Pipelines

Nick Brown

31 January 2017

1 Introduction

In the lecture we looked at the pipeline pattern. Imagine that the geologists using our pollution code want to do so in a more automated, high volume approach. They want to take some raw data (for instance produced by the pollution measuring device at each end), feed this into the simulation and then for the code to perform some (very simple) data analysis and write these results to an output file. Many of these raw data input files have been produced (in the ***data*** directory) which represent different locations at a specific site and hence you need to write a parallel code to handle this. In each file there are two groups of thirty samples, the groups represent the values at the left and right of the pipe, with thirty samples taken at each end - these samples are quite noisy.

2 Pipeline

Your pipeline will have five stages as per Figure 1.Code has been provided (on Learn) which implements the actual core work done by each stage of the pipeline and it will be your task to hook these different stages up together using MPI.

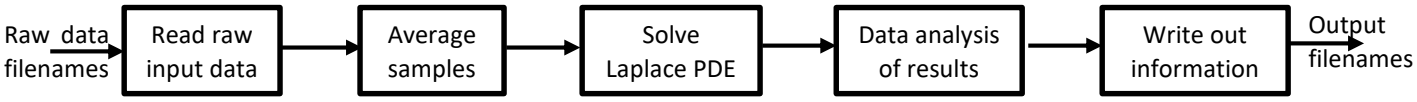

Figure 1 - Pipeline illustration

The file names for the raw data input files will be fed into pipeline stage one (provided via command line arguments to the code) which is in the function ***read_files*** in the provided code. Each data file contains sixty values (2 groups of thirty values, each group represents either end of the pipe.) These values should then be passed to the second stage, ***average_sample_values***, which will average each group, to produce two values – an averaged pollution level at the left end and an averaged pollution level at the right end. These are passed to the third stage, ***perform_calculation***, which solves the Laplace PDE (our current serial 1D code) and then passes the entirety of the pipe (and boundary values) to the next stage. The fourth stage, ***data_analysis*** in the provided code, will perform data analysis identifying two integer values - the specific point in the pipe where the pollution is at a certain threshold (i.e. where the clean-up would need to work to) and the number of points which are equal to or above this threshold. These two integers are passed to the fifth stage in the pipeline, ***write_data*** which will write the analysed results out to a file and output the name of the results file, each results file will be a slightly different name.

You will need to perform a number of tasks in order to get this pipeline working (you might find it helpful to refer to the slides of lecture 5 where I sketched out the general communication code involved):

1. In the program entry point (the ***main*** function in C and ***run_pipeline*** subroutine in Fortran) you will need to take the MPI rank and, depending upon the rank call one of the specific stages in the pipeline (one rank calls one stage).
2. Now hook up the communication between stages, stages 2, 3, 4 and 5 need to receive some data from the previous stage and stages 1, 2, 3 and 4 need to send their resulting data to the next stage. Each stage can only proceed when the previous stage has data ready for it, so these can be blocking calls.
3. Consider the termination aspect, the code is currently set up for each pipeline stage to loop indefinitely. Each stage needs to be ``instructed'' by the previous stage in the pipeline that it should terminate and this is often via a sentinel or poisoned pill message. You should check the message received from the previous stage for this specific criteria. There are a variety of different ways to do this, one way is for the previous stage to send an empty message (i.e. message size of zero) and the receiving process to get the number of elements in the message using ***MPI_Get_count*** and the status. Before quitting, each stage in the pipeline should instruct the next stage to shutdown via which ever approach you choose.

The path to each input file is provided as separate (delimited by a space) command line argument. The submission script ***subpipeline.pbs*** is provided and this will loop through and build command line arguments to provide each raw data file to the pipeline as arguments to the executable. You can execute your pipeline code on the login node via ***mpirun***, I would suggest initially testing your parallelisation with just one or two of the raw data files until you are happy the data is flowing and results are coming out as expected.

3 Advanced exercises

This section is for those who have completed the pipeline and wish to further explore the example. Don't worry if you don't get onto doing these, the most important thing is that you get the basic pipeline working.

1. The third stage, Laplace calculation, will likely be the most computationally intensive part of this code - especially if the pipe is long. The problem with this is that whilst the Laplace calculation is taking place, the previous stage (stage 2) is idle stuck waiting to send its data onto the third stage and stages 4 and 5 are idle waiting for data. Replace this third stage with your geometrically parallelised 1D code so that this stage in the pipeline is in itself parallelised. You probably want to have two MPI communicators, one for the pipeline processes and one for the Laplace geometric decomposition processes.
2. Instead of parallelising the third stage we could instead have multiple third stages, all running on different processes. Modify your code so that multiple third stages of the pipeline can run and the second stage selects which third stage process to use in a round robin fashion.
3. So far we have assumed the length of the pipe (100 elements.) Modify the code (and input files) so that the length of the pipe is provided as an additional entry in the raw data files and passed throughout the pipeline. Because the length of the pipe is an integer now stages 1, 2 and 3 of the pipeline now communicate both doubles and a real, construct your own MPI types to handle this.