
October 29-31, 2024
ALCF Hands-on
HPC Workshop

MPI

Ken Raffenetti
Research Software Engineer
Mathematics and Computer Science Division

October 29, 2024

Agenda

- MPI Background
- Costs of Unintended Synchronization
- Nonblocking vs. Asynchronous
- GPU-aware MPI
- MPI on ALCF Polaris and Aurora

What is MPI?

- MPI: Message Passing Interface
  - The MPI Forum organized in 1992 with broad participation by:
    - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
    - Portability library writers: PVM, p4
    - Users: application scientists and library writers
    - MPI-1 finished in 18 months
  - Incorporates the best ideas in a “standard” way
    - Each function takes fixed arguments
    - Each function has fixed semantics
      - Standardizes what the MPI implementation provides and what the application can and cannot expect
      - Each system can implement it differently as long as the semantics match
- MPI is not…
  - a language or compiler specification
  - a specific implementation or product

MPI-1

- MPI-1 supports the classical message-passing programming model: basic point-to-point communication, collectives, datatypes, etc.
- MPI-1 was defined (1994) by a broadly based group of parallel computer vendors, computer scientists, and applications developers.
  - —2-year intensive process
- Implementations appeared quickly and now MPI is taken for granted as vendor-supported software on any parallel machine.
- Free, portable implementations exist for clusters and other environments (MPICH, Open MPI)

Following MPI Standards

- MPI-2 was released in 1997
  - —Several new features including MPI + threads, MPI-I/O, remote memory access functionality and many others
- MPI-2.1 (2008) and MPI-2.2 (2009) were released with some corrections to the standard and small features
- MPI-3 (2012) several new features
  - —Updated RMA chapter, nonblocking collectives, Fortran 2008 binding …
- MPI-3.1 (2015) minor corrections and features
- MPI-4 (June 2021) several new features
  - —Large count, Persistent collectives, MPI Sessions, Partitioned Communication, MPI_T Events, …
- MPI-4.1 (November 2023) minor corrections and features, latest version of standard
- The Standard itself:
  - —at http://www.mpi-forum.org
  - —All MPI official releases, in both PDF and HTML

Important considerations while using MPI

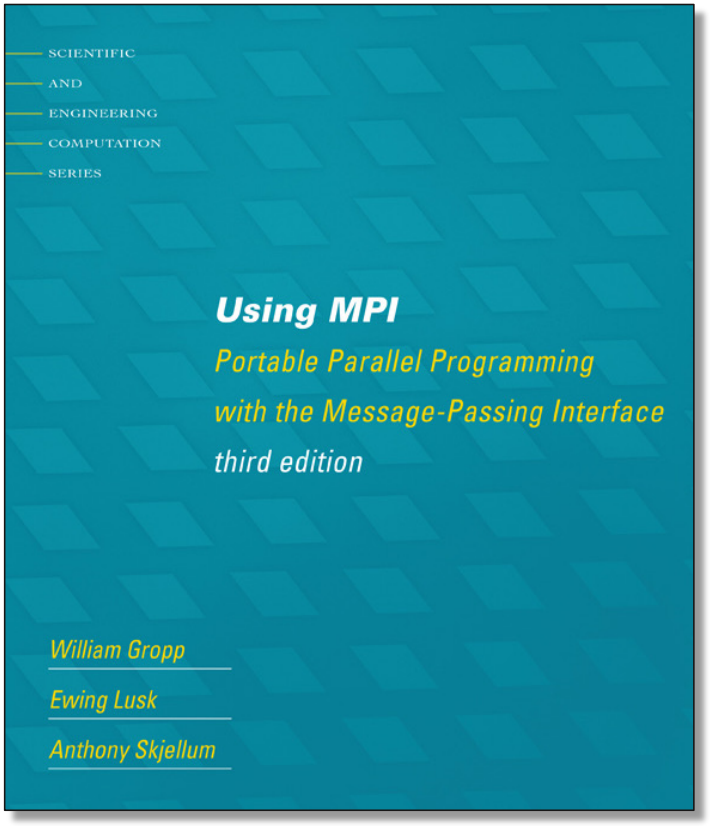
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

