Recap

There are a few “quick and easy” places where you might be able to take advantage of the naturally-available parallelism in an algorithm.

- Recursion: Whenever you have a divide & conquer algorithm that is naturally presented via independent recursive calls, you have a target for concurrency.

- Loops: If each iteration of a loop is independent of the others, you have a target for concurrency.
Recursion: Too much of a good thing?

If you have set a limited number of threads based on the number of processors and dependencies between the levels of recursion, you risk deadlock.

We discussed an approach to MergeSort where a task was set to create and submit two new MergeSort tasks to the pool, wait for them to complete, and then Merge their results. However, we saw that with a fixed size thread pool, the later tasks did not have threads in which to run and the earlier tasks didn’t give up their threads until the later tasks would complete.

We could write our code so that once we are running out of threads, we stop putting recursive calls into new tasks and rather run them “normally” in the thread that has them. This would avoid the previous problem but could lead to too little of a good thing…

Recursion: Too little of a good thing?

We might write our code so that once we are running out of threads, we stop putting recursive calls into new tasks and rather run a sequential version of the algorithms in the current existing thread.

However, as those tasks do complete, we might no longer have a way to take advantage of the newly-available threads (and processors). We discussed QuickSort as an example.

To avoid this issue, we might want to approach this along the lines of the CallerRunsPolicy handler – if a task is submitted and there isn’t an available thread, the thread that submitted the tasks is told to run that task. However, in order for this to work, tasks in threads would need to complete. How would our MergeSort example fare? Not well due to the merge() that’s waiting.
Fork/Join – Java 7 – Do not use in class projects!
There is a model of concurrency using thread pools that is newly supported in Java 1.7 – fork/join parallelism, which can address these issues by having a task yield its thread when it’s waiting for its recursive tasks to return so that it can use those results. We will discuss this model and Java’s support of it a little later…

Recursive BST Test
Consider being given a tree and wanting to test whether it is a binary search tree...

```java
boolean bstCheck(TreeNode p) {
    if ((p.left == null) && (p.right == null))
        return true;
    if (p.left == null) && (p.right.value > p.value))
        return bstCheck(p.right);
    if ((p.right == null) && (p.left.value < p.value))
        return bstCheck(p.left);
    return (p.right.value > p.value) &&
        (p.left.value < p.value) &&
        bstCheck(p.left) &&
        bstCheck(p.right);
}
```
Recursion: some recursion is your friend...

We don’t have to run into the same issue here as we did with MergeSort since any task that goes into a thread could be set up to eventually end, freeing up its thread for others to use. This is because the recursive call(s) are the last things the method has to do (tail recursion).

However, it needs to return those values, so the “obvious” approach won’t work - we have to think about how to implement this – if we (for example) submit two new tasks to a thread pool in the bstCheck example and then join on them, we’re back at the same problem as with MergeSort.

Recursion: using the current thread...

In the single-subtree cases, why spawn a new task? Have it use the current thread.

In the two-subtree case, should we have one use the current thread and the other spawn a new task? What if there is no free thread?

Could we check to see if something is waiting and pull it out of the queue and run it locally?
Quicksort: tail recursion concurrency idea

```
int pivotPos=partition(first, last);

QuickSorter qsInner1 =
    new QuickSorter(arrayToSort, first, pivotPos-1, qsTP);
qsTP.submit(qsInner1);

QuickSorter qsInner2 =
    new QuickSorter(arrayToSort, pivotPos+1, last, qsTP);
qsTP.submit(qsInner2);
```

Since there are no return values to deal with, we can spawn both and then end this call. We could also spawn one and have the other keep running in this thread.

Of course, we need to have a way of waiting for all of the tasks to complete before using the sorted list...

Waiting for your tasks to complete

Two available tools are Semaphores and CountDownLatch. However, they can have big costs associated with using them on large values.

If you know that there are $N$ things that need to be processed, you could start all of your tasks running and then acquire($N$) on a semaphore. As it task completed it could release(COUNT) where COUNT was how many things it had processed.

Similarly, you could set a countdown latch to $N$ and have each task countdown the number of things it completed (this is more of a hassle in Java since the CountDownLatch class doesn’t have a method to count by more than 1 at a time).

You could also have each submitted task return a Future and add it to a list. Then after all the tasks were known to be submitted, you could pull items from that list and get() on them, until the list was empty.

If you knew everything that will be has been put into a thread pool, you could even do a shutdown on the pool and then wait for termination…
Recursion approach #2: Sequential Threshold

With many optimization problems there will be a threshold beyond which it’s better to do things the “simple” way rather than the more “complex” way. For example, it turns out that if your list is smaller than 8 elements, you are statistically better off using **InsertionSort** rather than **QuickSort**.

The same is true with concurrent programming; you will reach a point where the cost of the sequential solution is less than the cost of the concurrent solution due to the overhead associated with it.

These can be (and are) found either through proofs or through empirical explorations, and they will often vary based on the specific architecture of a machine.

Our **Quicksort** could end up creating N/2 different tasks!

Quicksort: example of avoiding task explosion

One approach might use logic along the lines of:

```java
synchronized(sqTP) { //qsTP stands for quicksortThreadPool
   spawn1=(((TPE)qsTP).getPoolSize() > ((TPE)qsTP).getTaskCount());
   if (spawn1) {
      ConcurrentQuicksortInProgress qsInner1 =
         new ConcurrentQuicksort(
            arrayToSort, first, pivotPos-1,qsTP);
      qsCall1 = qsTP.submit(qsInner1);
   }
}
if (!spawn1) parallelQuicksort(first, pivotPos-1);
parallelQuicksort(pivotPos+1, last);
if (spawn1) qsCall1.get();
```

This says that if there's room in the thread pool, send the task there, otherwise run it here. The QuickSort of the other half is done in this thread, possible while the other was being run in another thread.

We could still be spending a lot of time towards the end submitting small tasks to the pool, so we might set a threshold below which we just run sequentially (like n/CPUs).
**Finding a balance example: SplitQuickMergeSort**

Split your list into CPUs equal-sized segments.

Call regular Quicksort on each of those segments in parallel.

Do a pair-wise merge of those results, again in parallel (using fewer processors as you merge).

While not optimal since the merge stages don’t use all of the available CPUs, for “small” sizes this approach worked very well on my quad-core machine (where the “small” sizes that I tested on were as large as 10 million ints).