|epcc|

Exercise: Fractals, Task Farms and Load Imbalance

May 24, 2015

1 Introduction and Aims

This exercise looks at the use of task farms and how they can be applied to parallelise a problem. We use the calculation of Mandelbrot or Julia sets to illustrate how task farms are affected by the number of processors, number of tasks and size of tasks. It requires development of a simple script to run test sets to look at the performance of the task farm across changing configurations.

The basic Mandelbrot algorithm is defined follows:

for each x,y **coordinate**
$x, y = x0, y0$
for (iterations <maxIterations)
colour = iterations
if ($x^2 + y^2 \geq 4$**)**
colour = maxIterations
else
$y = y0 + (2xy)$
$x = x0 + x^2 - y^2$

This exercise aims to introduce task farms, in particular

- Understand what is a task farm;
- How performance is affected by load balancing.

Remember Demonstrators are here to help as required. Please ask questions if anything is unclear or missing in the instructions or about the topic- this is what the demonstrators are here for.

2 Looking at the Concepts

2.1 What is a Task Farm?

Task farming is one of the common approaches to parallelisation of applications. A task farm is a technique for automatically creating and dispatching pools of calculations (tasks) in a large parallel system. Figure 1 shows the basic concept of how a task farm is structured.

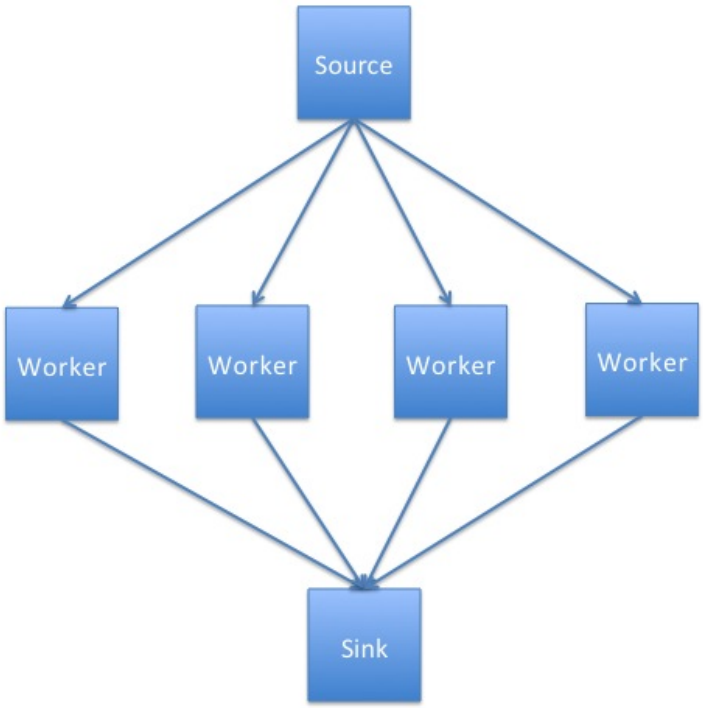

Figure 1: Basic Task Farm

A source process, also known as a master or controller process, generates a pool of jobs, while a sink process (often combined with the source process) consumes the results of the tasks. A 'farm' of one or more workers claim jobs from the source, process the task and dispatch those results to the sink.

The workers continually claim jobs until the pool is exhausted. When the number of jobs is greater than the number of workers, load balancing a computation becomes important.

The parts of a task farm:

- **Source** creates and distributes tasks.
- **Worker** processes tasks and passes on results.
- **Sink** gathers results and collates them.

2.2 Using a Task Farm

Using a task farm is most common in large computations composed of lots of independent calculations and can effectively speed up the overall calculation by using the available processors by assigning tasks to them as they request them. This would usually be when a processor finishes a task.

Allowing that some calculations will take longer than others, it is an effective method for getting more use out of the processors as opposed to using a lock-step calculation method (waiting on the whole set of processors to finish a current job) or pre-distributing all tasks at the beginning where all the long tasks could be given to a single processor.

Using a callback from the worker when it finishes helps to load balance the tasks. Without managing this, all the work could be carried out by a single processor, meaning the overall runtime will be dominated by that processor.

2.3 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 interprocess communication are ofen 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 should be noted that task farms are used where there are lots of communications; but the drawbacks and overheads have to be noted in these cases. The examples in this exercise are lots of independent tasks which can be collated by the sink process. In this exercise, there are no communications between the worker tasks.

2.4 Load Balancing

Load balancing is how a system determines how work or tasks are distributed across processing elements to ensure good performance. Successful load balancing will avoid overloading a single element, maximising the throughput of the system and make best use of resources available. Poor load balancing will lead to elements of the system being under-utilised and reducing the performance as it gets dominated by over-subscribed resources.

