Principles of Software Construction: Objects, Design, and Concurrency

Part 4: Concurrency

Introduction to concurrency, part 4

Design patterns and frameworks for concurrency

Charlie Garrod Bogdan Vasilescu

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 5b due 11:59 p.m. tonight
  - Turn in by Wednesday 9 a.m. to be considered as a Best Framework
- Optional reading due today:
  - Java Concurrency in Practice, Chapter 12
- Your in-class feedback

Key concepts from last Thursday

Policies for thread safety

- Thread-confined
- Shared read-only
- Shared thread-safe
  - Objects that perform internal synchronization
- Guarded
  - Objects that must be synchronized externally

Summary of our `RwLock` example

- Generally, avoid `wait/notify`
- Never invoke `wait` outside a loop
  - Must check coordination condition after waking
- Generally use `notifyAll`, not `notify`
- Do not use our `RwLock` – it's just a toy
  - Instead, know the standard libraries…
    - Discuss: `sun.misc.Unsafe`

Concurrency bugs can be very subtle

```
private final List<Observer<E>> observers = new ArrayList<>();
public void addObserver(Observer<E> observer) {
   synchronized(observers) { observers.add(observer); }
}
public boolean removeObserver(Observer<E> observer) {
   synchronized(observers) { return observers.remove(observer); }
}
private void notifyOf(E element) {
   synchronized(observers) {
       for (Observer<E> observer : observers)
           observer.notify(this, element);  // Risks liveness and
   }                                        // safety failures!
}
```

Today

- Design patterns for concurrency
- The Java executors framework
- Concurrency in practice: In the trenches of parallelism

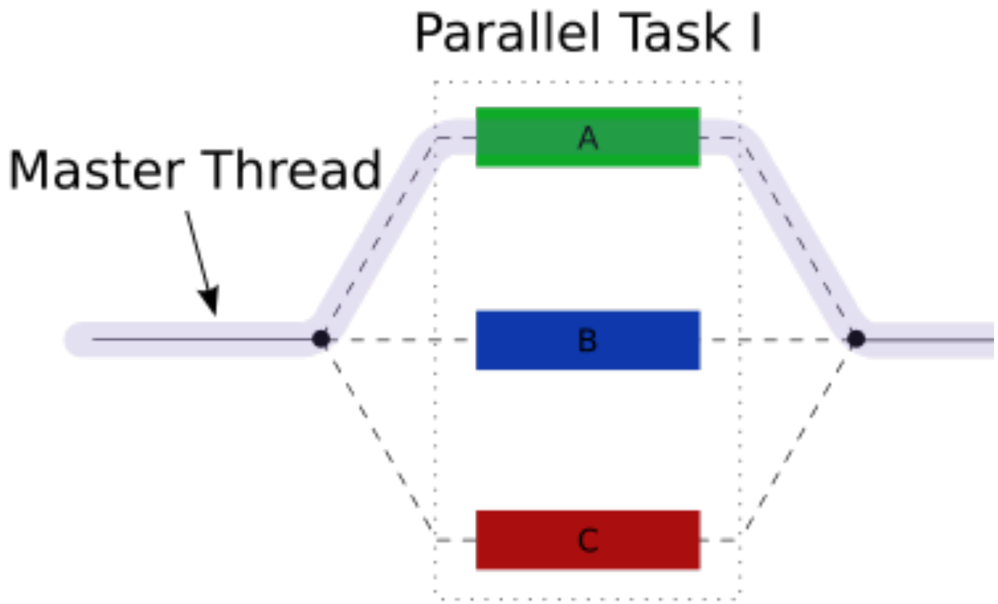
Producer-consumer design pattern

- Goal: Decouple the producer and the consumer of some data
- Consequences:
  - Removes code dependency between producers and consumers
  - Producers and consumers can produce and consume at different rates

`java.util.concurrent.BlockingQueue`

- Implements `java.util.Queue<E>`
- `java.util.concurrent.ArrayBlockingQueue`
  - `put` blocks if the queue is full
  - `poll` blocks if the queue is empty
- `java.util.concurrent.SynchronousQueue`
  - Each `put` directly waits for a corresponding `poll`

