Summary of "How To Build a ComputeFarm"

Anders Kristiansen, Eivind Kristoffersen, Samson Svendsen and Simen Aakhus

The paper "How To Build a ComputeFarm" by Tom White gives an introductory view on how to make use of *ComputeFarm,* to develop and run programs in parallel. In general running a program in parallel includes sharing the work among multiple processors, to solve a large scale problem. For most applications this is only effective when the problem is very large, because overhead will always be included when sharing the work. Applications where parallel programs may be found useful includes solving large linear systems for a FE-models, SETI@home, rendering of computer graphics, and brute-force searches in cryptography (all of which can be considered embarrassingly parallel). Throughout the paper the use of *ComputeFarm* with the use of code examples to provide the reader with further insight.

ComputeFarm is an open source Java framework which is based on the Replicated-Worker pattern (also referred to as Master-Worker pattern). For this pattern a master process (client) creates a collection of tasks, i.e. dividing the original task into several smaller tasks. The workers (computers) take tasks in a load balancing way, and return the computed result. It is important to emphasize that the communication between the client and the set of computers is done through an intermediary space, enhancing decoupling.

The process of which a given problem is solved is as follows. The client creates a *Job* which specifies several smaller tasks. They are put into the intermediary space, referred to as *ComputeSpace*. At this point, every worker are either computing results or waiting for new tasks. When a task is solved, the result is returned to the *ComputeSpace,* and the process is concluded when the *Job* collects all the returned results and return them to the client. A detailed code example is then provided, solving the problem of adding the squares of all number from 1 to an integer n. This is of course an embarrassingly parallel problem, which for large numbers will allow for good speed up and parallel efficiency.

The paper then covers the key differences between deploying the application on a single JVM and multiple JVMs. When using several computers each JVM must download the task implementation. This is a key to *ComputeFarm*, because it allows the workers to be generic. The paper covers two alternatives for allowing remote JVMs to download the code dynamically. The first option is use a codebase, which most commonly is a .jar file containing class files needed for the remote. The codebase specifies the URL for which the class files can be downloaded from, and is most often provided through the command line. This alternative often comes with configuration errors, and the paper therefore presents an alternative solution using a lightweight server called *ClassServer*. To conclude the example the paper presents a procedure for setting up a network of workers and running the JavaSpace together with the client.

One of the main advantages of *ComputeFarm* is that it provides the programmer with mechanisms for dealing with common failure problems in distributing computing, presented in the paper "The Eight Fallacies of Distributed Computing" by Peter Deutsch. C*omputeFarm* comes with flexibility meaning no specific order is needed when starting the client, workers and *JavaSpace*. It also offers robustness, meaning that the computation continues, even though one worker fails (as longs as there are multiple workers). In the end, the paper presents three exceptions (that may occur when using a remote *ComputeSpace*) that should be considered and taken into account when designing an application with a focus on robustness.