University of Edinburgh

MSc High Performance Computing with Data Science

Message-passing Programming

Coursework Report

Author:
B139964

November 30, 2018

Contents

List of Figures

1 Introduction

This work implements an image reconstruction algorithm using Message-passing Interface (MPI). An image in a form of a two dimensional grid is reconstructed given a file that represents the output from an edge-detection algorithm. Previously, a complete MPI parallel program that performs the image reconstruction algorithm was implemented using one-dimensional process grid. Here, this work is extended implementing an MPI parallel program for a two-dimensional lattice-based calculation that employs a two-dimensional domain decomposition and uses non-blocking communications.

Given the edge data file, (1) can be applied in order to reconstruct a grey-scale image of size M×N. Note that *old* and *new* are the image values at the beginning and end of an iteration.

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

Consequently, a pixel in the image is reconstructed based on the values of its four surrounding neighbours. As a result, starting from a plain white image and applying the above operation iteratively, will result in the initial image, before the edges were obtained. The above operation will constitute the heart of the calculations discussed in the later sections.

1.1 Specifications

The main focus of the work is the implementation of the image processing problem using a two-dimensional domain decomposition and non-blocking communication for halo-swapping. Further to that, the implementation should feature fixed boundaries with saw-tooth values and periodic on the horizontal and vertical directions respectively. In addition, during the execution of the program, the average pixels of the reconstructed image should be calculated and printed out at the appropriate intervals. The algorithm should be able to terminate once the reconstructed image has reached a sufficient accuracy.

1.2 Aims and Objectives

Writing a parallel program is not easy. What is more difficult though is to ensure about its correctness. The main aims and objectives of this work are listed below:

- Implement the parallel code and test that it works correctly.
- Determine heuristics and justifications for choosing an appropriate frequency where the early stop criterion is checked.
- Design and perform tests that investigate the performance of the parallel program.
- Demonstrate that the performance of the code improves as the number of processes is increased using appropriate metrics.
- Investigate the effects in performance as the problem size is varied.
- Investigate the effects in performance from overlapping communications with calculations.
- Investigate the effects in performance from executing the parallel program in multiple nodes of a network.

1.3 Structure of the Report

The report is separated into four major sections. Firstly, in Section 2, the implemented version of the image reconstruction algorithm using MPI is presented and discussed, providing insights about the design choices and considerations. As we progress in Section 3, the performed experiments are discussed. The discussion contains information about how each experiment was set up and performed. The results of each experiment are also presented and explained providing reasoning about them. Finally, in Section 4, the report is summarised and possible future work is suggested.

2 Implementation

The implemented program that performs the image reconstruction is written in *C*. The main body of the program is in `pimage.c` while message-passing functions are isolated into a separate library called `mplib.c`. The I/O interface and the memory allocation routines are located in the `pgmio.c` and `arralloc.c` files respectively. The later two files were provided for the particular work. All the rest functions used in the implementation of the code are located in `imagelib.c`. Moreover, all the defined structures that were designed in order to create a more readable and easy to use code are located inside `structures.h`. A serial version of the code that was used throughout the development of the MPI parallel code can be found at `simage.c`. Note that different versions of the parallel code can be compiled using the appropriate `-D` options. The current available options are `-DEARLY_STOP`, `-DOVERLAP` and `-DTIME` that allows for the code to stop based on the early stopping criterion, the communication and computation to overlap and the main loop to be timed respectively.

2.1 Overview

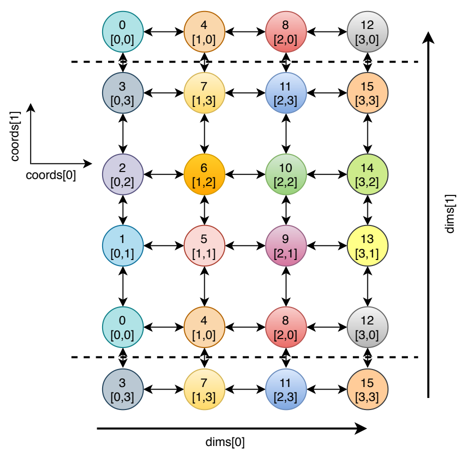
In order to familiarise the reader with the implemented code, a high-level overview about its structure is provided here. Insights about the design choices and considerations are provided in the following sections. The code is organised as follows:

