MPI Partitioned Communication

AN INTRODUCTION

RYAN E. GRANT
QUEEN’S UNIVERSITY, CANADA

Why do we need this?

MPI use cases continue to evolve

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

Potentially thousands of MPI processes on a single node

- HBM restricts memory sizes, no longer a easy resource
- Can complicate network resource management

Sources of concurrency

- Accelerators
  - GPUs need to send data easily, avoiding heavy weight synchronization
- Core counts
  - Traditional CPU design continues to add cores in new generations
- OpenMP
  - Can we use MPI inside of OpenMP parallel regions?

Living in a World with Threads

Desirable/required features:

Low overhead

Similar semantics to existing threading (minimal changes)

- Ease of programmability

Each thread has access to the communication library (no funneling)

- Also ease programmability, e.g. use MPI calls in an OpenMP region

Communication endpoint granularity matched to the work

- Not too fine, not too coarse, just right…
- Fine granularity at endpoints requires networking resources
  - Keeping track of many ranks, caching state related to these ranks, etc.

MPI Partitioned

Newest chapter to MPI standard (4.0)

Decouples the point-to-point (P2P) data movement from the message sending requirements on the network itself

- Prepare for data movement and match send/recv buffers before moving the actual data
- Move data in chunks when it becomes ready rather than all at once
  - Traditional P2P only happens when the whole buffer is ready

Numerous benefits in being able to optimize when to send data and how to manage partitions

- Depends on your compute architecture where you can take advantage of them

Basic Operations

MPI_Psend/recv_init – initialize a partitioned communication

- Like persistent ops only need to do this once and can repeat op without init again
- Note unlike persistent, this matches here, not with each start/op
- Includes src/dst and number of partitions

MPI_Start – start a partitioned communication just like persistent

MPI_Pready – call for sender to indicate a partition is ready to send

MPI_Parrived – check if a particular partition is available at the receiver

MPI_Test/Wait – check for completion of the partitioned operation

How does this work for accelerators?

Separates the setup of the communication from the data movement

- E.g. call pready on the GPU, all other calls on CPU

But MPI is not explicitly GPU aware, so how does this work?

- Currently works for CPUs, some prototypes exist but…

Better to build this into the semantics of GPU-MPI instead

- Side document in the works to define semantics of GPU interfaces
  - CUDA, RocM, SYCL, etc.
- Build out ability to expose data readiness at fine-grained levels without requiring heavy weight, wide synchronizations
- Well defined state of initialization on kernel launch

Usage model - Kernel communication triggering

Host:

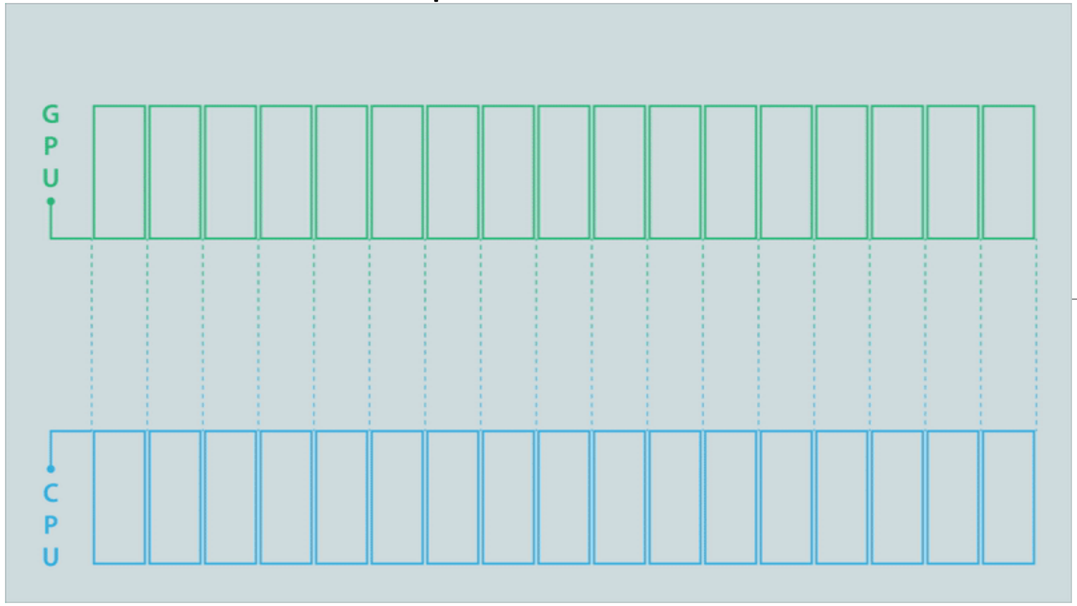
```
MPI_Psend_init(..., &request);
for (...) {
MPI_Start(&request);
kernel<<<...>>>(..., request);
MPI_Wait(&request);
}
MPI_Request_free(&request);
```

