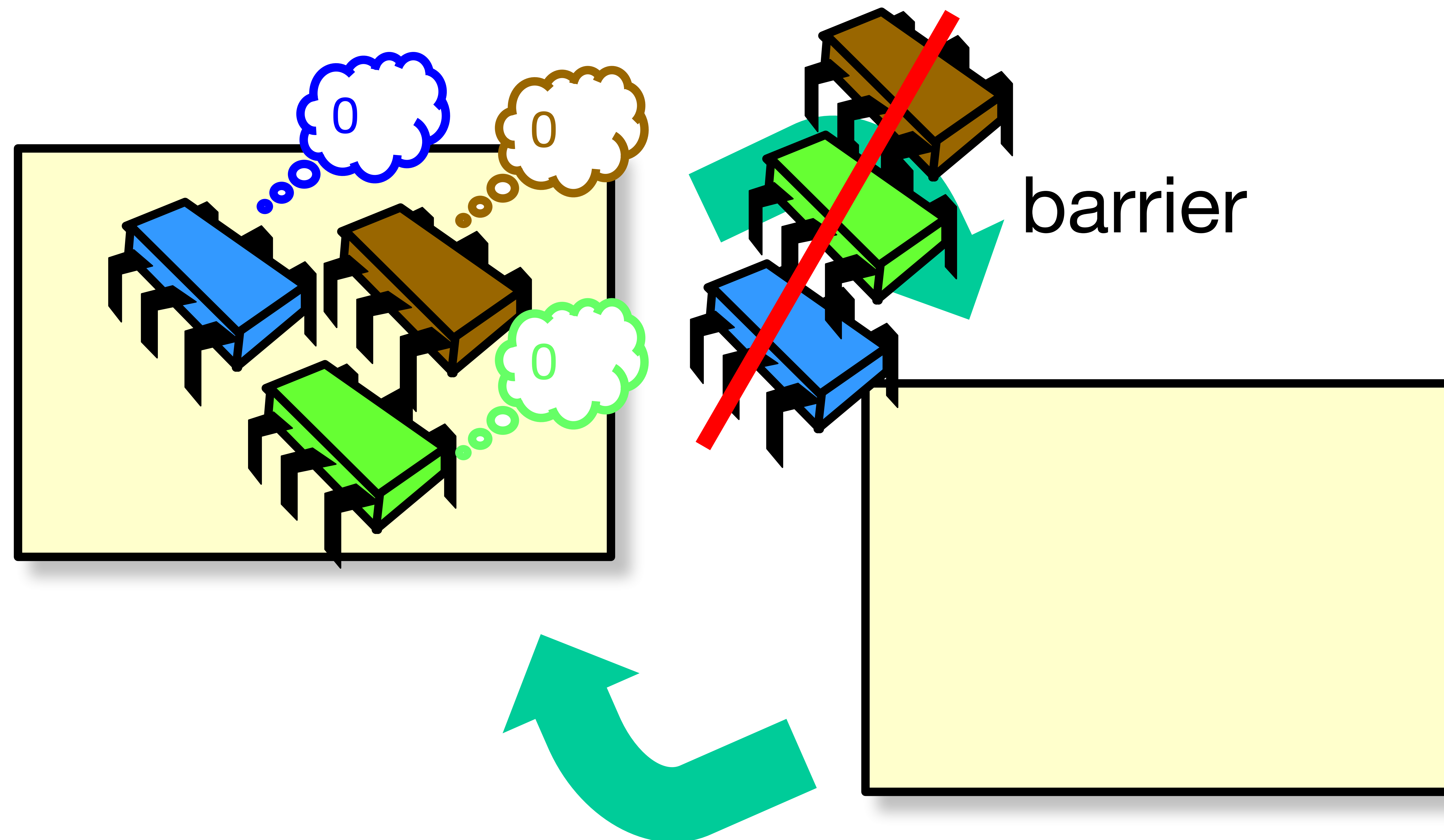
Concurrent Programming Models

CS 475, Fall 2019

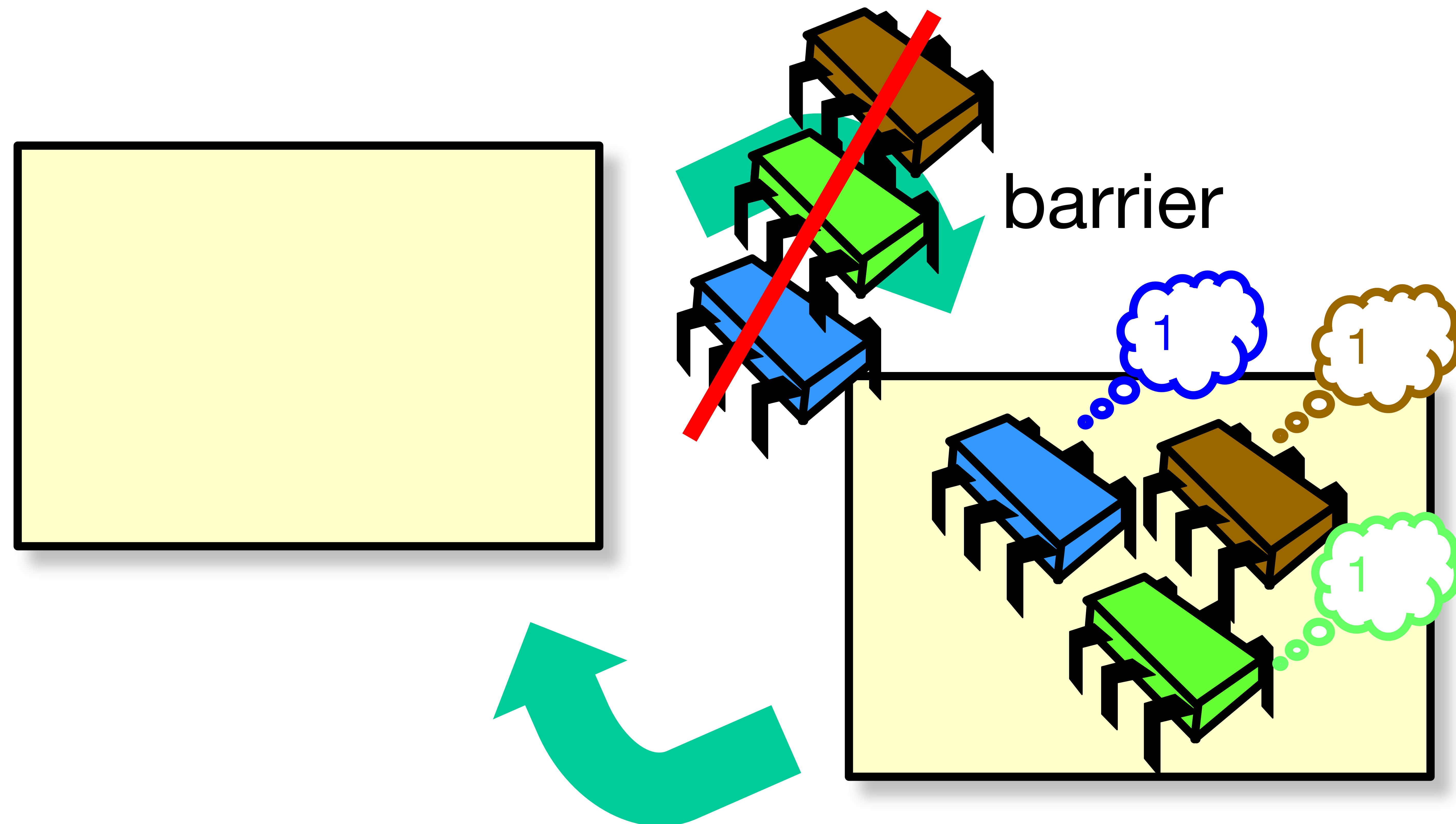
Concurrent & Distributed Systems

With material from Herlihy & Shavit, Art of Multiprocessor Programming

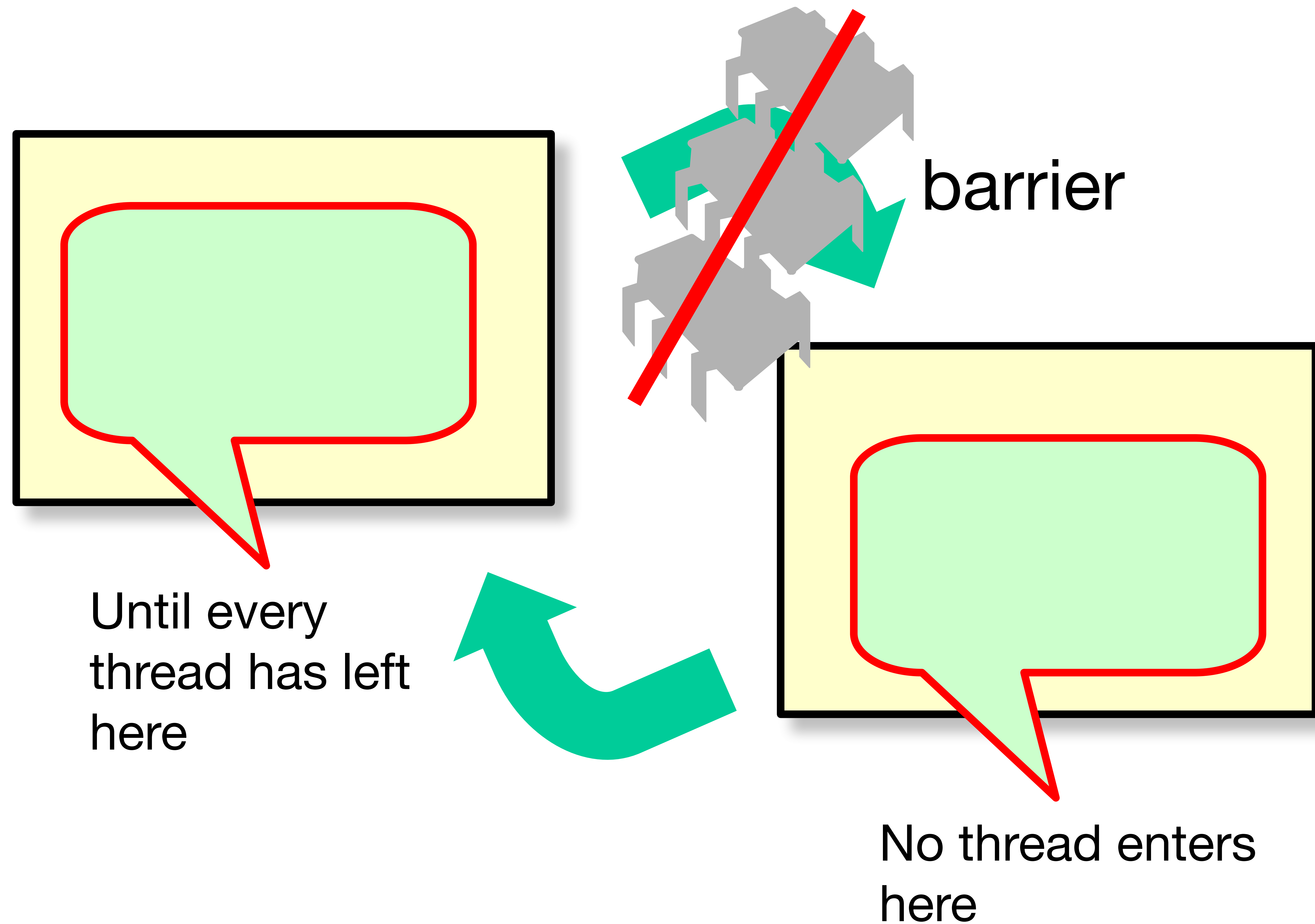
Barrier Synchronization



Barrier Synchronization



Barrier Synchronization



Combining Tree Barrier

- No sequential bottleneck
  - Parallel getAndDecrement() calls
- Low memory contention
  - Same reason
- Cache behavior
  - Local spinning on bus-based architecture
  - Not so good for non-uniform memory access architectures (NUMA) - common for large multiprocessor systems

So…How will we make use of multicores?

Back to Amdahl's Law:

Speedup = 1/(ParallelPart/N + SequentialPart)

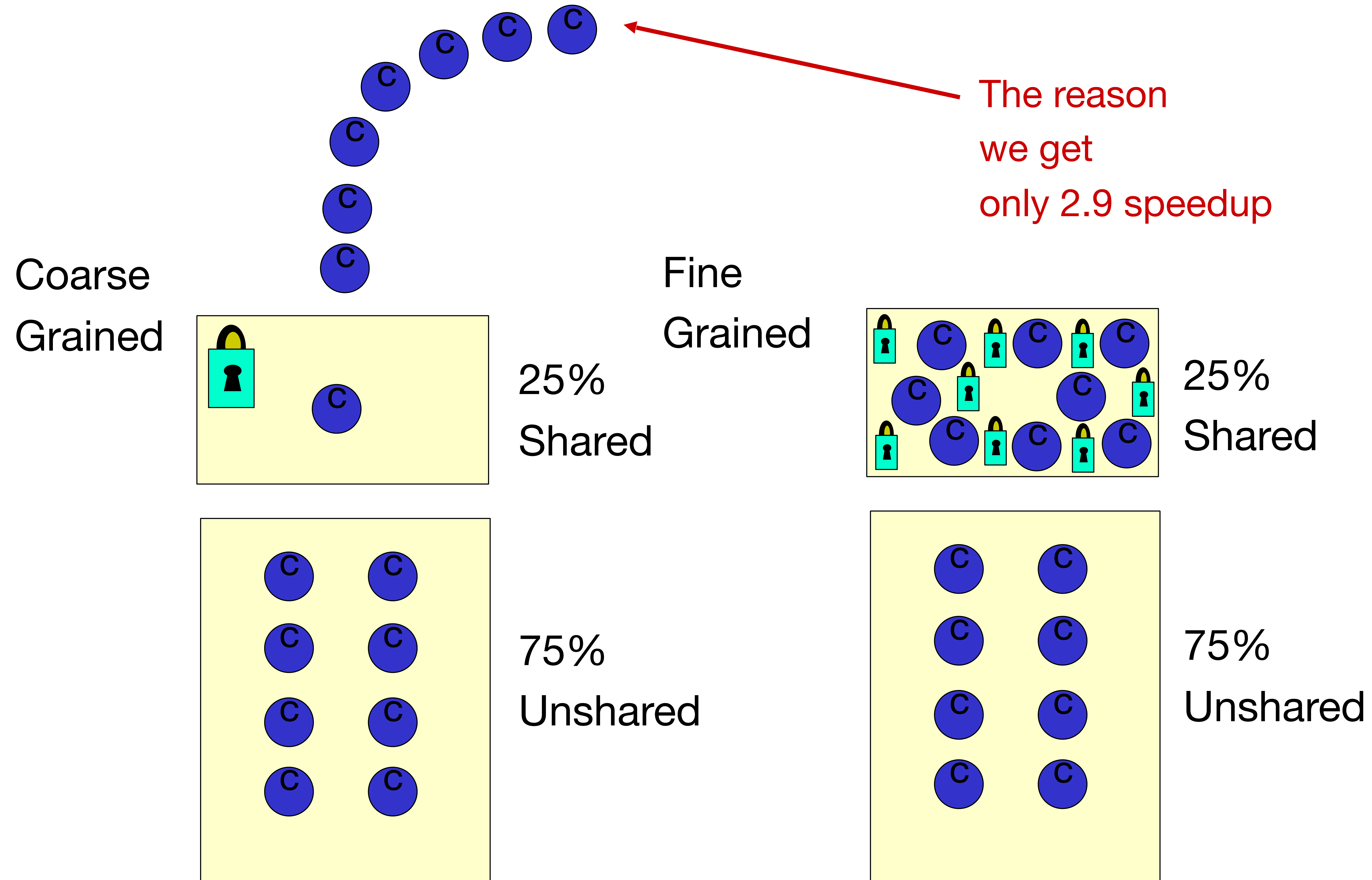
Pay for N = 8 cores

SequentialPart = 25%

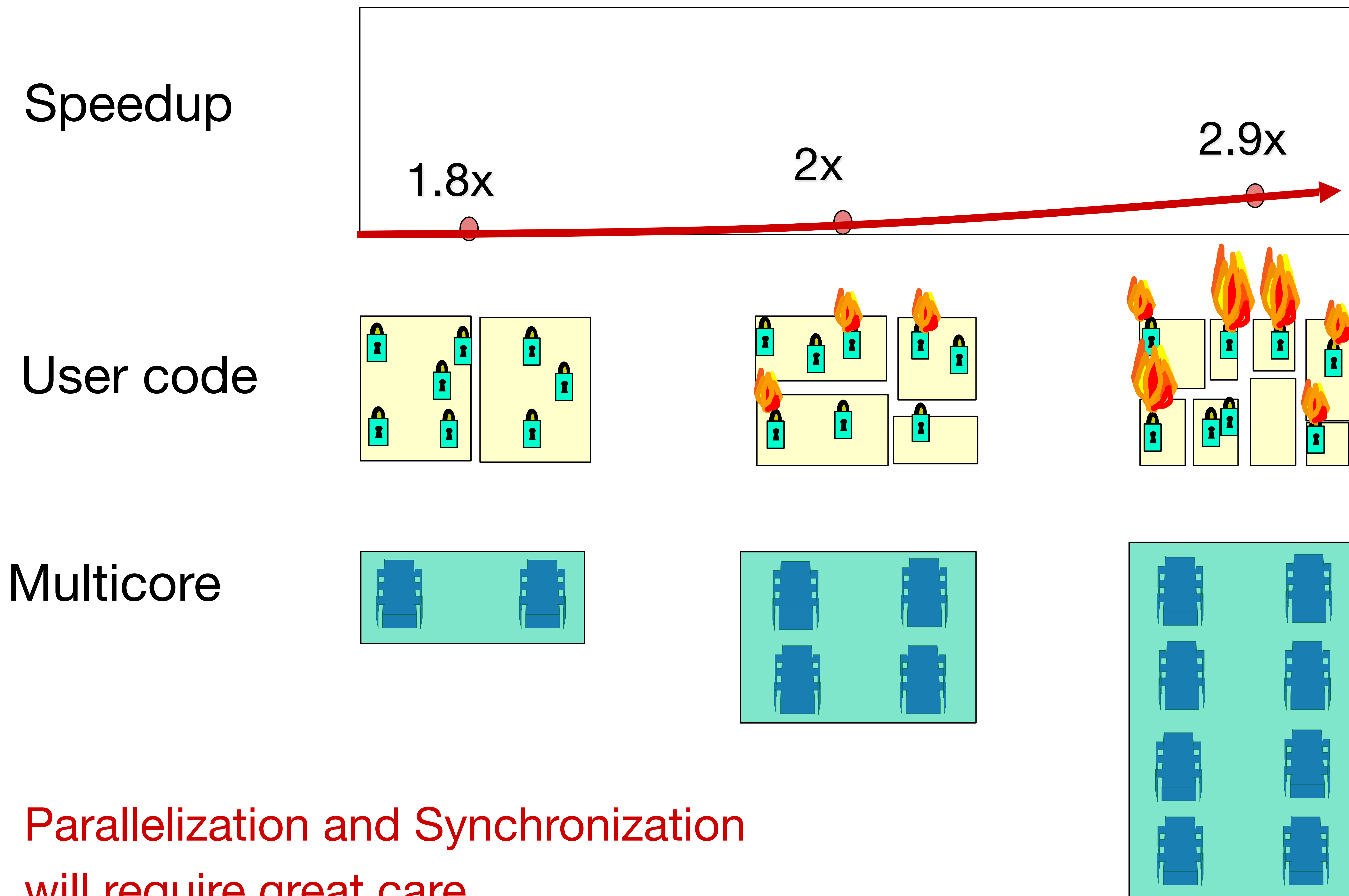
Speedup = only 2.9 times!

Must parallelize applications on a very fine grain!