Web Pointers

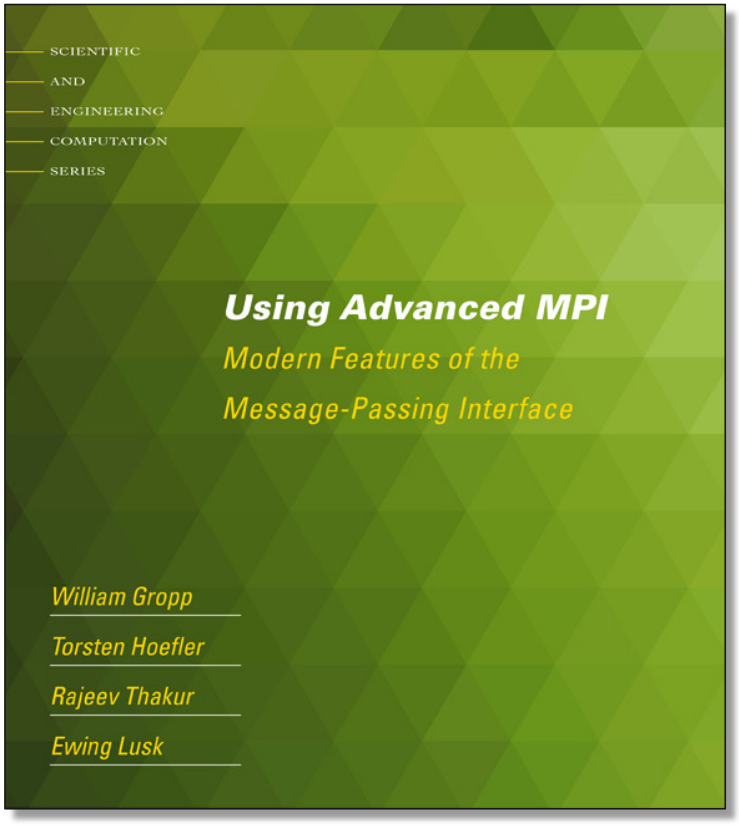
- MPI Standard : http://www.mpi-forum.org/docs/
- MPI Forum : http://www.mpi-forum.org/

- MPI implementations:
  - —MPICH : http://www.mpich.org/
  - —MVAPICH : http://mvapich.cse.ohio-state.edu/
  - —Intel MPI: http://software.intel.com/en-us/intel-mpi-library/
  - —Microsoft MPI: https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi
  - —Open MPI : http://www.open-mpi.org/
  - —IBM Spectrum MPI, Cray MPICH, ParaStation MPI, …
- General MPI Education
  - —https://rookiehpc.org/mpi/
  - —https://mpitutorial.com/tutorials/

Tutorial Books on MPI



Basic MPI



Advanced MPI, including MPI-3

This Tutorial

- Assume familiarity with MPI point-to-point and collective messaging
- Highlight a few important usage issues for performance
  - —Example codes in C
- GPU-aware MPI
- Polaris and Aurora information

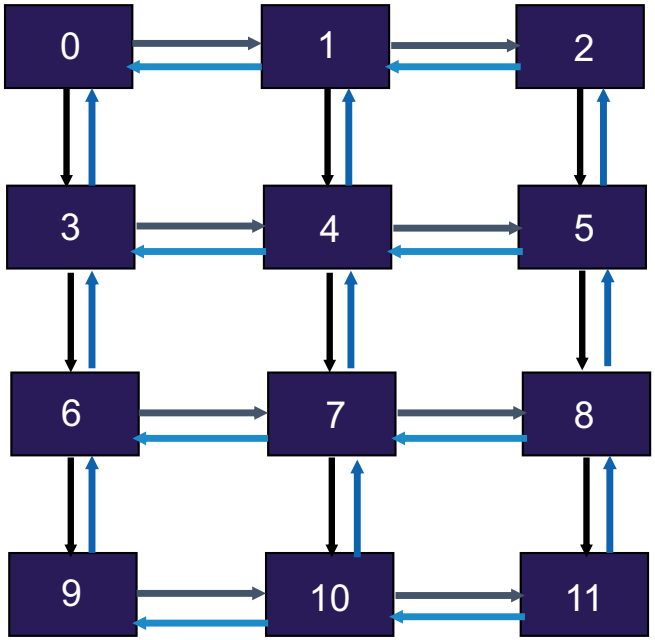
Costs of Unintended Synchronization

Hot Spots

- Simple operations can give surprising performance behavior
- Examples arise even in common grid exchange patterns
- Message passing illustrates problems present even in shared memory
  - —Blocking operations may cause unavoidable stalls

Mesh Exchange

- Exchange data on a mesh



Example Code

```
for (i = 0; i < n_neighbors; i++) {
   MPI_Send(sbuf[i], len, MPI_DOUBLE, nbr[i], tag, comm);
}
for (i = 0; i < n_neighbors; i++) {
   MPI_Recv(rbuf[i], len, MPI_DOUBLE, nbr[i], tag, comm, status);
}
```

- See unintended-sync/unintended-sync1.c

Deadlocks!

