Roshan Pasupathy
26593165

Message Passing Programming for Image Reconstruction

Introduction

Parallel computing provides significant performance improvements when applied to image processing. In this report a message passing parallel program which reconstructs an image from its edge image is discussed. The image is decomposed into a 2D lattice according to the number of processes available and each process then reconstructs its subsection of the image.
Image reconstruction is carried out iteratively over the entire image which makes this problem ideal for parallelisation.

Code Description

1. Domain decomposition and Input Output

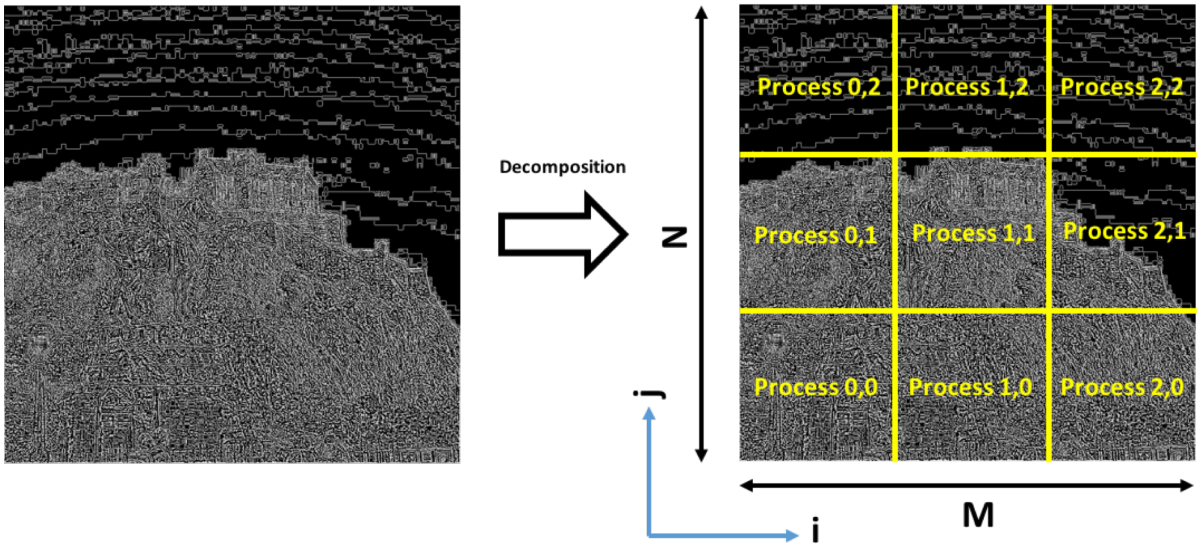

Figure 1: Edge input image (left) and virtual topology of processes on decomposed image (right).

As seen in figure 1 the image is decomposed into a process grid such that the number of processes along the `N` side of the image (`Pn`) exactly divide `N`. If the number of processes along the `M` direction of the image (`Pm`) do not exactly divide `M`, then the processes in the rightmost column evaluate the remaining `i` columns of pixels in their respective subsections.

A new communicator `cartcomm` is mapped onto the 2D topology using:
`MPI_Cart_create( …, ndims=2, dims=(Pm,Pn), periods=(0,0), …)`

The `root` process (process with coordinates 0,0) reads the edge image from file into a `MxN` buffer named `masterbuff`. Sub sections of `masterbuff` are sent to the processes according to their coordinates in the grid defined in `cartcomm`. Scattering

data to the processes is challenging due to the fact that the subsections distributed are not contiguous in memory.

The size of a subsection of `masterbuff` for each process is `Mp` x `Np`. Thus a subarray datatype with the subsection `Mp` x `Np` as a subarray of an `M` x `N` array is defined. However as seen in figure 2, the desired starting point of a subarray (shown in blue) now lies within the extent of the previous subsection.

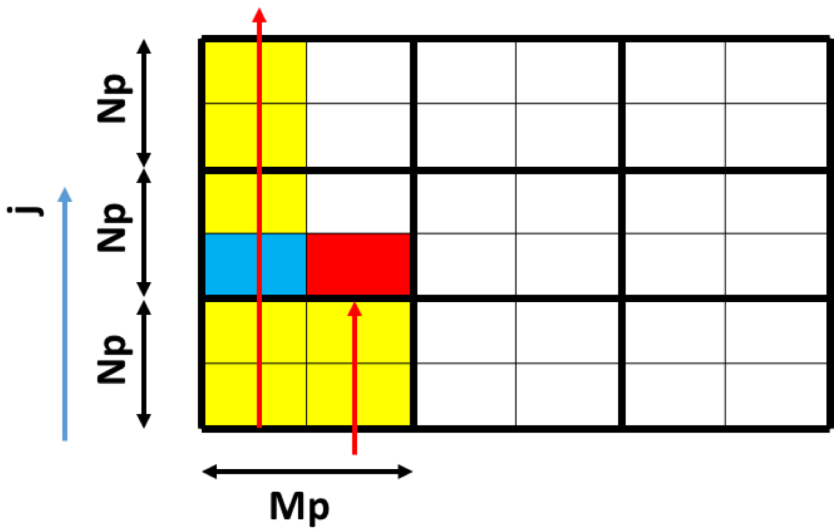

Figure 2: Extent of the subarray for the bottom left section of a 6 x 6 image.

