|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?

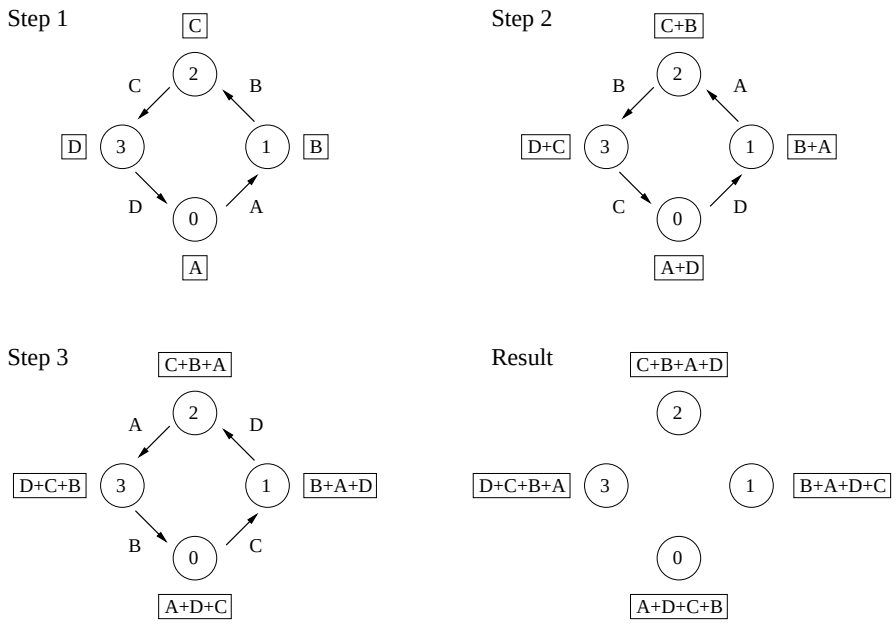

Figure 1: Global sum of four variables

5 Rotating information around a ring

Consider a set of processes arranged in a ring as shown in Figure 1. A simple way to perform a global sum of data stored on each process (a parallel reduction operation) is to rotate each piece of data all the way round the ring. At each iteration, a process receives some data from the left, adds the value to its running total, then passes the data it has just received on to the right.

Figure 1 illustrates how this works for four processes (ranks 0, 1, 2 and 3) who hold values A, B, C and D respectively. The running total on each process is shown in the square box, and the data being sent between processes is shown next to the arrow. After three steps ($P-1$ steps in general for P processes) each process has computed the global sum $A+B+C+D$.

1. Write a program to perform a global sum using this simple ring method. Each process needs to know the ranks of its two neighbours in the ring, which stay constant throughout the program. You should use synchronous sends and avoid deadlock by using non-blocking forms for *either* the send (`MPI_Issend`) *or* the receive (`MPI_Irecv`). Remember that you cannot assume that a non-blocking operation has completed until you have issued an explicit wait. You can use non-blocking calls for send *and* receive, but you will have to store two separate requests and wait on them both. We need to initialise the local variables on each process with some process-dependent value. For simplicity, we will just use the value of $rank$, i.e. in Figure 1 this would mean $A=0$, $B=1$, $C=2$ and $D=3$. You should check that every process computes the sum correctly (e.g. print the final value to the screen), which in this case is $P(P-1)/2$.

2. Your program should compute the correct global sum for any set of input values. If you initialise the local values to $(rank+1)^2$, do you get the correct result $P(P+1)(2P+1)/6$?

5.1 Extra exercises

1. Measure the time taken for a global sum and investigate how it varies with increasing P. Plot a graph of time against P — does the ring method scale as you would expect?

2. Using these timings, estimate how long it takes to send each individual message between processes. How does this result compare with the latency figures from the ping-pong exercise?

3. The `MPI_Sendrecv` call is designed to avoid deadlock by combining the separate send and receive operations into a single routine. Write a new version of the global sum using this routine and compare the time taken with the previous implementation. Which is faster?

4. Investigate the time taken when you use standard and buffered sends rather than synchronous mode (using `MPI_Bsend` you do not even need to use the non-blocking form as it is guaranteed to be asynchronous). Which is the fastest? By comparing to the time taken by the combined send and receive operation, can you guess how `MPI_Sendrecv` is actually being implemented?

6 Collective communications

1. Re-write the ring example using an MPI reduction operation to perform the global sum.

2. How does execution time vary with P and how does this compare to your own ring method?

3. Implement the broadcast and scatter from Section 3 using collective calls. Note there is a slight difference in the functionality of the collective scatter in that the root process sends data to itself.

