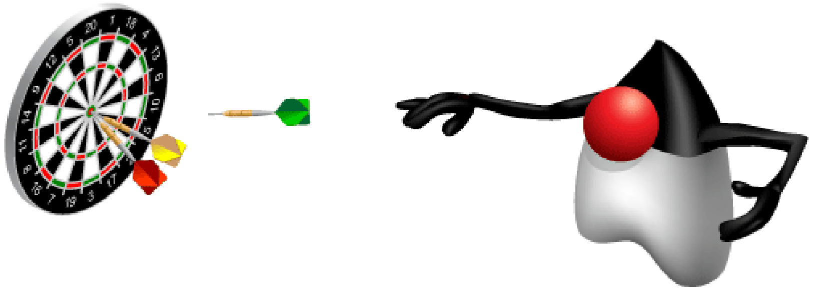
16

The Fork-Join Framework

Objectives

After completing this lesson, you should be able to:

- Apply the Fork-Join framework



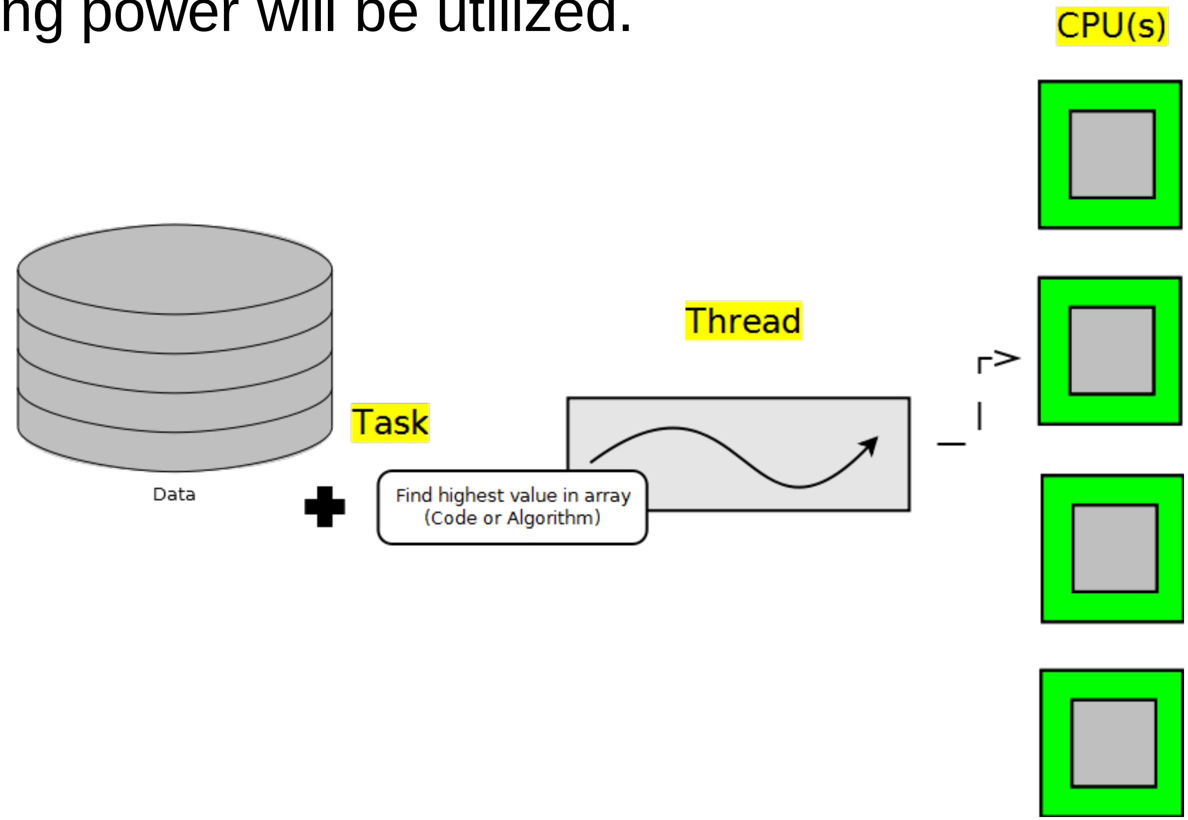
Parallelism

Modern systems contain multiple CPUs. Taking advantage of the processing power in a system requires you to execute tasks in parallel on multiple CPUs.

