Fractals, Task Farms and Load Imbalance: The Outcomes

This is a summary of what should be learned in the Fractals exercise. At the bottom is a set of results which can help with some of the questions posed in the exercise.

1 What We Should Have Accomplished

In the Fractals exercise, the following should have been attempted:

- Accessing a remote HPC resource.
- Obtained and compiled source code.
- Assessed the performance of a task farm across configurations.

2 What We Should Take Away

After attempting the exercise, it is important to reinforce a few key points. **Note:** if anything is unclear or missing, ask the demonstrators who are there to help.

2.1 Key Points

Task Farm, also known as the master/worker pattern, allows a master process to distribute work to a set of workers processes. This is suitable for calculations which can be split up into smaller chunks with little or no inter-worker communications. The pattern can be used for other types of tasks but it complicates the situation and other patterns may be more suitable for implementing. The master process is responsible for creating, distributing and gather the individual jobs. This will often include the collation of the results into an interpretable whole.

Tasks are the units of work in a task farm. These can vary in size and do not have to be of consistent execution time. If execution times are known it can help with load balancing. For a task farm, the tasks should as far as possible be discrete units of work from the overall calculation.

Queues are what the pool of tasks a master generates can be thought of. Each worker will be given a new task when they finish their current task until the pool is empty. If the scheme is to give out tasks as and when requested, then the pool will act like a queue. If there is some filtering of tasks, like known long and short tasks, they may be separated into multiple queues. Some systems will generate the tasks on-the-fly while others will prepare the pool of tasks before starting to distribute tasks and others will take a strategy between these two.

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.

The distribution of tasks is generally done in either a pre-planned or responsive mode. In responsive mode, the master will give out tasks as requested by workers. In pre-planned mode, where task lengths are known or estimated, the master will map out a set of tasks for each worker to complete. This allows a clear picture of the overall workload to be developed: most planned systems allocate larger tasks first.

2.2 Other

Cost is increasingly becoming a big factor in how resources are requested and used. Hopefully this exercise will have highlighted that often just throwing more power and resources at something eventually has diminishing returns in performance. This means the monetary cost can be a lot higher for only minor returns over a more efficient and less expensive configuration at run time. Is it necessary to run on 4000 cores, or can the same problem be run on 2000 cores?

Generic patterns and techniques are common to software development and the task farm is a common pattern to recognise. This is commonly use in parallel and high performance computing and while the exercise focus on a fractal problem the underlying framework could be adapted to any problem with similar properties.

3 Sample Results

The cost for a scalar kAU (Allocation Unit) on HECToR was approximately £8.05. One kAU corresponded to one Teraflop of performance (a thousand Gigaflops) for an hour.

3.1 Cores, Time, Worker Workloads, Cost

Example Results for an image size (-S flag): 1000, fixed task size and iterations (-i flag): 5000 varying the number of cores with Times in seconds:

Cores	Total Time	Calc. Time	Max Workload	Min Workload	Cost
2	7.20	6.26	2.65	2.65	0.4p
4	3.175	2.19	3.06	2.36	0.2p
8	2.01	1.00	3.30	1.76	0.1p
16	1.60	0.49	4.90	1.81	0.08p
32	1.58	0.25	6.86	1.00	0.08p

Table 1: Example Run-Times and Min/Max Workloads

3.2 Time, Cores, Size

Example Results for a varying image size (-S flag) and number of cores and iterations (-i flag): 5000:

Application Time	(seconds)			
Cores / Image Size	1000	2000	4000	8000
4	3.18	10.80	41.58	164.33
8	2.00	6.29	23.04	89.90
16	1.60	4.19	14.55	56.44
32	1.58	3.47	11.02	40.72

Table 2: Example Full Applications Times for Core and Image Sizes