6.1 Extra exercises

1. Compare the performance of a single reduction of an array of N values compared with N separate calls to reductions of a single value. Can you explain the results (the latency and bandwidth values from the pingpong code are useful to know)?

2. You can ensure all processes get the answer of a reduction by doing `MPI_Reduce` followed by `MPI_Bcast`, or using `MPI_Allreduce`. Compare the performance of these two methods.

3. Imagine you want all the processes to write their output, in order, to a single file. The important point is that only one process can have the file open at any given time. Using the `MPI_Barrier` routine, modify your code so each process *in turn* opens, appends to, then closes the output file.

7 Rotating information using a Cartesian topology

For a 1D arrangement of processes it may seem a lot of effort to use a Cartesian topology rather than managing the processes by hand, e.g. calculating the ranks of the nearest neighbours. It is, however, worth learning how to use topologies as these book-keeping calculations become tedious to do by hand in two and three dimensions. For simplicity we will only use the routines in one dimension. Even for this simple case the exercise shows how easy it is to change the boundary conditions when using topologies.

1. Re-write the passing-around-a-ring exercise so that it uses a one-dimensional Cartesian topology, computing the ranks of the nearest neighbours using `MPI_Cart_shift`. Remember to set the periodicity of the boundary conditions appropriately for a ring rather than a line.

2. Alter the boundary conditions to change the topology from a ring to a line, and re-run your program. Be sure to run using both of the initial values, i.e. $rank$ and $(rank + 1)^2$. Do you understand the output? What reduction operation is now being implemented? What are the the neighbouring ranks for the two processes at the extreme ends of the line?

3. Check the results agree with those obtained from calling an `MPI_Scan` collective routine.

7.1 Extra exercises

1. Measure the time taken for both periodic and non-periodic topologies – how does it vary with P?

2. Extend the one-dimensional ring topology to a two-dimensional cylinder (periodic in one direction, non-periodic in the other). Perform two separate reduction operations, one in each of the two dimensions of the cylinder.

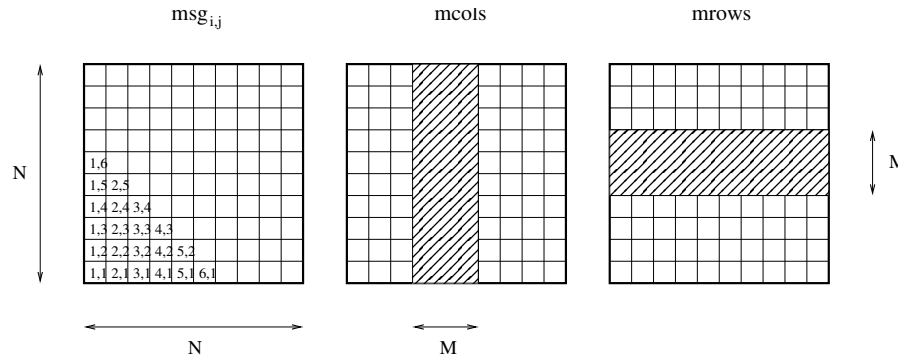

Figure 2: Diagrammatic representation of the $mcols$ and $mrows$ matrix subsections

8 Derived Datatypes

We will extend exercise 5 to perform a global sum of a non-basic datatype. A simple example of this is a compound type containing both an integer and a double-precision floating-point number. Such a compound type can be declared as a structure in C, or a derived type in Fortran, as follows:

```
struct compound                 type compound
{
  int    ival;                    integer          :: ival
  double dval;                    double precision :: dval
};
                                end type compound

struct compound x,y;            type(compound) :: x,y

x.ival = 1;                     x%ival = 1
y.dval = 9.0;                   y%dval = 9.0
```

If you are unfamiliar with using derived types in Fortran then I recommend that you go straight to exercise number 2 which deals with defining MPI datatypes to map onto subsections of arrays. This is, in fact, the most common use of derived types in scientific applications of MPI.

1. Modify the ring exercise so that it uses an `MPI_Type_struct` derived datatype to pass round the above compound type, and computes separate integer and floating-point totals. You will need to use `MPI_Address` to obtain the displacements of `ival` and `dval`. Initialise the integer part to $rank$ and the floating-point part to $(rank + 1)^2$ and check that you obtain the correct results.

