ENCE360 Lab 8: Distributed Processing

Objectives

This lab is an introduction to distributed processing using the Message Passing Interface (MPI).

This lab should:

1. Get you familiar with setting up a distributed system.
2. Introduce you to writing programs and performing calculations using a message-passing model.

Preparation

Download and extract your Lab 8 files from `lab8Distributed.zip` on Learn. This contains: - The scripts `run.sh`, and `run_remote.sh`. You may need to make these executable with `chmod +x`. - The plaintext files `hosts` and `remote_hosts`. - The source files `hello_world.c`, `pass_the_parcel.c`, `ping.c`, `sort.c`, and `vector_len.c`. - A `Makefile`.

Once you've completed the tasks below, make sure you also complete the quiz on the quiz server.

You can compile all the programs by running `make`. **Sections of this lab will be impossible to do unless you are on a lab machine**.

Setting Up a Small Cluster

To run one of the examples on only local processes, use the `run.sh` script, e.g.

```
./run.sh ./hello_world 8
```

will run the program `hello_world` on 8 local processes.

To run a program with remote processes, we have to do some setup first. First, edit the `remote_hosts` file to contain the names of some of the other lab computers around you - these can be worked out from your computer's name in the terminal and the sticker on the top of the machine. The first entry should be your machine - the others are placeholder examples and should be overwritten.

Next, you will need to set up password-free `ssh` access to the computers in your `remote_hosts` file. First, generate some ssh keys by running

```
ssh-keygen -t ed25519
```

Just keep hitting enter without any other input to use the default settings. Then, for each remote machine name in your `remote_hosts` file, perform the following:

```
ssh-copy-id abc123@csxxxxab
```

Replace `abc123` with your username and `csxxxxab` with the remote machine name.

Once you've copied your keys to each remote machine, you finally need to login to each remotely (you can close the connection right afterwards):

```
ssh abc123@csxxxxab
```

For more detail, refer to the guide at https://www.digitalocean.com/community/tutorials/how-to-set-up-ssh-keys-2.

You can test whether this has all worked by running

```
./run_remote.sh ./hello_world 16
```

So long as the output from the `hello_world` program is there, you can ignore any other errors.

Program `hello_world.c`

Each MPI node is a separate process, running identical code but having a unique identifier or "rank". It can be thought of as automatically `fork()`ing many times at the beginning of the program. There is no memory shared between process, and the processes can run across multiple machines.

The functions `MPI_Comm_size()` and `MPI_Comm_rank()` are used to identify the total number of processes and the ID of the current process respectively. As always, refer to the `man` page for details.

Examine `hello_world.c` and run both locally and using remote nodes (as described above). Notice that, even when running nodes on multiple machines, `stdout` and `stderr` are routed back to the terminal we launched from, so we can use functions like `printf()`.

Program `ping.c`

Instead of using system-level primitives like pipes, signals, mutexes, or semaphores, MPI uses message passing. The messages can be passed over a variety of protocols - TCP in the case of our lab, but dedicated interconnects for computing clusters.

Examine the example `ping.c` and run with `./run.sh ./ping 2`. Notice how we select even numbered processes to play ping-pong with odd numbered processes. The odd numbered processes increment the value each time and send it back to the even numbered ones.

Program `pass_the_parcel.c`

Open the file pass_the_parcel.c and implement a communication pattern where instead of playing ping pong, we pass the value in a circle, incrementing each

time. We start at the first node (`world_rank` 0) and pass the value to the next node, then the next, incrementing the value each time, until the final node sends it back to `world_rank` 0. For four processes, the order should be from `world_rank` 0, to 1, to 2, to 3, and back to 0.

