MPI Coursework

Ben Kitching-Morley - University of Southampton

May 27, 2019

bkm1n18@soton.ac.uk

Abstract

In this coursework submission the performance of an algorithm for inverting edge-detection is improved through the use of parallelisation of code. This is done through 2D domain decomposition using the MPI (Message Passing Interface) library implemented in C. Performance was tested using the Archer supercomputer running on a range of process numbers from 1 to 192, using 4 different test images. Analysis of the results was done using Pandas in Python, with the greatest speed up achieved being 24.1 times against the code being run on a single process.

1 Introduction

In the algorithm being optimized, the performance limiting step is an interative loop, performing the following calculation

```
for(int i=1; i<isize+1; i++)
  for(int j=1; j<jsize+1; j++)
    new[i][j] = 0.25 * (old[i-1][j] + old[i+1][j] + old[i][j-1] +
                        old[i][j+1] - edge[i][j])
for(int i=1; i<isize+1; i++)
  for(int j=1; j<jsize+1; j++)
    old[i][j] = new[i][j]
```

Here i and j can be thought of as indexing the pixals of the image. Cruicially, at each iteration, the value of any pixel will only be affected by its immediate neighbours. This allows for efficient parallelisation by breaking the image down into rectangular chunks, and distributing each of these chunks to a different process. To complete a single iteration, each process need only be given the values of the pixals immediately surrounding its chunk of the image. Throughout this coursework these surrounding pixals will be reffered to as halos.


Figure 1: Example of the algorithm reversing the edge detection on the left to give the output on the right. This image was produced using implementation on 192 processes over 8 nodes

2 Method and Implementation

2.1 Dividing the image

The image was divided into a rectangular grid, where the grid was chosen to be as square as possible (e.g. have a similar number of divisions in the x and y directions), while dividing the number of pixals in the image exactly. For example, in a 24 by 32 image on 4 processes the chunks would be of size 12 by 16, while on 6 processes the chunk size would be 8 by 16, because 32 isn't divisable by 3. For this reason the process numbers used in the performance analysis had to be chosen carefully so that they divided the image sizes. Below is the function used to implement this maximum squaring method. i and j are pseudonyms for the x and y directions respectively, size is the number of processes.

```
while( (temp <= sqrt(size)) || !solution ){
    if(nj%temp == 0 & // The j-axis divides exactly
       size%temp == 0 & // temp divides the number of processes exactly
       ni%(size/temp) == 0 // The i-axis divides exactly
    ){
      solution = true;
      max = temp;
    }
    temp++;
  }
return max;
```

Once jdim is determined then the dimensions are set into a read only array, for use in creating a Cartesian communicator, with MPI_Cart_create. The rank and coordinates of each process within the communicator are then obtained using MPI_Comm_rank and MPI_Cart_Coords for later use.

```
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &cart_comm);
```

2.2 Reading in and distributing data

In the implementation of this MPI program rank 0 was chosen to be the master process, meaning it had the task of reading in the data file and distributing the data to the other processes. Data was read into a masterbuf array on the master process using the utility function pgmread, provided with the coursework, in the pgmio.c file. Because the distribution of data doesn't make up a significant portion of the program runtime, simple MPI_Isend and MPI_Irecv methods were used for data distribution as opposed to the more highly optimised scatter method. The data was sent in the form of a vector type derived datatype.

```
MPI_Type_vector(isize, jsize, nj, MPI_DOUBLE, &block);
```

Here isize and jsize relate to the size of the image chunks being distributed, while ni and nj represent the size of the whole image. Using this method blocks could be sent with Isend by indexing appropriately.

```
for(int i = 0; i<idim; i++){
  for(int j = 0; j<jdim; j++){
    const int other_coords[2] = {i, j};
    MPI_Cart_rank(cart_comm, other_coords, &send_rank);
    MPI_Issend(&masterbuf[i*isize][j*jsize], 1, block, send_rank, 0, cart_comm, &request);
  }
}
```

Because the data was received by a smaller array buf, which has size (isize, jsize), then the Irecv call implemented simply received isize*jsize doubles. The gathering of the data at the end of the program is implemented in a similar way.

