Measuring point-to-point message latency with ping-pong

Dr. Christopher Marcotte — *Durham University*

In this exercise we will write a simple code that does a message ping-pong: sending a message back and forth between two processes.

We can use this to measure both the *latency* and *bandwidth* of the network on our supercomputer. Which are both important measurements when we're looking at potential parallel performance. Understanding the latency and bandwidth of a machine will help us to decide if our code is running slowly because of our bad choices, or limitations in the hardware.

A Model

We care about the total time to send a message, so our model is a linear model which has two free parameters:

1. α, the message latency, measured in seconds;
2. β, the inverse network bandwidth, measured in seconds/byte (so that the bandwidth is β^{-1}).

With this model, the time to send a message of size b bytes is

$$T(b) = \alpha + \beta \cdot b \tag{1}$$

Implementation

I provide a template in `ping-pong.c` that you can compile with `mpicc` on Hamilton, or any machine with a properly configured MPI. The compiled executable takes one command-line argument, the size of the message (in bytes) to exchange.

ping-pong.c

```c
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>

static void ping_pong(void *buffer, int count, MPI_Datatype dtype, MPI_Comm comm){
  /* Implement a ping pong.
  *
  * rank 0 should send count bytes from buffer to rank 1
  * rank 1 should then send the received data back to rank 0
  *
  */
}

int main(int argc, char **argv){
  MPI_Init(&argc, &argv);

  int nbytes, rank;
  char *buffer;
  double start, end;
  MPI_Comm comm;

  comm = MPI_COMM_WORLD;
  nbytes = argc > 1 ? atoi(argv[1]) : 1;

  buffer = calloc(nbytes, sizeof(*buffer));

  ping_pong(buffer, nbytes, MPI_CHAR, comm);

  free(buffer);

  MPI_Finalize();
  return 0;
```

```
    }
```

Exercise

Implement the `void ping_pong(...)` function which sends a message of the given size of b bytes from rank 0 to rank 1, after which rank 1 should send the same message back to rank 0. Ensure that the code also works with more than two processes (all other ranks should just do nothing). See the diagram below for an illustration.

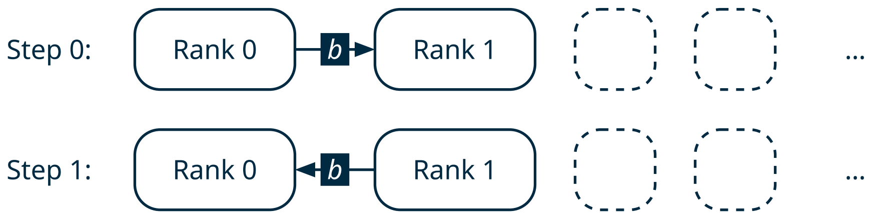

Figure 1: A diagram of the ping pong program; processes taking part in the message exhange are represented as ▢ while those not taking part are represented by ⬚ and the messages are represented by $\longrightarrow$.

Hint

This code should just use `MPI_Send(...)` and `MPI_Recv(...)`. Note that if you want a 1 byte resolution on the inputs, this constrains the datatypes you can use.

Exercise

Add timing around the `ping_pong(...)` call to determine how long the program takes to send a message of length `int count` back and forth. What happens if the different ranks disagree on the timing?

Hint

Use `MPI_Wtime()` to record the start and end wall-times, and calculate their difference to record the ping-pong time. For small messages you will probably need to do many iterations in a loop to get accurate timings.

How you select the number of iterations is a bit of an art – you want enough that the answer has little noise effects, but you also want all the ranks to agree on how many repetitions that is. You can do a warm-up where the number of repetitions is fixed, record the time, and extrapolate so that the total time is fixed and long enough to record sensible statistics.

Experiment

In the following, you will investigate the timing of the code you've written. I recommend you do so on Hamilton, because we will consider first the *intra*-node timing, and later the *inter*-node timing.

Intranode timing

Exercise

Run your code on the Hamilton compute nodes for a range of messages sizes from 1 byte to 64 megabytes. Produce a plot of the time to send a message as a function of message size. Using `numpy.polyfit(...)` (or `numpy.polynomial(...)` or your favourite linear regression scheme[1]), fit the proposed model Equation 1 to your data. What values of α and β do you get? Are these reasonable?

[1]It is possible, and not especially difficult, to write your own linear regression in straight *C* – see the *Challenge*.

Challenge

Write your own *simple* linear regression in *C*. This code should read the output of your timing code from a file, calculate α and β from Equation 1, and print the approximate values.

Exercise

Perform the same experiment, but this time, place the two processes on *different* Hamilton compute nodes. Do you observe a difference in the performance?

Hint

To do this, you'll need to write a SLURM batch script that specifies `--nodes=2` and `--ntasks-per-node=1` (whose product should be 2 in your script, so that you aren't underusing resources).

Variability

One thing that can affect performance of real MPI codes is the message latency, and particularly if there is any variability. This might be affected by other processes that happen to be using the network, or our own code, or operating system level variability. We'll see if we can observe any on Hamilton.

Exercise

Modify your code so that rather than just timing many ping-pong iterations, it records the time for each of the many iterations separately. Use this information to compute the mean ping-pong time, along with the standard deviation, and the minimum and maximum times. Produce a plot of these data, using the standard deviation as error bars and additionally showing the minimum and maximum times as outliers. What, if any, variability do you observe? Does it change if you move from a single compute node to two nodes?

Hint

You can allocate an array for your timing data with `double *timing = malloc(nreps * sizeof(*double));`, but you must remember to call `free(timing);` to release the memory.

Challenge

The simplest implementation of the above calculates the statistics locally on each rank, and does not perform a reduction to get the correct values across all ranks. Adapt your statistics calculation so that rank 0 now returns the mean of the timing, the standard deviation of the timing, and the minimum and maximum times *across all ranks and all repetitions*.

Network contention

Finally, we'll look at whether having more messages "in flight" at once effects performance. Rather than running with two processes, you should run with full compute nodes (128 processes per node, or using `#SBATCH --exclusive`).

Exercise

Modify your ping-pong code so that all ranks participate in pairwise messaging.

Divide the processes into a "bottom" and "top" half. Suppose we are using `size` processes in total. Processes with `rank < size/2` are in the "bottom" half, the remainder are in the "top" half.

A process in the bottom half should send a message to its matching process in the top half (`rank + size/2`), that process should then return the message (to `rank - size/2`).

Again measure the time and variability, and produce a plot. Do the results change from previously? When using one compute node? When using two? When using four?

Aims

- Introduction to MPI communication concerns and concepts
- Practice with benchmarking communication overheads
- Introduction to `MPI_Send(...)` and `MPI_Recv(...)`
- Performance modeling practice