



Parallel Programming 2

Johan Östlund

In part based on "Sophomoric Parallelism and Concurrency" by Dan Grossman

<http://www.cs.washington.edu/homes/djg/teachingMaterials/spac/>

Recap (concurrency)

- Concurrency is managing shared resources
- Accesses to shared resources by concurrent threads need to be synchronized
- We have seen locks and the built-in Java `synchronized`-statement
- A program that is not correctly synchronized is said to have races, and a program with races is broken
- Reordering and the memory hierarchy make the semantics of incorrectly synchronized programs surprising (at best)

Recap (parallelism)

- Clock rates cannot increase the way they have been the last 40 years

Power consumption/heat generation go through the roof

- The number of transistors we can fit on a chip still increases, however
- Chip makers use the space to put multiple cores on one chip, each running at a lower frequency
- Unless we write parallel programs that utilize the multiple cores, they will run slower than a few years ago

Parallelism for efficiency

- Parallelism could be defined as: using extra resources to solve a problem faster
- Or, with today's situation: using the available resources
- Already today we have at least two cores in any computer, so we need to make sure to utilize them both. This will become even more important in the coming years
- If two or more threads need access to shared resources concurrency needs managing

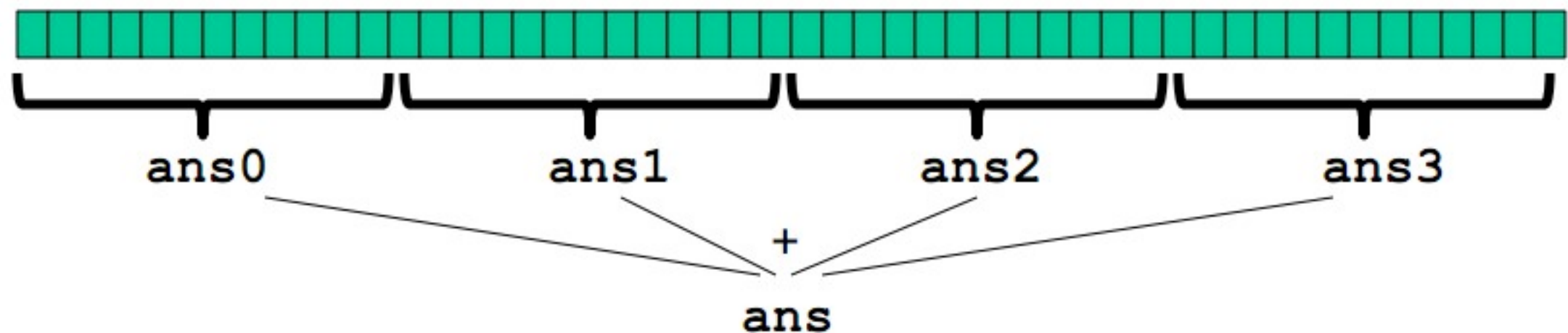
Running example: summing

- This is the sequential program that we'll aim to improve by parallelizing it

```
int sum(int[] arr) {  
    int ans = 0;  
    for (int i = 0; i < arr.length; ++i) {  
        ans += arr[i];  
    }  
    return ans;  
}
```

Basic parallel programming in Java

- In Java there is a class `java.lang.Thread`, which may be subclassed to specify the things we'd like to happen simultaneously
- Example: summing a large array of numbers



The basic idea is to divide the work into four equal parts and have four threads do the summing

First approach

```
class SumThread extends java.lang.Thread {  
    int lo; // arguments  
    int hi;  
    int[] arr;  
    int ans = 0; // result  
  
    SumThread(int[] a, int l, int h) {  
        lo = l; hi = h; arr = a;  
    }  
  
    // run() must have this signature  
    public void run() {  
        for (int i = lo; i < hi; i++)  
            ans += arr[i];  
    }  
}
```

```
int sum(int[] arr) {  
    int len = arr.length;  
    int ans = 0;  
    SumThread[] ts = new SumThread[4];  
    // do parallel computations  
    for (int i = 0; i < 4; i++){  
        ts[i] = new SumThread(arr, i*len/4,  
                                (i+1)*len/4);  
        ts[i].start();  
    }  
  
    // combine results  
    for (int i=0; i < 4; i++) {  
        // wait for helper to finish!  
        ts[i].join();  
        ans += ts[i].ans;  
    }  
    return ans;  
}
```

