Principles of Software Construction

Concurrency, part 4: In the trenches of parallelism

Josh Bloch           Charlie Garrod
Administrivia

• Homework 5b due tonight
  – Commit by 9 a.m. tomorrow to be considered as a Best Framework
• Still a few midterm 2 exams remain to be picked up
Key concepts from Thursday

• `java.util.concurrent` is the best, easiest way to write concurrent code
• It’s big, but well designed and engineered
  – Easy to do simple things
  – Possible to do complex things
• Executor framework does for execution what Collections framework did for aggregation
java.util.concurrent Summary (1/2)

I. Atomic vars - `java.util.concurrent.atomic`
   – Support various atomic read-modify-write ops

II. Executor framework
   – Tasks, futures, thread pools, completion service, etc.

III. Locks - `java.util.concurrent.locks`
   – Read-write locks, conditions, etc.

IV. Synchronizers
   – Semaphores, cyclic barriers, countdown latches, etc.
V. Concurrent collections
   – Shared maps, sets, lists

VI. Data Exchange Collections
   – Blocking queues, deques, etc.

VII. Pre-packaged functionality - java.util.arrays
   – Parallel sort, parallel prefix
Puzzler: “Racy Little Number”

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class LittleTest {
    int number;

    @Test
    public void test() throws InterruptedException {
        number = 0;
        Thread t = new Thread(() -> {
            assertEquals(2, number);
        });
        number = 1;
        t.start();
        number++;
        t.join();
    }
}
How often does this test pass?

```java
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class LittleTest {
    int number;

    @Test
    public void test() throws InterruptedException {
        number = 0;
        Thread t = new Thread(() -> {
            assertEquals(2, number);
        });
        number = 1;
        t.start();
        number++;  // This line is extra and not part of the thread's execution.
        t.join();
    }
}
```

(a) It always fails
(b) It sometimes passes
(c) It always passes
(d) It always hangs
How often does this test pass?

(a) It always fails
(b) It sometimes passes
(c) It always passes – but it tells us nothing
(d) It always hangs

JUnit doesn’t see assertion failures in other threads
import org.junit.*;
import static org.junit.Assert.*;

public class LittleTest {
    int number;

    @Test
    public void test() throws InterruptedException {
        number = 0;
        Thread t = new Thread(() -> {
            assertEquals(2, number); // JUnit never sees the exception!
        });
        number = 1;
        t.start();
        number++;
        t.join();
    }
}
How do you fix it? (1)

// Keep track of assertion failures during test
volatile Exception exception;
volatile Error error;

// Triggers test case failure if any thread asserts failed
@After
public void tearDown() throws Exception {
    if (error != null)
        throw error;
    if (exception != null)
        throw exception;
}
How do you fix it? (2)

Thread t = new Thread(() -> {
    try {
        assertEquals(2, number);
    } catch(Error e) {
        error = e;
    } catch(Exception e) {
        exception = e;
    }
});

*YMMV (It’s a race condition)

Now it sometimes passes*
The moral

- JUnit does not support concurrency
- You must provide your own
  - If you don’t, you’ll get a false sense of security
Puzzler: “Ping Pong”

public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread(() -> pong());
        t.run();
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
What does it print?

```java
public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread(() -> pong());
        t.run();
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
```

(a) PingPong  
(b) PongPing  
(c) It varies
What does it print?

