Fractal Exercise

Fractals, task farm and load imbalance

Contents

1   Introduction and Aims

In this exercise you will explore what a task farm is and how it can be used to parallelise a problem. Using a basic algorithm for calculating the Mandelbrot and Julia sets you will investigate how the number of processors and the number of tasks and their size affects the performance of task farms. To do that you will need to develop a simple script to run the code across changing configurations of processors and tasks.

1.1   Mandelbrot Set

The Mandelbrot set is a famous example of a fractal in mathematics. It is a set of complex numbers c for which the sequence $[c,\ c^2+c,\ (c^2+c)^2+c,\ ((c^2+c)^2+c)^2+c,\ \dots]$ does not approach infinity. If for a given value of c the sequence does not get bigger with each iteration then it is bounded and belongs to the Mandelbrot set. For a more detailed explanation see the [Wikipedia article](#).

The pseudo-code for the basic Mandelbrot algorithm is:

```
for each x,y coordinate
    x, y = x0, y0
    for ( iterations < maxIterations )
        colour = iterations
        if (x² + y² ≥ 4)
            return colour
        else
            y = y0 + (2xy)
            x = x0 + x² - y²
```

A repeating calculation is performed for each (x,y) point in the plot area and depending on how long it takes to reach the condition ($x^2 + y^2 \geq 4$), a colour is chosen for that point (pixel). Once this condition is met, the pixel is drawn and the next point is calculated. Note that for points within the Mandelbrot set the condition will never be met, hence the need to set the upper bound for iterations we wish to examine (*maxIteration*).

The Julia set is another example of complex number sets. In many cases, the Julia set of c looks like the Mandelbrot set in sufficiently small neighbourhoods of c. For more information on the Julia set see the relevant [Wikipedia article](#).

From the parallel programming point of view the useful feature of the Mandelbrot and Julia sets is that calculation for each point is independent i.e. whether one point lies within the set or not is not affected by other points.

2 Looking at the concepts

2.1 What is a task farm?

Task farming is one of the common approaches used to parallelise applications. Its main idea is to automatically create pools of calculations (called tasks), dispatch them to the processes and the to collect the results.

The process responsible for creating this pool of jobs is known as a **source**, sometimes it is also called a *master* or *controller process*. The process collecting the results is called a **sink**. Quite often one process plays both roles – it creates, distributes tasks and collects results. It is also possible to have a team of source and sink process. A '**farm**' of one or more **workers** claims jobs from the *source*, executes them and returns results to the *sink*. The *workers* continually claim jobs (usually complete one task then ask for another) until the pool is exhausted. Figure 1 shows the basic concept of how a task is structured.

In summary, processes can assume the following roles:

- **Source** – creates and distributes tasks
- **Worker** processes – complete tasks received from the source process and then send results to the sink process
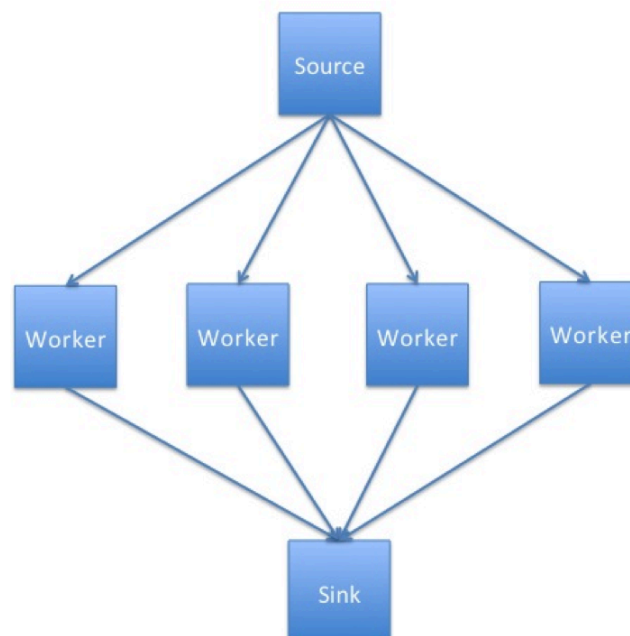- **Sink** – gathers results from worker processes

Figure 1 Schematic representation of a simple task farm.

Having learned what a task farm is, consider the following questions:

- What types of problems could be parallelised using the task farm approach? What types of problems would not benefit from it? Why?
- What kind of computer architecture could fully utilise the task farm benefits?

2.1.1 Using a task farm

As you may have guessed a task farm is commonly used in large computations composed of many independent calculations. Only when calculations are independent is it possible to assign tasks in the most effective way, and thus speed up the overall calculation with the most efficiency. After all, if the tasks are independent from each other, the processors can request them as they become available, i.e. usually after they complete their current task, without worrying about the order in which tasks are completed.

