Message Passing Programming Coursework Assignment

Exam number B136013

November 27, 2018

Contents

1 Introduction

The goal is to solve an image processing problem. It uses a two-dimensional domain decomposition in order to split the workload to the active processes. To achieve this we use MPI communication protocol for process communication. This approach arises a variety of challenges that need to be addressed, such as communication and decomposition.

2 Project Description

The project takes as input a file that can be reconstructed to a greyscale image using a simple edge-detection algorithm. The mathematical model to reconstruct the image is

$$edge_{i,j} = image_{i-1,j} + image_{i+1,j} + image_{i,j-1} + image_{i,j+1} - 4 * image_{i,j} \quad (1)$$

There are a variety of project requirements in order to produce a correct output. There are fixed “sawtooth” boundary conditions in the horizontal direction. In addition, there are periodic boundary conditions in the vertical direction. This means that when a top process performs halo swap to fill the upper edge of the local table it receives it from the according to bottom process.
Another specification is the terminate condition. The main loop of image reconstruction should finish when the maximum difference of a pixel in the image between the old and the value it's insignificant. This means that after some iterations when the produced image has not drastic variations from the previous the loop should be terminated.

3 Design Analysis

The design and control flow of the project is basically the same as the case study with a different implementation. The implementation of the project uses mainly the basic functions of the MPI API. Some of these features perform non-blocking communication, create a virtual topology and use derived types.

3.1 Initialization

In the initialization phase, MPI_Init, MPI_Comm_rank, and MPI_Comm_size functions are called to establish the communication for the active processes and for each process to learn its place in this network. Then the program does a series of tasks

1. Reads the input image
2. Decomposes of the problem
3. Creates the virtual topology
4. Discovers the topology
5. Creates the Derived Types
6. Allocates and Initializes the buffers

Input The input of this experiment is an edge image and the number of processes that are required to solve the problem. Every process reads the size of the input image but only the master process reads the raw data and stores it to its masterbuffer.

Decomposition In general, the algorithm has been designed to deal with any number of processes even for the occasion that they are not exactly divisible by the matrix size. The approach that we chose is that the border processes are going to deal with the extra workload. Decomposition process produces the size of the problem each process has to solve.

Virtual Topology In terms of the produced virtual topology, the main function was MPI_Cart_create that creates the new 2 dimension topology. Reorder of the given processes to the new topology is permitted for optimization reasons.

Discover Topology In this phase, it is necessary for each process to discover its place in the virtual topology and a new data structure called Cart_Info. Cart_Info contains all of the information a process needs to know when it comes to communication without the extra overhead of calling MPI functions to obtain this information on repeat. This information are the

- World rank
- MPI_Comm of the virtual topology
- World size
- Its coordinates
- Virtual topology's dimensions
- The world ranks of its neighbors

In addition, methods like MPI_Cart_coords and MPI_Cart_rank have been used to identify the neighbor's ranks or coordinates. We have to say at this point that if a neighbor does not exist it is set a MPI_PROC_NULL which helps as to avoid condition statements in the send functions.

Derived Types In order to reduce the code volume and avoid unnecessary memory allocations derived types are used extensively. Derived types such as row, column, and table are declared once in the main function. MPI_Type_vector has been the function that creates these derived types. Derived Types are used in the communication phase like the non-blocking functions. Their main goal is to avoid memory copies for the send and receive buffer. What we managed to is to read and write directly from and to the target buffers. As we mentioned before our Decomposition may assign different problem sizes to the process. In order to keep up the code in a generic form, different implementations for the derived types have been the tool to address this issue.

Buffers Once we gathered this information we are able to dynamically allocate and initialize our buffers. All of the processes allocate the necessary buffers for the calculations using the sizes that the decomposition phase has produced. After that, the old buffer is initialized with the white (255) value. The boundary conditions are applied. To be more specific, if the worker has been assigned a part of the image which belongs to the left or right side, then sawtooth values are computed and stored.

