Construction

Cyclic Mapping

In this project we must modify a program which uses a single thread to produce a Julia fractal into one which produces the same results but which runs over multiple processors. This will be achieved by the use of the MPI framework.

Precursor
Before any work is subdivided much of the framework needs to be set. Much of the preamble will be MPI definitions to allow the Master process to communicate with the Worker processes. The most important part of the work which is done is the definition of the malloc which holds the list of all textures in the image file. this can either be a collection of pointers of image size, each pointing to a list of image size, or one list of size "image size"[2]

Master
We designate our first core as the master processor, this would be core 0 as we are looking at a numerical list of cores, but this is not important unless we have cores of varying speeds and we do not. The master processor's first task will be to take the completed calculations from the worker processors and add them to the appropriate point image map. When the master receives all of the inputs its next task is to plot the image file.

Worker
Each Worker process will look at which chunk number it is working on, and from that information alone decide on where in the image it is. Each chunk is 4 integers in this specific program and each core knows it's working on pixel

$$4 * (i * \text{total number of worker processors} + \text{processor number} - 1)$$

It is important here to talk about this equation and why it comes about. We will start with the end as it is the easiest part to explain. If we consider an integer malloc, this is a list of numbers. When we begin this list we want to start at point 0, the first entry in the list, but if our master processor is numbered 0 our first worker is number 1, therefore the work would start at entry 1 in the list. The simple solution is to take one away, effectively shifting the entire list along by one and allowing point 0 to be expressed.

The next thing we're going to look at is the section

$$i * \text{total number of worker processors}$$

We note the number 4 because we stated earlier that 4 is the amount of integers sent by each process. Here "start chunk" is the next chunk to be calculated by core 1, because we're looking at concurrent processes we can't look at the chunk a specific processor is working on directly because theoretically the process before it may not have ended. We know the first set of chunks(until we reach core 1 again) is simply

$$4 * (\text{processor number} - 1)$$

The second step begins at the 2^{nd} task on processor 1 and so if we were to number this process it would be a process with the same number as the total number of processors. We can generalise this idea to the i^{th} term. The i^{th} chunk that processor 1 is scheduled to work on is simply denoted by

$$4 * i * \text{total number of processors}$$

We may take this solution and the solution for the first chunk being worked on for each core to find a general solution which tells us which chunk a processor is expected to work on given which process core 1 is working on. We write this as

$$\text{the current iteration} + \text{the core number - 1}$$

We have already found mathematical representations for both of these sections:

$$\text{the current iteration} = 4 * i * \text{total number of processors}$$

And so by combining these we obtain an equation which loops for which core is being used and in a for loop in i given by our starting equation

$$\text{starting position of sent list} = 4*(i * \text{total number of worker processors} + \text{processor number} - 1)$$

So in summary, each Worker processor can tell which chunk it is supposed to work on by knowing which number core it is and how many chunks it has worked on in the past. The processor then sends the data to the Master process to allow it to partition the work accordingly.

Client/Server Mapping

We will now modify the code so that the mapping is now in the client/server form. For this we need the workers to be held within a while loop. The while loop will begin with the first partition free. In this case the first partition is always the id of the core working on it. When a processor becomes free it will then request another partition, this will be done by applying an MPI_Recv after the MPI_Send command which requests 1 integer, the integer received is the id of the next chunk not being worked on.

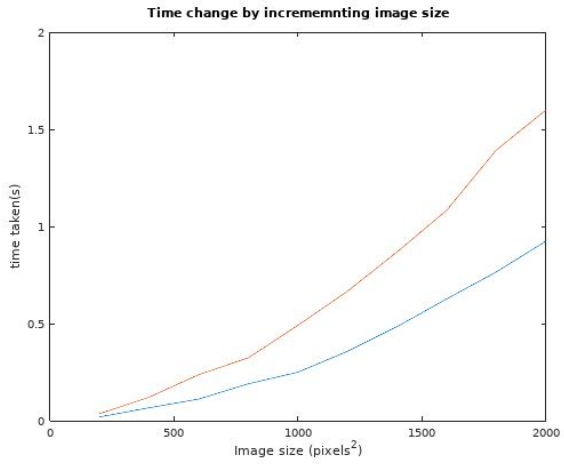
The modifications to the master process must also reflect this. The master process must loop over every chunk and apply MPI_Recv to receive the first chunk value(applied the same as in cyclic, where each core knows their first chunk), then sends the same core which sent the completed partition another core id using an MPI_Send operation equally to the opposite found inside of the worker process.

