|epcc|

Message Passing Programming

B087928

December 4, 2015

1 Introduction

In many aspects of computing, especially computational physics, we are required to perform iterative calculations on a 1, 2 or 3 dimension lattice where each lattice point has a relationship with its nearest neighbours or other points throughout the lattice. In many algorithms we update each lattice point once per iteration. Often we have a lattice so large that we have very computationally heavy iterations. Therefore a major application of parallel programming is domain decomposition of a lattice such that the lattice points in each local domain may be iterated by a processor assigned to that domain.

There are two main difficulties in achieving this:

1. **Processor's Virtual Topology**: We must define where each processor's local domain is so that we can determine which lattice points it shall iterate over and have a way to determine each processor's location with respect to other processors.
2. **Processor Boundary points**: When we are using a nearest neighbour iteraction, as is the case in this model, each processor needs to be told by a neighbouring processor- in the virtual topology- the neighbours to its boundary elements. It will write these values into its halo data.

The second point is really an extension of the first and we shall see how we deal with these difficulties in section 2.

The lattice which is considered in this project is a 2 dimensional lattice of greyscale pixels. These lattice points, or pixels, have a value between 0 and 255 that represent a shade, 0 is black and 255 is white. Thus we can display the lattice as an image. We considered 5 different images of different lattice sizes. The size of the lattice and the corresponding images may be found in appendix. A simple edge detection algorithm has been applied to produce an 'edge image' for each image, see appendix. This is produced by simply setting each pixel value to the difference between the sum of the four nearest neighbours and 4 times it's own value.

This is simple and fast as we only need to update each lattice point once. We are required to do the reverse operation and recover the original image, as close to it as possible, from these edge images. This is an iterative process and it is only possible to get infinitesimally close to the correct image, as shall be shown in section 4. We refer to this final image as the 'processed image.'

This report focuses on the parallelisation of a serial solution to this problem.

2 Code

2.1 Virtual Topology

We use a 2 dimensional domain decomposition. Thus to define our processor topology we assign an i and j component to each processor to relate the processor's rank to its local domain. We refer to these coordinates as ranki and rankj. We use the MPI function *MPI Dims create* to define sizei and sizej. *MPI Cart create* was used to create a communicator *'grid comm'* with periodic boundary conditions in the x direction.

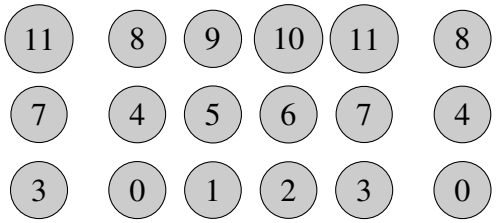


Figure 1: Virtual Topology of the processors. Numbers represent the rank of a processor. Number of cores ($size$) = 12. $sizei = 4$, $sizej = 3$. The columns at each end represent the peridoic boundaries in the x direction.

In figure 1, rank 0 will have coordinates (0, 0), rank 1 will have coordinates (1, 0), and so on. We define an array with two elements called *coords*. We use the MPI function *MPI Cart coords* to set the value of coords to the i and j coordinate of each processor respectively. We set ranki and rankj to to these values for the sake of clarity in subsequent use.

Since we are interested in a lattice point nearest neighbour interaction virtually adjacent processes must exchange halo data. Therefore it was necessary to define nearest neighbour processes. Therefore we use the MPI function *MPI Cart shift* to assign the variables rankleft, rankright, rankup and rankdown to the rank of the nearest neighbours as defined in the grid comm communicator.

We can verify that the virtual topology is of the form shown in figure 1 by printing each process' rank and its values of rankleft, rankright, rankup and rankdown. This was very important to know as we needed to ensure that we are splitting up data amongst processes following the same topology.

2.2 Data Topology

We must be careful when we decide how data is distributed on to the processors. When we perform halo swaps we must be confident that the data being exchanged is supposed to be adjacent and that we have not put data that's supposed to be on the bottom left of the image in a processor that is not bottom left in our virtual topology.