In figure 2, a 6 x 6 float image is considered. Each sub section has 2 by 2 floats. The image data is contiguous in the `j` direction. However, `MPI_Scatterv()` routine counts displacements in terms of multiples of the extent of the send type – 8 floats in this case. This is rectified by resizing the extent of the subarray to the number of bytes occupied by `Np` float values using the `MPI_Type_create_resized()` routine.

Also if `Pm` doesn't exactly divide `M`, the value of the `Mp` for the rightmost column of processes will be different. Hence a different datatype will be needed for the rightmost column. In this situation, the routine `MPI_Alltoallw()` (The most general form of the collective routines scatter and gather) allows for sending different datatypes to different processes. The same process is repeated while gathering data from the processes.

A strided vector type with count = `Mp`, blocklength = `Np` and stride = `Np + 2` is used for ignoring halos while reading and writing data to and from the buffers local to each process.

2. Iterative reconstruction of subsection of image

Three arrays are need to iteratively reconstruct the image.

`edge[Mp+2][Np+2]` which contains a sub section of the edge image

`old[Mp+2][Np+2]` which contains data from the previous iteration

`new[Mp+2][Np+2]` which contains data from the current iteration
In each of these arrays, the first and the last rows and columns contain halo values. All values of `old` and `new` are initialised to 255.

```
Looping over i = 1 to Mp; j = 1 to Np
new[i][j] = 0.25 * (old[i-1][j] + old[i+1][j] + old[i][j-1] +
                    old[i][j+1] - edge[i][j])                    (1)
```

At the end of each loop the `old` array is set equal to `new`. Instead of accessing each value of `new` and updating it to `old` a more efficient approach is to use pointers. Being static arrays the addresses of `old` and `new` cannot be changed. Hence pointers `oldp` and `newp` are used.

```
(* oldp)[Np+2] = old; (* newp)[Np +2] = new;
```

The image is reconstructed using the pointers and the `old` and `new` arrays effectively serve as allocated memory for these pointers. At the end of each iteration the addresses of `oldp` and `newp` are swapped.

In a 2D decomposition each process (except the ones at the boundaries of the image) has to both send and receive 2 rows and 2 columns of halos in every iteration as shown in figure 3.

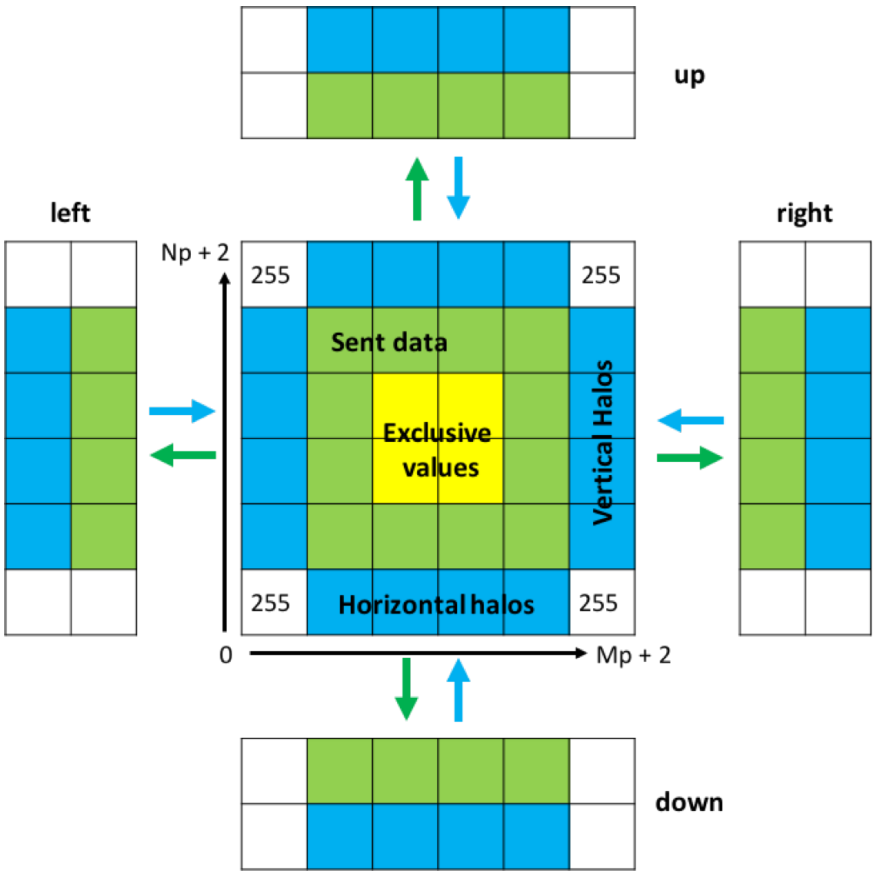

Figure 3: Communication of data with neighbouring processes.