2. Modify your existing ping-pong code to exchange $N \times N$ square matrices between the processes (`int msg[N][N]` in C or `INTEGER MSG(N,N)` in Fortran). Initialise the matrix elements to be equal to $rank$ so that you can keep track of the messages. Define `MPI_Type_contiguous` and `MPI_Type_vector` derived types to represent $N \times M$ (type $mcols$) and $M \times N$ (type $mrows$) subsections of the matrix, where $M \leq N$. Which datatype to use for each subsection depends on whether you are using C or Fortran. You may find it helpful to refer to Figure 2 to clarify this, where I draw the arrays in the standard fashion for matrices (indices start from one rather than zero, first index goes down, second index goes across).

Set $N = 10$ and $M = 3$ and exchange columns 4, 5 and 6 of the matrix using the $mcols$ type. Print the entire matrix on each process after every stage of the ping-pong to check that for correctness. Now modify your program to exchange rows 4, 5 and 6 using the $mrows$ type.

8.1 Extra exercises

1. For the compound type, print out the values of the displacements. Do you understand the results?
2. Modify the program that performed a global sum on a compound datatype so that it uses an MPI collective routine. You will have to register your own reduction operation so that the MPI library knows what calculation you want to be performed. Remember that addition is not a pre-defined operation on your compound type; it still has to be defined even in the native language.
3. Modify your ping-pong code for matrix subsections so that you send type $mcols$ and receive type $mrows$. Do things function as you would expect?

9 Global Summation Using a Hypercube Algorithm

Although you should always perform global summations by using `MPI_Reduce` or `MPI_Allreduce` with `MPI_Op=MPI_SUM`, it is an interesting exercise to program your own version using a more efficient algorithm than the previous naive "message-round-a-ring" approach.

A more efficient method for a number of processes P that is a power of two, $P = 2^D$, is to imagine the processes are arranged in a D-dimensional hypercube. In 2 dimensions, the 4 processes are arranged in a square; in 3 dimensions in a cube; ... The coordinates of the processes in the hypercube are taken from the binary representation of the rank, ensuring that exactly one process sits at each vertex of the hypercube. Processes operate in pairs, swapping partial sums in each dimension in turn.

Figure 3 illustrates how this works in three dimensions (i.e. $D = 3$ with $P = 2^3 = 8$ processes).

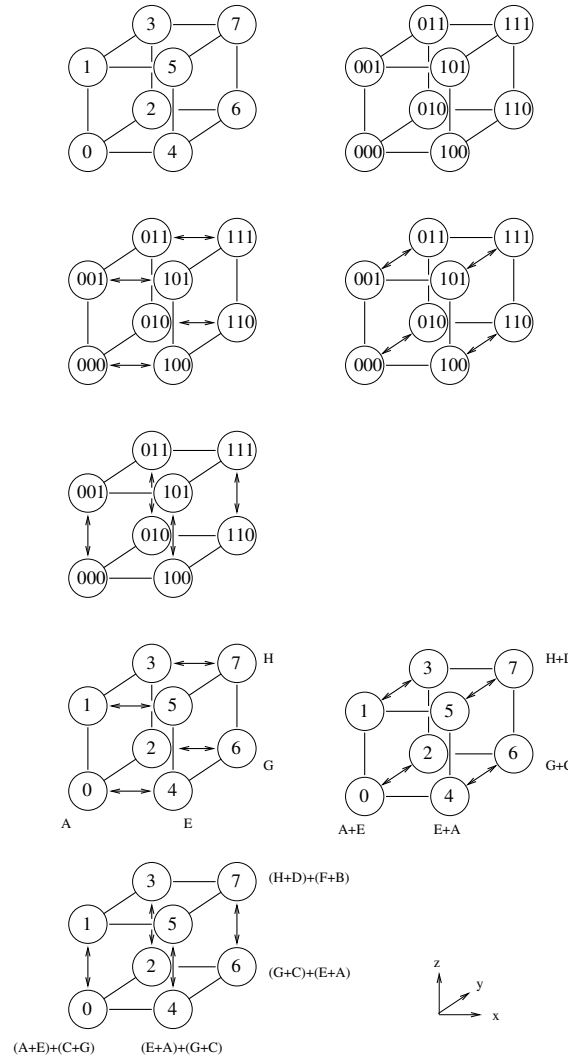

Figure 3: Communications pattern for global sum on 8 processes

An elegant way to program this is to construct a periodic cartesian topology of the appropriate dimension and compute neighbours using `MPI_Cart_shift`. When each process swaps data with its neighbour you must ensure that your program does not deadlock. This can be done by a variety of methods, including a ping-pong approach where each pair of processes agrees in advance who will do a send followed by a receive, and who will do a receive followed by a send. How do you expect the time to scale with the number of processes, and how does this compare to the measured time taken by `MPI_Allreduce`?