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.

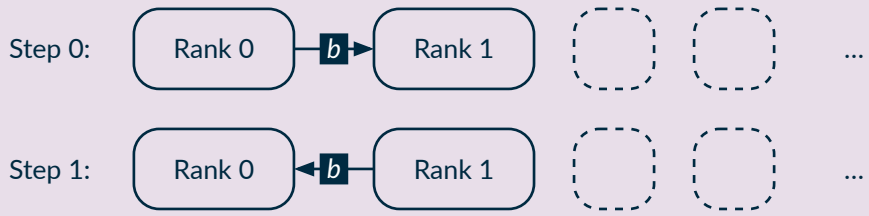

Figure 1: A diagram of the ping pong program; processes taking part in the message exhange are represented as ▢ while those not taking part are represented by ⬚ and the messages are represented by →.

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.

Solution

The implementation of this function is exceptionally simple – you just send/recv from rank 0 or recv/send from rank 1.

ping-pong-solution.c

```c
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 rank, size;
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);

  if (rank == 0) {
    MPI_Send(buffer, count, dtype, 1, 0, comm);
    MPI_Recv(buffer, count, dtype, 1, 0, comm, MPI_STATUS_IGNORE);
  } else if (rank == 1) {
    MPI_Recv(buffer, count, dtype, 0, 0, comm, MPI_STATUS_IGNORE);
    MPI_Send(buffer, count, dtype, 0, 0, comm);
  } else {
    /* Nothing to do */
  }
}

```

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.

Solution

The only subtlety here is the selection of a 'representative' time from amongst all the processes; one can do this any number of ways, but some are more honest than others. If you were a salesperson for a tech startup in the middle of a funding round, you might report the minimum time across all processes. Since all but two processes do not *do anything*, this will be effectively the resolution of `MPI_Wait()`, and not representative of the actual message time. If you were a (bad) statistician, you might compute the mean time, which suffers from similar issues. As an unimpeachably honest computer scientist, you know that the maximum time is representative of the actual messaging time, and thus call `MPI_Allreduce(...)` with the `MPI_MAX` operator.

ping-pong-solution.c

```c
  start = MPI_Wtime();
  for (int i = 0; i < 100; i++) {
    ping_pong(buffer, nbytes, MPI_CHAR, comm);
  }
  end = MPI_Wtime();

  /* Figure out how many repetitions to do so that we measure about 3 seconds of time */
  nreps = (int)(300 / (end - start));
  if (nreps <= 0) {
    nreps = 1;
  }else{
    if (nreps > 3000000){
      nreps = 3000000;
    }
  }
  // We might not have agreed on nreps, so pick the maximum with MPI_Allreduce
  MPI_Allreduce(MPI_IN_PLACE, &nreps, 1, MPI_INT, MPI_MAX, comm);
```

The actual timing of the ping-pong is very similar:

ping-pong-solution.c

```c
  start = MPI_Wtime();
```

```
61   for (int i = 0; i < nreps; i++) {
62     ping_pong(buffer, nbytes, MPI_CHAR, comm);
63   }
64   end = MPI_Wtime();
65   double tspan = (end - start);
66   /* We might not have agreed on the tspan, so pick the maximum with MPI_Allreduce */
67   MPI_Allreduce(MPI_IN_PLACE, &tspan, 1, MPI_DOUBLE, MPI_MAX, comm);
68   if (rank == 0) {
69     // Time for one message is half a pingpong
70     printf("%d %d %g\n", nbytes, nreps, tspan/(nreps*2));
71   }
```

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*), fit the proposed model Equation 1 to your data. What values of α and β do you get? Are these reasonable?

Solution

When I run it on one node using messages ranging from 1 Byte to 16MB (in powers of 2), I get the timing results in Figure 2.