The fork-join pattern

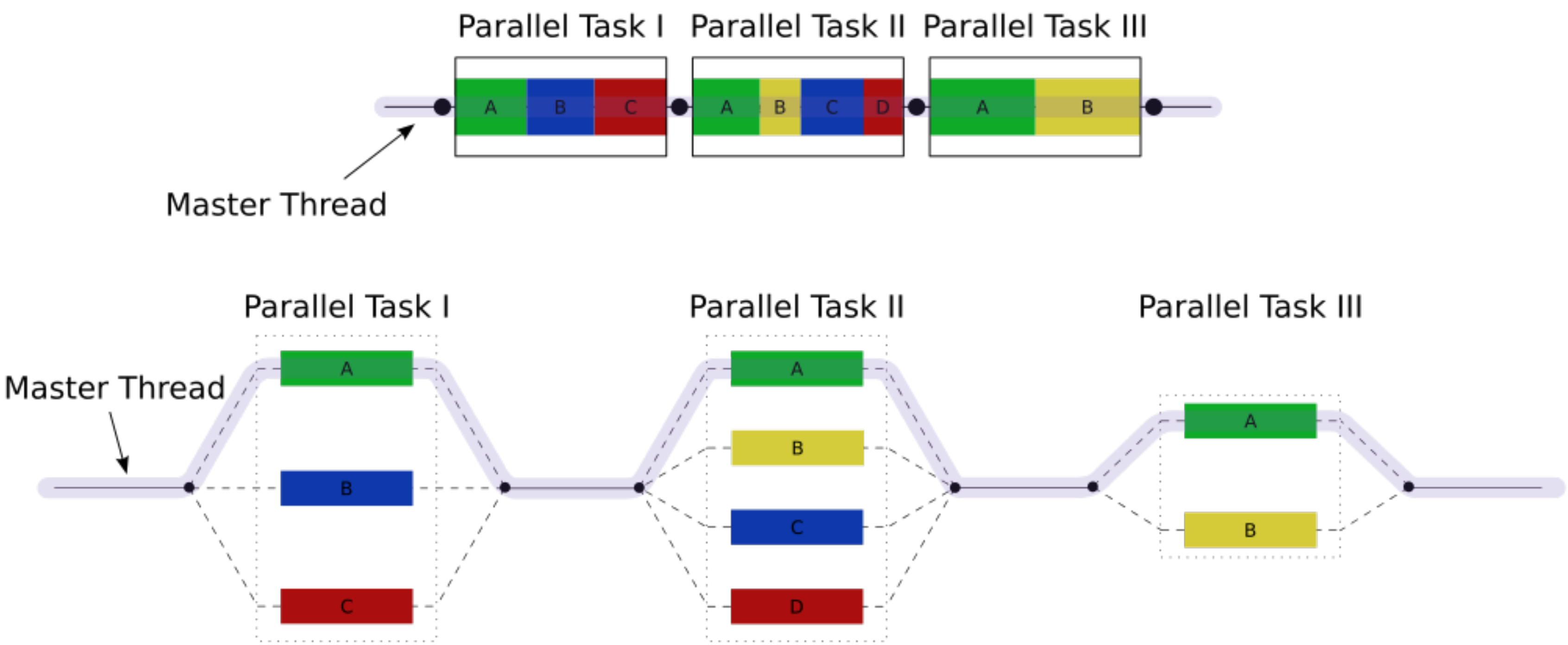



```
if (my portion of the work is small)
    do the work directly
else
    split my work into pieces
    invoke the pieces and wait for the results
```


Image from: Wikipedia

The membrane pattern

- Multiple rounds of fork-join, each round waiting for the previous round to complete



Today

- Design patterns for concurrency
- The Java executors framework
- Concurrency in practice: In the trenches of parallelism

Execution of tasks

- Natural boundaries of computation define tasks, e.g.:

```
public class SingleThreadWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }

    private static void handleRequest(Socket connection) {
        … // request-handling logic here
    }
}
```