In `cartcomm`, the ranks of the `left`, `right`, `up` and `down` processes are easily found using the `MPI_Cart_shift()` routines with dimension 0 for `left` and `right` processes and 1 for `up` and `down` processes.

The vertical halos are contiguous in memory and hence can the address of the first value of the halo can be sent a count of `Np`. The horizontal halos are not contiguous in memory however they are regularly spaced with a stride of `Np + 2`. Hence a vector datatype is defined for sending horizontal halos using the MPI routine:
`MPI_Type_vector(count = Mp, blocklen = 1, stride = Np + 2, …)`

Non-blocking communication is used to mitigate the communication overhead. The steps executed in each iteration of the image reconstructions are shown in figure 4.

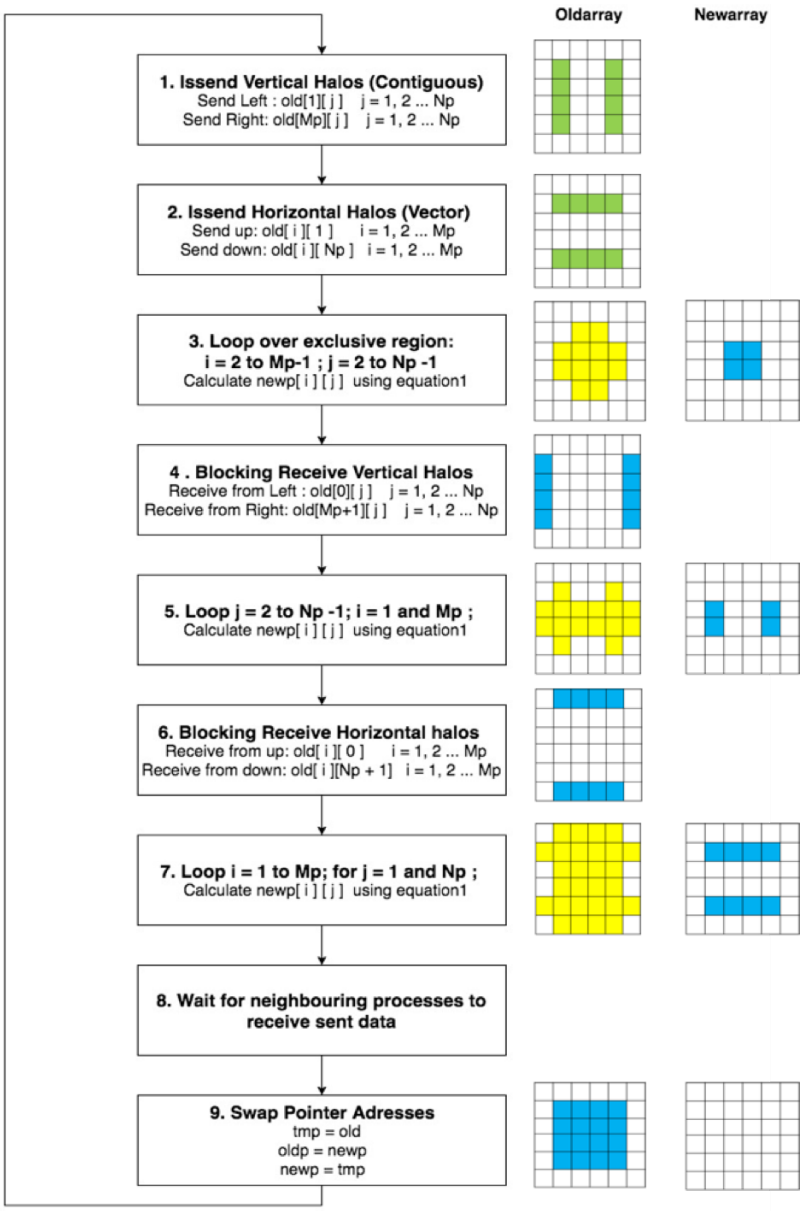

Figure 4: Steps executed in each iteration. Colours indicate operations being carried out on old and new. green: read and send, yellow: read, blue: write.

