THE UNIVERSITY OF EDINBURGH

MESSAGE PASSING PROGRAMMING

COURSE ASSIGNMENT

Student Exam Number: B138641

I Problem Statement

The aim of this assignment is to implement a parallel program that performs a form of image processing using the Message Passing Interface (MPI).

More specifically, given a file that corresponds to the output of an edge detection algorithm, the program should be able to reconstruct the original image used to create that edge file. This can be achieved using a formula that is able to calculate the previous value of a pixel - previous meaning before applying the edge detection algorithm - based on its 4 neighbours over a number of iterations. The aforementioned formula that calculates a pixel's previous value is as follows:

$$new_{i,j} = \frac{1}{4} \cdot (old_{i-1,j} + old_{i+1,j} + old_{i,j-1} + old_{i,j+1} - edge_{i,j})$$

Where:

- $edge_{i,j}$: the pixel of the edge detection algorithm's output
- $old_{i,j}$ and $new_{i,j}$: the pixel values at the beginning and end of each iteration

II Implementation - General

II.I Logistics

Details regarding the development of the implementation as well as the libraries used are mentioned below.

- Programming language used: C
- Compiler used: GCC 4.8.5 20150623 (Red Hat 4.8.5-28)
- MPI version: Open MPI 3.0.0

II.II Approach

The way this problem is approached is by using two-dimensional domain decomposition for the parallelization of the computational process as well as non-blocking communications for halo-swapping among processes.

The reason non-blocking communications are preferred over blocking ones is because an implementation using non-blocking communications minimises the program's waiting time, which results to the communications taking place in a faster and prompter manner. Although this assignment requires communications to be implemented in a non-blocking manner, in order to support the aforementioned argument, experiments were carried out based on both a blocking communications implementation and a non-blocking one.

III Implementation - Detailed Structure

III.I Program Logic

A brief outline of the program's logical steps is given below. Each part is analysed further in the subsections that follow.

- Read image file
- Distribute the image across processes
- Reconstruct original image
- Collect results
- Export output file

III.I.I Image I/O

An image containing edge values is given as an argument to the program's executable file. The image file is loaded in memory using the `pgmread` method provided, while the dimensions of the image file are retrieved using the `pgmsize` method.

Image Decomposition

We are interested in decomposing the image and distributing it to processes based on the number of processes specified. The number of processes is easily determined through the `MPI_Comm_size` routine. What we aim to do now is gradually create a two dimensional virtual topology, which will be mapped to our image. At first, we have to consider the number of processes on each dimension. A simple and effective approach is using the interface's `MPI_Dims_create` routine, which returns the suggested number of processes per dimension. We call these `nprocs_row` and `nprocs_col`.

We now know the `width`, `height` and numbers of processes for both dimensions, therefore the number of pixels each process will handle will be `pwidth=width/nprocs_row` on the row axis and on the column axis `pheight=height/nprocs_col` . At this point all four values (`width`, `height`, `pwidth`, `pheight`) are sent to all processes using the `MPI_Bcast` routine.

III.I.II Process Communication

Virtual Topology

Creating a two dimensional cartesian virtual topology will provide a naming scheme which is well suited for our communication pattern. The topology is created with the `MPI_Cart_create` function. We specify through the passed parameters our topology to be non-periodic on the `height` side of the image and periodic on the `width` side.

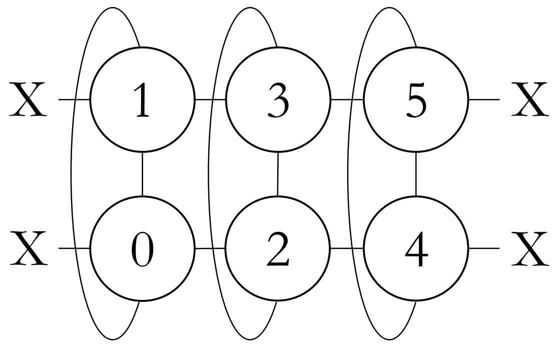

Figure 1: Virtual topology graphic

To better understand the communication pattern we consider Figure 1. We have 6 processes (0 to 5) within our newly specified topology - "X" marks represent `MPI_PROC_NULL`. Take, for instance, process 4. Its neighbour (as seen in the figure!) to the left is process 2, above is process 5, to the right a null defined process and below process 5 again. This is how periodic boundary conditions are achieved. Note that `MPI_PROC_NULL`s (black holes) are used to model the fixed values on our column dimension, so that any attempt of communication from processes on those two sides (0,1 and 4,5) is completed immediately, thus preventing us from deadlocks.

