AC2 Coursework - MPI

Joshua Greenhalgh

May 11, 2015

1 Introduction

This coursework will investigate a parallel iterative algorithm for reconstructing an image from its edges. The parallel library MPI will be used to introduce the parallelism into the algorithm.

MPI stands for message passing interface and is a collection of functions, for C and also Fortran, that allow for communication between processors. In contrast to OpenMP, which is based on a shared memory model, MPI processes do not have any shared memory and so state and data must be explicitly communicated in order to synchronise computation.

The particular algorithm under investigation is representative of much more complex algorithms that implement parallelism via the decomposition of a cartesian grid. The iterative scheme used is based on the following update rule,

$$x_{i,j}^{n+1} = \frac{1}{4}(x_{i-1,j}^{n} + x_{i+1,j}^{n} + x_{i,j-1}^{n} + x_{i,j+1}^{n} - Edge_{i,j}). \quad (1)$$

This particular method for reconstruction is only possible because of the simple method used for extracting the edges from the original image. The pixel values for the edge image are calculated as follows,

$$Edge_{i,j} = Image_{i-1,j} + Image_{i+1,j} + Image_{i,j-1} + Image_{i,j+1} - 4Image_{i,j}. \quad (2)$$

As $n \to \infty$ the iterative algorithm converges such that $x_{i,j}^{n} \to Image_{i,j}$. This particular algorithm is very similar to finite difference methods for the solution of partial differential equations.

The calculation of the updated pixels at each iteration is only dependent on its surrounding pixels from the previous iteration. This allows us to

decompose the domain into a set of contiguous regions on which the iterative scheme can be applied independently on a single processor. The only points at which an issue arises is at the boundaries of the regions since the update to these pixels needs to use the value of the pixels in other regions. In order to overcome this issue it is necessary to share the boundary pixels for each region between processors and to update these values after each iteration.

2 Implementation

This particular implementation of the reconstruction algorithm is written in C. The three folders "time_vs_numprocLargeIm", "time_vs_numprocMediumIm" and "time_vs_numprocSmallIm" run the algorithm for $P = 3, 4, 6, 8, 12, 16, 18, 24$ processors on a 768×768 pixel image, a 512×384 pixel image and a 192×128 pixel image respectively. The final folder, "printims" runs the algorithm with $P = 4$ processors on a 192×128 pixel image for values of the stopping criterion $\Delta = 1.0, 0.5, 0.1, 0.05$ - the stopping criterion shall be explained in more detail below. In order to run the code on Archer each folder contains a script "runscript" which can be executed by running the command "bash runscript". This script compiles a series of C files that have been changed to reflect the changes in the parameters, the compiled executables are then submitted to Archer's queue system to be run on the compute nodes. Each executable returns three output files;

1. A file containing the total runtime of the algorithm, the average time for a single iteration and the average time spent communicating between processors for a single iteration.

2. A file which contains the average value of the pixels in the image and the maximum difference in pixel values between iteration every fifty iterations.

3. A file containing the reconstructed image upon completion of the algorithm.

The structure of the code falls into three main phases. In the first phase the edge image is read in to the processor with rank 0, distributed to all other processors and then decomposed. The second phase is the main loop of the program in which the iterations take place. The final phase is the reverse

of the first in which the reconstructed image is collected onto the processor with rank 0 and then written to file.

2.1 Phase 1 - Input and Initialisation

This section of the code begins by initialising the three files that will be used as output. In order to vary the output name of these files based on the value of the parameter under investigation the "sprintf" command is used which allows for filenames that are determined at runtime. After this the MPI library is initialised and various MPI specific variables are declared including the rank of the current processor. It is necessary for each processor to be able to access its rank in order for correct control flow to take place in the code. A cartesian topology is then set up which creates a new MPI communicator in which processors have the notion of being next to each other. This is enabled by using the function "MPI_Dims_create" which takes as input the number of available processors and the dimension of the cartesian topology required, it will return an array which defines a particular decomposition that the MPI library deems the most efficient. As an example if the number of processors available is $P = 4$ then for a two dimensional cartiesian topology the function will return the array $[2, 2]$ which means that the processors should be split into a 2×2 grid. This array along with an array which defines wether the topology is periodic, for this algorithm this is never the case, is then used as input into the function "MPI_Cart_create" which sets up the actual MPI communicator.

