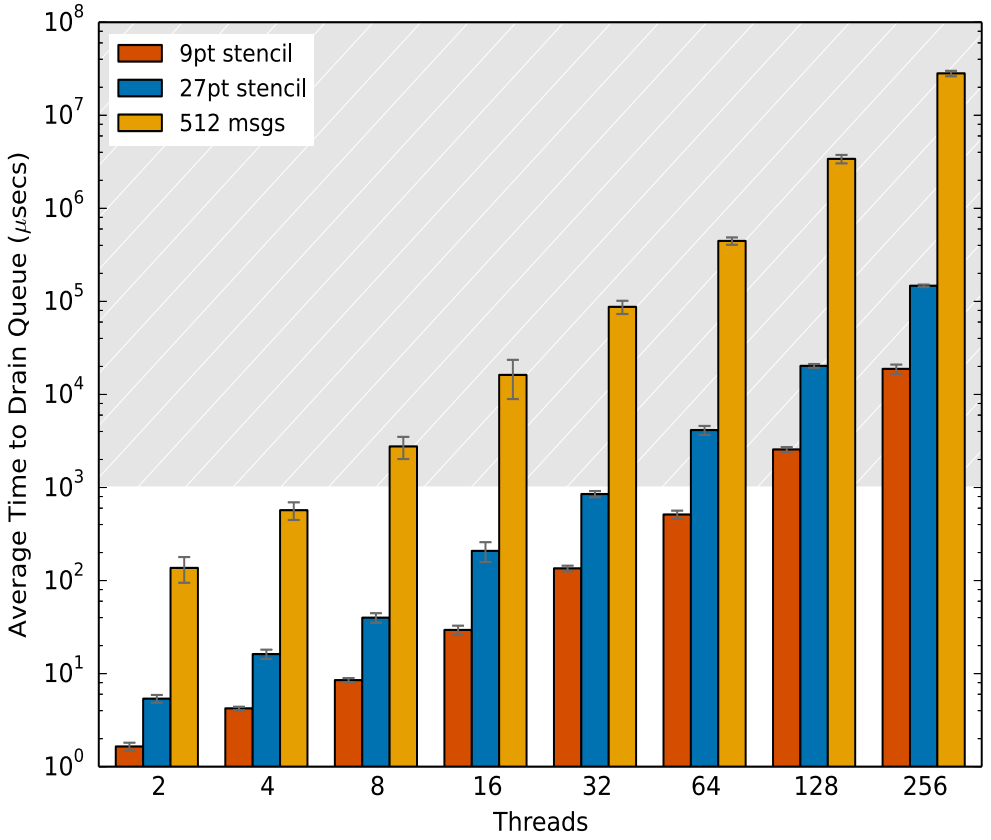
Concurrency costs

Need to stay out of the grey region

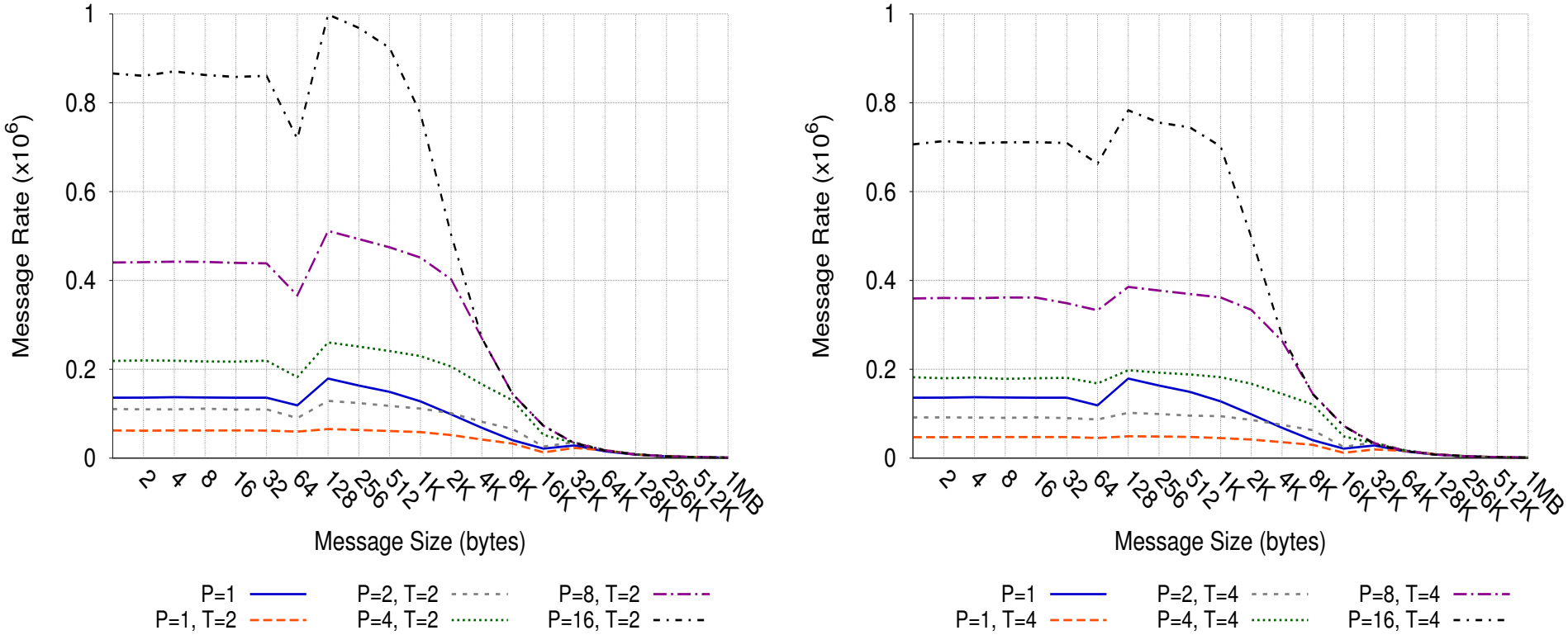
- Breakdown occurs here as we are using more time to communicate than the time needed to do an iteration of simulation

