Message Passing Interface (MPI)
Today’s Schedule

- Parallel Computing Background
- Diving in: MPI
- The Jetson cluster
Today’s Schedule

- Parallel Computing Background
- Diving in: MPI
- The Jetson cluster
Now that we’re all C masters, we can move on to the good stuff: parallelism

Specifically, we’ll be looking at distributed memory systems for the next section of the course

In these systems, we have a few different elements:

- Physical machines
- Processors
- Processor cores
Parallel computing can be summed up with a simple motto:

“Divide and conquer”

Let’s take a problem, break it into smaller pieces, and then have multiple cores/processors/machines work on it all at once

Challenge: getting all these processors to work together
We can use several different strategies to parallelize applications.

The first approach we’ll examine in this class is MPI:
- Message Passing Interface

MPI has a lot of functionality, but at its core is based on a very simple idea:
- Running multiple copies of your program
  - (Sometimes even across multiple computers)
Before we dive in, we need to take a look at the hardware architectures behind parallel systems.

There are several types:
- SISD, SIMD, MISD, MIMD

These classifications were proposed by Michael J. Flynn in 1966
- *Flynn’s Taxonomy*

See: [https://en.wikipedia.org/wiki/Flynn%27s_taxonomy](https://en.wikipedia.org/wiki/Flynn%27s_taxonomy)
Flynn’s Taxonomy: Breakdown

- Each architecture is composed of three elements
  - PUs – processing units / processing elements
  - The instruction pool
    - Your program, translated to machine code
  - The data pool
    - The data you’re working with
SISD

- Single instruction, single data
  - One CPU, one core, one thread (uniprocessor)
  - One pool of memory
  - One thing at a time!
- PCs up until 2010 or so

Source: Cburnett. CC BY-SA 3.0. https://en.wikipedia.org/wiki/Flynn%27s_taxonomy
SIMD

- Single instruction, multiple data stream
- Each PU executes the same instructions on a different piece of the data
- Great for highly-parallel workloads (GPUs)

Source: Cburnett. CC BY-SA 3.0. https://en.wikipedia.org/wiki/Flynn%27s_taxonomy
MISD

- Multiple instruction, single data
- More uncommon
- Here, the data is fed to multiple PUs
- Each PU executes the same instructions
- Then the results are compared
  - Fault tolerance

Source: Cburnett. CC BY-SA 3.0. https://en.wikipedia.org/wiki/Flynn%27s_taxonomy
- Multiple instruction, multiple data
- Nodes work independently
- Multi-core PCs, distributed systems
- Our focus in this class

Source: Cburnett. CC BY-SA 3.0. https://en.wikipedia.org/wiki/Flynn%27s_taxonomy
In the best case scenario, doubling the number of cores will halve your execution time.

In practice, this is difficult.

- There is **overhead** associated with parallelism.

Amdahl’s law puts a bound on potential speedup:

\[ S_N = \frac{1}{(1 - P) + \frac{P}{N}} \]

- \( S \) – speedup
- \( P \) – parallelizable portion
- \( N \) – number of PUs
Amdahl’s Law [2/2]

By Daniels220 at English Wikipedia, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=6678551
Today’s Schedule

- Parallel Computing Background
- Diving in: MPI
- The Jetson cluster
Message passing is the most common paradigm for programming distributed memory systems.

Processors coordinate their activities by sending messages to each other across the network:
- Infiniband
- Ethernet

Message Passing Interface, or just MPI, gives us C functions to do this.
With MPI, we won’t just be running a single program anymore

- Now, we’ll deal with multiple processes

- These processes are identified by nonnegative integer ranks

- If there are $p$ processes, the processes will have ranks $0, 1, 2, \ldots, p - 1$
Installing MPI

- On Linux, it’s as easy as installing the `openmpi` group of packages:
  - `apt-get install openmpi-bin openmpi-common libopenmpi`
- Newer Macs don’t come with MPI already installed, so you will need a 3rd party package manager:
  - Homebrew ([http://brew.sh](http://brew.sh)), MacPorts
  - Then install `openmpi`
- Windows: cygwin `openmpi` package is `buggy`
To compile your MPI code, you’ll need a new command:

- mpicc

This is just a wrapper around gcc or whatever compiler you have on your system

- Sets up compilation with the correct libraries and options
You can’t just run `a.out` or whatever your executable is called

Instead, you’ll need to use an MPI launcher:

- `orterun -n 4 ./a.out`  
  (will run `a.out` with four processes)
- `mpiexec -n 4 ./a.out`  
  (exactly the same thing!)
- `mpiexec -n 4 --hostfile=jets.txt ./a.out`  
  (runs on multiple machines)
Hello World

- As usual, we need to write a “hello world” application as our first step!
- In MPI, we can print out some more information: the hostname of the machine, its rank, and the total number of processes
- Let’s try this out...
MPI_Init()

- Needs to be run before you do anything else
- You can pass in NULL for both of its arguments, or you can pass in the argc and argv command line arguments
  - If you do that, it’ll remove any orterun/mpiexec/mpirun-related stuff from the command line
MPI Communicators

- You might’ve notice MPI_COMM_WORLD in the example
- This is the global communicator group
- You can create groups of processes to coordinate your distributed applications
  - For instance, maybe one group will work on the upper-left corner of an image
Helpful Functions

/* Total number of processes in this MPI communicator */

int comm_sz;
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

/* Get the rank of this processor */

int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/* Get the host name of this processor */

char hostname[MPI_MAX_PROCESSOR_NAME];
int name_sz;
MPI_Get_processor_name(hostname, &name_sz);
Cleaning Up

- At the end of your MPI program you must call:
  
  ```c
  MPI_Finalize();
  ```

- This cleans up all the MPI state information that was being held while your program ran

- Finishes all pending communications

- After calling this, executing any MPI function will raise an error
At a basic level, all MPI does is clone your process and run it multiple times.

Without any special intervention, the processes will all just do the same thing.

However, we can branch based on process ranks to organize processing activities and communicate.
Today’s Schedule

- Parallel Computing Background
- Diving in: MPI
- The Jetson cluster
The Jetsons

- We have 24 NVIDIA Jetson TK1 machines
- These are ARM-based boards for parallel computing and GPU programming using NVIDIA CUDA
- Hardware:
  - Quad-core ARM CPU
  - NVIDIA Kepler GPU with 192 CUDA Cores
  - 2 GB Memory
Jetson TK1
Jetson TK1

- Somewhat like a Raspberry Pi on steroids
- We’ll use this cluster for the rest of the semester
  - (Including GPU programming)
To reach the jet machines, you will need to use **ssh**
- You may have done this in previous courses

Furthermore, you will need **passwordless** ssh set up in order to effectively use MPI

This allows MPI to distribute your program across multiple servers
To get on the Jetson machines, you first need to log into stargate.cs.usfca.edu

Then ssh to:

- jet01
- jet02
- ...
- jet24