



Parallel Programming

Johan Östlund

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

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

Sequential execution

- One thing happens at a time
- The program has a given order of execution, and so do accesses to resources (e.g., memory)

- Will the assert in this program ever fail?

```
class C {  
    private int x = 0;  
    private int y = 0;  
  
    void f() {  
        x = 1;  
        y = 1;  
    }  
  
    void g() {  
        int a = y;  
        int b = x;  
        assert(b >= a);  
    }  
}
```

Why sequential programming does not work

- Moore's Law: "The number of transistors incorporated in a chip will approximately double every 24 months." - Gordon Moore, Intel Co-Founder
- This forecast has been more or less correct for 40 years
- Chip makers used the increasing space and decreasing distance to make faster chips
- This no longer works as increased clock rates lead to increased power consumption and heat generation
- In the last decade chip makers instead have started using the increased space for multiple cores, running at lower speed
- In order to take advantage of these multiple cores programs must be written differently

What can we do with multiple cores?

- In the coming years regular computers are likely to have 4, 8 or 16 cores
- Run multiple programs at the same time

We already do that, but with time-slicing

- Do multiple things simultaneously in the same program (this will be our focus)

Requires a different mindset, other algorithms, other data structures

Concurrency vs. Parallelism

- Concurrency and parallelism are often confused
- Concurrency is "managing access to shared resources"
- Concurrent Execution is "to seemingly do several things at the same time"
- Parallelism is "doing several things at the same time" (for efficiency)
- Threads are commonly used to achieve both, which is probably why they're easily confused
- If parallel tasks need access to shared resources, then concurrency needs to be managed

Shared memory

- A sequential program has
 - one call stack (with local variables)
 - one program counter
 - one heap (where objects are stored)
- A concurrent/parallel program has
 - many call stacks (with their own local variables)
 - many program counters
 - one heap (where objects are stored)
- See a problem? We need to coordinate (order) accesses to the shared heap.

So what do we need?

- A way to run multiple things simultaneously, let's call these things threads
- Ways for threads to share data (we already have that in the shared heap)
- Ways for threads to coordinate (synchronize)

Threads in Java

Example

- In Java there is a class `java.lang.Thread`
- You can subclass this class and override the `run ()` method
- Doing this and calling `start ()` will begin execution in a new thread

Concurrent programming

- Concurrency: "Correctly and efficiently managing access to shared resources from multiple - possibly simultaneous - clients"
- Concurrent execution requires coordination, particularly synchronization to avoid incorrect simultaneous access. We need a way to block other threads while we're using a shared resource
- Example: what if two threads update a bank account at the same time?

Concurrent execution

- We use threads for concurrent execution
- If the number of threads is greater than the number of cores they will not all run in parallel. Threads will run in an interleaved manner
- Threads may still be very useful for performance

Hide latency when reading from a file

Have the GUI respond while waiting for content from a web server

Sharing (again)

- Different threads might access the same resources in an unpredictable order or even at about the same time
- Program correctness requires that simultaneous access be prevented using synchronization
- Simultaneous access is rare, and hard to provoke

Makes testing and debugging very difficult

Canonical example: The Bank

- This code is correct for a single-threaded execution
- but not for a multi-threaded execution

```
class BankAccount {  
    private int balance = 0;  
  
    int getBalance() { return balance; }  
  
    void setBalance(int x) { balance = x; }  
  
    void withdraw(int amount) {  
        int b = getBalance();  
        if (amount > b)  
            throw new WithdrawTooLargeException();  
        setBalance(b - amount);  
    }  
  
    ...  
    // other operations like deposit, etc.  
}
```