Need Fine-Grained Locking



Real-World Scaling Process



Parallelization and Synchronization
will require great care…

Today

- How do we increase performance with parallelism?
- How do we split up our program into concurrent sections effectively?
- Different models for parallel computation
- Reading: H&S 16.1, 16.2

Designing for Performance

- What factors can impact performance?
  - Limits imposed by physics
  - Limits imposed by technology
  - Limits imposed by economics
- These limits can force us to make tradeoffs
  - Smaller chips are faster, but harder to dissipate heat
  - Need to serve X clients, can only spend Y on CPUs

Performance Metrics

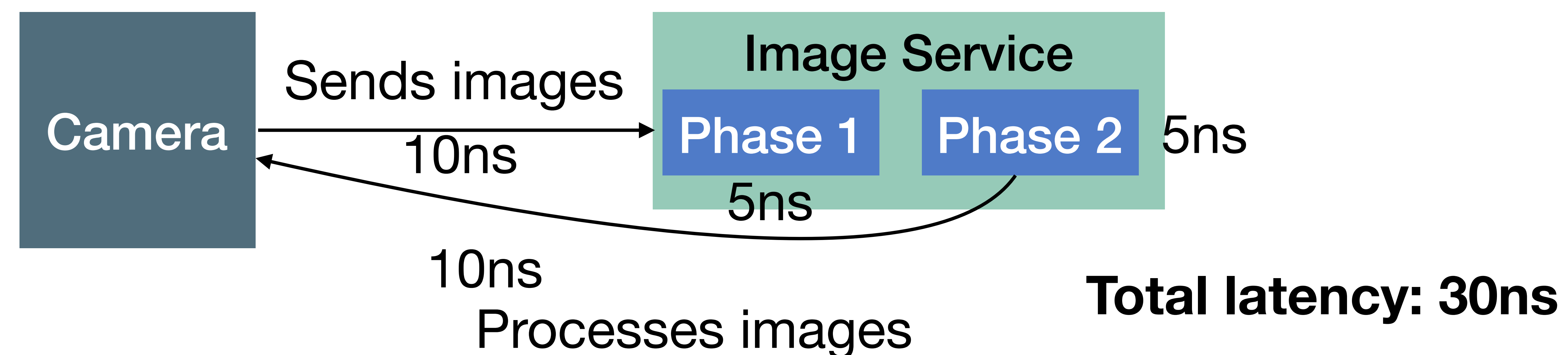
- Capacity
  - Consistent measure of a service's size or amount of resources
- Utilization
  - Percentage of that resource used for a workload
- Latency
  - How long it takes an input to propagate through a system and generate an output
- Throughput
  - Work done per time

Adjusted by buying more resources

Adjusted by thinking hard about the problem

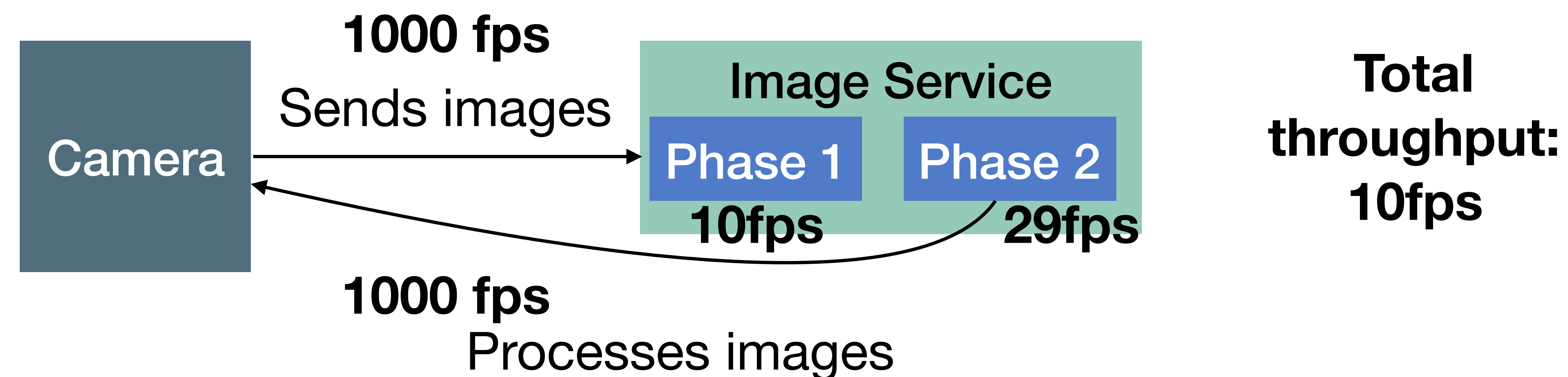
Latency

- In client/server model, latency is simply: time between client sending request and receiving response
- What contributes to latency?
  - Latency sending the message
  - Latency processing the message
  - Latency sending the response
- Adding pipelined components -> latency is cumulative



Throughput

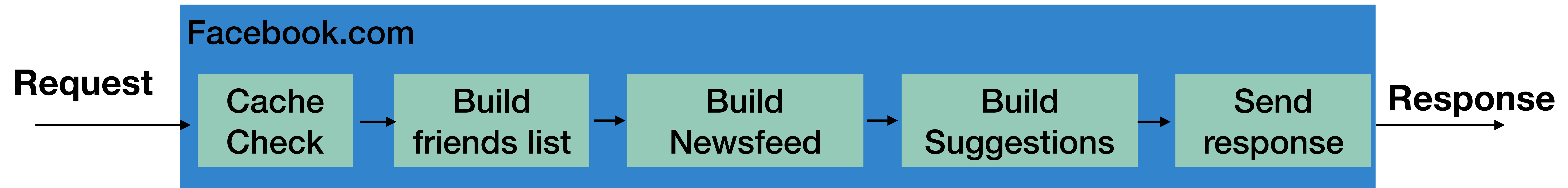
- Measure of the rate of useful work done for a given workload
- Example:
  - Throughput is camera frames processed/second
- When adding multiple pipelined components -> throughput is the minimum value



Designing for Performance

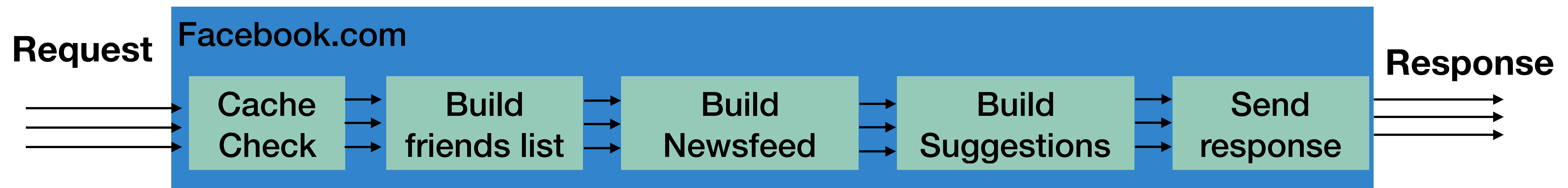
- Measure system to find which aspect of performance is lacking (throughput or latency)
- Measure each component to identify bottleneck
- Identify if fixing that bottleneck will realistically improve system performance
- Measure improvement
- Repeat

Improving Throughput



Improving Throughput

- Introduce concurrency into our pipeline
- Each stage runs in its own thread (or many threads, perhaps)
- If a stage completes its task, it can start processing the next request right away
  - E.g. our system will process multiple requests at the same time



Reducing Latency

- Often more challenging than increasing throughput
  - Examples:
    - Physical - Speed of light (network transmissions over long distances)
    - Algorithmic - Looking up an item in a hash table is limited by hash function
    - Economic - Adding more RAM gets expensive

Latency & Stock Trading

- Buy low/sell high
- Most of skill is in knowing what a stock will do **before** your competitors

Latency & Stock Trading

- Algorithmic trading -> computer programs look at various factors, place trades automatically
- Example:
  - President Trump tweets positively about a company -> price goes up
  - Write a script to check twitter for company mentions, immediately buy/sell stock
  - Get in and out before it hits CNN!
  - https://www.npr.org/sections/money/2017/04/07/522897876/meet-botus-planet-money-s-stock-trading-twitter-bot

Latency & Stock Trading

- This only works if you can make your trades **before** other people find out
- What if you set up this bot in Chicago, and I set one up in NYC?
  - I would beat you to it, every time.

Latency & Stock Trading

- What is the speed of light?
  - ~300,000 km/sec
- How fast does your CPU execute an instruction?
  - 0.33 nanoseconds (say, 3Ghz CPU)
- How far does light travel in 1 CPU cycle?
  - 10 cm
- How many instructions does your CPU execute in the time it takes light to travel from Chicago to NYC and back?
  - ~700 miles -> 7.4msec -> 22 million instructions
- Being in NYC would let me execute 22 million instructions in the time it took you to send your stock order to NYC and get a response!

Reducing Latency with $$$$

- People actually care a LOT about the latency between NYC and Chicago, because commodities are traded in Chicago and stocks are traded in NYC
  - Changes to commodities prices (e.g. **ethanol**) can dramatically impact price of some stocks

Reducing Latency with $$$$

- It’s not quite as simple as 700 miles -> 7.4msec
- There are streams, mountains, etc… more like 1,000 miles
- Light is refracted in a fiber optic cable is ~31% slower
- What do we do if money is no object?

Reducing Latency with Billions of Dollars

ORIGINAL CABLE

Technology
Buried fiber-optic cable

Completion
Mid-1980s

Path length
~ 1,000 miles

Round-trip time for data
14.5
milliseconds and up

Approach
Multiple routes followed the easiest rights-of-way—along rail lines. But that means time-sucking jogs and detours.

SPREAD NETWORKS

Technology
Buried fiber-optic cable

Completion
August 2010

Path length
825 miles

Round-trip time for data
13.1
milliseconds

Approach
Spread bought its own rights-of-way, avoiding a Philadelphia-ward dip in favor of a shorter path northwest through central Pennsylvania.

MCKAY BROTHERS

Technology
Microwave beams through air

Completion
July 4, 2012

Path length
744 miles

Round-trip time for data
9
milliseconds

Approach
Microwaves generally move faster than photons in optical fiber, and McKay's network uses just 20 towers on a nearly perfect great circle.

