# Concurrency with Mandelbrot

Alex Miller
http://tech.puredanger.com

St. Louis Java User's Group
Oct 11th, 2007

## The Mandelbrot Set

The Mandelbrot Set is defined by a recursive function on imaginary numbers:
Every black point in the middle of the familiar form

$$f_c(z) = z^2 + c.$$

represents a complex number (c) with x axis denoting the real part and y axis denoting the imaginary part. The members of the Mandelbrot Set are those complex numbers that do NOT diverge to infinity. The colors outside the set represent how quickly those points diverge to infinity.

## Threading

### Single threaded

The most basic example uses no threads and just walks through each point in the picture and calculates the proper value.

### Multi Threaded

But every point in the picture can be calculated independently and (potentially) in parallel. Java has built-in support to create `Threads` and start() them.

## Protecting Critical Sections

### Java Memory Model

The JMM specifies the minimum guarantees made by the JVM about how writes are seen by other threads. It is based on an ordering called *happens-before*. Certain actions are defined to require that one action **must** *happen-before* another. The JVM cannot reorder actions governed by a *happens-before* relationship. All other actions can be reordered.

For example, within a thread every action *happens-before* every action later in the thread, so single-threaded code happens in program order. A write to a `volatile` *happens-before* any subsequent read of the `volatile`. A monitor unlock *happens-before* any subsequent lock of the monitor. Etc.

### Data Race

A data race occurs when there is >1 reader and >0 writer but the reads/writes are not ordered by the *happens-before* relationship. Data race == bad!

### Volatile

As stated above, marking a field as `volatile` indicates that any read after a write is guaranteed to see the written value. This is NOT guaranteed for non-`volatile` fields and is why double-checked locking without volatile is broken!!!!

### Atomic Classes

Java 5 provides a set of `AtomicInteger`, etc classes that encapsulate common primitive and object reference types and provide similar semantics to volatile as well as some compound actions with the same guarantees.

### Synchronized

Writes that occur in a synchronized block are guaranteed to be seen by threads accessing those variables after the block is exited.

### Locks

Java 5 also added support for Lock classes, which act similar to synchronized blocks. In general, synchronized and wait/notify can be replaced with use of `ReentrantLock` (much like synchronized) and `ReentrantReadWriteLock`.

## Thread Coordination

### Thread.join()

A classic idiom for syncing up a bunch of threads is to loop on a set of `Threads` and `join()` with each of them.

### CountDownLatch

A `CountDownLatch` is a good way to have multiple threads wait for an event to occur. The latch is initialized with a count and counts down to 0 at which point waiting threads are released.

### CyclicBarrier

A `CyclicBarrier` waits for a specified number of threads to arrive at the barrier and then releases them all. `CyclicBarriers` can be reused after a release.

## Queues

### BlockingQueue

A `BlockingQueue` supports the producer-consumer paradigm and is perfect for sending work between thread pools. `BlockingQueue` supports the ability to block when either inserting (if full) or removing (if empty) items from the queue.

### LinkedBlockingQueue

This implementation supports an unbounded queue so producers should never block when adding elements although consumers may block retrieving items.

## Executors

### ExecutorService

Java 5 added support for the notion of an `ExecutorService`, which provides a generic interface for submitting work to a pool of workers.

### Callable and Future

`ExecutorService` can work with `Runnables` but can also support a new type called `Callable` which can throw exceptions and is parameterized by the type of the object returned from invoking it.

`Future` represents the future result of a Callable, parameterized by the type of the result that will exist. Users can wait or poll for the result to arrive.

### CompletionService

Java 5 also introduced something called a `CompletionService`, which combines an `ExecutorService` with a result queue (a common occurrence).

## Fork-Join

Java 7 will add a new fork-join framework for recursively breaking down work into component tasks and executing them efficiently over a thread pool. The API is simple but the implementation is efficient and uses work-stealing to keep threads busy.

## Distributed Systems

### Message-based

One approach to creating distributed systems is to pass messages between nodes using JMS, MDB, EAI, web services, etc.

### Stateless / Shared Database

Another approach is to make all nodes stateless and instead share the state in a single database (or filesystem) that is highly available and scalable.

### Durable Shared Memory (Terracotta)

Terracotta works on a concept of durable shared virtual Java heap which transparently extends the JMM over multiple nodes.

### Grid

There are numerous solutions that allow you to define a unit of work and pass it on to a grid server, where the work is farmed out to the grid nodes as appropriate.