- Message-passing system starts. The communicator is created in order to allow processes to communicate with each other.
- Processes are organised in a virtual grid by setting up a Cartesian topology. The topology sets horizontal and vertical boundaries to fixed and periodic respectively.
- User-inputs to the program in the form of command-line arguments are read. The inputs contain mainly information about the input and output files.
- The size of the image is read by the master process and broadcasted to all the processes in order to allocate their buffers. There are two I/O buffers and three buffers responsible for processing the image. One master buffer is allocated along with a smaller buffer that will receive the sliced image to be processed in each process. The three processing buffers are responsible one to hold the *edges* and the other two for holding the *old* and *new* image values at the beginning and end of an iteration.
- The master process reads the edges file and the image is then scattered to the local smaller buffers of the rest of the processes.
- Then the processing buffers are initialised. The *edges* buffer reads the contents of the local buffer while the *new* and *old* buffers are initialised to contain a white image. Note that the three processing buffers include halo dimensions. It is required here to initialised the horizontal boundaries using saw-tooth values.
- As now everything is set up, the main loop responsible for implementing the image processing algorithm starts. The stopping criterion of the loop can be either a pre-defined number of iterations or an early stopping criterion based on the accuracy of the image.
- Inside the loop, first the halos of each process are swapped, both vertical and horizontal, in order to obtain the necessary data for the update step. Once the halos are received, the update step is performed as shown in (1).
- Following the update step, the maximum change in the pixels as well as the average value of the pixels in the grid is determined using local and global reductions.
- The final step of the main loop is to set the *old* buffer equal to the *new*. This is performed by swapping their pointers.
- After the end of the main loop, the *old* buffer is copied back to the local buffer and the image slices are gathered back to the master buffer of the master process. Then, the reconstructed image is outputted in a PGM file.
- As a final step, the buffers are de-allocated and the message-passing system is terminated.

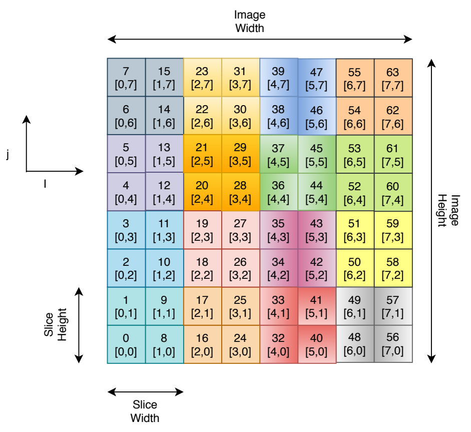
2.2 Virtual Topology

For the processes to be able to communicate with each other and exchange information a communicator needs to be established. This is handled when the MPI is initialised where each process is assigned a unique identifier known as the *rank*. All the necessary information regarding this communicator, such as its name, rank and size is contained within a structure of type `comm_str`. However, as one of the requirements of this work is to perform a two-dimensional lattice-based calculation that employs a two-dimensional domain decomposition it is convenient to transform the communicator into a Cartesian topology. Performing this kind of transformation will simplify the code and also might allow MPI to optimise communications.

Creating a Cartesian topology, effectively produces a new communicator where each process is connected to its neighbours in a virtual grid. Specifying such a topology requires us to specify the number of dimensions `dims[]`, the periodicity of the boundaries `period[]`, if any re-ordering is desired for possible optimisations by setting `reorder` to true and the displacement `disp` between each process. In our case, as we are dealing with a two-dimensional problem, a two-dimensional topology is specified with fixed and periodic horizontal and vertical axes respectively. Each process is displaced by 1 from its neighbours and no re-ordering was selected. In order for each process to establish its neighbours, 4 additional variables need to be introduced in order to hold the rank of the neighbours. This information is held within a struct of type `dir_str` along with a *status* and *request* variables that will be used later on in the communication part. All the information regarding the Cartesian topology created is organised within a struct of type `cart_str`. The reason behind this choice is that this information will be needed in various part of the code later on and by using a struct will make the code more readable but also will give us some additional flexibility. Having



(a) 4x4 Cartesian Topology for 16 Processes



(b) Visualised Memory Layout for an 8x8 image

Figure 1: Selected orientation of the Cartesian topology to follow the memory layout of the buffer used to store the image. Here a 4x4 topology is considered for an image of size 8x8. Arbitrary values can be used insted without changing the orientation.

established the Cartesian topology, now each process can be identified by Cartesian coordinates where `coords[0]` and `coords[1]` represent the (x, y) coordinates respectively. Figure 1a shows the implemented topology when the number of processes is 16. While in general the number of processes can be set to an arbitrary value, for simplicity we stick to 16 in order to demonstrate how the topology was selected. The actual topology shown below is enclosed between the dashed lines. The 16 processes are now mapped into a 4x4 grid. We can see that the leftmost process at the bottom is the process zero specified by $[0, 0]$ while the rest follow the coordinate system. The arrows demonstrate how each process communicate with each other effectively showing which process is neighbouring to which. For the processes at the left and right boundaries notice that there are no neighbouring processes to the left and right respectively following the specification. On the other hand, the horizontal boundary is periodic as can be seen from the figure.

