CS 220: Introduction to Parallel Computing

Communication Patterns

Lecture 18

Today's Agenda

- General Announcements
- Communication Approaches
- Tree Broadcast/Reduce
- Bitwise Operations

Today's Agenda

- **General Announcements**
- Communication Approaches
- Tree Broadcast/Reduce
- Bitwise Operations

General Announcements

- P1 Grades are on Canvas
- Check your repo for a file named ‘grade-info.md’
  - This contains the outputs from the test cases we ran on your code
  - Any deductions are listed at the end
- Please let me know if you have any questions!

Non-Blocking Communication

- I mentioned we'd go over non-blocking communication for P2
- Let's take a look at the example code:
  - blocking.c
  - non-blocking.c

Today's Agenda

- General Announcements
- **Communication Approaches**
- Tree Broadcast/Reduce
- Bitwise Operations

Communication

- We've looked at a few different communication paradigms:
  - Point-to-point
  - Ring
  - One-to-many
- What are the advantages/disadvantages of each?

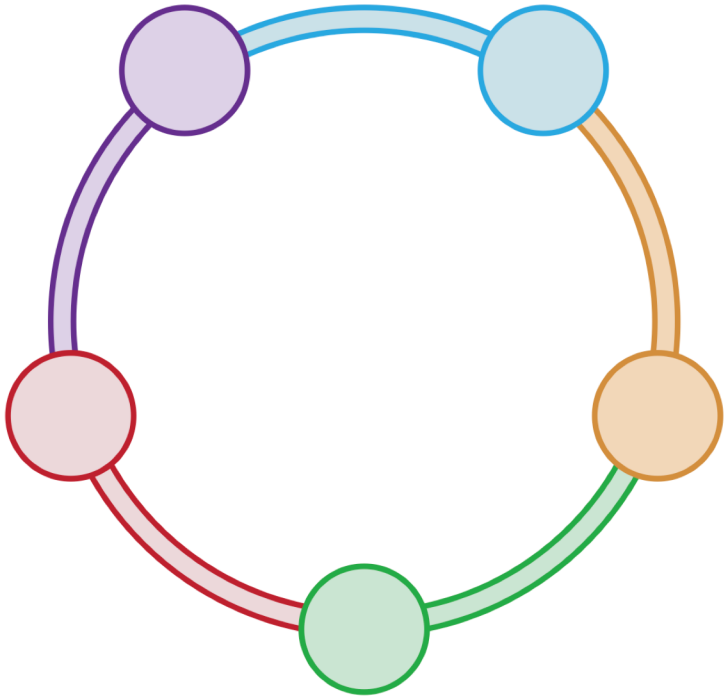
Point-to-Point Communication

- Process A sends a message to Process B
  - MPI_Send
  - MPI_Recv
- Or in other words:
  - Read a buffer on Process A
  - Copy the value to a buffer on Process B
- Free-form!
- Lets us define whatever communication we want

Ring Topology

- Here, we form a ring of processes
- Each process only needs to know about its:
  - Predecessor
  - Successor
- Great for **workflows**: each process does something different and then sends its result to the next
- Not so great for doing a single thing in parallel!

