# Message-Passing Programming Coursework

B066600

30/11/2018

## 1 Introduction

In this exercise, a message-passing parallel programme has been developed in order to implement a two-dimensional lattice-based calculation that employs a two-dimensional domain decomposition and uses non-blocking communications. The developed code was applied within the context of image reconstruction.

## 2 Design and Implementation

The program has the following structure:

- Reading in image
- Splitting image among processes
- Improving quality of each sub-image iteratively
- Joining images and writing to file

### 2.1 Reading in image

The first step is defining the size of the image (i.e. the number of pixels in the M and N dimensions) to be processed and number of iterations to be performed. The reading in of the actual image file is handled by the *pgmwrite* routine (provided within the complementary *pgmio.c* script), which stores the image pixels into a *masterbuf* array of size M*N. This routine is only called by the root process.

### 2.2 Splitting image among processes

Next, the available processes are arranged in a cartesian topology (see Figure 1). After having called the `MPI_Comm_rank` and `MPI_Comm_size` routines and stored the rank of each process and the total size of the communicator in appropriate variables, `MPI_Dims_create` is called in order to identify a suitable distribution of the available processes among the number of dimensions (i.e. 2 for this coursework) the image is to be decomposed in. The output of this function is the number of processes present in each dimension (stored in *cartcoordinate* array of two elements). The `MPI_Cart_Create` routine is then called in order to generate a new communicator that incorporates the topology arrangement defined after calling the `MPI_Dims_create`.

All the processes then call the `MPI_Bcast` routine such that a copy of the *masterbuf* is owned by all processes. Subsequently, each process copies the relevant part of the *masterbuf* such that all local copies are non-overlapping and equally sized (i.e. M/$p$ * N/$p$ dimensions, where $p$ is the number of processes in each dimension, assuming a perfect decomposition), by taking into account its coordinates within the newly-constructed cartesian communicator:

$$localarray_{i,j} = masterbuf_{cartcoordinate[0]*(M/p)+i,cartcoordinate[1]*(N/p)+j}$$

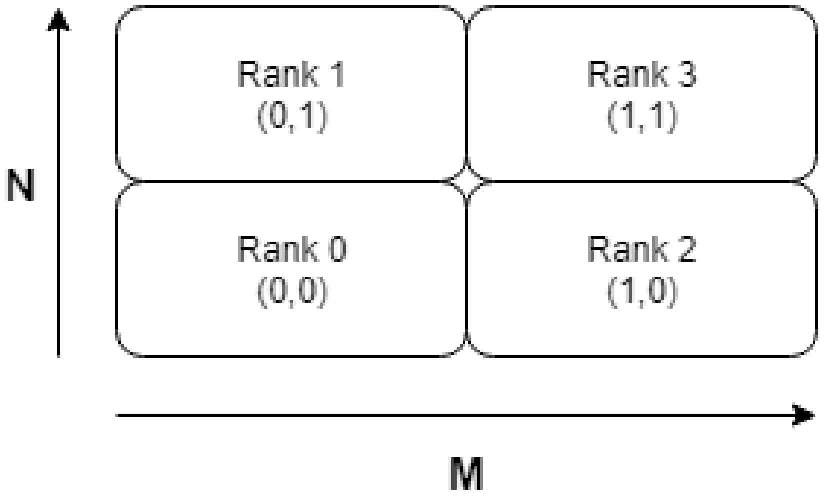

Figure 1: Picture of Cartesian topology used for 2 dimensional image decomposition on 4 processes.

## 2.3 Improving quality of each sub-image iteratively

Prior to the start of the iterative loop, *old*, *new* and *edge* arrays of size (M/$p$)+2 and (N/$p$)+2 are declared statically in order to include surrounding halo rows and columns. Unlike rows, columns are not contiguous in memory in C, therefore, a new datatype is declared using the `MPI_Type_Vector` routine, with count M/$p$, blocklength 1 and stride N/$p$ in order to handle the sending and receive of rows into halo cells.

The `MPI_Cart_shift` routine is called such that each process is able of calculating the ranks of its up, down, left and right neighbours. This information is crucial for enabling halo swapping through non-blocking communications (i.e. `MPI_Issennd` and `MPI_Recv`) among all processes. Every outgoing and ingoing communication is followed by a `MPI_Wait` in order to free up the buffer of each communication.

The iterative loop implements the same equation introduced in the Case study exercise, whereby the image is iteratively reconstructed from input edge pixel values using repeated operations of the following form:

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

## 2.4 Joining images and writing to file

At the end of the iterative loop, each process writes the resulting values stored in the *old* array (excluding halo cell values) back into each respective *localarray.* Again, by taking into consideration its coordinates, each process copies the contents of its *localarray* to the relevant part of *bigbuf* of size M*N:

$$bigbuf_{cartcoordinate[0]*(M/p)+i,cartcoordinate[0]*(N/p)+j} = localarray_{i,j}$$

Finally, the `MPI_Reduce` routine is called in order to achieve a global reduction among processes that effectively joins the contents of each individual *localarray* into a final *bigbuf* array, which is written to file by executing a call to the *pgmwrite* function provided within the complementary *pgmio.c* script.

