Java 1.7 and Fork/Join

A new thread pool is provided in Java 1.7; the ForkJoinPool.

The basic idea here is to create a fixed-size thread pool but to allow a task to temporarily yield the thread it is using if it knows that it will need to wait for the results of the tasks that it is forking off.

A task extends RecursiveAction, has a method named compute() that “does the work” and then in there uses something such as invokeAll(Task1, Task2) to simultaneously put those new tasks onto the front of its own double-ended queue (each thread has its own queue) and effectively yields its own thread back to the pool by then “stealing” work from the front of its own queue.

If a thread’s queue is empty, it will “steal” work from the back of another thread’s queue.
**Size of the task being stolen**

If you are using a divide and conquer algorithm, then the problem size will get smaller as things progress. By stealing from the back of other queues (they are double-ended queues, which are often called deques) when your thread is free, you increase the chances of grabbing a large task (and as we’ve seen, big tasks can be good).

**MergeSort**

*MergeSort* can clearly benefit from this “work stealing” idea of fork/join.

The heart of *MergeSort* in the `compute()` method would become something like:

```java
    MergeSortTask left = new MergeSortTask(lo, mid);
    MergeSortTask right = new MergeSortTask(mid+1, hi);
    invokeAll(left, right);
    merge(lo, mid, hi);
```
**Maze Solving**

The `invokeAll` method can be sent a `Collection` of objects. Consider our project where we had several choices that we could follow from a location.

We could create a class that extends `RecursiveAction` whose constructor takes a move and whose `compute()` method would see if there was a forced path to the solution and if not, search all the paths starting from the current point.

```java
listOfMoves = new ArrayList<Seeker>();

for (Direction next : nextChoice.choices) {
    listOfMoves.add(new Seeker(new Move(nextChoice.at, next, localMove)));
}

invokeAll(listOfMoves);
```

**New pool, old problems**

Many of the issues we discussed previously still apply;
- Pick a good number of threads for the pool to have based on factors such as the number of processors and the ratio of CPU to IO work in a task.
- Create tasks that are large enough that the amount of time they do work is more than the amount of time that it takes to set up and invoke the tasks.
- Might want to empirically determine threshold for where to switch from concurrent to sequential.
- As with others, problems that don’t involve a lot of CPU time will not benefit as much (comparing `ints` is fast but if you are comparing other types fork/join max-finding might be good).
More Fork/Join Syntax

Creating a new thread pool using this type or concurrency model is done using the `ForkJoinPool` class. There are two possible constructors. The default constructor creates a pool containing the same number of threads as there are processors available on that machine. There is also a constructor that allows you to pass in the number of threads to use.

Your worker class needs to extend either 
`RecursiveAction` and override `void compute()`
or extend
`RecursiveTask<T>` and override `<T> compute()`

Ways to send tasks off to threads...

You can start the first worker of a divide and conquer algorithm running by calling `.invoke` via the `ForkJoinPool` object you created and passing in an instance of your worker class. If the worker class extends `RecursiveTask<T>` then it will return `<T>`.

Within the worker class, you can use `invokeAll()` to add several new task instances to the current thread’s deque.

You can also (as the name suggests) explicitly `fork()` and `join()` using the pool. The `fork()` will add a new task to the current thread’s deque. The `join()` will wait until that task completes.

One difference between this call to `join()` and the one we’ve seen previous is that for this one you don’t need to catch an interrupted exception.