3.2 Scatter

At this point the data exchange takes place. The master scatters the image which is stored in the masterbuffer directly to the edge buffers of the workers. Like in all of the communication processes MPI_Isend and MPI_Irecv functions are used accordingly. More of the communication phase will be explained in the following section.

3.3 Calculation

At this point, the main loop is ready to start. The calculation phase is decomposed as followed

1. Halo swaps are sent to the neighbors
2. The middle calculations are computed (excluding borders)
3. The program waits to the halo swaps and then calculate the borders
4. Every 100 iteration, the average pixel is logged and the program checks if the loop can be terminated
5. The new buffer is overwritten to the old one
6. Step 1 is executed again

Periodic Checks As mentioned before, at specific intervals in the loop the algorithm does some extra work. This work includes the logging of the average pixel's value. A

Tab Separated Value (TSV) file is generated to store this information. In addition, a function is called to check if the execution can stop, which is the only way to exit the loop. This method checks the maximum difference between the pixels and if this value is below 0.1 for each of them the process is stopped. We have to point out that intervals have been set to 100 loop iterations as a design decision. In extension, MPI_Allreduce function has been used twice in the periodic checks. First, to calculate the average pixel value which is part of the Output. Second, to calculate the max difference of the pixels.

Communication The cornerstone of the communication process are the MPI_Isend and MPI_Irecv functions. These methods are used for the halo swaps which are necessary for the calculations and the custom Scatter and Gather mechanisms. At the end of each of these procedures, the program calls the MPI_Wait function to ensure that there are not ongoing communications so the algorithm can resume its execution.

3.4 Gather

Using the same tools as Scatter, Gather process is implemented, only in the opposite direction. The master gathers all of the old buffers and reconstructs the masterbuffer which will be written to the new output image.

3.5 Finalization

The main job is done now the master process has gathered the calculated results. It will create the necessary files and store the logged information. In detail, the outputs are

- A new reconstructed image
- A TSV file that contains the input file, number of processes and average iteration running time (ms)

In the end, each process will free the allocated buffers and call MPI_Finalize function to end properly the communication between the processes.

4 Tools

This project is developed in C programming language due to its performance in low-level calculations. We used mpicc compiler with -O3 flag for serial optimization. In addition, GNU Make was selected for the build phase and Python as the scripting language to compare the output for testing reasons. Cirrus supercomputer is the platform that all of the experiments have been executed. In order to submit the job to the backend of Cirrus a Portable Batch System (PBS) has been created. In case you want to rerun all of the experiments that we used for the evaluation, run

```
$ ./run.sh
```

This file is a script to

1. Create the required folders
2. Make the project targets using GNU Makefile based on the experiment option
3. Define the arguments for the PBS script (nodes, input, processes, target)
4. Submit the jobs to Cirrus

There are several experiment options that are enabled using the appropriate definitions in the C preprocessor through the Makefile. The experiments are investigating

1. The average pixel for each iteration
2. The average iteration time
3. The average iteration time for big input files
4. The average iteration time with terminate condition calls
5. The production executable correctness

For each of the above experiments, a different executable has been created with minor configuration through C preprocessor define commands. These configurations include

- MAXITER the maximum iterations number
- PRINTFREQ the frequency of print pixel and terminate condition checks
- AVERAGE_PIXEL_FOLDER the folder to store average pixel results
- TIME_RESULTS the file to store the timing results

All of the data results will be saved in the data/ folder in different files. Once the submitted jobs are completed, a Python script is responsible to analyze the data and create the graphs. To run the script execute `python data_analyzer.py`. In addition, we have created our edge files, to reproduce that run `./converter.sh` in converter/ folder.

In the following section, we evaluate these results.

5 Evaluation

Evaluation has been done in order to find out if the program behaves as it should in terms of correctness and performance.