Derived Data Types

Another key concept used in this implementation are derived data types. More specifically, due to the fact that in our arrays column elements are not contiguous in memory, we define a new type called `column_halo`, which is a type that corresponds to the elements of a whole column.

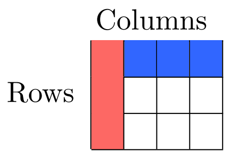

Table 1: Memory layout

As depicted in Table 1, blue elements are contiguous in memory, while red ones are not. Declaring a `column_halo` data type helps in our communications, since we can now send entire columns across processes without having any difficulty referencing the positions of their elements in memory.

Similarly, another custom data type is also used called `array_block` (See Table 2). This one is created to simplify the way of referencing an array block (sub-array) within the image. It is used when process 0 distributes and collects the sub-arrays to and from the other processes.

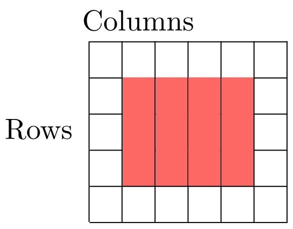

Table 2: Array block data type

Communications

Communication among processes in the new communicator (from the virtual topology) starts with process 0 distributing the corresponding part of the image to each process. This is done using the `MPI_Isend` routine. What happens is that process 0 issues a non-blocking send for each process (sending the `array_block` that corresponds to the process) and each process issues a `MPI_Irecv` along with a `MPI_Wait`. The reason a process has to wait is because it cannot proceed to the next part of the code without having first received the array data.

Once each process receives its corresponding sub-array it initialises values on its `old` buffer to 255.0 (white). Then, based on its position on the topology grid a process may need to modify some values. Processes on the edges (`height` sides) fave fixed boundary values, which are calculated based on a given function.

Note that the value that corresponds to each pixel needs to be referenced based on a global perspective. What this means is that each process processes elements that are mapped to a specific section of the whole image. A key concept here is to understand that the fixed values computed are referenced on a global scope, therefore the arguments passed in the `boundaryval` routine are not the indices of the pixel within the process, but the indices based on the element's position in the global image. For example, in a local scope, a row element is referenced using the `j` index, but in a global scope the index referencing that element is `j + rank_col_coordinate * pheight`.

III.I.III Image Reconstruction

Halo Swapping & Boundary Independent Pixels

The first thing that takes place in the iterative process are all necessary communications among processes, that is, halo swaps. The basis for this is that it allows us to issue non-blocking send and receives and while waiting for the communications to finish we are able to do other things - i.e. calculate halo independent values. In a way, we try to make use of the latency caused by process communications.

To clarify what is meant by halo independent values we can consider the representation in Table 3. We have two neighbouring processes and their local buffers (`old` - which contain 2 extra rows and 2 extra columns for halo values). The yellow section indicates the part in which the neighbouring process' boundary values will be stored (receives). The red section indicates the boundary pixels of the process (sends), and lastly, the blue section indicates the independent elements. That means, that for blue elements we can calculate the pixels' new values without the need of communication (green cross is within boundary). On the other hand, to calculate the new value of a boundary element we need at least one value that a neighbouring process possesses (black cross is outside of image boundary). That's the reason we need to perform halo swaps.

Therefore, while the halo swapping is taking place, each processes calculates the new values of its halo independent elements.

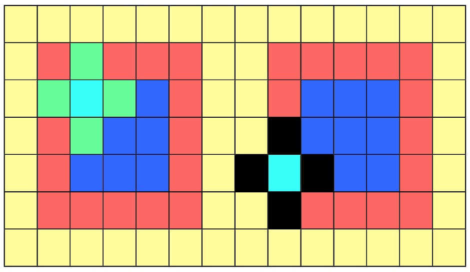

Table 3: Halo swapping between neighbouring processes

Boundary Pixel Calculation

As soon as all communications are completed (`MPI_Waitall`) we can start calculating the left-over values, that is, the elements in the red section. Once the whole array has been calculated, its values are passed to the `old` buffer and the process progresses to calculating the `new` values in the next iteration.

III.I.IV Stopping Criterion

The iterative calculation process is terminated either if the maximum pixel value difference between steps is smaller than a threshold (=0.1) or if the number of iterations reaches a predefined value called `MAX_ITERS`.

The difference between the new and the old pixel values of each process is calculated every 500 steps. Then, a `MPI_Allreduce` is called among all processes so that every process knows whether it has to break or not based on a global value (max of deltas).

