<u>How to Build a ComputeFarm</u>

Parallel programming facilitates faster execution of programs by breaking them into meaningful chunks and processing them in parallel across multiple execution units or processors. This programming model is employed mostly across a host of NP-complete problems where a brute-force approach is needed to solve the problem as a whole. The paper is about an open-source Java framework called "ComputeFarm" which is used to develop and run such parallel programs to achieve high efficiency. It runs on a platform called Jini that brings code mobility and fault tolerance to distributed network applications.

ComputeFarm follows the Master-Worker pattern, wherein a master process creates a collection of tasks that need to be run and the workers take tasks from the collection and run them, and then hand the computed result to the master. In order to decouple the functionality of the master and worker processes, there is another component called "Space" or "ComputeSpace" which acts as the medium of communication between master and workers.

A ComputeSpace holds the Task objects and Result objects of the type Object. Each worker waits for an available Task from the ComputeSpace; executes the Task; puts the Task result back into the ComputeSpace; and repeats the work as and when tasks are available. The granularity of the work that is done by the workers is an important factor in that it defines the amount of work (computation) that each worker does and it should be moderate enough to not allow one worker to hold up the computation, thereby ensuring that the load is equally balanced among all the workers. A client of ComputeFarm (the master process) will not usually think in terms of the workers doing their work, but in terms of the overall problem they have to solve, called a Job. So the client creates a Job and specifies how to divide it into Tasks. These tasks are passed to the ComputeSpace, which makes them available to the workers or Computes. The results computed by the Computers are then returned to the Space. The Client then retrieves these results from the Space, composing them into a solution to the original problem.

All these processes typically run concurrently. So, the client may be still dividing the Job into Tasks as the computed results of earlier Tasks may become ready to be processed. The client is completely oblivious about the workers and they simply consider the ComputeSpace as a raw computing resource where tasks are automatically executed as soon as they are dropped.

The author cites an example of computing the sum of squares of the first 'n' natural numbers. As per the ComputeFarm infrastructure, the idea is to break the problem into smaller sub-problems (squaring) using the Job's generateTasks() method, and then re-combine the results of these sub-problems which were computed by the workers to produce the final result (addition) using its collectResults() method.

When we are considering such multiple distributed processes, there is a change in kind in the types of failure that programs can exhibit. Jini provides facilities for dealing with some of the fallacies of distributed computing, problems of system evolution, resilience, security and the dynamic assembly of service components. For instance, the Space can accommodate faulty computers, i.e., if a computer that is running a task crashes and returns a RemoteException, the task is assigned to another computer. We also need to consider the ways to handle all possible types of exception that can arise when dealing with the remote systems in order to make the entire infrastructure robust.

Code mobility is a core concept of the platform and provides many benefits including non-protocol dependence. Being in a distributed environment, the components would need access to the class files that are possessed by the other components. So the Java classes are made available to the remote JVMs by providing a codebase, i.e., a URL from where the classes can be downloaded.