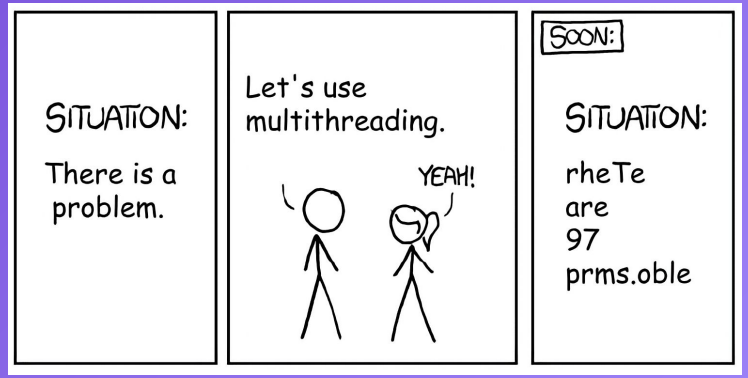
Parallel Programming

CSE 332 – Section 7

Slides by James Richie Sulaeman



Parallel

Suppose we wanted to take the sum of all the elements in the array `arr`

```
// pseudocode
function parallelSum(arr, start, end):
    if (end - start) < threshold:
        return sumSequential(arr, start, end)
    else:
        mid = (start + end) / 2
        left = parallelSum(arr, start, mid)
        right = parallelSum(arr, mid, end)
        return left + right
```

Library

Conceptual Action	Java's Fork/Join Library	Explanation
Define a unit of recursive work	`RecursiveTask<T>`	Abstract class used to define a task that returns a result.
Implement the recursive logic	`compute()`	Method where the task defines its base case and recursive splitting.
Start a subtask in parallel	`fork()`	Asynchronously execute a subtask.
Wait for subtask result	`join()`	Block and wait for the result of a previously forked task.
Run the top-level task	`ForkJoinPool.invoke(task)`	Starts the task and returns its final result.

ForkJoin

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;
  SumTask(int[] arr, int low, int hi) { … }

  protected Integer compute(){
    // Base case: sequential



    // Recursive









  }


}
```

Conceptual Action	Java's Fork/Join Library
Define a unit of recursive work	`RecursiveTask<T>`
Implement the recursive logic	`compute()`
Start a subtask in parallel	`fork()`
Wait for subtask result	`join()`
Run the top-level task	`ForkJoinPool.invoke(task)`

ForkJoin

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;
  SumTask(int[] arr, int low, int hi) { … }

  protected Integer compute(){
    if (hi - lo <= CUTOFF) {
      return sequential(arr, lo, hi);
    }

    // Recursive









  }

  protect Integer sequential(...){...}
}
```

Conceptual Action	Java's Fork/Join Library
Define a unit of recursive work	`RecursiveTask<T>`
Implement the recursive logic	`compute()`
Start a subtask in parallel	`fork()`
Wait for subtask result	`join()`
Run the top-level task	`ForkJoinPool.invoke(task)`

ForkJoin

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;
  SumTask(int[] arr, int low, int hi) { … }

  protected Integer compute(){
    if (hi - lo <= CUTOFF) {
      return sequential(arr, lo, hi);
    }

    // Recursive
    // Divide into 2 subtasks


    // start a parallel subtask
    // implement recursive logic
    // wait for subtask

    // return result
  }

  protect Integer sequential(...){...}
}
```

Conceptual Action	Java's Fork/Join Library
Define a unit of recursive work	`RecursiveTask<T>`
Implement the recursive logic	`compute()`
Start a subtask in parallel	`fork()`
Wait for subtask result	`join()`
Run the top-level task	`ForkJoinPool.invoke(task)`

ForkJoin

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;
  SumTask(int[] arr, int low, int hi) { … }

  protected Integer compute(){
    if (hi - lo <= CUTOFF) {
      return sequential(arr, lo, hi);
    }

    int mid = lo + (hi - lo) / 2;
    SumTask left = new SumTask(arr, lo, mid);
    SumTask right = new SumTask(arr, mid, hi);

    // start a parallel subtask
    // implement recursive logic
    // wait for subtask

    // return result
  }

  protect Integer sequential(...){...}
}
```

Conceptual Action	Java's Fork/Join Library
Define a unit of recursive work	`RecursiveTask<T>`
Implement the recursive logic	`compute()`
Start a subtask in parallel	`fork()`
Wait for subtask result	`join()`
Run the top-level task	`ForkJoinPool.invoke(task)`

ForkJoin

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;
  SumTask(int[] arr, int low, int hi) { … }

  protected Integer compute(){
    if (hi - lo <= CUTOFF) {
      return sequential(arr, lo, hi);
    }

    int mid = lo + (hi - lo) / 2;
    SumTask left = new SumTask(arr, lo, mid);
    SumTask right = new SumTask(arr, mid, hi);

    left.fork();
    int rightSum = right.compute();
    int leftSum = left.join();