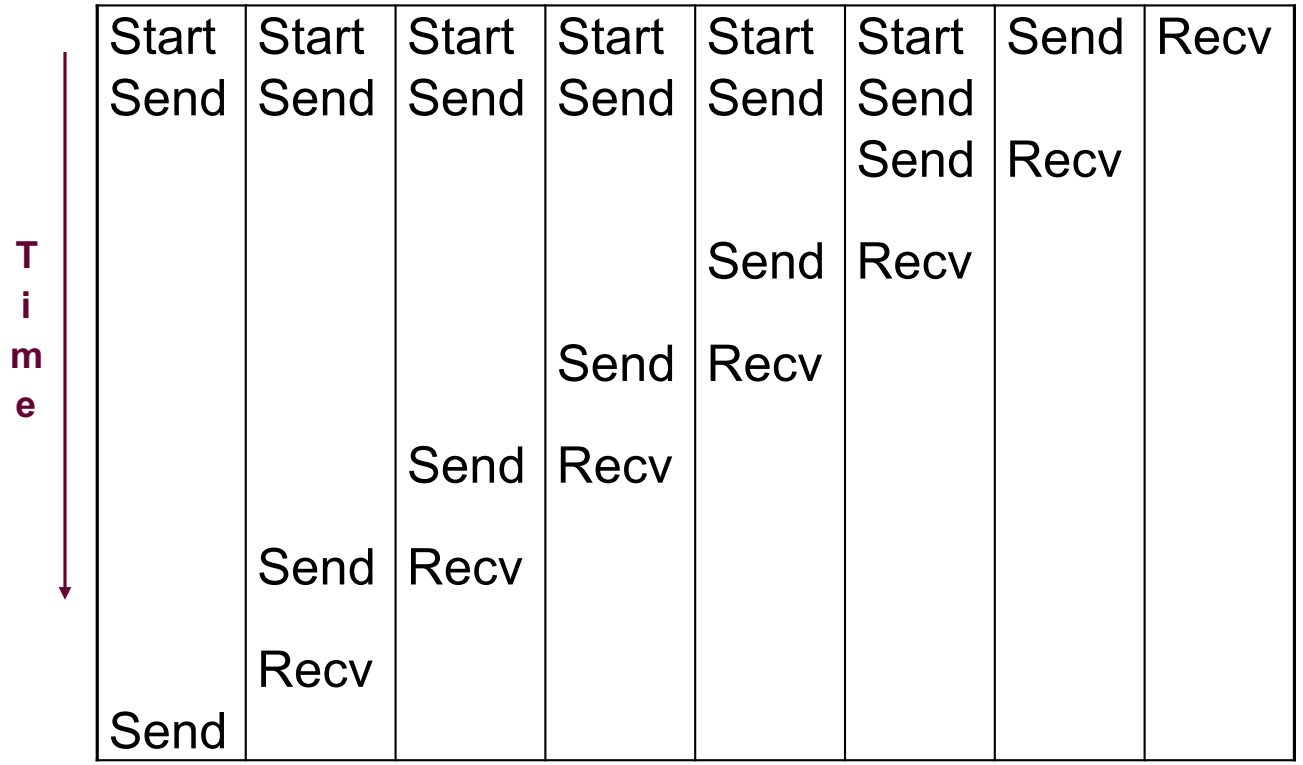
- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of

```
if (has down nbr) {
    MPI_Send( … down … );
}
if (has up nbr) {
    MPI_Recv( … up … );
}
…
```

  sequentializes (all except the processes on the bottom of the mesh)

