|epcc|

Message Passing: Coursework

B083194

December 3, 2015

Contents

1 Introduction

The aim of this project is write a message passing parallel program for a simple two-dimensional lattice-based calculation, that employs a two-dimensional domain decomposition, and uses non-blocking communications. The program must perform pixel calculations to enhance an image, based on edge detection. The image must utilise halo swapping, and two different types of boundary conditions: periodic and sawtooth. Stopping rules will be introduced, and these are defined through monitoring the maximum absolute change in pixel value (delta), then stopping after an appropriate number of iterations.

Testing is a crucial part of this task, and there are a variety of ways we are going to do this, for example: check parallel results against serial and checking the absolute value of the maximum pixel change. Execution times should speed up as more processors are used, especially for large images. To send and receive the work, we used derived data types and non blocking communications. Cartesian topologies are introduced to find send and receive destinations.

There are 5 edge files to perform calculations on. They are called `edgenew[`$size$`].png`, where size specifies the size of the image. Hereafter, these will be referred to as I[$dim1$]. For example, edgenew192x128.png will hereafter be referred to as I192. All tests were run on all 64 cores of Morar, and compiled with on the command line with `mpicc -lm -fastsse attempt3.c pgmio.c -o run`. The equation for updating pixel values uses the value of a pixel, plus four (non-updated) values above, below, right and left of it. This equation is shown if Equation (1).

$$new_{i,j} = 0.25 * (old_{i-1,j} + old_{i+1,j} + old_{i,j-1} + old_{i,j+1} - edge_{i,j}) \quad (1)$$

2 Code Design

2.1 Breaking Up The Iteration Space and Work Distribution

The whole picture can be considered to be the iteration space. This must be broken up into chunks for processing. To do this, each processors is first sent a copy of the `masterbuf` (whole image before processing). Each processor should perform calculations on it's part of the iteration space only, but how do we assign it iterations? For this we need to introduce a Cartesian coordinate system.

2.1.1 Cartesian Coordinate System

Cartesian topologies create a new communicator, which is designed in a more structured way, and can significantly improve performance [2]. In other words, each process is connected to its neighbours via a virtual grid [1]. Neighbours can then be identified

by by their Cartesian coordinates. In order to do this, we must first find the dimensions of the system using size (number of threads) as an argument. The new communicator is then created from the old communicator using `MPI_Cart_create`. We then determine process coordinates in the Cartesian topology given rank in group. The process is shown below.

```
MPI_Dims_create(size, ndims, dims);
MPI_Cart_create(comm, ndims, dims, period, reorder, &comm2d);
MPI_Cart_coords(comm2d, rank, ndims, coords);
MPI_Cart_shift(comm2d, 0, disp, &left, &right);
MPI_Cart_shift(comm2d, 1, disp, &down, &up);
```

The last two lines above are used to find the neighbours to the left, right, up and down. This become important during halo swapping later on.

2.1.2 Breaking Up The Iteration Space

Each processor has a copy of the masterbuf, however, we only want calculations to be performed on a particular section of the picture. To do this we need to use a function, called blocks, which uses pointers to return an array. This function requires the array, $dims[]$, from 2.1.1 as an argument. Blocks finds the start and end pixel assigned to each thread, and we subtract these to get the dimensions of the block each processor will perform calculations on (MP and NP). From this we also get start and end points of the assigned iteration space using the MP and NP, and $coords[]$. The region of the masterbuf which assigned to each processor is copied into a smaller buffer with dimensions MP and NP, called buf, ready to perform calculations. Figure 1 displays the portion of the image each processor performs calculations on.

1	3	5	7
0	2	4	6

Figure 1: The iteration space should be divided like this between 8 processors. Each number represents thread rank.