- Divide and conquer: A task should be divided into subtasks. You should attempt to identify those subtasks that can be executed in parallel.
- Some problems can be difficult to execute as parallel tasks.
- Some problems are easier. Servers that support multiple clients can use a separate task to handle each client.
- Be aware of your hardware. Scheduling too many parallel tasks can negatively impact performance.

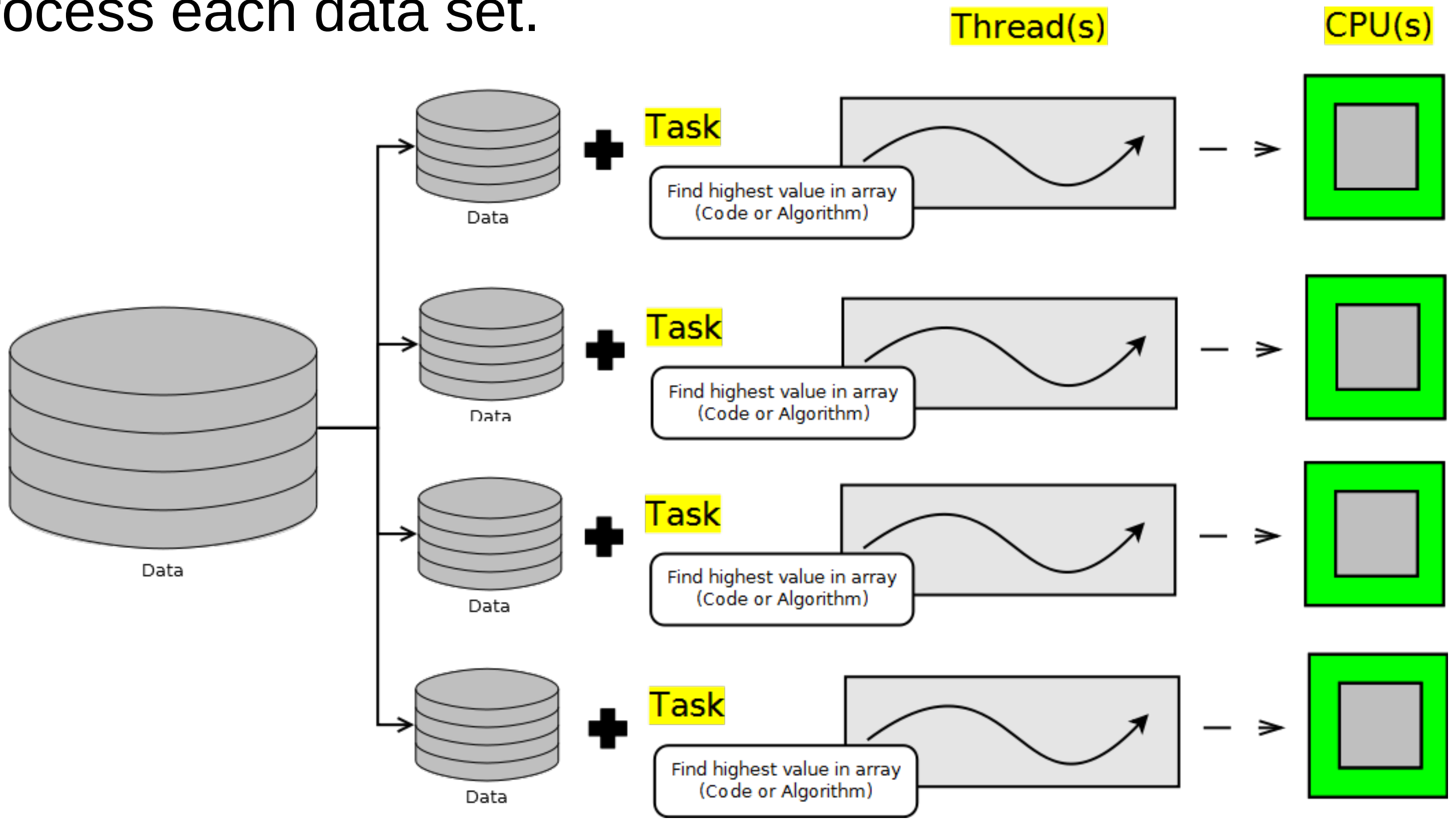
Without Parallelism

Modern systems contain multiple CPUs. If you do not leverage threads in some way, only a portion of your system's processing power will be utilized.