TRADEWORX

Technology
Microwave beams through air

Completion
Winter 2012

Path length
~ 731 miles

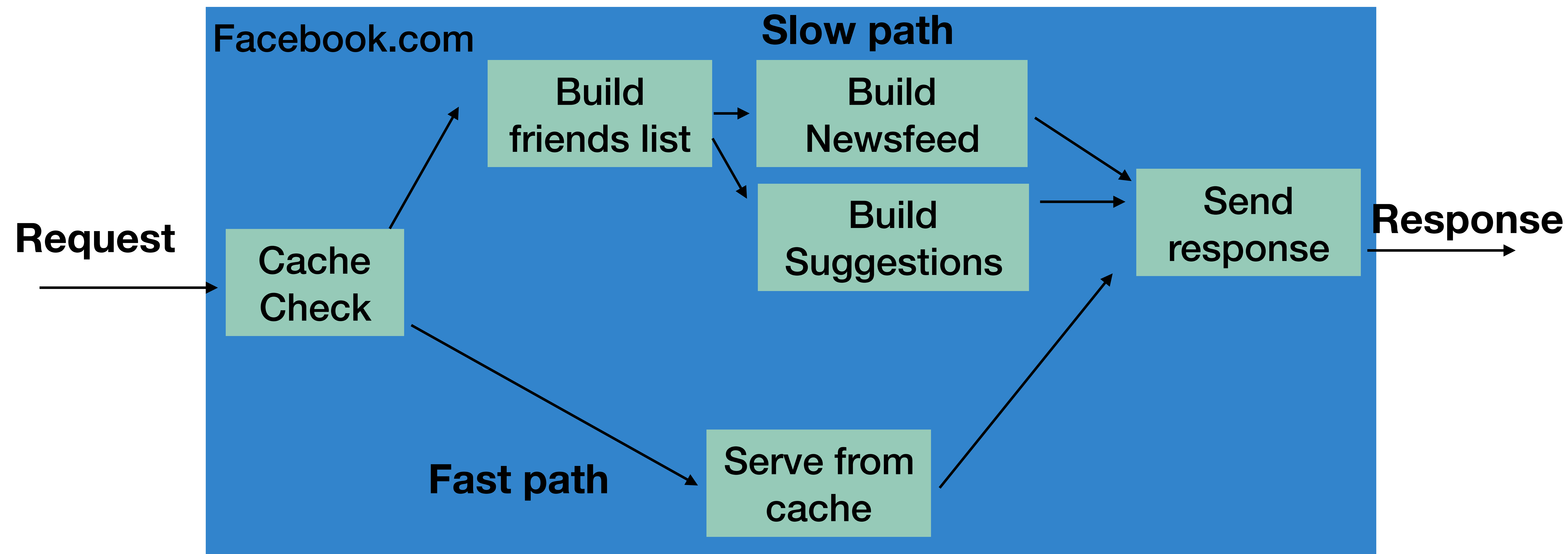
Round-trip time for data
8.5
milliseconds (est.)

Approach
Tradeworx is highly secretive, but the company is open about the price of a subscription: $250,000 a year.

https://www.zerohedge.com/news/chicago-new-york-and-back-85-milliseconds

Reducing Latency without lots of $$$

- Approach: use **concurrency**
- Limited by serial section



Exploiting Concurrency

- These examples are at a very high level (components in a large server system)
- For this lecture, we'll focus on smaller, more concrete examples
- First: Matrix Multiplication

$$(C) = (A) \cdot (B)$$

Matrix Multiplication

$$c_{ij} = \sum_{k=0}^{N-1} a_{ki} * b_{jk}$$

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += a[row][i] * b[i][col];
    c[row][col] = dotProduct;
  }}}
```

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += a[row][i] * b[i][col];
    c[row][col] = dotProduct;
  }}}
```

a thread

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += a[row][i] * b[i][col];
    c[row][col] = dotProduct;
}}}
```

Which matrix entry to compute

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += a[row][i] * b[i][col];
    c[row][col] = dotProduct;
}}}
```

Actual computation

Matrix Multiplication

```
 void multiply() {
   Worker[][] worker = new Worker[n][n];
   for (int row …)
     for (int col …)
       worker[row][col] = new Worker(row,col);
   for (int row …)
     for (int col …)
       worker[row][col].start();
   for (int row …)
     for (int col …)
       worker[row][col].join();
}
```

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row …)
    for (int col …)
      worker[row][col] = new Worker(row,col);
  for (int row …)
    for (int col …)
      worker[row][col].start();
  for (int row …)
    for (int col …)
      worker[row][col].join();
}
```

Create nxn threads

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row …)
    for (int col …)
      worker[row][col] = new Worker(row,col);
  for (int row …)
    for (int col …)
      worker[row][col].start();
  for (int row …)
    for (int col …)
      worker[row][col].join();
}
```

Start them

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row …)
    for (int col …)
      worker[row][col] = new Worker(row,col);
  for (int row …)
    for (int col …)
      worker[row][col].start();
  for (int row …)
    for (int col …)
      worker[row][col].join();
}
```

Start them

Wait for them to finish

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row …)
    for (int col …)
      worker[row][col] = new Worker(row,col);
  for (int row …)
    for (int col …)
      worker[row][col].start();
  for (int row …)
    for (int col …)
      worker[row][col].join();
}
```

Start them

Wait for them to finish

What's wrong with this picture?

Thread Overhead

- Threads Require resources
  - Memory for stacks
  - Setup, teardown
- Scheduler overhead
- Worse for short-lived threads

Thread Pools

- More sensible to keep a pool of long-lived threads
- Threads assigned short-lived tasks
  - Runs the task
  - Rejoins pool
  - Waits for next assignment

Thread Pool = Abstraction

- Insulate programmer from platform
  - Big machine, big pool
  - And vice-versa
- Portable code
  - Runs well on any platform
  - No need to mix algorithm/platform concerns

ExecutorService Interface

- In java.util.concurrent
  - Task = **Runnable** object
    - If no result value expected
    - Calls **run()** method.
  - Task = **Callable<T>** object
    - If result value of type **T** expected
    - Calls **T call()** method.
    - Interesting question: how do you get the return value from call?

Future<T>

```
Callable<T> task = …;
…
Future<T> future = executor.submit(task);
…
T value = future.get();
```

Future<T>

```
Callable<T> task = …;
…
Future<T> future = executor.submit(task);
…
T value = future.get();
```

Submitting a Callable<T> task returns a Future<T> object

Future<T>

```
Callable<T> task = …;
…
Future<T> future = executor.submit(task);
…
T value = future.get();
```

The Future's get() method blocks until the value is available

Future<?>

```
Runnable task = …;
…
Future<?> future = executor.submit(task);
…
future.get();
```

Future<?>

```
Runnable task = …;
…
Future<?> future = executor.submit(task);
…
future.get();
```

Submitting a Runnable task returns a Future<?> object

Future<?>

```
Runnable task = …;
…
Future<?> future = executor.submit(task);
…
future.get();
```

The Future's get() method blocks until the computation is complete

Note

- Executor Service submissions
  - Like Maryland traffic signs
  - Are purely advisory in nature
- The executor
  - Like the Maryland driver
  - Is free to ignore any such advice
  - And could execute tasks sequentially …

Matrix Addition

$$\begin{pmatrix} C_{00} & C_{00} \\ C_{10} & C_{10} \end{pmatrix} = \begin{pmatrix} A_{00} + B_{00} & B_{01} + A_{01} \\ A_{10} + B_{10} & A_{11} + B_{11} \end{pmatrix}$$

Matrix Addition

$$\begin{pmatrix} C_{00} & C_{00} \\ C_{10} & C_{10} \end{pmatrix} = \begin{pmatrix} A_{00} + B_{00} & B_{01} + A_{01} \\ A_{10} + B_{10} & A_{11} + B_{11} \end{pmatrix}$$

4 parallel additions

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // multiply this!
  public void run() {
    if (a.dim == 1) {
      c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
      (partition a, b into half-size matrices aij and bij)
      Future<?> f00 = exec.submit(add(a00,b00));
      …
      Future<?> f11 = exec.submit(add(a11,b11));
      f00.get(); …; f11.get();
      …
    }}
```

This is not real Java code (see book)

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // multiply this!
  public void run() {
    if (a.dim == 1) {
     c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
     (partition a, b into half-size matrices aij and bij)
     Future<?> f00 = exec.submit(add(a00,b00));
     …
     Future<?> f11 = exec.submit(add(a11,b11));
     f00.get(); …; f11.get();
     …
    }}
```

Base case: add directly

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // multiply this!
  public void run() {
    if (a.dim == 1) {
      c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
      (partition a, b into half-size matrices aij and bij)
      Future<?> f00 = exec.submit(add(a00,b00));
      …
      Future<?> f11 = exec.submit(add(a11,b11));
      f00.get(); …; f11.get();
      …
    }}
```

Constant-time operation

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // multiply this!
  public void run() {
    if (a.dim == 1) {
     c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
     (partition a, b into half-size matrices a_ij and b_ij)
     Future<?> f_00 = exec.submit(add(a_00,b_00));
     …
     Future<?> f_11 = exec.submit(add(a_11,b_11));
     f_00.get(); …; f_11.get();
     …
    }}
```

