Outline

• Parallel Computing
• Program
Parallel computing studies software systems where components located on connected components communicate through message passing

› Individual threads have only a partial knowledge of the problem
› Message passing is typically accomplished through message queues, which is a form of shared memory
› Parallel computing is a term used for programs that operate within a shared memory space with multiple processors or cores
Redundant Hardware Determination

- To determine what CPU or CPUs you have in your computer:
  - On Windows, go to 📱 -> 🛠️, then click System, then About.
  - On Mac, go to 🍊 -> About this Mac.
Fork/Join Framework

- The fork/join framework is an implementation of the `ExecutorService` interface that allows taking advantage of multiple cores or 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.
- The `ForkJoinPool` class implements the core algorithm for parallel computing in Java and can execute `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()`).
## Multi-Threading vs Parallel Comparison

<table>
<thead>
<tr>
<th></th>
<th>Multi-Threading</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class/Interface from which to inherit</td>
<td>Thread/Runnable</td>
<td>RecursiveAction or RecursiveTask</td>
</tr>
<tr>
<td>Method to override</td>
<td>void run()</td>
<td>protected abstract void compute()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(RecursiveAction)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>protected abstract V compute()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(RecursiveTask)</td>
</tr>
<tr>
<td>Method called to start thread</td>
<td>public void start()</td>
<td>public final ForkJoinTask&lt;V&gt; fork()</td>
</tr>
<tr>
<td>Managing class</td>
<td>ExecutorService</td>
<td>ForkJoinPool</td>
</tr>
<tr>
<td>Method to call on managing class to start thread</td>
<td>void execute(Runnable)</td>
<td>public void execute(ForkJoinTask&lt;?&gt;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(no return value)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>public T invoke(ForkJoinTask&lt;T&gt;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(with return value)</td>
</tr>
<tr>
<td>Method to stop current thread from continuing until another thread completes</td>
<td>public final void join()</td>
<td>public final V join()</td>
</tr>
</tbody>
</table>
Running Time

- Any time you fork a task, there is overhead in moving that to a new CPU, executing it, and 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 SumNoParallel {
    public static void main(String [] args) {
        long maxNumber = 1_000_000_000;
        long before = System.currentTimeMillis();
        Sum sum = new Sum(maxNumber);
        long num = sum.compute();
        long after = System.currentTimeMillis();
        System.out.println("time without parallelism = " + (after - before));
        System.out.println("SUM(0.." + maxNumber + ") = " + num);
    }
}

static class Sum {
    private long maxNum;
    Sum(long maxNum) {
        this.maxNum = maxNum;
    }
    protected Long compute() {
        long sum = 0;
        for (int i=0; i <= maxNum; i++) {
            sum += i;
        }
        return sum;
    }
}

Not Parallel Sum

USC CSCI 201L 10/14

time without parallelism = 710
SUM(0..1000000000) = 5000000000500000000

time without parallelism = 716
SUM(0..1000000000) = 5000000000500000000

time without parallelism = 695
SUM(0..1000000000) = 5000000000500000000
import java.util.concurrent.*; // import * to save space

public class SumParallel {
    public static void main(String [] args) {
        long minNumber = 0;
        long maxNumber = 1_000_000_000;
        long before = System.currentTimeMillis();
        ForkJoinPool pool = new ForkJoinPool();
        int numProcessors = Runtime.getRuntime().availableProcessors();
        long numElements = maxNumber / numProcessors;
        Sum sum = new Sum(minNumber, maxNumber, numElements);
        pool.execute(sum);
        pool.shutdown();
        sum.join();
        long num = sum.getSum();
        long after = System.currentTimeMillis();
        System.out.println("time with parallelism = " + (after - before));
        System.out.println("SUM(0.." + maxNumber + ") = " + num);
    }
    static class Sum extends RecursiveTask<Long> {
        protected Long compute() {
            if ((maxNum - minNum) < numElements) {
                for (long i=minNum; i <= maxNum; i++) {
                    sum += i;
                }
                return sum;
            }
            Sum s1 = new Sum(minNum,minNum+(maxNum-minNum)/2,numElements);
            Sum s2 = new Sum(minNum+(maxNum-minNum)/2+1,maxNum,numElements);
            s1.fork();
            s2.fork();
            sum = s1.join() + s2.join();
            return sum;
        } // ends compute()
    } // ends Sum
} // ends SumParallel

time without parallelism = 338
SUM(0..1000000000) = 500000000500000000

time without parallelism = 350
SUM(0..1000000000) = 500000000500000000

time without parallelism = 353
SUM(0..1000000000) = 500000000500000000
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

class ParallelMergeSort {
  public static void main(String[] args) {
    final long 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[] 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");
    }
    return after - before;
  }

  class SortTask extends RecursiveAction {
    public 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);
      // recursively sort the two halves
      new SortTask(firstHalf).invoke();
      new SortTask(secondHalf).invoke();
      // merge halves together
      merge(firstHalf, secondHalf, list);
    }
  }

  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) {
      if (list1[i1] < list2[i2]) {
        merged[i3++] = list1[i1++];
      } else {
        merged[i3++] = list2[i2++];
      }
    }
    // any trailing ends
    while (i1 < list1.length) {
      merged[i3++] = list1[i1++];
    }
    while (i2 < list2.length) {
      merged[i3++] = list2[i2++];
    }
  }

  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[] 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");
    }
    return after - before;
  }
}

private static class SortTask extends RecursiveAction {
  public 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);
    // recursively sort the two halves
    new SortTask(firstHalf).invoke();
    new SortTask(secondHalf).invoke();
    // merge halves together
    merge(firstHalf, secondHalf, list);
  }
}

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) {
    if (list1[i1] < list2[i2]) {
      merged[i3++] = list1[i1++];
    } else {
      merged[i3++] = list2[i2++];
    }
  }
  // any trailing ends
  while (i1 < list1.length) {
    merged[i3++] = list1[i1++];
  }
  while (i2 < list2.length) {
    merged[i3++] = list2[i2++];
  }
}

num processors: 4
  1 processors=1936 ms
  2 processors=1790 ms
  3 processors=1671 ms
  4 processors=1822 ms
  5 processors=1756 ms
  6 processors=1678 ms
  7 processors=1634 ms
  8 processors=1950 ms
Outline

• Parallel Computing
• 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?