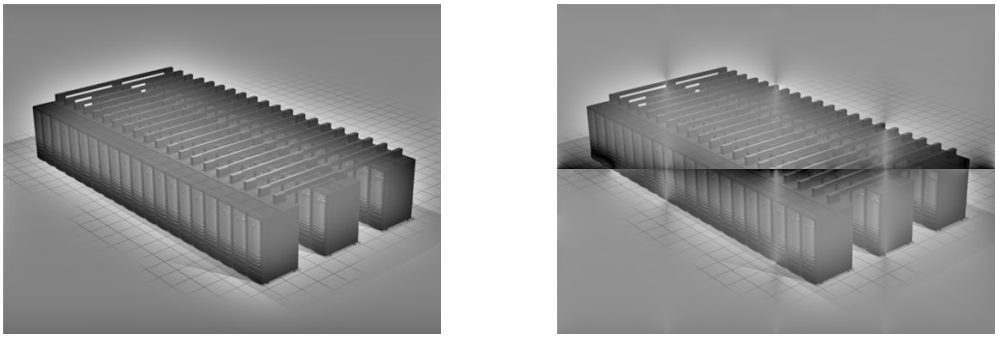
2.2 Boundary Conditions

There are two types of boundary conditions we are going to consider: periodic and sawtooth. Periodic boundary conditions are useful for modelling large computer sim-

ulations and mathematical models [3]. In our case, we have horizontal periodic condition. This means if the formula attempts to use a pixel value outside the left horizontal boundary, it will instead use the pixel value one in from the right boundary. Saw tooth boundary condition are similar to periodic, except follow a "sawtooth" shape in the form of equilateral triangles [4]. The periodic boundary on the right and left hand sides are initiated with a for loop, updating out-of-bounds values around the edge. The saw tooth boundary condition also uses a for loop, but finds a value from a function `boundaryval`, and uses this to calculate the out-of-bounds values around its edges.

2.3 Halo Swapping

Halo swapping is an essential part of multi-core image processing for pixel calculations use the values of pixels around them. Because of the nature of our code, if we do not perform halo swapping, the image will use incorrect halo data when calculating new pixel values around the edges. This results in in "lines" through the transformed image. This is shown in Figure 2.



(a) Image processed with halo swapping. (b) Image processed without halo swapping.

Figure 2: The processed image with and without haloswapping.

Figure 2(b) is clearly wrong. Two dimensional halo swapping cannot be done with simple data types, because these halo data are non-contiguous. We introduce a derived data type, called block. Block is of vector type, and sends the halo data of the top and bottom edges. The derived data type block is has MP elements (does not include halo ring), creating by striding over NP+2 elements each time. It is then committed, ready for use.

The halo data is sent using non-blocking sends and receives. The send destinations, and receive sources are one of: up, down, left or right, as found from the Cartesian topology in 2.1.1. Because of the way C creates arrays, communications between left and right processors can send their halo data as a float, as it is saved contiguously. However, communications between up and down must used the derived data type, block, because they are not saved contiguously. After the sends and receives have been called for both left and right, or up and down, the wait function is called to make sure the data transfer is complete. With halo data in place, we can now continue with our calculations.

2.4 Calculations

We want to perform calculations on each pixel until the picture become clear, but how many iterations is this? One way to try this, would be to iterate for different numbers of calculations until the image looks clear. But some people may have a different definition of clear, and this number will vary from picture to picture. We introduce a new variable to quantify image clarity, delta, where delta is calculated by equation (2).

$$\Delta = \max_{i,j}(|new_{i,j} - old_{i,j}|) \tag{2}$$

Delta is the maximum pixel change for any pixel in the image, and we only iterate while delta is greater than some user defined value, min. Throughout this project we have set min to be 0.1. Now the clarity of the image can be quantified, and ambiguity from image sharpness is removed.

Calculating delta every iteration can however be time consuming, and affects performance. To optimise the performance, we define a new parameter, CRITFREQ. If this value is set to 10, then we only calculate delta every ten iterations. However how do we define CRITFREQ so optimise performance, whilst also getting a value of delta close to 0.1? Because this will be different for different images, this will hereafter be set to 10. This way, the number of iterations completed will always be will be with 9 of the ideal number, without the overhead of calculating it every time. This is explored further in section 4.4.