Submit 4 tasks

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // multiply this!
  public void run() {
    if (a.dim == 1) {
     c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
     (partition a, b into half-size matrices aij and bij)
     Future<?> f00 = exec.submit(add(a00,b00));
     …
     Future<?> f11 = exec.submit(add(a11,b11));
     f00.get(); …; f11.get();
     …
    }}
```

Let them finish

Dependencies

- Matrix example is not typical
- Tasks are independent
  - Don't need results of one task …
  - To complete another
- Often tasks are not independent

Fibonacci

- Note: potential parallelism, but subject to dependencies

$$F(n) \begin{cases} 1 \text{ if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) \text{ otherwise} \end{cases}$$

Disclaimer

- This Fibonacci implementation is
  - Egregiously inefficient
    - So don't deploy it!
  - But illustrates our point
    - How to deal with dependencies

Multithreaded Fibonacci

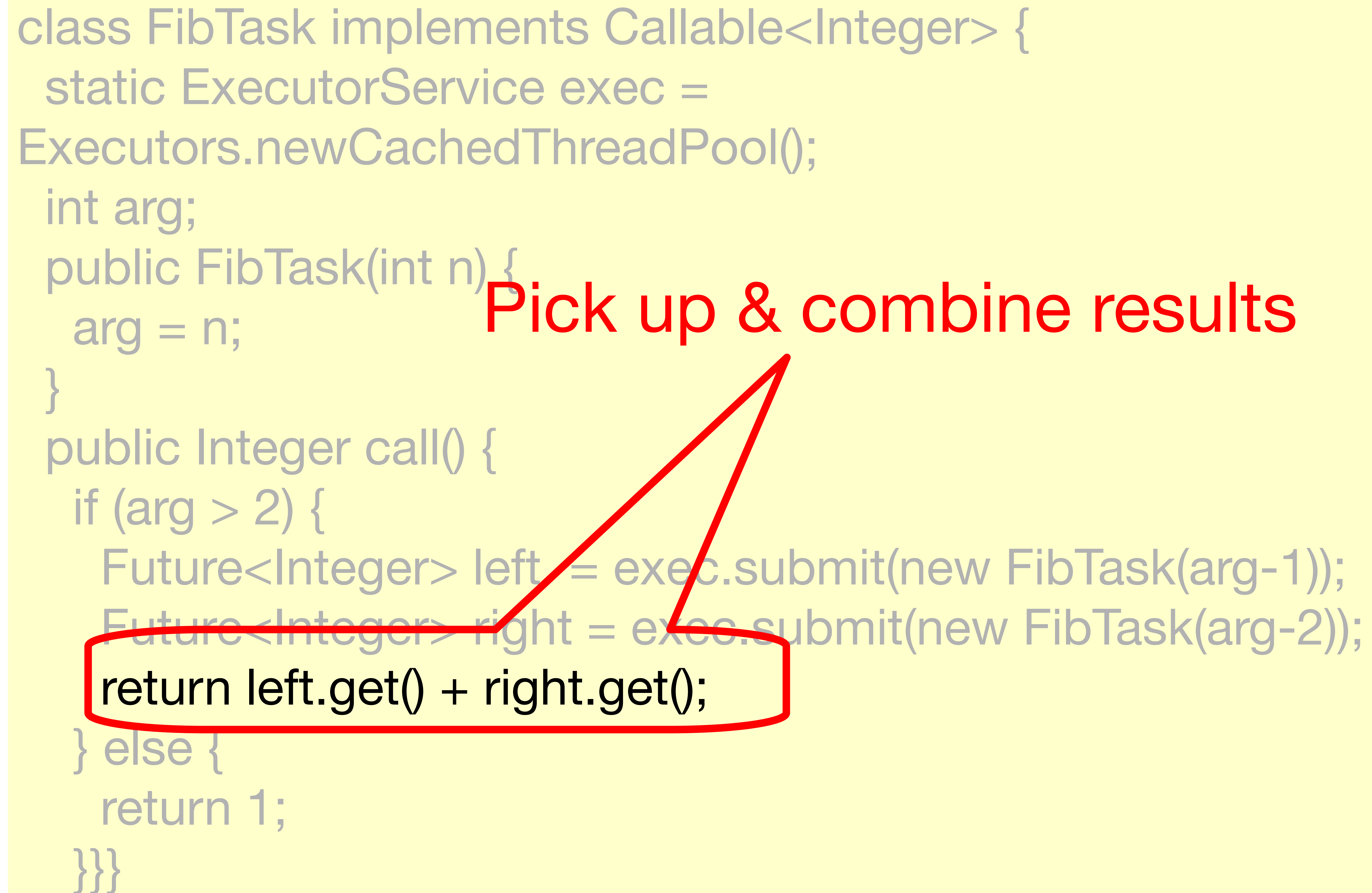
```
class FibTask implements Callable<Integer> {
  static ExecutorService exec = 
Executors.newCachedThreadPool();
  int arg;
  public FibTask(int n) {
    arg = n;
  }
  public Integer call() {
    if (arg > 2) {
      Future<Integer> left  = exec.submit(new FibTask(arg-1));
      Future<Integer> right = exec.submit(new FibTask(arg-2));
      return left.get() + right.get();
    } else {
      return 1;
    }}}
```

Multithreaded Fibonacci

