Exercise: Fractals, Processors and Task Farming

April 21, 2014

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.

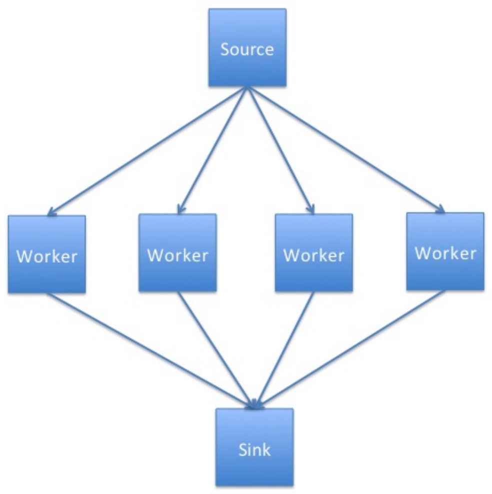

Figure 1: Basic Task Farm

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

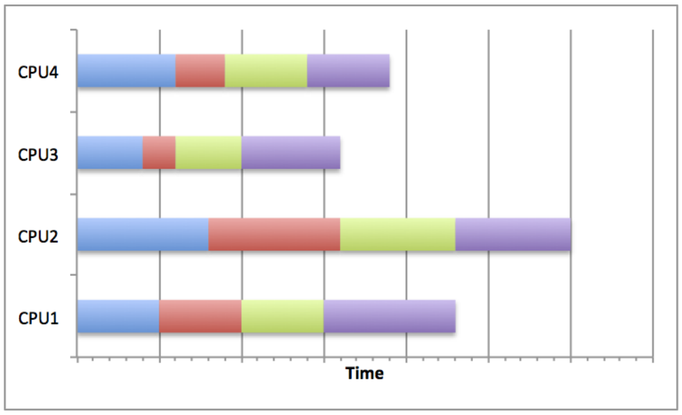

Figure 2: Four process task farm with a poor distribution of tasks

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

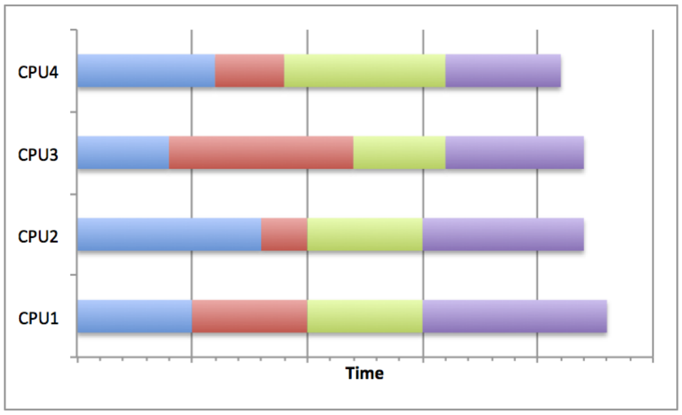

Figure 3: Four process task farm with a good distribution of tasks

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 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.

3 Tasks

There are four main tasks to accomplish in this exercise: Compilation, Single Run, Write A Script and Run Tests.

3.1 Compilation

A guest account ID and password should have been provided for ARCHER. In the event this is not the case, please contact a demonstrator.

3.1.1 Log into ARCHER fronted nodes

Open a command line Terminal and enter the following command, entering password when prompted:

```
local$ ssh guestXX@login.archer.ac.uk
Password:
```

3.1.2 Switch To Work

The exercise files are contained in an archive hosted on the EPCC web server. There is a command line utility called wget (Web GET) that allows you to download files from the web into your directories on ARCHER.

