Pipelines

1 Introduction

We have talked about the pipeline pattern in the lecture. 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. Skeleton code has been provided 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.

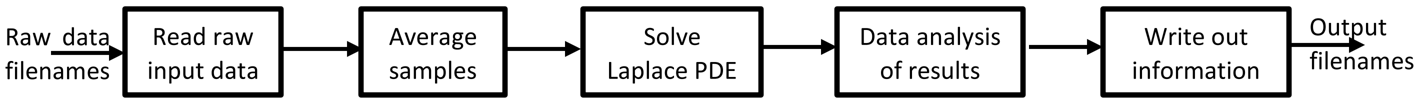

Figure 1 - Pipeline illustration

The file names for the raw data input files will be fed into pipeline stage one (provided via command line arguments to the code, this is done for you in the submission script) which is in the function ***read_files*** in the provided C code, or ***read_data_points*** in the Fortran 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*** in the C code and ***run_solver*** in the Fortran, 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*** in the C code and ***write_values*** in Fortran, 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.

We will be using MPI for this practical, if you are not so familiar or a bit rusty with this then the course materials of a recent ARCHER course at http://www.archer.ac.uk/training/course-material/2018/07/mpi-epcc/index.php are a good reference. A good reference for the MPI API can be found at https://www.open-mpi.org/doc/v3.0/

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 the pipeline lecture):

1. In the program entry point (the ***main*** function in C and ***run_pipeline*** subroutine in Fortran) you will need to get hold of the MPI rank and, depending upon the rank call one of the specific stages in the pipeline (one rank calls one stage, rank two stage two etc).
2. Now hook up the communication between the stages. Stages 2, 3, 4 and 5 need to receive some data from their previous stage and stages 1, 2, 3 and 4 need to send data that they have generated to the next stage. Each stage can only proceed when the previous stage has data ready for it, so these can be blocking MPI point to point 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 done 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, probably the easiest 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*** from the status. If the message length is zero, then it's time to quit. Before quitting, each stage in the pipeline should instruct the next stage to shutdown via which ever approach you choose (e.g. by sending an empty message).

The path to each input file is provided as separate (delimited by a space) command line argument. The submission script we have provided will loop through and build the command line arguments for you. You can change this in the submission script and 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. The first few arguments are to the solver and are the same as those used in practical one (pipe size in X and Y, termination residual and maximum number of iterations).

3 Calculating the load imbalance

With pipelines it is important that each stage does approximately the same amount of work as other stages. If this is not the case, then it can result in significant load imbalance between the UEs. Remember that the Load Imbalance Factor, *LIF = maximum load* ***/*** *average load*.

1. Each UE is already calculating its local active time by contributing to the ***active_time*** variable. The simplest approach is for each process to dump this out at the end of execution to stdio and then you can manually calculate the average and divide the maximum load by this.
2. The manual approach is quite tiresome, especially as we increase the number of UEs in the next section. Therefore, calculate the LIF in code – you can use two reductions, one to sum up all the local ***active_time*** variables (and then divide it by the number of UEs on the root), and the other to find the maximum ***active_time*** variable. Once you have these global values, just do a simple division and display the LIF to stdio.
3. Bearing in mind a perfectly load balanced problem has a LIF of 1.0 and the factor tells you how much faster the code could run given a balanced load, how do you think the load is balanced in this case?

You can also run this through the BSC extrae tool, like in practical one by loading the ***bsctools/extrae/3.4.1-cray-mpich*** module, issuing ***make clean*** and then ***make instrumented***, followed by uncommenting the two appropriate lines in the submission script before submitting the job. The results can be viewed with the wxparaver tool. This might help illustrate the source of any load imbalance.

4 Advanced exercise: Improving the load imbalance

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.

The third stage, Laplace calculation, is the most computationally intensive part of this code and is one of the main factors behind the large LIF. 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. In this section we will modify the 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.

1. Copy your pipeline so you don't overwrite what has been already written, then work on this copied file.
2. In selecting which UEs execute which stage of the pipeline, up until this point we have assumed one UE per stage. That is no longer the case, in selecting which stage runs on which UE, the logic around the first two stages can remain unchanged. The last two stages will be mapped to the last two UEs (in the code ***size-1*** and ***size-2*** where ***size*** is the number of MPI processes) and the rest of the UEs will execute the third stage. You will need to modify the conditional in the ***main*** C function (***run_pipeline*** Fortran subroutine) to represent this.
3. The second stage, ***average_sample_values*** will need to select the appropriate third stage UE in a round robin fashion. I suggest having a ***next_rank*** integer, initially set to 2, and then incremented on each pipeline send. It will need to wrap around, so when you reach the limit of stage three UEs (when it reaches ***size-2***), it is reset back to 2.
4. In the second stage, ***average_sample_values***, you will also need to send the termination poisoned pill to every stage 3 UE. Probably the easiest way is via a loop.
5. The third stage, ***perform_calculation*** C function or ***run_solver*** Fortran subroutine, will need to know which UE rank the fourth stage is and you will need to update the sending of results and poisoned pill to this rank.
6. You will need to update the fourth stage, ***data_analysis***, to receive from any third stage process. I suggest replacing the rank with ***MPI_ANY_SOURCE***. You will also need to consider termination here. Based on the fact that there are multiple stage trees communicating with stage four, what is the best way of this stage knowing when to terminate? (Hint: For termination is it not enough for just one of the stage three UEs to send the termination poisoned pill to stage four, as other stage three UEs might be working on their calculations and need to pass the results on at some later point.)
7. You will need to update the fifth stage, ***write_data*** in C and ***write_values*** in Fortran, to receive data from the fourth stage (***size-2***) UE.

Once you have done this, compare the LIF of this parallelised pipeline with that of the initial version. What difference does this make to the load balance and overall performance?