Lab 7: Distributed processing with MPI

Objectives

This lab is an introduction to distributed processing using the MPI programming model.

Source Code

The files on learn are:

src/

- hello_world.c
- ping.c
- pass_the_parcel.c
- sort.c
- vector_len.c

Makefile	// To compile the programs use “make”
run.sh	// For running programs on several local processes
run_remote.sh	// For running programs on remote hosts
hosts	// Contains “localhost”
hosts_remote	// Contains a list of remote hosts to use for run_remote.sh

Setting up

All the programs can be compiled by typing “make” in the base directory, and run by using **run.sh** for local processes “./run.sh ./hello_world 4” for example, which runs the hello_world example on 4 local processes.

To run one of the examples remotely use “./run_remote.sh ./hello_world 4”, before doing this edit the hosts_remote file and change it to use some of the unoccupied lab computers around you (or using computers used by others in the lab). If any of the computers are not turned on MPI will give an error.

Note:

OpenMPI uses ssh to run processes on other computers. To be able to utilize other lab computers, you will need to setup password-free ssh access, a step by step guide on this is available here: https://www.digitalocean.com/community/tutorials/how-to-set-up-ssh-keys--2

In addition, you can prevent ssh from attempting to ask you about authorized hosts you can add an entry like this into your **~/.ssh/config** file:

```
Host cs*

  StrictHostKeyChecking=no
```

Introduction

MPI (Message Passing Interface) uses separate processes, conceptually similar to calling “fork()” many times at the very beginning of a program, so each process is identical but differs only in that has a different “rank” (just as fork() returns a different value for the parent and child). Just like using fork – there's no shared memory between processes, MPI programs run in separate processes which may and commonly do exist on different computers.

MPI_Comm_size, and **MPI_Comm_rank** are used to identify the total number of processes, and the process ID of the current process. View the **man** pages for details.

Examine **hello_world.c** and run it with “./run.sh ./hello_world 4”, you will notice that even if you run the example on multiple computers, stdout and stderr are piped back to the console from which it was launched, so you may use normal stdio routines such as printf().

Concurrency with processes

Instead of using system level programming primitives such as pipes, signals, mutexes or semaphores MPI uses message passing. Message passing occurs over any one of several protocols such as TCP or fast supercomputer interconnects.

The most basic way of communicating is with **MPI_Recv**, and **MPI_Send**

Examine the example **ping.c**, and run with “./run.sh ./ping 2” and 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.

Task 1

Open the file **pass_the_parcel.c** and implement a communication pattern where instead of playing ping pong we instead start at the first (world_rank 0) process and pass the value in a circle around the proccesses, incrementing the value each time, then finally send it back to the first process to finish (and print out the output at each step of the way). Like this (for 4 processes):

world_rank: 0 → 1 → 2 → 3 → 0

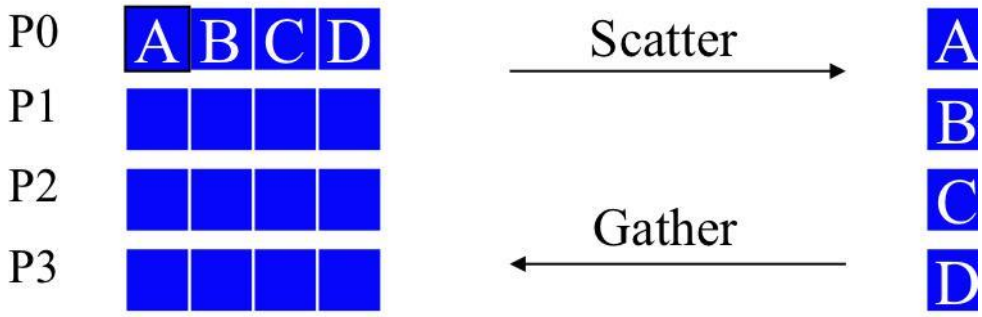
Giving output looking like:

```
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
```

Scatter and Gather

MPI provides several ways to share values between processes. Such as scatter, gather, broadcast, and reductions.

One such way is through the use of **MPI_Scatter** and **MPI_Gather** which splits a large array up evenly between all the processes, and join smaller arrays from all the processes to one process respectively.



Task 2

Take 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 has the result of adding the values e.g. A+ B+ C + D

Run the example with "./run.sh ./vector_len 1", how does the number of processes change the running time?

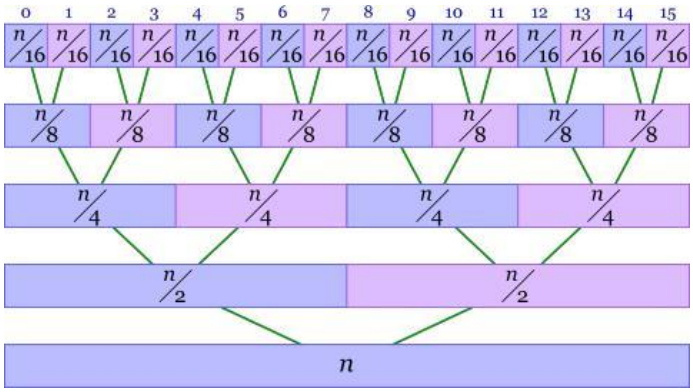
Run the example remotely with "./run_remote.sh ./vector_len 16", how does this change the runing time? What is the optimal number of processes? Why do you think this is?

Implement the reduction (final summation) using **MPI_Gather** instead of **MPI_Reduce**, note that this requires doing the summation manually and allocating an extra (small) buffer.

Task 3

A more advanced example is a parallel merge sort which can be found in **sort.c** you will notice that some of the communication code has been removed! Implement the communication code (using: MPI_Scatter, MPI_Send and MPI_Recv). Comments have been left in to give hints.

The sort program given here first scatters the array to all the processes, then each process sorts it's own part of the array, followed by a recursive merge (merge_tree) which merges between pairs of processes recursively until finally the whole result is in the master process (world_rank 0). The following diagram shows the merge process, with the process indicated at the leftmost edge containing the merged result, (e.g. at the first level (0 , 1) merges to 0 and on the second level (8, 10) merges to 8)



The only slightly more complicated part is when there's not a power-of-2 number of processes, in which case the processes which don't have a partner simply passes their smaller array up the tree without merging anything.

When sending arrays between processes in the merge_tree method, there's an unknown number of elements – so one way to send/recieve an unknown number of elements is to first send through the size, then follow it up with the actual data.

As before – run the program with "./run.sh ./sort 1 1000000", the extra argument is the number of elements to sort. You will notice the code contains a test, when the implementation is correct the output will print something like this:

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

Experiment with the number of processes, what is the optimal number? Does this code work well using remote processes?