The reason behind this choice of the arrangement was mainly due to the memory layout.

When an image will be read and stored inside a two-dimensional array of size $Image\ Width * Image\ Height$, the layout of the memory will be similar to the one shown in Figure 1b. The figure shows how an 8x8 image will be eventually stored in memory. In the position indexed by $[0, 0]$ effectively the left-most bottom pixel of the image will be stored while in $[0, 7]$ the left-most top pixel. As data are contiguous in memory in the j dimension or `dims[1]`, effectively the selected topology follows the memory layout. In other words, when distributing the image slices to each process the thinking and operations on the data will be straight forward. Note that both figures are colour coded as we are trying to show how each slice will be eventually be assigned to each process. Consequently the light blue slice in Figure 1b containing the memory elements located in positions $0, 1, 8, 9$ will be assigned to process $[0, 0]$.

2.3 Slicing the Image

The next step after defining the Cartesian topology is to allocate the necessary buffers, read the edges file and distribute the data to each process for processing. The buffers are dynamically allocated using the `arralloc` routine provided. As mentioned in Section 2.1, the allocated buffers are: *masterbuf*, *buf*, *edges*, *old* and *new*. The first buffer is responsible to initially host the edges file and at the end of the calculations the reconstructed image while the second to host the sliced edges. The later buffers are used to perform the processing in order to reconstruct the image. From here on we assume that an image has size of WxH.

A single slice has shape of W_sxH_s given by

$$W_s = \frac{W}{dims[0]}, \quad H_s = \frac{H}{dims[1]} \tag{2}$$

where `dims[0]` and `dims[1]` is the size of the Cartesian topology in the horizontal and vertical dimension respectively. This however assumes that the dimensions of the imported image are exactly divisible with the dimensions of the topology. Therefore the size of *buf* is equal to W_sxH_s. The processing buffers have the same size size as *buf* but with halos added hence their size will be $(W_s + 2)$x$(H_s + 2)$.

Regarding the *masterbuf*, this was selected to have size of $W_s * Size$xH_s where size is the size of the communicator. Note that the size is not WxH as demonstrated in Figure 1b. The reason behind that is due to the way the slices are distributed to the processes. As only the master process reads the edges file, the distribution of the slices to the other threads is performed using the `scatter` operation and collected back using `gather`. The utilisation of these function in a two-dimensional domain decomposition is not straight forward therefore to be able to use them some modifications were required to be made. The first modification was indeed the size of the master buffer. The chosen size represents the same memory layout presented before but this time flattened. The flattened array is illustrated in Figure 2. Note that the slices are colour coded in the same way as in Section 2.2 showing how each slice was translated. In addition to the change in memory layout, the provided I/O routines were needed to be modified in order to perform the necessary translations and place the data in the right order.

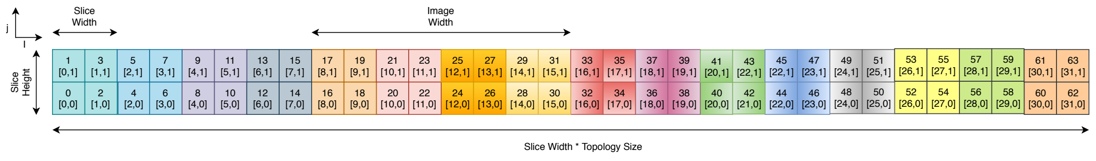


Figure 2: Flattened Memory Layout for an 8x8 image.