Thus while data is being sent from neighbouring processes, a process evaluates its exclusive region and the cost of data transfer between processes is not felt.
Processes at the boundaries of the image will have `MPI_NULL_PROC` neighbours and send operations to these null processes complete immediately.

3. Checking for Completion

If the maximum difference between `newp` and `oldp` at every point in the reconstructed image is less than a certain criterion value, further iterations will not be necessary and the loop can be terminated by adding a `!stopflag` check to the loop execution condition.

A criterion value of 0.03 was chosen and the check is performed with an iteration period of 60 iterations (See Appendix A and B for derivation of these values).
Thus in iterations (`it`) which satisfy `it%60 == 0`, the maximum difference `max_diff` is found by each process in its sub-section. `MPI_Allreduce` reduces all the local maximums to a single global maximum:

```
MPI_AllReduce(send=MPI_IN_PLACE,recv=max_diff,…,OP=MPI_MAX,…)
```

If `max_diff` is now less than the stopping criteria, `stopflag` is set to 1 which terminates the loop.

4. Average of pixel values in the reconstructed region

Every 200 iterations the `root` process prints out the average value of the pixels in the reconstructed image. Since this operation does not affect the final result of the reconstruction and might cause unnecessary synchronisation, the non-blocking reduce `MPI_Ireduce(…, MPI_SUM, …)` routine is used. Each process calculates the sum of the pixel values in its sub-section and sends it to `root` via this routine and wait for the message to be received only when it has to send a new value. For the `root` process, the wait call is issued before printing out the average as the `global_sum` divided by (`M*N`).

Testing

1. Correctness

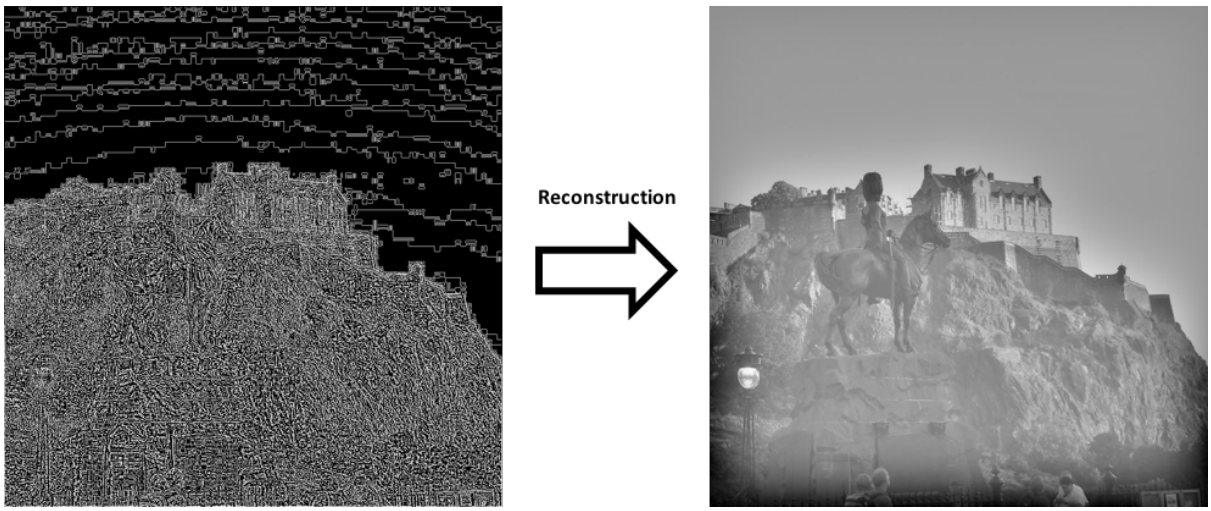

Figure 5: Reconstructed 768 x 768 image over 5000 iterations.

The program described in the previous sections was run on 9 processes to obtain the result shown in figure 5 (termination criteria not reached).

Figure 6 was by obtained by modifying the code such that the odd ranked processes do not send their reconstructed sub sections to `root` when the gather routine is called. Thus it demonstrates that all processes were indeed working in parallel to reconstruct the image i.e. the image was divided into 9 subsections (3 x 3).

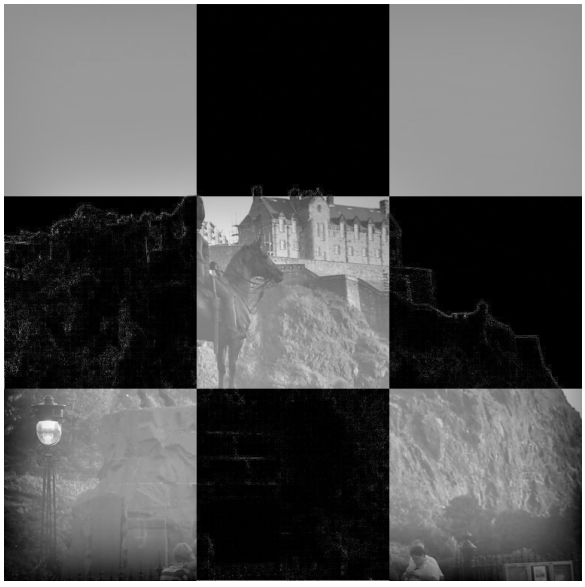

Figure 6: Image obtained when odd ranked processes do send their sub-section data in the gather routine.