Waiting for threads to finish

- The `join()` method blocks the current thread until the joined thread has exited
- What would happen if we did not call `join` on the sum threads?

```
int sum(int[] arr) {  
    int len = arr.length;  
    int ans = 0;  
    SumThread[] ts = new SumThread[4];  
    // do parallel computations  
    for (int i = 0; i < 4; i++){  
        ts[i] = new SumThread(arr, i*len/4,  
                               (i+1)*len/4);  
        ts[i].start();  
    }  
  
    // combine results  
    for (int i=0; i < 4; i++) {  
        // wait for helper to finish!  
        ts[i].join();  
        ans += ts[i].ans;  
    }  
    return ans;  
}
```


Waiting for threads to finish

- The `join()` method blocks the current thread until the joined thread has exited
- What would happen if we did not call `join` on the sum threads?

```
int sum(int[] arr) {  
    int len = arr.length;  
    int ans = 0;  
    SumThread[] ts = new SumThread[4];  
    // do parallel computations  
    for (int i = 0; i < 4; i++){  
        ts[i] = new SumThread(arr, i*len/4,  
                               (i+1)*len/4);  
  
        ts[i].start();  
    }  
  
    // combine results  
    for (int i=0; i < 4; i++) {  
        // wait for helper to finish!  
        // ts[i].join();  
        ans += ts[i].ans;  
    }  
    return ans;  
}
```

Likely the wrong value

Race condition

Race condition

- A race condition is usually defined as "two threads, both accessing the same location, and at least one is a write."
- Or, when the result of a computation depends on scheduling
- Race conditions are often hard to debug

They depend on timing, they may happen sometimes and other times not

The debugger changes timing, so perhaps it never happens there

Values may seem to randomly change

- Sometimes race conditions don't show on single (or few) core CPUs but surface when run on many cores
- A race condition is **always** an error! (Unless your name is Doug Lea)

Improving our approach

- The least we can do is parameterize the method over number of threads (we'll have more cores in the future, remember?)

```
int sum(int[] arr, int numThreads) {  
    ... // note: shows idea, but has integer-division bug  
    int subLen = arr.length / numThreads;  
    SumThread[] ts = new SumThread[numThreads];  
    for (int i = 0; i < numThreads; i++) {  
        ts[i] = new SumThread(arr, i*subLen, (i+1)*subLen);  
        ts[i].start();  
    }  
    for (int i = 0; i < numThreads; i++) {  
        ...  
    }  
    ...  
}
```

Improving our approach

- When writing the program we don't know the number of cores we'll have at runtime
- And even if we know which machine will run the program there may be other tasks running in parallel using cores
- We could check the number of available cores dynamically by calling `Runtime.availableProcessors()`, but we still don't know if the cores are being used
- So, we need a better solution

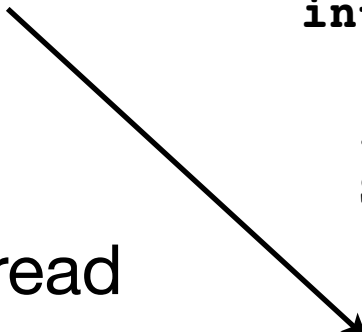
Load imbalance

- In general (not summing), sub tasks may vary greatly in computation time
Ex: an `isPrime()` function will take much longer for a large number
- If we just divide the work into some given number of parts, perhaps all difficult cases end up on one thread. This will kill the benefits of parallelizing

A better approach (still poor)

- Counterintuitive as it may seem, the solution is to create lots of threads, far more than the number of available processors, but
- let's say we create one thread to process every 1000 elements
- combining the results will still be linear in the size of the array

