CS 220: Introduction to Parallel Computing

Randomness and Collective Communication

Lecture 19

Today's Agenda

- Q&A
- Random number generation
- Collective Communication

Today's Agenda

- **Q&A**
- Random number generation
- Collective Communication

Blocking or Non-Blocking?

- We've made extensive use of MPI_Send and MPI_Recv
- These are both **blocking** operations
- The program won't move on to the next instruction until the function call completes
- If we're sending and nobody hears our cry, then we'll just sit there and wait
  - Same thing for receiving…

Non-Blocking Operations

- Non-blocking functions are prefixed with an "I"
- MPI_Iprobe is our first: if there is no message available, it just returns immediately
- MPI_Isend, MPI_Irecv as well
- **Question**: should you use MPI_Isend or just MPI_Send in P2 when you shut down the threads?

Getting Everyone to Stop

- In an MPI program, all the processes work independently unless they're passing messages
- If we use blocking operations, the processes will stop
- How about getting everyone to sync up?
  - MPI_Barrier(MPI_COMM_WORLD)
- A **barrier** is a gate that only opens when **ALL** the processes call the function
  - Or: they're all executing the same function call

Today's Agenda

- Q&A
- **Random number generation**
- Collective Communication

Random

- You might've noticed two new function calls in HW6
- random() and srandom()
- These:
  - Generate a **pseudo**random number
  - **Seed** the random number generator
- So, thinking back to HW6…

Generating Random Numbers

```
srandom(my_rank + 1);
my_contrib = random() % MAX_CONTRIB;
```

- Here, we're **seeding** the random number generator with our rank + 1
- The seed is used as the starting point for the random number generation algorithm
  - Use the same seed? Then the same random numbers will be produced.

Making it Really Random

- So if we use the seed "1," we'll get the same random numbers every time
- Maybe we actually want the value to be truly random – not something we could predict ahead of time
- The common approach here is to use the current timestamp to seed the random number generator
  - If you run the program, time has continued ticking on and a new seed will be used every time
  - time() function

Seeding with time()

- Seeding the random number generator with time() doesn't actually work with MPI programs though!
- Why?
  - MPI launches your program multiple times in parallel
  - time() will return the same thing at all ranks
- To solve this problem, we shift time ahead by the rank number
  - Ensures we'll always get unique numbers each run!