5.1 Correctness

First and foremost, before performance analysis, we have to ensure that the produced outputs are valid regardless of the number of processes that are used underneath. The approach is very simple. Once the job has been successfully executed till completion (using production mode imagenew executable), we run the serial program for all of the given input images and then store the outputs to the test_output/ folder. The serial program has been modified to stop when there are no significant changes in the image's pixels between each step. To be more specific, the max different has been defined to be 0.1. It worths to mention that the termination condition has to remain the same in all of the executions because different conditions create different results. This comparison is made automatically through a Python script that uses the filecmp function. If the files are the same then our experiment is correct.

To check if your produced images are correct run

```
$ python test.py
```

The test will print to the standard output if each image produced in the output/ folder is exactly the same as its pair in the test_output/ folder.

5.2 Performance

Performance analysis is essential for this experiment in order to compare our results and extract useful information about them.

5.2.1 Speedup

For the timing of the experiment, we have executed the project for 1500 iterations without the terminate condition and average pixel prints. Of course, this method will not produce the correct output images, but this is not a problem in this part of the performance analysis. We just need to take a representative sample of calculation loop timing without wasting CPU time. The logged time for each experiment is the mean iteration running time. Using this measurement we present the speedup of the experiments. This a very useful experiment because we can decide how good our input scales for a variety of process. This would help us decide which is the optimal number of processes to run if our budget or resources are limited.

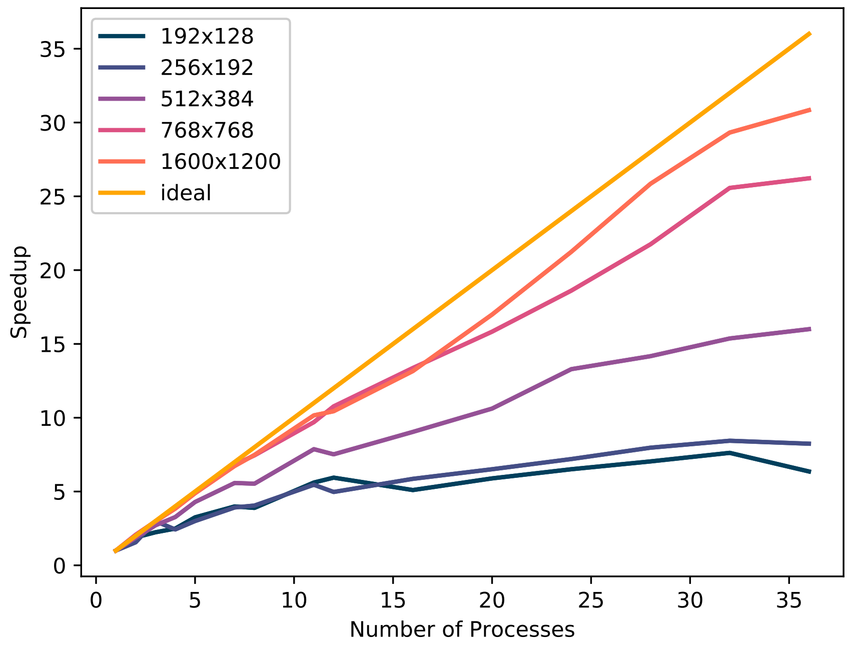


Figure 1: Speedup for all the input images

Small Input Running on 1 node from 1 to 36 cores speedup is increasing or stays constant. Let's take for example the case where the input image is 192x128 or 256x192 pixels. In Figure 1, it plain to see that the problem does not scale as it should. When we use even more processors the speedup is slightly inclined. This possibly because the arithmetic calculations consume a little amount of time as opposed to the halo swaps. As a result, for the specific size of problems 10 till 15 processes would be the best option. We don't have to waste 10 or 20 more cores if the speedup is not going to exceed 5 and 6 accordingly.

Medium Input As long as we increase the input image size the speedup is growing faster. From Figure 1 we can denote that for the image 512x384 choosing 32 processes seems a very appealing option. Reaching speedup 15 times faster than running it on one process.

