

MPI Performance on ARCHER

Introduction

The purpose of this exercise is to investigate the performance of basic point-to-point MPI operations on ARCHER. You are given a simple ping-pong code that exchanges messages of increasing size between two ranks then 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 underlying protocols for different message sizes;
- different communication hardware (e.g. the ARIES interconnect or memory copies);
- different NUMA regions within a node (if any - ARCHER only has one per CPU);
- different MPI send modes (e.g. synchronous vs. buffered vs. ready);
- different MPI data-types (e.g. contiguous or strided data).

Compiling and Running

The code is contained in `pingpong.tar` on the ARCHER web page for the course. 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 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.

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 (rank zero) and the last rank with all the other processes remaining idle.

This can be useful when altering the assignment of processes to cores. By varying `-l select` and the `-n` argument to `aprun` you should have sufficient 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. Run on a single node, i.e. with 24 processes.
2. 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).
3. Change the location of the processes within a node (e.g. core 0 with core 1, instead of core 0 with core 23).
4. Can you explain any variations in the latency and bandwidth figures (as you change the process locations)?
5. Change the threshold for eager sends ũ see the PBS script or the MPI man page on ARCHER for how to do this with the environment variable `MPICH_GNI_MAX_EAGER_MSG_SIZE`.
6. Try investigating the performance of buffered mode sends or strided sends (using MPI data-types).

For all but the last option you should only need to alter the PBS batch script ũ the ping-pong program itself can be left unchanged.

Finer Control

For the ping-pong example, you are only concerned about the placement of the first and last processes. However, for a general code you care about the placement of all the processes.

Process placement across nodes

You can alter the way MPI processes are allocated to nodes at runtime using a number of flags to `aprun`.

The option `-N <procs per node>` specifies how many MPI processes to place on each of the 24-core nodes of ARCHER. With 24 MPI processes (`-n24`) you could compare the performance of having all the processes in the same node (`-N24 -l select=1`) against having only 4 processes per node using 6 nodes (`-N4 -l select=6`). Reducing the number of processes per node will increase the number of MPI messages that have to go over the network. However, each process will have a greater share of the resources on a node (such as memory bandwidth) so the performance may vary in interesting ways. You should look at both small and large message sizes.

Note that when you change the `-n` and `-N` flags to `aprun` you must also change the PBS option `-l select` to match. In order to check whether the distribution of MPI processes is as you expect, you can use the MPI call `MPI_Get_processor_name` which will return a different name on each node.

An alternative, which only works with MPICH, is to set the environment variable `MPICH_CPUMASK_DISPLAY=1` in your jobscript.

Process placement within a node

The 24-core shared-memory nodes are basically made up of two 12-core CPUs. Within each 12-core CPU, memory access is symmetric; however, accessing memory from a different CPU will be slightly slower. These CPUs are referred to as `$NUMA regions`ũ in the documentation.

You can alter how processes are assigned to the cores using the `-S <procs per NUMA region>` option. The default is 12, meaning that if you run 12 MPI processes per node then they will all be allocated to the first of the two NUMA regions. Try altering this parameter so that the processes are spread out more evenly, for example if you have 8 processes per node then try `-S 4`.

The `-d` option specifies the *stride* between the cores, so `aprun -n 24 -N 12 -S 6 -d 2` would ensure that the 6 processes on each NUMA region are assigned to the even-numbered cores (0, 2, 4, 8, 10 and 12).

Documentation

You can find out details about Cray MPI by typing `man mpi` on ARCHER. For example, this explains what limits can be changed using environment variables which include additional thresholds such as when RDMA is used. You should also look at the ARCHER web pages:
<http://www.archer.ac.uk/documentation/userguide/>.

Porting to Other Systems

You should be able easily to port the ping-pong code to other machines such as Morar. Note, however, that the code that works out the precise location of the MPI ranks on the various cores is specific to Linux: you may need to remove the calls to `printlocation()` from the main program, and just compile `pingpong.c` on its own without including `location.c`.