Concurrent Programming Models

CS 475, Spring 2019
Concurrent & Distributed Systems

With material from Herlihy & Shavit, Art of Multiprocessor Programming
Review: Course Grained Locking

Simple but hotspot + bottleneck
Review: Fine-Grained Synchronization

• Instead of using a single lock..
• Split object into
  – Independently-synchronized components
• Methods conflict when they access
  – The same component …
  – At the same time
Review: Fine Grained Locking List
Review: Optimistic Synchronization

- Search without locking …
- If you find it, lock and check …
  - OK: we are done
  - Oops: start over
- Evaluation
  - Usually cheaper than locking
  - Mistakes are expensive
Review: Optimistic List

add(c)
Review: Lazy Synchronization

• Postpone hard work
• Removing components is tricky
  – Logical removal
    • Mark component to be deleted
  – Physical removal
    • Do what needs to be done
Returns false, because b is marked. No need to validate or lock or re-traverse

Intuition: Now can judge if b is in list ONLY by looking at b, don’t also need to look at a
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
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

![Diagram showing latency in a client/server model](image)

Camera

Image Service

Phase 1

Phase 2

Total latency: 30ns
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

![Diagram showing camera, image service, phase 1, phase 2, and total throughput of 10fps]
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

Facebook.com

Request

Cache Check → Build friends list → Build Newsfeed → Build Suggestions → Send response → Response
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!
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 beet 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

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

\[ c_{ij} = \sum_{k=0}^{N-1} a_{ki} \times b_{jk} \]
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;
    }
}
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;
    }
}
Matrix Multiplication

```java
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

```java
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

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

```java
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();
        }
    }
}
```

What's wrong with this picture?

Start them

Wait for them to finish
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&lt;T&gt; object
    • If result value of type T expected
    • Calls T call() method.
    • Interesting question: how do you get the return value from call?
Callable<T> task = ...;
...
Future<T> future = executor.submit(task);
...
T value = future.get();
Callable<T> task = ...;
...

Future<T> future = executor.submit(task);
...

T value = future.get();

Submitting a Callable<T> task returns a Future<T> object
Callable<T> task = ...;
...
Future<T> future = executor.submit(task);
...
T value = future.get();

The Future’s get() method blocks until the value is available
Runnable task = ...;
...
Future<?> future = executor.submit(task);
...
future.get();
Submitting a Runnable task returns a Future<?> object
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

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

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 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();
            ...
        }
    }
}
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();
      ...
    }
  }
}

Base case: add directly
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<?> f00 = exec.submit(add(a00, b00));
            ...
            Future<?> f11 = exec.submit(add(a11, b11));
            f00.get(); ...; f11.get();
        }
    }
}
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 {
            Future<?> f00 = exec.submit(add(a00, b00));
            ...
            Future<?> f11 = exec.submit(add(a11, b11));
            f00.get(); ...; f11.get();
            ...
        }
    }
}
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<? extends Number> f_{00} = exec.submit(add(a_{00}, b_{00}));
            ...
            Future<? extends Number> f_{11} = exec.submit(add(a_{11}, b_{11}));
            f_{00}.get(); ...; f_{11}.get();
            ...
        }
    }
}
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

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

- Note: potential parallelism, but subject to dependencies
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;
        }
    }
}
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;
        }
    }
}
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

work is 17
Fib Critical Path
Fib Critical Path

Critical path length is 8
Notation Watch

- $T_P = \text{time on } P \text{ processors}$
- $T_1 = \text{work (time on 1 processor)}$
- $T_\infty = \text{critical path length (time on } \infty \text{ processors)}$
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)
  - $T_1/T_\infty$
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

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

4 parallel additions
Addition

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

• Work is

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

4 spawned additions

Partition, synch, etc
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 ...

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

8 multiplications
Second Phase ...

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

4 additions
Multiplication

- Work is

\[ M_1(n) = 8 M_1(n/2) + A_1(n) \]

8 parallel multiplications

Final addition
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

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

Final addition

Half-size parallel multiplications
Multiplication

• Critical path length is

\[ M_\infty(n) = M_\infty(n/2) + A_\infty(n) \]
\[ = M_\infty(n/2) + \Theta(\log n) \]
\[ = \Theta(\log^2 n) \]
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
Concurrent Programming Models
Asynchronous Programming

• AKA event-driven programming
• A paradigm that lends itself well to scaling, especially in a multi-stage systems (like the example with Facebook)
• Allows us to think about what is done, abstract away how it is done
• We will discuss two asynchronous models: streams, and Promises, neither of which make you think about threads (or locks?)
Java 8 introduced the concept of **Streams**

A stream is a sequence of objects

Streams have functions that you can perform on them, which are (mostly) **non-interfering** and **stateless**

- Non-interfering: Does not modify the actual stream
- Stateless: Each time the function is called on the same data, get same result