Interleaving

Example

- Suppose

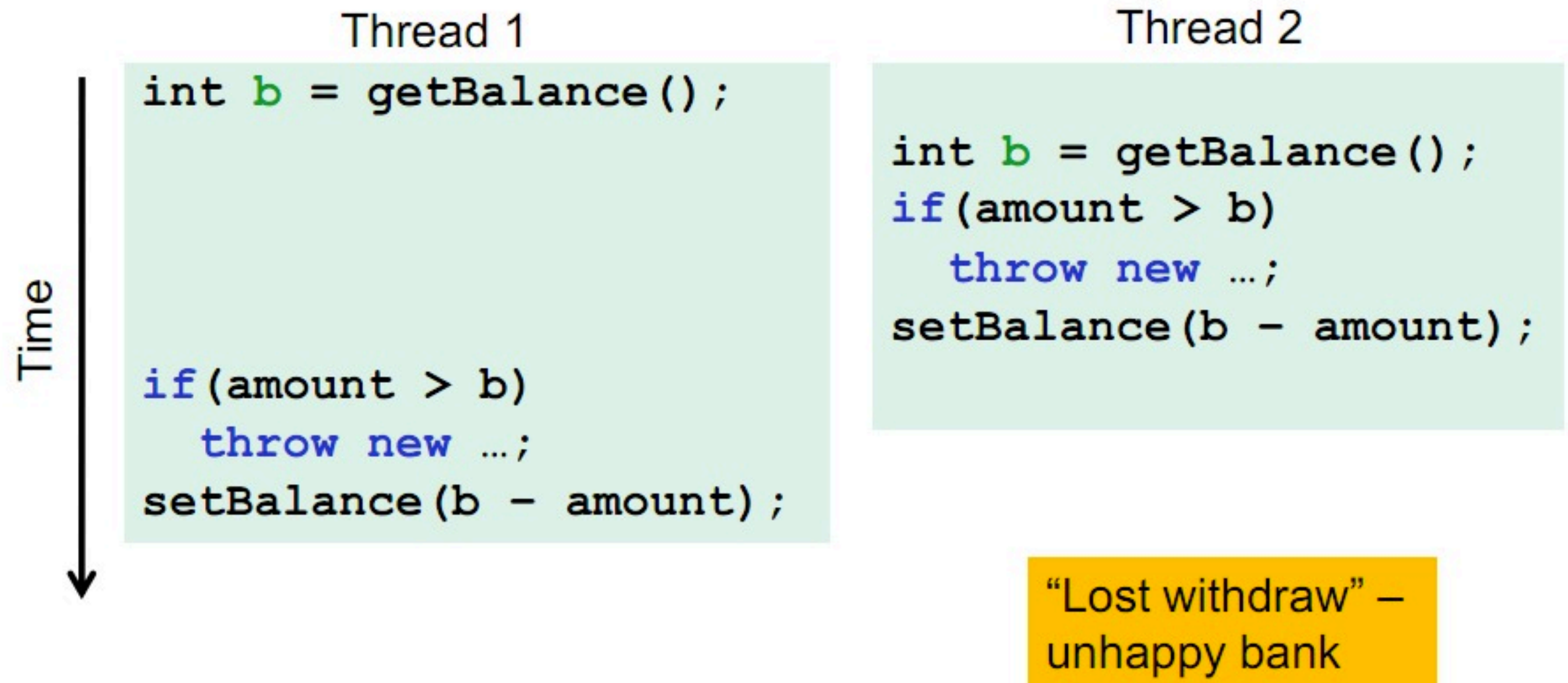
Thread 1 calls `x.withdraw(100)`

Thread 2 calls `y.withdraw(100)`

- If second call starts before first finishes, we say the calls interleave. Could happen even with one processor since a thread can be pre-empted at any point for time-slicing
- If `x` and `y` refer to different accounts, no problem, but if `x` and `y` refer to the same object, there's trouble ahead

Bad interleavings

- Interleaved `withdraw(100)` calls on the same account (Assume initial `balance == 150`)



The non-fix

- You may think that the problem is that we're saving the value on the stack and that reading it again will solve the problem?
- Well, it doesn't
- The balance may still change between the if-statement and the call to `setBalance()`
- This is just moving the problem a little, not solving it

```
class BankAccount {  
    private int balance = 0;  
  
    int getBalance() { return balance; }  
  
    void setBalance(int x) { balance = x; }  
  
    void withdraw(int amount) {  
        if (amount > getBalance())  
            throw new WithdrawTooLargeException();  
        // maybe balance changed  
        setBalance(getBalance() - amount);  
    }  
  
    ...  
    // other operations like deposit, etc.  
}
```

Assert example revisited

- Remember this example?
- Are there any bad interleavings?
- No! Right? If $\neg (b \geq a)$, then $a == 1$ and $b == 0$. But if $a == 1$, then $a = y$ happened after $y = 1$. And since programs execute in order, $b = x$ happened after $a = y$ and $x = 1$ happened before $y = 1$. So by transitivity, $b == 1$.
- Well, unfortunately that's not the whole story

```
class C {  
    private int x = 0;  
    private int y = 0;  
  
    void f() {  
        x = 1;  
        y = 1;  
    }  
  
    void g() {  
        int a = y;  
        int b = x;  
        assert(b >= a);  
    }  
}
```


Reordering

- For performance reasons the compiler and hardware often reorder memory operations
- A reordering will never be performed that would affect the result of a single-threaded program
- A bit simplified: a reordering will never be performed that crosses a synchronization boundary, so a data-race free program will not be observably affected by reorderings
- If there are no data-races in your program, then you can forget about reorderings
- Data-races are errors! A program with a data-race is always incorrect.