This dynamic allocation of tasks is an effective method for getting more use out of the compute resources. It is inevitable that some calculations will take longer to complete than others, so using methods such as a lock-step calculation (waiting on the whole set of processors to finish a current job) or pre-distributing all tasks at the beginning would lead to wasted compute cycles.

Of course, not all problems can be parallelised using a task farm approach.

2.1.2 Not always a task farm

While many problems can be broken down into individual parts, there are a sizeable number of problems where this approach will not work. Problems which involve lots of inter-process communication are often not suitable for task farms as they require the *master* to track which *worker* has which element, and to tell *workers* which other workers have which elements to allow them to communicate. Additionally, the *sink* progress may need to track this as well in cases of output order dependency.

It is possible to use task farms to parallelise problems that require a lot of communications, however, in such cases drawbacks and overheads impacting the performance would be incurred.

As mentioned before, to determine the points lying within the Mandelbrot set there is no need for the communications between the worker tasks, which makes it an embarrassingly parallel problem that is suitable for task-farming.

Although the calculation can employ the task farm approach, we still need to consider how to use it in the most optimal way.

Consider the following scenarios:

- How do you think the performance would be affected if you were to use more, equal and fewer tasks than workers?
- In your opinion what would be the optimal combination of the number of workers and task? What would it depend on the most? Task size? Problem size? Computer architecture?

2.2 Load Balancing

The factor deciding the effectiveness of a task farm is a task distribution. A way in which a *master* process determines how the tasks are distributed across the workers it called a **load balancing.**

A successful load balancing will avoid overloading a single worker, maximising the throughput of the system and making best use of resources available. Poor load balancing will cause some workers of the system to be idle and consequently other elements to be 'overworked', leading to increased computation time and significantly reduced performance.
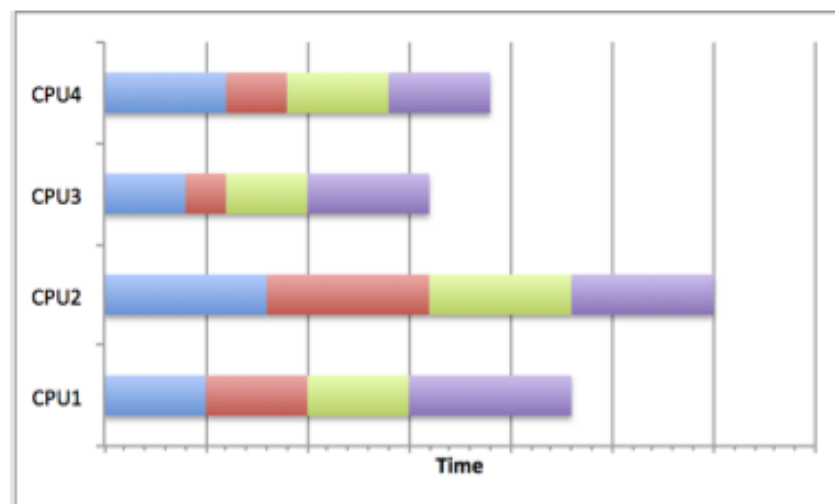
Figure 2 Example of a poor load balancing on four processes.

2.2.1 Poor load balancing

Figure 2 shows how careless task distribution can affect the completion time. Clearly, CPU2 needs more time to complete its tasks, particularly compared to CPU3. The total runtime is equivalent to the longest runtime on any of the CPUs so the calculation time will be longer than it would be if the resource were used optimally. This can occur when load balancing is not considered, random

scheduling is used (although this is not always bad) or poor decisions are made about the job sizes.

2.2.2 Good Load Balancing

Figure 3 shows how by scheduling jobs carefully, the best use of the resources can be made. By choosing a distribution strategy to optimise the use of resources, the CPUs in the diagram all complete their tasks at roughly the same time. This means that no one task has been overloaded with work and dominated the running time of the overall calculation. This can be achieved by many different means.

For example, if the task sizes and running times are known in advance, the jobs can be scheduled to allow best resource usage. The most common distribution is to distribute large jobs first and then distribute progressively smaller jobs to equal out the workload.

