|epcc|

Exercises: Message-Passing Programming

David Henty

1 Hello World

1. Write an MPI program which prints the message "Hello World".
2. Compile and run on several processes in parallel, using the both the frontend login and backend compute nodes of Cirrus (you will need to use `sbatch` to run on the compute nodes).
3. Modify your program so that each process prints out both its rank and the total number of processes P that the code is running on, i.e. the size of `MPI_COMM_WORLD`.
4. Modify your program so that only a single controller process (e.g. rank 0) prints out a message – this is very useful when you run with hundreds of processes.
5. Increase the number of MPI processes by altering the value of `ntasks` in the SLURM script.
6. What happens if you omit the final MPI procedure call in your program?

1.1 Extra Exercises

Since both Cirrus and ARCHER2 are clusters of shared-memory nodes (36 and 128 cores per node respectively), it can be interesting to know if two MPI processes are on the same or different nodes. This is not specified by MPI – it is a function of the job launcher program which is called `srun` for SLURM.

1. Use the function `MPI_Get_processor_name()` so each rank prints out where it is running.
2. Run on multiple nodes by increasing the value of `nodes` in the SLURM script. What is the default policy for allocating processes to nodes when there are fewer processes than physical CPU cores?
3. What happens if you try and run more MPI processes than physical CPU-cores?

2 Parallel calculation of π

An approximation to the value π can be obtained from the following expression

$$\frac{\pi}{4} = \int_0^1 \frac{dx}{1+x^2} \approx \frac{1}{N} \sum_{i=1}^{N} \frac{1}{1+\left(\frac{i-\frac{1}{2}}{N}\right)^2}$$

where the answer becomes more accurate with increasing N. Iterations over i are independent so the calculation can be parallelised.

For the following exercises you should set $N = 840$. This number is divisible by 2, 3, 4, 5, 6, 7 and 8 which is convenient when you parallelise the calculation!

1. Modify your Hello World program so that each process independently computes the value of π and prints it to the screen. Check that the values are correct (each process should print the same value).

2. Now arrange for different processes to do the computation for different ranges of i. For example, on two processes: rank 0 would do $i = 1, 2, \ldots, \frac{N}{2}$; rank 1 would do $i = \frac{N}{2} + 1, \frac{N}{2} + 2, \ldots, N$. Print the partial sums to the screen and check the values are correct by adding them up by hand.

3. Now we want to accumulate these partial sums by sending them to the controller (e.g. rank 0) to add up:

   - all processes (except the controller) send their partial sum to the controller
   - the controller receives the values from all the other processes, adding them to its own partial sum

   You should use the MPI routines `MPI_Ssend` and `MPI_Recv`.

4. Use the function `MPI_Wtime` (see below) to record the time it takes to perform the calculation. For a given value of N, does the time decrease as you increase the number of processes? Note that to ensure that the calculation takes a sensible amount of time (e.g. more than a second) you will probably have to perform the calculation of π several thousands of times.

5. Ensure your program works correctly if N is not an exact multiple of the number of processes P.

2.1 Timing MPI Programs

The `MPI_Wtime()` routine returns a double-precision floating-point number which represents elapsed wall-clock time in seconds. The timer has no defined starting-point, so in order to time a piece of code, two calls are needed and the difference should be taken between them.

There are a number of important considerations when timing a parallel program:

1. Due to system variability, it is not possible to accurately time any program that runs for a very short time. A rule of thumb is that you cannot trust any measurement much less than one second.

2. To ensure a reasonable runtime, you will probably have to repeat the calculation many times within a do/for loop. Make sure that you remove any print statements from within the loop, otherwise there will be far too much output and you will simply be measuring the time taken to print to screen.

3. Due to the SPMD nature of MPI, each process will report a different time as they are all running independently. A simple way to avoid confusion is to synchronise all processes when timing, e.g.

   ```
   MPI_Barrier(MPI_COMM_WORLD);  // Line up at the start line
   tstart = MPI_Wtime();         // Fire the gun and start the clock
   ...                           // Code to be timed in here ...
   MPI_Barrier(MPI_COMM_WORLD);  // Wait for everyone to finish
   tstop  = MPI_WTime();         // Stop the clock
   ```

   Note that the barrier is only needed to get consistent timings – it should not affect code correctness.

   With synchronisation in place, all processes will record roughly the same time (the time of the slowest process) and you only need to print it out on a single process (e.g. rank 0).