Performance

Measuring Cyclic performance

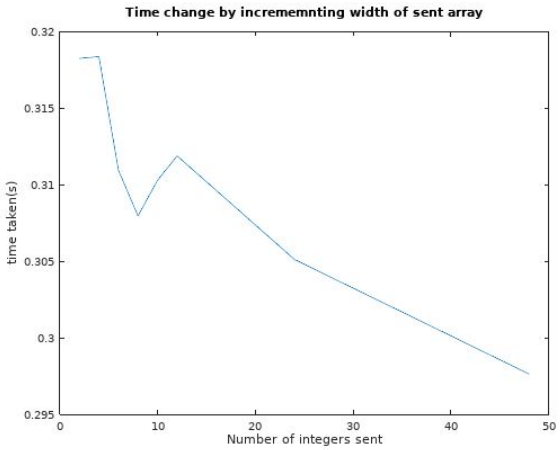
In this section we will be discussing max iterations and image size. The image size is the number of pixels in both x and y directions as the image is square. The max iterations is the number of times the fracFun function loops.

Varying image size
What is consistent
In this section we will be keeping the number of iterations and cores consistent and increment the size of the image. We set the number of cores as 16 and the number of iterations at 12000.



In this image the linear code is orange and the blue is the cyclic version. It is plain to see that the cyclic program is much faster at every instance and the code is in fact much better at higher image sizes.

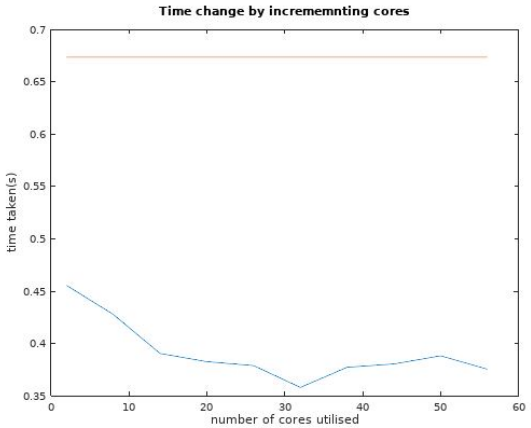
Varying partition length
In this section we will be maintaining constant image size, number of cores and max iterations. This section will be used to test if the size of the partition sent by the worker effects the time taken.



Apart from the 6 and 8 integer length partitions the values seem to be consistently decreasing. We may suppose that the two outliers may be caused by anomalies.

Varying number of cores

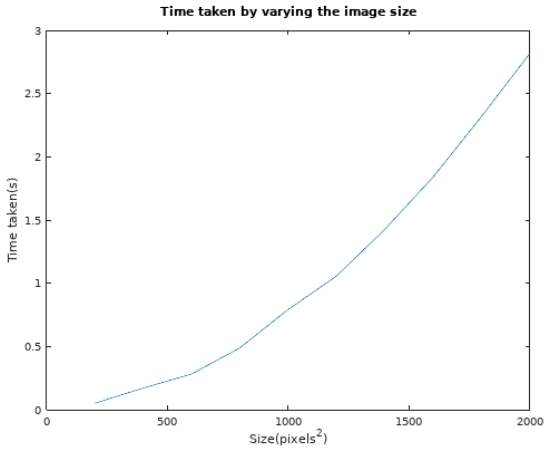
In this section we vary the number of cores, keeping the image at a consistent 1200, with a consistent max iterations of 12000.



As a frame of reference, the linear code at the same consistent points has been superimposed over the graph, it has the ornge colour. We see that even over two cores, the cyclic code still vastly outclasses the single thread version of the program, and after that the program only improves in performance.

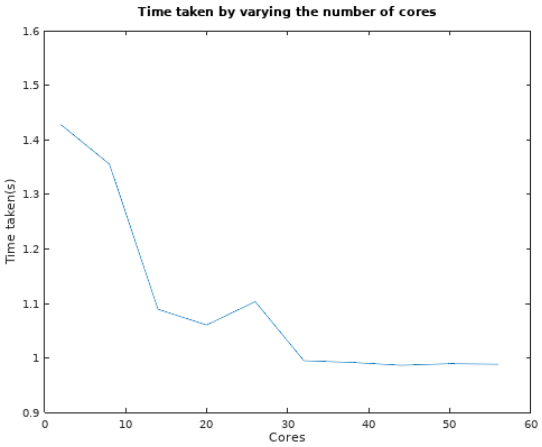
Measuring Client/Server performance

In this section we will be constructing the same **Varying image size**



The varying size graph for the Client/Server model has a much smoother curve for size growth, and if we observe the upper and lower bounds, they are lower than the Cyclic version. We may suppose that this is bacause of more efficient work allocation, which represents itself much more in larger image sizes.

Varying number of cores



We see that this graph has a neat decline, showing that the more cores used, the more efficient the work done, apart from a bump at 26. This may be because of interference, or an inefficient value of cores for distributed work. For the constant slope, we see that otherwise the Client/Server version is very fast, and improves until 32, where the efficiency doesn't seem to have any more af a noticeable effect.

Conclusion

Overall the cyclic program has vastly outperformed the linear version in every instance, but on sheer speed it is outmatched by the performance of the Client/Server structure.