A set of two integer variables are then defined which determine the size of a local section of the image that a particular processor will work on. In this implementation it has been decided that all processors will work on a section of the image of the same size however this means that it is only possible to use a number of processors such that this is possible. Given an image of size $M \times N$ and a cartesian topology such that there are $P_x \times P_y$ processors then each processor will have a local section of the image of size $\frac{M}{P_x} \times \frac{N}{P_y}$. Once these integer variables are defined it is possible to create the five arrays needed for storing the values of the image. We must create three arrays that are of size $\frac{M}{P_x} + 2 \times \frac{N}{P_y} + 2$ which will contain the pixel values for the local section along with room for the shared halos form a processors neighbours. Two of these arrays will be very important during the iterations of the algorithm - the array "new" will contain the values of the pixels for the

current iteration and the array "old" will contain the values of the pixels in the previous iteration. The array "masterarray", "local_padded_array" and "local_array" are used as temporary storage during the initial phase of the code.

The processor with rank 0 reads the edge image file into "masterarray", this is then broadcast to all other processors. Each processor then copies its local section of the "masterarray" into the "local_array" this is possible since each processor in the cartisean topology has a co-ordinate which can be accessed using the function "MPI_Cart_coords". The local array is then copied into a padded array which allows for the shared halos to be incorporated. Finally the "old" array is initialised such that all pixels take a value of 255.0.

Before the iterations can commence it is necessary for each processor to know the rank of its neighbouring processors in order to be able to successfully swap halos each iteration - the function "MPI_Cart_shift" enables this. We must also create a custom MPI datatype which will allow us to communicate the columns, at the edge of the local section of the image, between processors, this is enabled by using the "MPI_Type_vector" function.

2.2 Phase 2 - Iterations

The iterations for this algorithm are contained in a for loop which runs from zero to "MAXITER", which has been set to 10000. Each iteration begins by swapping the halos of each local section with the processors neighbours. It is necessary to run four sends and receives on each processor, one for the processor above, below, left and right within the cartesain topology. Once the Swaps have been made the actual pixel update can be processed and stored in the "new" array. Every 50 iterations two quantities are then calculated, the average pixel value of the image and the maximum difference between pixels. These values are calculated on each processors local section of the image before using the function "MPI_Reduce" which allows for the calculation of these values across the entire image. If the current maximum difference between pixel values is less than the parameter "DEL" then a flag is set, on the processor with rank 0, which will be used to break out of the for loop and finish the iterative algorithm early. If this flag is only set on a single processor and is then used to break out of the for loop the algorithm will not work because the other processes will continue the iterations - it is therefore necessary to broadcast the flag value to all processes before breaking out of the loop. At the end of each iteration the values in the "new" array must be

copied into the "old" array.

2.3 Phase 3 - Writing reconstruction to file

Upon completion of the iterations, either due to the early stopping criterion or because the maximum number of iterations has been reached, the algorithm proceeds to collect the localised sections of the images and bring them back on to the processor with rank 0. This is performed by initialising, with zeros, an $M \times N$ array called "outarray". The local section of the image belonging to each processor is then copied into the correct location in this array. Each processors array is then reduced into the "masterarray" using a sum - this is possible because each processors array consists of zeros in all positions other than its local section of the image so the sum of these arrays is the full array containing the reconstruct image. The "masterarray' is then written to file which completes the algorithm. The final part of the code is to call "MPI_Finalize" to free memory and then exit the program.

2.4 Code verification

The main technique used to enable verification of the correctness of the code was the use of a small 12×12 input array that was filled such that the element at position (i, j) was given by $i + j$. The use of this array allowed me to check that the halo swaps were being implemented correctly via the printing out of the local sections for each processor along with the halos for five iterations. The clarity of the output reconstruction also allowed for the verification of the algorithm. At one point the swaps where not being done in the correct way - the left halo was being sent to the right hand processor and vice versa - the output is shown in figure 1.

Figure 1: Incorrect Reconstruction.

As another check on the correctness of the code the average pixel intensity at a particular iteration should be the same regardless of the number of processors used since the actual calculations that are undertaken do not change they are just done on different processors. In figure 2 the average pixel value as the iterations increase is plotted for various numbers of processors. It can be seen that the decrease in the average pixel intensity does not depend on the number of processors as would be expected.