4. To get meaningful timings for more than a few processes you must run on the backend nodes using `sbatch`. If you run interactively on Cirrus then you will share CPU-cores with other users, and may have more MPI processes than physical cores, so will not observe reliable speedups.

2.2 Extra Exercises

1. Write two versions of the code to sum the partial values: one where the controller does explicit receives from each of the $P - 1$ other processes in turn, the other where it issues $P - 1$ receives each from any source (using wildcarding).

2. Print out the final value of π to its full precision (e.g.. 10 decimal places for single precision, or 20 for double). Do your two versions give *exactly* the same result as each other? Does each version give *exactly* the same value every time you run it?

3. To fix any problems for the wildcard version, you can receive the values from all the processors first, then add them up in a specific order afterwards. The controller should declare a small array and place the result from process i in position i in the array (or $i + 1$ for Fortran!). Once all the slots are filled, the final value can be calculated. Does this fix the problem?

4. You have to repeat the entire calculation many times if you want to time the code. When you do this, print out the value of π after the final repetition. Do both versions get a reasonable answer? Can you spot what the problem might be for the wildcard version? Can you think of a way to fix this using tags?

3 Broadcast and Scatter

You should write a simple program using send and receive calls to implement *broadcast* and *scatter* operations. We will see later in the course how to do this using MPI's built-in collectives, but it is useful to implement them by hand as they illustrate how to communicate all or part of an array of data using point-to-point operations rather than just a single variable (as illustrated by the pi example).

Write an MPI program to do the following, using `MPI_Ssend` for all send operations.

1. declare an integer array x of size N (e.g. $N = 12$)
2. initialise the array
   - on rank 0: $x_i = i$
   - on other ranks: $x_i = -1$
3. print out the initial values of the arrays on all processes
4. implement the broadcast:
   - on rank 0: send the entire array x to all other processes
   - on other ranks: receive the entire array from rank 0 into x
5. print out the final values of the arrays on all processes

Now extend your program to implement scatter:

1. declare and initialise the array x as before
2. for P processes, imagine that the array x on rank 0 is divided into P sections, each of size N/P
3. implement the scatter:
   - on rank 0: send the i^{th} section of x to rank i
   - other ranks: receive an array of size N/P from rank 0 into x
4. print out the final values of the arrays on all processes

To ensure that output from separate processes is not interleaved, you can use this piece of code for printing the arrays. To print 10 elements from an array x: `printarray(rank, x, 10);`

```
void printarray(int rank, int *array, int n)
{
  int i;
  printf("On rank %d, array[] = [", rank);

  for (i=0; i < n; i++)
    {
      if (i != 0) printf(",");
      printf(" %d", array[i]);
    }
  printf(" ]\n");
}
```

4 Optional exercise: Ping Pong

1. Write a program in which two processes (say rank 0 and rank 1) repeatedly pass a message back and forth. Use the synchronous mode `MPI_Ssend` to send the data. You should write your program so that it operates correctly even when run on more than two processes, i.e. processes with rank greater than one should simply do nothing. For simplicity, use a message that is an array of integers. Remember that this is like a game of table-tennis:
   - rank 0 should send a message to rank 1
   - rank 1 should receive this message then send *the same data* back to rank 0
   - rank 0 should receive the message from rank 1 and then return it
   - etc. etc.
2. Insert timing calls to measure the time taken by all the communications. You will need to time many ping-pong iterations to get a reasonable elapsed time, especially for small message lengths.
3. Investigate how the time taken varies with the size of the message. You should fill in your results in Table 1. What is the asymptotic bandwidth for large messages?
4. Plot a graph of time against message size to determine the latency (i.e. the time taken for a message of zero length); plot a graph of the bandwidth to see how this varies with message size.

The bandwidth and latency are key characteristics of any parallel machine, so it is always instructive to run this ping-pong code on any new computers you may get access to.

Size (bytes)	# Iterations	Total time (secs)	Time per message	Bandwidth (MB/s)

Table 1: Ping-Pong Results for Exercise 4

4.1 Extra exercises

1. How do the ping-pong bandwidth and latency figures vary when you use buffered or standard modes (`MPI_Bsend` and `MPI_Send`)? (note that to send large messages with buffered sends you will have to supply MPI with additional buffer space using `MPI_Buffer_attach`).
2. Write a program in which the controller process sends the same message to all the other processes in `MPI_COMM_WORLD` and then receives the message back from all of them. How does the time taken vary with the size of the messages and with the number of processes?