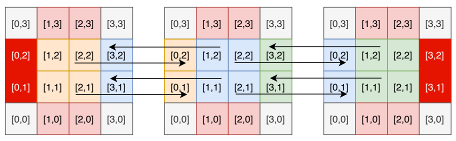
It is worth pointing out that the current implementation is also capable to run for even when the dimensions of the image are not exactly divisible with the dimensions of the topology. When this is the case, the dimensions of the slices are evaluated based on

$$W_s^a = floor\left(\frac{W}{dims[0]}\right), \quad H_s^a = floor\left(\frac{H}{dims[1]}\right) \tag{3}$$

where a stands for the actual dimensions of the slice. As the image will not be exactly divisible, the slices at the boundaries will have some remaining pixels that need to allocate space for. Therefore, in order to still be able to use the aforementioned distribution routines, all the slices need to be of the same size. To address this issue, zero-padding was used. In other words, all the slices with size smaller than W_s^pxH_s^p where p stands for the padded dimensions of the slides, they were set to that size introducing zeros to the additional memory locations. This approach comes at the expense of utilising extra memory. However, the I/O routine can take into consideration the zero-padding and still provide the right image to the master buffer. As now the buffer dimensions were set to the padded dimensions of the slice, one might thing that we need to deal with complex transformations throughout the code. However this is not the case. As the both the actual and padded dimensions are stored inside as structure of type `slc_str` responsible for holding all the information regarding the slices, we can actually use throughout the program the actual dimensions without any problem or complex thinking. In addition, we can completely ignore how the I/O and flattened memory layout are implemented and still visualise the problem as shown in Figure 1.

2.4 Non-Blocking Communication

Another important aspect of the program is how the processes actually communicate between them. Communication takes place inside the main loop where the reconstruction of the image is performed. As the algorithm requires values from the 4-neighbouring pixels of the *old* buffer then this is unavoidable. This is due to the values at the boundaries, where in order to be able to perform the update step they require data they do not have access to. This data can be obtained by communicating with the neighbouring processes to actually collect them. Therefore additional memory is added on each processing buffer, known as the halo dimension. This will provide space for the required values lying in neighbouring processes to be hosted in a methodical way. For the purposes of this work, we have chosen non-blocking communication. In other words, we use asynchronous send `Issend` and receive `IRecv` to perform the halo swapping. The exchange is illustrated in Figure 3 where 2x2 slices within a topology with dimensions of 3x3 are exchanging the appropriate data. On the left, in Figure 3a the vertical halos between three neighbouring


(a) Vertical halo exchange

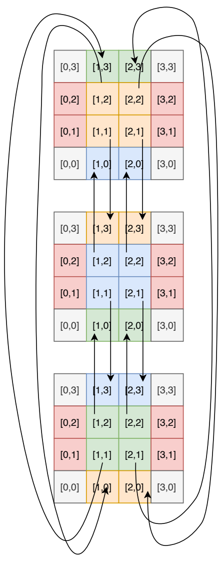

(b) Horizontal halo exchange

Figure 3: Halo exchange for a 2x2 slice.

processes within a 3x3 topology are exchanged. Note how the middle one sends and receives from both neighbours in the horizontal but the two at the boundaries can only send and receive only to the middle one. This is dye to the fixed boundary conditions in the vertical dimension. The same case is illustrated in Figure 3b where the horizontal halos are exchanged. Note the periodicity here. The light red elements represent halos that have not been updated from the swapping in that dimensions, while the bold red colour the boundaries to the vertical dimension.

Implementing the vertical halo swapping is straight forward as the elements within a halo are contiguous memory locations therefore can be send and received without any issues. This is not the case however for horizontal swapping. As a row needs to be sent, doing the exchange naively using the provided MPI datatypes, the wrong data will be transmitted as memory locations are not contiguous in the horizontal. To resolve this issue, a new MPI datatype needs to be derived. The datatype will be responsible to choose the appropriate data to be transferred during the communication. The best way to describe this is through an example. In Figure 3b, we will assume the process of interest is the middle one and wants to send data to the halo of the process below. In other words it only needs to send the [1,1] and [2,1]. Therefore, the derived datatype will be picked to be the one that filters 2 elements, one by one that are H_s^h memory spaces apart, where H_s^h represents the halo height, in that case equal to 3.

2.5 Updating the Image

Once the halo swapping is completed and all the necessary information is collected, the image can be updated. The *new* buffer therefore is updated according to (1). Each pixel in the slice needs to be updated apart from the halo values. Consequently this requires us to iterate through the actual dimensions of the buffer each time updating the buffer. The iteration space can be regarded as $i = [1, W_s^a + 1)$ and $j = [1, H_s^a + 1)$.