Mutual exclusion

- We need to make sure that at most one thread can withdraw from an account at the same time (and similar for other account operations, e.g., deposit)
- This is called mutual exclusion: One thread using a resource (here: an account) means all other threads must wait if they want to use that same resource
- The programmer must implement critical sections, i.e., tell the compiler which interleavings are allowed
- We need locks

Locks

- A lock is (usually) an object with three basic operations

Create a new lock

Acquire a lock, "block the current thread if the lock is held, otherwise grab the lock"

Release a lock, "make the lock not held, if other threads are waiting on the lock give it to exactly one of those threads"

- Locks use special hardware and OS support to ensure that the locking is atomic

Bank example (almost correct)

- Why is this just "almost" correct?

```
class BankAccount {
    private int balance = 0;
    private Lock lk = ...;

    ...

    void withdraw(int amount) {
        lk.lock(); // may block
        int b = getBalance();
        if (amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
        lk.unlock();
    }

    // deposit would also lock/unlock lk
}
```

Bank example (correct)

- We need to release the lock on all exit paths.
- Is there a nice way to do that in Java?

```
class BankAccount {
    private int balance = 0;
    private Lock lk = ...;

    ...

    void withdraw(int amount) {
        lk.lock(); // may block
        int b = getBalance();
        if (amount > b) {
            lk.unlock();
            throw new WithdrawTooLargeException();
        }
        setBalance(b - amount);
        lk.unlock();
    }

    // deposit would also lock/unlock lk
}
```

Bank example (also correct)

- We need to release the lock on all exit paths. We can use Java's try-finally construct
- the finally-clause is always run, no matter how the try-block is exited

```
class BankAccount {
    private int balance = 0;
    private Lock lk = ...();

    ...

    void withdraw(int amount) {
        try {
            lk.lock(); // may block
            int b = getBalance();
            if (amount > b) {
                throw new WithdrawTooLargeException();
            }
            setBalance(b - amount);
        } finally {
            lk.unlock();
        }
    }

    // deposit would also lock/unlock lk
}
```

Java: `synchronized`

- In Java there is a built-in language construct called `synchronized`
- Every Java object "is itself a lock"
- The `synchronized` statement evaluates the expression and uses the resulting object as lock for the `synchronized` block
- Java automatically releases the lock when the `synchronized` block is exited, regardless of how it's exited

```
synchronized (expression) {  
    statements  
}
```

Bank with synchronized

```
class BankAccount {
    private int balance = 0;
    private Object lk = new Object();

    int getBalance() {
        synchronized (lk) { return balance; }
    }
    void setBalance(int x) {
        synchronized (lk) { balance = x; }
    }
    void withdraw(int amount) {
        synchronized (lk) {
            int b = getBalance();
            if (amount > b)
                throw ...
            setBalance(b - amount);
        }
    }
    // deposit would also use synchronized(lk)
}
```


Bank with `synchronized` (take 2)

```
class BankAccount {
    private int balance = 0;

    int getBalance() {
        synchronized (this) { return balance; }
    }
    void setBalance(int x) {
        synchronized (this) { balance = x; }
    }
    void withdraw(int amount) {
        synchronized (this) {
            int b = getBalance();
            if (amount > b)
                throw ...
            setBalance(b - amount);
        }
    }
    // deposit would also use synchronized(this)
}
```

Bank with `synchronized` (take 3)

```
class BankAccount {
    private int balance = 0;

    synchronized int getBalance() {
        return balance;
    }
    synchronized void setBalance(int x) {
        balance = x;
    }
    synchronized void withdraw(int amount) {
        int b = getBalance();
        if (amount > b)
            throw ...
        setBalance(b - amount);
    }

    // deposit would also use synchronized
}
```

The `synchronized` method modifier is a short-hand for enclosing the entire method body in a `synchronized` block with `this` as lock expression

