CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Parallelism with Fork-Join

Executor Interfaces

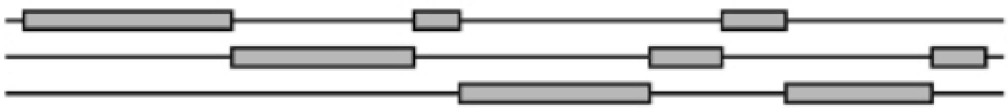
- The java.util.concurrent package defines three executor interfaces
- **Executor**, a simple interface that supports launching new tasks.
  - provides a single method, *execute*. If r is a **Runnable** object, and e is an **Executor** object you can replace

    (new Thread(r)).start(); with e.execute(r);
- **ExecutorService**, a subinterface of **Executor**, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself
  - The **ExecutorService** interface supplements execute with a *submit* method.
  - Like *execute*, *submit* accepts **Runnable** objects, but also accepts **Callable** objects, which allow the task to return a value.
  - The *submit* method returns a **Future** object, which is used to retrieve the **Callable** return value and to manage the status of both **Callable** and **Runnable** tasks.
- **ScheduledExecutorService**, a subinterface of **ExecutorService**, supports future and/or periodic execution of tasks.

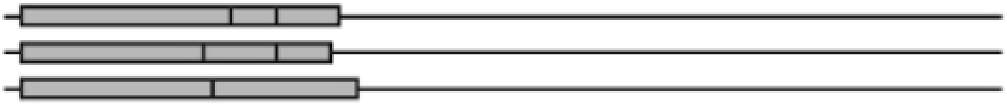
Fork/Join

- The fork/join framework is an implementation of the ExecutorService interface that helps you **take advantage of multiple processors**.
- It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

Concurrency vs Parallelism



Concurrent, non-parallel execution



Concurrent, parallel execution

Java Threads (Concurrent) vs. Fork/Join Framework (Parallel)

- Threads
  - When task is relatively large and self-contained
  - Usually when you are waiting for something so you would benefit even if there is only one processor (e.g. GUI)
- Fork/Join or parallel streams
  - When task starts large but can be broken up repeatedly into smaller pieces combined for final result.
  - No benefit if there is only one processor
  - As with any ExecutorService implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a *work-stealing* algorithm.
  - Worker threads that run out of things to do can steal tasks from other threads that are still busy.

Basic Use

- The center of the fork/join framework is the **ForkJoinPool** class, an extension of the AbstractExecutorService class.
- **ForkJoinPool** implements the core work-stealing algorithm and can execute **ForkJoinTask** processes.
- The first step for using the fork/join framework is to write code that performs a segment of the work. Your code should look similar to the following pseudocode:

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

- Wrap this code in a **ForkJoinTask** subclass, typically using one of its more specialized types, either **RecursiveTask** (which can return a result) or **RecursiveAction**. Override *compute()*
- After your **ForkJoinTask** subclass is ready, create the object that represents all the work to be done and pass it to the *invoke()* method of a **ForkJoinPool** instance.

Example

- Summing Large Array of doubles
- Sequential version:

```
public class MathUtils {
  public static double arraySum(double[] nums,
                                int lowerIndex, int upperIndex) {
    double sum = 0;
    for(int i=lowerIndex; i<=upperIndex; i++) {
      sum += nums[i];
    }
    return(sum);
  }

  public static double arraySum(double[] nums) {
    return(arraySum(nums, 0, nums.length-1));
  }

  ...
}
```

Parallel Version

- Create your class that extends RecursiveTask

```
public class ParallelArraySummer extends RecursiveTask<Double> {
  private static final int PARALLEL_CUTOFF = 1000;
  private double[] nums;
  private int lowerIndex, upperIndex;

  public ParallelArraySummer(double[] nums,
                             int lowerIndex, int upperIndex) {
    this.nums = nums;
    this.lowerIndex = lowerIndex;
    this.upperIndex = upperIndex;
  }
```

Parallel Version

- Override compute

```
@Override
protected Double compute() {
  int range = upperIndex - lowerIndex;
  if (range <= PARALLEL_CUTOFF) {
    return(MathUtils.arraySum(nums, lowerIndex, upperIndex));
  } else {
    int middleIndex = lowerIndex + range/2;
    ParallelArraySummer leftSummer =
      new ParallelArraySummer(nums, lowerIndex, middleIndex);
    ParallelArraySummer rightSummer =
      new ParallelArraySummer(nums, middleIndex+1, upperIndex);
    leftSummer.fork();
    Double rightSum = rightSummer.compute();
    Double leftSum = leftSummer.join();
    return(leftSum + rightSum);
  }
}
}
```

Parallel Version

- The client Class

```
public class MathUtils {
  private static final ForkJoinPool FORK_JOIN_POOL = new ForkJoinPool();
  ...

  public static Double arraySumParallel(double[] nums) {
    return(FORK_JOIN_POOL.invoke(new ParallelArraySummer(nums,
                                                         0,
                                                         nums.length-1)));
  }
}
```

Results

Array Size	Sequential Time (Seconds)	Parallel Time (Seconds)
1,000	0.002	0.001
10,000	0.002	0.002
100,000	0.003	0.003
1,000,000	0.004	0.003
10,000,000	0.011	0.010
100,000,000	0.106	0.055

- Little benefit of parallel approach except with extremely large arrays

Deciding on Sequential vs. Parallel Approaches

- Parallel is often better
  - Problem is large
    - E.g., 5,000,000 element array
  - Computations for smallest size is expensive
    - E.g., Sum of some expensive operation, finding primes
  - Your computer has many processors
    - Two or more, but more is better
- Sequential is often better
  - Problem is small
    - E.g., 5,000 element array
  - Computation for smallest size is fast
    - E.g., sum of doubles
  - Your computer has few processors
    - Obviously, always use sequential for 1-core machines