However, we can also take advantage of the non-blocking communications and overlap some communication with computations in order to hide some communication overhead. The only values that take place during the communication are the ones right below the halo. For those, the communication needs to be completed before reading them to avoid any race conditions. In other words, the elements within the iteration space $i = [2, W_s^a)$ and $j = [2, H_s^a)$ do not participate in the halo exchange and can updated once while the communication is still executed. The effect of this approach is examined later on.

Additionally, after each update iteration of the loop is completed, the *old* buffer is updated with the values of the *new* buffer. Instead of copying the values of each element in the array, the pointers of each array are swapped.

2.6 Early Stopping and Average Pixels

The last requirement of the implementation is that the program has to stop once the reconstructed image reaches to a certain accuracy. This can be achieved by monitoring how much the image changes at every iteration. In order to do so, the maximum absolute change Δ of the pixels needs to be computed by evaluating:

$$\Delta = \max_{i,j} |new_{i,j} - old_{i,j}| \tag{4}$$

The maximum absolute change is first determined in each slice locally by performing local reduction where max operation is used and then once all processes determined their Δ_l a global reduction is performed to determine Δ_g. Once Δ_g reaches to a value lower than a threshold, which is set to 0.1, the reconstruction algorithm terminates.

In a similar way, the average value of the pixels in the reconstructed image can be determined. First the local sum p_l of the pixels in each slice is accumulated using local reduction and sum operation. Then once all the local sums are ready, global reduction is performed to determine the global sum p_g of the pixels. After that, the average value of the pixel $\tilde{p}_g$ is evaluated by dividing with the size of the image. Note that here the contents of the *new* buffer were used to perform the sum operations. In addition, performing these operations at every iteration can result in some noticeable overheads, therefore for each a parameter is created to indicate every how often they are evaluated. However, the selection of the frequency needs to be determined experimentally.

3 Experiments

3.1 Verification

No matter how fast a parallel program is if the produced results are wrong then there is no use for it. In order to ensure the implemented parallel MPI program works as expected two different tests were implemented. Each of the tests compares the reconstructed image of the parallel program to the one of the serial version. Each comparison is performed over a range of number of processes. The number of processes varies and considers prime, even and odd numbers and numbers that are exactly divisible with the image and the ones not. Additionally it is tested up to 72 processes. The first test tests the outputs produced over 50 iterations without considering the early stop criterion. The second test takes into consideration the early stopping criterion and runs again over the same number of processes. Both tests were implemented using a bash script (`.sh`) and the comparison was performed using the `diff` instruction. After running the scripts, all the tests were successful. Note that the scripts can be run at the front end of `CIRRUS` as timing is not of interest here.

3.2 Setting Up the Performance Tests

In order to assess the performance of the implementation, a variety of performance tests were designed. Firstly an investigation was performed about how the frequency at which the early stopping criterion and the average pixels are calculated impacts the overall performance of the program and appropriate frequencies were selected. After that, the performance of the algorithm was evaluated in terms of how the increase in the number of processes affects the execution time and the scaling of the program. Here, different sized images were considered too. The next step was to re-run the previous test but this time overlapping communications with computations in order to investigate how that affects the performance relatively to non-overlapping communications. Finally, the effects of the network communication costs were considered where the program was run for a given number of processes each time varying the number of nodes it was running on. All the experiments were executed on the back-end of `CIRRUS` using the `mpicc` compiler. The compiler optimisations were set to `-O3` throughout the whole experiments. Note that different versions of the implementation were selected using the `-DEFINE` option and recompiling the code. As the I/O is of no particular interest in this work, throughout the experiments, execution time was measured by placing a barrier right above and below of the main loop of the program. Each test was performed 6 times and average times were recorded.

