**What is Parallel Computing?**

SLIDE 1

Some programs can be made to run faster by dividing them up into smaller pieces and running these pieces on multiple processors. This is known as *parallel computing*, and a large number of hardware and software systems exist to facilitate it

* Traditionally, software has been written for ***serial* computation**:
  + To be run on a single computer having a single Central Processing Unit (CPU);
  + A problem is broken into a discrete series of instructions.
  + Instructions are executed one after another.
  + Only one instruction may execute at any moment in time.

SLIDE 2

* In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem:
  + To be run using multiple CPUs
  + A problem is broken into discrete parts that can be solved concurrently
  + Each part is further broken down to a series of instructions
  + Instructions from each part execute simultaneously on different CPUs

SLIDE 3

The most famous example of a (distributed) parallel program is SETI@home, but there are many other applications, including ray tracing, database searching, code breaking, neural network training, genetic algorithms, and a whole host of NP-complete problems where a brute force approach is needed.

**SETI@home** ("SETI at home") is an Internet-based public [volunteer computing](http://en.wikipedia.org/wiki/Volunteer_computing) project employing the [BOINC](http://en.wikipedia.org/wiki/Berkeley_Open_Infrastructure_for_Network_Computing) software platform, hosted by the [Space Sciences Laboratory](http://en.wikipedia.org/wiki/Space_Sciences_Laboratory), at the [University of California, Berkeley](http://en.wikipedia.org/wiki/University_of_California,_Berkeley), in the [United States](http://en.wikipedia.org/wiki/United_States). *SETI* is an acronym for the [Search for Extra-Terrestrial Intelligence](http://en.wikipedia.org/wiki/SETI). Its purpose is to analyze radio signals, searching for signs of extraterrestrial intelligence, and is one of many activities undertaken as part of [SETI](http://en.wikipedia.org/wiki/SETI).

**Ray tracing** (graphics) is a technique for generating an [image](http://en.wikipedia.org/wiki/Digital_image) by tracing the path of [light](http://en.wikipedia.org/wiki/Light) through [pixels](http://en.wikipedia.org/wiki/Pixel) in an [image plane](http://en.wikipedia.org/wiki/Image_plane)and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical [scanline rendering](http://en.wikipedia.org/wiki/Scanline_rendering" \o "Scanline rendering) methods, but at a greater [computational cost](http://en.wikipedia.org/wiki/Computation_time).

**Database searching:** The largest web search engines such as [Google](http://en.wikipedia.org/wiki/Google) and [Yahoo!](http://en.wikipedia.org/wiki/Yahoo!) utilize tens or hundreds of thousands of computers to process billions of web pages and return results for thousands of searches per second. High volume of queries and text processing requires the software to run in highly distributed environment with high degree of redundancy

**Code breaking: Cryptanalysis** is the art and science of analyzing [information systems](http://en.wikipedia.org/wiki/Information_system) in order to study the hidden aspects of the systems.[[1]](http://en.wikipedia.org/wiki/Cryptanalysis#cite_note-1) Cryptanalysis is used to defeat [cryptographic](http://en.wikipedia.org/wiki/Cryptographic) security systems and gain access to the contents of encrypted messages, even if the [cryptographic key](http://en.wikipedia.org/wiki/Key_(cryptography)) is unknown.

**Neural network training:** An **artificial neural network**, often just called a **neural network**, is a [mathematical model](http://en.wikipedia.org/wiki/Mathematical_model) inspired by [biological neural networks](http://en.wikipedia.org/wiki/Biological_neural_network). A neural network consists of an interconnected group of [artificial neurons](http://en.wikipedia.org/wiki/Artificial_neuron), and it processes information using a [connectionist](http://en.wikipedia.org/wiki/Connectionism) approach to [computation](http://en.wikipedia.org/wiki/Computation). In most cases a neural network is an [adaptive system](http://en.wikipedia.org/wiki/Adaptive_system) that changes its structure during a learning phase. Neural networks are used to model complex relationships between inputs and outputs or to [find patterns](http://en.wikipedia.org/wiki/Pattern_recognition) in data.

**Genetic algorithms:**  A **genetic algorithm (GA)** is a [search](http://en.wikipedia.org/wiki/Search_algorithm) [heuristic](http://en.wikipedia.org/wiki/Heuristic) that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to [optimization](http://en.wikipedia.org/wiki/Optimization_(mathematics)) and [search](http://en.wikipedia.org/wiki/Search_algorithm) [problems](http://en.wikipedia.org/wiki/Problem). Genetic algorithms belong to the larger class of [evolutionary algorithms](http://en.wikipedia.org/wiki/Evolutionary_algorithm) (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as [inheritance](http://en.wikipedia.org/wiki/Heredity), [mutation](http://en.wikipedia.org/wiki/Mutation_(genetic_algorithm)), [selection](http://en.wikipedia.org/wiki/Selection_(genetic_algorithm)), and [crossover](http://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)).