By keeping queues very short we can work at fast speeds

- Do this by avoiding message matching (almost) entirely with partitioned
- But still get matching-type semantics for completion



But concurrency helps



Impact of communication concurrency on message rate per second
for P processes and T threads per process on an Intel Phi

Takeaway: Fewer threads = better performance, due to poor MPI multi-threading

James Dinan, Ryan E. Grant, Pavan Balaji, Dave Goodell, Douglas Miller, Marc Snir, Rajeev Thakur, "Enabling communication concurrency through flexible MPI Endpoints", International Journal of High Performance Computing Applications, Volume 28, Issue 4, pp. 390-405, November 2014. Impact factor: 1.63

Partitioned Communication

MPI Partitioned Communication Concepts

Many actors (threads) contributing to a larger operation in MPI
- Same number of messages as today!
- No new ranks – no need to understand target thread

Many threads work together to assemble a message
- MPI only has to manage knowing when completion happens
- These are actor/action counts, not thread level collectives

Persistent-style communication
- Init…(Start…test/wait)…free

No heavy MPI thread concurrency handling required
- Leave the placement/management of the data to the user

No more complicated packing of data, send structures when they become available

How to use Partitioned MPI

Like persistent communications, setup the operation

```
int MPI_Partitioned_send_init(void *buf, int partitions, int count,
    MPI_Datatype data_type, int to_rank, int to_tag, int num_partitions,
    MPI_Info info, MPI_Comm comm, MPI_Request *request)
```

Start the request

```
int MPI_Start(MPI_Request request)
```

Add items to the buffer

```
int MPI_Pready(int partition, MPI_Request request)
```

MPI_Pready is thread-safe and meant to be called from separate threads

Wait on completion

```
int MPI_Wait(MPI_Request request)
```

Optional: Use the same partitioned send over again

```
int MPI_ Start(MPI_Request request)
```

Persistent Partitioned Buffers

Expose the “ownership” of a buffer as shared to MPI

Need to specify the operation to be performed before contributing segments

MPI implementation doesn’t have to care about sharing

- Only needs to know how many times it will be called

Applications are required to manage thread buffer ownership such that the buffer is valid

- The same as would be done today for codes where many threads work on a dataset (with the exception of reductions)

Result: MPI is thread agnostic with a minimal synchronization overhead (atomic_fetch_and_add)

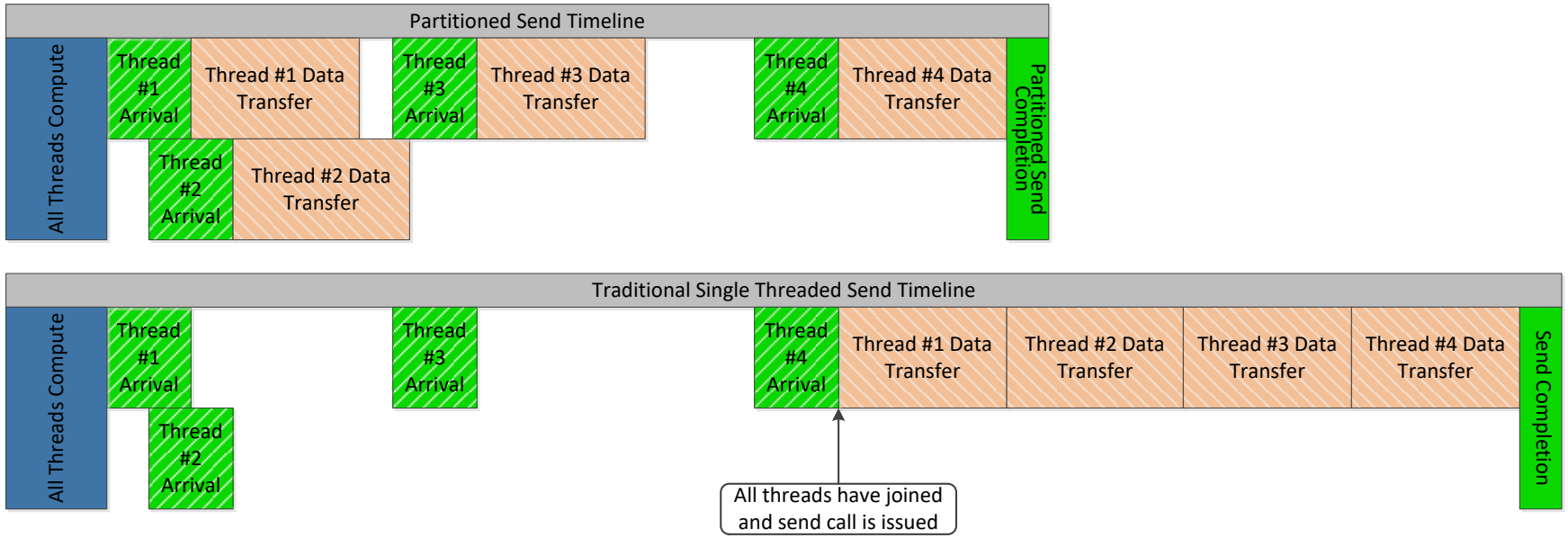
- Can alternatively use a task model instead of threads, IOVEC instead of contiguous buffer

