Parallel Programming in Java 7

Wolf Schlegel and Fabian Koehler, Thoughtworks – BCS SPA Conference, 3 July 2012

Some Theory

Why parallel programming?

Grand challenge problems such as DNA structure analysis, weather forecasting are amenable to splitting up into parallel strands.

Other areas include parallelising downloads of large files, searches.

Java 7 has features to allow exploitation of multicore processor architectures.

Moore's Law has held since the early 1970s – but it has hit the buffers a few years ago, the only way to get increases in processor power now are to turn to parallel processing.

Partitioning

Data partitioning is applied to data (aka domain decomposition).

Functional decomposition executes independent functions in parallel – much less common than data partitioning.

Divide-and-Conquer

Divide a problem into similar sub-problems – sub-problems that have the same form as the original. The problem can thus be repeated recursively. For example, summing a large number of integers; sorting a very long list, searching for text strings, finding shortest path through a directed graph…

Deciding which way is faster

Splitting and merging introduces overheads, which could outweigh the savings in execution time. Certain problems have inherent bottlenecks that prevent parallelisation.

Amdahl's Law: S(n,f) = single-processor execution time / multi-processor execution time where n = number of processors and f = fraction of the problem that is inherently serial. With f=20%, you can't get any benefit from more than 5 processors.

How to benefit from multiple CPU cores in Java

Implicitly – magic under the hood of the JVM

Explicitly – using Java 7's "fork/join" framework (focus of this session)

New Classes on the Block

- Thread subclass: ForkJoinWorkerThread

- Callable<V>
- Executor
- ExecutorService (implements Executor)
- ForkJoinPool (implements ExecutorService)
- Future<V>
- ForkJoinTask<V> (implements Future<V>)
- ForkJoinTask subclasses: RecursiveAction, RecursiveTask<V>

Tasks are very lightweight, so low-cost to create.

Fork/Join Interaction

A RecursiveAction can fork to create two new ones. This carries on until a new RecursiveAction decides not to fork any more, but instead to process its data.

The action that forked waits for the forked actions to join. Ultimately a result is returned.

RecursiveActions are "thrown into" a ForkJoinPool with a certain number of worker threads. The threads will pick up RecursiveActions and put them on their work queue. Different worker threads can "steal" units of work from one another's input queues in order to keep the workload balanced across threads.

When instantiating a ForkJoinPool, you could condition it to use just one processor core in order to ensure that all related RecursiveActions execute close to one another.

It is not advisable to use your own threading primitives within RecursiveActions!

How to Implement a ForkJoinTask

In general:

If (my portion of the work is small enough)

    Do the work

Else

    Split my work into 2 (or in certain applications more) pieces and fork them

    Merge/join the results

Testing Fork/Join

Assert that:

- The complete task works as expected
- The core computation works as expected
- Splitting the task does not corrupt the input data
- Merging results behaves as expected

Example: calculating a checksum

Split array of numbers; fork new objects initialised with each sublist; wait for each to finish by using join. The fork() and join() methods are defined on RecursiveTask<V>.

Set up Environment

Install JVM (JDK7)

https://github.com/fkoehler/TW-Fork-Join-Workshop

Follow instructions in the README file

Project files are provided for IDEA and Eclipse.

Exercise 1: Bucket Sort

Implement both parallel and sequential versions of bucket sort. Compare results.

You don't have to use recursive bucket sort for each bucket – decide what is best.