Naive Parallelism

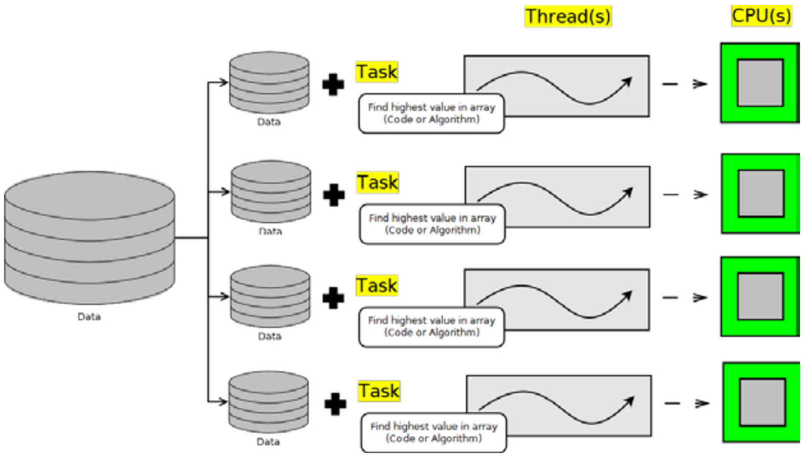
A simple parallel solution breaks the data to be processed into multiple sets: one data set for each CPU and one thread to process each data set.



The Need for the Fork-Join Framework

Splitting datasets into equal sized subsets for each thread to process has a couple of problems. Ideally all CPUs should be fully utilized until the task is finished, but:

- CPUs may run at different speeds
- Non-Java tasks require CPU time and may reduce the time available for a Java thread to spend executing on a CPU
- The data being analyzed may require varying amounts of time to process

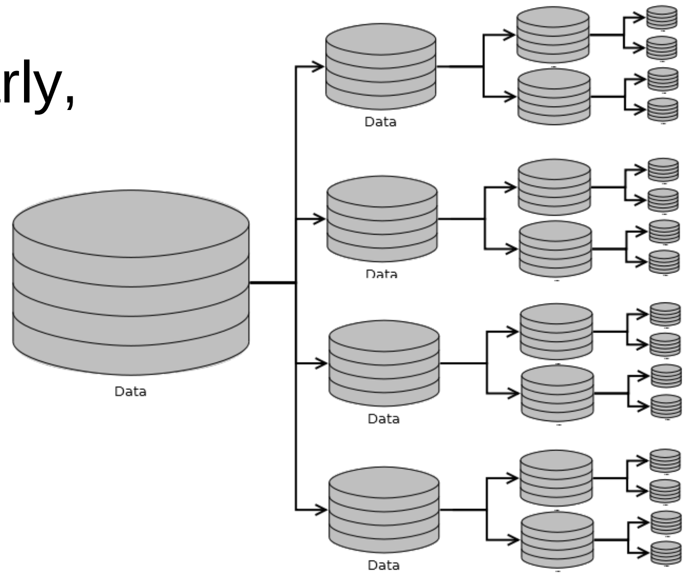


Work-Stealing

To keep multiple threads busy:

- Divide the data to be processed into a large number of subsets
- Assign the data subsets to a thread's processing queue
- Each thread will have many subsets queued

If a thread finishes all its subsets early, it can "steal" subsets from another thread.



A Single-Threaded Example

```
int[] data = new int[1024 * 1024 * 256]; //1G

for (int i = 0; i < data.length; i++) {
    data[i] = ThreadLocalRandom.current().nextInt();
}

int max = Integer.MIN_VALUE;
for (int value : data) {
    if (value > max) {
        max = value;
    }
}
System.out.println("Max value found:" + max);
```

A very large dataset

Fill up the array with values.