2.3 Halo Swaps

In order for nieghbouring processes to swap halos, each process uses two things. Firstly it must know if it is on an edge of the image (e.g. what neighbours it has) and secondly, if it does have a neighbour in a given direction, what is that neighbours rank. For the first task a simple function (find_edges()) which takes in the Cartesian coordinates of a given processs was written. Neighbour ranks were found using MPI_Cart_shift.

```
MPI_Cart_shift(cart_comm, 0, 1, &left, &right);
MPI_Cart_shift(cart_comm, 1, 1, &down, &up);
```

When initialising arrays used in this coursework, the utility function arralloc, from the arralloc.c script provided with the coursework, was used. This function creates arrays which are contiguous in memory. More specifically, cycling through the j-direction cycles through memory addresses one at a time. Therefore, when performing a halo swap it was necessary to consider whether the halo is laid out in the i or the j direction. If the halo is in the j direction (column), then the swap involves swapping jsize MPI_DOUBLE types, while if its in the i direction (row) then a derived datatype must be used.

```
MPI_Type_vector(isize, 1, jsize + 2, MPI_DOUBLE, &row);
```

Non-blocking communication (Isend/Irecv) was used for all halo swaps, with an MPI_Allwait and MPI_Barrier at the end of the swapping.

2.4 Algorithm Termination

Each iteration the average pixel value, and the maximum pixel change, Δ, were calculated. The latter was used so that the iterative part of the algorithm could be terminated it went below a certain value. Since finding the maximum change requires if statements, which can be computationally expensive due to branch

prediction, then this calculation was only done every 100 iterations. Finding the maximum across processes was implemented through the highly optimized MPI_Allreduce directive. The average pixel value was also printed every 100 iterations.

```
MPI_Allreduce(&temp, &delta, 1, MPI_DOUBLE, MPI_MAX, cart_comm);
```

3 Results

3.1 Strong Scaling

The performance of the MPI parallelisation was tested across a range of process numbers from 1 to 192. The metric used in this testing is the time per iteration during the iterative part of the algorithm.