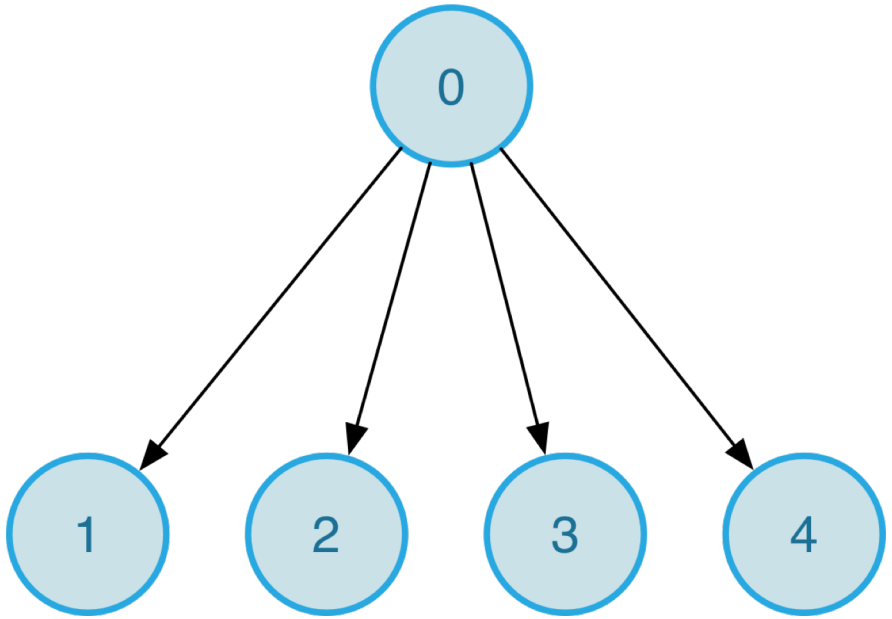
Ring Layout



One-to-many

- Sometimes we need to pick a single process and let it coordinate things
- In this case, that process may **broadcast** to others in a particular MPI communicator
  - MPI_Bcast
    - Optimized by the MPI library
- We commonly use **rank 0** to read input, organize data, or parameterize our computations

Broadcasting



Broadcast Note

- One unintuitive thing about MPI_Bcast: **all** the participating processes call the function
- Process 8 can't call MPI_Bcast and then the rest just MPI_Recv to get the value!
- So remember folks: when you use MPI_Bcast, make sure all the processes involved are calling the function!

Implementing our own Broadcast

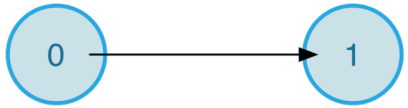
- Let's say we want to talk to everybody, but don't want to make them all call MPI_Bcast
- We can implement this with MPI_Send, right?

```
int i;
for (i = 0; i < comm_sz; ++i) {
    if (i != my_rank) {
        MPI_Send(buffer, 100, MPI_CHAR, i, 0, MPI_COMM_WORLD);
    }
}
```

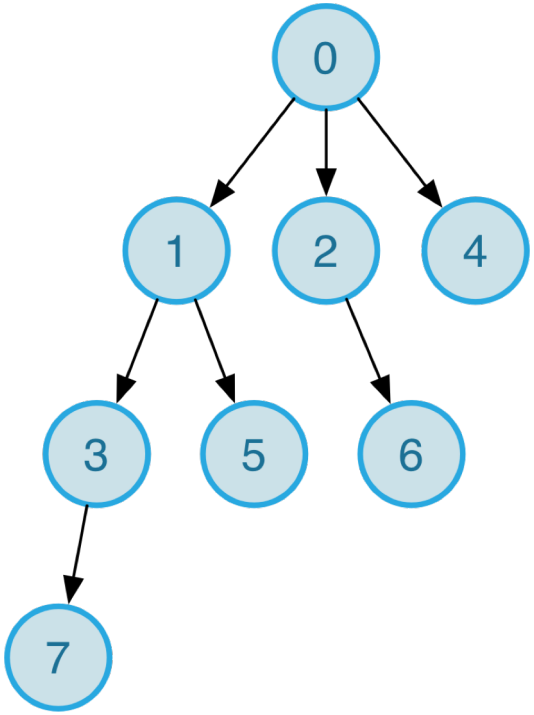
Broadcast Performance

- Our loop-based broadcast is not efficient
  - Why?
- We have to send to all the processes in order
- After we send the data to process 1, what does it do?
  - It moves on (or waits around for something else to happen, depending on your code!)
- It would be better to have the other processes help us out with our broadcast, right?

Tree-Structured Broadcast



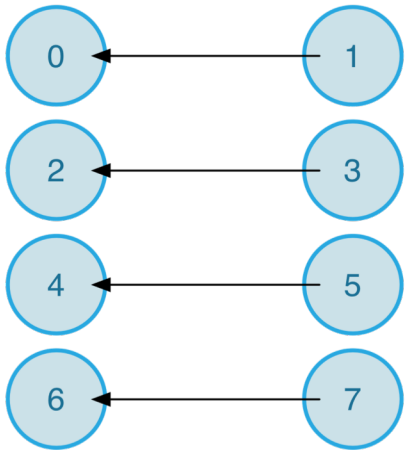
Tree Layout



Tree Reduce

- We can also do this in reverse to collect a value
- For instance: a global sum
  - We want to sum the numbers across all nodes
  - Rather than spamming a single node with values, let's pass the values on to our neighbors
  - Kind of like the ring topology, but much more efficient!