If the job sizes can change or the running times are unknown, then an adaptive system could be used which tries to infer future task lengths based upon observed runtimes.
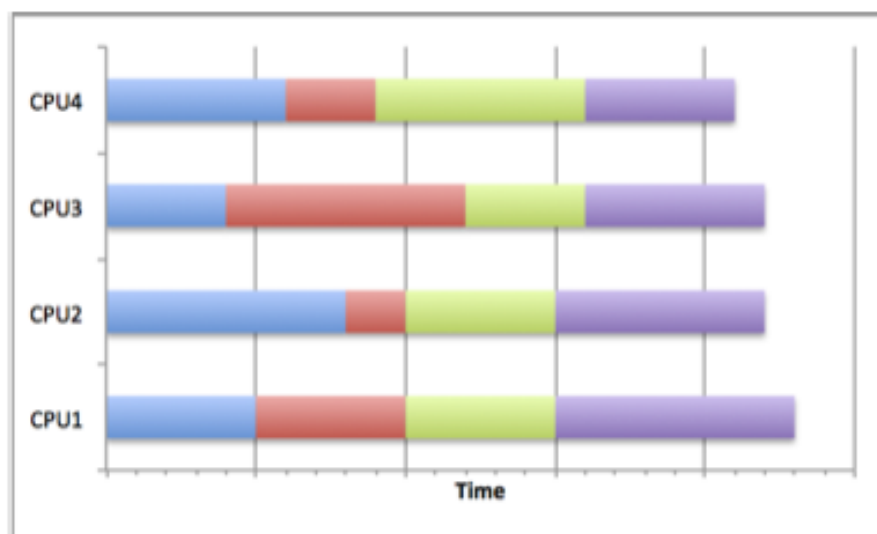

Figure 3 Example of a good task distribution on four processes.

The `fractal` program you will be using employs a queue strategy – tasks are queued waiting for workers, which completed their previous task, to claim them from the top of the queue. This ensures that workers that happen to get shorter tasks will complete more tasks, so that they finish roughly at the same time as workers with longer tasks.

2.2.3 Quantifying the load imbalance

We can try to quantify how well balanced a task farm is by computing the **load imbalance factor**, which we define as:

$$load\ imbalance\ factor = \frac{workload\ of\ most\ loaded\ worker}{average\ workload\ of\ workers}$$

For a perfect load-balanced calculation this will be equal to 1.0, which is equivalent to all workers having exactly the same amount of work. In general, it will be greater than 1.0.

It is a useful measure because it allows you to predict what the runtime would be for a perfectly balanced load on the same number of workers, assuming that no additional overheads are introduced due to load balancing. For example, if the load imbalance factor is 2.0 then this implies that, in principle, we could halve the runtime (reduce it by a factor of 2) if the load were perfectly balanced.

3   Exercise

Now that you are familiar with the Mandelbrot problem and the task farm and load balancing concepts, you are ready to experiment with different configurations of parameters to see how the performance of the code changes.

There are three main tasks to accomplish in this exercise: compilation, initial run and test runs.

3.1   Compilation

First, download the **fractal.tar.gz** file from the course webpage and copy it to ARCHER as you did in the previous exercises. Make sure you are in your */work* filesystem. Then unpack the archive using the **tar —xzvf** command, then switch to *fractal* directory and issue the **make** command.

After compilation the **fractal** executable will have been created.

3.2   Initial Run

Remember you will need to submit a job using a batch script. A working batch script is provided for you, but you will need to modify it slightly i.e. change the number of workers, the number of tasks etc. This involves changing the **aprun** command as detailed below.

The *fractal* executable will take a number of parameters and produce a fractal image in a file called *output.ppm*. By default the image will be overlaid with blocks in different shades, which correspond to the work done by different processors. This way we can see how the tasks were allocated. An example of this is presented in figure 4 – the image is divided into 256 tasks (squares) and different shade corresponds to each of 16 workers.
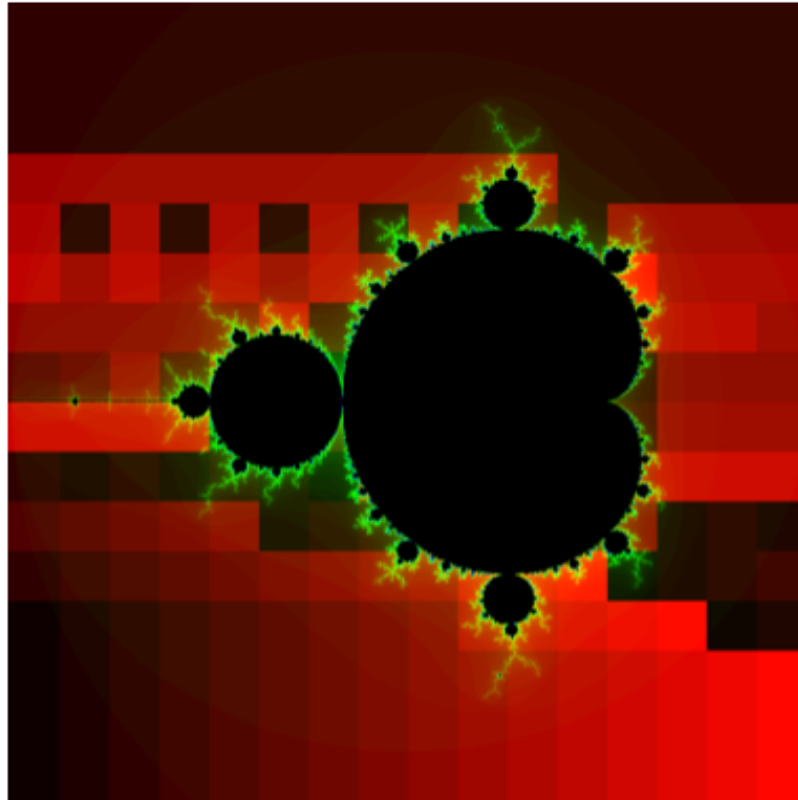
Figure 4: Example output image created using 16 workers and 256 tasks.