## 3 Testing

In order to test the correctness of the developed algorithm, the provided *edgenew256x192.pgm* image was independently processed using the newly developed algorithm and the simple parallel solution to the Case study exercise (provided before start of coursework) on 4 processes and 1600 iterations. Upon visual inspection of the two processed images, most of the image details can be visualized, however, vertical and horizontal lines are present in the image reconstructed with the newly developed algorithm, suggesting that the bug most likely lies in the communication underlying halo swapping. The output *.pgm* files generated by the two algorithms were then compared using the built-in Linux *diff* tool. Consistent with the visual inspection, "5,2735c5,2735" was generated as output by the *diff* tool, meaning that lines 5 through 2735 in either file would need to be changed in order to match the lines in the other file.

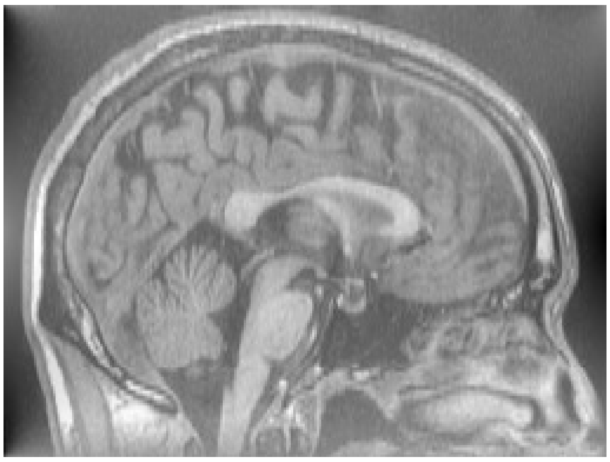

Figure 2: Processed 256x192 pixels image by Case study algorithm on 4 processes and 1600 iterations.

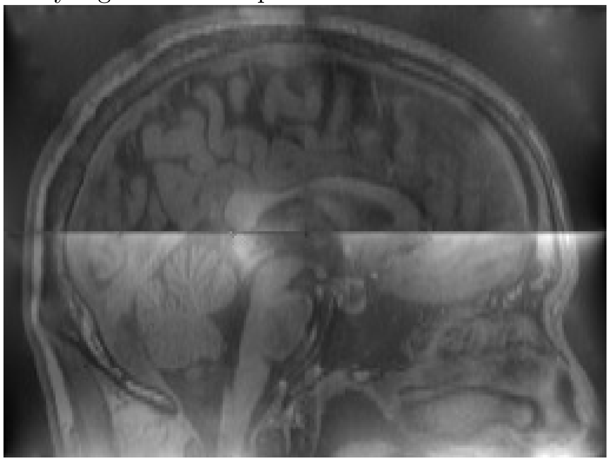

Figure 3: Processed 256x192 pixels image by Coursework algorithm on 4 processes and 1600 iterations.

## 4 Instructions on using the programme

The program can be easily run from command-line while being logged on Cirrus by editing a few lines of source code, in particular:

- M and N dimensions of the image to be processed (i.e. lines 8 and 9)
- Maximum number of iterations to be performend (i.e. line 10)
- Names of input and output image files in *.pgm* format (i.e. lines 40 and 220).

After loading the "mpt" and "intel-compilers-17" modules, the program can be compiled using Intel compilers, including an optimization flag, along with the *pgmio.c* script:

"mpicc -cc=icc -O3 -o `name_of_executable` `MPP_coursework_submission_B066600.c` pgmio.c"

Finally, the user has to specify the number of processes in order to run the program:

"mpirun -n 4./`name_of_executable`"

## 5 Analysis of Performance Data

In order to determine the parallel performance of the algorithm, the total execution of the programme from after the call to MPI_Bcast to immediately before the write to file was measured. This choice was made based on the fact that the MPI_Bcast routine requires a lot of memory and therefore including it would unnecessarily introduce a not insignificant and consistent overhead to the timing measurements.

The first step in the performance analysis was to benchmark the newly developed algorithm against the Case study parallel algorithm. Hence, both programs were independently run on three images of varying size as input, while keeping the number of iterations and processes fixed to 1600 and 4, respectively.

Table 1: Execution time (sec.) for different image sizes - Benchmarking Test

| Image Size | Coursework algorithm | Case study algorithm |
|---|---|---|
| 256x192 | 0.075 | 0.049 |
| 512x384 | 0.201 | 0.145 |
| 768x768 | 0.511 | 0.418 |

As it can be seen in Table 1, the coursework algorithm performs in a similar manner as the Case study algorithm. This suggests that the overhead of performing a 2 dimensional decomposition and non-blocking communications for halo swaps across 4 directions may not be very large. Based on these results, the performance of the coursework algorithm was fully profiled.

In figure 4, the execution time for processing images of 3 different sizes is plotted against number of processes. From this graph it be can be seen that, as expected, increasing the number of processes leads to reduced execution times, reaching an execution time of less than 0.12 seconds across all the tested images when run on 32 processes.