2.5 Collecting Image Fragments

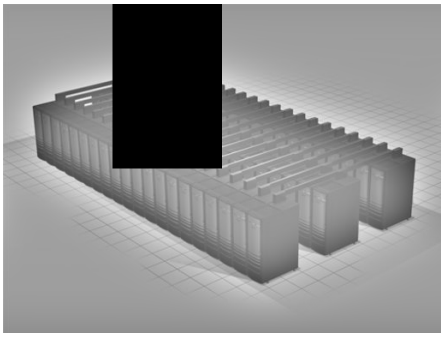
The different pieces of the image have all been transformed by different processors, now we must collect them together. We employ a crude method by setting all masterbufs = 0, then writing each buf into its appropriate place in its corresponding masterbuf. We reduce all the masterbufs, by way of a sum, into a new buffer called finalbuf. This works because all the values in each masterbuf are zero, other than those which have been transformed. Now each pixel value is the sum of seven 0s and the transformed pixel value, giving us our final transformed image in finalbuf. Finalbuf is then written into a .pgm file to give the final image.

There are a variety of places in which we could start and stop timing to measure performance. It seems a bit redundant to measure the serial part of the code, which remains largely unchanged. All tests hereafter have been timed immediately before, and immediately after the while loop, unless otherwise stated. This should produce reproducible times, which can provide good insight into the performance of this message passing program.

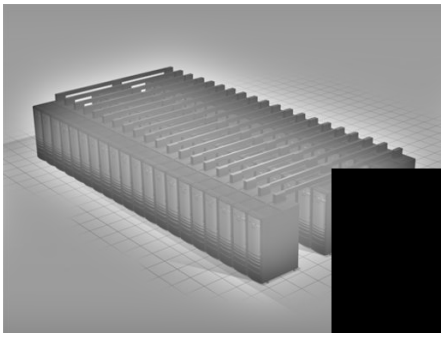
3 Validation Testing

3.1 Processor Location

We want to check that the processors have been created a virtual topology (as in Figure 1). To do this, we tell a processor to set all pixels values to 0 (black), instead of their transformed value. We can then identify the portion of the image each processor has transformed. This was tested using 8 processors, first blacking out processor rank 3, then processor rank 6. This is shown in figure 3.



(a) Rank 3 blacked out.

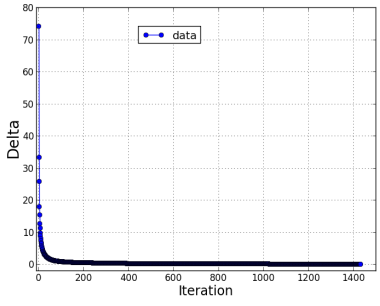


(b) Rank 6 blacked out.

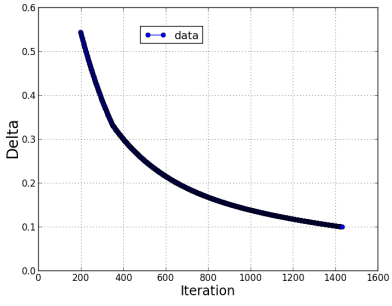
Figure 3: The processed image with different processor ranks blacked out.The iterations are being performed by the expected processors.

3.2 Convergence of Delta

Delta should converge towards the value of min with exponential decay. The first step to to test for convergence, this is shown in Figure 4.



(a) Delta convergence for all iterations.



(b) Delta convergence for last 1238 iterations.

Figure 4: Convergence of delta. Delta approaches the value of min, 0.1, asymptotically.

Delta converges towards 0.1, as required. This indicates that the image is approaching a "final state", or equilibrium where further iterations will no longer alter the image. The value of delta should decrease exponentially, and this is demonstrated in Figure 5.