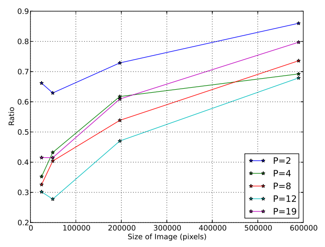
3.3 Determining Appropriate Frequencies

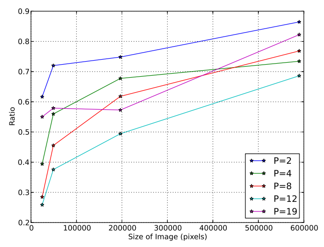
In order to determine appropriate frequency values for both the evaluation of the maximum absolute change Δ and the average value of the pixels in the image, two similar experiments were designed. First the program was executed for a fixed number of iterations set to 5000, without evaluating either Δ or average pixel value, and the average execution time t_{norm} for 6 repetitions of the test was recorded. After that, the same experiment was repeated this time also timing the evaluation of Δ at every iteration obtaining the t_{max}. In a similar way the time t_{avg} for the average pixel value was recorded. The overheads for each case were determined by subtracting t_{norm} from each of the individual times resulting to the overheads introduced by each evaluation. Each overhead is denoted by O_{max} and O_{avg} respectively. The experiment was repeated for multiple inputs and a range of number of processes in order to see how the two parameters will affect the choice of each frequency. In Figure 4 the ratio of each overhead over its corresponding t_{norm} is presented. Observing the figure, we can deduce that for the larger the image becomes the more overheads are introduced and they become almost identical to the execution time.

In addition when running on smaller number of threads the overheads are more significant as the amount of local reduction operations performed within each slice is much larger compared to when more processes are available. It is interesting to notice that for the case where $P = 19$ the overheads are much higher compared to the other processes, suggesting that the shape of the topology can also play a significant factor. After running the algorithm evaluating the early stop criterion rather than running for a fixed number of iterations, we can deduce by observing Figure 4c that a sensible choice for both frequency values is 200.

3.4 Performance Evaluation by Varying the Number of Processes and Image Size

Assessing the performance of the parallel MPI program a series of tests was constructed at which the program was executed over a range of number of processes and for different image sizes. The test was executed for 5000 iterations of the algorithm and was repeated 6 times out of which the


(a) Max Absolute Change Overheads


(b) Average Pixel Value Overheads

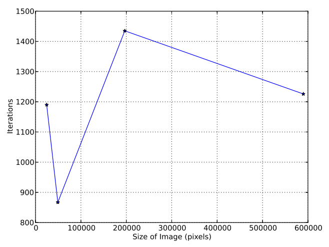
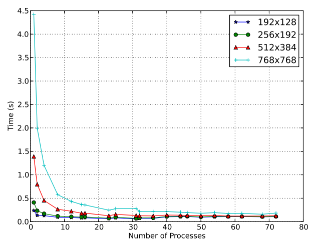
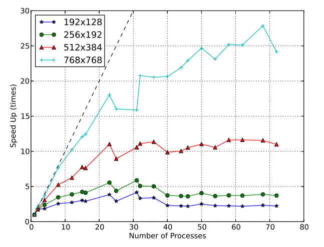

(c) Early Stop Criterion

Figure 4: Ratio of the overheads from evaluating the delta and average pixel value at every iteration to the execution time of the program with any of those evaluations performed. The test is performed over different sized inputs and over a range of number of processes.

average execution time was measured for each process number and each image size. The obtained results are shown in Figure 5.


(a) Execution Time


(b) Speed Up

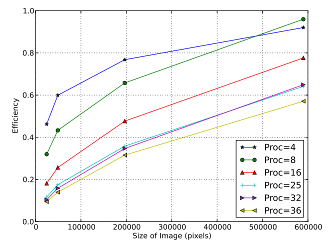
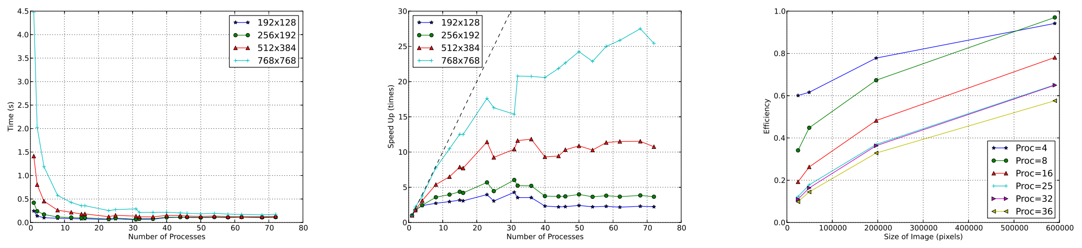

(c) Efficiency

Figure 5: Performance experiment where different sized images are reconstructed over a variety of number of processes. Each time the execution time, achieved speed up and efficiency are measured. Communication here is not overlapped with computations.

Observing Figure 5a we can see that as the number of processes increases, the execution time decays irrespective of the image size and converges around $t = 0.3s$. However, the larger the image is, the larger the decay rate suggesting that larger images scale better compared to smaller ones. This becomes more apparent in Figure 5b where the speed up T_1/T_p is plotted against the number of processes. The reason that larger images scale better compared to the smaller ones is because the overheads incurred form the communication between processes are somehow fixed. Therefore by increasing the image size, the computational cost is dominated by actually performing the update steps when reconstructing the image rather than the communication costs. As they all have to pay that cost, the effect however is more relevant to smaller images where the communication costs as the number of processes increases dominate. For that reason we observe speed up of smaller images to decrease eventually while for larger to keep increasing.