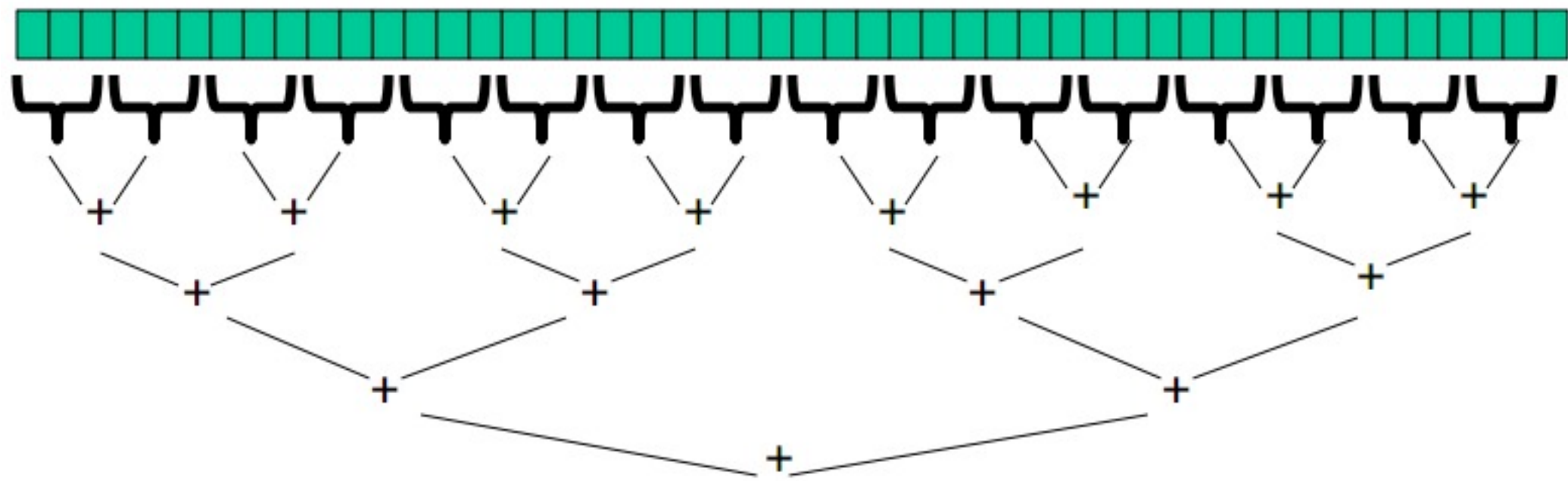
```
int sum(int[] arr) {  
    ...  
    int numThreads = arr.length / 1000;  
    SumThread[] ts = new SumThread[numThreads];  
    ...  
    for (int i=0; i < numThreads; i++) {  
        ...  
        ans += ts[i].ans;  
    }  
}
```



- In fact, if we create one thread for each element, we recreate a sequential algorithm (only it uses tons of memory)

A better idea - divide and conquer

- Implement as a recursive algorithm, where the recursive calls fork off parallel tasks
- For summing (and many other problems) this is straightforward



Divide and conquer

```
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(){
        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;
        }
    }
}

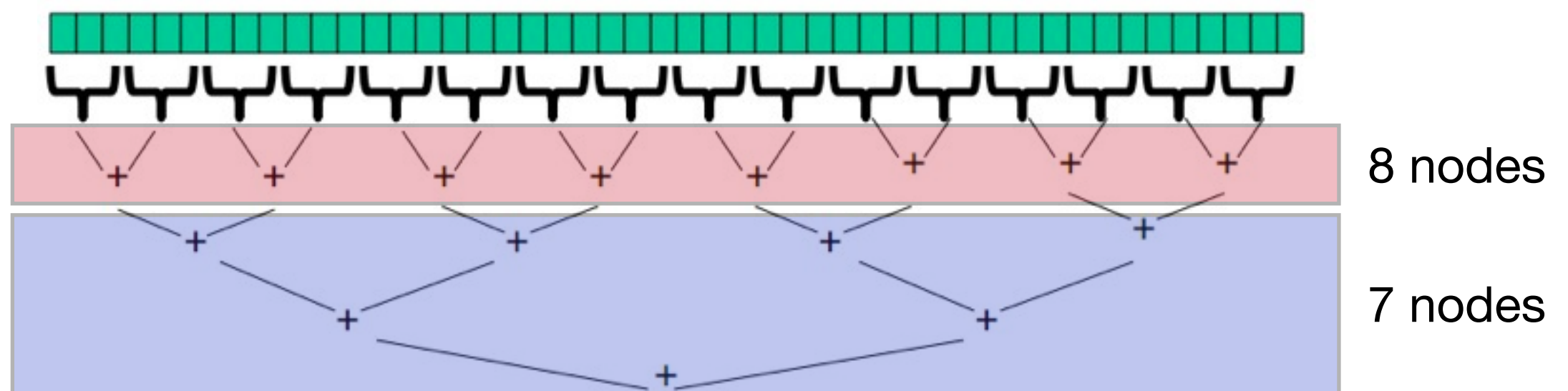
int sum(int[] arr){
    SumThread t = new SumThread(arr, 0, arr.length);
    t.run();
    return t.ans;
}
```


Divide and conquer

- The key is that this approach parallelizes the result combining
- If you have enough processors total time is the depth of the tree, which is $O(\log n)$. That's exponentially faster than the sequential $O(n)$
- However, this approach usually requires the result combination operation to be associative (i.e., you can combine the results in any order, like +)

Resorting to sequential computation

- In theory divide and conquer works all the way down to individual elements
- But in practice object creation, book-keeping, scheduling and thread communication swamps the savings
- That's why you usually have a point where you resort to sequential computation
- It eliminates almost all the recursive thread creation (approaching the bottom of the tree).