New Type of Overlap

"Early bird communication"

Early threads can start moving data right away

Can implement using RDMA to avoid message matching



GPUs and Accelerators

GPUs cannot run MPI libraries natively – at least not well

Need coordination for network transfers
- But don't want to setup communications from step one on the GPU/Accelerator

Minimize overhead of communication initialization
- Many potential notifications – must be lightweight

Existing NIC hardware can use triggering
- Need a mechanism in MPI to do lightweight triggering

Communication can be optimized by host CPU
- CPU can optimize to network before the transfer takes place
- Optimize number of transfers, when things trigger

Allow for Better Parallelism in MPI

Concept of many actors (threads/warps/thread blocks) contributing to a larger operation in MPI

- Same number of messages as you use in isend/irecv today
- No new ranks

Many threads work together to assemble a message

- MPI only has to manage knowing when completion happens
- These are actor/action counts, not thread level collectives, to better enable tasking models

No heavy MPI thread concurrency handling required

- Leave the placement/management of the data to the user
- Knowledge required: number of workers, which is easily available

Bonus: Match well with Offloaded NIC capabilities

- Use counters for sending/receiving
- Utilize triggered operations to offload sends to the NIC

Persistent Partitioned Buffers

Expose the “ownership” of a buffer as a shared to MPI

Need to describe the operation to be performed before contributing segments

MPI implementation doesn’t have to care about sharing

- Only needs to understand how many times it will be called

Threads are required to manage their own buffer ownership such that the buffer is valid

- The same as would be done today for code that has many threads working on a dataset (that’s not a reduction)

Result: MPI is thread agnostic with a minimal synchronization overhead (atomic_inc)

- Can alternatively use task model instead of threads, IOVEC instead of contiguous buffer

Example from MPI-4.0

```
#define NUM_THREADS 8
#define NUM_TASKS 64
#define PARTITIONS NUM_TASKS
#define PARTLENGTH 16
#define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
int main( int argc, char *argv[]) /* send-side partitioning */
{
 double message[MESSAGE_LENGTH];
 int send_partitions = PARTITIONS,
    send_partlength = PARTLENGTH,
    recv_partitions = 1,
    recv_partlength = PARTITIONS*PARTLENGTH;
 int count = 1, source = 0, dest = 1, tag = 1,
 flag = 0;
 int myrank;
 int provided;
 MPI_Request request;
 MPI_Info info = MPI_INFO_NULL;
 MPI_Datatype send_type;
 MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
 if (provided < MPI_THREAD_SERIALZED) MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
 MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
 MPI_Type_contiguous(send_partlength, MPI_DOUBLE, &send_type);
 MPI_Type_commit(&send_type);

 if (myrank == 0)    /* code for process zero */
 {
   MPI_Psend_init(message, send_partitions, count, send_type, dest, tag,
          info, MPI_COMM_WORLD, &request);
   MPI_Start(&request);

   #pragma omp parallel shared(request) num_threads(NUM_THREADS)
   {
     #pragma omp single
      {
        /* single thread creates 64 tasks to be executed by 8 threads */
        for (int partition_num=0;partition_num<NUM_TASKS;partition_num++)
        {
          #pragma omp task firstprivate(partition_num)
          {
 /* compute and fill partition #partition_num, then mark ready: */
           /* buffer is filled in arbitrary order from each task */
           MPI_Pready(request, partition_num);
          } /*end task*/
        } /* end for */
      } /* end single */
   } /* end parallel */
   while(!flag)
   {
     /* Do useful work */
     MPI_Test(&request, &flag,  MPI_STATUS_IGNORE);
     /* Do useful work */
   }
   MPI_Request_free(&request);
 }
 else if (myrank == 1)  /* code for process one */
 {
   MPI_Precv_init(message, recv_partitions, recv_partlength, MPI_DOUBLE,
          source, tag, info, MPI_COMM_WORLD, &request);

   MPI_Start(&request);
 while(!flag)
   {
     /* Do useful work */
     MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
     /* Do useful work */
   }
   MPI_Request_free(&request);
 }
 MPI_Finalize();
 return 0;
}
```

