|epcc|

Advanced Parallel Programming Exercises
MPI RMA Performance on ARCHER

Daniel Holmes

27th October 2015

1 Introduction

The purpose of this exercise is to investigate the performance of MPI single-sided operations on ARCHER. You are given a simple ping-pong code that exchanges messages of increasing size between two ranks, and prints out the average time taken and the bandwidth over a large number of repetitions. With this code you can investigate how the following factors affect the performance:

- different communication operations (put, get);
- the synchronisation method (active - fence, active - PSCW, passive - shared, passive - exclusive);
- using the ARIES interconnect or local memory copies;
- which cores are used within a node;
- sending contiguous or strided data.

2 Compiling and Running

The code is contained in `APP-pingpong.tar` on the APP web pages. You should be able to compile it using `make`, and submit the supplied PBS script unchanged.

By default, the code runs on two processes each placed on different nodes (so communications will be over the network), and benchmarks standard and synchronous sends. Note that **the program also prints out the exact location of the two processes**.

For each mode, two ".plot" files are written containing the times and the bandwidths as a function of message size. The results for the two different modes can be compared using:

```
gnuplot -persist plot_time.gp
gnuplot -persist plot_bandwidth.gp
```

Check that you understand the general form of the graphs before proceeding.

3 Experiments

The supplied gnuplot ".gp" files compare the results for standard and synchronous modes. If you want to compare different modes, or more than two modes, you will have to edit the gnuplot files – the format should be self-explanatory. Also note that the code overwrites the ".plot" files so you will need to make copies after each run if you want to keep a record of the results (e.g. when changing the placement of processes on the cores or nodes).

If you run on more than two processes, the program sends messages between the first (rank zero) and last ranks with all the other processes remaining idle.

This is useful when altering the assignment of processes to cores. By varying `mppwidth` and the `-n` argument to `aprun` you should have complete control of the placement of the first and last processes. The `-N`, `-S` and `-d` options to `aprun` give even finer control, but things can get complicated quite quickly ...!

You should experiment with the following:

1. Modify the code so that it creates a window before any communication takes place and destroys that window after all communication has finished.
2. Modify the code so that it includes RMA synchronisation function calls forming an epoch around each iteration of the ping-pong communication pattern. Use active target synchronisation via the fence operation, initially. Later change this to use other synchronisation methods.
3. Modify the code so that it uses single-sided communication operations instead of point-to-point operations. Initially, treat the sender from the point-to-point version as the origin and "put" data directly into the memory of the process that was the receiver in the point-to-point version. Later change this to treat the receiver process as the origin and "get" data directly from the memory of the sender process.
4. Run on a single node. Change the location of the processes within a node (e.g. core 0 with core 1, instead of core 0 with core 23).
5. Run on multiple nodes. Change the location of the processes across nodes (e.g. core 0 on node 0 with core 0 on node 1, instead of core 0 on node 0 with core 23 on node 1).
6. Can you explain any variations in the latency and bandwidth figures within an ARCHER node (as you change the process locations) in terms of the cache and memory structures of the Intel processors?
7. Try strided data using a strided datatype or using multiple single-sided communication operations.