Parallel Computing
CSCI 201L

Jeffrey Miller, Ph.D.
jeffrey.miller@usc.edu

HTTP://WWW-SCF.USC.EDU/~CSCI201
Outline

- Parallel Computing
- Programming
Parallel computing studies software systems where components located on networked components communicate through message passing:

- Individual processes have only a partial knowledge of the problem.
- Message passing can be accomplished through remote procedure calls (RPC) or message queues (shared memory).
- Parallel computing is a term also used for programs with multiple pieces running within multiple processors or cores in the same computer.
The fork/join framework is an implementation of the \texttt{ExecutorService} interface that allows taking advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively, allowing all available processing power. We can create threads and add them to a thread pool. Worker threads that run out of things to do can steal tasks for other threads that are still busy (work-stealing algorithm). The \texttt{ForkJoinPool} class implements the core work-stealing algorithm and can execute \texttt{ForkJoinTask} processes.
Fork/Join Framework Structure

- Pseudocode for the fork/join framework is:
  
  ```java
  if (condition)
      do the work directly
  else
      split work into multiple pieces
      invoke the pieces and wait for results
  ```

- The above code should be inside a child of `ForkJoinTask`
  - `RecursiveTask` can return a result
  - `RecursiveAction` does not return a result

- Create the object that represents the work to be done and pass it to the `invoke()` or `execute()` method of a `ForkJoinPool`
The `ForkJoinTask` class has a method `compute()` that should be overridden.

- The `compute()` method will contain the main computation performed by the task.
- The `fork()` method allows asynchronous execution (starts the thread and calls `compute()`).
- The `join()` method will not proceed until the task’s `compute()` method has completed as a result of the call to `fork()`.
- The `invoke()` method on a `ForkJoinPool` will execute the `compute()` method asynchronously and return the value (through a `fork()` then a `join()` call).
- The `execute()` method on a `ForkJoinPool` will execute the `compute()` method asynchronously but will not return any value (through a `fork()` call with no `join()`).
Running Time

- Any time you fork a task, there is overhead in moving that to a new CPU, executing it, and then getting the response.
- Just because you are parallelizing code does not mean that you will have an improvement in execution speed.
- If you fork more threads than you have CPUs, the threads will execute in a concurrent manner (time-slicing) in each CPU.
public class FibonacciNoParallel {
    public static void main(String [] args) {
        int index = 30;
        long before = System.currentTimeMillis();
        Fibonacci fib = new Fibonacci(index);
        int num = fib.compute();
        long after = System.currentTimeMillis();
        System.out.println("time without parallelism = " + (after - before));
        System.out.println(index + "th Fibonacci number = " + num);
    }
}

static class Fibonacci {
    private int n;
    Fibonacci(int n) {
        this.n = n;
    }
    protected Integer compute() {
        if (n == 0 || n == 1) {
            return n;
        }
        Fibonacci f1 = new Fibonacci(n - 1);
        Fibonacci f2 = new Fibonacci(n - 2);
        for (int i=0; i < 30000; i++) {}  
        this.n = f1.compute() + f2.compute();
        return this.n;
    }
}

time without parallelism = 7614
30th Fibonacci number = 832040

time without parallelism = 9168
30th Fibonacci number = 832040

time without parallelism = 7645
30th Fibonacci number = 832040
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class FibonacciParallel1 {
    public static void main(String[] args) {
        int index = 30;
        long before = System.currentTimeMillis();
        ForkJoinPool pool = new ForkJoinPool();
        Fibonacci fib = new Fibonacci(index);
        pool.execute(fib);
        pool.shutdown();
        long after = System.currentTimeMillis();
        System.out.println("time with parallelism = " + (after - before));
        System.out.println(index + "th Fibonacci number = " + fib.getNum());
    }

    static class Fibonacci extends RecursiveTask<Integer> {
        private static final long serialVersionUID = 1;
        private int n;
        Fibonacci(int n) {
            this.n = n;
        }
        public int getNum() {
            return this.n;
        }
        protected Integer compute() {
            if (n == 0 || n == 1) {
                return n;
            }
            Fibonacci f1 = new Fibonacci(n - 1);
            Fibonacci f2 = new Fibonacci(n - 2);
            for (int i=0; i < 30000; i++) {}
            f1.fork();
            f2.fork();
            this.n = f2.join() + f1.join();
            return this.n;
        }
    }
}
Parallel Fibonacci Example #2

```java
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
public class FibonacciParallel2 {
    public static void main(String[] args) {
        int index = 30;
        long before = System.currentTimeMillis();
        ForkJoinPool pool = new ForkJoinPool();
        Fibonacci fib = new Fibonacci(index);
        pool.execute(fib);
        pool.shutdown();
        while (pool.getActiveThreadCount() > 0) {
            Thread.yield();
        }
        long after = System.currentTimeMillis();
        System.out.println("time with parallelism = " + (after - before));
        System.out.println(index + "th Fibonacci number = " + fib.getNum());
    }
    static class Fibonacci extends RecursiveTask<Integer> {
        private static final long serialVersionUID = 1;
        private int n;
        Fibonacci(int n) {
            this.n = n;
        }
        public int getNum() {
            return this.n;
        }
        protected Integer compute() {
            if (n == 0 || n == 1) {
                return n;
            }
            Fibonacci f1 = new Fibonacci(n - 1);
            Fibonacci f2 = new Fibonacci(n - 2);
            f1.fork();
            f2.fork();
            for (int i=0; i < 30000; i++) {} // AUTO-INITIALIZED
            this.n = f2.join() + f1.join();
            return this.n;
        }
    }
}
```