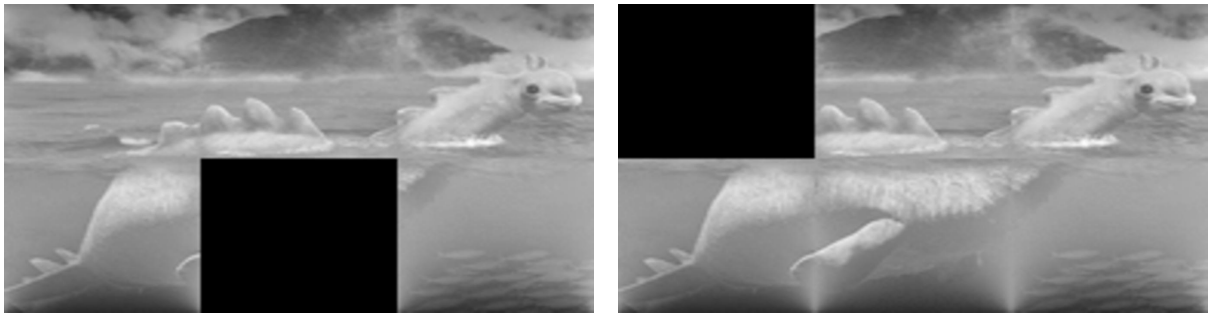
M_p and N_p are the number of pixels in the x and y direction respectively on each process. These will have a lesser value on the rightmost, and/or uppermost, processes (than the other processes) if the number of pixels in that direction is not divisible by sizei or sizej respectively.

Each processor reads the whole data into an array masterbuf of size $M \times N$. We define starti, endi, startj and endj as the position (in the masterbuf) of the leftmost, rightmost, lowermost and uppermost element on each process. Each process has a copy of masterbuf and uses a function packbuf to write the appropriate values of masterbuf, defined by the data topology, to its buf array. A snippet of the packbuf function may be seen below.

```
x = 0;
for (i=starti; i<endi; i++){
   y = 0;
   for (j=startj; j < endj; j++){
      buf [x][y] = masterbuf[i][j];
      y++;
   }
   x++;
}
```

Data was copied from buf back to masterbuf in a similar way once all iterations had been completed.

As a conclusive test that the data topology was consistent with the virtual topology the buf array elements were set to zero on a particular process. This made it obvious to what location on the image the data on that process belonged. This was compared to the virtual topology. An example of one of these test can be seen in figure 2 where we have created two images using 6 processors without employing halo swaps. In each image a different process has set its buf array to zero.



(a) Data on rank 1 set to zero.

(b) Data on rank 3 set to zero.

Figure 2: Data Topology Test. Image size: 192x128 pixels. No Halo swaps. Number of processors = 6.

Since halo swaps were not used for these images we can see clearly distinctive lines between the domains of different processes. Which can be compared to figure 7b in the appendix.

2.3 Halo Swaps and Derived Data Types

To understand why we require derived data types we must consider how 2 dimensional arrays are stored in memory and how we use *MPI Issend* and *MPI Irecv* to send and receive data.

When performing halo swaps horizontally the data we wish to send is contiguous in memory. Thus we can specifying the address of the first element to send and a count of N_p with a type MPI float which sends N_p contiguous elements. For vertical Halo swaps however the data is not contiguous in memory. Thus we must define a new datatype.

We define the datatype noncontiguous with a count of M_p, a block size of 1, a block spacing of $N_p + 2$ and a type MPI float. This produces a vector of M_p floats.

Now we send data vertically by specifying the address of the first element to send and a count of 1 with a type noncontiguous. This sends M_p elements starting with the element at the address we specify and jumping $N_p + 2$ addresses in memory for the next element.

3 Stopping Criteria and Test Frequency

During each iteration, the algorithm changes the greyscale values of the pixels, which asymptotically approaches a fixed value, as will be shown in section 4. We thus define a stopping

criteria based on the 'closeness' we get to this fixed value. We call this the tolerance.

Calculating the global max pixel change every iteration would be wasteful. However, if we don't check it often enough we may do many more iterations than required. To determine an appropriate number of iterations we calculated performance for different test frequencies. We calculate the performance 100 times for each test frequency and used the average as our expectation value for the performance using that test frequency. We used the standard deviation of these values to produce error bars. This was tested using the 192x128 image and a tolerance of 0.1. Figure 3 shows the runtimes for different values of TESTFREQ. Note here that test frequency is actually the number of iterations between tests.