```
class FibTask implements Callable<Integer> {
  static ExecutorService exec = 
Executors.newCachedThreadPool();
  int arg;
  public FibTask(int n) {
    arg = n;
  }
  public Integer call() {
    if (arg > 2) {
      Future<Integer> left  = exec.submit(new FibTask(arg-1));
      Future<Integer> right = exec.submit(new FibTask(arg-2));
      return left.get() + right.get();
    } else {
      return 1;
    }}}
```

Parallel calls

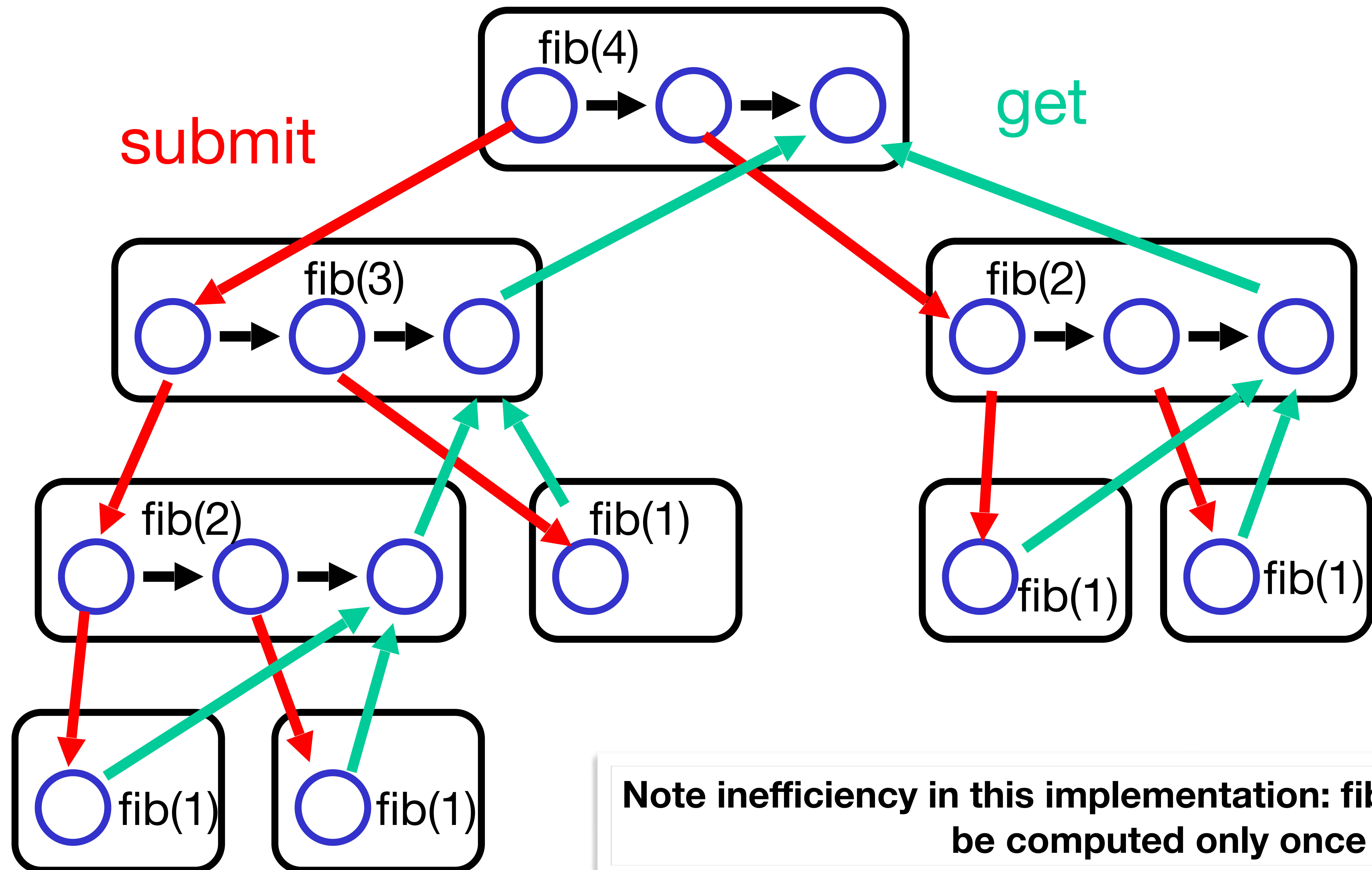
Multithreaded Fibonacci



Dynamic Behavior

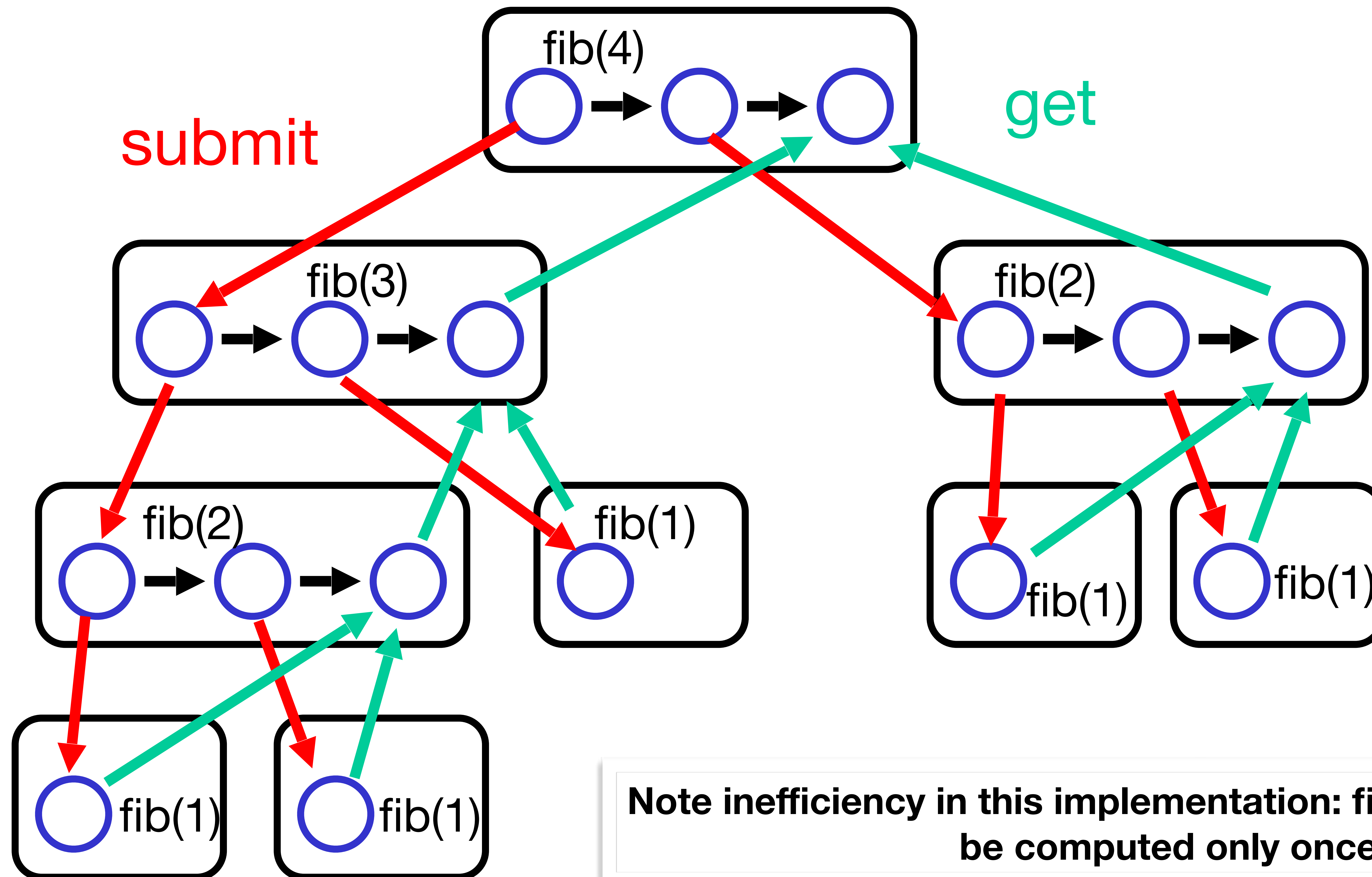
- Multithreaded program is
  - A directed acyclic graph (DAG)
  - That unfolds dynamically
- Each node is
  - A single unit of work

Fib DAG



Note inefficiency in this implementation: fib(2)'s result should be computed only once

Arrows Reflect Dependencies

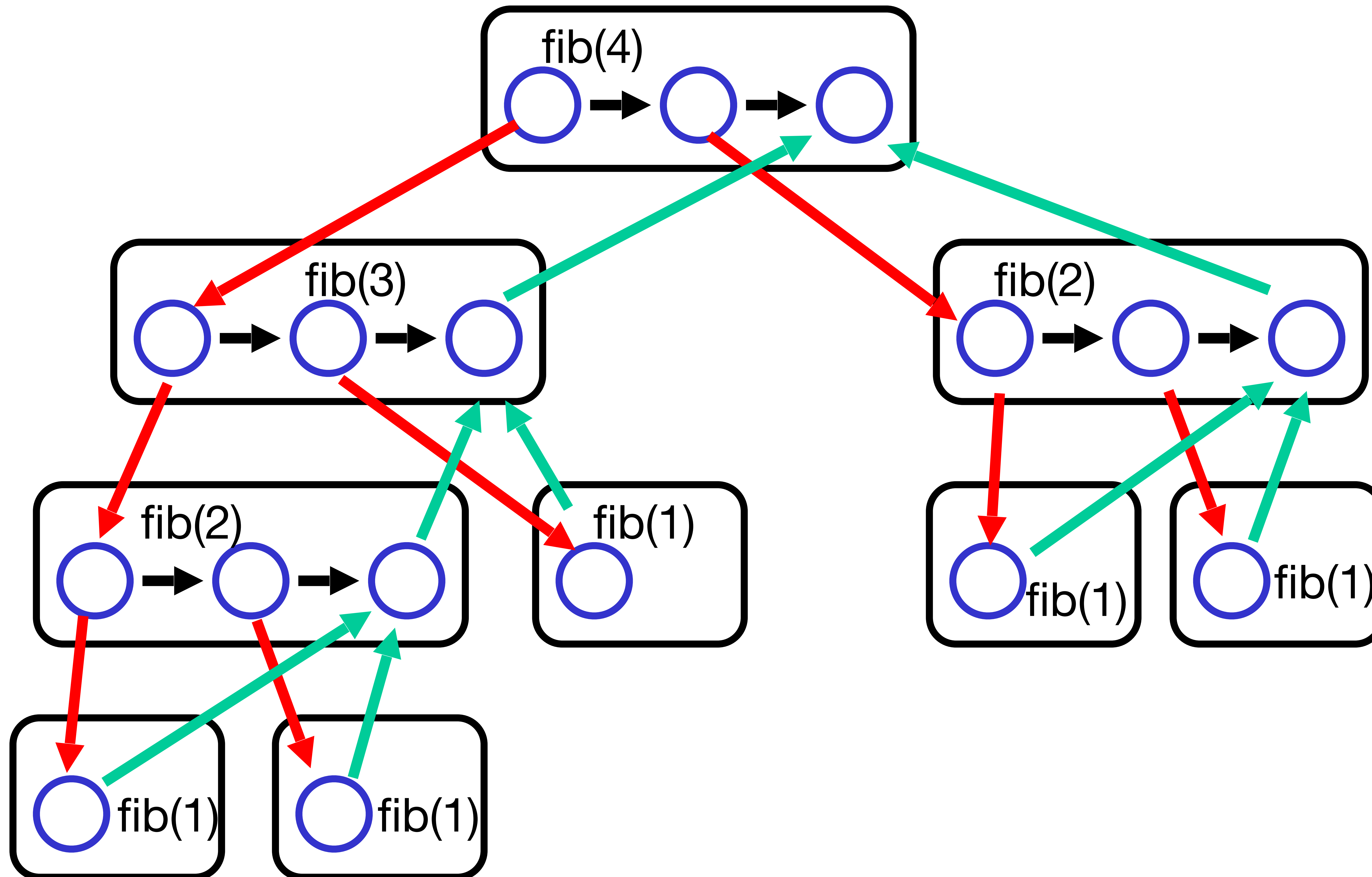


Note inefficiency in this implementation: fib(2)'s result should be computed only once

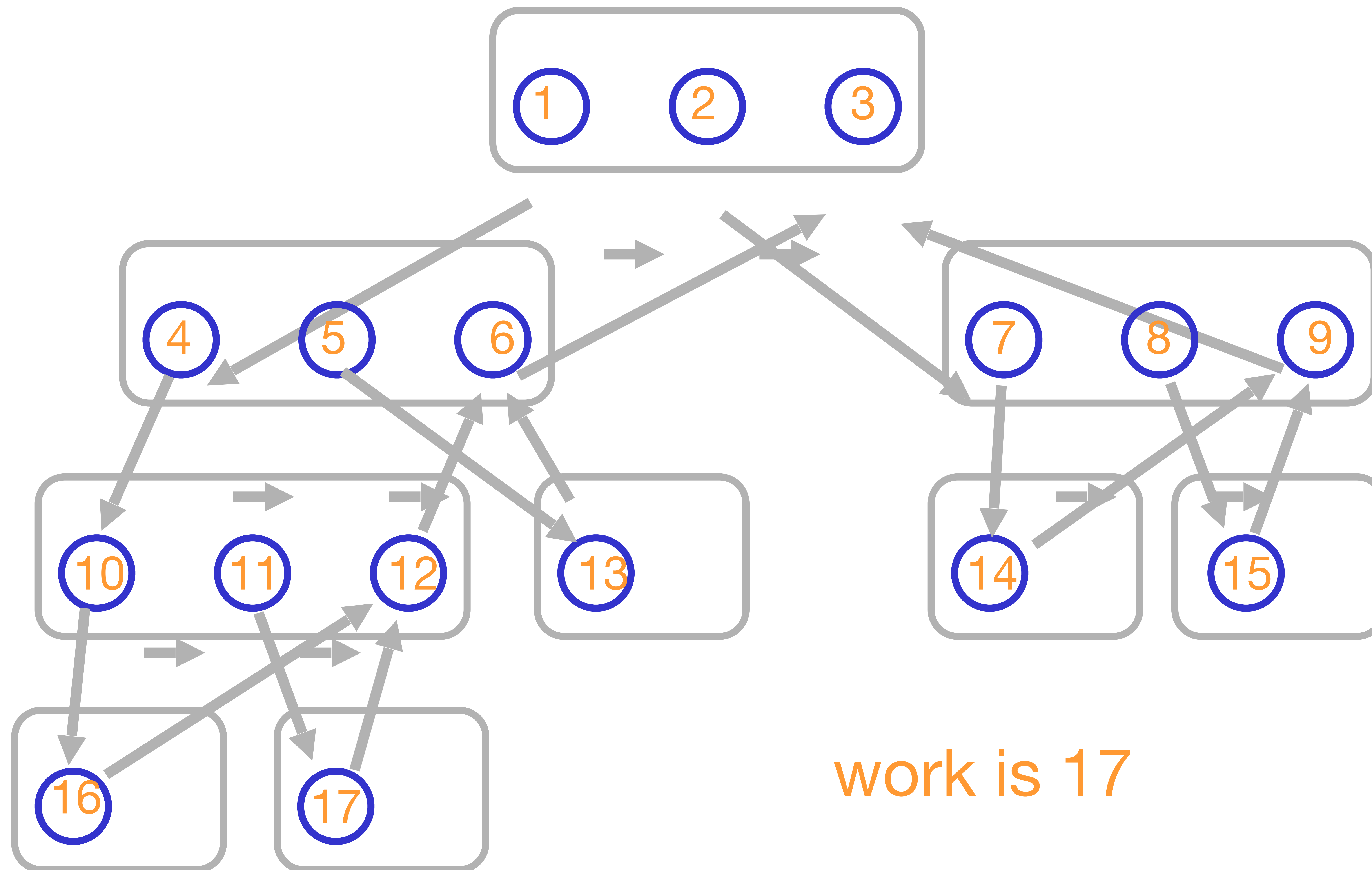
How Parallel is That?

- Define work:
  - Total time on one processor
- Define critical-path length:
  - Longest dependency path
  - Can't beat that!

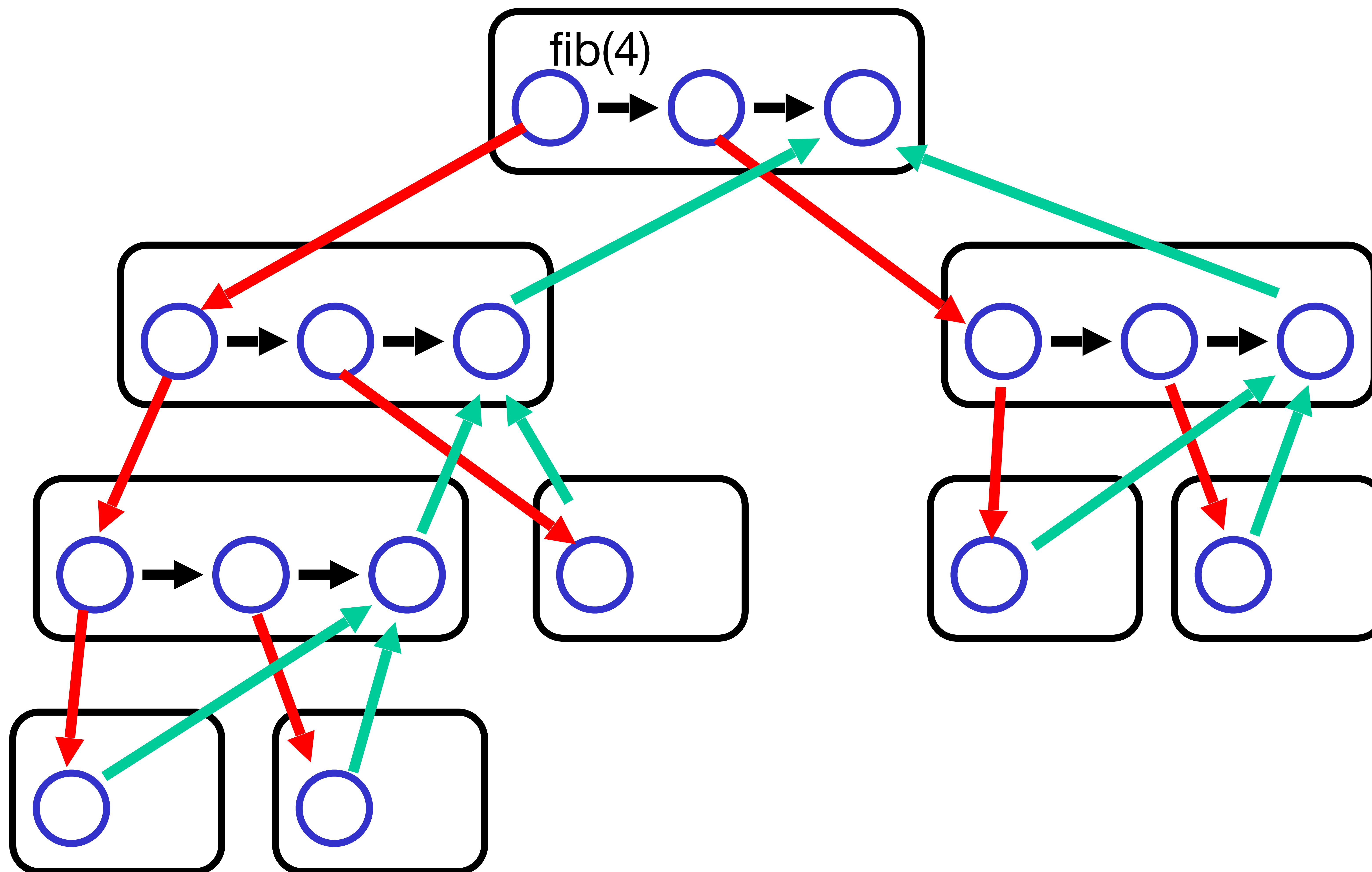
Fib Work



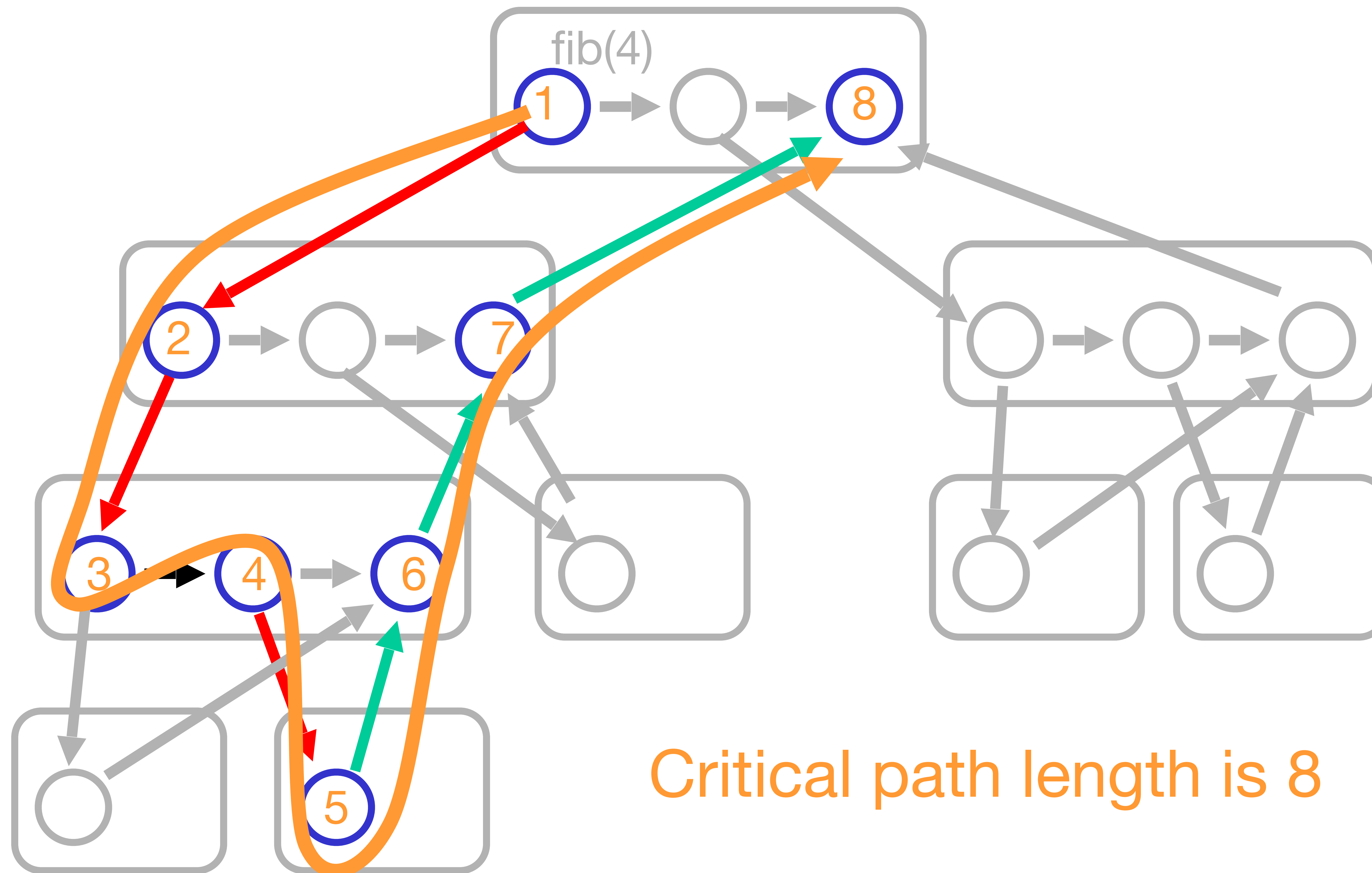
Fib Work



Fib Critical Path



Fib Critical Path



Notation Watch

- T_P = time on P processors
- T_1 = work (time on 1 processor)
- T_∞ = critical path length (time on ∞ processors)

Simple Bounds

- $T_P \geq T_1/P$
  - In one step, can't do more than P work
- $T_P \geq T_\infty$
  - Can't beat infinite resources