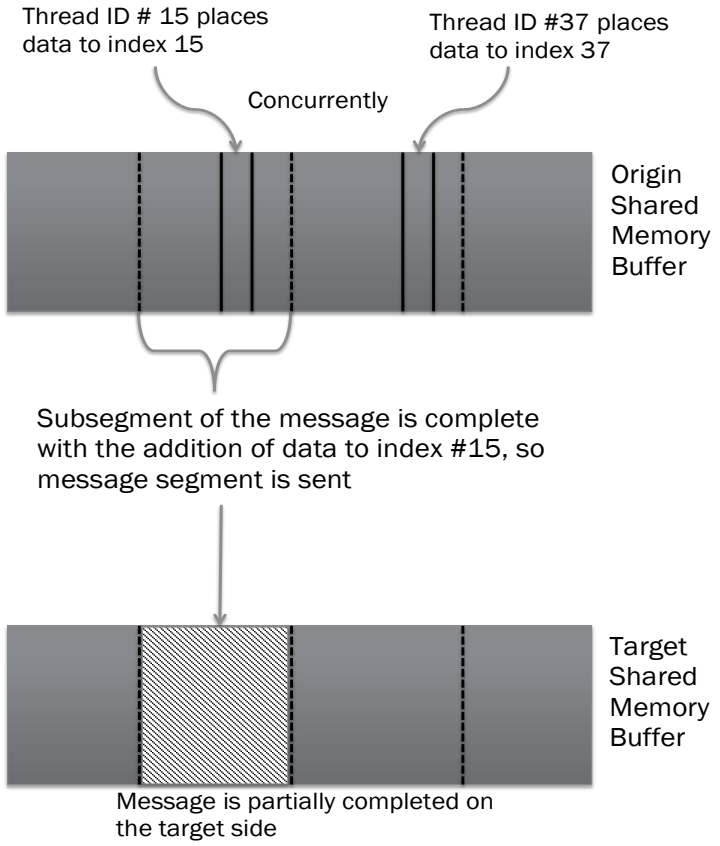
Opportunities for Optimization

MPI implementations can optimize data transfer under the covers:

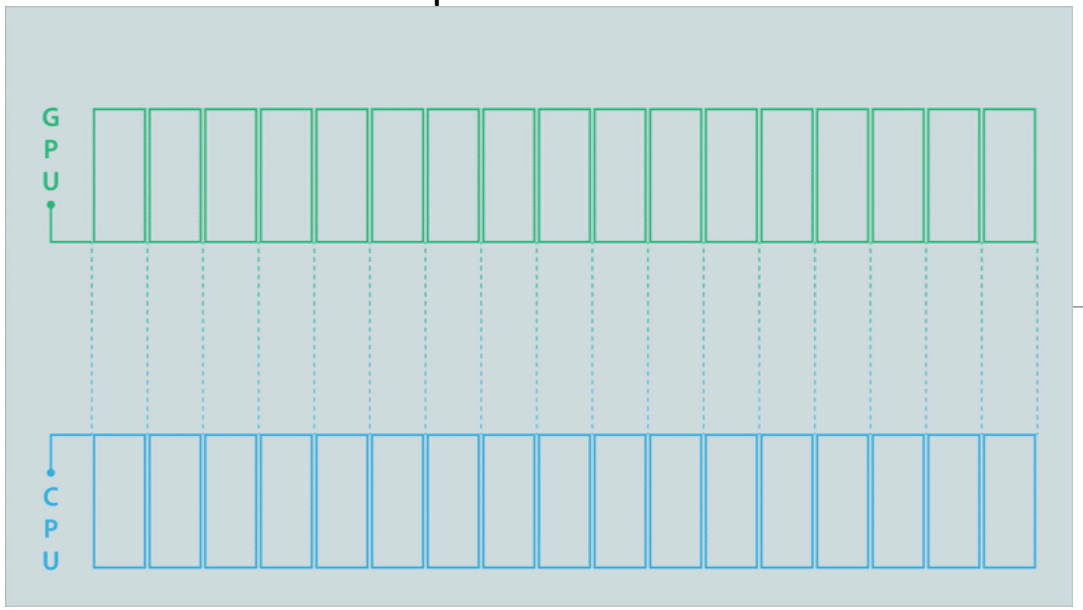
- Subdivide larger buffers and send data when ready
- Could be optimized to specific networks (MTU size)
- Number of messages will be:

  $1 < \#messages \leq \#threads/tasks$

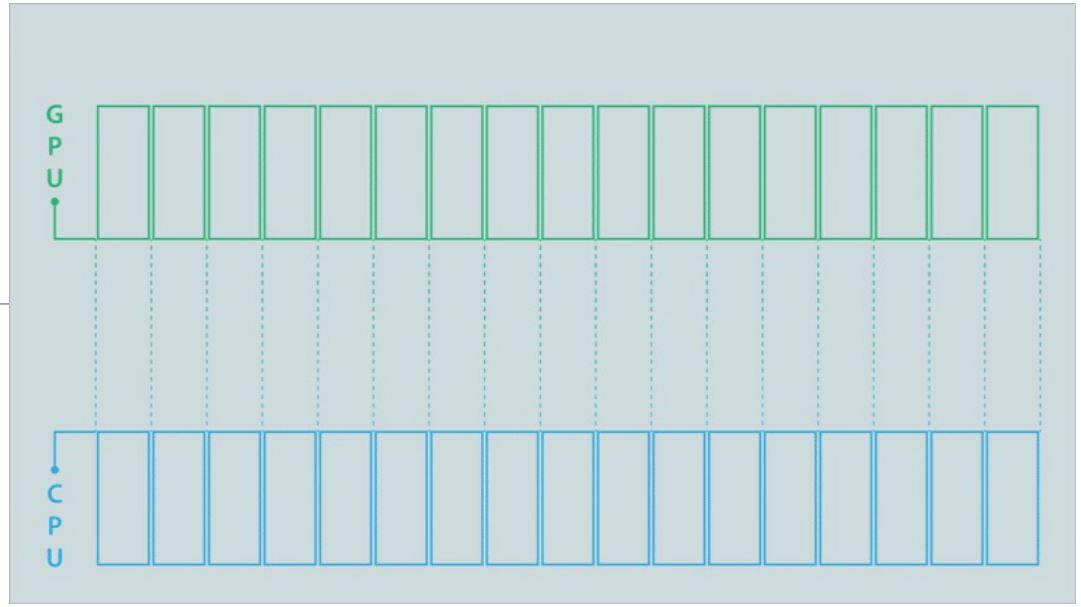
  For a partition with 1 part per thread
- Reduces the total number of messages sent, decreasing matching overheads