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

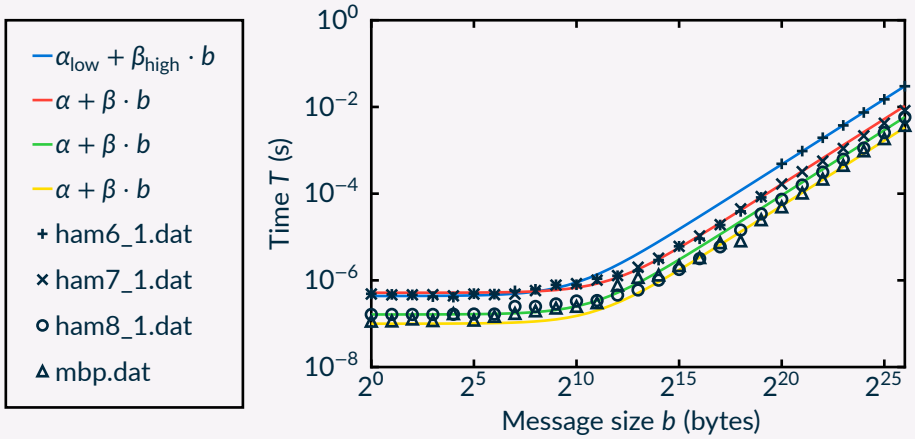

Figure 2: Intra-node ping-pong time on Hamilton 6, 7, & 8 and my MacBook Pro. Note that $\alpha_6 \approx 436$ns, $\alpha_7 \approx 163$ns, and $\alpha_8 \approx 100$ns while $\beta_6^{-1} \approx 2$GiB/s, $\beta_7^{-1} \approx 12$GiB/s, $\beta_8^{-1} \approx 20$GiB/s. Notably, my laptop achieves similar recordings to Hamilton 8, and is perhaps slightly faster due to unusually high-bandwidth on the CPU.

It looks like a piecewise linear model might be more fitting for the MPI implementation on Hamilton 6. Between 512 KB and 1 MB, the time for Hamilton 6 jumps up significantly. This is probably when the MPI implementation is switching protocols from `MPI_Bsend(...)`-like to `MPI_Ssend(...)`-like.

The fit values seem imminently reasonable – Hamilton 8 is quite a jump up from Hamilton 7 and Hamilton 6 – and a yet the story is quite strange for my laptop.

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).

Solution

If I do this, I see that the inter-node latency on Hamilton 6 is pretty bad, although asymptotically it seems like the bandwidth is the same as for inter-node. The slow message at 32 MB appears to be repeatable, but it's unclear what the cause is – since Hamilton 6 is long-decommissioned, we may never know.

Notice that when going across nodes, the switch in protocol happens at a lower size (it looks like 256 KB, rather than 1 MB).

Figure 3 has results for Hamilton 7 which performs somewhat better, and Hamilton 8 which is much more consistent.

If I fit our linear model to the inter-node Hamilton 6 data, I get $\alpha_6^{\text{inter}} \approx 6\mu\text{s} \approx 14\alpha_6^{\text{intra}}$. For the inverse bandwidth $\beta_6^{\text{inter}} \approx 2\text{s/GiB} \approx 4\beta_6^{\text{intra}}$, resulting in a network bandwidth of around $\beta_6^{-1} \approx 1\text{GiB/s}$.

Fitting the model to the Hamilton 7 data, the intra-node latency is $\alpha_7^{\text{intra}} \approx 1\mu\text{s} \ll 5\mu\text{s} \approx \alpha_7^{\text{inter}}$, but the asymptotic intra-node bandwidth is $\beta_7^{-1} \approx 6\text{GiB/s}$, compared to the inter-node bandwidth of $\beta_7^{-1} \approx 3\text{GiB/s}$, so the network is much better inside a node.

The Hamilton 8 results really blow Hamiltons 6 & 7 out of the water – for $b \gtrsim 2^{16}$ the simple model clearly needs fine-tuning and Hamilton 8 is about as fast as Hamilton 7 with a 1000 × larger message, or 10 × faster with the same size message. What is unclear is why the two bandwidths on Hamilton 8 comes out so similar – perhaps I am not effectively saturating the intra-node bandwidth with separate calls to `MPI_Send(...)` and `MPI_Recv(...)` – if you find different values, let me know so I can update this plot.

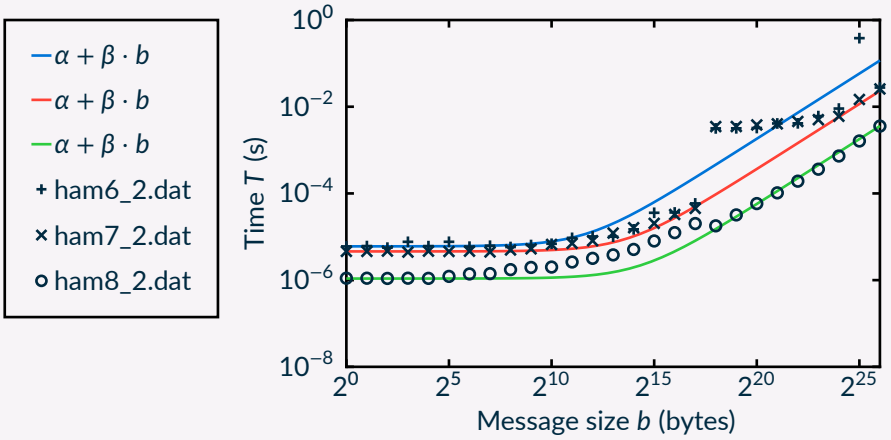

Figure 3: Inter-node ping-pong time on Hamilton 6, 7, & 8. Note that $\alpha_6 \approx 6\mu\text{s}$, $\alpha_7 \approx 5\mu\text{s}$, *and* $\alpha_8 \approx 1\mu\text{s}$ *while* $\beta_6^{-1} \approx 1\text{GiB/s}$, $\beta_7^{-1} \approx 3\text{GiB/s}$, $\beta_8^{-1} \approx 19\text{GiB/s}$.