Figure 5 reports the speedup achieved thanks to parallelization compared against a "serial" (i.e. when run on 1 process) equivalent of the coursework algorithm executed for reconstructing three images of varying size. When compared to the smaller image sizes, the largest image nearly approximates a linear curve. Moreover, as the image size is increased by a factor of 4 each time between each image, a noticeable, indirect increase in speedup occurs (i.e. 2.56 fold change in speedup as the image size goes from 256x192 to 512x384, and a 1.61 fold change in speedup as the image size goes from 512x384 to 768x768). In particular, when the programme is run with the largest image on 32 cores, a maximum speed-up of 14.26x is achieved. It is clear from all these results that the performance of the newly developed MPI programme scales with problem size, suggesting that it is an efficient, parallel programme.

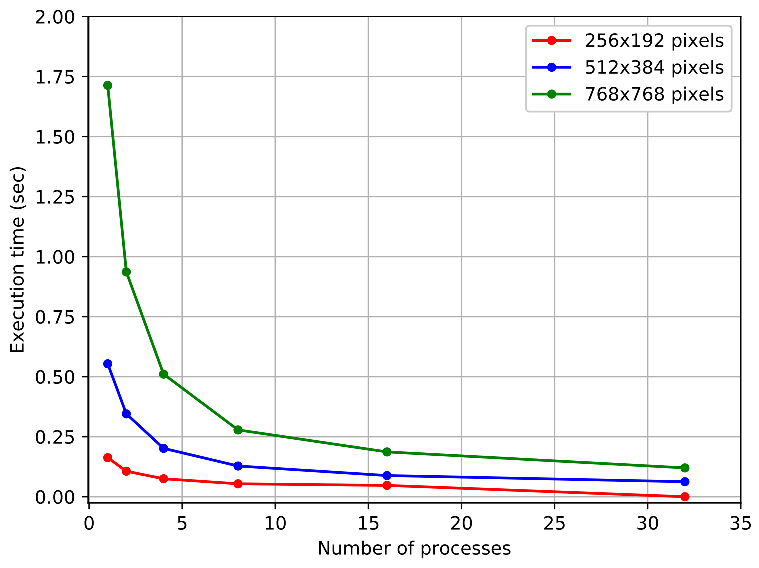

Figure 4: Execution time for the Coursework algorithm when used to process images of size 256x192, 512x384 and 768x768 pixels on 1, 2, 4, 8, 16 and 32 processes.

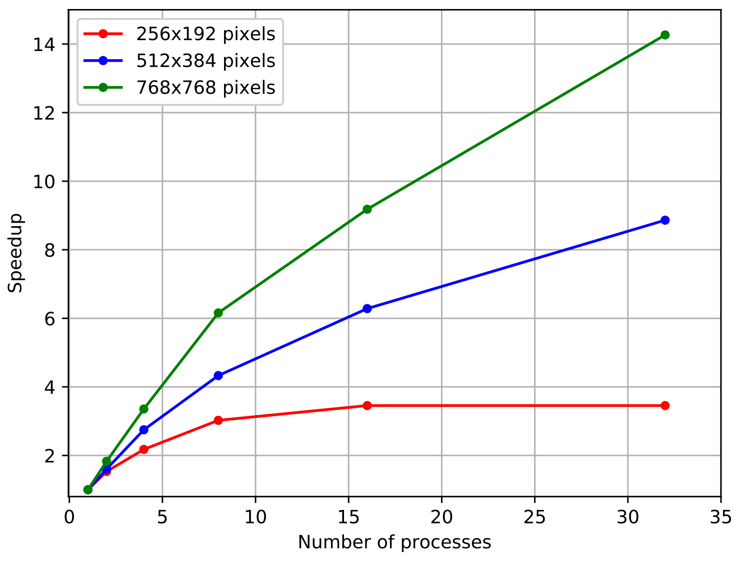

Figure 5: Performance speedup (calculated as T1/T$p$, where T is execution time and $p$ is the number of processes) for the Coursework algorithm when used to process images of size 256x192, 512x384 and 768x768 pixels on 1, 2, 4, 8, 16 and 32 processes.

## 6 Conclusions

A message-passing, parallel algorithm for a simple two-dimensional lattice-based calculation that employs two-dimensional decomposition and non-blocking communications has been developed. This programme has been benchmarked against a high-performing parallel algorithm for image reconstruction and exhibited good parallel performance when tested against increasing problem size. Although this programme exhibits an obvious bug that can be most likely traced back to the implementation of the halo swaps, it can be argued that this has an insignificant effect on performance and therefore the above results and analysis can still be considered valid.

Due to time constraints, the function needed for terminating the calculation in the iterative loop when the reconstructed image is sufficiently accurate could not be successfully introduced in the current algorithm version. A potential version of this function could be put together by having each process iterate over the bounds of each respective *old* and *new* arrays, compute the element-wise $\Delta$ between these arrays, reducing these values to a single one using the MPI_Reduce routine and finally leaving the program if the value of $\Delta$ is less than a threshold value (e.g. 0.1). Moreover, as it cannot be predicted at which iteration this function will cause the program to terminate the iterative loop, it can be expected that it will introduce a variable overhead that could potentially affect the interpretation of further performance experiments.