A poor design choice: A thread per task

```
public class ThreadPerRequestWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            new Thread(() -> handleRequest(connection)).start();
        }
    }

    private static void handleRequest(Socket connection) {
        … // request-handling logic here
    }
}
```

Recall the Java primitive concurrency tools

- The `java.lang.Runnable` interface

```
void            run();
```

- The `java.lang.Thread` class

```
Thread(Runnable r);
void            start();
void            join();
```

Recall the Java primitive concurrency tools

- The `java.lang.Runnable` interface

```
void              run();
```

- The `java.lang.Thread` class

```
Thread(Runnable r);
void              start();
void              join();
```

- The `java.util.concurrent.Callable<V>` interface
  - Like `java.lang.Runnable` but can return a value

```
V                 call();
```

A framework for asynchronous computation

- The `java.util.concurrent.Future<V>` interface

```
V           get();
V           get(long timeout, TimeUnit unit);
boolean     isDone();
boolean     cancel(boolean mayInterruptIfRunning);
boolean     isCancelled();
```

A framework for asynchronous computation

- The `java.util.concurrent.Future<V>` interface:

```
V           get();
V           get(long timeout, TimeUnit unit);
boolean     isDone();
boolean     cancel(boolean mayInterruptIfRunning);
boolean     isCancelled();
```

- The `java.util.concurrent.ExecutorService` interface:

```
Future<?>  submit(Runnable task);
Future<V>  submit(Callable<V> task);
List<Future<V>>
      invokeAll(Collection<? extends Callable<V>> tasks);
Future<V>
      invokeAny(Collection<? extends Callable<V>> tasks);
void shutdown();
```

`Executors` for common computational patterns

- From the `java.util.concurrent.Executors` class

```
static ExecutorService newSingleThreadExecutor();
static ExecutorService newFixedThreadPool(int n);
static ExecutorService newCachedThreadPool();
static ExecutorService newScheduledThreadPool(int n);
```

Example use of executor service

```
public class ThreadPoolWebServer {
    private static final Executor exec
            = Executors.newFixedThreadPool(100); // 100 threads
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            exec.execute(() -> handleRequest(connection));
        }
    }

    private static void handleRequest(Socket connection) {
        … // request-handling logic here
    }
}
```

Today

- Design patterns for concurrency
- The Java executors framework
- Concurrency in practice: In the trenches of parallelism

Concurrency at the language level

- Consider:

```
Collection<Integer> collection = …;
int sum = 0;
for (int i : collection) {
    sum += i;
}
```

- In python:

```
collection = …
sum = 0
for item in collection:
    sum += item
```

Parallel quicksort in Nesl

```
function quicksort(a) =
  if (#a < 2) then a
  else
   let pivot   = a[#a/2];
       lesser  = {e in a| e < pivot};
       equal   = {e in a| e == pivot};
       greater = {e in a| e > pivot};
       result  = {quicksort(v): v in [lesser,greater]};
   in result[0] ++ equal ++ result[1];
```

- Operations in { } occur in parallel
- 210-esque questions: What is total work? What is depth?

Prefix sums (a.k.a. inclusive scan, a.k.a. scan)

- Goal: given array `x[0…n-1]`, compute array of the sum of each prefix of x

```
[ sum(x[0…0]),
  sum(x[0…1]),
  sum(x[0…2]),
  …
  sum(x[0…n-1]) ]
```

- e.g., `x =` `[13, 9, -4, 19, -6, 2, 6, 3]`

  prefix sums: `[13, 22, 18, 37, 31, 33, 39, 42]`

Parallel prefix sums

- Intuition: If we have already computed the partial sums `sum(x[0…3])` and `sum(x[4…7])`, then we can easily compute `sum(x[0…7])`
- e.g., `x =      [13,  9, -4, 19, -6,  2,  6,  3]`

Parallel prefix sums algorithm, upsweep

Compute the partial sums in a more useful manner

```
[13,    9,   -4,   19,   -6,    2,    6,    3]

[13,   22,   -4,   15,   -6,   -4,    6,    9]
```

Parallel prefix sums algorithm, upsweep

Compute the partial sums in a more useful manner

```
[13,    9,   -4,   19,   -6,    2,    6,    3]

[13,   22,   -4,   15,   -6,   -4,    6,    9]

[13,   22,   -4,   37,   -6,   -4,    6,    5]
```