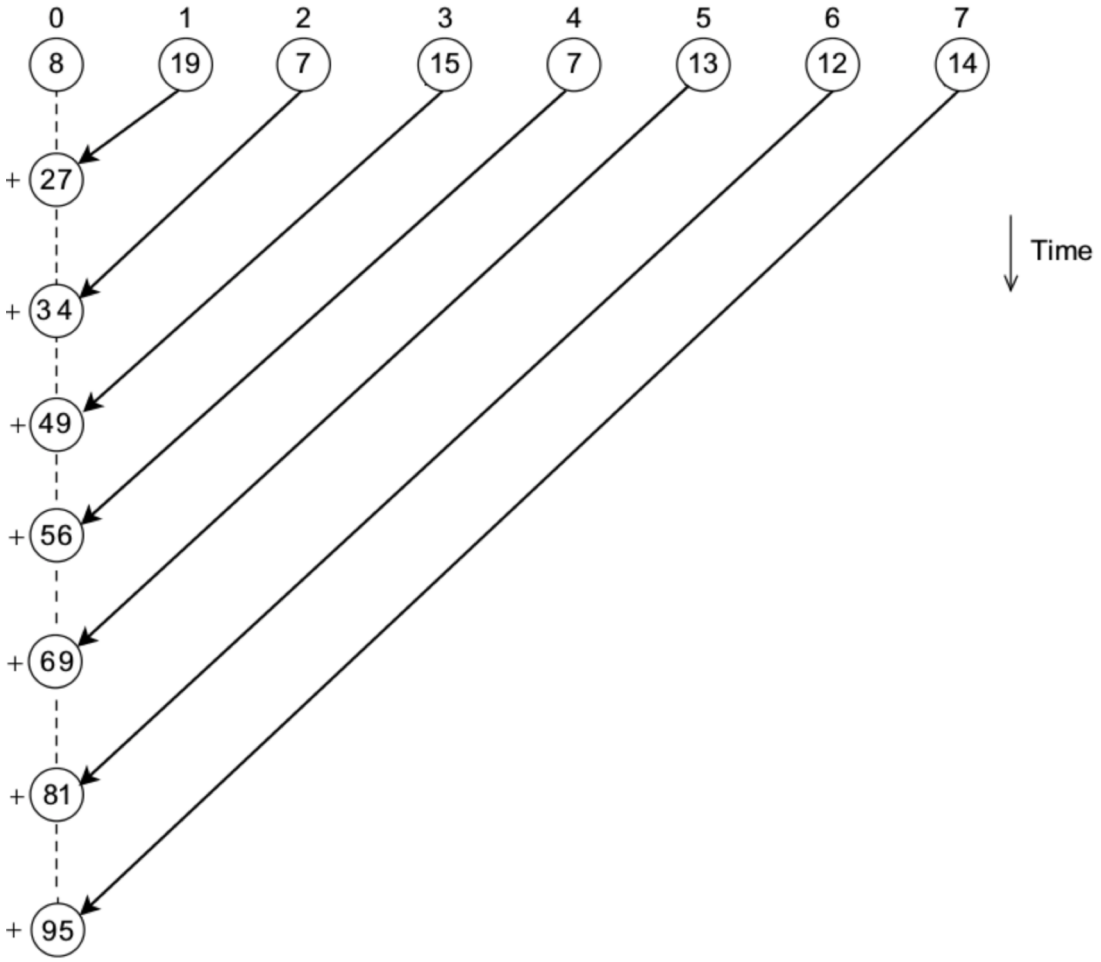
Today's Agenda

- Q&A
- Random number generation
- **Collective Communication**

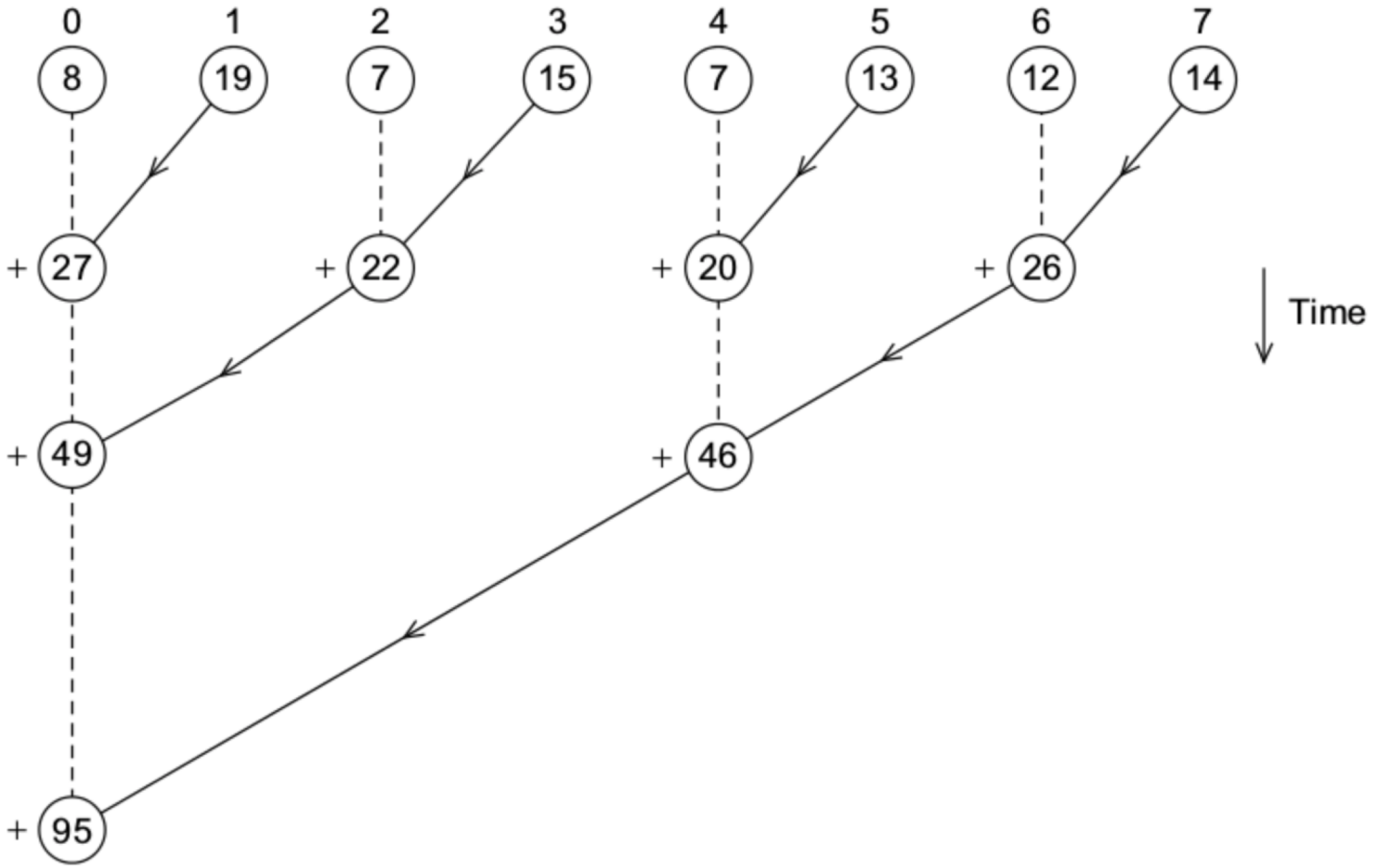
Collective Communication

- In HW6, we implemented a much more efficient global sum
- Rather than simply sending all values to a single rank (process), we split the workload up
- There are a few other types of **collective communication** to cover…