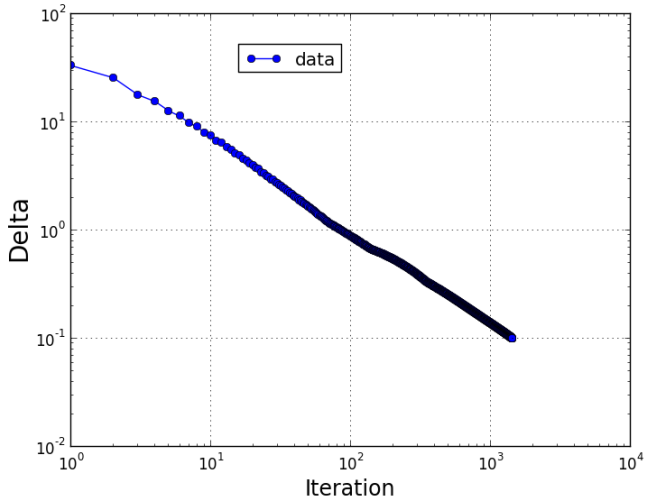


Figure 5: Log scale graph of delta decay. This approximately follows a straight line as required.

Figure 5 approximate follows as straight line. This indicates exponential decay. Exponential decay implies that the image is changing by a smaller margin with each iteration, reinforcing the idea of approaching a "final image", as iterations approach infinity.

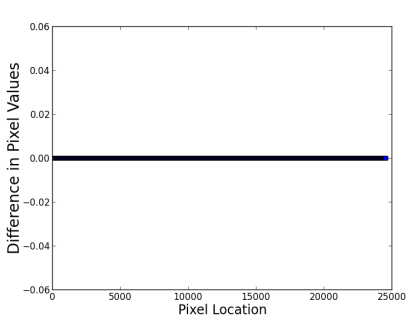
3.3 Pixel Difference

The image produced in our message passing program should obtain the exact same results as `imagenew.c`, the serial program within the coursework bundle. This can be tested mathematically and graphically. To do this graphically, we plot the difference between pixels values generated from the serial program, and plot them against the pixel values generated from our parallel program. Both programs ran for 500 iterations, and their differences are plotted in Figure 6.

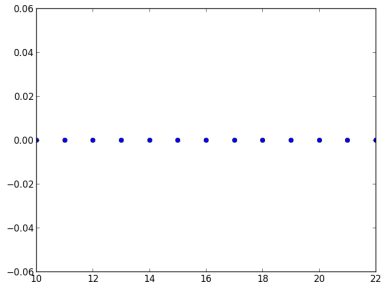
Perhaps a more elegant method is the mathematical approach. The 2D array is converted into a 1D array, and a cross product is performed. The result of this cross product in shown below.

```
y=[0, 0, 0, ..., 0]
```

A cross product was performed, and produced the zero vector. This means that the vectors are identical, and hence so are our images[5].



(a) Pixel difference parallel and serial programs.



(b) Closer inspection of the difference between serial a parallel programs.

Figure 6: Pixel difference between results obtained using a serial program and a parallel program.

4 Performance Testing

4.1 Execution Time

The execution time for each image are shown in Figure 7.

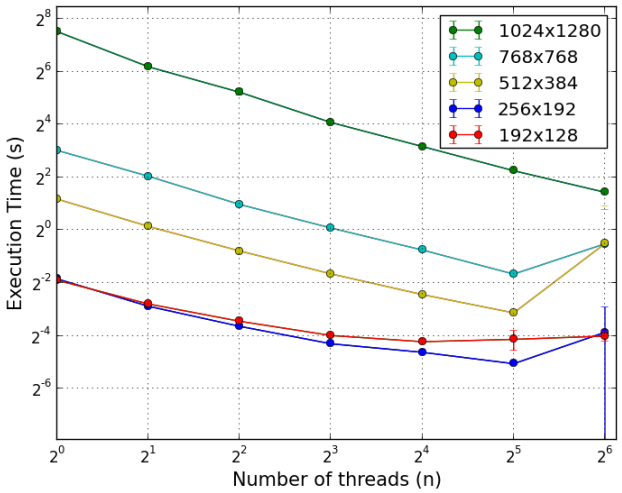


Figure 7: Execution times for each image, on n threads, on a standardised log scale, where n=1, 2, 4, 8, 16, 32, 64.