Numerical tests were carried out to ensure numerical correctness of the code as well. The output of the code is compared to the output image (reference image) generated by the reference serial code provided for this exercise. The comparison can be either

comprehensive where all points are tested or random where 1000 random points on the image are tested. The type of test is determined by the `test_all` flag passed to the `test_result` function in `test_functions.c`. The error tolerance is set to 1 to account for floating point rounding errors. Figure 7 shows the steps in the `test_result()` function.

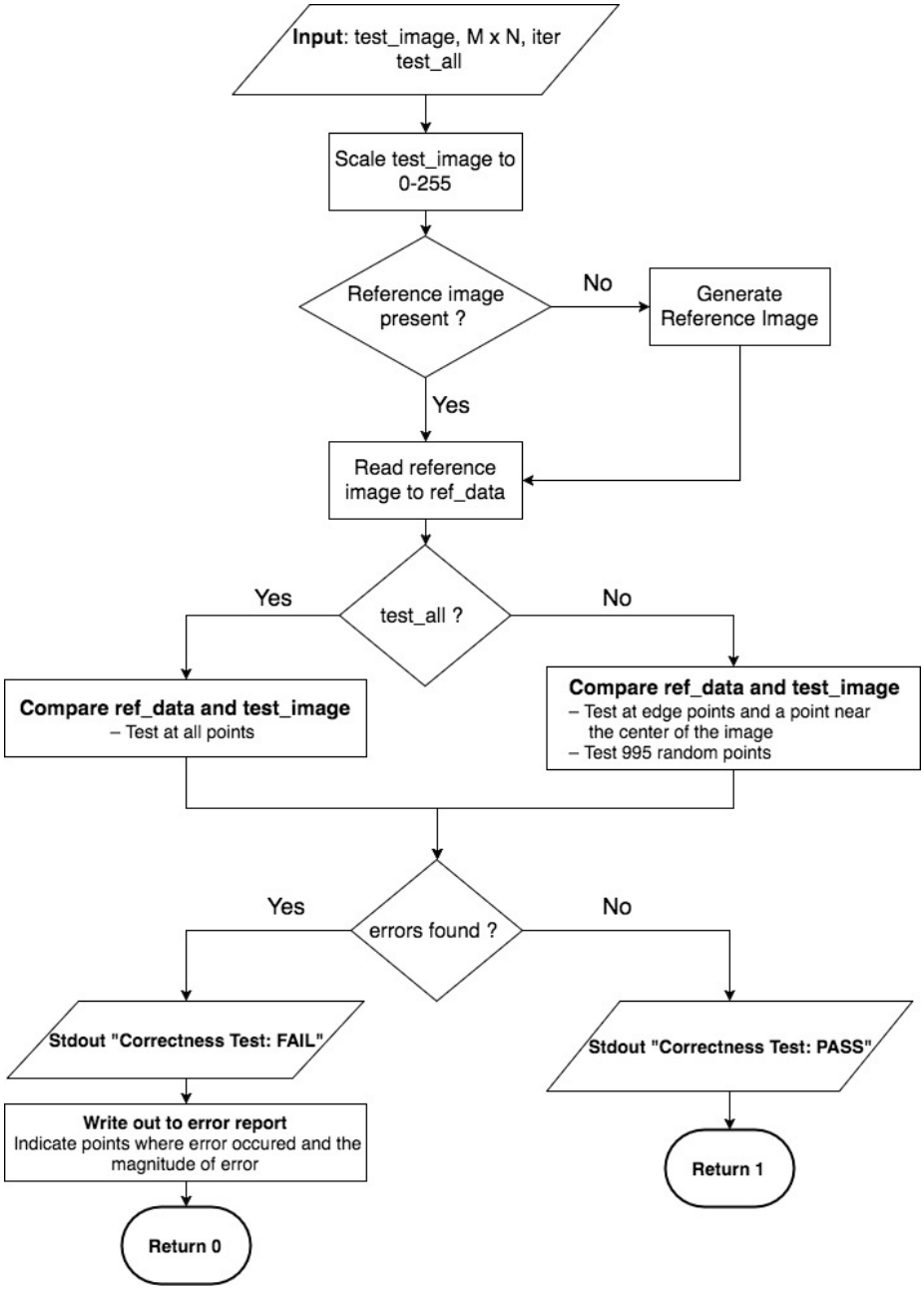

Figure 7: Routine for testing for correctness

Edge images of sizes 192 x 128, 256 x 192 and 512 x 384 were also reconstructed. In all cases, the reconstructed image passed the comprehensive correctness test. However, only for the 256 x 192 image, the termination criteria was satisfied at 2821 iterations indicating that a smaller larger criteria may be needed for early termination at the cost of accuracy.

