GPU postmortem

- Big cost of moving data to / from GPU
- 80% on M*V
- 80% on BFS
  - 90% top down, 10% bottom up
- Bottom up, work item per directed edge, ~ 30% faster than faster, ignoring initialization, S16
- Ran out of local memory
- Top down then bottom up, beamer's algorithm

524 - Lecture 3

MPI

The problem MPI is trying to solve

- Given a bunch of machines, each with:
  - memory, compute, network
  - operating system, user land programs
- How do you:
  - start a process on each machine
  - have those processes communicate with each other
  - work on a variety of hardware and software
  - …and be reasonably fast at it.

MPI

- MPI is a *standard*. There are multiple implementations: openmpi, mpich2, etc.
  - Pick your poison. They are mostly compatible.
- Basic MPI mechanisms:
  - Spawn a bunch of processes
  - Send/receive messages
  - Barrier processes

MPI Continued.

- MPI is an example of a **message passing** framework.
- Key ideas for you as a developer:
    - Each process in the group has its own private address space.
    - You must explicitly send/receive messages to move data around
    - You must be synchronize across processes when necessary
        - Communication is synchronization
- MPI is derived from a long history in the supercomputing space.
    - As such, it’s terminology is stupid. Sorry, makes sense if you are a physicist.
    - It’s API is overly complicated at times. But that’s life.

Starting MPI

```
int my_process, processes;


//// Initial the MPI subsystem.  Yes, this modifies the arguments
  MPI_Init(&argc, &argv);

//// find out my process ID
  MPI_Comm_rank(MPI_COMM_WORLD, &my_process);

////  Find out how many processes there are
  MPI_Comm_size(MPI_COMM_WORLD, &processes);
```

Sending data

```
int MPI_Send(void *buf,
  int count,
  MPI_Datatype datatype,
  int dest,
  int tag,
  MPI_Comm comm)
```

```
MPI_Send(&my_pid, sizeof(int), MPI_BYTE,
    process, 0, MPI_COMM_WORLD);
```

Receiving data

```
int MPI_Recv(void *buf,
 int count,
 MPI_Datatype datatype,
 int source,
 int tag,
 MPI_Comm comm,
 MPI_Status *status)
```

MPI_Recv(&other_pid, sizeof(int), MPI_BYTE,
process, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

Building and running

- MPI programs require a header file and library.
  - *Typically* you compile MPI programs with mpicc or mpic++ which sets the include path and links the correct library
    - Not necessary. mpicc is just a wrapper around your C compiler
- MPI programs are typically run with the “mpirun” command.
  - While also not strictly necessary, it’s very hard not to.

MPI Example

Useful MPI Stuff (in my opinion)

- MPI_Barrier
- MPI_Bcast —- broadcast
- MPI_Isend, MPI_Irecv — non-blocking send and receive
    - MPI_Alloc_Mem —- allocate receive buffers
    - It’s important for performance to post receive buffers before sends occur.

Stuff I avoid

- Partitioning my communication world, I always use MPI_COMM_WORLD and forget about anything else
- Tags. I always use 0 or MPI_ANY_TAG. There is talk about greater use of tags for GPU communication and such. But generally this is unused.
- MPI data types. Complete waste of time. Just use MPI_BYTE and send/receive the exact number of bytes.
- MPI threading. I’ve **never** come across an MPI multithreaded implementation that wasn’t broken. Just use the single threaded one and lock around it.
- The file I/O. This looks useful, but for cluster file I/O we use HDFS.
- The remote op and reduction operators. I’ve always just implemented this manually.
- The All to All and SendRecv functions. Maybe these are useful on really large systems, but I never found it worth the time to bother.

Common MPI bugs

- Receive without Send
  - Or more generally, mismatched send/receives
  - Some but not all nodes sending
- Messing up the arguments to Send/Recv
  - That's why I always use MPI_BYTE, tag=0 and MPI_COMM_WORLD
- Deadlock on blocking send/receive
- Performance bug:
  - You must send/receive in big chunks.
  - Hard to balance work well.
- *Subtle bug:* Address space randomization means function pointers passed across machines are not valid.

Why you should use MPI (in scientific computing)

- Your fellow programmers will know what you are talking about. It is *the* standard for this thing.
- There are vendor-optimized and supported versions (Cray, IBM, Intel, …)
- It is the thing other subsystems plug into. For example, we use a user-mode RDMA library for our infiniband interconnect.
  - Despite trying, my students could not beat the performance of this when talking to the infiniband verbs layer directly…

Why you (probably) shouldn't use MPI in a commercial setting

- You have to buy the whole enchilada.
  - Do you really think your program can start with MPI_Init(argc, argv)? Seriously?
  - MPI comes from the scientific computing world. It didn't come from the universe of plugging together hundreds of software packages across dozens of systems and standards.
- In my experience, commercial software has harder robustness constraints.
  - MPI doesn't understand what it means for a node to go down.
    - In the commercial world, you have to build around crashing.
- Despite this, understanding **message passing** as a fundamental concept is important. Even if you end up having to use some other library.

Final thoughts

- You can mix and match
  - MPI + pthreads
  - MPI + openmp
  - MPI + openmp + openCL
- Active Messages
  - GASNet
- PGAS, Partitioned Global Address Space
  - Grappa