Tree-Structured Reduction



* That's right, I just reused the same diagram in reverse

Communication Workflow

- Pass 0:
  - Process 0 receives from 1
  - Process 2 receives from 3
  - Process 4 receives from 5
  - Process 6 receives from 7
- Pass 1:
  - Process 0 receives from 2
  - Process 4 receives from 6
- Pass 2:
  - Process 0 receives from 4

Today's Agenda

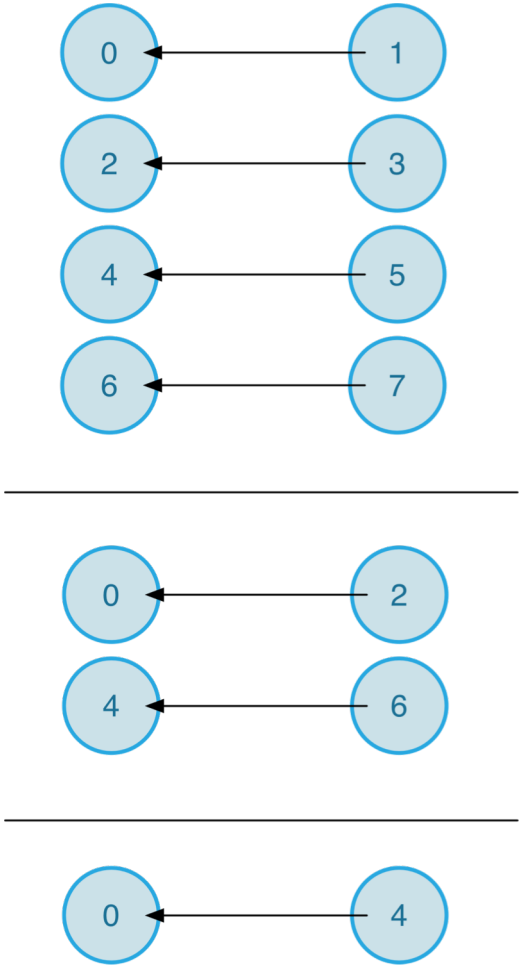
- General Announcements
- Communication Approaches
- **Tree Broadcast/Reduce**
- Bitwise Operations

Implementing Tree Reduce

- This communication pattern can save us time, but how do we actually implement it?
- We could use a bunch of `if` statements, but that seems like it would be a pain
  - Plus, how would we scale out with more processes?
- We could also compute the number of phases required, figure out who will send what, and then do it
- We have an easier way…

Looking at the Bits

- Pass 0:
  - Process 000 receives from 001
  - Process 010 receives from 011
  - Process 100 receives from 101
  - Process 110 receives from 111
- Pass 1:
  - Process 000 receives from 010
  - Process 100 receives from 110
- Pass 2:
  - Process 000 receives from 100



A Pattern Emerges…

- In Phase 0, we flip the first bit
- In Phase 1, we flip the second bit
- In Phase 2, we flip the third bit

- C has a handy way to do this:
  Exclusive Or (XOR)

Exclusive Or

- XOR is a ***bitwise operation***
- Gives **True (1)** when the number of **True** inputs is odd

Input		Output
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

- In C, use the ^ operator: **A ^ B**

Putting it Together

- To find our *"partner,"* we'll do:

```
my_rank ^ 1
```

- This works in the first phase. Then what?
  - We need to XOR with binary 010 **(2)**
  - `my_rank ^ 2`
- Ok, how about the last phase?
  - XOR with binary 100 **(4)**
  - Hrm…

Bit Shifting

- A great way to figure out what to XOR with is using a **bitmask**
- We'll start our bitmask as 1 and then **shift** its value with each phase:

```
unsigned int bitmask = 1
/* Next phase: (left shift by one) */
bitmask = bitmask << 1
```

Global Sum, In Pseudocode:

```
while not done and bitmask < comm_sz:
      partner = my_rank ^ bitmask
      if my_rank < partner:
            receive value
            update our sum
            bitmask = bitmask << 1
      else:
            send our value
            done = 1


return sum
```

Today's Agenda

- General Announcements
- Communication Approaches
- Tree Broadcast/Reduce
- **Bitwise Operations**

Bitwise Operations

- C supports the following **bitwise operations**:
  - OR (|)
  - AND (&)
  - XOR (^)
  - Unary bitwise complement or (~)
- And of course, **bit shifting**:
  - Left shift (<<)
  - Right shift (>>)

Bitwise Or

Input		Output
A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise And

Input		Output
A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Unary bitwise complement or

- Flips the bits!
  - ~ 1011 = 0100
  - ~ 1111 = 0000
  - Etc.

Printing a Number in Binary

```
/* Prints out a 32-bit unsigned integer in binary */
void print_binary32(uint32_t num) {
    int i;
    for (i = 31; i >= 0; --i) {
        uint32_t position = 1 << i;
        printf("%c",
            ((num & position) == position) ? '1' : '0');
    }
}
/* Hmm… What's uint32_t? */
```

HW6: Global Sum

- In Homework 6, we will implement the global sum
- Head to the assignments page and get started!