Sequentially search the array for the largest value.

`java.util.concurrent.ForkJoinTask<V>`

A `ForkJoinTask` object represents a task to be executed.

- A task contains the code and data to be processed. Similar to a `Runnable` or `Callable`.
- A huge number of tasks are created and processed by a small number of threads in a Fork-Join pool.
  - A `ForkJoinTask` typically creates more `ForkJoinTask` instances until the data to processed has been subdivided adequately.
- Developers typically use the following subclasses:
  - `RecursiveAction`: When a task does not need to return a result
  - `RecursiveTask`: When a task needs to return a result

RecursiveTask Example

```
public class FindMaxTask extends RecursiveTask<Integer> {
    private final int threshold;
    private final int[] myArray;
    private int start;
    private int end;

    public FindMaxTask(int[] myArray, int start, int end,
int threshold) {
        // copy parameters to fields
    }
    protected Integer compute() {
        // shown later
    }
}
```

Result type of the task

The data to process

Where the work is done.
Notice the generic return type.

compute Structure

```
protected Integer compute() {
    if DATA_SMALL_ENOUGH {
        PROCESS_DATA
        return RESULT;
    } else {
        SPLIT_DATA_INTO_LEFT_AND_RIGHT_PARTS
        TASK t1 = new TASK(LEFT_DATA);
        t1.fork();
        TASK t2 = new TASK(RIGHT_DATA);
        return COMBINE(t2.compute(), t1.join());
    }
}
```

Asynchronously execute

Process in current thread

Block until done

compute Example (Below Threshold)

```
protected Integer compute() {
    if (end - start < threshold) {
        int max = Integer.MIN_VALUE;
        for (int i = start; i <= end; i++) {
            int n = myArray[i];
            if (n > max) {
                max = n;
            }
        }
        return max;
    } else {
        // split data and create tasks
    }
}
```

You decide the threshold.

The range within the array

`compute` Example (Above Threshold)

```
protected Integer compute() {
    if (end - start < threshold) {
        // find max
    } else {
        int midway = (end - start) / 2 + start;
        FindMaxTask a1 =
    new FindMaxTask(myArray, start, midway, threshold);
        a1.fork();
        FindMaxTask a2 =
    new FindMaxTask(myArray, midway + 1, end, threshold);
        return Math.max(a2.compute(), a1.join());
    }
}
```

Task for left half of data

Task for right half of data

`ForkJoinPool` Example

A `ForkJoinPool` is used to execute a `ForkJoinTask`. It creates a thread for each CPU in the system by default.

```
ForkJoinPool pool = new ForkJoinPool();
FindMaxTask task =
 new FindMaxTask(data, 0, data.length-1, data.length/16);
Integer result = pool.invoke(task);
```

The task's `compute` method is automatically called .

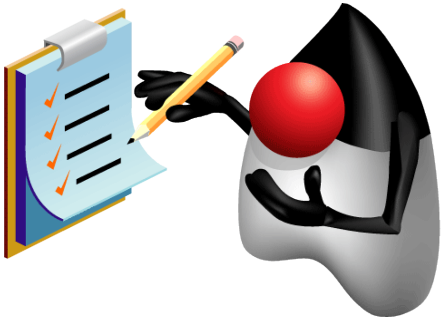
Fork-Join Framework Recommendations

- Avoid I/O or blocking operations.
  - Only one thread per CPU is created by default. Blocking operations would keep you from utilizing all CPU resources.
- Know your hardware.
  - A Fork-Join solution will perform slower on a one-CPU system than a standard sequential solution.
  - Some CPUs increase in speed when only using a single core, potentially offsetting any performance gain provided by Fork-Join.
- Know your problem.
  - Many problems have additional overhead if executed in parallel (parallel sorting, for example).

Summary

In this lesson, you should have learned how to:

- Apply the Fork-Join framework



Practice 16-1 Overview: Using the Fork-Join Framework

This practice covers the following topics:

- Extending `RecursiveAction`
- Creating and using a `ForkJoinPool`

Quiz

Applying the Fork-Join framework will always result in a performance benefit.

a. True

b. False