    // return result
  }

  protect Integer sequential(...){...}
}
```

Conceptual Action	Java's Fork/Join Library
Define a unit of recursive work	`RecursiveTask<T>`
Implement the recursive logic	`compute()`
Start a subtask in parallel	`fork()`
Wait for subtask result	`join()`
Run the top-level task	`ForkJoinPool.invoke(task)`

ForkJoin

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;
  SumTask(int[] arr, int low, int hi) { … }

  protected Integer compute(){
    if (hi - lo <= CUTOFF) {
      return sequential(arr, lo, hi);
    }

    int mid = lo + (hi - lo) / 2;
    SumTask left = new SumTask(arr, lo, mid);
    SumTask right = new SumTask(arr, mid, hi);

    left.fork();
    int rightSum = right.compute();
    int leftSum = left.join();

    return rightSum + leftSum;
  }

  protect Integer sequential(...){...}
}
```

Conceptual Action	Java's Fork/Join Library
Define a unit of recursive work	`RecursiveTask<T>`
Implement the recursive logic	`compute()`
Start a subtask in parallel	`fork()`
Wait for subtask result	`join()`
Run the top-level task	`ForkJoinPool.invoke(task)`

ForkJoin

Invoke the recursive task

```
static final ForkJoinPool POOL = new ForkJoinPool();
static int parallelSum(int[] arr){
  SumTask task = new SumTask(arr,0,arr.length)
  return POOL.invoke(task);
}
```

Conceptual Action	Java's Fork/Join Library
Define a unit of recursive work	`RecursiveTask<T>`
Implement the recursive logic	`compute()`
Start a subtask in parallel	`fork()`
Wait for subtask result	`join()`
Run the top-level task	`ForkJoinPool.invoke(task)`

ForkJoin

Final version of compute

```
protected Integer compute() {
    // base case
    if (hi - lo <= CUTOFF) {
        return sequential(arr, lo, hi);          // sequentially handles arr[lo, ..., hi-1]
    }
    // recursive case
    int mid = lo + (hi - lo) / 2;
    SumTask left = new SumTask(arr, lo, mid);    // handles [lo, mid)
    SumTask right = new SumTask(arr, mid, hi);   // handles [mid, hi)

    left.fork();                        // fork a Thread (i.e. call compute() on a new Thread)
    int rightSum = right.compute();     // compute the right task using current Thread
    int leftSum = left.join();          // possibly wait and get result from left

    return rightSum + leftSum;          // combine outputs from left task and right task
}
```

ForkJoin

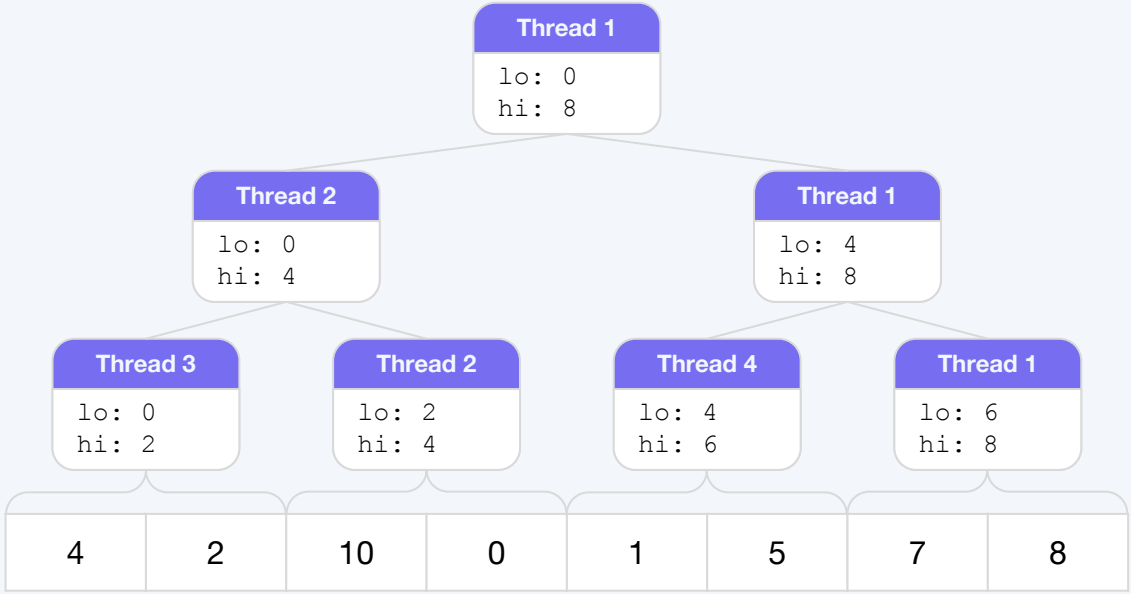
Suppose we wanted to take the sum of all the elements in the array `arr`

```
protected Integer compute() {
    // base case
    if (hi - lo <= CUTOFF) {
        return sequential(arr, lo, hi);
    }
    // recursive case
    int mid = lo + (hi - lo) / 2;
    SumTask left = new SumTask(arr, lo, mid);
    SumTask right = new SumTask(arr, mid, hi);

    left.fork();
    int rightSum = right.compute();
    int leftSum = left.join();

    return rightSum + leftSum;
}
```

CUTOFF = 2



Your Turn!

Download the code. The link is available on the schedule on the course web page.

Work in the following order:

1. LessThan7
2. CountStrs
3. Parity
4. PowMod
5. SecondSmallest

Thank You!