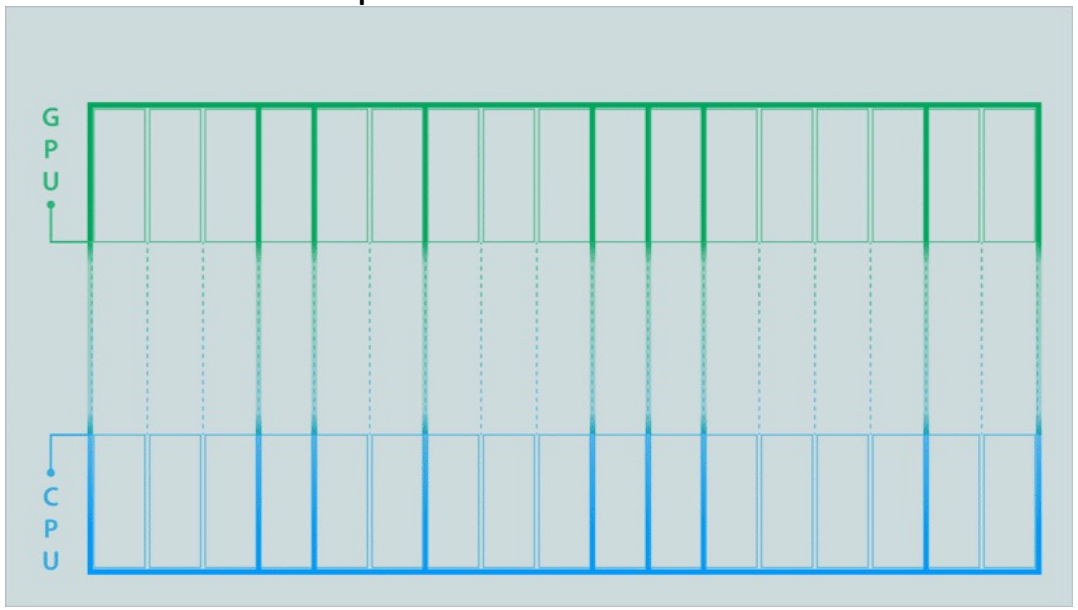
Non-partitioned



Naïve Partitioned



Optimized Partitioned



Hardware adds more parallelism

More device contexts to separate traffic

- More hardware to use for sending data independently

More paths through the network

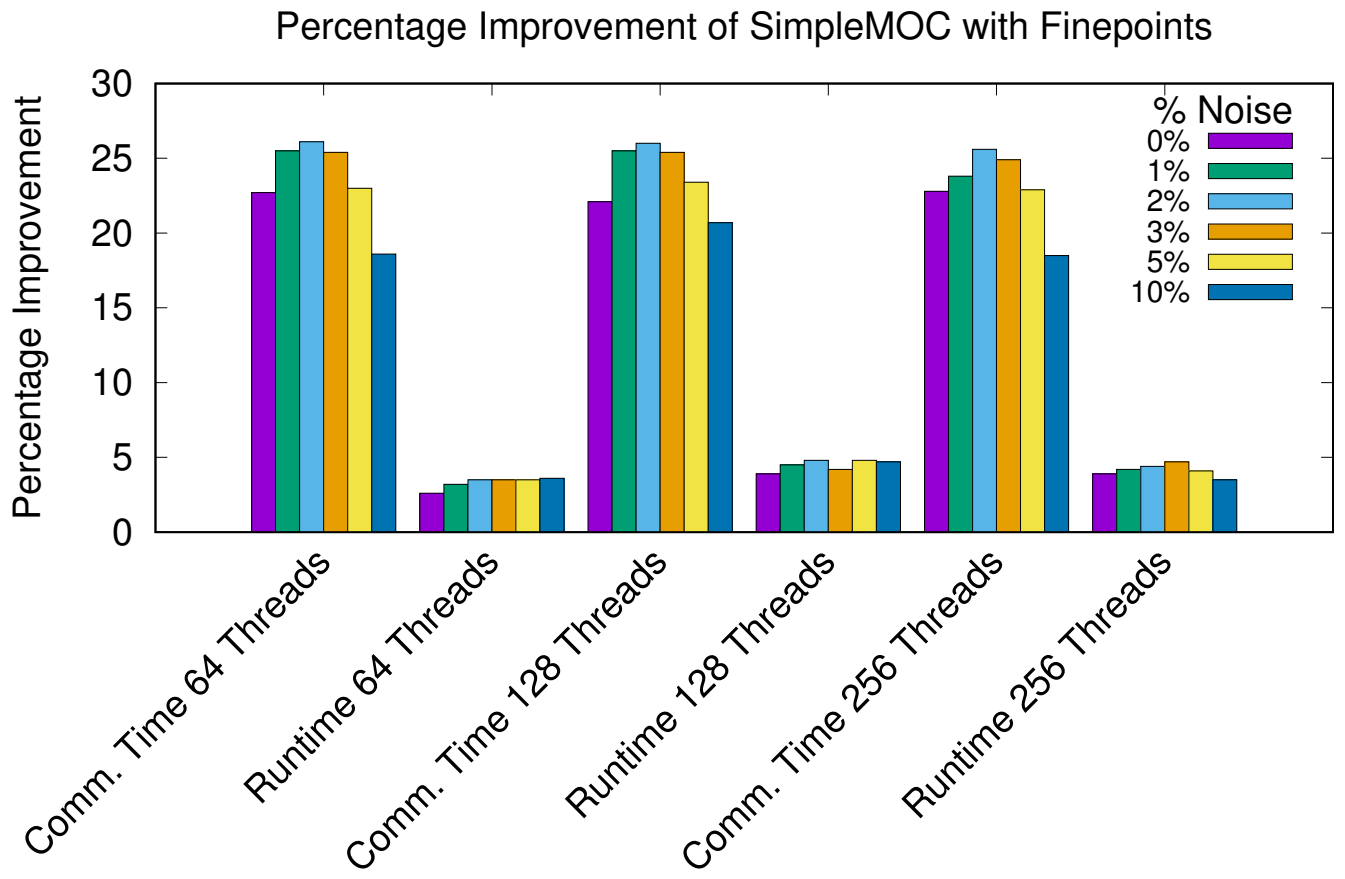
- Leverage the network resources more than single path routing