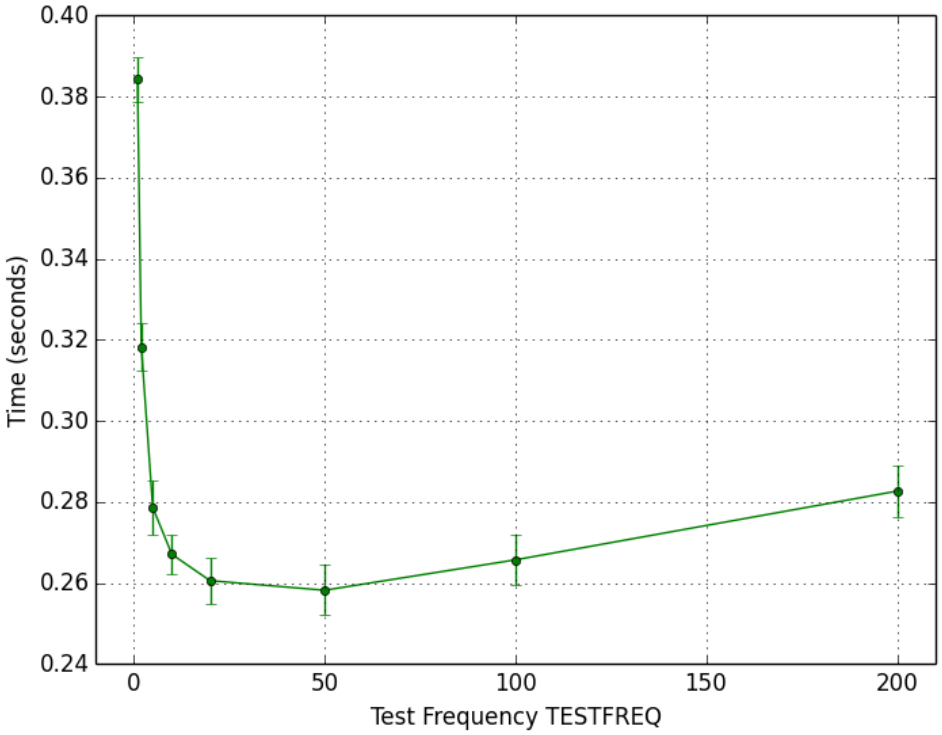


Figure 3: Runtime against value of TESTFREQ

This is not a parallel dependent problem as every processor must do their own local sum. Thus we ran on just one processor.

The speed will change depending on whether the last required iteration is close to the next test. Therefore we expect that a test frequency of 100 or 200 will certainly be quicker for a certain tolerance. Judging from figure 3 we can conclude that a test frequency of 50 is certainly a good choice and therefore we set this to be the value for all subsequent calculations.

4 Testing the Parallel Code

Examining the images produced by the program was a useful way to regularly check if the parallel code was working correctly. However more reliable tests were required to produce convincing and conclusive evidence that the parallel code produces exactly the same result as the serial code.

To produce figure 4 we wrote the values of finalbuf, from the parallel code, and buf, from the serial code, into two respective 1 dimensional arrays and plot them against each other. We can see that each point lies along the line $y = x$ thus confirming that the image created by the parallel code (with 64 cores) and serial codes were identical. A simple subtraction of each element yielded an array of zeros and verified the same result.