- See unintended-sync/unintended-sync2.c

Sequentialization



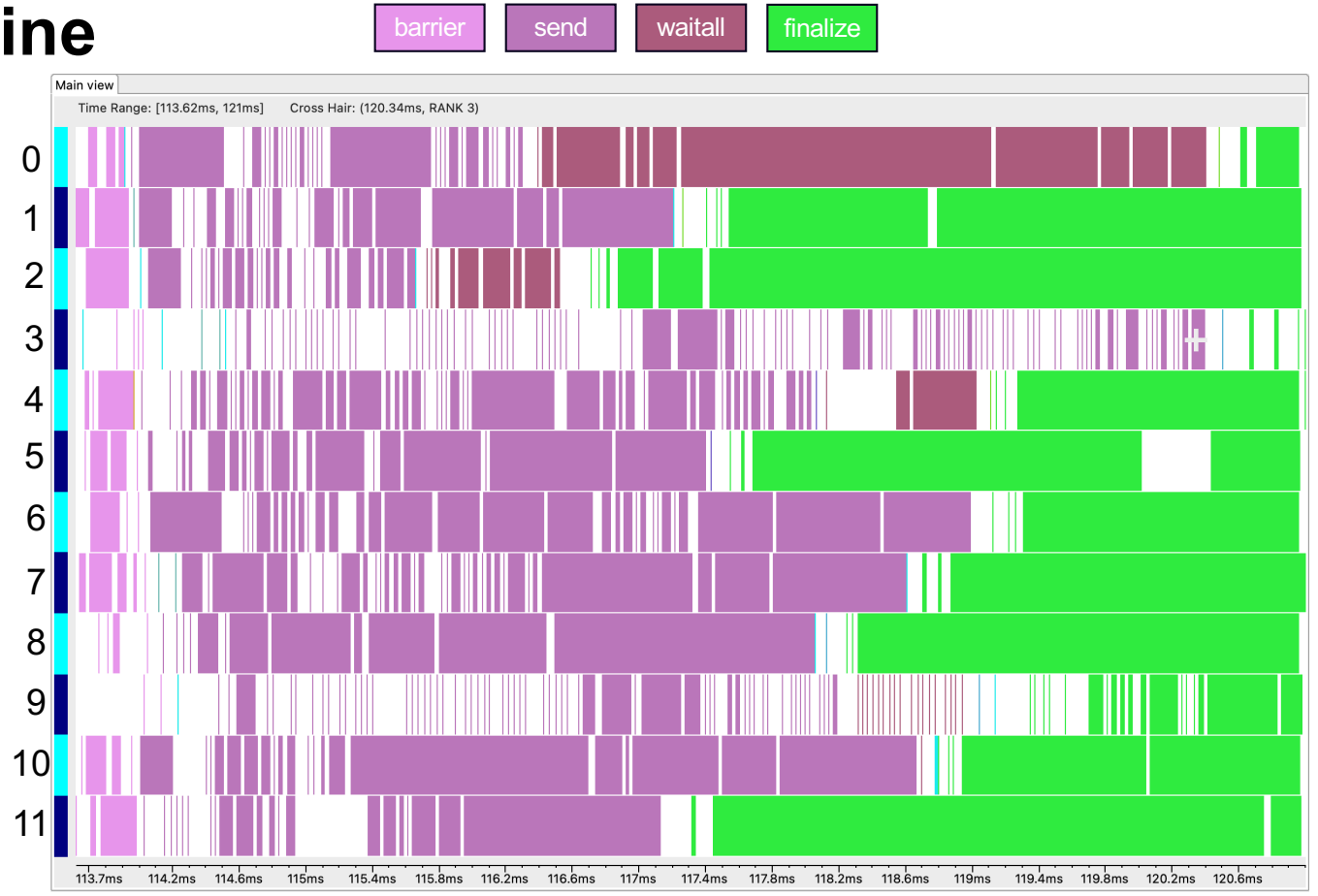
Fix 1: Use Irecv

```
for (i = 0; i < n_neighbors; i++) {
   MPI_Irecv(rbuf[i], len, MPI_DOUBLE, nbr[i], tag, comm, &requests[i]);
}
for (I = 0; I < n_neighbors; i++) {
   MPI_Send(sbuf, len, MPI_DOUBLE, nbr[i], tag, comm);
}
MPI_Waitall(n_neighbors, requests, statuses);
```

- Does not perform well in practice. Why?

- See unintended-sync/unintended-sync3.c

Timeline


barrier
send
waitall
finalize
Main view
Time Range: [113.62ms, 121ms]
Cross Hair: (120.34ms, RANK 3)
0
1
2
3
4
5
6
7
8
9
10
11
113.7ms
114.2ms
114.6ms
115ms
115.4ms
115.8ms
116.2ms
116.6ms
117ms
117.4ms
117.8ms
118.2ms
118.6ms
119ms
119.4ms
119.8ms
120.2ms
120.6ms

Why?

