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
- 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, Run ( + R) and type `msinfo32`
  - On Mac, go to 🍊 -> About this Mac -> System Report
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><strong>Class/Interface from which to inherit</strong></td>
<td>Thread/Runnable</td>
<td>RecursiveAction or RecursiveTask</td>
</tr>
<tr>
<td><strong>Method to override</strong></td>
<td><code>void run()</code></td>
<td><code>protected abstract void compute()</code> (RecursiveAction)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>protected abstract V compute()</code> (RecursiveTask)</td>
</tr>
<tr>
<td><strong>Method called to start thread</strong></td>
<td><code>public void start()</code></td>
<td><code>public final ForkJoinTask&lt;T&gt; fork()</code></td>
</tr>
<tr>
<td><strong>Managing class</strong></td>
<td>ExecutorService</td>
<td>ForkJoinPool</td>
</tr>
<tr>
<td><strong>Method to call on managing class to start thread</strong></td>
<td><code>void execute(Runnable)</code></td>
<td><code>public void execute(ForkJoinTask&lt;?&gt;)</code> (RecursiveAction)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>public T invoke(ForkJoinTask&lt;T&gt;)</code> (RecursiveTask)</td>
</tr>
<tr>
<td><strong>Method to stop current thread from continuing until another thread completes</strong></td>
<td><code>public final void join()</code></td>
<td><code>public final V join()</code></td>
</tr>
</tbody>
</table>

---

**USC Viterbi**  
School of Engineering
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 similar to multi-threading) 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);
    }
}

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;
    }
}
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class SumParallel {
    public static void main(String[] args) {
        long minNumber = 0;
        long maxNumber = 1_000_000_000;
        int numThreads = 4;
        long before = System.currentTimeMillis();
        ForkJoinPool pool = new ForkJoinPool();
        SumP sum[] = new SumP[numThreads];
        long start = minNumber;
        long end = maxNumber / numThreads;
        for (int i=0; i < numThreads; i++) {
            sum[i] = new SumP(start, end);
            start = end + 1;
            end += (maxNumber / numThreads);
            pool.execute(sum[i]); // no return value, so we will join later
        }
        long num = 0;
        for (int i=0; i < numThreads; i++) {
            num += sum[i].join();
        }
        long after = System.currentTimeMillis();
        System.out.println("time with parallelism = " + (after-before));
        System.out.print("SUM(" + minNumber + "." + maxNumber + ") = ");
        System.out.println(num);
    }
}

class SumP extends RecursiveTask<Long> {
    private long minNum;
    private long maxNum;
    public static final long serialVersionUID = 1;
    public SumP(long minNum, long maxNum) {
        this.minNum = minNum;
        this.maxNum = maxNum;
    }
    protected Long compute() {
        long sum = 0;
        for (long i=minNum; i <= maxNum; i++) {
            sum += i;
        }
        return sum;
    }
}

time with parallelism = 457
SUM(0..1000000000) = 500000000500000000

time with parallelism = 436
SUM(0..1000000000) = 500000000500000000

time with parallelism = 433
SUM(0..1000000000) = 500000000500000000
public class NonParallelMergeSort {
    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);
        }
        long timing = 0;
        long sum = 0;
        // run it 8 times to see if there are variations
        for (int i=0; i < 8; i++) {
            timing = nonParallelMergeSort((int[])list.clone());
            System.out.println(timing + " ms");
            sum += timing;
        }
        System.out.println("average = " + (sum / 8) + " ms");
    }

    public static long nonParallelMergeSort(int[] list) {
        long before = System.currentTimeMillis();
        new SortTask(list).compute();
        long after = System.currentTimeMillis();
        long time = after - before;
        return time;
    }

    private static class SortTask {
        private int[] list;
        SortTask(int[] list) {
            this.list = list;
        }
        protected void compute() {
            if (list.length < 2) return; // base case
            // split into halves
            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).compute();
            new SortTask(secondHalf).compute();
            // 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) {
            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++];
        }
    }
}
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);

        // timing[i] : time to sort on i processors
        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 {
        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
            // split into halves
            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
            SortTask st1 = new SortTask(firstHalf);
            SortTask st2 = new SortTask(secondHalf);
            st1.fork();
            st2.fork();
            st1.join();
            st2.join();
        }
    }

    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++];
        }
        while (i1 < list1.length) {
            merged[i3++] = list1[i1++];
        }
        while (i2 < list2.length) {
            merged[i3++] = list2[i2++];
        }
    }
}

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
        // split into halves
        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
        SortTask st1 = new SortTask(firstHalf);
        SortTask st2 = new SortTask(secondHalf);
        st1.fork();
        st2.fork();
        st1.join();
        st2.join();
    }
}

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 {
    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
        // split into halves
        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
        SortTask st1 = new SortTask(firstHalf);
        SortTask st2 = new SortTask(secondHalf);
        st1.fork();
        st2.fork();
        st1.join();
        st2.join();
    }
}

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
        // split into halves
        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
        SortTask st1 = new SortTask(firstHalf);
        SortTask st2 = new SortTask(secondHalf);
        st1.fork();
        st2.fork();
        st1.join();
        st2.join();
    }
}

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++];
    }
    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 split the array into the same number of sub-arrays as processors/cores on your computer. Does that provide a lower execution time than splitting the array into two sub-arrays? Why or why not?