School of Computing, University of Leeds

COMP3221
Parallel computation

Worksheet 2: Distributed memory parallelism with MPI

This worksheet will give you an opportunity to get some hands-on practice with MPI, relevant to distributed memory systems.

Compiling and running MPI programs

On school machines, you will need to load some modules before you can use the MPI command line tools to build and launch executables. Please see the slide 'Install MPI' towards the end of Lecture 8 for details. If these do not work, please post a query to the Teams page for this module.

Once everything has been loaded, you can compile MPI-C code using `mpicc`, *e.g.*

```
mpicc -Wall -o <executable_name> <source_code>.c
```

To run p processes on the local machine:

```
mpiexec -n p ./<executable_name>
```

If you attempt to launch with more processes than available cores, you may receive an error message along the lines of *'too many slots'*. To avoid this, add the argument `-oversubscribe` to the above line *before* the executable name.

MPI error messages

If an MPI program fails, the error message is rarely very informative. If you cannot decide which MPI call is causing problems, note that (almost) all MPI functions return an error code: If this code is *not* `MPI_SUCCESS`, then that call is failing. Unfortunately other error codes are non-standard and depend on the implementation, and proper MPI error handling is messy and not covered in this module. Generally, the best principle is to develop code incrementally, checking at each stage that it is working correctly.

Questions

1. *[Lecture 9]* Download the code `vectorAddition.c` from the Minerva page for Lecture 9, and try to understand how it works. Use this as the basis for a simple problem where every element of a vector `a` is doubled. In serial this would be: `for(i=0;i<N;i++) a[i]*=2;`

2. *[Lecture 9]* Download the code `cyclicSendReceive.c` from the Minerva page for Lecture 9, compile it, and run with a single command line argument to give the array size (instructions in comments at start of the file). Increase the data size until you exceed the supported buffer capacity and deadlock, as explained in the lecture notes. Now change the code to use staggered send-and-receives as explained towards the end of the lecture, and confirm it no longer deadlocks even for large arrays.

3. *[Lecture 10]* Download the code `distributedCount.c` from Minerva, understand how it works, and then replace the loops of point-to-point communication with collective communication calls as covered towards the end of the lecture. You can do this progressively, *i.e.* replace Step 1 first of all, make sure the code still works, then Step 2, and finally Step 4 (Step 3 is purely computation and need not be replaced). Note Step 4 requires a little more work as you will also need to create an array of size `numProcs` on rank 0 to gather together all of the local counts, which you can then sum over to get the total.

4. *[Lecture 11]* Continuing from your answer to Question 4, replace the final step of the calculation with a single call to `MPI_Reduce()`.

5. *[Lecture 11]* Download the code `work2_q5.c` from Minerva, compile and execute with 2 processes as explained in the comments near the start of the file. This code evaluates the natural logarithm of 2, $\ln(2)$, using the series expansion

$$\ln(2) = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7} - + \ldots$$

   Executing on two processes means that all of the positive terms are calculated on rank 0, and all of the negative terms are calculated on rank 1, before being combined using `MPI_Reduce()` to give the final answer.

   Increase the number of terms (the command line argument), starting from 10 and increasing in powers of 10. When you get to *e.g.* 100000000, you should notice the approximation is starting to break down. Look at the code and uncomment the line that prints each rank's partial sum. Can you understand why the problem occurred? What does this tell you about the associativity of finite precision floating point addition?

6. *[Lecture 12]* Download the code `heatEqn.c`, compile and execute on 4 processes. You should be presented with the solution to the heat equation problem initially, and again after 10 iterations. Note the dividers denote the domains computed by each process; try launching on 16 processes to see this. The initial value for each domain is set to `rank+1` so you can see which rank is responsible for which domain.

   Look through the main iteration loop in `main()` to roughly understand how it works (don't worry to much about the details of what's being sent and received). Change the communications across the 'upper' and 'lower' boundaries to use non-blocking communication instead. Where should you put the corresponding calls to `MPI_Wait()` to ensure correct results but improved performance? Be careful to ensure you only wait for communications you have actually started!

   You may also like to similarly change the 'left' and 'right' communications, but these are slightly more tricky and will take more time.

7. *[Lecture 13]* For this lecture just download the code `Mandelbrot_MPI.c` and the corresponding `makefile` from Minerva, and try to understand how the key routines work. These were highlighted towards the end of the lecture. Note there is a `#define WORK_POOL` near the start of the code that, if commented out and recompiled, results in the strip partitioning version (but still using a master process to plot the graphics).