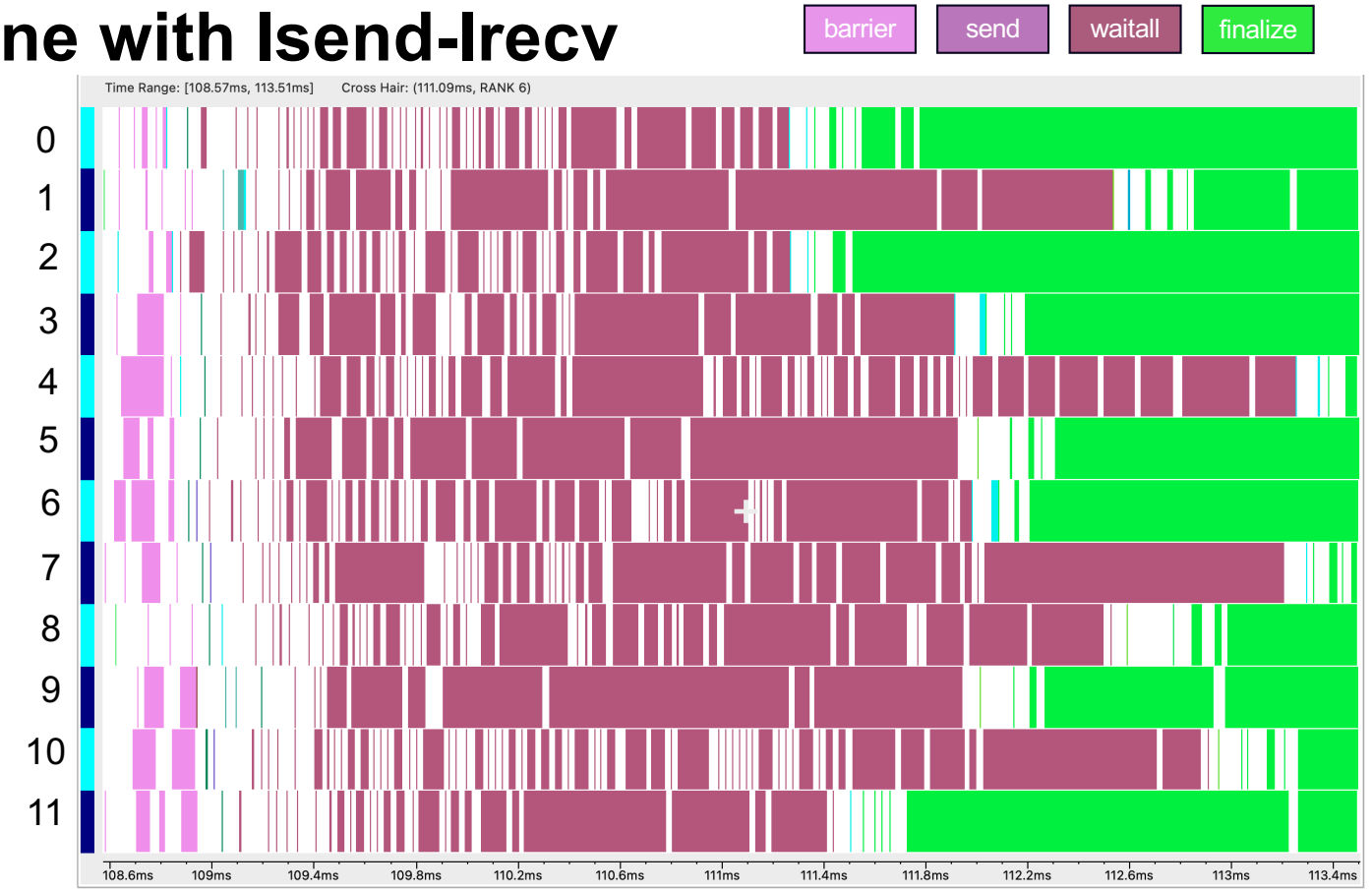
- Sending is delayed when there is contention at the receiver
- Available bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory

Fix 2: Use Isend and Irecv

```
for (i = 0; i < n_neighbors; i++) {
        MPI_Irecv(rbuf[i], len, MPI_DOUBLE, nbr[i], tag, comm, &requests[i]);
}
for (i = 0; i < n_neighbors; i++) {
        MPI_Isend(sbuf[i], len, MPI_DOUBLE, nbr[i], tag, comm,
                  &requests[n_neighbors+i]);
}
MPI_Waitall(2*n_neighbors, requests, statuses);
```

- See unintended-sync/unintended-sync4.c

Timeline with Isend-Irecv



Note processes 4 and 7 are the only interior processes; these perform more communication than the other processes. Overall communication time is ~30% less than previous timeline.