(a) PingPong
(b) PongPing
(c) It varies

Not a multithreaded program!
Another look

public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread(() -> pong());
        t.run(); // An easy typo!
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
How do you fix it?

```java
public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread(() -> pong());
        t.start();
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
```

Now prints PingPong
The moral

• Invoke `Thread.start`, not `Thread.run`
  – Can be very difficult to diagnose

• `java.lang.Thread` should not have implemented `Runnable`
  – ...and should not have a public `run` method
Today: In the trenches of parallelism

- A high-level view of parallelism
- Concurrent realities
  - ...and java.util.concurrent
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 \( x \)

\[
\begin{align*}
\text{sum}(x[0…0]), \\
\text{sum}(x[0…1]), \\
\text{sum}(x[0…2]), \\
\ldots \\
\text{sum}(x[0…n-1]) 
\end{align*}
\]

• 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 $\text{sum}(x[0...3])$ and $\text{sum}(x[4...7])$, then we can easily compute $\text{sum}(x[0...7])$

• e.g., $x = [13, 9, -4, 19, -6, 2, 6, 3]$
Parallel prefix sums algorithm, \textit{upsweep}

Compute the partial sums in a more useful manner

\[
\begin{bmatrix}
13, & 9, & -4, & 19, & -6, & 2, & 6, & 3 \\
13, & 22, & -4, & 15, & -6, & -4, & 6, & 9 \\
\end{bmatrix}
\]
Parallel prefix sums algorithm, *upsweep*

Compute the partial sums in a more useful manner:

\[
\begin{align*}
[13, & 9, -4, 19, -6, 2, 6, 3] \\
[13, & 22, -4, 15, -6, -4, 6, 9] \\
[13, & 22, -4, 37, -6, -4, 6, 5]
\end{align*}
\]
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, \textit{downsweep}

Now unwind to calculate the other sums

\[
\begin{bmatrix}
13, & 22, & -4, & 37, & -6, & -4, & 6, & 42 \\
13, & 22, & -4, & 37, & -6, & 33, & 6, & 42
\end{bmatrix}
\]
Parallel prefix sums algorithm, **downsweep**

- Now unwinds to calculate the other sums

\[
\begin{array}{cccccccc}
13 & 22 & -4 & 37 & -6 & -4 & 6 & 42 \\
13 & 22 & -4 & 37 & -6 & 33 & 6 & 42 \\
13 & 22 & 18 & 37 & 31 & 33 & 39 & 42 \\
\end{array}
\]

- Recall, we started with:

\[
\begin{array}{cccccccc}
13 & 9 & -4 & 19 & -6 & 2 & 6 & 3 \\
\end{array}
\]
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
    parallel
      for 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:
  parallel
    for 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:

```java
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:
  ```java
  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
  ```java
  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
  ```java
  for(int i = left+gap-1; i+gap < right; i += 2*gap) {
    a[i+gap] = a[i] + a[i+gap];
  }
  ```
Recall the Java primitive concurrency tools

- The `java.lang.Runnable` interface
  ```java
  void run();
  ```

- The `java.lang.Thread` class
  ```java
  Thread(Runnable r);
  void start();
  static void sleep(long millis);
  void join();
  boolean isAlive();
  static Thread currentThread();
  ```
Recall the Java primitive concurrency tools

- The `java.lang.Runnable` interface
  ```java
  void run();
  ```

- The `java.lang.Thread` class
  ```java
  Thread(Runnable r);
  void start();
  static void sleep(long millis);
  void join();
  boolean isAlive();
  static Thread currentThread();
  ```

- The `java.util.concurrent.Callable<V>` interface
  - Like `java.lang.Runnable` but can return a value
  ```java
  V call();
  ```
A framework for asynchronous computation

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

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

• The `java.util.concurrent.ExecutorService` interface:
  
  ```java
  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);
Fork/Join: another common computational pattern

• In a long computation:
  – Fork a thread (or more) to do some work
  – Join the thread(s) to obtain the result of the work
Fork/Join: another common computational pattern

• 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 `java.util.concurrent.ForkJoinPool` class
  – Implements `ExecutorService`
  – Executes `java.util.concurrent.ForkJoinTask<V>` or `java.util.concurrent.RecursiveTask<V>` or `java.util.concurrent.RecursiveAction`
The RecursiveAction abstract class

```java
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(...),  // tasks
                  ...);  // ...
    }
}
```
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
Coming Thursday...

- Distributed systems (MapReduce?)
In-class example for parallel prefix sums

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