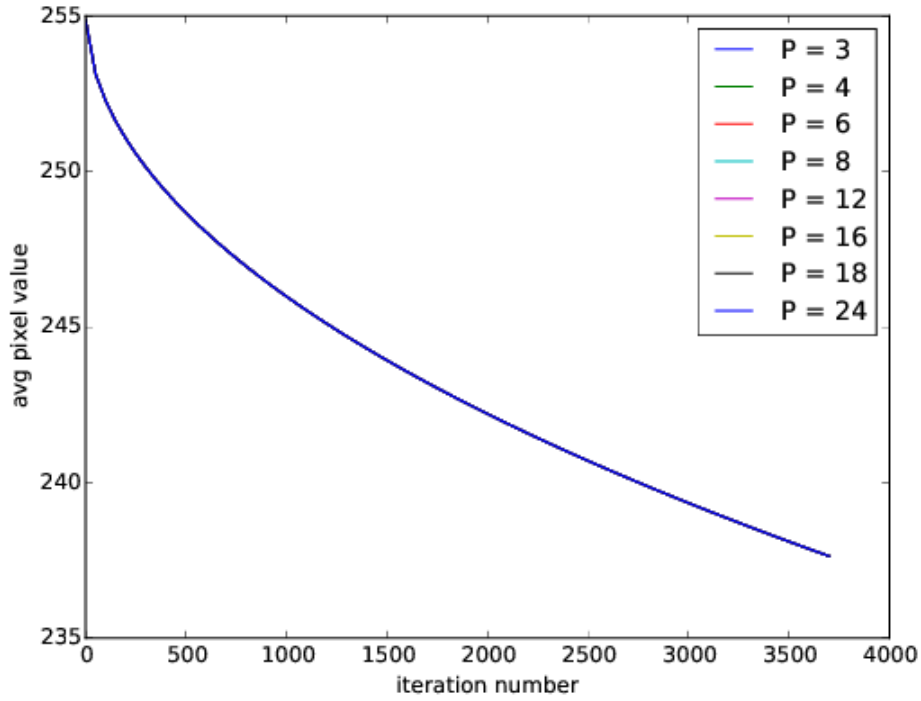


Figure 2: Average pixel value by iteration number for different numbers of processors.

3 Results

The iterative reconstruction algorithm was run on three images (large - 768×768, medium - 512×384 and small - 192×128) for a maximum of 10000 iterations with an early stopping criterion of $\Delta = 0.05$. Each image was run on $P = 3, 4, 6, 8, 12, 16, 18, 24$ processors. The average time taken for a single iteration was calculated by timing the execution of all iterations and then dividing by the number of iterations that had run when the algorithm stopped. The average time per iteration spent communicating between processors, specifically in swapping halos, was also recorded. As can be seen in figure 3 both the large and medium images show a linear increase in speed as the number of processors increases as would be expected.

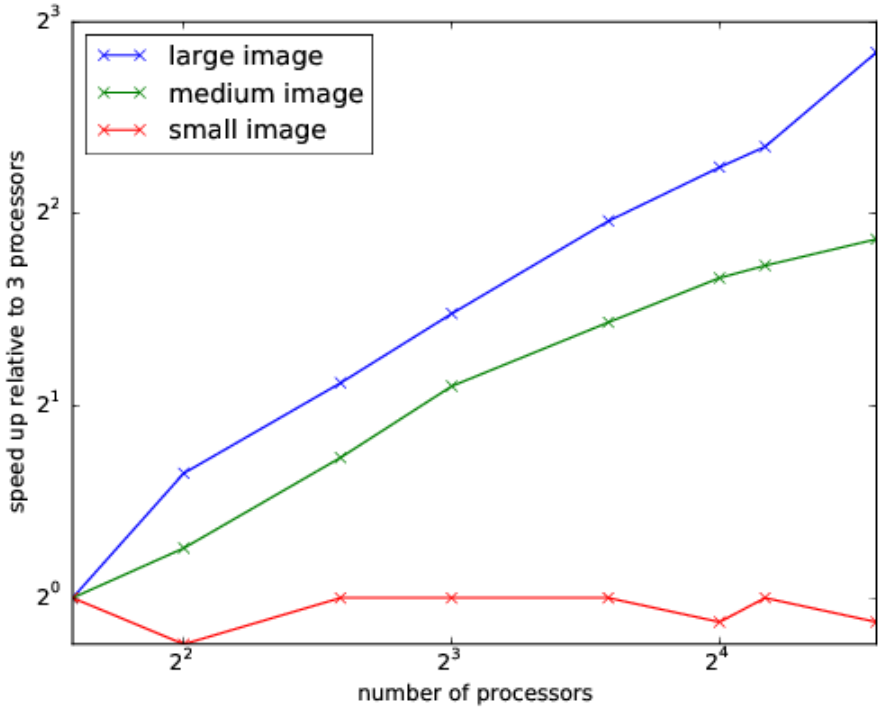


Figure 3: Speedup of the calculation, relative to the time taken with three processors, as the number of processors increased.