Change directory to make the /work filesystem on ARCHER. /work is a high performance parallel file system that can be accessed by both the frontend and compute nodes. All jobs on ARCHER should be run from the /work filesystem. Jobs run from /home will run much slower and cost more. Use the cd (Change Directory) command to move to user space on /work:

```
eslogin001:~> cd /work/y14/y14/guestXX
```

Use wget (on ARCHER) to get the exercise fles archive from the EPCC webserver:

```
eslogin001:> wget tinyurl.com/archer230414/fractal.tar.gz
--2014-04-21 20:26:29--  http://tinyurl.com/archer230414/fractal.tar.gz
Resolving tinyurl.com... 64.62.243.92, 64.62.243.89, 64.62.243.90
Connecting to tinyurl.com|64.62.243.92|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: http://www2.epcc.ed.ac.uk/~acollis/archer230414/fractal.tar.gz
[following]
--2014-04-21 20:26:29--
http://www2.epcc.ed.ac.uk/~acollis/archer230414/fractal.tar.gz
Resolving www2.epcc.ed.ac.uk... 129.215.62.177
Connecting to www2.epcc.ed.ac.uk|129.215.62.177|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 9015 (8.8K) [application/x-gzip]
Saving to: 'fractal.tar.gz'

100%[==============================================>] 9,015  --.-K/s
in 0s

2014-04-21 20:26:29 (632 MB/s) - 'fractal.tar.gz' saved [9015/9015]
```

Unpack the archive using the tar command:

```
eslogin001:~>  tar -xzvf fractal.tar.gz
fractal/
```

```
fractal/C/
fractal/C/utils/
fractal/C/utils/arralloc.c
fractal/C/utils/write_ppm.c
fractal/C/utils/read_options.h
fractal/C/utils/read_options.c
fractal/C/utils/write_ppm.h
fractal/C/utils/arralloc.h
fractal/C/utils/Makefile
fractal/C/exercise/
fractal/C/exercise/template/
fractal/C/exercise/template/fractal-set.c
fractal/C/exercise/template/Makefile
fractal/C/exercise/provided/
fractal/C/exercise/provided/fractal-set.c
fractal/C/exercise/provided/Makefile
fractal/config.mak
```

To compile the C-MPI Fractal code, switch to the fractal/C/exercise/provided directory and issue the make command. Note exact command output may not match exactly.

```
eslogin001:/work/y14/y14/guest01/fractal/C/exercise/provided> make
make -C ../../utils
make[1]: Entering directory '/fs4/y14/y14/guest01/fractal/C/utils'
cc    -c -o arralloc.o arralloc.c
cc    -c -o write_ppm.o write_ppm.c
cc    -c -o read_options.o read_options.c
make[1]: Leaving directory '/fs4/y14/y14/guest01/fractal/C/utils'
cc -c  -I../../utils  -o fractal-set.o fractal-set.c
cc  -I../../utils -o fractal-set fractal-set.o ../../utils/arralloc.o
../../utils/write_ppm.o ../../utils/read_options.o
```

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

3.2 Single Run

Remember you will need to submit a job using a batch scipt. A batch script is provided in but you will need to modify it with the relevant aprun commands as detailed below.

Remember 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_set 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:

- **aprun -n** number of processors.
- **-S** number of pixels in 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.
- **-t** task size (in lines).

For example to run this code with 1000 pixels in the X dimension you would use the following command in your submission script:

```
aprun -n 5 ./fractal-set -S 1000
```

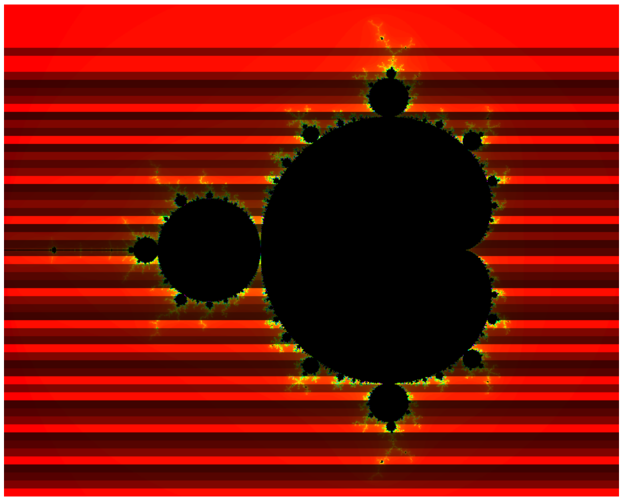

Figure 4: Example output

To run the program in default settings using four processes with one source and four workers and using the Mandelbrot function of the code use the following line in your batch submission script:

```
aprun -n 5 ./fractal-set
```