time with parallelism = 6037
30th Fibonacci number = 832040

time with parallelism = 7126
30th Fibonacci number = 832040

time with parallelism = 5897
30th Fibonacci number = 832040
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class FibonacciParallel3 {
    public static void main(String[] args) {
        int index = 30;
        long before = System.currentTimeMillis();
        ForkJoinPool pool = new ForkJoinPool();
        Fibonacci fib = new Fibonacci(index);
        int num = pool.invoke(fib);
        pool.shutdown();
        long after = System.currentTimeMillis();
        System.out.println("time with parallelism = "+(after - before));
        System.out.println(index + "th Fibonacci number = " + num);
    }

    static class Fibonacci extends RecursiveTask<Integer> {
        private static final long serialVersionUID = 1;
        private int n;
        Fibonacci(int n) {
            this.n = n;
        }
        public int getNum() {
            return this.n;
        }
        protected Integer compute() {
            if (n == 0 || n == 1) {
                return n;
            }
            Fibonacci f1 = new Fibonacci(n - 1);
            Fibonacci f2 = new Fibonacci(n - 2);
            f1.fork();
            f2.fork();
            for (int i=0; i < 30000; i++) {} // dummy code to delay execution
            this.n = f2.join() + f1.join();
            return this.n;
        }
    }
}
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class FibonacciParallel4 {
  public static void main(String [] args) {
    int index = 30;
    int poolSize = Runtime.getRuntime().availableProcessors();
    for (int i=1; i <= poolSize; i++) {
      long before = System.currentTimeMillis();
      ForkJoinPool pool = new ForkJoinPool(i);
      Fibonacci fib = new Fibonacci(index);
      int num = pool.invoke(fib);
      pool.shutdown();
      long after = System.currentTimeMillis();
      System.out.println("time with parallelism " + i + " = " + (after - before));
      System.out.println(index + "th Fibonacci number = " + num);
    }
  }
}

static class Fibonacci extends RecursiveTask<Integer> {
  private static final long serialVersionUID = 1;
  private int n;
  Fibonacci(int n) {
    this.n = n;
  }
  public int getNum() {
    return this.n;
  }
  protected Integer compute() {
    if (n == 0 || n == 1) {
      return n;
    }
    Fibonacci f1 = new Fibonacci(n - 1);
    Fibonacci f2 = new Fibonacci(n - 2);
    f1.fork();
    f2.fork();
    for (int i=0; i < 30000; i++) {} // Dummy computation
    this.n = f2.join() + f1.join();
    return this.n;
  }
}

time with parallelism 1 = 15968
30th Fibonacci number = 832040

time with parallelism 2 = 9266
30th Fibonacci number = 832040

time with parallelism 3 = 7310
30th Fibonacci number = 832040

time with parallelism 4 = 6864
30th Fibonacci number = 832040
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class FibonacciParallel5 {
  public static void main(String [] args) {
    int index = 30;
    int poolSize = Runtime.getRuntime().availableProcessors();
    for (int i=poolSize; i >= 1; i--) {
      long before = System.currentTimeMillis();
      ForkJoinPool pool = new ForkJoinPool(i);
      Fibonacci fib = new Fibonacci(index);
      int num = pool.invoke(fib);
      pool.shutdown();
      long after = System.currentTimeMillis();
      System.out.println("time with parallelism "+i+" = "+(after-before));
      System.out.println(index + "th Fibonacci number = " + num);
    }
  }

  static class Fibonacci extends RecursiveTask<Integer> {
    private static final long serialVersionUID = 1;
    private int n;
    Fibonacci(int n) {
      this.n = n;
    }
    public int getNum() {
      return this.n;
    }
    protected Integer compute() {
      if (n == 0 || n == 1) {
        return n;
      }
      Fibonacci f1 = new Fibonacci(n - 1);
      Fibonacci f2 = new Fibonacci(n - 2);
      f1.fork();
      f2.fork();
      for (int i=0; i < 30000; i++) {} // Simulate some computation
      this.n = f2.join() + f1.join();
      return this.n;
    }
  }
}
public class FibonacciParallel6 {

    public static void main(String[] args) {
        int index = 30;
        int poolSize = Runtime.getRuntime().availableProcessors();
        for (int i = 5 * poolSize; i >= 1; i--) {
            long totalTime = 0;
            int num = 0;
            for (int j = 0; j < 20; j++) {
                long before = System.nanoTime();
                ForkJoinPool pool = new ForkJoinPool(i);
                Fibonacci fib = new Fibonacci(index);
                num = pool.invoke(fib);
                pool.shutdown();
                long after = System.nanoTime();
                totalTime += (after - before);
            }
            long averageTotalTime = totalTime / 20;
            System.out.println("time with parallelism "+ i + "+ = "+ averageTotalTime);
        }
    }