Big Input In case of input size bigger than 768x768 pixels, running on 1 node the results are promising. Speedup is increased almost in a linear fashion, reaching the ideal values and making the communication overhead of the halo swaps to be insignificant. So we decided to do a further investigation and run the experiment for the big images on more nodes. In Figure 2 we see the speedup results running on 4 nodes from 40 to 136 number of processes for the big input files.

As a result, the speedup continues to increase. The images are big enough to be divided into more than 40 processes so the speedup keeps increasing as long as we increase the number of processes without negative effects in the performance.

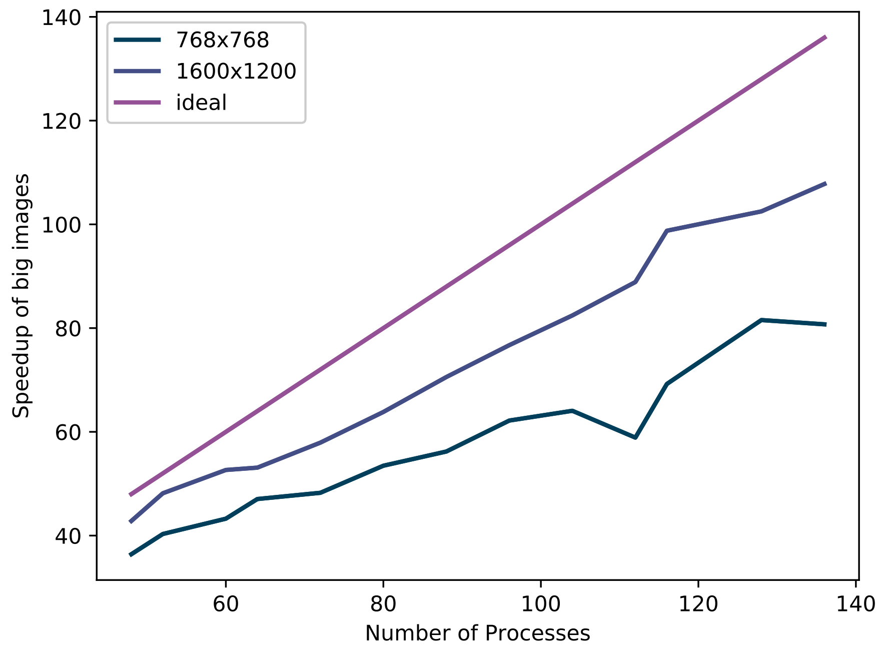


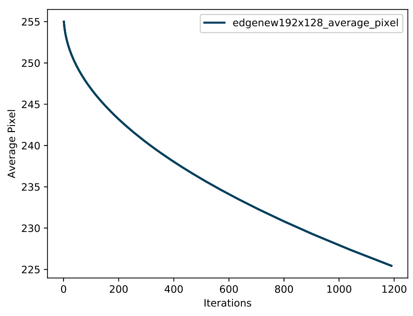
Figure 2: Speedup for big input images

5.2.2 Average Pixel

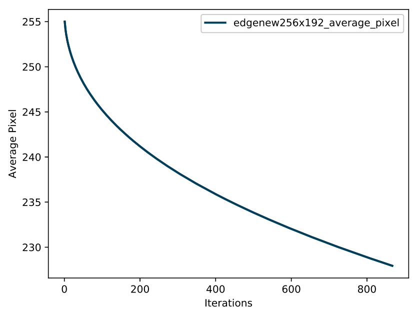
In this section, we have performed an analysis regarding the average pixel results. In each iteration, the average pixel is calculated. In order to achieve that we had to run the experiment till completion for all of the input images. We chose to execute it just once on 16 processes. Obviously, we did not have to run it again on a different processors configuration. This occurs due to the fact that the average pixel is depending on the input image (size, edges) and not in how many processes are participation in solving the problem. The specific experiment has been run using the imagenew_average_pixel target.