Lesson: Defer Synchronization

- Send-receive accomplishes two things:
  - Data transfer
  - Synchronization
- In many cases, there is more synchronization than required
- Consider the use of nonblocking operations and `MPI_WAITALL` to defer synchronization
  - Effectiveness depends on how data is moved by the MPI implementation
  - E.g., If large messages are moved by blocking RMA operations “under the covers,” the implementation can’t adapt to contention at the target processes, and you may see no benefit.
  - This is more likely with larger messages

Nonblocking vs. Asynchronous

Nonblocking MPI Communication

- Nonblocking communication allows for communication *overlap*
  - Overlap with other communication (as seen in previous example)
  - Overlap with computation

- Overlap with communication
  - Calling some variant of `MPI_TEST` or `MPI_WAIT` will progress outstanding communication
  - Actual overlap depends on underlying implementation, but in general you will observe better communication performance when grouping multiple requests together. “Message rate” is a common benchmark.

- Overlap with computation
  - If application threads are busy doing computation, communication might not progress in the background. Referred to as the “weak progress model” in the MPI community.
  - Communication progress characteristics are dependent on many factors, but not limited to:
    - System architecture
    - MPI library configuration and implementation
    - Runtime parameters
    - Process locality
    - Communication arguments, e.g. datatypes
    - …

Example Code

```
 if (rank == 0) {
     MPI_Isend(buf, COUNT, MPI_BYTE, 1, 0, MPI_COMM_WORLD, &req);
} else if (rank == 1) {
     MPI_Irecv(buf, COUNT, MPI_BYTE, 0, 0, MPI_COMM_WORLD, &req);
}
do_work(&req);
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

- See progress/async1.c in examples
  - —async1 work is a no-op
  - —async1-s work sleeps for 0.1s
  - —async1-t work sleeps for 0.05s, calls MPI_TEST, sleeps another 0.05s

Example Code

Single node of Polaris

```
raffenet@x3006c0s13b1n0$ make
mpicc     async1.c   -o async1
mpicc  -DSLEEP -o async1-s async1.c
mpicc  -DTEST -o async1-t async1.c
raffenet@x3006c0s13b1n0$ mpiexec -n 2 ./async1
send + work time 0.135587
recv + work time 0.135581
raffenet@x3006c0s13b1n0$ mpiexec -n 2 ./async1-s
recv + work time 0.234885
send + work time 0.234886
raffenet@x3006c0s13b1n0$ mpiexec -n 2 ./async1-t
recv + work time 0.236948
send + work time 0.186890
raffenet@x3006c0s13b1n0$
```

Two nodes of Polaris