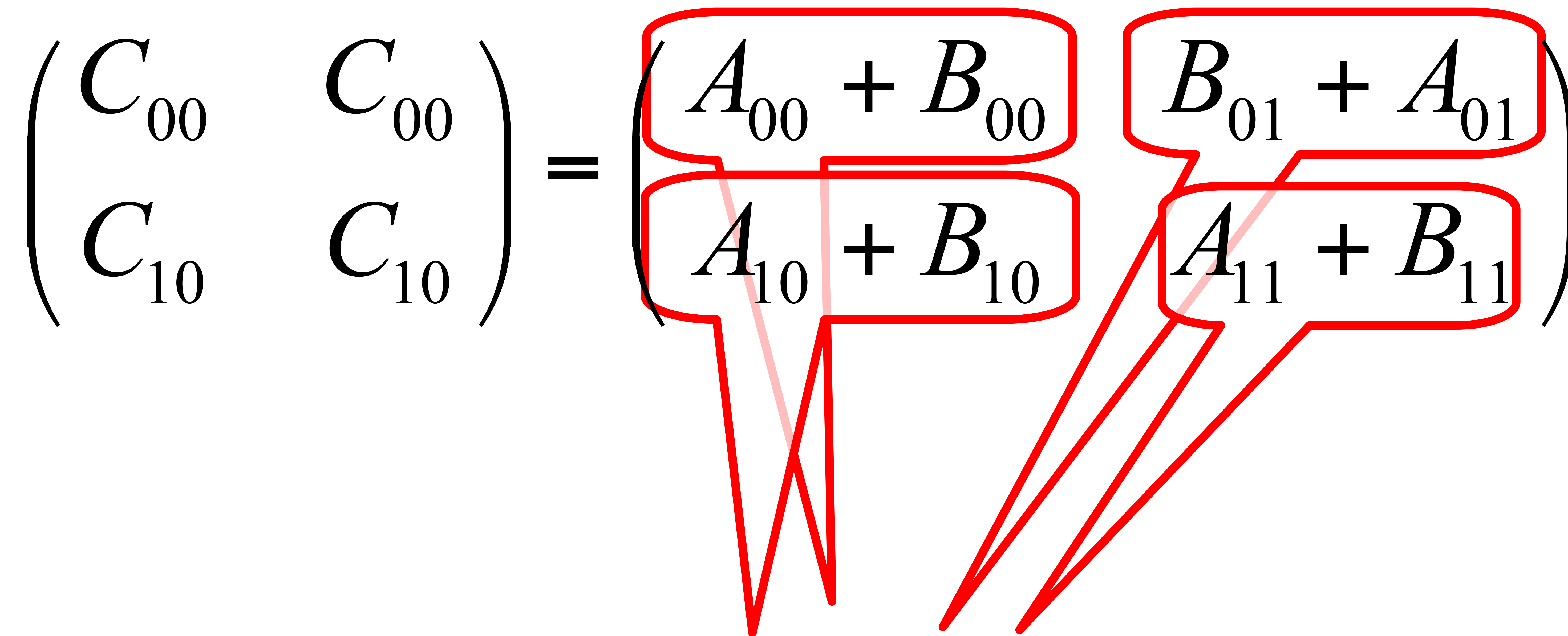
More Notation Watch

- Speedup on P processors
  - Ratio T_1/T_P
  - How much faster with P processors
- Linear speedup
  - $T_1/T_P = \Theta(P)$
- Max speedup (average parallelism)
  - T_1/T_∞

Matrix Addition

$$\begin{pmatrix} C_{00} & C_{00} \\ C_{10} & C_{10} \end{pmatrix} = \begin{pmatrix} A_{00} + B_{00} & B_{01} + A_{01} \\ A_{10} + B_{10} & A_{11} + B_{11} \end{pmatrix}$$

Matrix Addition



$$\begin{pmatrix} C_{00} & C_{00} \\ C_{10} & C_{10} \end{pmatrix} = \begin{pmatrix} A_{00} + B_{00} & B_{01} + A_{01} \\ A_{10} + B_{10} & A_{11} + B_{11} \end{pmatrix}$$

Addition

- Let $A_P(n)$ be running time
  - For n x n matrix
  - on P processors
- For example
  - $A_1(n)$ is work
  - $A_\infty(n)$ is critical path length

Addition

- Work is

Partition, synch, etc

$$A_1(n) = 4\ A_1(n/2) + \Theta(1)$$

4 spawned additions

Addition

- Work is

$$A_1(n) = 4\ A_1(n/2) + \Theta(1)$$
$$= \Theta(n^2)$$

Same as double-loop summation

Addition

- Critical Path length is

$$A_\infty(n) = A_\infty(n/2) + \Theta(1)$$

spawned additions in parallel

Partition, synch, etc

Addition

- Critical Path length is

$$A_\infty(n) = A_\infty(n/2) + \Theta(1)$$
$$= \Theta(\log n)$$

Matrix Multiplication Redux

$$(C) = (A) \cdot (B)$$

Matrix Multiplication Redux

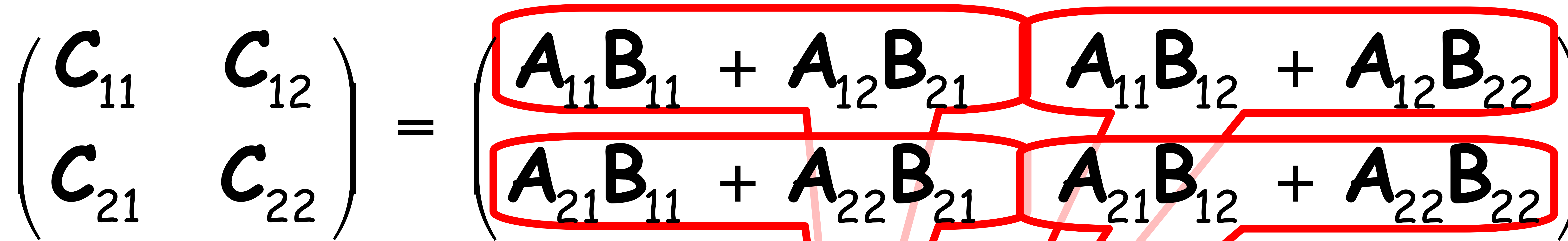
$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

First Phase ...

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

8 multiplications

Second Phase …



Multiplication

- Work is

Final addition