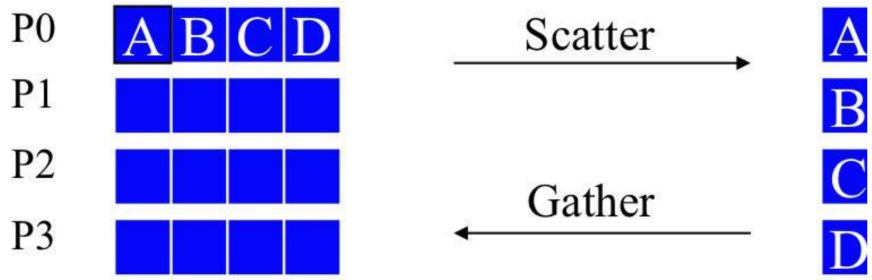
This should give the following output:

```
0: sending value 0 to process 1
1: received value 0 from process 0
1: sending value 1 to process 2
2: received value 1 from process 1
2: sending value 2 to process 3
3: received value 2 from process 2
3: sending value 3 to process 0
0: received value 3 from process 3
```

Program `vector_len.c`

MPI provides several ways to share values between processes. Some of these operations are Scatter, Gather, Broadcast, and Reduce.

`MPI_Scatter()` and `MPI_Gather()` are often used together. `MPI_Scatter()` splits a large array up evenly between nodes. `MPI_Gather()` then can rejoin these scattered smaller arrays spread across multiple nodes into a single large array in a single node.



Now have a look at the example `vector_len.c`, which currently uses `MPI_Scatter()` and `MPI_Reduce()`. `MPI_Reduce()` joins values together with an operator. In this case we use `MPI_SUM` which adds the values held in the nodes.

Run the example locally with varying numbers of processes. How does the node count change the running time?

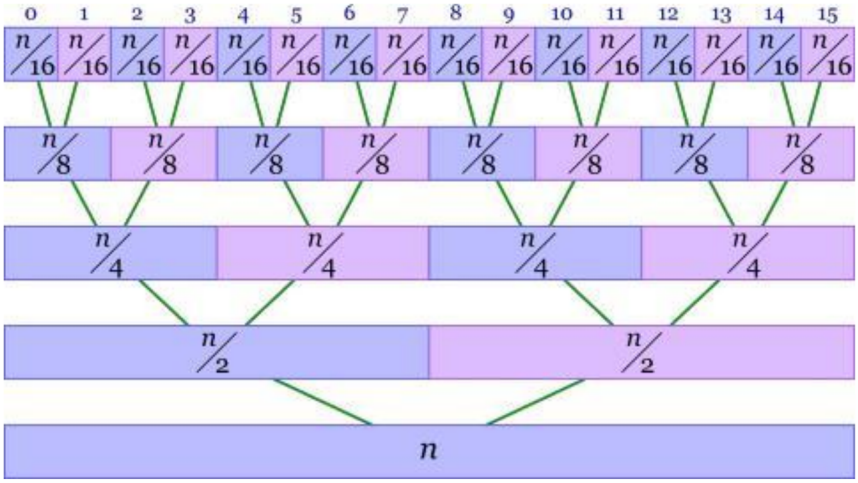
Next, run the example remotely. How does this change the running time? What is the optimal number of process? Why?

Implement the reduction (i.e. the final summation) using `MPI_Gather()` instead of `MPI_Reduce()`. Note that this requires doing the summation manually, and allocating an extra (though small) buffer.

Program `sort.c`

A more advanced parallel task is a parallel merge sort, as seen in `sort.c`. You will notice that some of the communication code has been removed! Implement the communications using `MPI_Scatter()`, `MPI_Send()`, and `MPI_Recv()`. Comments have been left in to give hints.

The sort program first scatters the array to all processes, then each process sorts its own part of the array. These subarrays are then merged recursively via `merge_tree()`, which merges between pairs of processes until finally the whole result is in the master process (`world_rank` 0). The following diagram shows all the merge operations, where for each merge the leftmost node of the pair contains the result. For example, ranks 0 and 1 merge into rank 0 in the first step. In the second step, ranks 0 and 2 merge into rank 0.



The scheme becomes more complicated when there isn't a power-of-2 number of nodes. In this case, the processes which don't have a partner simply pass their smaller array up the tree without merging anything.

When sending arrays between processes in the `merge_tree()` function, there's an unknown number of elements. You should overcome this by first sending a message containing the number of elements, and only *then* sending the array itself.

This time we'll need an extra argument to our run command, saying how many elements we're going to sort. For example,

```
./run.sh ./sort 4 10000000
```

The program contains a correctness test; when the implementation is correct, the output should look something like this:

```
time taken: 1.2331 result_size: 10000000, test: pass
```

Again, experiment with the number of processes. What is the optimal number? Does the code work well using remote processes? Why or why not?