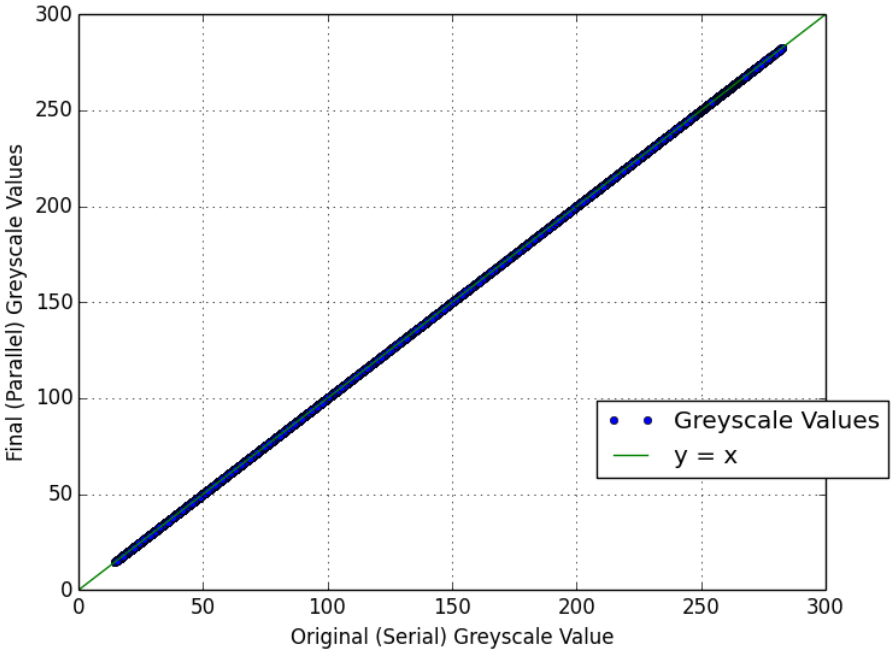


Figure 4: Greyscale values produced from 10^5 iterations of the parallel code against greyscale values produced from 10^5 iterations of the original serial code. Image size: 512x384. Number of cores used for parallel code = 64

With correctly implemented boundary conditions and halo swapping the parallel code should perform the exact same calculations as the serial code. We thus expect that after n iterations finalbuf (or masterbuf) in the parallel code and buf in the serial code should have the the same floating point values for corresponding array elements.

A further test was conducted to demonstrate that the greyscale values are approaching an equilibrium state. We changed the test frequency to produce a global max pixel change every iteration which we recoreded. This is the maximum change in greyscale pixel value from one iteration to the next. A straight line on a log-log plot of max pixel change per iteration against iteration number, as seen in figure 5, demonstrated that global max pixel change per iteration is exponentially decreasing and therefore pixel average is approaching an asymptotically fixed value.

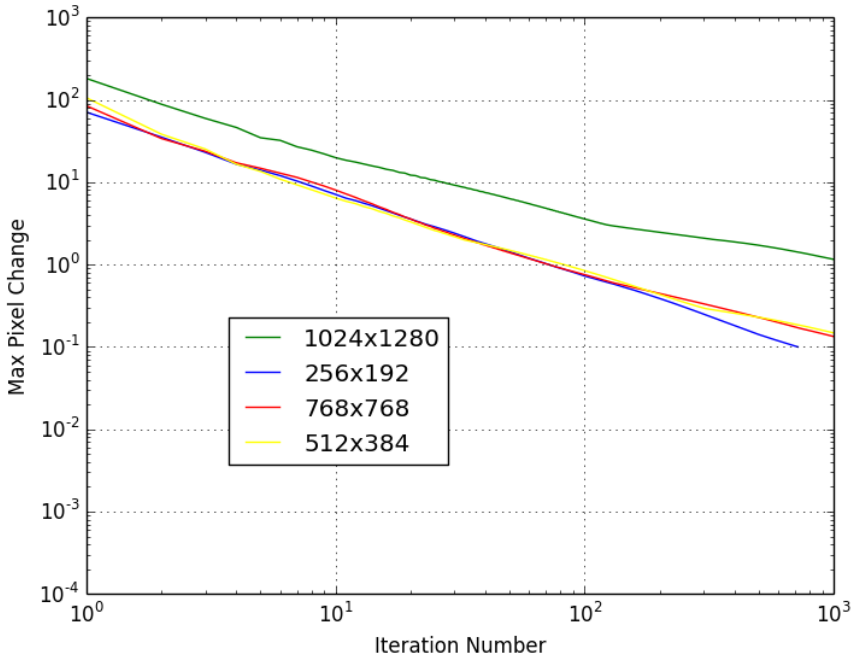


Figure 5: log of Max pixel change per iteration against the log of iteration number for the image sizes labelled above.

We then ran the parallel code on number of cores between 1 and 64 with a tolerance of 0.1. We confirmed that the global max change of pixel value and the pixel average were the same in each case when the image was produced hence confirming that the code was consistant on any number of cores. An exception to this was for the smallest image (192x128). Our program terminated with the error 'Bad number of processors!' for any prime number of processors greater than 17 that did not divide into 192. This is because we terminate the program if the last processor in i or j direction has an iteration endpoint less than its iteration startpoint.

5 Speed up

For an accurate speed up it is necessary to maintain a constant test frequency and tolerance. In this way we ensure that we have the same number of iterations for each run as well as the same number of more computationally expensive iterations. From our results in section 3 we decided that a test frequency of every 50th iteration gave a good performance. We also used a tolerance of 0.1.

We were not interested in timing tasks we only did once, such as defining the topology, and the arrays, or gathering all the data together at the end since they are irrelevant to parallelisation. To get an accurate measure of speedup we only time the iterative process.