III.I.V Average Pixel Value

Similar to the delta difference calculation, we calculate the average pixel value across the whole image. This is achieved by issuing a `MPI_Reduce` from process 0 which sums all other processes' values, and then divides by the total number of pixels so that it determines the mean pixel value. This value is afterwards printed as requested in the assignment guidelines.

III.I.VI Data Gathering

Once the iterative process has finished, process 0 has to gather all sub-array results from each of the other processes. This is implemented with non-blocking send and receives, as in the distribution process, but the other way around. Each process, except 0, issues a `MPI_Isend` and process 0 issues a corresponding `MPI_Irecv`. Then, it issues a `MPI_Waitall` so that it receives all sub-arrays before progressing to the next step.

III.I.VII Writing Output

Since process 0 has obtained all sub-arrays in its original master buffer, it can now copy its own sub-array there and write the outcome to a file. The result image is rendered using the provided `pgmwrite` routine.

III.II System Testing

The program developed was tested multiple times in order to ensure its correctness and efficiency. The run scenarios were:

- Local machine - 1 to 4 procs including all procs in-between
- On student.compute - 1 to 16 procs including all procs in-between
- On cirrus (front-end) - 1 to 72 procs including all procs. in-between
- On cirrus (back-end) - 1 to 144 procs including all procs in-between

For each of the test cases all provided files were given as input. The output was consistent on every case (given a specific input the program produces always the same result). This was checked through the mean pixel values.

III.III Performance Evaluation

The implementation's performance is calculated using time measurements. The runtime of each of the following sections of the program is measured and printed in a readable format:

- Initial I/O
- Total number of iterations
- Each iteration (used for the calculation of the mean iteration run time)
- Output I/O
- Total runtime

III.III.I Initial I/O

In Figure 2 we can observe how the initial I/O process is almost constant over the number of processes used. This visualisation also explains the small increase we get in total runtime after almost 34 processes.

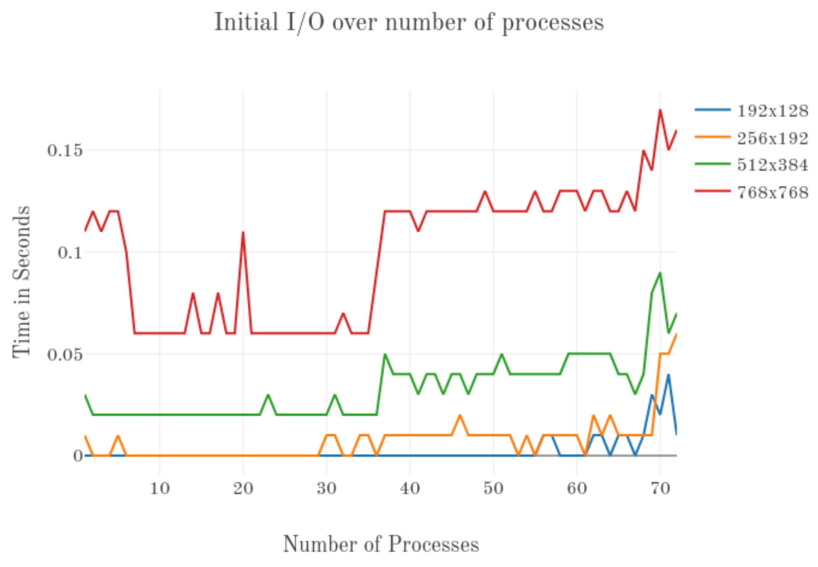

Figure 2: Initial I/O runtime

III.III.II Single Iteration Mean Runtime

The average time spent in a single iteration among all processes is calculated and depicted over the number of processes in Figure 3.

As wee can see, the mean iteration runtime drops significantly while the number of processes increases. This makes sense, since the more the processes we use the smaller the sub-arrays that will be allocated to them in order to compute the original image.

