


Lecture 23: Divide and Conquer Parallelism

Slides adapted from Dan Grossman

+ Today



- P&C Section 3
- Objectives
  - Threads in Java
  - Summing an array of integers using threads
    - Work up to a correct implementation!
- Announcements
  - This week's assignment is cancelled
  - Turn in any (running) code by Friday for extra credit

## + Cooking Analogy

- Sequential programming
  - One cook performing each step of a recipe
  - Each step finished before next one started
- Parallelism:
  - Extra cooks (or equipment) to finish faster
  - At some point, extra hands don't help anymore
- Concurrency:
  - Lots of cooks but only one oven!
  - Coordinate access to oven so no burning or crowding



## + Recap

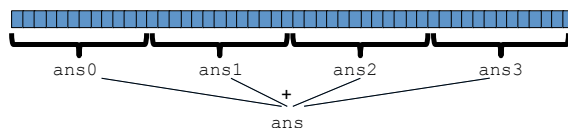
- Sequential programming
  - Limitations of this approach
- Parallelism versus concurrency
- Explicit threads with shared memory
  - Alternative models in text
- Thread is a single unit of execution
  - Separate call stacks, program counter, local variable
  - Shared static fields and objects

## + Creating threads in Java

1. Define a class C extending `java.lang.Thread` and override the `public void run()` method
2. Create an object of class C (i.e. using `new` keyword)
3. Call the `start` method of the new object
  - `start` creates a new thread and calls its `run` method
  - Directly calling `run` doesn't create a new thread
    - Just a normal method call in the current thread

## + Running Illustrative Example

- Sum an array of integers
- First attempt
  - Use 4 threads to sum 1/4 of the array in parallel
  - Add together the result from the 4 threads for final answer



## + First Attempt

```

class SumThread extends Thread {
    int lo, int hi, int[] arr; // fields to know what to do
    int ans = 0; // for communicating result
    SumThread(int[] a, int l, int h) { ... }
    public void run() { ... }
}

int sum(int[] arr) {
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) { // do parallel computations
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
        ts[i].start(); // use start not run
    }
    for(int i=0; i < 4; i++) // combine results
        ans += ts[i].ans;
    return ans;
}

```

## + Second Attempt

```

class SumThread extends Thread {
    int lo, int hi, int[] arr; // fields to know what to do
    int ans = 0; // for communicating result
    SumThread(int[] a, int l, int h) { ... }
    public void run() { ... }
}

int sum(int[] arr) {
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) { // do parallel computations
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
        ts[i].start(); // start not run
    }
    for(int i=0; i < 4; i++) // combine results
        ts[i].join(); // wait for helper to finish!
    ans += ts[i].ans;
    return ans;
}

```

## + Thread Class Methods

- `void start( )`
  - calls `void run( )`
- `void join( )`
  - blocks until receiver thread done
  - can throw an `InterruptedException`
- Style called fork-join parallelism
- Some memory sharing: `lo`, `hi`, `arr`, `ans` fields
- Later learn how to protect using `synchronized`.

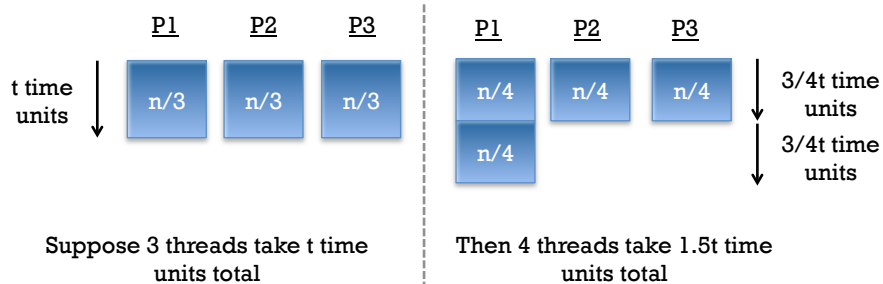
## + Problems with Second Attempt

1. The number of threads is hard coded (magic number)
  - Don't use numbers where a variable is appropriate
  - Pass in the number of threads to use as parameter to `sum`