**Example:**

```java
IntStream.range(1, 1000000) //Generate a stream of all ints 1 - 1m
  .filter(x -> isPrime(x))  //Retain only values that pass some expensive isPrime function
  .forEach(System.out::println); //For each value returned by filter, print it
```
Sidebar: Lambdas

• I don't know if you have seen this before
  
  IntStream.range(1, 1000000)
  .filter(x -> isPrime(x))
  .forEach(System.out::println);

• This line is called a lambda expression

• We should have shown it to you before, because it's a core part of Java syntax since Java 8 was released in 2014

• Effectively, think of this as shorthand for:
  
  IntStream.range(1, 1000000)
  .filter(new IntPredicate() {
      @Override
      public boolean test(int x) {
          return isPrime(x);
      }
  })
  .forEach(System.out::println);

• In fact, javac generates exactly the long-hand code for that shorthand (but that println is cool, right?)
Streams

IntStream.range(1, 1000000) //Generate a stream of all ints 1 - 1m
.filter(x -> isPrime(x)) //Retain only values that pass some expensive isPrime function
.forEach(System.out::println); //For each value returned by filter, print it

• Why use the stream interface instead of

for(int i = 1; i < 1000000; i++)
    if(isPrime(x))
        System.out.println(x);

• Who wants to write the parallel version of this?

IntStream.range(1, 1000000) //Generate a stream of all ints 1 - 1m
.filter(x -> isPrime(x)) //Retain only values that pass some expensive isPrime function
.parallel() //Do the filtering in parallel
.forEach(System.out::println); //For each value returned by filter, print it

• The magic works as long as isPrime is stateless!
Streams - what can’t be parallelized

• Interference

List<String> list = new ArrayList<>(Arrays.asList("Luke", "Leia", "Han"));
list.stream()
  .peek(name -> {
    if (name.equals("Han")) {
      list.add("Chewie"); // Adds to list that we are peeking into
    }
  })
.forEach(i -> {});

• Stateful

boolean tooBusy = false;
public void isPrime(int x)
{
  if(tooBusy)
    return false;//don't bother running if another thread set tooBusy
  else
    //do a sieve of erasthenes
}

• Side effects

List<Integer> list = new ArrayList<>(
  Arrays.asList(1, 3, 5, 7, 9, 11, 13, 15, 17, 19));
List<Integer> result = new ArrayList<>();
list.parallelStream()
  .filter(x -> isPrime(x))
  .forEach(x -> result.add(x)); //Changing external state, which may not (is not) thread safe
• Just adding more parallel() doesn't always make it faster! (see: law of leaky abstractions)
• There is some overhead to how a parallel operation occurs
• Internally, Java keeps a pool of worker threads (rather than make new threads for each parallel task)
• Streams use a special kind of pool, called a ForkJoinPool
Fork/Join Programming

- Special kind of task - `fork()` defines how to create subtasks, `join()` defines how to combine the results
- Similar to map/reduce, but not distributed
- For streams:
  - **Fork** a task into subtasks for many threads to work on
  - **Join** the results together
Fork/Join Programming

• Obligatory array sum example

```java
class Sum extends RecursiveTask<Long> {
    static final int SEQUENTIAL_THRESHOLD = 5000;
    int low;
    int high;
    int[] array;
    
    Sum(int[] arr, int lo, int hi) {
        array = arr;
        low   = lo;
        high  = hi;
    }
    
    protected Long compute() {
        if(high - low <= SEQUENTIAL_THRESHOLD) {
            long sum = 0;
            for(int i=low; i < high; ++i)
                sum += array[i];
            return sum;
        } else {
            int mid = low + (high - low) / 2;
            Sum left  = new Sum(array, low, mid);
            Sum right = new Sum(array, mid, high);
            left.fork();
            long rightAns = right.compute();
            long leftAns  = left.join();
            return leftAns + rightAns;
        }
    }
}

static long sumArray(int[] array) {
    return ForkJoinPool.commonPool().invoke(new Sum(array,0,array.length));
}
```
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 CompleteFuture in Java 8
    ```java
    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

```java
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
```

Create an asynchronous task
Chaining CompletableFuture

Task will return string “Jon” eventually

```java
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

```java
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 = whatsYourNameFuture.thenApply(returnValue -> {
    return 
    "Hello, " + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
Chaining CompletableFuture

```java
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
```

Block the main thread for both futures to finish
Chaining CompletableFutures

- We can chain asynchronous activities together with the `thenAccept` term

```java
Promise to get some data

then

Promise to make some other changes to that data

then

Promise to make some changes to that data

then

Report on those changes to the user

thenCombine

then

Report on the error

If there's an error...

If there's an error...```
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
CompletableFutures

• Catch errors by providing a callback function for exceptionally (called when an exception occurs in any of those threads)

• API: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html
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.