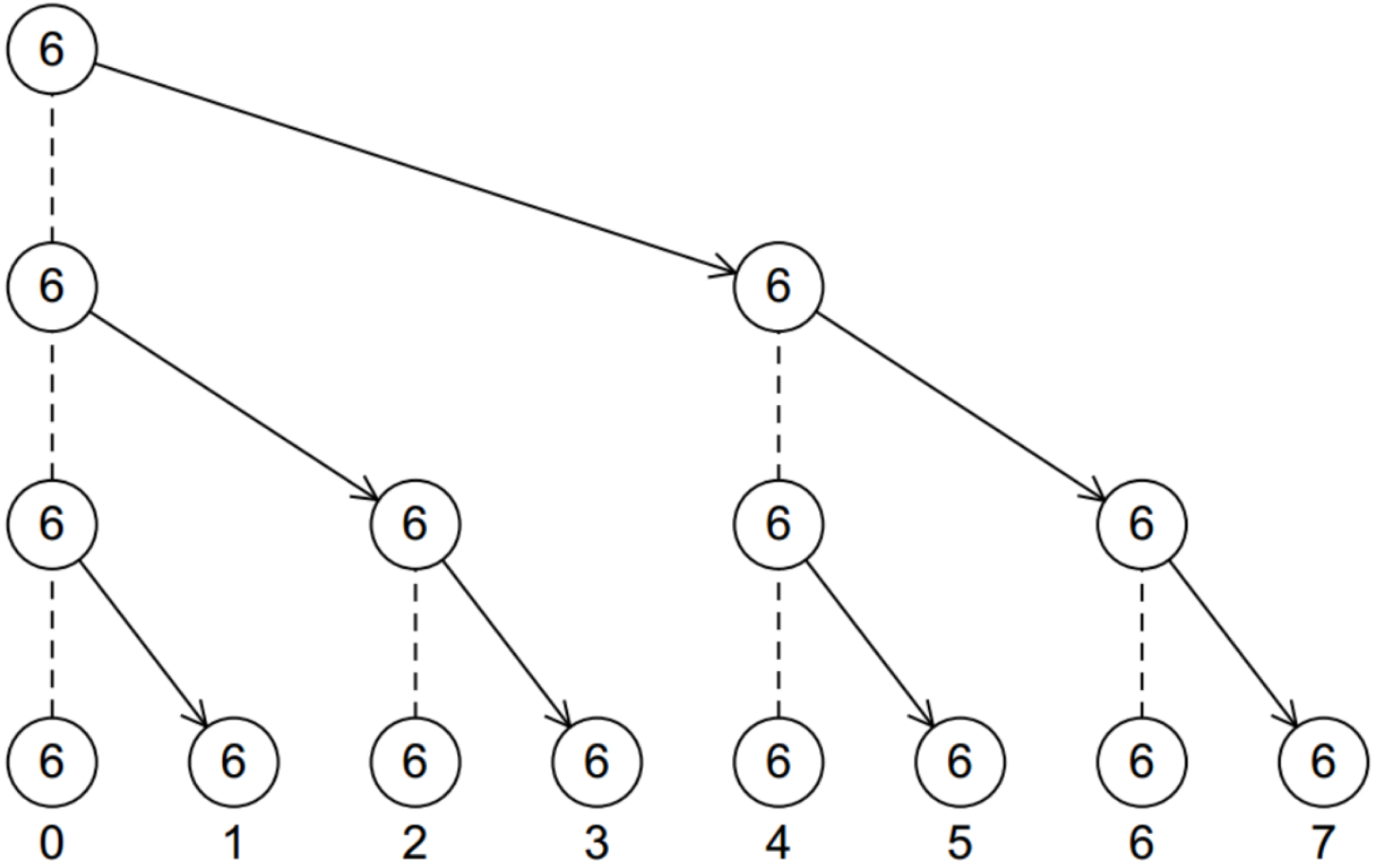
Basic All-to-One Sum



Tree Sum



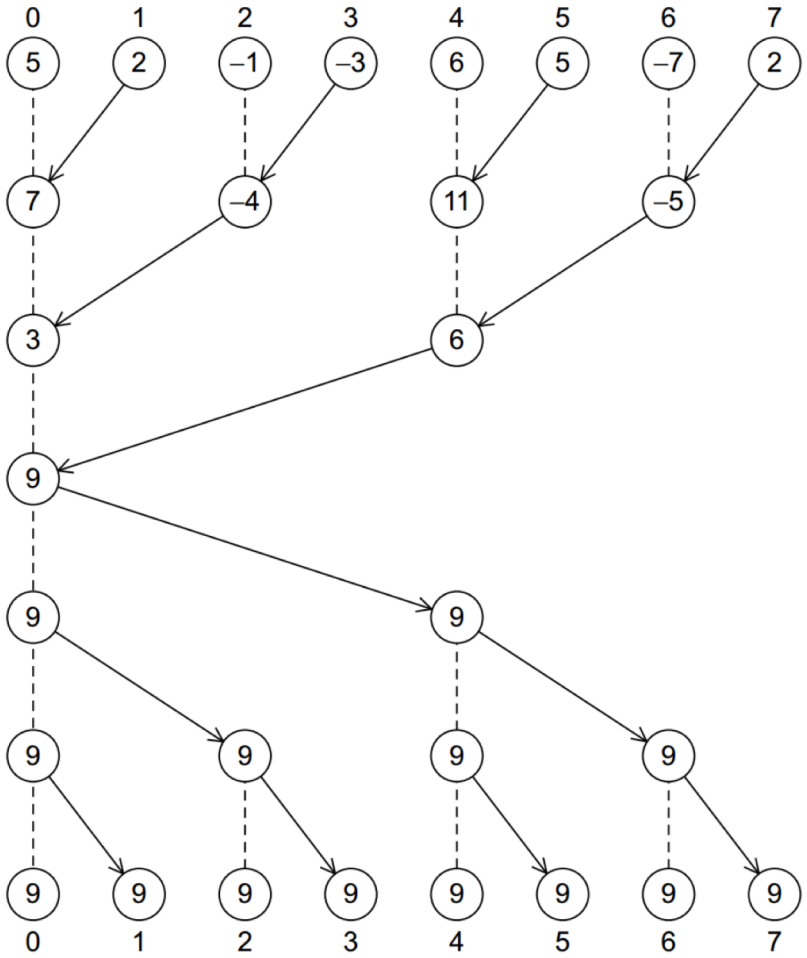
Tree Broadcast



MPI_Reduce

- As we've seen with Project 2, we can use MPI_Reduce to have MPI do a tree sum for us
- There's another, similar function: MPI_Allreduce
  - What's the difference?
- Allreduce shares the result across all the ranks
  - Whereas reduce will just give one "root" rank the answer
- How is this communication pattern implemented?

One Approach: Sum & Broadcast



Optimization: Butterfly