More options for direct accelerator messaging

- Build in pready support to GPUs for triggering

CPU Application Benefit

Real reactor physics proxy app: SimpleMOC



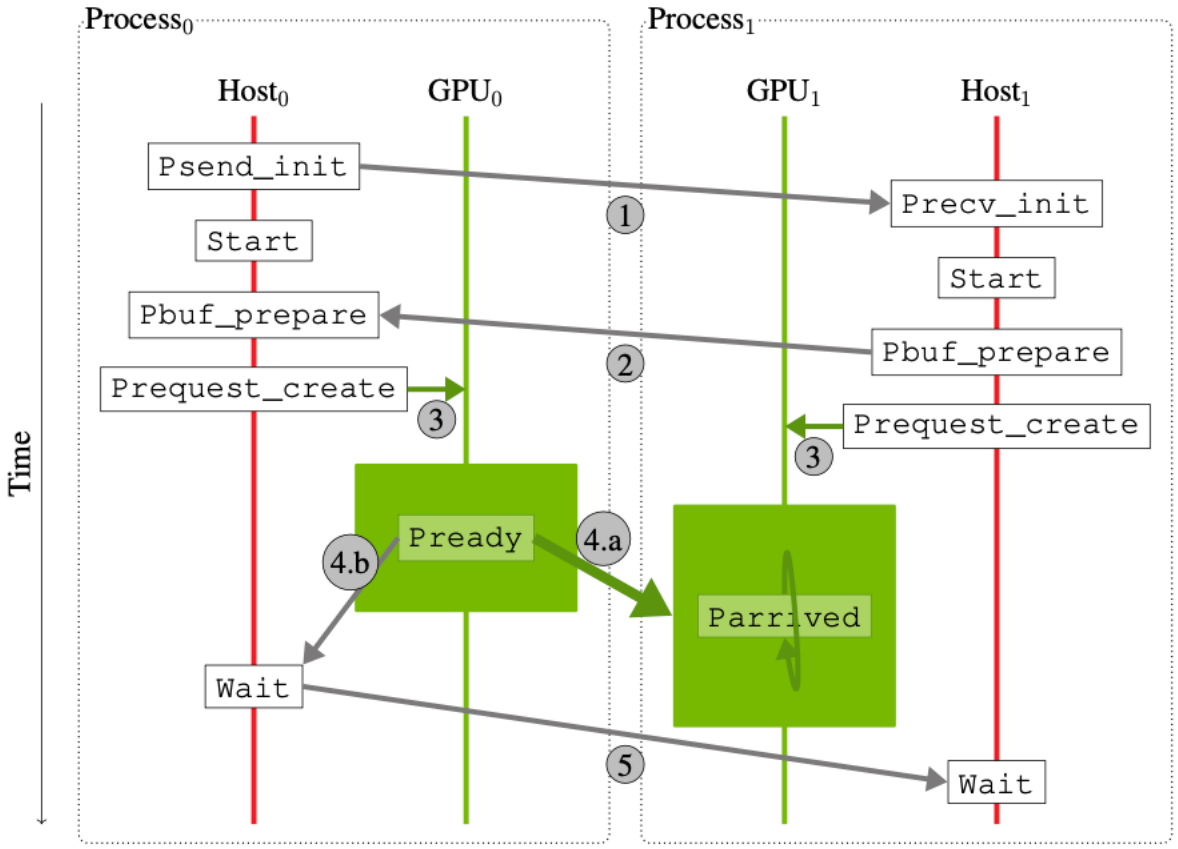
GPU-initiated Partitioned

Example

Adding in GPUs, we have similar flow to CPU only partitioned

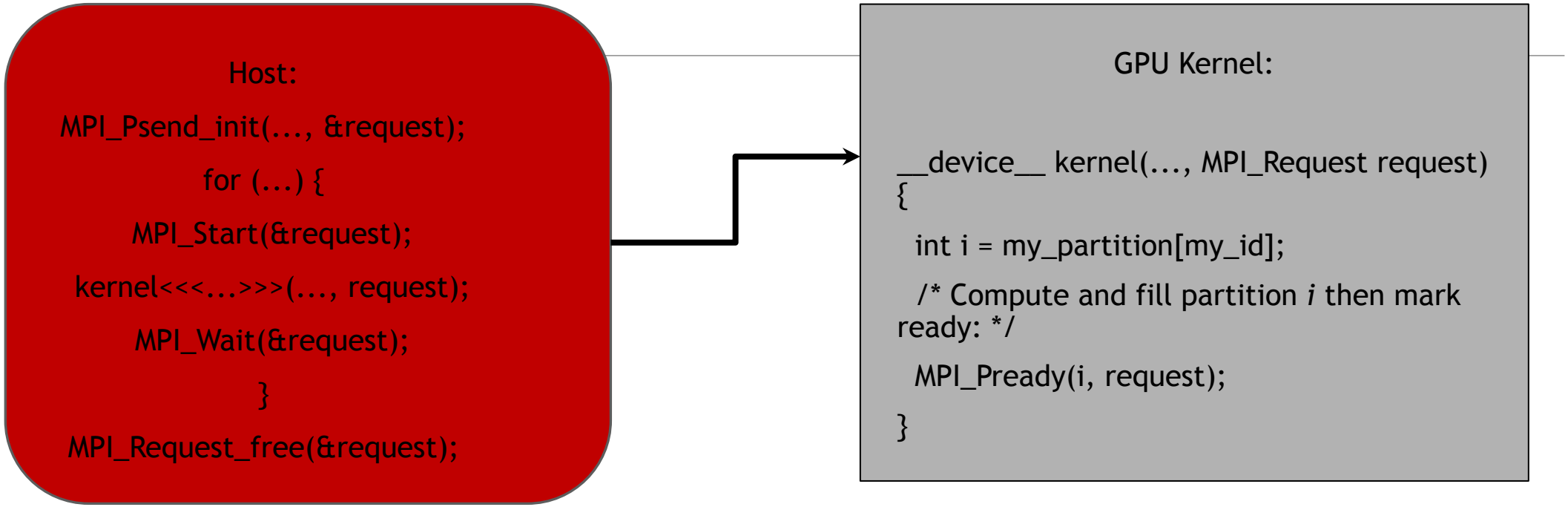
Pready/arrived are implemented in GPU space

Can bypass CPU for communication



Credit: Yiltan Temucin et al. To appear ExaMPI

Usage model - Kernel communication triggering



Note: CPU does communication setup and completion steps for MPI. Setup commands on NIC and poll for completion of entire operation. Kernel just indicates when NIC/MPI can send data. Ideally want to trigger communication from GPU to fire off when data is ready without communication setup/completion in kernel

One way to do it – with extras

```
MPIX_Device int MPIX_Pready(int partition, MPIX_Prequest preq);