SLIDE 4

ComputeFarm is an open source Java framework for developing and running parallel programs. Under the covers, ComputeFarm runs on Jini, which brings code mobility and fault tolerance to network applications.

Jini also called **Apache River,** is a network architecture for the construction of *distributed systems* in the form of modular co-operating services.

Originally developed by Sun, Jini was released under an open source license (Apache license).

SLIDE 5

**The Replicated-Worker Pattern**

ComputeFarm grew out of an implementation in JavaSpaces (itself a part of Jini) of the *Replicated-Worker pattern.*

In this pattern, also know as the *Master-Worker pattern*, a master process creates a collection of tasks that need to be run. Workers take tasks from the collection and run them, then hand the computed result to the master. A space is a natural conduit for passing messages between master and workers, due to the decoupled programming style it encourages.

In ComputeFarm, a ComputeSpace holds Task objects and result objects of the type Object. Each worker's lifecycle is as follows:

1. Wait for an available Task from the ComputeSpace.
2. Execute the Task.
3. Put the Task result back into the ComputeSpace.
4. Go to step 1.

There are typically many workers, and they are identical; hence the term *replicated*. This pattern neatly provides load balancing, whereby each worker contributes whatever resources it can afford. The worker on a faster machine will execute more Tasks than the worker on a slower or otherwise heavily loaded machine; and as long as the granularity of the Tasks is sufficiently fine, no one worker will hold up the computation.

SLIDE 6

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. From the client's point of view, this is what happens:

* The client creates a Job and specifies how to divide it into Tasks.
* Each Task is written into a ComputeSpace by the Job.
* Each Task in the ComputeSpace is turned into a result by one of the replicated workers.
* Results of executed Tasks are read from the ComputeSpace by the Job and combined into an overall result for the client.

Notice that these processes typically run concurrently. So, for example, the client may be still dividing the Job into Tasks as the computed results of earlier Tasks come back to be processed. For the client, the ComputeSpace is simply a raw computing resource where tasks are automatically executed as soon as they are dropped into the space. In fact, the client doesn't know about the workers, and they do not appear in the core API.

SLIDE 7

That's enough theory. Let's write a program to run on ComputeFarm.

To see how ComputeFarm can be useful, consider the following very simple example. If we wanted to calculate the sum of the first *n* squares, that is,

12 + 22 + 32 + ... + *n*2

then we might write the following piece of code

(ignoring the fact that there is a simple formula for this sum: *n*(*n* + 1)(2*n* + 1)/6).

Now imagine for the sake of example that multiplication is a significantly more expensive operation than addition. If we had multiple processors to run the program on, it would be worthwhile to arrange for the squaring operations to be shared among the processors to reduce the time to calculate the sum. In other words, we break the problem into smaller sub-problems (squaring), and then re-combine the sub-results to produce the final result (addition).

To run a Job, a ComputeFarm client gets a JobRunner from a JobRunnerFactory. There are different implementations of JobRunner that have different strategies for parallelizingTask execution. For unit testing, the JobRunner created by a SimpleJobRunnerFactory is handy, as it executes all tasks in the same JVM. We shall see later how to run a computation on multiple machines using the JavaSpaces implementation of JobRunner.

SLIDE 8

1. First of all, let's see how we can implement a program to calculate the sum of squares, starting with the unit test.

We create a new SquaresJob to calculate the sum for *n* = 10, and then obtain a JobRunner for it from the SimpleJobRunnerFactory. Calling the run() method on the runner blocks until the job completes, at which point we can retrieve the overall result from the runner and check that it has the correct value.

1. The Job interface specifies two abstract methods:

* void generateTasks(ComputeSpace space), where the implementor specifies how the problem is broken up into Tasks.
* void collectResults(ComputeSpace space), where the implementor recombines the results from each Task into the final result.

In the generateTasks() method, we create a SquaresTask for each term in the sum and then write them into the supplied ComputeSpace. Conversely, in thecollectResults() method, we take task results as they appear from the space and sum them. Both method implementations deal with various exceptions that may arise when interacting with the  
ComputeSpace; we shall look at them in more detail later, but for the moment just note that we exit if anything "bad" happens.

1. Finally, the listing for SquaresTask is very straightforward. It implements the single execute() method of the Task interface, which is an example of the [Command pattern](http://c2.com/cgi/wiki?CommandPattern). For this example, the implementation squares an integer. SquaresTask has also been marked Serializable to allow it to be marshalled using RMI, as task instances will be when we use the JavaSpaces runner to share tasks between machines. Similarly, the return type of the execute() method must be Serializable. In this example java.lang.Integer, the return type, satisfies this requirement.