In figure 6 we can see the speedup found using the parallel code on each image plotted against an ideal linear speedup. We repeated the calculation of speed at each number of cores 10 times, taking the average to be our expected value and used the standard deviation of the values as error bars. Since the smallest image was very computationally inexpensive, we performed the calculation of speed 100 times per core number.

We can see that we have a much better speedup for the larger images as we get to higher number of cores. This is easily explained: The time taken for halo swaps increase as we increase the number of cores as the amount of data being sent and received increases and we must wait until all cores have received their halo data to ensure that our updates are correct. This extra cost is compensated by the increase in speed of the update for large images, however, on small images the time taken for the update may be insignificant compared to the extra halo swap expense.

We can see a superlinear speedup for the largest image. This is likely because the smaller arrays used on each processor when we use more cores is more appropriate for cache sizes.

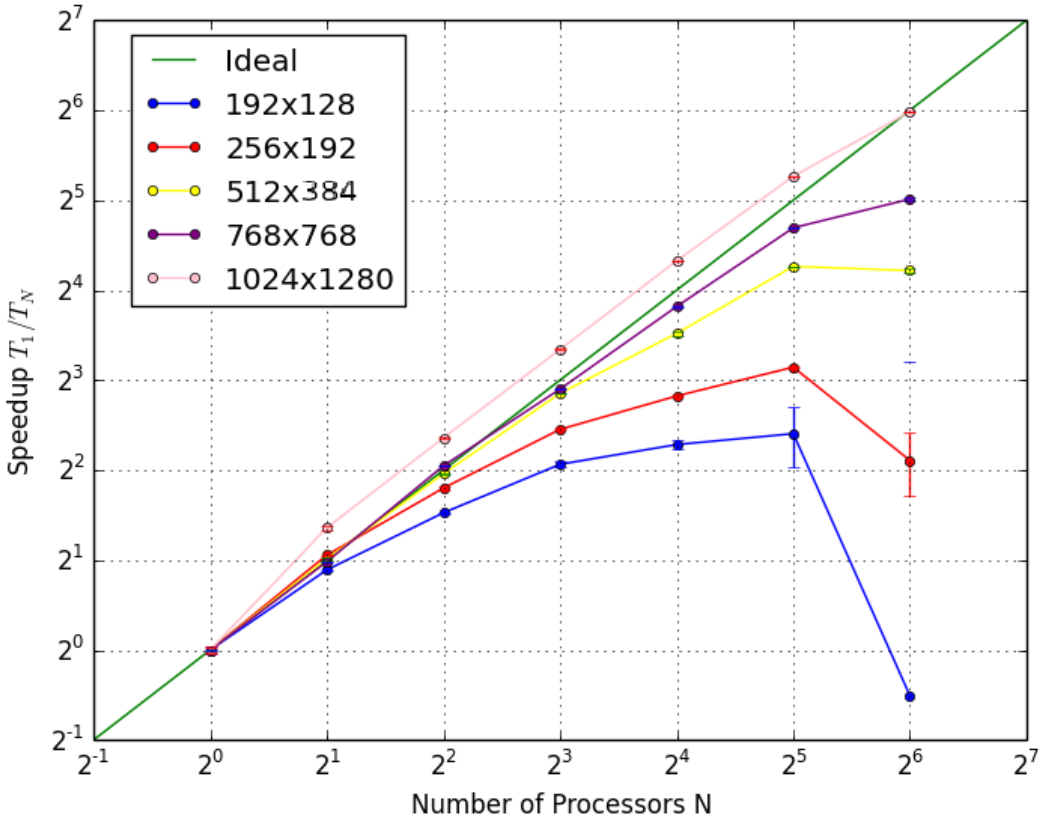


Figure 6: Code speedup for each image against ideal linear speedup.

6 Conclusions

When we consider a 2 dimensional domain decomposition we must first define a virtual processor topology and ensure that it is consistent with how we distribute the data amongst the processors.