The default submission script does the following:

```
aprun –n 17 ./fractal –t 192
```

This creates a task farm with one *master* process and 16 *workers*. The *master* divides the image up into tasks, where each task is a square of the size of 192 by 192 pixels. The default image size is 768 x 768 pixels, which means there is exactly one task per worker, i.e. we are not yet doing anything to balance the load. *Note* that we control the number of tasks through the task size.

Submit your job to the short queue on ARCHER using the **qsub –q course1** command.

Once the program has successfully completed the directory you are working in should contain the file *fractal.o<jobID>* with the following output:

```
--------- CONFIGURATION OF THE TASKFARM RUN ---------

Number of processes:                    17
Image size:                      768 x 768
Task size:                       196 x 196 (pixels)
Number of iterations:            5000
Coordinates in X dimension:      -2.000000 to 1.000000
Coordinates in Y dimension:      -1.500000 to 1.500000

-----Workload Summary (number of iterations)---------

Total Number of Workers: 16
Total Number of Tasks:   16

Total    Worker Load: 8100198
Average Worker Load: 506262
Maximum Worker Load: 1438664
Minimum Worker Load: 52937

Time taken by 16 workers was 0.012049 (secs)
Load Imbalance Factor: 2.841738
```

The load of a worker is estimated as the total number of iterations of the Mandelbrot calculation summed over all the pixels considered by that worker. The assumption is that the time taken is proportional to this. The only time that is actually measured is the total time taken to complete the calculation.

You can view the output file using display *output.ppm*, an example of which is found in figure 4. If you want to see how the image looks without the shading use the *—n* option to the fractal program – see section 4 for more details.

Remember that the black areas inside the Mandelbrot set are 'expensive' to compute and the red areas outside the set are 'cheap' to compute.

3.3 Run tests

To explore the effect of the load balancing run the code with different number of workers and tasks and try to answer the following questions:

- From the default run with 16 workers and 16 tasks, what is your predicted best runtime based on the load imbalance factor?
- Look at the output for 16 tasks – can you understand how the load was distributed across workers by looking at the colours of the bands and the structure of the Mandelbrot set?
- Increase the number of tasks by decreasing the task size; does the runtime approach what you predicted? Look at the colour bands, how

does the task distribution change? Are the results qualitatively different if you use more workers?

- For a 16 workers, run the program with ever smaller task sizes (i.e. more tasks) and plot a graph of runtime against number of tasks. You should ensure you measure all the way up to the maximum number of tasks, i.e. a task size of a single pixel.
- Can you explain the form of the graph? Does the minimum runtime approach what you predicted from the load imbalance factor?
- You can experiment with varying the parameters, e.g. plotting the same graph with more workers or using the Julia set rather then the Mandelbrot set.

The number of workers controlled by the **–n** argument to *aprun* and is one fewer than **n** (the total number of processes) as we always require one dedicated process to be the controller.

For more details on the parameters available in the fractal program see section 4.

4   Fractal Parameters

The following options are recognised by the `fractal` program:

- **–S** - number of pixels in the x-axis of image
- **–I** - maximum number of iterations
- **–x** - the x-minimum coordinate
- **–y** - the y-minimum coordinate
- **–X** - the x-maximum coordinate
- **–Y** - the y-maximum coordinate
- **–f <fractal function>** - set to J for Julia set
- **–c** - the real part of the parameter $c+iC$ for the Julia set
- **–C** - the imaginary part of the parameter $c+iC$ for the Julia set
- **–t** - task size (pixels x pixels)
- **–n** - do not shade the output image based on task allocation to workers

The Wikipedia page on the Julia set has some suggestions for interesting parameters. For example, try:

```
aprun –n 17 ./fractal –t 96 –n –f J –c –0.8 –C 0.156 –x –2 –X 2
```

THE UNIVERSITY *of* EDINBURGH