    static class Fibonacci extends RecursiveTask<Integer> {
        private static final long serialVersionUID = 1;
        private int n;
        Fibonacci(int n) {
            this.n = n;
        }
        public int getNum() {
            return this.n;
        }
        protected Integer compute() {
            if (n == 0 || n == 1) {
                return n;
            }
            Fibonacci f1 = new Fibonacci(n - 1);
            Fibonacci f2 = new Fibonacci(n - 2);
            f1.fork();
            f2.fork();
            for (int i = 0; i < 30000; i++) {}
            this.n = f2.join() + f1.join();
            return this.n;
        }
    }
}
```java
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class FibonacciParallel6 {
    public static void main(String[] args) {
        int index = 30;
        int poolSize = Runtime.getRuntime().availableProcessors();
        for (int i=5*poolSize; i >= 1; i--) {
            long totalTime = 0;
            int num = 0;
            for (int j=0; j < 20; j++) {
                long before = System.nanoTime();
                ForkJoinPool pool = new ForkJoinPool(i);
                Fibonacci fib = new Fibonacci(index);
                num = pool.invoke(fib);
                pool.shutdown();
                long after = System.nanoTime();
                totalTime += (after - before);
            }
            long averageTotalTime = totalTime / 20;
            System.out.println("time with parallelism "+i+" = "+averageTotalTime);
        }
    }

    static class Fibonacci extends RecursiveTask<Integer> {
        private static final long serialVersionUID = 1;
        private int n;
        Fibonacci(int n) {
            this.n = n;
        }
        public int getNum() {
            return this.n;
        }
        protected Integer compute() {
            if (n == 0 || n == 1) {
                return n;
            }
            Fibonacci f1 = new Fibonacci(n - 1);
            Fibonacci f2 = new Fibonacci(n - 2);
            for (int i=0; i < 30000; i++) {}
            this.n = f2.compute() + f1.compute();
            return this.n;
        }
    }
}
```

```

time with parallelism 20 = 6558
time with parallelism 19 = 6664
time with parallelism 18 = 7057
time with parallelism 17 = 7598
time with parallelism 16 = 6691
time with parallelism 15 = 7205
time with parallelism 14 = 7149
time with parallelism 13 = 7083
time with parallelism 12 = 6647
time with parallelism 11 = 6989
time with parallelism 10 = 7059
time with parallelism 9 = 6927
time with parallelism 8 = 6849
time with parallelism 7 = 7011
time with parallelism 6 = 7076
time with parallelism 5 = 6843
time with parallelism 4 = 6519
time with parallelism 3 = 6947
time with parallelism 2 = 6849
time with parallelism 1 = 6819
```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
public class ParallelMergeSort {
    public static void main(String[] args) {
        int SIZE = 2_000_000;
        int[] list = new int[SIZE];

        for (int i = 0; i < SIZE; i++) {
            list[i] = (int)(Math.random() * Integer.MAX_VALUE);
        }

        int numProcessors = Runtime.getRuntime().availableProcessors();
        System.out.println("num processors: "+numProcessors);

        long[] timing = new long[numProcessors*2+1];

        for (int i=1; i <= numProcessors * 2; i++) {
            timing[i] = parallelMergeSort((int[])list.clone(), i);
            System.out.println(i + " processors="+ timing[i]+" ms");
        }

        public static long parallelMergeSort(int[] list, int proc) {
            long before = System.currentTimeMillis();
            ForkJoinPool pool = new ForkJoinPool(proc);
            pool.invoke(new SortTask(list));
            pool.shutdown();
            while (!pool.isTerminated()) {
                Thread.yield();
            }
            long after = System.currentTimeMillis();
            long time = after - before;
            return time;
        }

        private static class SortTask extends RecursiveAction {
            private static final long serialVersionUID = 1;
            private int[] list;
            SortTask(int[] list) {
                this.list = list;
            }

            protected void compute() {
                if (list.length < 2) return; // base case
                int firstHalf = new int[list.length / 2];
                System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
                int secondLength = list.length - list.length / 2;
                int[] secondHalf = new int[secondLength];
                System.arraycopy(list, list.length / 2, secondHalf, 0, secondLength);

                public static void merge(int[] list1, int[] list2, int[] merged) {
                    int i1 = 0, i2 = 0, i3 = 0; // index in list1, list2, out
                    while (i1 < list1.length && i2 < list2.length) {
                        merged[i3++] = (list1[i1] < list2[i2]) ? list1[i1++] : list2[i2++];
                    }
                    // any trailing ends
                    while (i1 < list1.length) { merged[i3++] = list1[i1++]; }
                    while (i2 < list2.length) { merged[i3++] = list2[i2++]; }
                }
            }
        }
    }
}
Outline

- Parallel Computing
- Program
Program

- Modify the ParallelMergeSort code to sort with merge sort without threads. Does it execute faster than the ParallelMergeSort?
- Does the input size matter for relative run times?