2.5 Poor Load Balancing

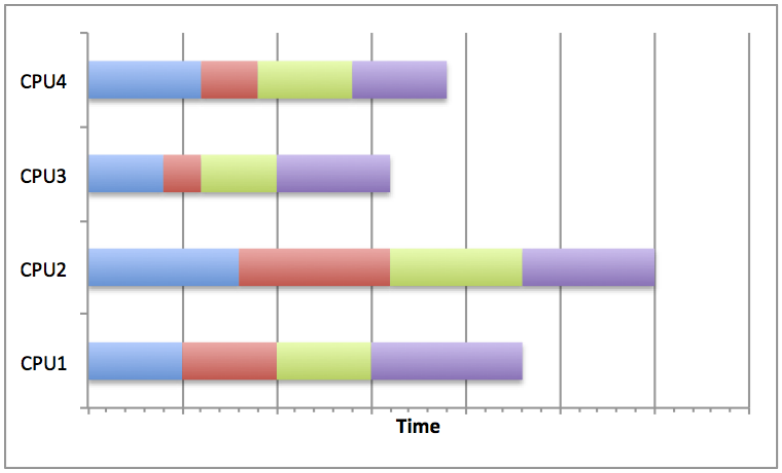

Figure 2: Four process task farm with a poor distribution of tasks

Figure 2 shows how some distributions of jobs will skew the task completion time. In the diagram it shows that CPU2 has a far longer completion time for its tasks, particularly compared to CPU3. This means that the calculation will take longer (as the total runtime is equivalent to the longest runtime on any of the CPUs) and the resources are not being 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.

2.6 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

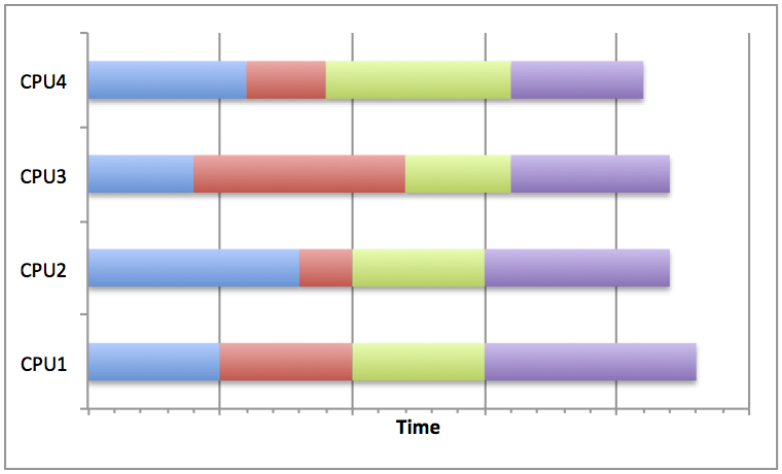

Figure 3: Four process task farm with a good distribution of tasks

roughly the same time. This means that no one task has been oversubscribed and dominated the running time of the overall calculation. This can be achieved by many different means.

Where the task sizes and running time 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 running time is unknown, then an adaptive system could be used which tries to infer future task lengths based upon observed runtimes.

The program in this exercise uses an approach where a set of tasks form a queue and as each worker finishes a task it takes the next one from the top of the queue. This load balances by ensuring that a worker with short tasks will take more to complete compared to a worker with long tasks.

2.7 Quantifying the Load Imbalance

We can try an quatify how well balanced a task farm is by computing the *Load Imbalance Factor*. We look at the loads of the different workers and compute:

$$Load\ Imbalance\ Factor = \frac{workload\ of\ most\ loaded\ worker}{average\ workload\ of\ workers}$$

For a perfectly load-balanced calculation this will be equal to 1.0; 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 blanced load on the same number of workers, assuming that balancing the load incurs no additional overeheads.

For example, if the load imbalance factor is 2.0 then this implies that the runtime can in principle be halved by better load balancing.

3 Exercise

There are three main tasks to accomplish in this exercise: Compilation, Initial Run and Run Tests.

3.1 Compilation

ARCHER First, log on to the ARCHER frontend as before. Use `wget` to obtain the file `fractal.tar.gz` from the ARCHER course material pages. Make sure that you are in the `/work/` filesystem on ARCHER.

To compile the C-MPI Fractal code, switch to the fractal/C-MPI directory and issue the make command.

```
guestXX@archer:~> make -f Makefile_archer
cc -g -c arralloc.c
cc -g -c fractal.c
cc -g -c read_options.c
cc -g -c write_ppm.c
cc -g -o fractal arralloc.o fractal.o read_options.o write_ppm.o -lm
```

Morar First, log on to the Morar frontend as before. Use `wget` to obtain the file `fractal.tar.gz` from the ARCHER course material pages.

To compile the C-MPI Fractal code, first load the MPI module: `module load mpich2-pgi`. Switch to the fractal/C-MPI directory and issue the make command.

```
bash-4.1$ make -f Makefile_morar
mpicc -g -c arralloc.c
mpicc -g -c fractal.c
mpicc -g -c read_options.c
mpicc -g -c write_ppm.c
mpicc -g -o fractal arralloc.o fractal.o read_options.o write_ppm.o
          -lm
```

After compilation a executable file will have been created called **fractal**.

3.2 Initial Run

Remember you will need to submit a job using a batch script. Batch scripts for both ARCHER and Morar are provided. For ARCHER you will need to modify the script with the relevant aprun commands as detailed below.