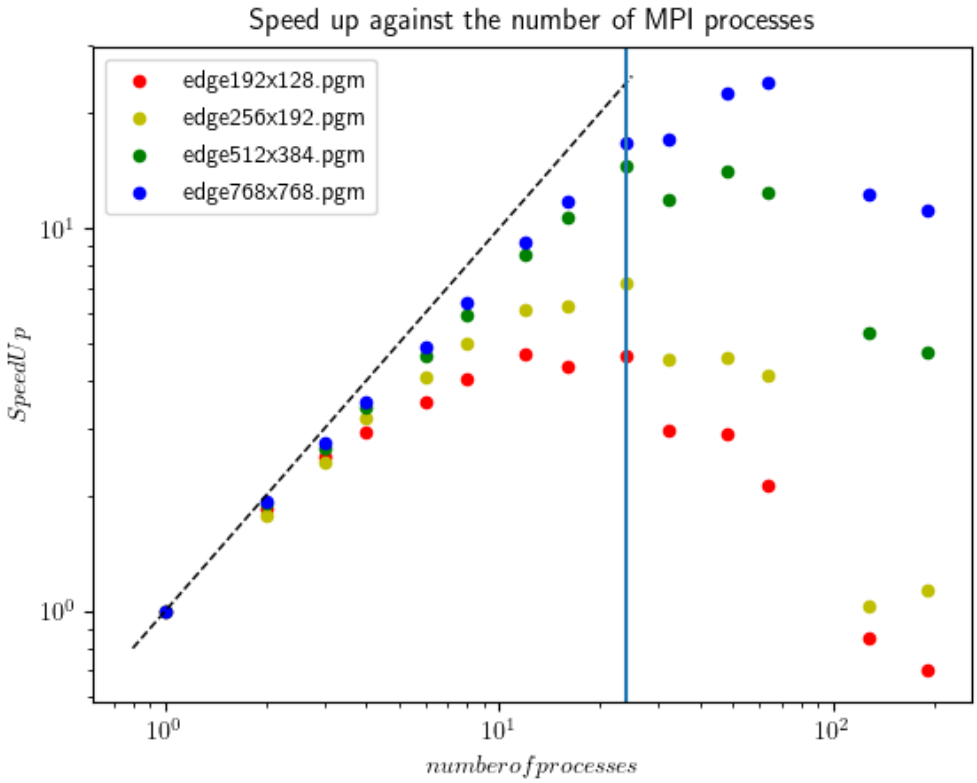

Figure 2: Figure demonstating the strong scaling of the edge detection reversal algorithm. Dashed black line shows the ideal scaling relation, while the blue line is at process number 24, representing the number of processes that can be fit onto one Archer node

Figure 2 displays the result of this performance testing. Firstly it is worth noting that performance on all test images saw almost linear scaling at low process numbers, showing that MPI parallelisation was an effective method of improving code run time. However, speed up for all but the largest image wasn't observed to improve when multiple nodes were used. In general, figure 2 demonstrates that the performance speed up that can be achieved with this implementation of MPI directives is greater for the larger images. This is due to a trade off between MPI overhead and the amount of work each individual process must do. In the

case of a small image, such as edge192x128.pgm, there is less work to do overall, so the benefits of sharing this across multiple processes is quickly outweighed by the overhead necessary. Table 1 demonstrates one way this overhead builds up as process number increases. For smaller images, the halos are a relatively large fraction of the domain size, creating a lot of overhead with little payoff. In comparison this ratio becomes more favourable as the images become larger. Another significant piece of overhead comes about in the implementation of the MPI_Barrier.

Figure 3 demonstrates one perhaps suprising quality of this implementation; the uncertainty in run-time length increases as the number of processes increases. This variation may be due to uncertainty in how the processes are laid out across the nodes when Cart_create is called. For each data point 5 runs were executed so that a mean and standard deviation could be calculated. One especially prominent result is for 128 processes on the 256x192 image. What was different with these parameters is unclear, and the resulting image was verified to be correct. It is worth noting that the mean is significantly above where it would be expected, so there was likely a single outlier data point, which took far longer than the other points.

image	Domain size	Size of Halos	Ratio
edge192x128.pgm	128	48	3.0
edge256x192.pgm	256	64	4.0
edge512x384.pgm	1024	128	8.0
edge768x768.pgm	3072	224	13.7

Table 1: The size of the individual domains and halo size on the 192 process test case for different image sizes

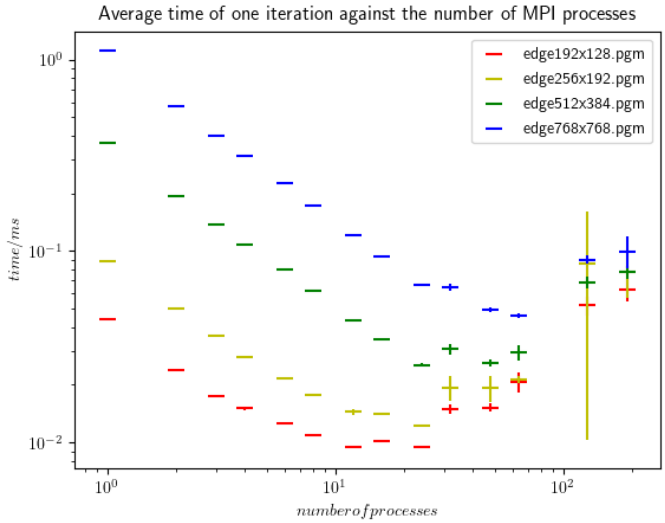

Figure 3: Iteration time against the number of processors for each of the images. The vertical lines show the standard deviation in the run times

