|epcc|

THE UNIVERSITY OF EDINBURGH

MPI Casestudy: Parallel Image Processing

David Henty

1 Introduction

The aim of this exercise is to write a complete MPI parallel program that does a very basic form of image processing. We will start by writing a serial code that performs the required calculation but is also designed to make subsequent parallelisation as straightforward as possible. The image itself is a large two-dimensional grid, so the natural parallel approach is to use regular domain decomposition. As the image processing method we will use involves nearest-neighbour interactions between grid points, this will require boundary swaps between neighbouring processes. The exercise also utilises parallel scatter and gather operations. For simplicity, we will use a one-dimensional process grid (i.e. decompose the problem into slices).

The solution can easily be coded using either C or Fortran. The only subtlety is that the natural direction for the parallel decomposition, whether to slice up the 2D grid over the first or second dimension, is different for the two languages. Some sample images and IO routines in C and Fortran can be found in the file `MPP-casestudy.tar` on the MPP course web pages.

2 The Method

You will be given a file that represents the output from a very simple edge-detection algorithm applied to a greyscale image of size $M \times N$. The edge pixel values are constructed from the image using

$$edge_{i,j} = image_{i-1,j} + image_{i+1,j} + image_{i,j-1} + image_{i,j+1} - 4\, image_{i,j}$$

If an image pixel has the same value as its four surrounding neighbours (i.e. no edge) then the value of $edge_{i,j}$ will be zero. If the pixel is very different from its four neighbours (i.e. a possible edge) then $edge_{i,j}$ will be large in magnitude. We will always consider i and j to lie in the range $1, 2, \ldots M$ and $1, 2, \ldots N$ respectively. Pixels that lie outside this range (e.g. $image_{i,0}$ or $image_{M+1,j}$) are considered to be white, i.e. to have the value 255.

Many more sophisticated methods for edge detection exist, but this is a nice simple approach. See Figure 1 for an example of how this works in practice.

The exercise is actually to do the reverse operation and construct the initial image given the edges. This is a slightly artificial thing to do, and is only possible given the very simple approach used to detect the edges in the first place. However, it turns out that the reverse calculation is iterative, requiring many successive operations each very similar to the edge detection calculation itself. The fact that calculating the image from the edges requires a large amount of computation, including many boundary swaps in the parallel code, makes it a much more suitable program than edge detection itself for the purposes of timing and parallel scaling studies.

As an aside, this inverse operation is also very similar to a large number of real scientific HPC calculations that solve partial differential equations using iterative algorithms such as Jacobi or Gauss-Seidel.

Figure 1: Result of simple edge detection

3 Serial Code

You are provided with two routines *pgmread* and *pgmwrite* in C (see *pgmio.c*) or Fortran (see *pgmio.f90*). The first routine reads the input file, and the second writes out an array as a Portable Grey Map (PGM) file that can be viewed using the program *display*. There is also a helper function *pgmsize* to compute the size of an image file, which is useful if you are using dynamic array allocation. Instructions detailing how to call these routines from your own program are included as comments at the top of each file - *you should not waste time looking at the code that actually implements the routines!*.

The file formats are very simple, e.g. the edge PGM file is a text file which contains the image dimensions M and N and the $M \times N$ integer values of $edge_{i,j}$. Writing an array as a greyscale image using *pgmwrite* requires a small amount of processing as the greyscale values must be positive, whereas a general data array (such as $edge_{i,j}$) can have both positive and negative entries.

Although the input and output files both contain only integer values, note that *pgmread* reads the edge data into an array of double-precision floating-point numbers (i.e. `double` / `DOUBLE PRECISION`) and *pgmwrite* writes out from an array of floating-point numbers. This is because the reverse calculation that we are performing cannot easily be done in integer arithmetic.

3.1 Viewing the edges

Write a program that simply declares a floating-point array (i.e. `double` in C or `DOUBLE PRECISION` in Fortran) of just the right size for the smaller image *edge192x128.pgm*. Use the two supplied routines to read in the edge data file and then immediately write it out again as a PGM file. View the image using *display*. Is it obvious what the full image should look like when you can only see the edges?

3.2 Reconstructing the image

It turns out that the full image can be reconstructed from the edges by *repeated* operations of the form

$$new_{i,j} = \frac{1}{4}(old_{i-1,j} + old_{i+1,j} + old_{i,j-1} + old_{i,j+1} - edge_{i,j})$$

where old and new are the image values at the beginning and end of each iteration. We will take the initial value for the image array (the value of old at the start of the first iteration) to be pure white.

The simplest way to set pixels off the edge of the array to 255 is to declare all arrays in your program with explicit halos, i.e. `double old[M+2][N+2]` in C or `DOUBLE PRECISION old(0:M+1,0:N+1)` in Fortran (similarly for new and $edge$), and set the halo values to 255. For IO you will need one more array called buf which has *no halos*.