However the small image shows no speed up at all. In order to understand why the small image did not show the expected behaviour I decided to look at the average time spent swapping halos in each iteration. The results for this can be seen in figure 4 which shows the actual average time spent on communication. As would be expected the larger the image the more time was spent on communication since larger amounts of data needed to be exchanged, also as the number of processors increased there is a general increase in communication time because more messages must be passed between processors.

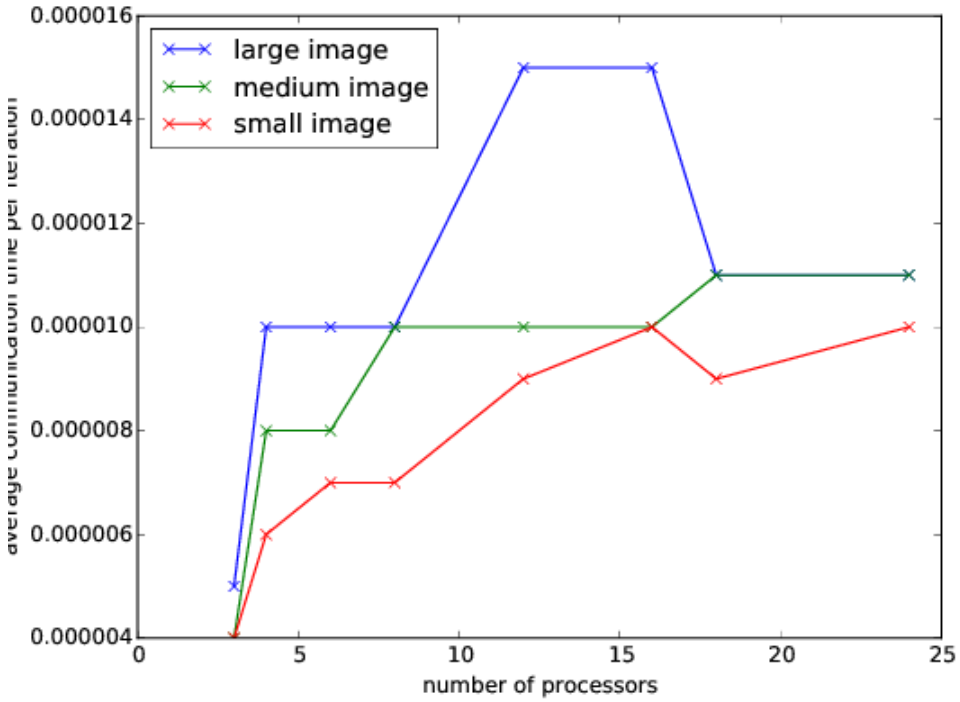


Figure 4: Average time spent communicating between processors per iteration

It is not however clear why the small image shows no speed up until we look at the proportion of each iteration spent on communication. In figure 5 it is clearly shown that the overhead in communication is significantly larger for the smaller image.