3.2 Weak Scaling

Following expectations, weak scaling performance appears to be closer to the theoretical ideal than strong scaling. Figure 4 demonstrates how the time per iteration varies with process number, as the Domain size

on each process remains the same (e.g. as image size increases). The ideal result here would be a constant time for a given domain size. For the larger domain size (in blue) this is roughly the case, with only a 49% increase in computation time associated with a 24 times increase in the problem size. In comparison, the weak scaling was much poorer for the smaller domain sizes. One factor that explains this is the fact that the smaller domain sizes are faster at the their smaller process numbers to start with, meaning for a fixed increase in run-time there will be a larger percentage increase recorded. However, there is a second effect which is more of an artifact of the data which is available. A careful look at this figure reveals that the time increase from going from x processes to y is roughly equal for any domain size. Instead, what changes between the different domain sizes is the range of process numbers which is spanned. At the smaller domain size weak scaling is poor as a large number of processes (up to 192) are being used, which introduces a large amount of overhead from calls such as MPI_Barrier() and large number of halo swaps.

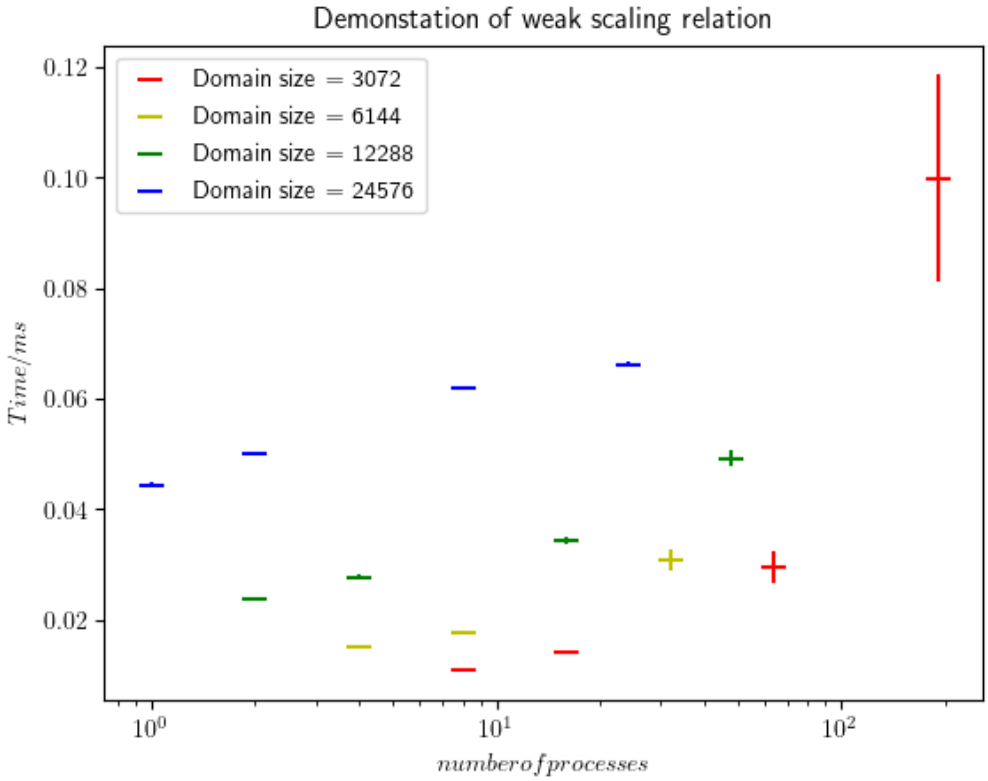

Figure 4: Weak scaling of the edge detection reversal algorithm. Domain size is the number of pixels per processes.

4 Testing

Several tests were done to check the validity of the code during the implementing phase of this project. Firstly, the distribution part of the implementation was tested by writing out the smaller buffer images which comprise part of the image, and cross referencing them with the Cartesian rank of the communicators. After the MPI_Cart_shift call, the neighbours were assessed for self consistancy, e.g. if one rank has a left neighbour of another, does that rank have a right neighbour of the orignal rank, etc.

One validation method used to check that the parallel code was giving the same results as the serial code (apart from trivially checking the images matched by eye) was through printing the number of iterations taken to converge to tolerance. In all cases this was found to be the same regardless of the number of processes used. This metric is recorded in the results.csv file.

