epcc



Exercise: Parallel Traffic Modelling

David Henty, EPCC

1 Introduction and Aims

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 both C and Fortran, that contains:

- 1. a working serial code;
- 2. a working MPI code;
- 3. a working OpenMP code;
- 4. Makefiles and template batch submission scripts for ARCHER2.

The exercises below range from fairly basic to very advanced. Please start at whatever level you want to!

2 Instructions

Obtain the code as traffic.tar from the course web page.

2.1 Verification

The way it is written the code should give **exactly** the same answer for both languages regardless of whether it is serial or parallel. For a road of length 100000, the velocity at iteration 2000 should be 0.931989 (to 6 decimal places). If you get a different answer then your code is probably wrong!

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 decide to use a road of a different length, you should re-run the serial code to get a new baseline result. Note that the way the MPI code is implemented, it assumes that the **length of the road** is an **exact multiple** of the number of processes.

The number of iterations is set quite low and you may wish to increase this when measuring performance so that the parallel code still runs for a reasonable length of time (e.g. for around a second). Simply multiply the value of maxiter by, say, a factor of 10:

```
maxiter = 200000000/ncell;
maxiter = maxiter*10;
printfreq = maxiter/10;
...
```

3 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, 5, 10, 20, 40 and 50) 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 with separate calls to MPI_Send and MPI_Recv and check that the code still runs correctly. Note that you have probably written a technically incorrect code because MPI_Send could be implemented as a synchronous send.
- 5. Replace the call to MPI_Send with MPI_Ssend to guarantee that it is implemented synchronously. As it stands, your code will probably deadlock (see Section 3.1 for more exercises on this topic).
- 6. Fix the deadlock by arranging the processes into odd/even pairs where, in each pair, one process sends first then receives and the other receives first then sends.
- 7. Solve the deadlock problem 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.

- 8. 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?
- 9. 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?
- 10. Try to overlap some of the computation with communications, i.e. update the interior cells of the road while waiting for the halo data to arrive.
- 11. 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*.
- 12. Combine deep halos with overlapping communication and calculation.
- 13. Perform the halo swaps the neighbourhood collective operation MPI_Neighbor_alltoallv. You will need to create a 1D Cartesian topology using MPI_Cart_create with periodic boundary conditions.

3.1 Non-periodic boundary conditions

The reason that the traffic model deadlocks with a naive implementation using synchronous sends is because of the periodic boundary conditions: all processes are trying to send at the same time. With non-periodic boundaries, however, processes at the extreme ends of the road will not need to send or receive some of the data and the deadlock is broken.

Update the traffic model to use non-periodic boundaries:

- 1. Set the halo at the exit of the road (on the last process) to be zero so cars are always free to leave;
- 2. Each iteration, set the halo data at the entrance to the road (on the first process) to be 1 with probability density, otherwise set to zero. This maintains the same density of traffic for the whole simulation; see initroad() to see how to call the random number generator.
- 3. Run the code on one and many processes; does it give the same answer? Do you understand why?
- 4. Implement the halo swapping using blocking synchronous sends and receives (MPI_Ssend / MPI_Recv); unlike the periodic case, the code should not deadlock.
- 5. Compare the performance with the version using MPI_Sendrecv and using MPI_Send/MPI_Recv. Although it works correctly, the synchronous version should be slower; do you understand why?

4 **OpenMP exercises**

- 1. Run the OpenMP code on a range of thread counts (e.g. 1, 2, 6, 12 and 24) and plot a graph of the speedup against the number of threads.
- 2. Change the code so it only computes the total number of moves when absolutely necessary. How does the impact on performance compare with the equivalent optimisation in MPI?
- 3. Before the first call to initroad(), zero both the oldroad and newroad arrays using a parallel loop. With a very long road (e.g. at least ten million cells), what is the effect on performance for small and large numbers of threads? Do you understand what is happening here?
- 4. Minimise the overhead of creating parallel regions by enclosing the entire iteration loop with a single parallel region. What is the effect on performance? Note that this is **very challenging**, e.g. you will now need to implement updateroad() using orphaned directives.

5 Hybrid MPI/OpenMP

5.1 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.

- 1. Set the length of the road to include a factor of 128 (e.g. use 128000) so you have more freedom to vary the number of processes.
- 2. 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).

5.2 Process placement

A very important thing to investigate when running a Hybrid MPI/OpenMP code on any parallel system is to check you know how 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 exmaple, you may see that using 8 processes each with 2 threads is much slower than using 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.