__global__ int kernel_B(MPIX_Prequest preq, double *sbuf)

{

int idx = threadIdx.x + blockDim.x * threadIdx.y; /* Do Work */
MPIX_Pready(idx, preq);

}
__host__ int host_function(MPI_Request req,

MPI_Start(req)

double *sbuf)

{ MPIX_Pbuf_Prepare(req);

if (first_iteration) {MPIX_Prequest_create(preq, req); }

kernel_B<<<stream>>>(preq, sbuf);

/* Do work on host */

MPI_Wait(req); }
```

Wait, what are those MPIX calls?

We need to be able to define what requests look like on a GPU

- GPU MPI_Request may not equal CPU-side MPI request struct
- Why? Much better ways to lay out data for GPUs to use efficiently than CPU-type structures

We also need the GPU to have requests that are already setup for it

- Helps with performance

Is this what will be in future versions?

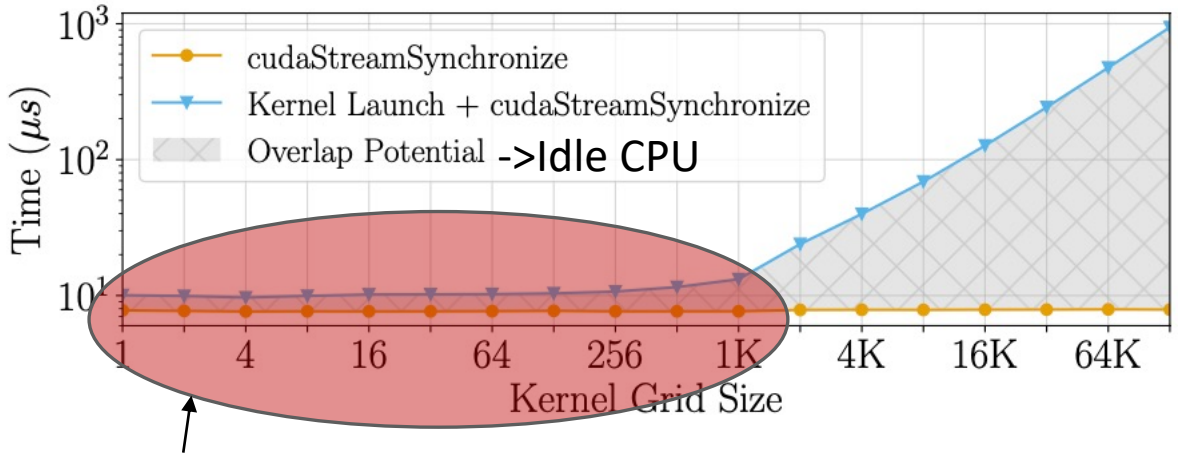
- No, we will define the needs in bindings for the kernel environments