All images follow a generally decreasing trend, however, there is a spike in execution times for images on high numbers of processors. I1024 approximately follows a straight line, indicating exponential decay in execution time for all numbers of threads. All other images experience exponential decay up to a point. This point is generally met later by larger images, demonstrating the saturation point of each image.

The saturation point is reached when increasing the number of processors no longer speeds up a program. The parallel set up time, halo swapping and distribution of work takes more time than that which is saved by sharing out the iterations. Hence the program is slowed down, and high variability is induced. The large variability can be explained through thread spawning from non-blocking communications. These spawned threads, and now there are more threads than processors, and these have to be time sliced. Note that on multiple processors, I256 has lower execution times than I192, however they are extremely close.

4.2 Speedup For each Image

The speedup for each image is shown in Figure 8.

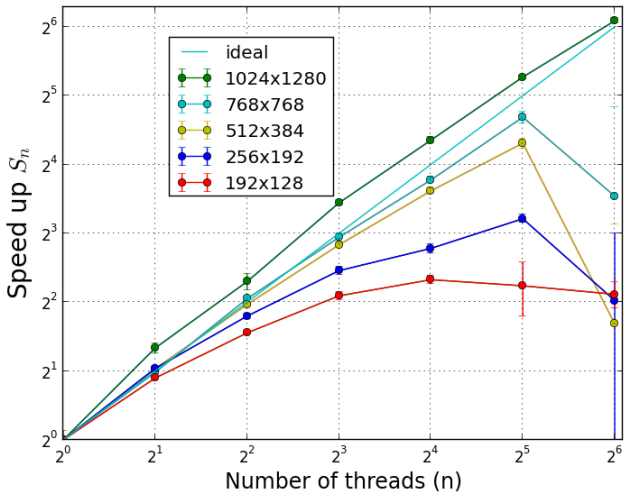


Figure 8: Speedup for each image, on n threads, on a standardised log scale, where n=1, 2, 4, 8, 16, 32, 64. Ideal speedup is plotted as $y = x$, note super linear speedup for I1024.

All threads initially speed up, and there is a generally increasing trend. The speedup is in order of image size, smallest speed up for the smallest image, and largest speedup for the largest image. Super-linear speed up is achieved for I1024. We expected this to be the best, but how can it be super linear? This can be attributed to the cache effect within computer hierarchies [6]. As the numbers of processors increase, so does the accumulated capacity of cache memory. More of the working set can now fit into cache, hence super-linear speedup.

I768, and I512 fall just below ideal, until too many processors are added, and performance is saturated. I256 and I192 start close to ideal, but fall away quickly. These images are too small for large numbers of processors processors to be worthwhile.

4.3 The Effect of Halo Swapping on Performance

Halo swapping is a contributor to the overhead involved in adding extra processors, as more processors are added, more halo swapping takes place. The effect halo swapping has on performance can be measure by timing with, and without halo swapping for I1024, I512, and I192 (biggest, medium and smallest). We expect the biggest effect to be on smaller images, where less calculations are taking place. Table 1 shows the ratio of timings with halo swapping:timings without halo swapping.

Threads (n)	I192 Ratio	I512 Ratio	I1024 Ratio
1	1.024419783	1.004476059	1.096740789
2	1.60411838	1.01196962	1.060793437
4	1.45741521	1.040838029	1.064988164
8	2.172285881	1.086914559	1.076168497
16	3.975375943	1.26500076	1.032147707
32	6.903760471	2.000470092	1.015125499
64	0.5134136581	0.7796482523	1.014143126

Table 1: Ratios demonstrating the effect of halo swapping on performance. Each image was run 5 times, and average values were taken to calculate the ratios.