An interesting insight is given when evaluating the results from Figure 5c, where the parallel efficiency of the program is presented. Even though we have established that larger images will scale better compared to smaller one, here we see that speed up is increased more efficiently when the number of processes remains small. The reason behind that boils down again to communication overheads. As the number of processes is decreased, less communication needs to be performed between the processes therefore the overheads are reduced. This increase in efficiency is consistent to any size of images used. Considering the two observations just made, we can safely conclude to when we want to scale up the performance of our program efficiently, it is not only necessary to increase the number of processes but also the available work to them. In order to use them more efficiently, the image size also has to be increased.

Since the implemented program uses non-blocking communications this allows us to overlap communication with possible calculations in contrast to when blocking communications are used. For that reason, the same experiment was repeated but this time overlapping the two together. The update step of the algorithm was separated into the pixels that are not dependent on the communication and the ones that do. During the communication the former were updated and once the communication was finished, the later were updated as well as explained in Section 2.5. The results of the experiment are shown in Figure 6.



(a) Execution Time (b) Speed Up (c) Efficiency

Figure 6: Performance experiment where different sized images are reconstructed over a variety of number of processes. Each time the execution time, achieved speed up and efficiency are measured. Communication here is overlapped with computations.

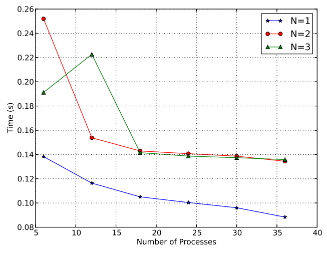
The obtained results are consistent to the observations made above. However, there is no noticeable boost in performance compared to when no overlapping communications were used. This would suggest therefore that the communications performed are short and by therefore overlapping computations with them will not impact the performance. Maybe this would be more helpful when the communications involved the exchange of larger chunks of data rather than just the halos as they would result in longer communication times. Additionally, such an approach might have been useful when the code was executed on more than one nodes where the overheads would be higher.

3.5 Performance when Running on Multiple Nodes of the Network

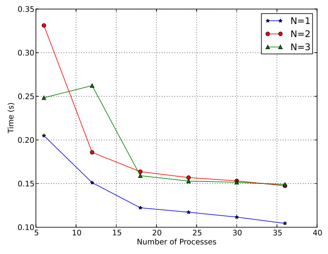
Following the findings above regarding the effect of the communication overheads to the performance of the program, a further experiment was motivated. The aim behind this experiment is to investigate the impact on performance when considering the overheads of communication through multiple nodes of the network. In order to do so the program was executed for a fixed 5000 iterations and for a number of processes. This time the processes were scattered across the network. In order to get comparable results we decided to run the code for `6, 12, 18, 24, 30` and `36` processes and scatter them around `1,2` and `3` nodes of the network every time. Note that in order to have a point of reference with our previous experiments, the number of processes was limit to 36, which is the maximum we can run on a single node. As test cases we used the usual 4 images that were used throughout the rest of the experiments. Each time the average time for performing each test 6 times was recorded. The results from the experiment are presented in Figure 7.

Observing Figures 7a and 7b that correspond to the smaller images, we can see that as we increase the number of nodes while keeping the number of processes the same the execution time increases. As discussed earlier, smaller images are more sensitive to communication overheads. Therefore by mapping the same amount of processes on more than one nodes effectively results in higher communication overheads as now they are charged for using the network as well. Consequently the performance becomes worse and plateaus as we increase the number of processes. Moving on to the larger images, we can observe that indeed are more robust to communication overheads. This is confirmed from Figures 7c and 7d where we see the execution time converges to the same values irrespective of the number of nodes. However, when comparing the converged execution time to the respective one obtained in Figure 5a, we can see from that for the same number of processes the program performs worse. This shows that even though large images are more robust to communication overheads, they are not completely immune. Therefore, the overheads incurred from using the network can affect the performance of your program and they should be acknowledged during the development.

