Message Passing Interface

Week 10

Amittai Aviram – 17 November 2020 – Boston University

CS 591 A1 – Parallel Computing and Programming

Welcome to MPI

Message Passing Interface

- First developed in 1994.
- A genereral framework for parallel computing
    - Distributed systems—co-ordinates across nodes
    - Shared-memory systems within each node
- Defines a standard interface, with various implementations
    - We use OpenMPI
- Provides a convenient abstraction for parallelism
    - Assumes that you must explicitly *send* and *receive* data from one process to another
    - Within a single process, threads can share data.

MPI Basics

Hello, World

- Every MPI program has
  - At least one **communicator** to pass mesages among processes
  - A **rank** for each process within a communicator.
- The MPI program is sandwiched between **Init** and **Finalize** calls.
- Compile with MPI headers and libraries
  - For convenience, use the OpenMPI **mpic++** *wrapper compiler*.

Execution

Distributed System Management

- You run the MPI program with **mpirun**.
- This causes your master node to *send copies* of the executable to the other nodes in your distributed system.
- It launches execution on the other nodes.
- The output is routed back to the master node.

Data Transfer

Send and Receive

- Data must be explicitly transferred from process to process.
- `void Send(const void * data, int count, const MPI::Datatype& data_type, int destination_id, int tag);`
- `void Recv(const void * data, int count, const MPI::Datatype& data_type, int source_id, int tag);`
- Note that data are assumed to uniform in type!
  - Scalar or array.
- The receiver must “know” how many objects to expect and whence to expect them.
- The **tag**s must agree.
- This kind of communication is *blocking*.
  - The sender has to wait until its buffer is free for re-use.
  - The receiver has to wait to receive the message from the sender.

Ping Pong

Pairwise Blocking Communication

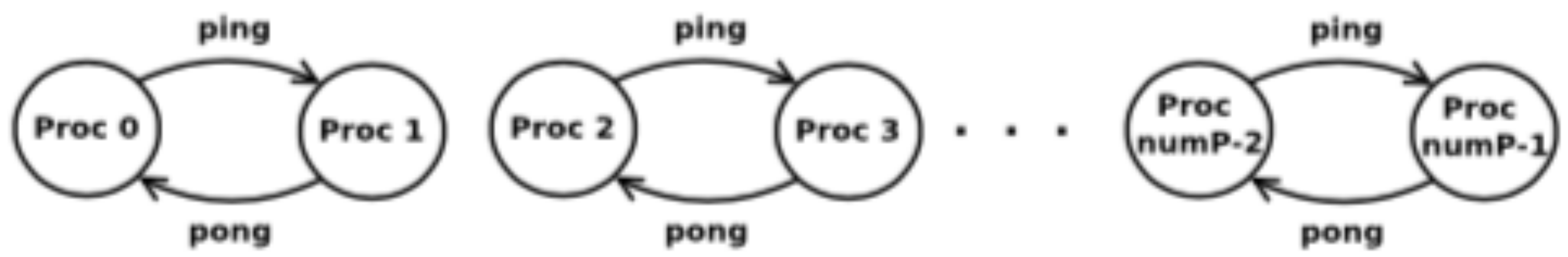

FIGURE 9.3

Pairs of processes involved in the *ping-pong* communication scheme.

What If You Ping in a Ring?

The Problem With Blocking Communication

```
void ping_pong(int num_ping_pongs, int id) {
    int ping_pong_count = 0;
    int next_id = id + 1, prev_id = id - 1;

    if(next_id >= num_p) {
        next_id = 0;
    }
    if(prev_id < 0) {
        prev_id = num_p - 1;
    }

    while(ping_pong_count < num_ping_pongs) {
        ping_pong_count++;

        // Send the ping.
        MPI::COMM_WORLD.Send(&ping_pong_count, 1, MPI::INT, next_id, 0);

        // Wait and receive the ping.
        MPI::COMM_WORLD.Recv(&ping_pong_count, 1, MPI::INT, prev_id, 0);

        // Send the pong
        MPI::COMM_WORLD.Send(&ping_pong_count, 1, MPI::INT, prev_id, 0);

        // Wait and receive the pong
        MPI::COMM_WORLD.Recv(&ping_pong_count, 1, MPI::INT, next_id, 0);
    }
}
```

What If You Ping in a Ring?