More Java locks

- The package `java.util.concurrent` contains other types of locks (and tons of other cool stuff)
- You can use these if you need them, but then you cannot use the `synchronized` construct
- Always make sure you enclose the critical section in a try-finally

Common mistakes

- Locks are very primitive, it's up to you to implement the critical sections and make sure that locks are released correctly
- Incorrect use: if withdraw and deposit use different locks it won't work: you must use the same lock for the same resource!
- The size of a critical section has impact on performance
 - if a single lock is used to lock all access to all accounts we get no parallelism
 - if we use too many locks and too small critical sections, well then we lose atomicity (see above). Also, the risk for deadlock increases

Deadlock

- Deadlock is when two (or more) threads wait on each other
- Ex:
 - Thread 1 grabs lock A
 - Thread 2 grabs lock B
 - Thread 1 tries to grab lock B
 - Thread 2 tries to grab lock A
- These threads will wait forever and never make progress

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

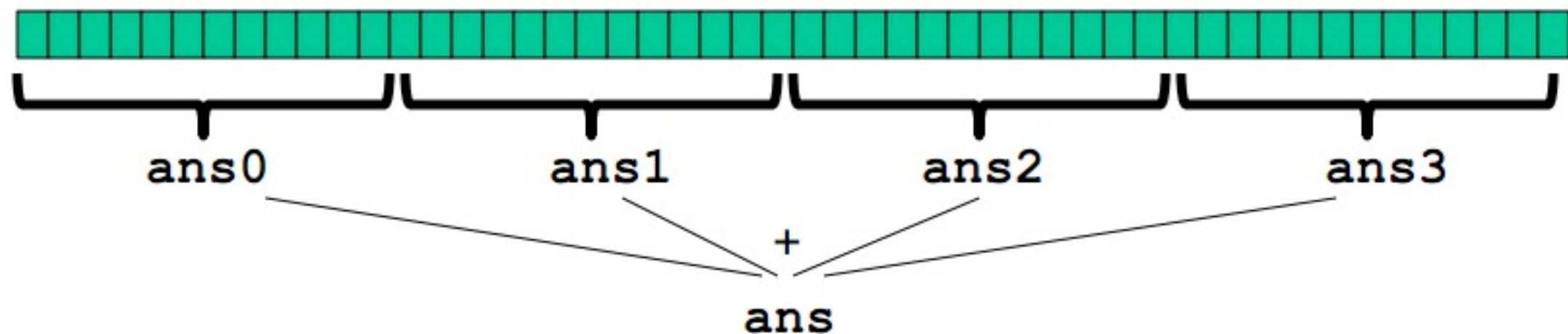
Running example: summing

- 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

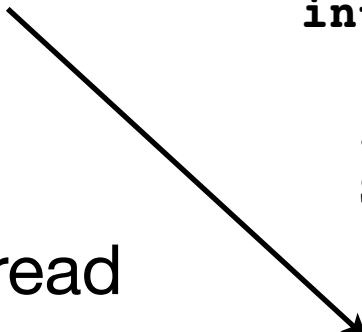
Load imbalance

- 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
- So, we need a better solution