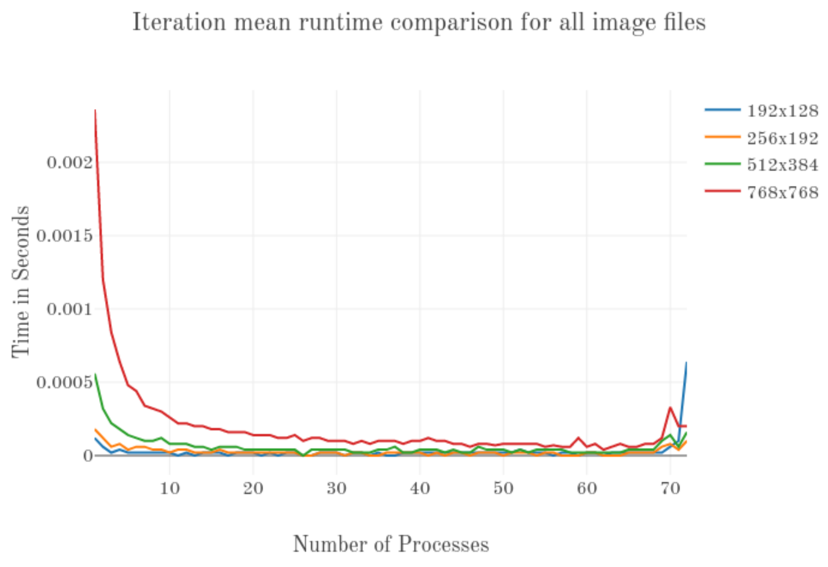

Figure 3: Single iteration mean runtime

III.III.III Iterations Total Runtime

Figure 4 shows the time spent calculating all cell values in the whole iteration process over the number of processes.

In a similar fashion, the total runtime of the whole iterative process decreases significantly as the number of processes becomes bigger.

III.III.IV Program Total Runtime

The total runtime of the program is also measured on each execution. Figure 5 displays total runtimes for each file over the number of processes used. After a certain point (about 34 processes) our runtime becomes steady and gradually may increase a little bit. This is due to the initial I/O, which is depicted in Figure 2.

The reason behind this is that having so many processes results to a bigger and bigger amount of messages exchanged. So based on the figures

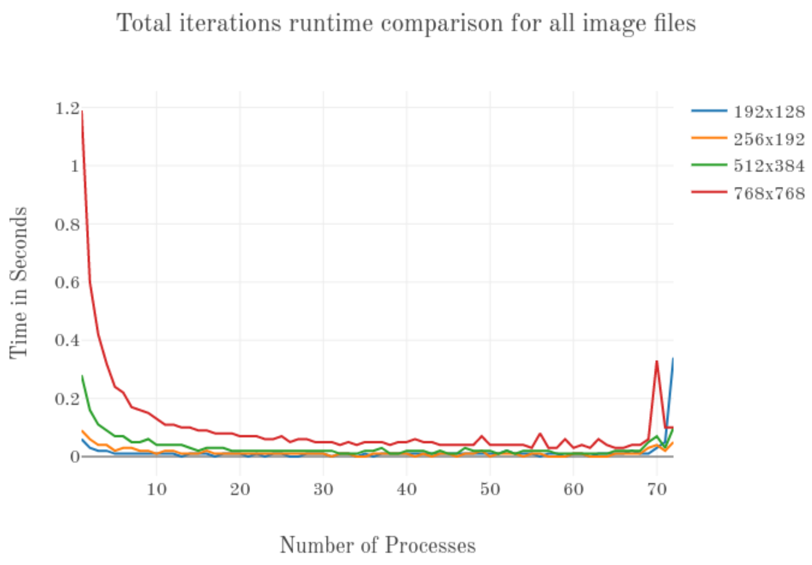

Figure 4: All iterations total runtime

we can observe a dependency between the number of processes used and the I/O runtime.