$$M_1(n) = 8\, M_1(n/2) + A_1(n)$$

8 parallel multiplications

Multiplication

- Work is

$$M_1(n) = 8\ M_1(n/2) + \Theta(n^2)$$
$$= \Theta(n^3)$$

Same as serial triple-nested loop

Multiplication

- Critical path length is

Final addition

$$M_\infty(n) = M_\infty(n/2) + A_\infty(n)$$

Half-size parallel multiplications

Multiplication

- Critical path length is

$$
\begin{aligned}
M_\infty(n) &= M_\infty(n/2) + A_\infty(n) \\
&= M_\infty(n/2) + \Theta(\log n) \\
&= \Theta(\log^2 n)
\end{aligned}
$$

Parallelism

- $M_1(n)/ M_\infty(n) = \Theta(n^3/\log^2 n)$
- To multiply two 1000 x 1000 matrices
  - $1000^3/10^2=10^7$
- Much more than number of processors on any real machine

Shared-Memory Multiprocessors

- Parallel applications
  - Do not have direct access to HW processors
- Mix of other jobs
  - All run together
  - Come & go dynamically
- Hence, we have **no control** over how many processors we get at any given point
- Instead, shoot for the best parallelism that we can get given however many processors we actually get

Back to the `Future`

Promises & CompleteableFutures

- What if we want to run some task, and do stuff while we are waiting for it to be done?
- You COULD do it with a complicated combination of **synchronized**, **wait**, and **notify**
- You can use the **Promise** abstraction instead
  - Called a **CompletableFuture** in Java 8

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Result of the asynchronous computation";
});
// Block and get the result of the Future
String result = future.get();
System.out.println(result);
```