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

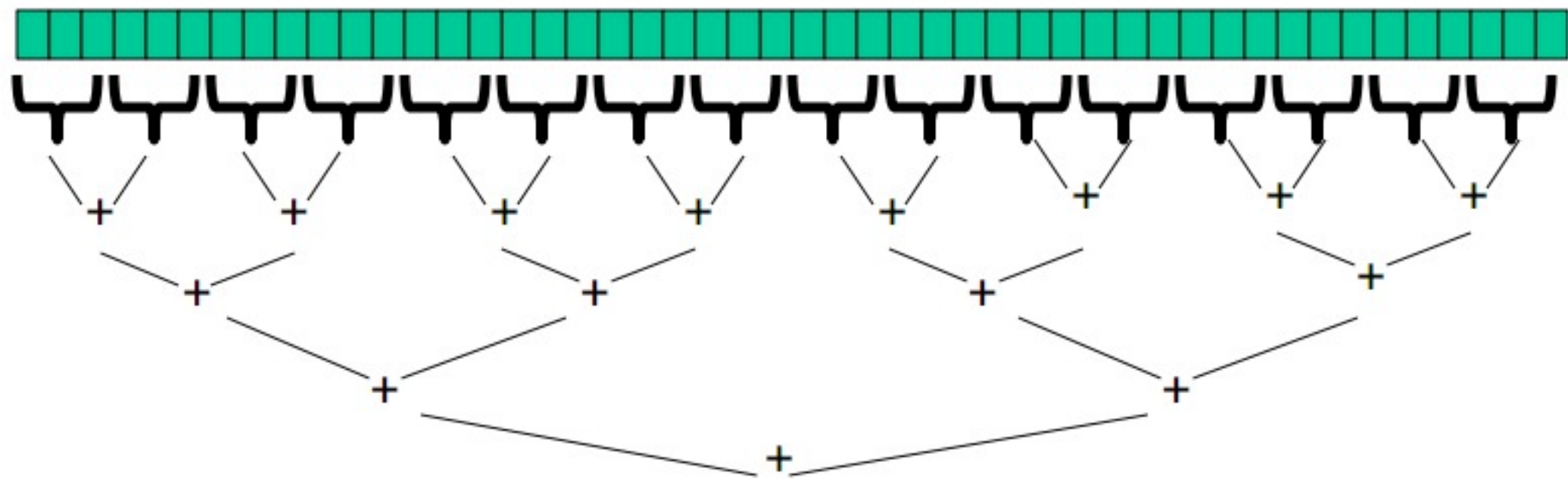
```
int sum(int[] arr) {  
    ...  
    int numThreads = arr.length / 1000;  
    SumThread[] ts = new SumThread[numThreads];  
    ...  
    for (int i=0; i < numTreads; i++) {  
        ...  
        ans += ts[i].ans;  
    }  
}
```



- 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;
        }
    }
}

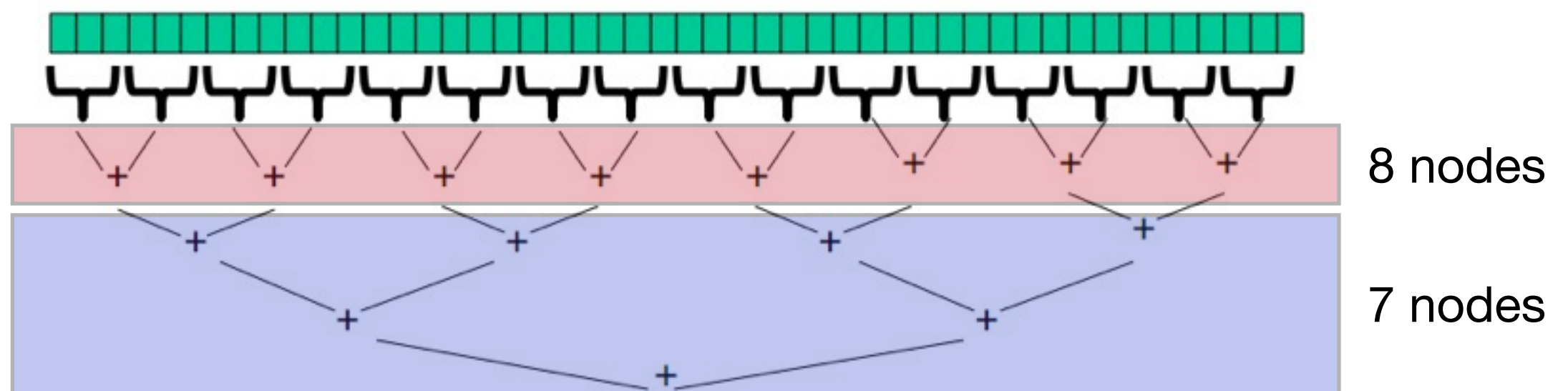
int sum(int[] arr){
    SumThread t = new SumThread(arr, 0, arr.length);
    t.run();
    return t.ans;
}
```

Divide and conquer

- The key is that this approach parallelizes the result combining
- If you have enough cores 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 greatly reduces the number of threads created

```
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 in these slides is 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 collection in place, 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 multi-threaded execution and briefly the difference between concurrency and parallelism
- In a multi-threaded program (concurrent or parallel) accesses to shared resources must be synchronized

To prevent bad interleavings (sometimes called high-level races), and data-races (low-level races)

- Data-races are errors, and a program with a data-race is broken
- We use locks (synchronization) for mutual exclusion
- Locks are primitive and difficult to get right

Summary (cont.)

- 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