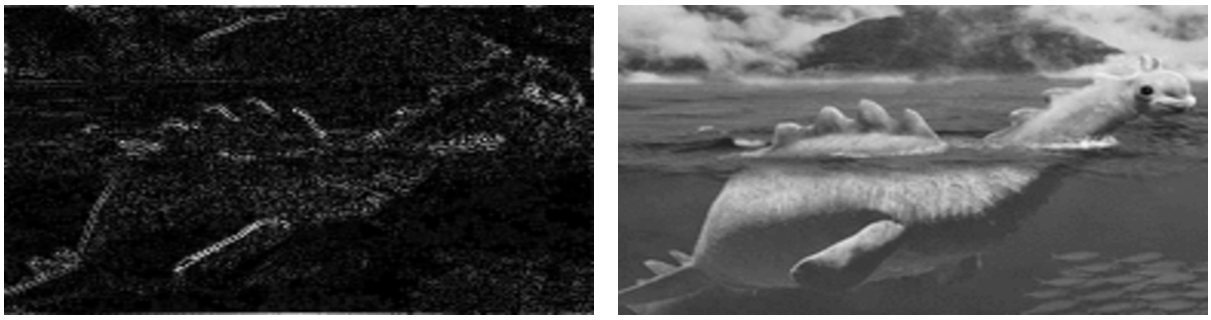
It is more straightforward to halo swap when we are sending/receiving contiguous data. If we wish to send non contiguous data we must derive a new datatype. This is essential if we use 2 dimensional domain decomposition.

When we use halo swaps for domain decomposition we are using the exact algorithm as we would be for a serial code. Thus we can verify that our parallel code is working correctly if it give the exact same result as the serial code.

If we are using an iterative process then we can check that we are asymptotically approaching a fixed solution by checking that the plot of the log of the maximum lattice point change from one iteration to the next against the log of the number of iterations gives a straight line.

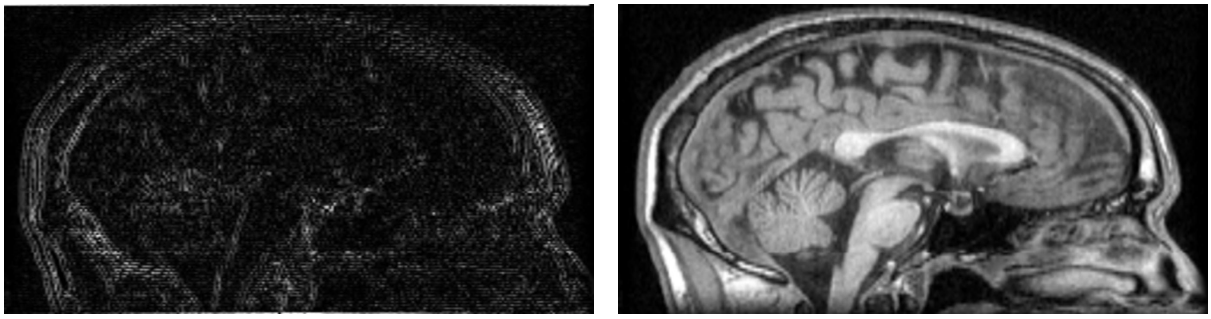
Non-blocking communication is needed for halo swaps so that no processes stall, however all halo swaps must be completed before we begin calculations. This means that halo swapping is costly in the runtime of our program: more processors does not necessarily imply a speedup as the additional halo swaps may be too expensive.

Appendix



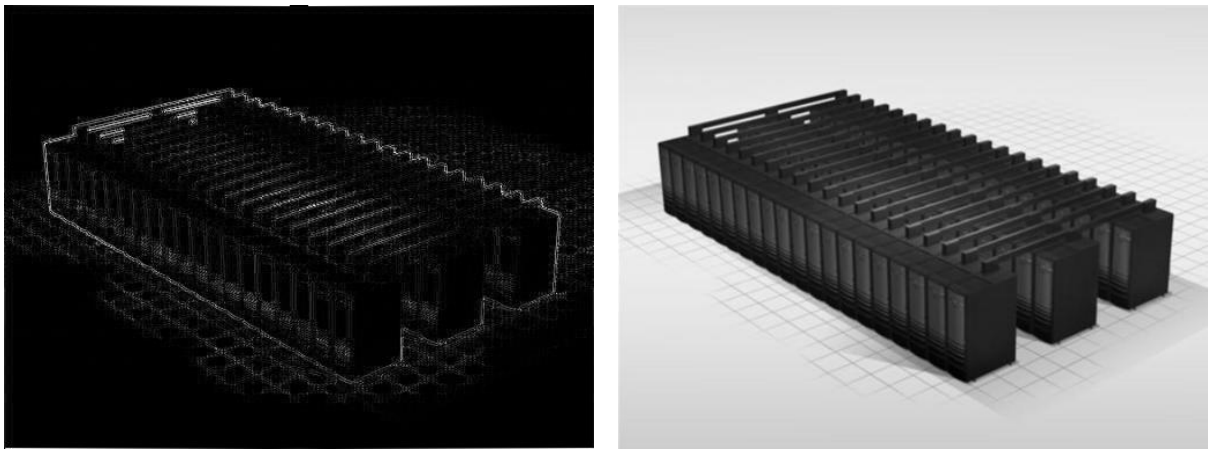
(a) Edge Image. (b) Processed image

Figure 7: Image size: 192x128 pixels. Original edge image and the result after applying the program to it. Tolerance = 0.01. Number of cores used: 64.