```
raffenet@x3203c0s13b0n0$ mpiexec -n 2 -ppn 1 ./async1
send + work time 0.155919
recv + work time 0.155966
raffenet@x3203c0s13b0n0$ mpiexec -n 2 -ppn 1 ./async1-s
recv + work time 0.211524
send + work time 0.211380
raffenet@x3203c0s13b0n0$ mpiexec -n 2 -ppn 1 ./async1-t
recv + work time 0.210736
send + work time 0.210798
raffenet@x3203c0s13b0n0$
```

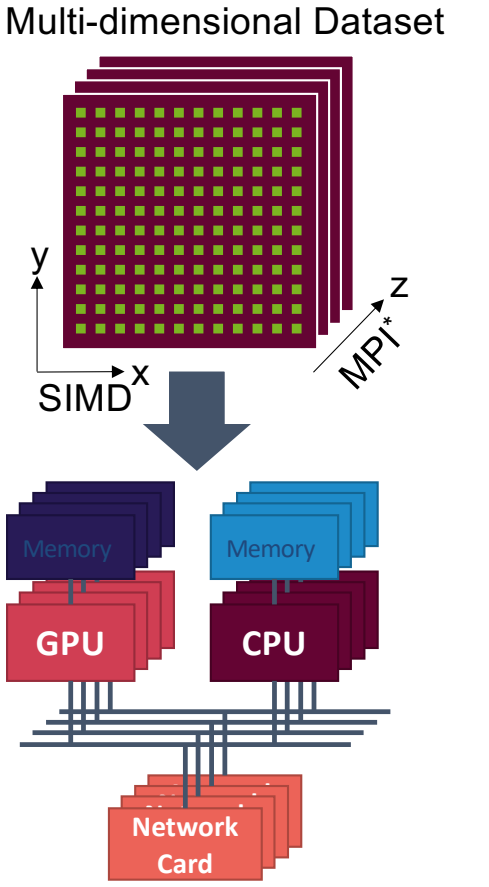
Lesson: Call MPI to make progress

- Nonblocking != Asynchronous
- Some, but not all communication will progress without user intervention
  - —E.g. rendezvous protocol implemented in software without progress thread(s)
- Break up your computation phases with occasional calls to MPI_TEST to give MPI a chance to drive communication
  - —Free up internal resources
  - —Advance protocols controlled in user space
  - —There is little penalty for calling MPI_TEST when strong progress is provided by the implementation

GPU-aware MPI

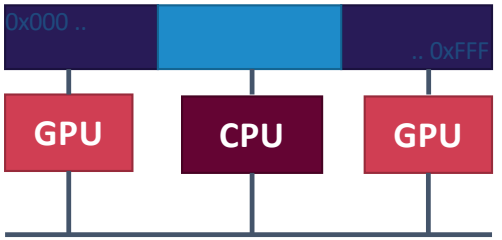
Programming Model for Accelerators

- **GPUs** are well suited for fine grain data level parallelism
- Shared Memory, Single Instruction Multiple Data (SIMD) model
- Many available compute platforms and programming frameworks (focus on their memory model and interaction with MPI)
  - —NVIDIA CUDA
  - —AMD ROCm & HIP
  - —OpenMP
  - —OpenACC (mentioned but not covered)
  - —OpenCL & SYCL



Unified Virtual Addressing (UVA)

- UVA is a memory address management system supported in modern 64-bit architectures
  - —Requires device driver support
- The same virtual address space is used for all processors, host or devices
- No distinction between host and device pointers
- The user can query the location of the data allocation given a pointer in the unified virtual address space and the appropriate GPU runtime library query APIs (“GPU-aware” MPI library)

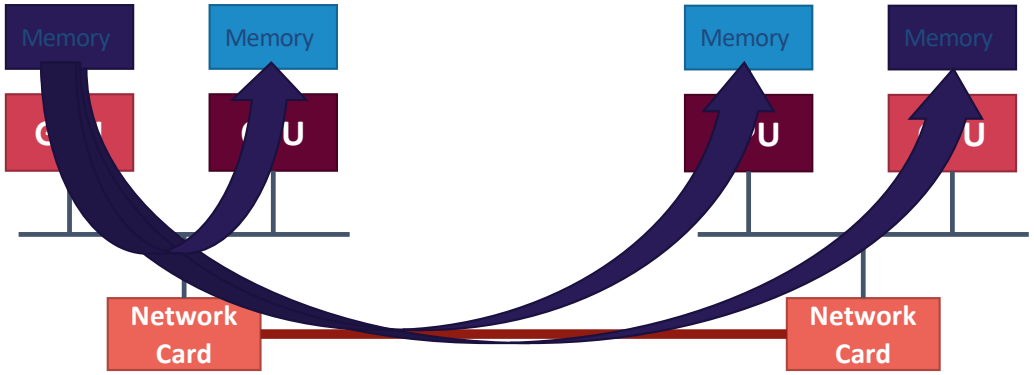


UVA: Single virtual address space for the host and all devices

Using MPI with GPU memory

GPUs have a separate physical memory subsystem

Question: How to move data between GPUs with MPI?



Answer: It depends on what GPU runtime, what hardware and what MPI implementation you are using

Simple answer: For modern GPUs and GPU-aware MPI implementations, "just like you would with non-GPU memory"

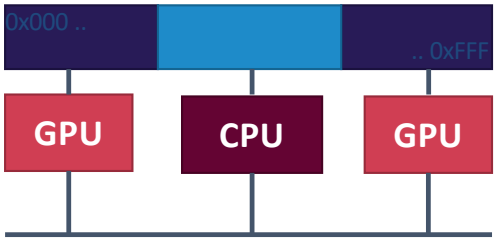
Lesson: Let MPI do the work

- There are many mechanisms for moving data efficiently to/from GPU memory
- MPI abstracts away the nitty-gritty details. For example:
  - GDRCopy for fast access to GPU memory from the CPU (when desirable)
  - GPUDirectRDMA for communicating GPU data across the network without extra copies to the CPU
  - GPU inter-process Communication (IPC) for communicating between processes on a node with one or more GPU devices. Utilize special node-level interconnects like NVLink and XeLink
  - Each of these complex techniques is employed by modern MPI libraries

- Notes for performance
  - Device allocations are most common, therefore the most optimized. Shared or managed allocations will also work, but may not be best.
  - If you suspect MPI is not performing to expectations, work with your MPI experts to identify and investigate the problem

Unified Virtual Addressing (UVA)

- UVA is a memory address management system supported in modern 64-bit architectures
  - —Requires device driver support
- The same virtual address space is used for all processors, host or devices
- No distinction between host and device pointers
- The user can query the location of the data allocation given a pointer in the unified virtual address space and the appropriate GPU runtime library query APIs (“GPU-aware” MPI library)



UVA: Single virtual address space for the host and all devices

Future directions in MPI + GPU

- General purpose GPU programming is still evolving
- Synchronization between GPUs and CPUs/NICs is a bottleneck
  - Pinning memory incurs huge overhead (up to millisecond)
  - cudaStreamSynchronization et al. are expensive
- Potential solutions
  - Memory type hints
  - Stream-aware MPI
  - In-kernel triggering