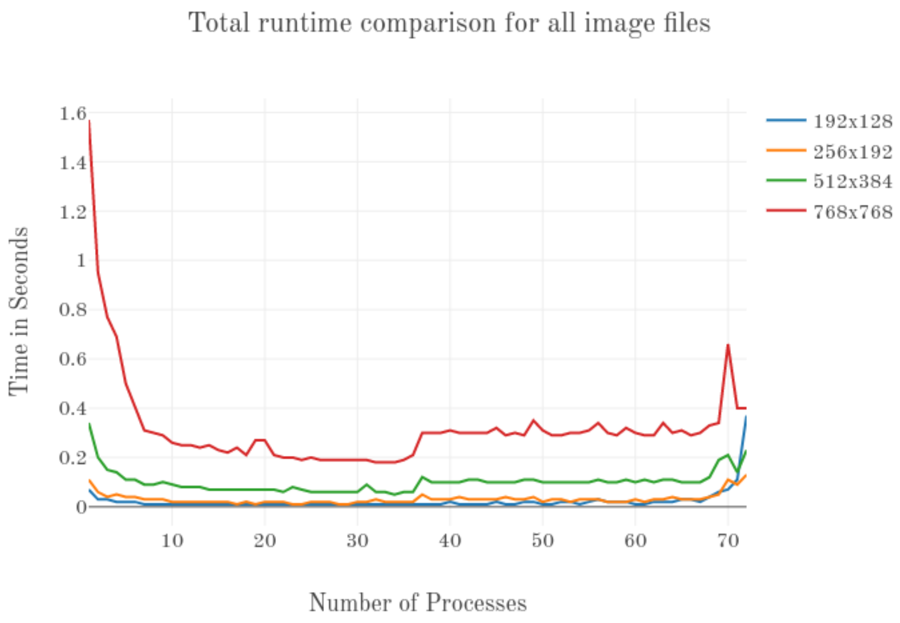

Figure 5: Total runtime measurement

III.III.V Blocking vs Non-Blocking Communication

Another idea that came up when running tasks to measure performance was comparing the performance of the implementation using blocking communications vs. non-blocking.

The implementation was branched and modified so that all communications were blocking. The performance tests were run using only the big image file (768x768). As expected, in Figure 6 we see that the non-blocking communication implementation achieves better runtimes than the blocking one.

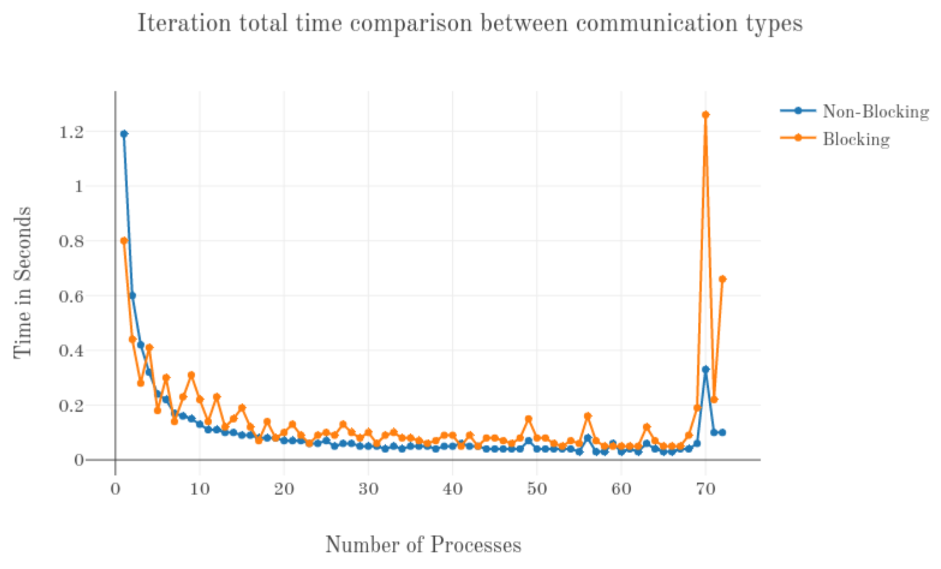

Figure 6: Communication type performance comparison

III.IV Conclusion

To conclude, we described the whole process of implementing a parallel program to reconstruct the original images based on the given edge files using the Message Passing Interface and then measured the performance of this implementation by running several experiments. We also tested the correctness of the program by comparing output results (diff).

The most interesting conclusions drawn were:

- Runtime drops drastically as the number of processes increases.
- The I/O is almost constant. Performance mostly depends on the communications and the calculations taking place is each process.

- An implementation using non-blocking communications achieves better runtimes than an implementation that uses blocking communications.

III.V Notes & Future Improvements

In this section I mentions various points concerning my implementation and discuss other areas in which there is room for improvement.

- The implementation is developed in a way that the only things needed to be specified when executing are the number of processes (through the terminal), the path leading to the input file (argument in the execution file) and the name of the output file. The dimensions are loaded using functions and are not hard-coded.
- All requirements mentioned in the assignment description are met (derived data types, virtual topology, non-blocking communications, fixed and periodic boundary conditions, termination conditions, experiments to show performance improves when number of processes is increased).
- There is a more elegant way of implementing the I/O, by which each process can read directly its corresponding sub-array from the file (parallel I/O). It is something I tried implementing, but didn't complete due to time pressure.
- I was a bit hesitant when trying to explain the slight increase in runtime when running with 72 and 144 processes. My understanding is that it's a critical point after which parallel communications become an overkill for such images sizes. The reason Moreover, I am not that confident because the experiments for more than 72 processes were conducted only in cirrus' back-end, so I am unsure whether this could also be due to hardware change.
- I tried to keep the report short since the limit was 10 pages, so I am not that sure if I included everything expected. I would be more than happy to further discuss any part of the implementation if needed.