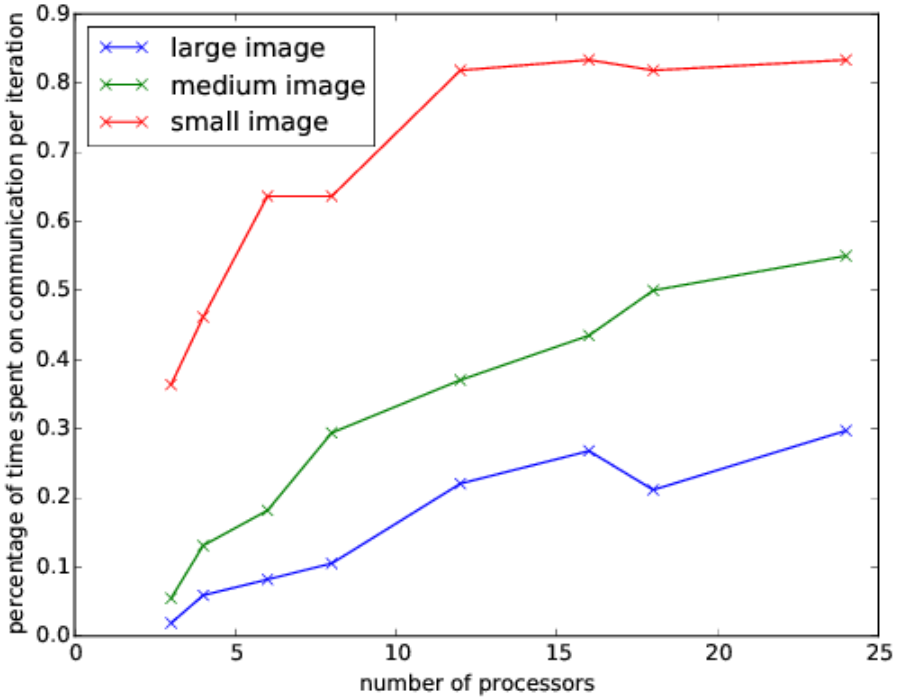
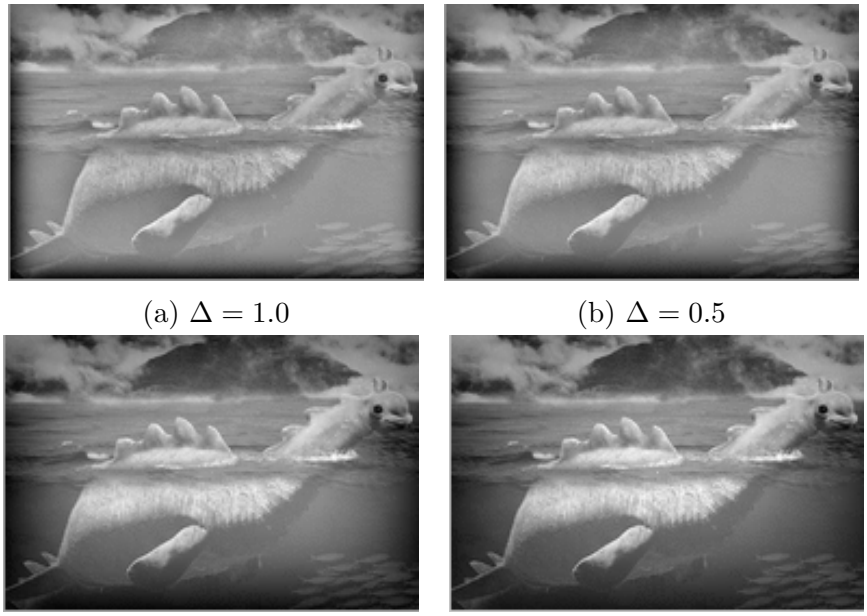


Figure 5: Average proportion of time spent communicating between processors per iteration

When the algorithm is applied to the small image on more than 6 processors 50% of the time per iteration is spent in communication. It is no wonder that there is no increase in the speed of the algorithm in the case of the small image since the majority of the time is spent on operations that are not actually calculating the reconstruction of the image. In the case of the large image the proportion of the time spent on communication never reaches more than 30% for any number of processes and so we do see a linear increase in speed as the number of processors increases. In the case of the medium image the proportion of time stays below 50% up until 18 processors are used but is generally more than for the large image which explains the smaller increases in speed we see.

In figure 6 the output reconstruction is shown for values of $\Delta = 1.0, 0.5, 0.1, 0.05$ as can be seen in these images as the value for the stopping criterion is decreased we gradually improve the quality of the reconstruction.



(a) $\Delta = 1.0$ (b) $\Delta = 0.5$

(c) $\Delta = 0.1$ (d) $\Delta = 0.05$

Figure 6: Reconstructed image for $\Delta = 1.0, 0.5, 0.1, 0.05$.

4 Conclusion

This coursework has implemented a simple, but nevertheless representative, example of parallel computation using MPI. It has been shown that the computation time is decreased in a linear fashion as the number of processors used is increased. However it has also been noted that if a computation is not particularly intensive then the overhead of inter-processor communication can mean that no speed up is seen in the runtime of the algorithm. It could be of interest to use a larger number of processors, such that more than a single node of archer would be used, and see if the increases in communication would eventually lead to a plateauing of the algorithms speedup even for the large and medium image.

The particular implementation could be improved such that any number of processors, not just quantities such that the image is equally divisible, could be used. This would require the implementation of the local sets to be able to deal with variable sizes or could involve the zero padding of the original image in order to enforce divisibility.