Avoiding synchronization

Synchronization is costly, so Partitioned avoids this by creating logical separation on data movement based on the partition numbering

See that simple vector addition kernel with synchronization happening grows in cost as grid size grows but synchronization costs remain the same

Avoid this by just not having to synchronize



Inefficient region to sync

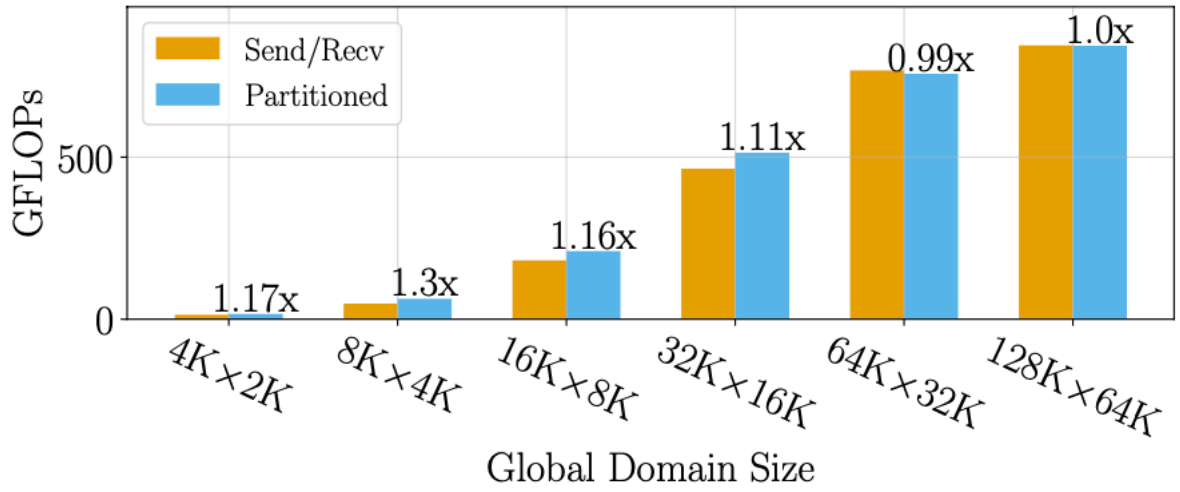
Performance Benefits

Example: Jacobi solver

8-GPUs

We can see that not synchronizing is helping in the smaller ranges, we're seeing a combination of not synchronizing with early bird communication

Results scale up better with increasing node counts

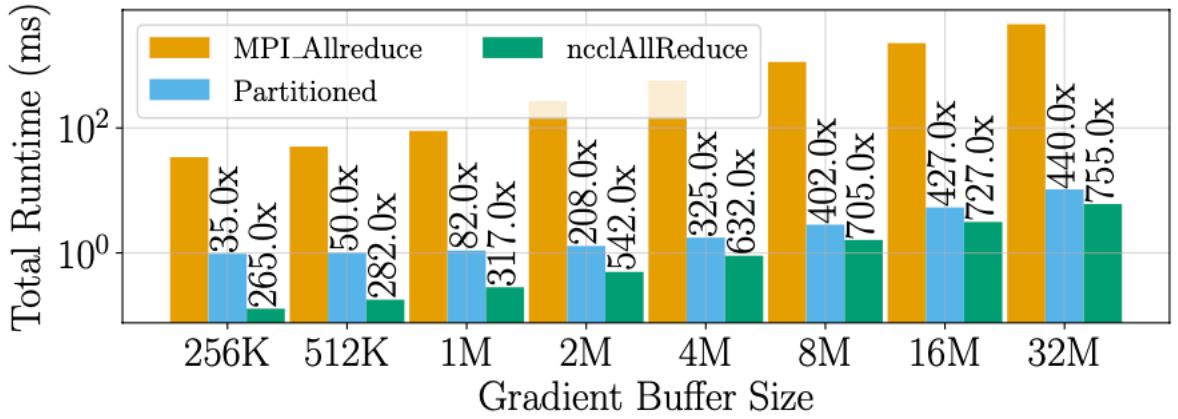


The Future

No need to stop at point-to-point

Partitioned collectives show promise, here is first example of partitioned collectives prototype running with GPU-initiated communication

Can see that things are much better but still not at vendor proprietary levels



Still plenty to be done

As shown with previous presentation, libraries are just deploying Partitioned, more tuning needs to be done

- Many possible new optimizations, we are just scratching the surface with recent ideas, plenty more to come

Adoption path still in front of us

- Applications can take advantage of partitioned communication
- Especially GPU applications with native bindings

Writing bindings for GPUs

- Side document to MPI

Thank You!

Questions?

Natural Sciences and Engineering Research Council of Canada
Conseil de recherches en sciences naturelles et en génie du Canada
Canada

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).