|epcc|

International HPC Summer School 2022
MPI and OpenMP Exercises

David Henty, EPCC

1 Setup

I assume you have cloned the repository `https://github.com/davidhenty/ihpcss2022.git` and followed the instructions in the *Bridges2 Crib Sheet* so that you can log on to Bridges2 and compile and run parallel programs in C, Fortran and Python (as appropriate).

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.

You have been given solutions in C, Fortran and Python that set $N = 840$. This number is divisible by 2, 3, 4, 5, 6, 7 and 8 which is convenient when you parallelise the calculation!

The algorithm is as follows:

1. Different processes 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 $i = \frac{N}{2}+1, \frac{N}{2}+2, \ldots, N$. Each process records its own partial sum in `partialpi`.
2. We now accumulate these partial sums by sending them to a controller (e.g. rank 0) to add up:
    - all processes (except the controller) send their partial sum to the controller
    - the controller receives values from all the other processes, adding them to its own partial sum

The program uses the MPI routines `MPI_Ssend` (synchronous send) and `MPI_Recv`.

Note that:

- In real MPI programs you would use standard mode `MPI_Send` not synchronous `MPI_Ssend`. However, it is much better to use synchronous send when developing programs as its behaviour is completely predictable, unlike `MPI_Send` which is may be either synchronous or asynchronous depending on circumstances.
- For a standard reduction pattern like this, you would use the collective call `MPI_Reduce` in a real program. However, implementing it by hand using point-to-point communications makes a very good teaching exercise.

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. Use the function `MPI_Wtime` (see above) 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 the calculation takes a sensible amount of time (e.g. more than a second) you will probably have to perform the calculation of π thousands of times or use a very large value of N.
2. Ensure your program works correctly if N is not an exact multiple of the number of processes P.
3. 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).
4. 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?
5. 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?
6. 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 Traffic Model

The aim of this exercise is to take a simple example, the 1D traffic model as described in the lectures, and parallelise it in various ways. Although this code is probably not sufficiently complicated to show substantial performance differences, the idea is for you to experiment with different techniques in the simple traffic model before implementing them in a real MPI application.

You will be given source code, implemented in C, Fortran and Python, that contains:

1. a working serial code;
2. a working MPI code;
3. a working OpenMP code (not in Python).

You can compile the C and Fortran codes using the supplied `Makefile` by typing `make`. Whenever you make any changes, type `make` again and the relevant files will automatically be recompiled.

3.1 Verification

The way it is written the code should give **exactly** the same answer for C and Fortran regardless of whether it is serial or parallel. For a road of length 320000000 and a target density of 0.52, the velocity at iteration 100 should be 0.898955 (to 6 decimal places). If you get a different answer then your code is probably wrong!

Python uses a different random generator, but the serial and parallel answers should still be identical. For a road of length 320000000 and a target density of 0.52, the velocity at iteration 100 should be 0.898960.

Note that my simple MPI parallelisation assumes that N is an exact multiple of P and will give incorrect answers if this is not the case.

You can experiment with increasing the length of the road – just change the value of `ncell` near the top of the main program. If you change the road length, you should re-run the serial code to get a new baseline result. The length of the road cannot be larger than 2 billion as it is stored in a standard integer.

3.2 MPI exercises

These range from fairly basic to very advanced. Please attempt whatever ones you feel comfortable with.

1. Run the MPI code on a range of process counts (e.g. 1, 2, 4, 8, 16, 32, 64 and 128) and plot a graph of the speedup against the number of processes.
2. Do the same with very short and very long roads and compare the performance results.
3. The global sum requires a parallel reduction `MPI_Allreduce` which is an expensive operation. Modify the code so it only calls reductions when necessary; investigate the effect on performance.
4. Replace the `MPI_Sendrecv` call using non-blocking operations. There are a number of options:
   (a) a non-blocking send (`MPI_ISsend`), a blocking receive (`MPI_Recv`) and a wait (`MPI_Wait`);
   (b) a non-blocking receive, a blocking send and a wait;
   (c) a non-blocking send, a non-blocking receive and multiple waits (e.g. `MPI_Waitall`).

   When developing the non-blocking code, you should use synchronous send (i.e `MPI_Ssend` / `MPI_Issend`) as opposed to standard send (`MPI_Send` / `MPI_Isend`). This ensures an incorrect code is guaranteed to deadlock; with standard sends, an incorrect code might just happen to run correctly which is not desirable.

   Once working you can switch to `MPI_Isend` which should improve performance.

5. Put an `MPI_Barrier` at the end of every iteration. Although this is **completely unnecessary**, many people do this kind of thing "just to be safe". What is the effect on performance?
6. Implement the non-blocking halo exchange using *persistent communications*. In the case of running on 2 processes, where the up and down neighbour are both the same process, can you guarantee that the sends and receives will match correctly?
7. Try to overlap some of the computation with communications. The way to do this is:
   - issue non-blocking sends and receives for the halo cells;
   - update the interior cells $i = 2, 3, \ldots, N-1$ which do not depend on the halo cells;
   - wait for the halo data to arrive;
   - update the boundary cells $i = 1$ and $i = N$.

   The idea here is that the communications takes place at the same time as the calculation of the interior cells, thus speeding the program up. Note that whether this works in practice very much depends on the details of your MPI implementation and the network.
8. Minimise the latency overhead by using deep halos, i.e. exchange more than one boundary element and then do multiple iterations before the next halo swap. It is much easier to implement an explicit depth first (e.g. 2 halo points) before attempting the general case of depth D. For $D = 2$ you would swap two halo cells every second interation, and update the entire road twice each iteration.
9. Combine deep halos with overlapping communication and calculation.

3.3 OpenMP exercises

1. Run the OpenMP code on a range of thread counts (e.g. 1, 2, 4, 8, 16, 32, 64 and 128) and plot a graph of the speedup against the number of threads. How does this compare to the MPI speedup?
2. Before the first call to `initroad()`, zero both the `oldroad` and `newroad` arrays **using a parallel loop**. What is the effect on performance for small and large numbers of threads? Do you understand what is happening here?

3.4 Hybrid traffic model

It should be quite simple to add OpenMP directives to the MPI code to make a hybrid code using the master-only model. Parallelise the computation in `updateroad()`, and the copy-back step in `main()`, exactly as in the pure OpenMP code. You should be able to leave the MPI communications as they are.

Compare performance for different numbers of processes or threads on the same number of cores. For example, on 2 nodes with a total of 256 cores you could use: 256 processes (P) with 1 thread per process (T); ($P = 128, T = 2$); ($P = 64, T = 4$); ...; ($P = 2, T = 128$).

3.5 Process placement

A very important thing to investigate when running a Hybrid MPI/OpenMP code on any parallel system is to ensure that the process and threads are distributed correctly across nodes and cores.

By default, your a hybrid MPI/OpenMP program may not achieve good performance on Bridges2 as the process and threads are not optimally allocated to the available CPU cores. For example, you may see that using 8 processes each with 2 threads is much slower than 16 processes each with single thread.

See if you can find out the information you require from the Bridges2 user guide:
`https://www.psc.edu/resources/bridges-2/user-guide-2/`.

Phrases to look for include *process binding*, *thread binding*, *process affinity* and *thread affinity*.