2. Performance test

The execution time of the iterations was measured to estimate the performance of the code since the reconstruction is where the reconstruction is carried out and input output was not the focus.

For a given image the problem size is = `M x N`. Thus since the problem size is fixed for the reconstruction of an image, this algorithm is strongly scaled. For a strongly scaled problem, the ideal speed up vs number of processors graph is linear. Figure 8 and 9 illustrate this parallel performance of the code for various problem sizes.

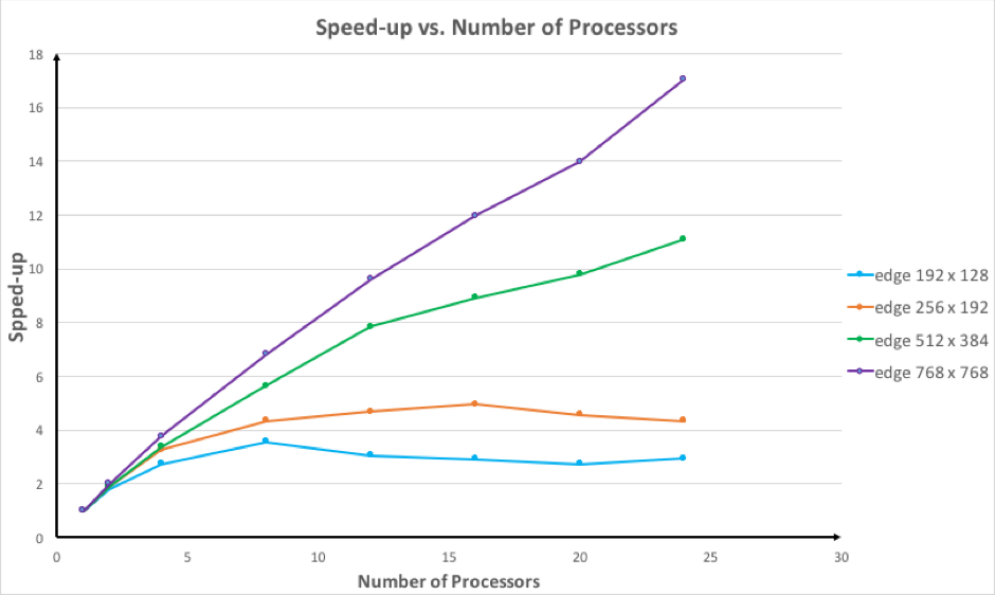

Figure 8: Per iteration speed-up vs. number of processors for different problem sizes.

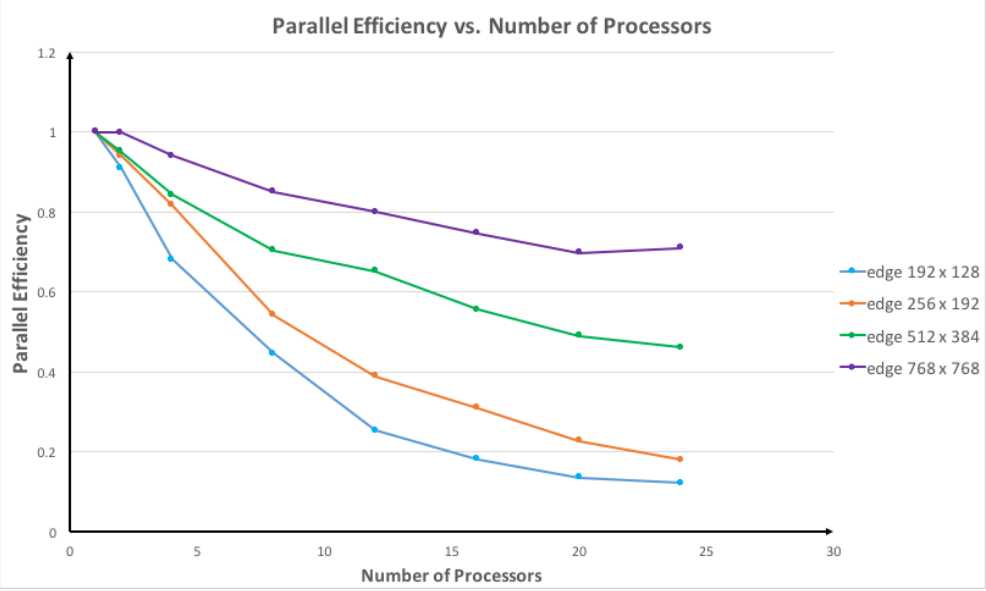

Figure 9: Parallel efficiency vs. number of processes for different problem sizes

