Solution Construction

This problem is very closely related to Module 2 – Task 3D. The task was to implement a traffic control simulator program, as in Module 2 – Task 3D, but add MPI to the solution. Because of this, it made sense to use the sequential implementation already developed in the previous task as a skeleton, and develop the solution from there.

A couple of important considerations must be made when implementing the MPI solution. These first occurred to me as:

CONSIDERATION 1:

MPI operates across multiple processes and potentially across multiple hosts, therefore there is no shared memory. This means the buffer cannot be used in the same way as the sequential solution.

Possible solution:

Use the host to co-ordinate access to a single buffer that is stored on the host, and use send/receive/wait calls to receive data produced by another process and send data to a consuming process.

Problem: This is not an efficient way to solve this problem and introduces unnecessary complexity with send and receive/wait MPI calls. Not to mention, this will also cause a severe bottleneck scenario on the host where all processes want to produce data (add it into the buffer on the host) but the host can only accept one at a time, and also consume one at a time to send to the consumer process. While this can be improved with extra threads on the host, this is still a horrendously complex and unnecessary solution that does not scale well to an increasing number of processes.

Implemented solution:

Use a buffer within each process and combine the results of the consumer work within the master node at the end.

Explanation: This solution is both effective and far simpler than the previous one. By initialising a bounded buffer on each process, each process can operate as an instance of the sequential solution and manage their own producer and consumer threads. The only problem then becomes how to read in the correct data from the input file to ensure that processes are not overlapping the data they are processing, and also how to combine the results at the end. The former is explained below in **CONSIDERATION 2**, while the latter is a very trivial problem, and is explained in **CONSIDERTION 3.**

CONSIDERATION 2:

The file read needs to be efficiently parallelised in order to speed up the program and make MPI more useful in this scenario in addition to the implemented solution to consideration 1.

Possible solution:

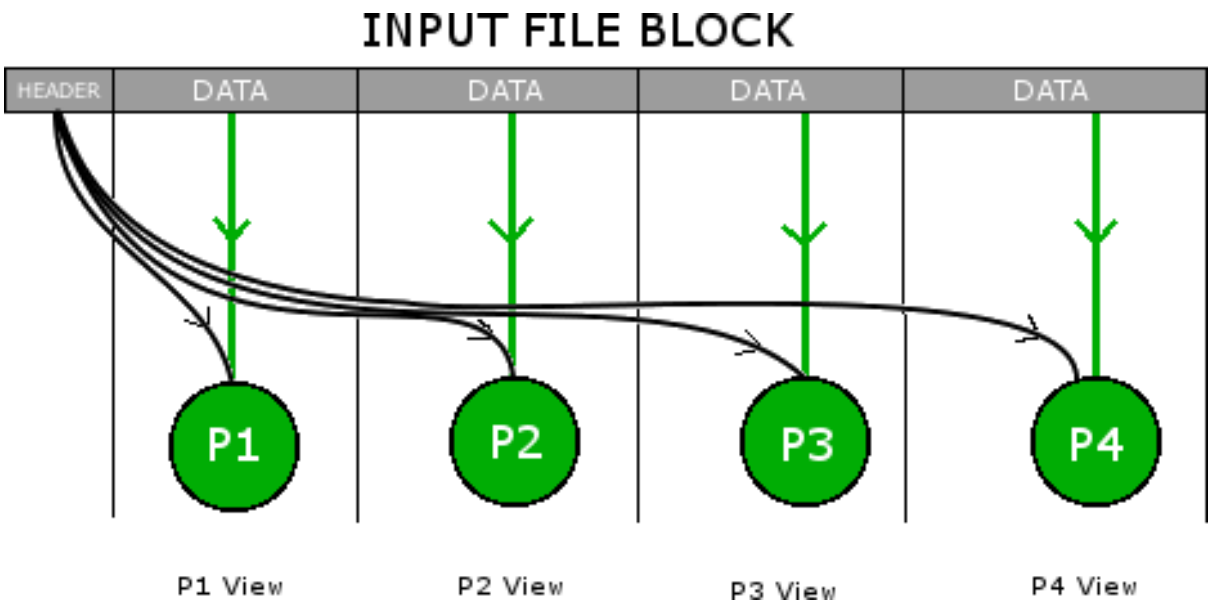
Read in the entirety of the input within each file into each process from a shared location, such as a Cloud folder, and use a combination of process ID and thread ID to determine which chunk of the file the producer threads in each process are allocated