Halo swaps are having a larger effect as the number of threads are increased, and as the image size gets smaller. This demonstrates why the speed up for smaller images declines more rapidly than for bigger images. The ratio for I192 and I512 on 64 cores breaks the increasing ratio trend. This is likely due to the high variability in their execution times, and these values can be ignored.

4.4 Delta and Average Calculations

It takes a significant time to calculate delta and the average, every iteration. The performance of the program can be improved by only calculating these every CRITFREQ iterations, where CRITFREQ is user defined. The draw back of this is that the program may do unnecessary extra iterations, so we do not want CRITFREQ to be large. Figure 9 shows the execution times for for CTITFREQ = 1, 10, 20 on I512.

There is a significant difference between CRITFREQ = 1, and CRITFREQ = 10. However, there is very little difference between CRITFREQ = 10, and CRITFREQ = 20. We dont really want to use values bigger than 20, because the number of iterations will be too far away from the "real" value, so for this reason we choose CRITFREQ to be 10 throughout the project. The end points have not factored into our analysis of CRITFREQ because of their high variability.

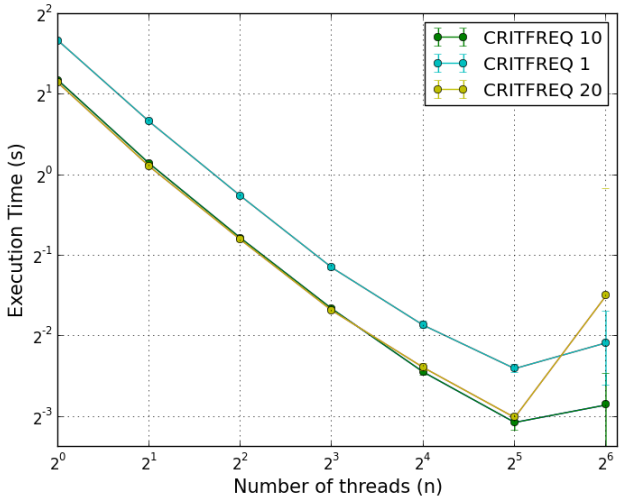


Figure 9: Execution times of I512 with different CRITFREQs on n threads, where n=1, 2, 4, 8, 16, 32, 64, CRITFREQ takes values 1, 10, 20. This was run 5 times, and average values were taken.

5 Conclusions

Message passing programs for two dimensional image decomposition requires Cartesian topologies to find ranks of neighbouring processes. These virtual topologies help with halo swapping, setting boundary conditions, and constructing simple (if inefficient) I/Os. Derived data types are used to send non-contiguous halo data for the vertical edges. Image clarity can be measured mathematically, and iterations should only take place if the image is not of sufficient clarity. However, these calculations are expensive, and performance can be improved by calculating every n iterations, where n is user defined.

Performance has a generally increasing trend as more processors are added, however the performance can get saturated for high numbers of processors, especially on small images. Smaller images achieve smaller speedups, because the overheads involved in splitting up the processors take up a larger proportion of time.

On the whole, this parallel program works well, and achieves good performance results. The images produced from the serial code are identical to those produced from the parallel code, as required. However, the program only works for thread numbers which are a power of two. Furthermore, the I/O implemented here is inelegant, and slow. Derived data types could have been used further to improve this part of the code.

References

[1] MPP Lectures: Lecture 9 - Virtual Topologies

[2] Rabenseifner, Rolf, Georg Hager, and Gabriele Jost. "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes." Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on. IEEE, 2009.

[3] Periodic boundary conditions, Wikipedia https://en.wikipedia.org/wiki/Periodic_boundary_conditio

[4] Effect of a Sawtooth Boundary on Couette Flow, Mateesu, Wang, Ribbons and Watson. Virgina Polytechnic University

[5] Cross product, Wikipedia https://en.wikipedia.org/wiki/Cross_product

[6] Speedup, Wikipedia https://en.wikipedia.org/wiki/Speedup