```
double *dev_buf;
cudaMalloc(&dev_buf, size);

if(my_rank == sender) {
    gpu_kernel<<<grid,block,0,stream0>>>(dev_buf);
    cudaStreamSynchronization(stream0);
    MPI_Isend(dev_buf, size, MPI_DOUBLE, receiver, 0, comm, req);
} else {
    MPI_Recv(dev_buf, size, MPI_DOUBLE, sender, 0, comm, &status);
    gpu_kernel<<<..>>>(dev_buf);
}
```

Polaris

- By default, Cray MPICH is not GPU-aware. Must be explicitly enabled by the user. Why?
  - GPU-awareness adds overhead. MPI must determine what type of memory a pointer represents.
  - In short, CPU-based communication gets better performance by disabling GPU-awareness

- How to enable GPU-aware Cray MPICH
  - `export MPICH_GPU_SUPPORT_ENABLED=1`
  - `module load craype-accel-nvidia80`
  - `cc -cuda foo.cu -o foo`

- Notes on usage
  - Use the official Cray compiler wrappers `cc, CC, ftn. mpicc, mpicxx, mpifort` may be present but WILL NOT WORK for GPU applications.
  - Cray MPICH is based on the MPICH 3.4.x release series. Supports MPI 3.1 standard.
  - Be aware of which ranks use which GPUs
    - https://github.com/argonne-lcf/GettingStarted/tree/master/Examples/Polaris/affinity_gpu for scripts to set GPU affinity for MPI ranks
    - Also try `saxpy.cu` example on Polaris to practice
  - Take care to synchronize CUDA stream before passing buffers to MPI

Aurora

- Aurora MPICH is GPU-aware by default
  - The expectation is that apps running on Aurora use the GPU
  - Aurora MPICH is purpose built for Aurora. It is a separate distribution from the widely-used Intel MPI.

- How to *disable* GPU-aware Aurora MPICH
  - `export MPIR_CVAR_ENABLE_GPU=0`

- Notes on usage
  - Aurora MPICH modules
    - `mpich/icc-all-pmix-gpu/20240717`
    - `mpich/icc-all-debug-pmix-gpu/20240717`
  - Upstream MPICH builds also available, e.g.
    - `mpich/dbg/git.063ef64`
    - `mpich/opt/git.063ef64`
  - Aurora MPICH is based on MPICH 4.2.x series. Supports MPI-4.1 standard.
  - `mpiexec` is Cray PALS, same as Polaris
    - `-l` option labels stdout/stderr with node,rank information. Useful for debugging.
  - Be aware of which ranks use which GPUs
    - https://docs.alcf.anl.gov/aurora/running-jobs-aurora/#binding-mpi-ranks-to-gpus

Questions?