The Problem of Blocking Communication

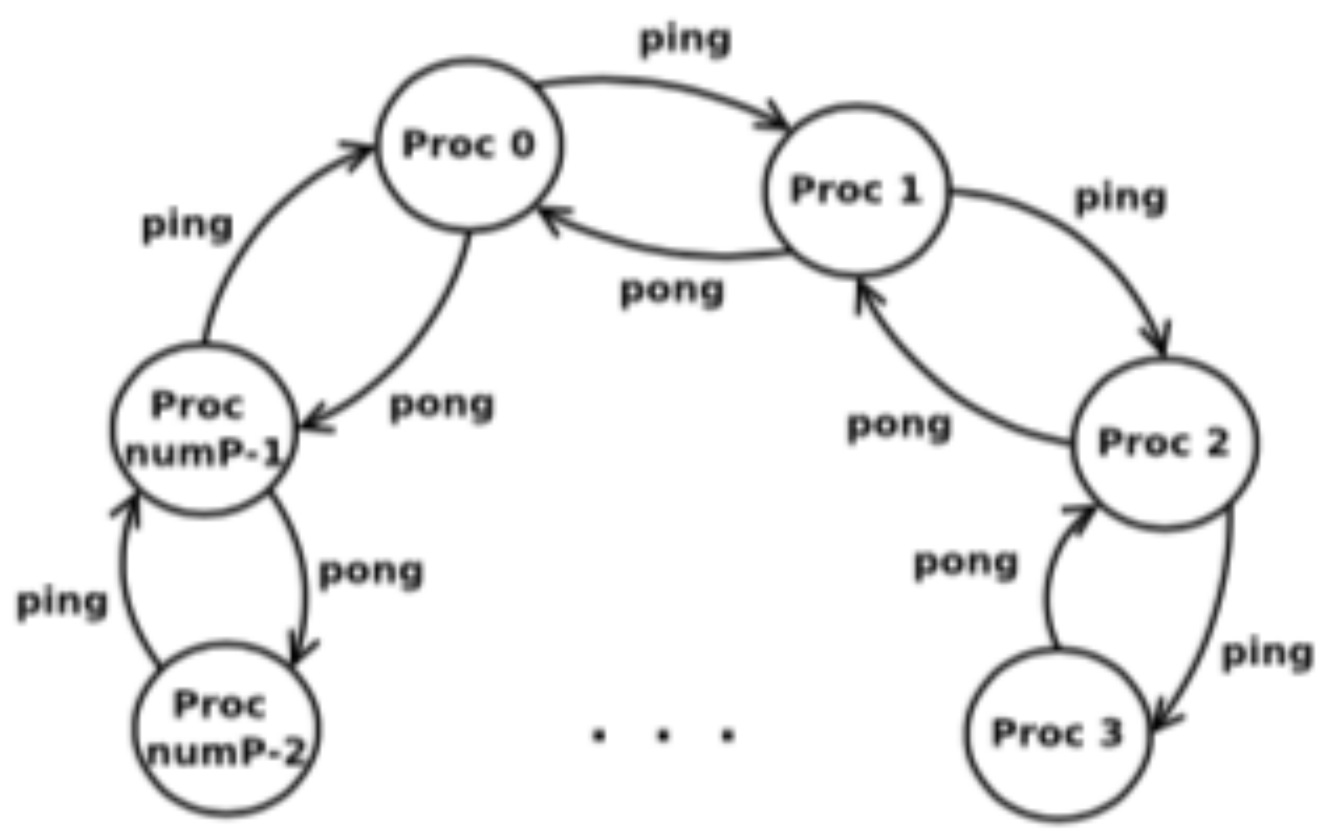

FIGURE 9.4

Abstraction of the *ping-pong* messages on an ordered ring of processes.

Deadlock

The Need for Non-Blocking Communication

- Each thread sends to id + 1 and then receives from id - 1.
- Thread id’s **send** may not return until thread id + 1’s **receive** completes.
- Each thread never gets to receive because it is waiting for **send** to return.
- No **send** can return because each **receive** is waiting.
- This only actually happens when the data load is big enough.

Non-Blocking Communication

Safe Against Deadlock

- MPI::Request Isend(const void * data, int count, const MPI::Datatype& data_type, int destination_id, int tag);
- MPI::Request Irecv(const void * data, int count, const MPI::Datatype& data_type, int source_id, int tag);
- These functions return immediately.
- To synchronize, i.e., wait until data are available, use Wait or Test.

Deadlock-Safe Ping Pong

Using Non-Blocking Communication

```
// …
   MPI::Request rq_receive;

   while (ping_pong_count < num_ping_pongs) {
      ping_pong_count++;

      // Send the ping.
      MPI::COMM_WORLD.Isend(&ping_pong_count, 1, MPI::INT,next_id, 0);

      // Wait and receive the ping.
      rq_receive = MPI::COMM_WORLD.Irecv(&ping_pong_count, 1, MPI::INT, prev_id, 0);

      rq_receive.Wait();

      // Send the pong
      MPI::COMM_WORLD.Isend(&ping_pong_count, 1, MPI::INT, prev_id, 0);

      // Wait and receive the pong
      rq_receive = MPI::COMM_WORLD.Irecv(&ping_pong_count, 1, MPI::INT, next_id, 0);

      rq_receive.Wait();
  }
```