Figure 5: Example of output from the program during the debugging phase. This image contains vertical lines due to a failure in the halo swaps. The final program was checked to not produce such errors

5 Discussion

One limitation to the scope of this implementation, is the requirement that the domains of all processes must be the same size. This has the knock on effect that only certain process numbers can be used for a given image size as the pixels have to divide exactly. This isn't ideal if the size of the image is not known before hand. Moreover, if an exotic sized image, (nx, ny) where nx and ny are prime, were used then parallelisation would be impossible. This issue could be resolved by generalising the code to allocate variable sized domains to each process.

In terms of the performance speed up observed, this method of parallelisation is only worthwhile for larger images. The author suspects quite an impressive speed up would be observed if the method were applied to a 4K image. However, at this point the reading in of the image to masterbuf on the master process would become expensive, and so the reading in of data would require adjusting.

One area of possible performance improvement is in doing the work that doesn't require data halos, before the processes wait for the halo swaps is completed. This would be reasonably easy to achieve and would involve looping over the central portion of each domain before the wait statement, and the remaining edge afterwards.

6 Appendix

6.1 Running the scripts

The run method used to collect the data used in this report has been designed to be as straight-forward as possible, for ease of reproducability. For the results obtained on X nodes, run the following from the main

directory of the submission.

```
qsub ./pbs/performance_X.pbs
```

It is very important that the command is run from the top directory, as opposed to inside the pbs directoy as compilation, log file storage and result output all occur with respect to this directory. The results should be appended to the file results.csv.

To run just a single test, use the run executable. The template for this is run {number of processes} {image name}. This script will then compile and run the script with those parameters, by creating a .pbs file and submitting it. The logs will end up in the logs directory, while the c output goes to c_logs and the output image goes to /im/ directory.

```
run 24 edge192x128.pgm
```

Note that if the implementation isn't being tested on Archer, but rather on a PC then the user must type commands similar to the following, where filename represents the image file to be run with.

```
mpicc ./c/parallel.c -o ./ex/parallel
mpirun -n {no processes} ./ex/parallel {filename}
```

6.2 Breakdown of the contents of the submission

6.2.1 Directories

The top directory is bkm1n18, inside this directory are the following directories:

- **c** - Contains c scripts. Inside is parallel.c which is my implementation of the coursework algorithm for any number of processes. Also contained in this directory are pgmio.c and arralloc.c which are the helper scripts distributed with the coursework.
- **im** - Contains the four test images, e.g. edge192x128.pgm etc. This folder is also where output from script execution will go. Output files are named with the convention 'out_no. processes_filename' where filename is for example edge192x128.pgm. Included with the submission are some example output files which have been included for the purpose of showing the validity of this courseworks implementation of the algorithm.
- **ex** - This folder is where executables are saved. In practice since in the end only one script was used, only one executable (parallel) will be stored here. It is still important for the executing that this folder remains.
- **logs** - This is where the log files produced by Archer accumulate
- **c_logs** - This is where the C output from execution is redirected. The naming convention used is no processes_filename.txt, where filename is for example edge192x128.pgm
- **pbs** - This is where the .pbs scripts which were used to produce the data seen in this report were stored. Contained within this are scripts of the form 'performance_X.pbs' where the X represents the number of nodes the .pbs file requests.

6.2.2 Files

- **test_0.pgm** and **test_1.pgm** Files demonstrating the correct reading in and distribution of the of the image data. These are the contents of buf distributed to processes at Cartesian ranks [0, 0] and [0, 1] respectively on a 4 process run of with edge512x384.pgm. These images demonstrate the convention

that the indexing begins at the bottom-left corner of the image, with the i direction being left to right and the j direction being down to up.

- **archermpi.pbs** This template script is copied, edited and submitted by the run executable in execution of a specific run
- **run** An executable used for individual test runs
- **results.csv** A csv file used in data collection for performance analysis. Contained in the file.
- **strong_scaling.png**, **weak_scaling.png** and **time_processes.png** - Figures 2 4 and 3 respectively. These are produced as output by analysis.py
- **analysis.py** - Python3 script performing the data analysis of this project using the pandas library.
- **error.pgm** - Used in figure 5