Strong scaling speed-up = T(`M x N`, 1)/T(`M x N`, `P`)

Strong scaling efficiency = Speed-up/`P`

In figure 8, the speed-up vs number of processors graph is almost linear for large problem sizes such as 768 x 768 for which the parallel efficiency is also more than 70%. For smaller problem sizes, the speed-up stagnates or increases at a slower rate with a drastic decrease in parallel efficiency as the number of processes decrease.

As the problem size decreases, the size of the exclusive set of iterations (iterations which don't depend on halo values), to be carried out by each process, decreases. A process completes its exclusive iterations while the halo values sent by its neighbours reach it via the communication pipes. Smaller exclusive sets result in the communication overhead becoming more apparent. For example, when `Mp` = 2 or `Np` = 2 pixels, there are no iterations in the exclusive set and a process has to wait for the halo values from its neighbours to arrive in order to perform calculations.

To ensure that the code consistently provides the same results, it was run 8 times each for reconstructing edge768x768.pgm on 4, 12 and 24 processes. The results are illustrated in figure 10 below. See Appendix C for table of results.

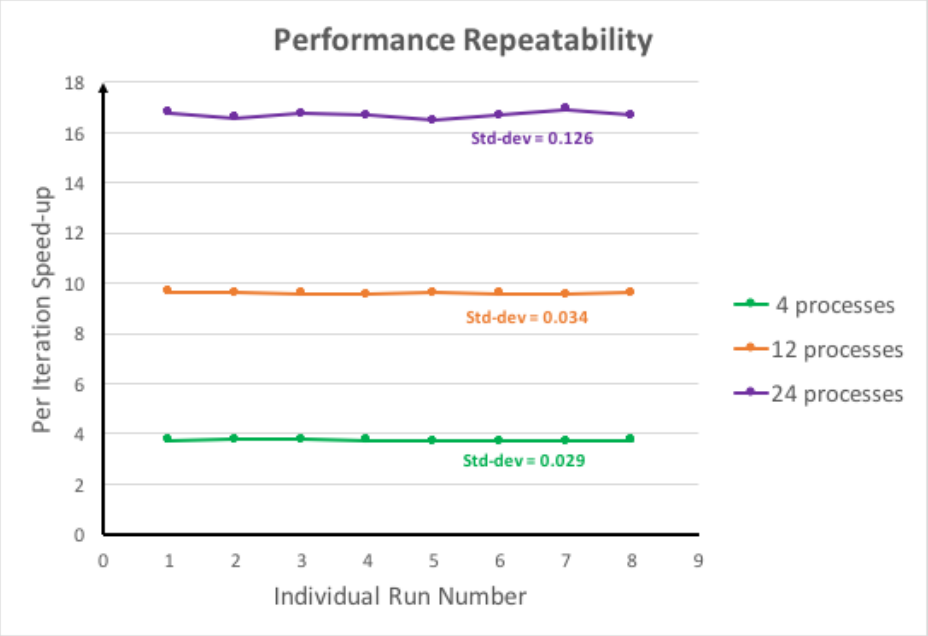

Figure 10: Performance repeatability over 8 runs of parallelised code

Conclusion

MPI derived datatypes, virtual topologies and communication routines greatly simplify writing parallel programs. The code developed using these routines accurately reconstructs an image from its edge image and this was demonstrated by qualitative visual analysis as well as comprehensive quantitative testing.
It performs almost ideally for reconstructing large images and the high parallel efficiency justifies the use of computational resources for parallelisation.

Appendix

Appendix A: Calculation of termination criteria

All values in the reconstructed image buffer are scaled to integers between 0 to 255 as follows:

$$pix_{i,j,it} = \left\lfloor \left(255 * \frac{\left| val_{i,j,it} \right| - val_{min,it}}{val_{max,it} - val_{min,it}} \right) + 0.5 \right\rfloor$$

Let's call the termination criteria c_{term}. The aim is to roughly quantify the error range of $pix_{i,j}$ in terms of c_{term}.

When the maximum value over all values of $i,\ j\ \left| val_{i,j,it} - val_{i,j,it-1} \right| < c_{term}$, the loop will terminate. If subsequent iterations were to be carried out the maximum difference will be less than c_{term} hence the maximum possible error between the current $val_{i,j,it}$ and its convergence value ($val_{i,j,conv}$) is proportional to c_{term}.