Remember for ARCHER when submitting the job to the batch system to replace resID with the reservation ID provided by the trainer.

3.2.1 Starting Off

The fractal executable will take a number of parameters and produce a fractal image in a file called `output.ppm`. The image will be banded in different shades which show which processor did which section. This allows how the tasks were allocated to be observed. An example of this is presented in figure 4.

The executable can take a number of parameters which will determine how the program functions, but we are mainly interested in the varying the number of tasks so we will use the default values for the other parameters (see Section 4 for a complete list of the options).

ARCHER

The default submission script does the following:

```
aprun -n 17 ./fractal -t 192
```

Submit your job to the batch system using the command:

```
guestXX@archer:~> qsub -q <resID> fractal.pbs
58306.sdb
```

Remember the number returned *****.sdb is the job ID.

Morar

To run the same defaults on Morar submit the script without modifying the contents, with the following:

```
qsub -pe mpi 17 fractal
```

To run on the front end of Morar, use:

```
mpiexec -n 17 fractal -t 192
```

This creates a task farm with one master process and 16 workers. It 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 768×768 pixels, so this means there is exactly one task per worker, i.e. we are not yet doing anything to balance the load.

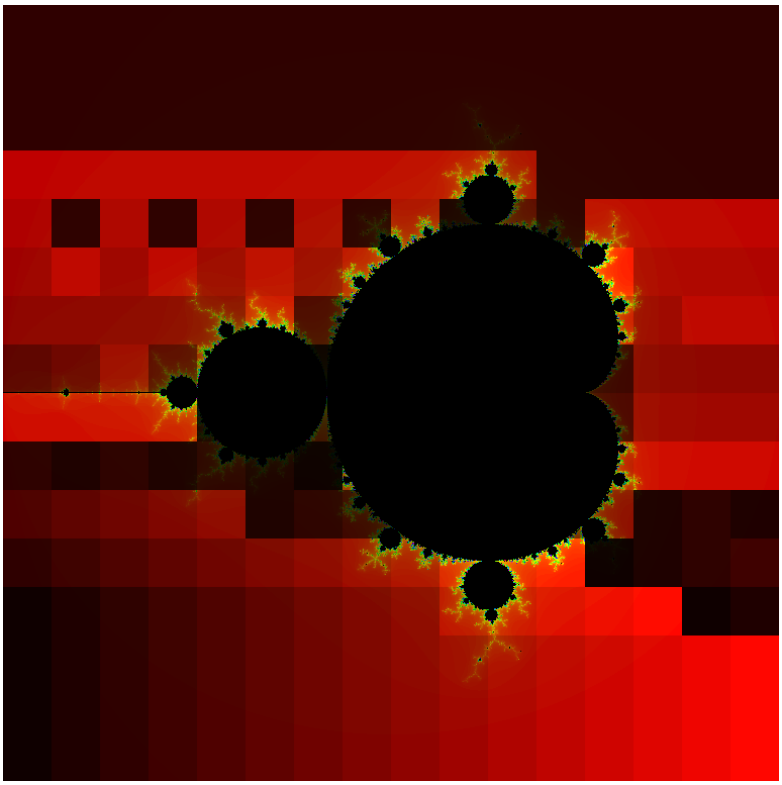

Figure 4: Example output with more tasks (256) than workers (16)

Remember you can monitor the progress of your job in the batch queue using

```
qstat -u $USER
```

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:                           192 x 192 (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: 498023053
Average Worker Load: 31126440
Maximum Worker Load: 156694685
Minimum Worker Load: 62822

Time taken by 16 workers was 1.926423 (secs)
Load Imbalance Factor: 5.034134
```

The loads of the workers are 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. Remember that you must have already loaded the `imagemagick` module.

3.3 Run Tests

In the Using HPC Resources exercise, there were a series of questions to explore about how speedup is considered, how well the code actually scales and how cheaply something could be run quickly. This involved doing multiple runs to get better average runtimes. In this case these questions are important, but workload and load balance is very important here. It is best to get maximum use out of resources and not pay for resources which are not doing work. So there are a series of questions to explore here:

1. From the default run with 16 workers and 16 tasks, what is your predicted best run time based on the Load Imbalance Factor?
2. Increase the number of tasks by decreasing the task size and note the runtimes
3. Does the runtime approach what you predicted?
4. 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?
5. Are the results qualitatively different if you use more workers?
6. Experiment with varying the parameters, e.g. using the Julia set rather than the Mandelbrot set.

If you are only interested in looking at the output picture it is best to use a single worker.

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

4 Fractal Parameters

- **aprun -n** number of processors.
- **-S** number of pixels in x-axis of image.
- **-i** maximum number of iterations.
- **-x** the x-minimum coordinate (set to -1.5 for Julia set).
- **-y** the y-minimum coordinate.
- **-X** the x-maximum coordinate (set to 1.5 for Julia set).
- **-Y** the y-maximum coordinate.
- **-f** fractal function (J for Julia set)
- **-t** task size (pixels $\times$ pixels).