The entire loop then becomes:

1. read the edges data file into the buf array
2. loop over $i = 1, M$; $j = 1, N$
   - $edge_{i,j} = buf_{i-1,j-1}$ (in C)
   - $edge_{i,j} = buf_{i,j}$ (in Fortran)
3. end loop
4. set the entire old array to 255 *including* the halos
5. begin loop over iterations
   - loop over $i = 1, M$; $j = 1, N$
     - set $new_{i,j} = \frac{1}{4}(old_{i-1,j} + old_{i+1,j} + old_{i,j-1} + old_{i,j+1} - edge_{i,j})$
   - end loop
   - set the old array equal to new, making sure that you *do not* copy the halos
6. end loop over iterations
7. copy the old array back to buf excluding the halos
8. write out the final image by passing buf to *pgmwrite*

The initialisation of old to 255 achieves two things. Most importantly it sets the very outer edges of the image to 255. However, it also sets up our initial guess for the solution to be a completely white image.

3.3 Testing the serial code

Run your code for 0, 1, 10, 100 and 1000 iterations and view the reconstructed image each time. Do you see what you expect? How many iterations do you think it will take to recover an "exact" image?

4 Initial parallelisation

For the initial parallelisation we will simply use trivial parallelism, i.e. each process will work on different sections of the image but with no communication between them. Although this will not reconstruct the image exactly, since we are not performing the required halo swaps, it serves as a good intermediate step to a full parallel code. Most importantly it will have exactly the same data decomposition and parallel IO approaches as a fully working parallel code.

Before progressing any further, please ensure you have a backup copy of your working serial code.

The entire parallelisation process is made much simpler if we ensure that the slices of the image operated on by each process are contiguous pieces of the whole image (in terms of the layout in memory). This means dividing up the image between processes over the first dimension i for a C array `edge[i][j]`, and the second dimension j for a Fortran array `edge(i,j)`. Figure 2 illustrates how this would work on 4 processes where the slices are numbered according to the rank of the process that owns them.

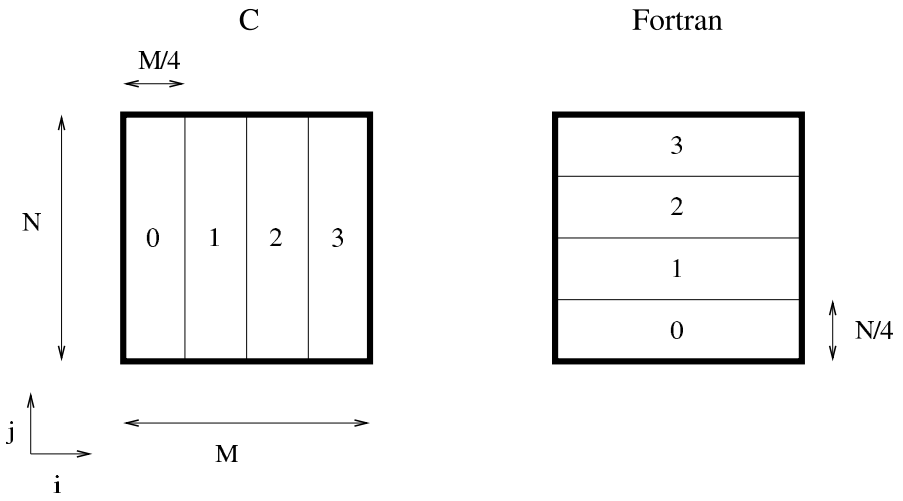

Figure 2: Decomposition strategies for 4 processes

Again, for simplicity, we will always assume that M and N are exactly divisible by the number of processes P. It is easiest to program the exercise if you make P a compile time constant (a `#define` in C or a `parameter` in Fortran).

The simplest approach to doing the IO is to read the entire file into an array on one master process (usually $rank = 0$) and then to distribute it amongst the other processes. Note that this is not particularly efficient in terms of memory usage as we need enough space to store the whole image on a single process.

I will call the dimensions of the image slices M_P and N_P where

- for C: $M_P = M/P$ and $N_P = N$
- for Fortran: $M_P = M$ and $N_P = N/P$

4.1 The parallel program

The steps to creating the first parallel program are as follows.

1. re-dimension all the arrays with M and N replaced by M_P and N_P
2. create a new array called $masterbuf$ of size $M \times N$
3. initialise MPI, compute the $size$ and $rank$, and check that $size$ = P
4. on the master process only, read the edges data file into $masterbuf$
5. split the data up amongst processes using `MPI_Scatter` with $sendbuf = masterbuf$ and $recvbuf = buf$
6. now follow steps 2 through 7 of the original serial code, exactly as before, except with M and N replaced by the local sizes M_P and N_P
7. transfer the data from all the buf arrays back to $masterbuf$ on the master using a call to `MPI_Gather`
8. on the master process only, write out the final image by passing $masterbuf$ to *pgmwrite*

4.2 Testing the parallel program

Run your code for 0, 1, 10, 100 and 1000 iterations and look at the output images. How do they compare to the (correct) serial code? Is the total execution time decreasing as you expect? In terms of Amdahl's law, what are the inherently serial and potentially parallel parts of your (incomplete) MPI program?