Half the threads

- In the previous implementation the current thread was just waiting for its two child threads to finish
- It could just as well perform one of the child tasks itself
- This will start half as many threads

```
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.run();
            left.join(); // don't move this up a line - why?
            ans = left.ans + right.ans;
        }
    }
}
```

A note on threads

- Threads in the JVM are operating system threads, and they are expensive, really expensive
- So expensive, in fact that this approach is not very feasible (for large data)
- You simply can't create 1 000 000 threads on your laptop, it'll choke
- We need something more light-weight

A fork/join framework

- Fortunately there is a framework for us to use
- The JSR166 framework is included in Java 7, but also available as a jar-file if you're using Java 6
- It is a part of the `java.util.concurrent` package written by Doug Lea

Same but different

- Don't subclass Thread, subclass RecursiveTask<V> (or RecursiveAction)
 - Don't override run(), override compute()
 - Don't use an "ans" field, compute() returns the result
 - Don't call start(), call fork() (or invokeAll(...))
 - Don't just call join(), join() now returns the result
-
- Google for "A Beginner's Introduction to the ForkJoin Framework" (that's also where some of the content for these slides are from.)

Using the F/J framework

```
class SumArray extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr; //fields to know what to do

    SumArray(int[] a, int l, int h) { ... }

    protected Integer compute() {
        if (hi - lo < SEQUENTIAL_CUTOFF) {
            int ans = 0;
            for (int i=lo; i < hi; i++)
                ans += arr[i];
            return ans;
        } else {
            SumArray left = new SumArray(arr, lo, (hi+lo)/2);
            SumArray right= new SumArray(arr, (hi+lo)/2, hi);
            right.fork();
            return left.compute() + right.join();
        }
    }
}

static final ForkJoinPool fjPool = new ForkJoinPool();
int sum(int[] arr) {
    return fjPool.invoke(new SumArray(arr, 0, arr.length));
}
```

What else can we do?

- By using divide and conquer we have seen summing go from $O(n)$ to $O(\log n)$ (that's for very large n and lots of cores..)
- Anything that can divide its work into two parts and combine the results in $O(1)$ has the same property
- Examples:

Maximum or minimum element

Is there an element with some property (e.g., is there a prime number)

Counting something

etc.

Reductions

- These kinds of computations are called reductions (or reduces)
- They "reduce to some value"
- Note: the result doesn't have to be a single scalar value, it could be a collection

Maps

- An even easier computation is a map
- A map doesn't need to combine results, it just applies some function to all elements (and usually the result is a new collection of the same size)
- Sometimes you update the current collection, and then you don't even return a new one (your lab is like this)
- Example: adding two vectors

Map example: adding two vectors

```
class VecAdd extends RecursiveAction {  
    int lo; int hi; int[] res; int[] arr1; int[] arr2;  
    VecAdd(int l, int h, int[] r, int[] a1, int[] a2){ ... }
```

```
    protected void compute(){  
        if (hi - lo < SEQUENTIAL_CUTOFF) {  
            for (int i = lo; i < hi; i++)  
                res[i] = arr1[i] + arr2[i];  
        } else {  
            int mid = (hi+lo)/2;  
            VecAdd left = new VecAdd(lo, mid, res, arr1, arr2);  
            VecAdd right = new VecAdd(mid, hi, res, arr1, arr2);  
            left.fork();  
            right.compute();  
            left.join();  
        }  
    }  
}
```

We extend RecursiveAction

No value is returned

```
static final ForkJoinPool fjPool = new ForkJoinPool();
```

```
int[] add(int[] arr1, int[] arr2){  
    assert (arr1.length == arr2.length);  
    int[] ans = new int[arr1.length];  
    fjPool.invoke(new VecAdd(0, arr.length, ans, arr1, arr2));  
    return ans;  
}
```

The result instead is
put in this array

Side note: "map/reduce"

- You may have heard about Google's map/reduce (or the open source Hadoop)
- It's the same idea as here, only it uses clusters of machines on a network

The essence of fork/join (map/reduce)

- You need to formulate your problem in a way that allows splitting and combining
- Figure out how to divide data into two parts
- Figure out a way to combine results
- Perhaps you need to "prepare" your data in a way that allows splitting and combining. Does it pay off?

Summary

- We have talked about parallelism as a way to utilize the available resources
- Fork/join parallelizes the result combining
- The trick with fork/join parallelism is dividing data and combining results
- With fork/join one rarely needs to focus on synchronization, it happens when we `join()` child tasks
- We have multiple cores, it would be wasteful not to use them