Parallel prefix sums algorithm, upsweep

Compute the partial sums in a more useful manner

```
[13,  9, -4, 19, -6,  2, 6,  3]
[13, 22, -4, 15, -6, -4, 6,  9]
[13, 22, -4, 37, -6, -4, 6,  5]
[13, 22, -4, 37, -6, -4, 6, 42]
```

Parallel prefix sums algorithm, downsweep

Now unwind to calculate the other sums

```
[13,   22,   -4,   37,   -6,   -4,    6,   42]

[13,   22,   -4,   37,   -6,   33,    6,   42]
```

Parallel prefix sums algorithm, downsweep

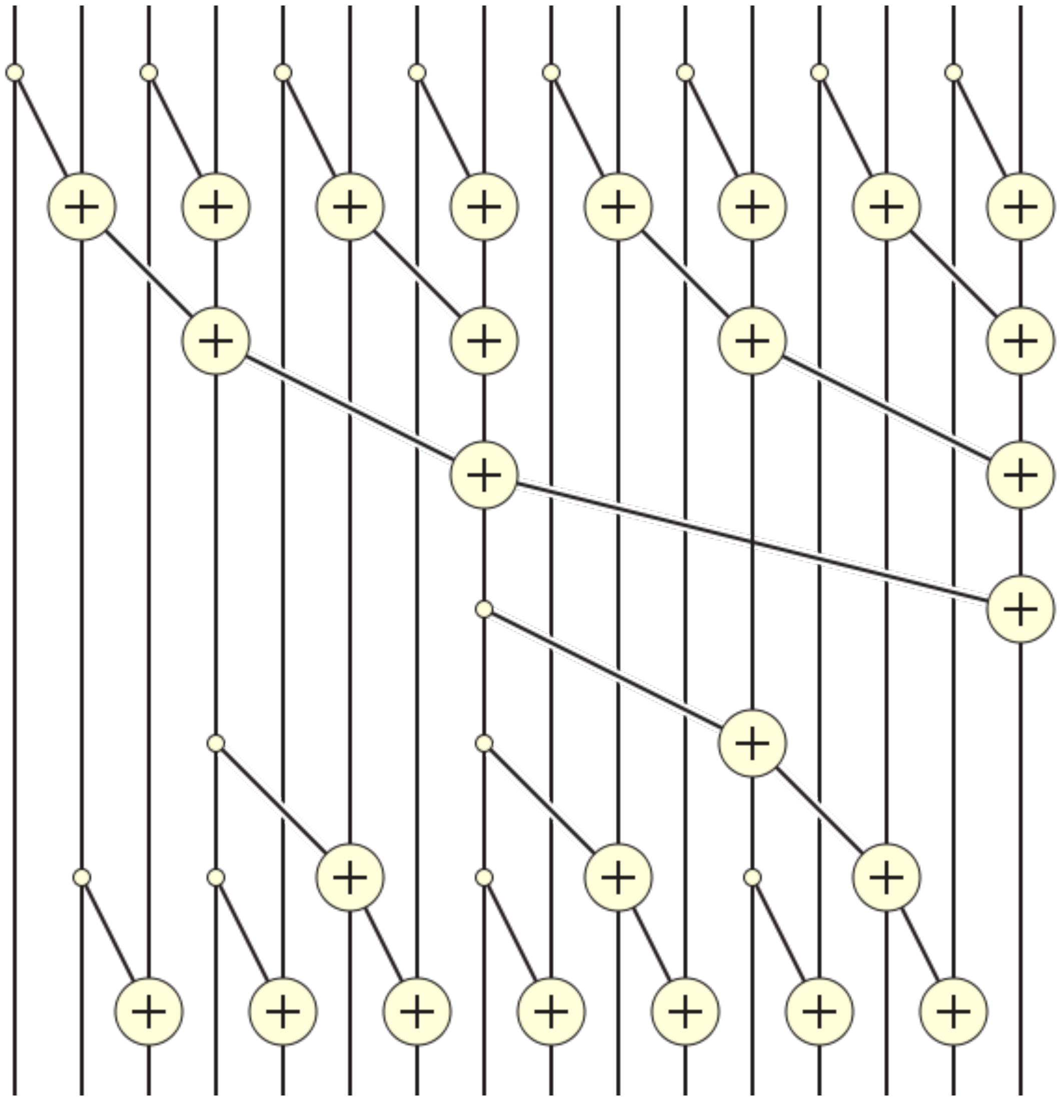
Now unwind to calculate the other sums

```
[13,   22,   -4,   37,   -6,   -4,    6,   42]

[13,   22,   -4,   37,   -6,   33,    6,   42]

[13,   22,   18,   37,   31,   33,   39,   42]
```