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?

Solution

This is a fairly simple modification from the previous solution code. The substantive differences appear in the `int main(...)` function:

ping-pong-statistics.c

```c
  for (int i = 0; i < nreps; i++) {
     start = MPI_Wtime();
     ping_pong(buffer, nbytes, MPI_CHAR, comm);
     end = MPI_Wtime();
     timing[i] = end - start;
  }
  // Compute statistics
  double stats[4] = {0.0, 0.0, 0.0, 100.0};
  for (int i = 0; i < nreps; i++){
    stats[0] += timing[i];
  }
  stats[0] = stats[0] / nreps;
  for (int i = 0; i < nreps; i++){
    stats[1] += (timing[i] - stats[0]) * (timing[i] - stats[0]);
  }
  stats[1] += stats[1] / nreps;
  for (int i = 0; i < nreps; i++){
    stats[2] = fmax(stats[2], timing[i]);
    stats[3] = fmin(stats[3], timing[i]);
  }
  if (rank == 0){
    printf("mean = %e, stdv = %e, max = %e, min = %e\n", stats[0], stats[1], stats[2],
stats[3]);
  }
```

I will resist making a plot of the results, instead we can discuss something kind of interesting and – nowadays – relevant: heterogeneous systems.

You will find that there is relatively little variability on Hamilton – this is desirable on working clusters. My laptop has an Apple Silicon M1 Pro CPU – this is a heterogeneous design with a $4P + 4P + 2E$ clustering; thus, we might expect that for $4 \to 5$ and $8 \to 9$ processes, we can see a marked change in the variability.

Indeed, the output of the call `mpirun -np 10 ppstats 10000 > ppstats.dat` is:

ppstats.dat

```
mean = 3.916000e-08, stdv = 2.419003e-08, max = 5.400000e-05, min = 0.000000e+00
mean = 3.955667e-08, stdv = 1.100599e-07, max = 1.690000e-04, min = 0.000000e+00
mean = 4.578667e-08, stdv = 4.257135e-06, max = 2.058000e-03, min = 0.000000e+00
mean = 5.157000e-08, stdv = 1.167717e-06, max = 6.080000e-04, min = 0.000000e+00
mean = 4.762000e-08, stdv = 4.047710e-07, max = 5.570000e-04, min = 0.000000e+00
mean = 4.206000e-08, stdv = 3.032343e-07, max = 4.550000e-04, min = 0.000000e+00
mean = 5.357667e-08, stdv = 4.253713e-07, max = 3.910000e-04, min = 0.000000e+00
mean = 5.259333e-08, stdv = 5.700381e-07, max = 5.220000e-04, min = 0.000000e+00
mean = 3.171157e-06, stdv = 5.681785e-05, max = 6.899000e-03, min = 1.000000e-06
mean = 3.170197e-06, stdv = 5.723502e-05, max = 6.901000e-03, min = 1.000000e-06
```

Where we can see mean time jumps in the expected places, and similar jumps in the standard deviation and maximum times. The minimum time, perhaps expectedly, is relatively uniform save for the inclusion of the 2*E* core cluster.

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?

Solution

The difference from earlier codes is very subtle, appearing only in the `MPI_Send(...)` and `MPI_Recv(...)` calls and the surrounding conditionals.

ping-pong-pairwise.c

```c
17   if (rank < size / 2) {
18     MPI_Send(buffer, count, dtype, rank + size/2, 0, comm);
19     MPI_Recv(buffer, count, dtype, rank + size/2, 0, comm, MPI_STATUS_IGNORE);
20   } else {
21     MPI_Recv(buffer, count, dtype, rank - size/2, 0, comm, MPI_STATUS_IGNORE);
22     MPI_Send(buffer, count, dtype, rank - size/2, 0, comm);
23   }
```

Note that I do not do any input verification in this code – one would in a research code – so if you supply an odd number of processes, the whole thing will deadlock.

The actually interesting thing about this exercise is the expectation; we expect that more buffers yields more pressure on MPI, and things slow down. In practice this effect is quite small, *until it suddenly isn't*.

Aims

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