- Just like Future's from before, but supports *chaining*

Chaining CompletableFuture

```
CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Jon";
});
// Chain on some more code to run when the future is done
CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(returnValue -> {
    return “Hello, ” + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```

Chaining CompletableFuture

```
CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Jon";
});
// Chain on some more code to run when the future is done
CompletableFuture<String> greetingFuture =
    return “Hello, ” + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```

Create an asynchronous task

Chaining CompletableFuture

Task will return string “Jon” eventually

```
CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Jon";
});
// Chain on some more code to run when the future is done
CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(returnValue -> {
    return “Hello, ” + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```

Chaining CompletableFuture

Task will return string “Jon” eventually

```
ompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Jon";
});
// Chain on some more code to run when the future is done
CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(returnValue -> {
    return “Hello, ” + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```

Chaining CompletableFuture

```
CompletableFuture<String> whatsYourName
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Jon";
});
// Chain on some more code to run when the future is done
CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(returnValue -> {
    return “Hello, ” + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```

Create ANOTHER future that is *chained* to the first

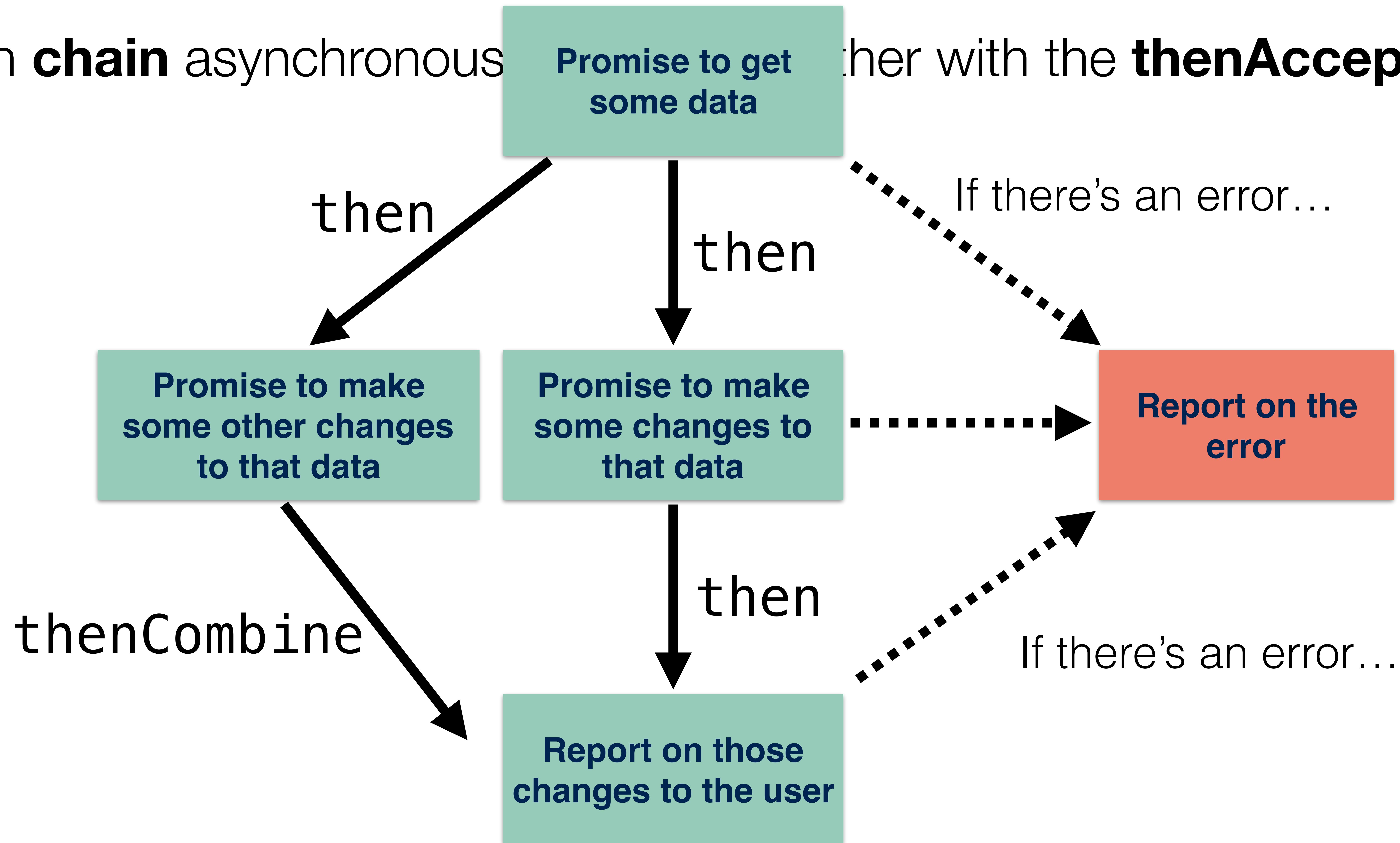
Chaining CompletableFuture

```
CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    re
});
// Chain on some more code to run when the future is done
CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(returnValue -> {
    return "Hello, " + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```

Block the main thread for both futures to finish

Chaining CompletableFutures

- We can **chain** asynchronous [obscured] [obscured]her with the **thenAccept** term



CompleteableFuture Use-Cases

- Any case where you need to have multiple things happen in the background, but care about the result, and care about them happening in some order
- Asynchronous I/O
  - Read data from a web service
  - Then process it
  - Then save it to a file

This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/
- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - for any purpose, even commercially.
- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.