- Recall, we started with:

```
[13,    9,   -4,   19,   -6,    2,    6,    3]
```

Doubling array size adds two more levels



Upsweep

Downsweep

Parallel prefix sums

pseudocode

```
// Upsweep
prefix_sums(x):
  for d in 0 to (lg n)-1:           // d is depth
    parallelfor i in 2^d-1 to n-1, by 2^(d+1):
      x[i+2^d] = x[i] + x[i+2^d]

// Downsweep
for d in (lg n)-1 to 0:
  parallelfor i in 2^d-1 to n-1-2^d, by 2^(d+1):
    if (i-2^d >= 0):
      x[i] = x[i] + x[i-2^d]
```

Parallel prefix sums algorithm, in code

- An iterative Java-esque implementation:

```
void iterativePrefixSums(long[] a) {
  int gap = 1;
  for ( ; gap < a.length; gap *= 2) {
    parfor(int i=gap-1; i+gap < a.length; i += 2*gap) {
      a[i+gap] = a[i] + a[i+gap];
    }
  }
  for ( ; gap > 0; gap /= 2) {
    parfor(int i=gap-1; i < a.length; i += 2*gap) {
      a[i] = a[i] + ((i-gap >= 0) ? a[i-gap] : 0);
    }
  }
```

Parallel prefix sums algorithm, in code

- A recursive Java-esque implementation:

```
void recursivePrefixSums(long[] a, int gap) {
  if (2*gap – 1 >= a.length) {
    return;
  }

  parfor(int i=gap-1; i+gap < a.length; i += 2*gap) {
    a[i+gap] = a[i] + a[i+gap];
  }

  recursivePrefixSums(a, gap*2);

  parfor(int i=gap-1; i < a.length; i += 2*gap) {
    a[i] = a[i] + ((i-gap >= 0) ? a[i-gap] : 0);
  }
}
```

Parallel prefix sums algorithm

- How good is this?

Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See `PrefixSums.java`, `PrefixSumsSequentialWithParallelWork.java`

Goal: parallelize the PrefixSums implementation

- Specifically, parallelize the parallelizable loops

```
parfor(int i = gap-1;  i+gap < a.length;  i += 2*gap) {
  a[i+gap] = a[i] + a[i+gap];
}
```

- Partition into multiple segments, run in different threads

```
for(int i = left+gap-1;  i+gap < right;  i += 2*gap) {
  a[i+gap] = a[i] + a[i+gap];
}
```

Recall: The membrane pattern

- Multiple rounds of fork-join, each round waiting for the previous round to complete