## + Problems with Second Attempt

2. Want to use (only) processors available to us now

- Can change while your thread is running
- Fewer threads than processors leads to under-utilized resources
- More threads than processors leads to slow down



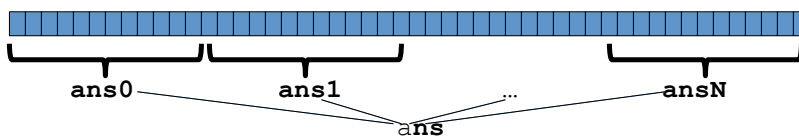
## + Problems with Second Attempt

3. Divided array equal among threads

- Equal amount of data does not imply an equal amount of work
- Referred to as load imbalance
- Load imbalance hurts efficiency since must wait till all threads done

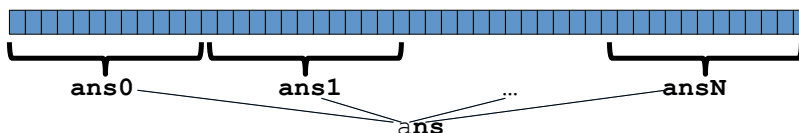
## + Third Attempt

- Use lots of threads! Far more than the number of processors
  - Each cook takes a pile of potatoes, slices them, comes back for more when done
  - Independent of the number of processors
  - Change in the number of processors translates to a change in the amount of time
  - Each thread will do a small amount of work (helps with load imbalance)



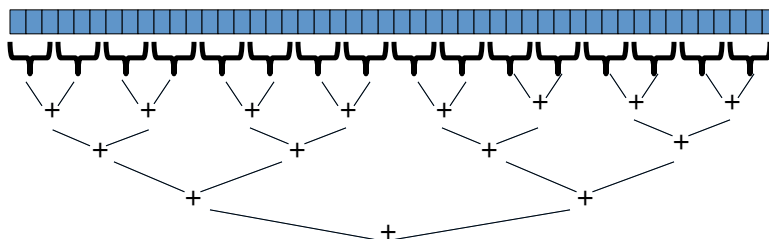
## + Problems with Third Attempt

- Use lots of threads! Far more than the number of processors
  - Lots of overhead from creating so many threads
  - In the extreme, one thread per entry in the array requires  $O(N)$  work to combine final answers!



## + Final Attempt

- Divide-and-conquer Parallelism!
  - Change our algorithm
  - Use parallelism for the recursive calls



## + Divide and Conquer Parallelism

```

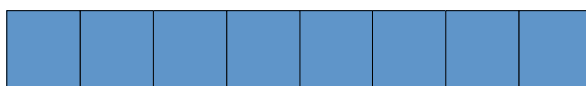
class SumThread extends java.lang.Thread {
    int lo; int hi; int[] arr; // arguments
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run(){ // override
        if(hi - lo < SEQUENTIAL_CUTOFF)
            for(int i=lo; i < hi; i++)
                ans += arr[i];
        else {
            SumThread left = new SumThread(arr,lo,(hi+lo)/2);
            SumThread right= new SumThread(arr,(hi+lo)/2,hi);
            left.start();
            right.start();
            left.join(); // don't move this up a line - why?
            right.join();
            ans = left.ans + right.ans;
        }
    }
}

```



## + Divide and Conquer Parallelism

- Divide array in half with one thread per half
  - Create  $\sim 2N$  threads, each thread doing  $O(1)$  work
  - If we have  $O(N)$  processors, run time is  $O(\log N)$ !
  - Why? Because each level is done in parallel



Carry out divide-and-conquer on small array

## + Divide-and-conquer Parallelism

- Final improvements
  - Choose cutoff value: below cutoff switch to sequential programming
  - Don't create two threads: create one thread and have the calling thread do the other half of the work
- Use ForkJoin Framework for lightweight threads
  - Look at this after Spring Break!