GPU Kernel:

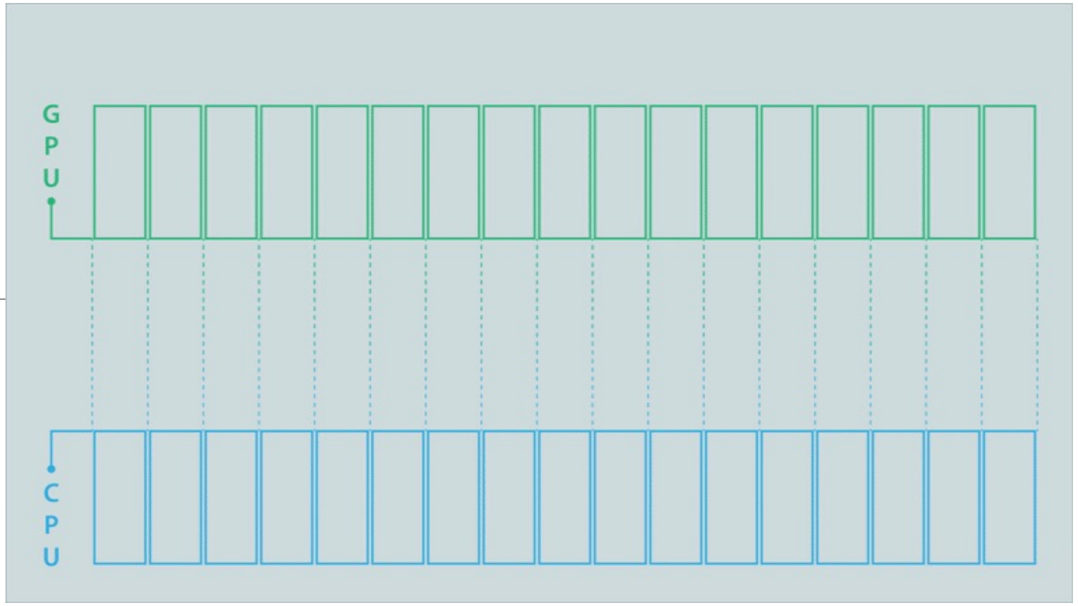
```
__device__ kernel(..., MPI_Request request)
{
  int i = my_partition[my_id];
  /* Compute and fill partition i then mark
ready: */
  MPI_Pready(i, request);
}
```

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

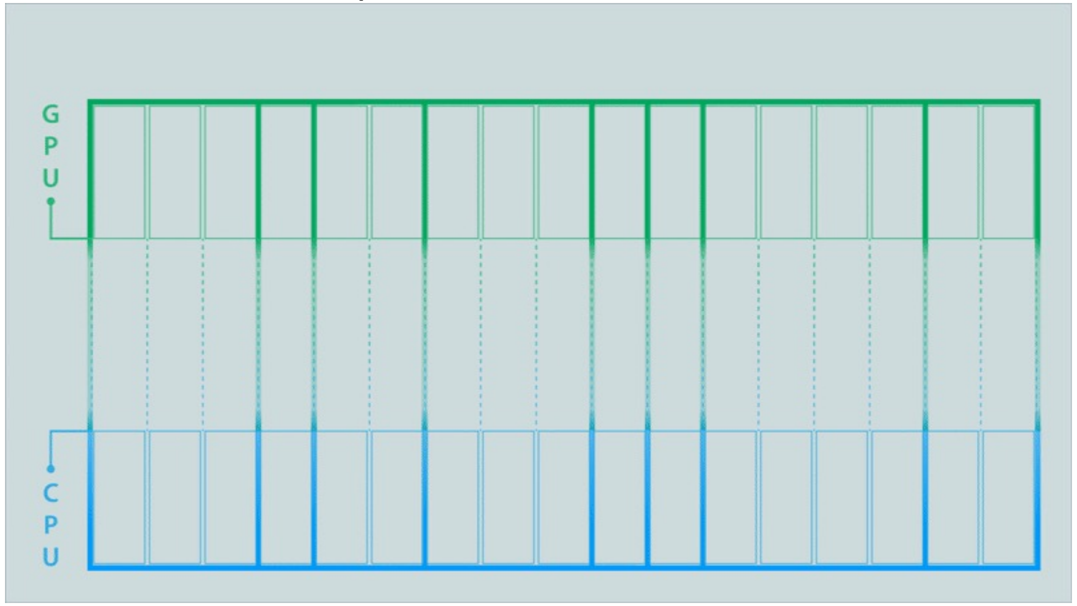
Non-partitioned



Naïve Partitioned



Optimized Partitioned



Thank you

Questions?