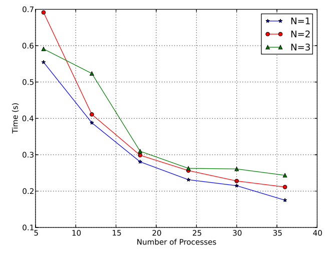
In Figure 8 the parallel efficiency achieved from running the code over a number of network nodes for different sized images is presented. The number of processes is varied. Note that same colour lines represent the same number of processes where lines with the same marker mean that the number of nodes is the same. Here again the results are consisted with what was discovered earlier. The increase in efficiency is consisted to any size of images used and the smaller the



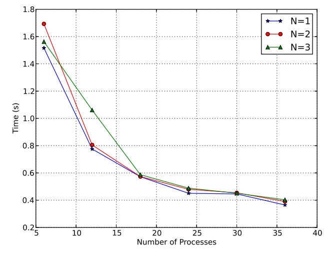
(a) Execution Time for a 192x128 image



(b) Execution Time for a 256x192 image



(c) Execution Time for a 512x384 image



(d) Execution Time for a 768x768 image

Figure 7: Execution times achieved from running the code for a variety of number on processes on more than one nodes to investigate the effects of the network overheads. The test was performed for different sized images.

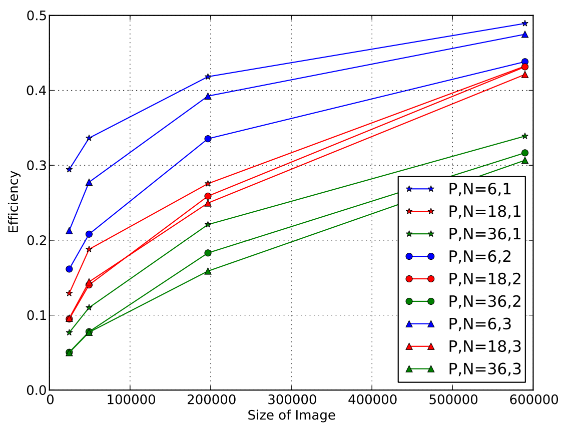


Figure 8: Parallel efficiency achieved from running the code for a variety of number on processes on more than one nodes to investigate the effects of the network overheads. The test was performed for different sized images.

number of processes is the better the efficiency becomes irrespective of the number of nodes. The result arises from the observation that blue lines in the figure -which denote the smallest number of processes presented here- are consistently above the red ones -second largest- and green ones. Additionally, comparing the lines marker-wise we can see that the lines with the start marker consistently result in better efficiency compared to the other markers for the same number of processes. As a consequence we can safely deduce that by increasing the number of nodes, the number of communication overheads is increased. Moreover, as it increases the efficiency of the program is reduced, regardless of the image size. Therefore we can say that the performance of the program will be highly dependent on the performance characteristics of the network.

4 Conclusion

In this work, we have presented how the implementation of a parallel MPI program for a two-dimensional lattice-based calculations that employs a two dimensional decomposition and uses non-blocking communications was carried out. Insights and justifications behind the design choices were provided in a high-level rather than going deep into referencing the implemented code. The verification tests implemented in order to ensure about the correctness of the program were also explained.

The key points from here can be summarised into the followings:

- The choice of the layout used by the Cartesian topology can simplify things if chosen carefully. On the other hand, exactly the opposite is true if chosen naively.
- Communications can be overlap with computations but if the duration of the communication isn't large enough no significant gains arise. In addition, when performing the halo swapping great care needs to be taken as derived data types might be needed. That was the case when the horizontal halos were exchanged.
- Naively evaluating the max absolute change in the pixels or the average pixel in the image can result into significant overheads. This should be handled with care and an appropriate calculation frequency should be used for each. As the operations are similar to each other, we have concluded that 200 is a good value to balance the losses from introducing overheads and the gains of actually stopping early the algorithm.
- Increasing the number of processes doesn't always mean better speed up or efficient scaling. Smaller images are more sensitive to communication overheads and eventually their performance saturates as the number of processes is increased since the overheads now dominates. For an efficient scaling, the number of resources should be increase in accordance to the size of the problem.
- The above point becomes more sensible when the processes are scattered on multiple nodes of the network as the communication overheads now become more significant and depend on the performance characteristics of the network. Here also the larger images seem to experience some effects from the overheads.

During the execution of the experiments, there were indications that the shape of the Cartesian topology also affects the performance of the program. For example for a prime number of processes the topology resulted in one-dimensional rather than two-dimensional and the performance seemed worse. Therefore as a future work, it would be interesting to perform tests targeting the effects in performance for different shapes of the topology.