Submit your job to the batch system using the commnad:

```
guestXX@eslogin001:~> qsub -q <resID> fractal-set.pbs
58306.sdb
```

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

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-set.o<jobID> with the following output:

```
———————— CONFIGURATION OF THE TASKFARM RUN ————————

          Number of CPUs:                        5
          Image size:                            768 x 614
          Task size:                             10 (lines)
          Number of iterations:                  5000
          Coordinates in X dimension:            -2.000000 to 1.000000
          Coordinates in Y dimension:            -1.200000 to 1.200000

<cut>

———————Workload Summary (number of iterations)————————

          Total Load: 497311033
          Max Unit Load: 22112748
          Min Unit Load: 7100
          Average Load: 8021145.693548
          Workload: 2.756807
          Total Number of Tasks: 62
          Time taken by 4 workers  was 2.618707 (secs)

<cut>

CPU: 1
          CPU Total Load: 121700453
          CPU Tasks: 15
          Max CPU Load: 22112748
          CPU Average Task Load: 8113363.533333
          CPU Workload: 2.725472

CPU: 2
          CPU Total Load: 124225936
          CPU Tasks: 22
          Max CPU Load: 20864117
          CPU Average Task Load: 5646633.454545
          CPU Workload: 3.694966

CPU: 3
          CPU Total Load: 130535484
```

```
        CPU Tasks: 12
        Max CPU Load: 21182190
        CPU Average Task Load: 10877957.000000
        CPU Workload: 1.947258

CPU: 4
        CPU Total Load: 120849160
        CPU Tasks: 13
        Max CPU Load: 20355776
        CPU Average Task Load: 9296089.230769
        CPU Workload: 2.189714
```

A file called output.ppm should be produced. An example of which is found in figure 4

3.2.2 Changing the Settings

There are a number of options which can be varied in the fractal_set program.

The parts of a task farm:

- **aprun -n** number of processors.
- **-S** number of pixels in 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.
- **-t** task size (in lines).

Experimenting with this settings will allow exploration of how the different factors affect the workload and task balance in the program.

The -n flag on the aprun command is used to vary the number of processors. For example:

```
aprun -n 4 ./fractal-set
aprun -n 8 ./fractal-set
```

Note: The tasks will be done on n-1 processors with the other processor taking the controller (sink/source) role.

The -S flag will allow larger or smaller images to be processed.

```
aprun -n 4 ./fractal-set -S 500
aprun -n 4 ./fractal-set -S 5000
```

The -i flag alters the maximum number of iterations each point in the fractal to be calculated over (affecting the colour output).

```
aprun -n 4 ./fractal-set -i 100
aprun -n 4 ./fractal-set -i 100000
```

The -x, -y, -X and -Y flags alter the coordinates in the system, these can be altered to change where the output is taken from.

```
aprun -n 4 ./fractal-set -x -2.1 -y -1.3 -X 2.1 -Y 1.1
```

The -f flag chooses the function used to generate the image. M is used for Mandelbrot and J is used for Julia.

```
 aprun -n 4 ./fractal-set -f M
aprun -n 4 ./fractal-set -f J
```

The -t flag sets the number of image lines in a task. This will affect how large or small tasks are.

```
aprun -n 4 ./fractal-set -t 10
aprun -n 4 ./fractal-set -t 20
```

The next stage will require writing some scripts to evaluate how these factors affect the program performance, so take some time to try out some different combinations.

3.3 Write A Script

The next task is to modify the job submission script for the fractal-set executable with the configuration of your choice.

You can modify the aprun line to use as many of the flags as you wish and to use different numbers of processes. You can also have multiple aprun lines in the script one after the other. Note that these will execute sequentially. If the total time to run all the commands in the pbs script is greater than the walltime specified your job will be killed by the batch system before fully completing.

3.4 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. How well does the code actually scale across different numbers of cores (2-24)?
2. What is the most effective in terms of time and cost (wall time * cores) for a given image and task size?

3. How does the workload vary across cores for a given size?
4. How does workload vary across number of cores, does one core dominate?
5. How does workload vary across task size?
6. Can a reasonable configuration be recommended for this problem which will cover the majority of cases?

This is an good example to try varying parameters in a run to evaluate how these can affect workloads.