Due to the complexity in determining the exact difference, for the purpose of our calculation we will assume that the maximum value error between the convergence iteration and the current iteration = c_{term}.

The corresponding pixel error = pix_{err} . The greatest possible error in pixel value, pix_{err} is as follows (subscript $conv$ indicates convergence and it indicates last iteration):

Case 1: When $pix_{i,j,it} > pix_{i,j,\ conv}$

$val_{i,j,conv} = val_{i,j,it} - c_{term}$

$val_{min,conv} = val_{min,it} + c_{term}$

$val_{max,conv} = val_{max,it} + c_{term}$

$$pix_{err} \simeq 255 * \frac{2c_{term}}{val_{max,it} - val_{min,it}}$$

Case 2: When $pix_{i,j,it} < pix_{i,j,\ conv}$

$val_{i,j,conv} = val_{i,j,it} + c_{term}$

$val_{min,conv} = val_{min,it} - c_{term}$

$val_{max,conv} = val_{max,it} - c_{term}$

$$pix_{err} \simeq 255 * \frac{2c_{term}}{val_{max,it} - val_{min,it}}$$

The minimum value of $val_{max,it} - val_{min,it}$ is 1 in the worst possible scenario

Hence

$$pix_{err} \leq 510 * c_{term}$$

With a $c_{term} = 0.03$ the maximum possible pix_{err} by the above equation is 15 for the scaled pixel values which is an acceptable error. Note this is an approximation and only serves to provide an estimate.

Appendix B: Calculation of frequency of check for termination

Experimentally, it was found that the greatest overhead incurred by checking in every iteration occurs when only one process is executing the program: (tested on edge256x192.pgm)
This overhead was found to be 36.789 microseconds per iteration.
Execution time per iteration without check = 61.599 microseconds
Standard deviation of execution time per iteration without check =0.667 microseconds.
Thus if the total overhead incurred by performing the check is equated to the standard deviation of the total execution time a suitable iteration frequency can be obtained.

$$36.789 * \frac{ITR}{f} = 0.667 * ITR$$

$\therefore f = 55.15$ iterations

Thus an iteration period of 60 is used when executing on multiple processes as the standard deviation of the execution time per iteration is much lower than 0.667 microseconds.

Appendix C: Performance results

Performance tests

Table 1: Performance results per iteration for reconstruction of edge192x128.pgm

Nprocs	Time (ms)	Speed up	Efficiency
1	0.03123	1	1
2	0.01714	1.8216	0.91081619
4	0.01146	2.7261	0.68153404
8	0.00875	3.5699	0.4462421
12	0.01022	3.057	0.25474751
16	0.01074	2.9078	0.18173813
20	0.01143	2.733	0.13665093
24	0.01061	2.9423	0.1225957

Table 2: Performance results per iteration for reconstruction of edge256x192.pgm

Nprocs	Time (ms)	Speed up	Efficiency
1	0.0623	1	1
2	0.03308	1.8833	0.94163338
4	0.01904	3.2712	0.81780533
8	0.01434	4.3442	0.54302811
12	0.01333	4.6748	0.38956345
16	0.01255	4.9653	0.31033274
20	0.01362	4.5728	0.22864141
24	0.01435	4.3411	0.18088007

Table 3: Performance results per iteration for reconstruction of edge512x384.pgm

Nprocs	Time (ms)	Speed up	Efficiency
1	0.24179	1	1
2	0.127	1.9038	0.95189804
4	0.07165	3.3745	0.84362686
8	0.04288	5.6388	0.7048439
12	0.03085	7.8368	0.65306923
16	0.02715	8.906	0.55662811
20	0.02468	9.7978	0.48988995
24	0.02182	11.0836	0.46181778

Table 4: Performance results per iteration for reconstruction of edge768x768.pgm

Nprocs	Time (ms)	Speed up	Efficiency
1	0.77099	1	1
2	0.38608	1.997	0.99848221
4	0.20472	3.7662	0.94154068
8	0.11324	6.8082	0.85102699
12	0.08025	9.6074	0.80061485
16	0.06445	11.9622	0.74764046
20	0.05519	13.9698	0.698489
24	0.04521	17.0528	0.71053177

Repeatability test

Table 5: Speed-up obtained by running the code on 4, 12 and 24 processes 8 times

Processes	**4**	**12**	**24**
Run 1	3.7312	9.6646	16.7891
Run 2	3.7711	9.6178	16.5907
Run 3	3.769	9.5859	16.7518
Run 4	3.7539	9.5678	16.6854
Run 5	3.6939	9.6155	16.5037
Run 6	3.7096	9.578	16.6974
Run 7	3.7099	9.5688	16.9255
Run 8	3.7359	9.6225	16.6953
Standard deviation	**0.028765726**	**0.033540996**	**0.126494121**