(a) Edge Image. (b) Processed image

Figure 8: Image size: 256x192 pixels. Original edge image and the result after applying the program to it. Tolerance = 0.01. Number of cores used: 64.

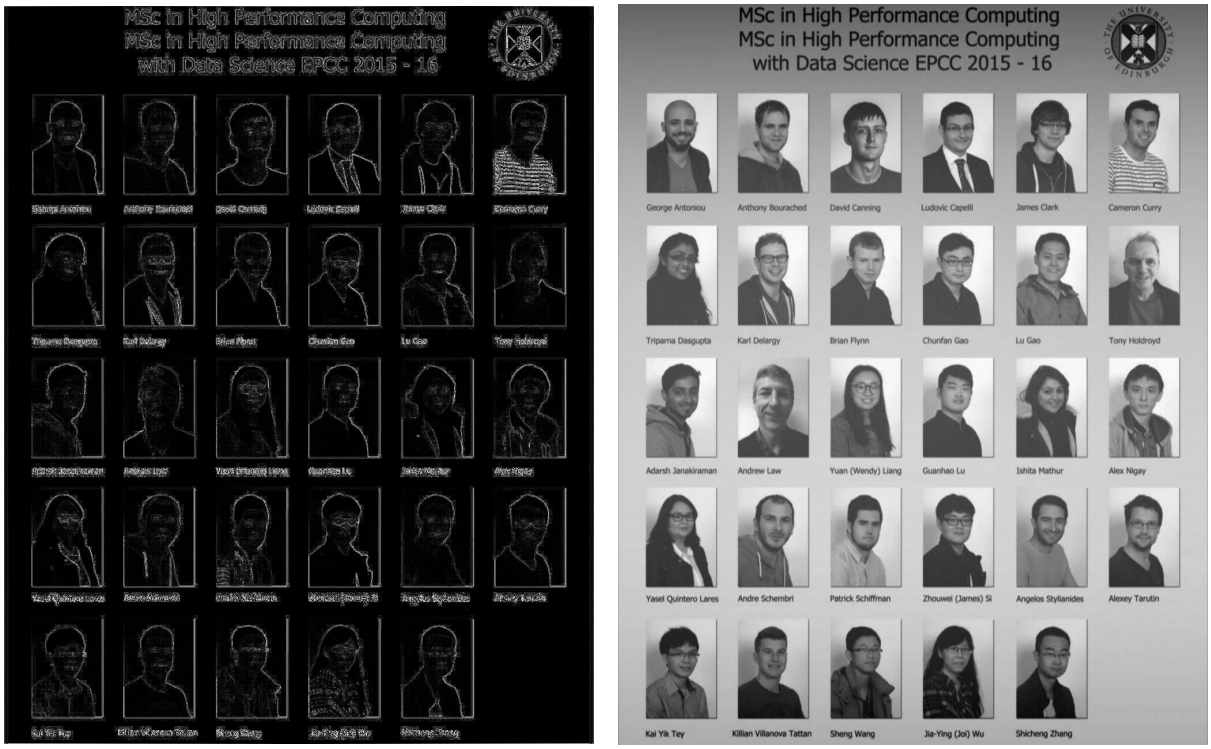


(a) Edge Image. (b) Processed image

Figure 9: Image size: 512x384 pixels. Original edge image and the result after applying the program to it. Tolerance = 0.01. Number of cores used: 64.

(a) Edge Image. (b) Processed image

Figure 10: Image size: 768x768 pixels. Original edge image and the result after applying the program to it. Tolerance = 0.01. Number of cores used: 64.



(a) Edge Image.

(b) Processed image

Figure 11: Image size: 1024x1028 pixels. Original edge image and the result after applying the program to it. Tolerance = 0.01. Number of cores used: 64.