According to Figure 2, which includes the average pixel values of 4 input images against the iteration number we can make some useful observations. First of all, the number of iterations is on some occasions independent of the size of the input image. This is because the number of iterations until we can terminate the process is depending on how much easy is to reconstruct the specific image. For example, we can see doing a comparison in Figure 2 between (a) and (b) that image 192x128 is indeed smaller than 256x192 but it takes 300 more iterations to be terminated.

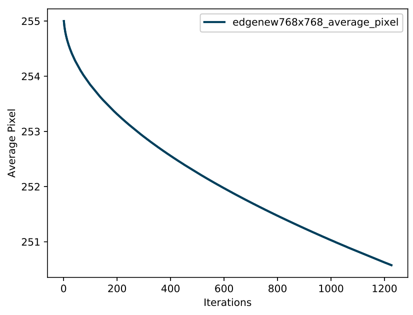
Secondly, it is plain to see that the average pixel's value is declined in every occasion.



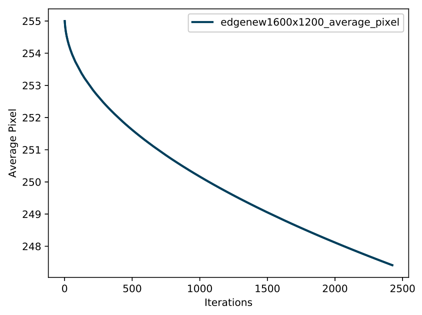
(a) Average pixel for image 192x128



(b) Average pixel for image 256x192



(c) Average pixel for image 768x768



(d) Average pixel for image 1600x1200

Figure 3: Average pixel for input images

That observation is logical due to the fact that the initialized old buffer contain white (255) pixels. As long as the loop is processed the average value is decreasing, meaningly the whole picture is becoming darker. At this point, it worths to mention that in each image for the first iterations the pixel value is declined very fast. After that, the change rate is slower, meaning that each iteration has a small impact on the newly calculated output.

5.2.3 Intervals

We made the design decision to print the average pixel and check the terminate condition every 100 iterations. That decision has been made in order to reduce the execution time. We made an experiment running for 1500 iterations the image 768x768 with and without intervals on 1 to 16 processors. The imagenew_timing_intervals executable has been created for this purpose.

According to Figure 4, we can denote that adding the intervals reduces drastically the speedup of the experiment. For example, running on 16 processes we could improve the speedup from 6 to 13.

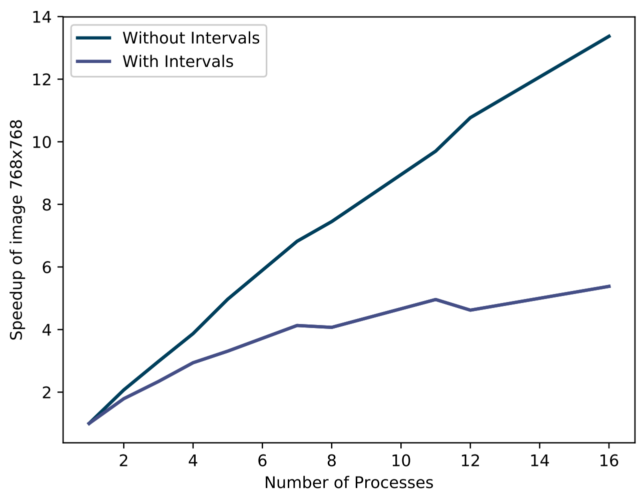


Figure 4: Speedup for all the input images

6 Conclusion

In conclusion, depending on the size of the given problem the answer for the best approach to solve it always varies. We could easily say that as long as the problem size is increased we have to do the same with the resources, in order to achieve the minimum running time. But the truth is that if we want to exploit the best out of our system, further investigation is required. Luckily, this doesn't mean that we have to run our experiment until completion to identify the best approach. A small number of iterations of the main calculation loop are enough to identify the best suited-system configurations. This choice is based on the experiment results, but always taking into account our available resources and budget, which is energy consumption and as always the given time.