5 Full parallel code

The only addition now required to get a full parallel code is to add halo swaps to the old array (which is the only array for which there are non-local array references of the form $old_{i-1,j}$, $old_{i,j+1}$ etc). This should be done once every iteration, immediately after the start of the iteration loop and before any other computation has taken place.

To do this, each process must know the $rank$ of its neighbouring processes. Referring to the Fortran decomposition as shown in Figure 2, these are given by $rank - 1$ and $rank + 1$. However, since we have fixed boundary conditions on the edges, $rank$ 0 does not send any data to (or receive from) the left and $rank$ 3 need not send any data to (or receive from) the right. This is best achieved by defining a 1D Cartesian topology with non-periodic boundary conditions - you should already have code to do this from the previous "message round a ring" exercise. You also need to ensure that the processes do not deadlock by all trying to do synchronous sends at the same time. Again, you should re-use the code from the same exercise.

The communications involves sending and receiving entire rows or columns of data (depending on whether you are using C or Fortran). The process is as follows - it may be helpful to look at the appropriate decomposition in Figure 2.

For C:

- send the N array elements (`old[`M_P`][j]; j = 1,N`) to $rank + 1$
- receive N array elements from $rank - 1$ into (`old[0][j]; j = 1,N`)
- send the N array elements (`old[1][j]; j = 1,N`) to $rank - 1$
- receive N array elements from $rank + 1$ into (`old[`M_P`+1][j]; j = 1,N`)

For Fortran:

- send the M array elements (`old[i][`N_P`]; i = 1,M`) to $rank + 1$
- receive M array elements from $rank - 1$ into (`old[i][0]; i = 1,M`)
- send the M array elements (`old[i][1]; i = 1,M`) to $rank - 1$
- receive M array elements from $rank + 1$ into (`old[i][`$N_P + 1$`]; i = 1,M`)

Each of the two send-receive pairs is basically the same as a step of the ring exercise except that data is being sent in different directions (first clockwise then anti-clockwise). Remember that you can send and receive entire halos as single messages due to the way we have chosen to split the data amongst processes.

5.1 Testing the complete code

Again, run your program for 0, 1, 10, 100 and 1000 iterations and compare the output images to the serial code. Are they exactly the same? How do the execution times compare to the serial code and the previous (incomplete) parallel code? How does the time scale with P?

Plot parallel scaling curves for a range of problem sizes. You may want to insert explicit timing calls into the code so you can exclude the IO overheads.

6 Further Work

Congratulations for getting this far! Here are a number of suggestions for extensions to your current parallel code, in no particular order.

6.1 Stopping criterion

It is possible to quantify how many iterations of the main loop are required rather than simply stopping after some fixed number. The easiest thing to do is to monitor how much the image changes at each iteration; we will stop when none of the pixels change by more than a certain amount.

To do this, compute the maxium absolute change of any pixel in the image, i.e. compute Δ given by:

$$\Delta = \max_{i,j} |new_{i,j} - old_{i,j}|$$

where $|x|$ means the absolute value of x which is always positive. In a parallel code you will need to compute Δ locally on each process, then do a global reduction across processes.

Rewrite your code so that it terminates when Δ is less than some amount, say 0.1. Given the overhead of calculating Δ, particularly considering the need for a global sum, is it worth checking it every iteration? By timing your code with and without computing Δ, estimate the optimal frequency at which you should check for completion.

6.2 Overlapping communication and calculation

One of the tutorial problems concerns overlapping communication and calculation, i.e. doing calculation that does not require the communicated halo data at the same time as the halo is being sent using non-blocking routines. Calculations involving the halos are done separately after the halos have arrived.

See if you can implement this in practice in your code. Does it improve performance?

6.3 Derived Data Types for IO

The parallel IO currently proceeds in three phases, e.g. on input

1. read the data into $masterbuf$
2. scatter the data from $masterbuf$ to buf
3. copy buf into $edge$

and in the reverse order for output. By defining a derived datatype that maps onto the internal region of $edge$, excluding the halos, see if you can transfer data directly between $masterbuf$ and $edge$.

6.4 Reducing IO memory

By altering the IO routines so that they can read in the data in smaller chunks, see if you can create a working parallel code without the need for the large $masterbuf$ array (you may need to change the order in which the data is stored in the input file). How does the performance of the new parallel input and/or output routines compare to the original, especially on large numbers of processes?

6.5 Alternative decomposition

The way that the slices were mapped onto processes was chosen to simplify the parallelisation. How would the IO and halo-swap routines need to be changed if you wanted to parallelise your code by decomposing over the other dimension (over j for C and i for Fortran). You should be able to make only minor changes to your existing code if you define appropriate derived datatypes for the halo data, although the IO will be more complicated. What is the effect on performance?