Problem: While this is a simple solution, it is also not very efficient. The main benefit of using MPI for the producer-consumer problem would be if the input data file was so large that it would take a single process too long to read in all the data and also take up large amounts of memory. Reading in the entire file within each process just to access only a portion of it is a very unnecessary waste of memory and time.

Implemented solution:

Use the inbuilt MPI_File type and MPI_File_* calls to read only portions of the file in each process.

Explanation: A powerful tool in MPI is parallel I/O. While it is not designed to read text files line by line, I made some modifications which introduced important limitations to the input text file. MPI I/O reads data in chunks based on a 'file view'. In this case, the file view is set to read contiguous blocks of 24 chars starting from an initial displacement until the file buffer on the process is filled.
The displacement for each process is determined by the process number and the number of lines assigned to each process. Importantly, each data line in the input file MUST be the same amount of characters (in this case, 24). This means that numerical data must be left-padded with 0s to ensure consistency.
Because each line is a uniform length, calculating buffer sizes and number of lines, etc, is easily achievable. Additionally, any leftover lines as a result of (lines % np) are allocated to the last process. The diagram I prepared below shows how the file is split up, and what each process can see and receive. After each process has it's view of the file and has read it into its own char buffer, it can then be dissected by the producer thread function in a similar way to the sequential solution. Note how each process only sees its allocated portion of the file. This saves memory space and is also much faster within MPI parallel I/O. Each process also receives a copy of the file header, which is accessed using a separately created file view. The header contains information like the number of traffic lights.



CONSIDERATION 3 (Resulting from Solution to Consideration 1):

The TrafficCongestion data generated by the consumer threads in each process needs to be merged into a collective data store on the host in order to generate the hourly traffic reports. The current data within each process is stored in a TrafficCongestion struct containing a data field which has type std::map<int, std::map<int, std::vector<int>>>.

Essentially this is just an associative array with data for each hour and traffic light id, with a template like the following:

[hour => [light_id => [measurement1, measurement2, measurement3, …],

light_id => [measurement1, measurement2, measurement3, …],

…],

hour => ….]

Problem: How do we pass the complicated nested std::map object using MPI? It would take some form of serialization on the slave nodes and deserialization on the master node. This is difficult to achieve and also adds some unnecessary complexity to the problem.

Implemented solution

Instead, we know that there can only be a set number of hours (0 – 23) and similarly a set number of traffic lights, determined from the file header. The object associated with each hour and light ID is simply a std::vector<int>, which we are very used to passing across MPI processes from previous work on matrix multiplication and quicksort.

This means we can iterate across every hour and every light ID, using MPI_Gather to first retrieve the number of elements within each processes vector, and then MPI_Gatherv to retrieve the vector data for the given hour and light ID from each process. We can simply then add this to an auxillary TrafficCongestion object on the master node, and that's all there is to it.

Conclusion

MPI can be used to scale out problems such as this with relative ease, as long as you are familiar with the functions that MPI has to offer. Importantly, use of MPI's parallel I/O implementation allows file reads and writes to be split up efficiently across multiple processes without blocking code, which is a major bonus.

Considering the problem from many different perspectives allowed me to see the positives and negatives of each, allowing me to formulate an efficient and simple solution that made use of MPI's various tools.

For comparison, the sequential implementation took, on average, 780ms using two producer and consumer threads in order to process 9600 traffic data entries. The MPI implementation takes on average 600ms using 2 processes each with 2 producer and consumer threads. While this was ran using processes on the same machine, it is clear to see the performance benefit this would have scaling to multiple machines with larger input files.