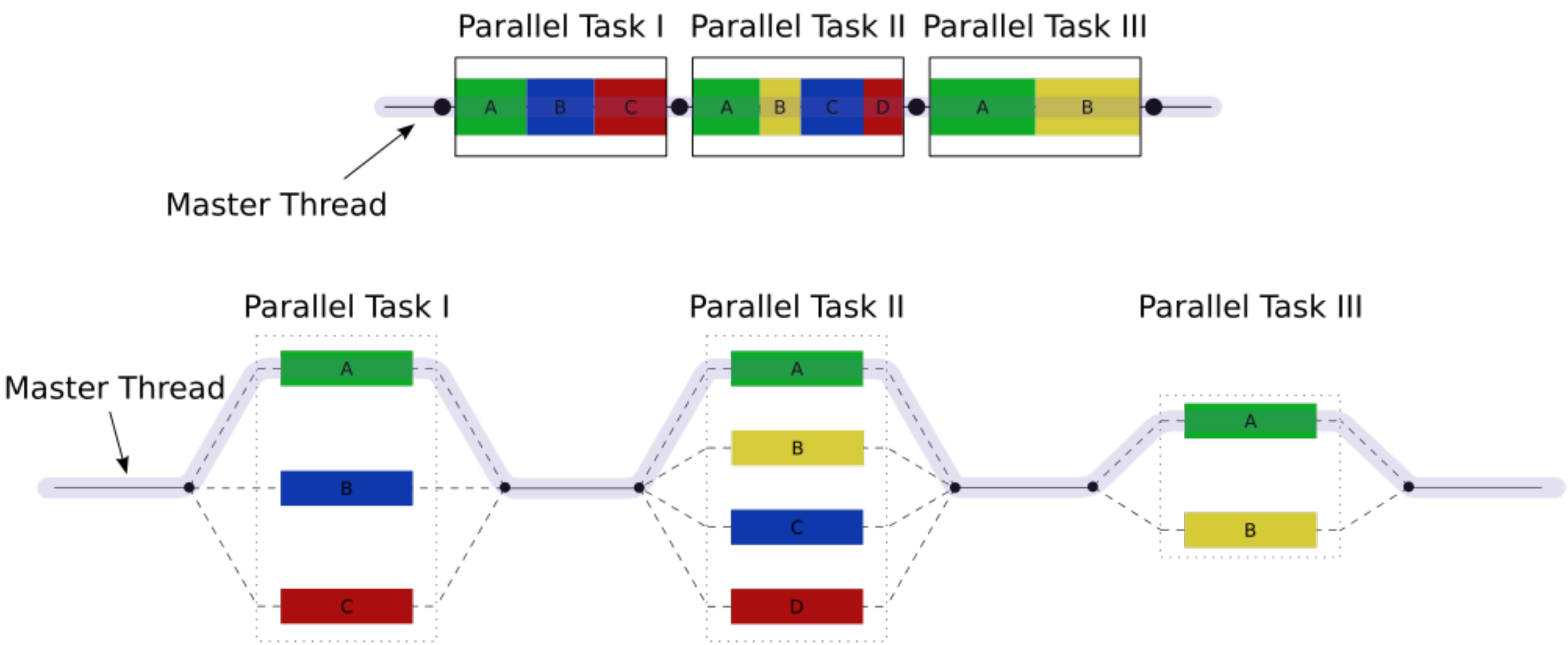


Image from: Wikipedia

Fork/join in Java

- The `java.util.concurrent.ForkJoinPool` class
  - Implements `ExecutorService`
  - Executes `java.util.concurrent.ForkJoinTask<V>` or
    `java.util.concurrent.RecursiveTask<V>` or
    `java.util.concurrent.RecursiveAction`
- In a long computation:
  - Fork a thread (or more) to do some work
  - Join the thread(s) to obtain the result of the work

The `RecursiveAction` abstract class

```
public class MyActionFoo extends RecursiveAction {
    public MyActionFoo(…) {
        store the data fields we need
    }

    @Override
    public void compute() {
        if (the task is small) {
            do the work here;
            return;
        }

        invokeAll(new MyActionFoo(…),  // smaller
                  new MyActionFoo(…),  // subtasks
                  …);                  // …
    }
}
```

A ForkJoin example

- See `PrefixSumsParallelForkJoin.java`
- See the processor go, go go!

Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See `PrefixSumsParallelArrays.java`

Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See `PrefixSumsParallelArrays.java`
- See `PrefixSumsSequential.java`

Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See `PrefixSumsParallelArrays.java`
- See `PrefixSumsSequential.java`
  - n-1 additions
  - Memory access is sequential
- For `PrefixSumsSequentialWithParallelWork.java`
  - About 2n useful additions, plus extra additions for the loop indexes
  - Memory access is non-sequential
- The punchline:
  - Don't roll your own
  - Cache and constants matter

In-class example for parallel prefix sums

```
[7,     5,     8,  -36,    17,     2,    21,    18]
```