Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Concurrency

Introduction to concurrency, part 4
*In the trenches of parallelism*

Charlie Garrod    Chris Timperley
Administrivia

• Homework 5b due 11:59 p.m. tonight
  – Turn in by tomorrow 9 a.m. to be considered as a Best Framework

• Optional reading due today:
  – Java Concurrency in Practice, Chapter 12
NEW COURSE: LANGUAGE DESIGN & PROTOTYPING
17-396/17-696 – SPRING 2020

Little languages are everywhere! Would you like to – or do you need to – design your own?

In this course, you will:
• Learn how to critique a language design
• Practice several language prototyping approaches (interpreters, transpilers, fluent APIs)
• Apply techniques for evaluating language designs with users
• Design and prototype your own language in the final project

Prof. Jonathan Aldrich – T/Th 3-4:20
http://www.cs.cmu.edu/~aldrich/courses/17-396/
Software Engineering (SE) at CMU

• 17-214: Code-level design
  – Extensibility, reuse, concurrency, functional correctness
• 17-313: Human aspects of software development
  – Requirements, teamwork, scalability, security, scheduling, costs, risks, business models
• 17-413 Practicum, 17-415 Seminar, Internship
• Various courses on requirements, architecture, software analysis, SE for startups, API design, etc.
• SE Minor: [http://isri.cmu.edu/education/undergrad](http://isri.cmu.edu/education/undergrad)
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
Shared thread-safe

• "Thread-safe" objects that perform internal synchronization
• Build your own, or know the Java concurrency libraries
Advice for building thread-safe objects

• Do as little as possible in synchronized region: get in, get out
  – Obtain lock
  – Examine shared data
  – Transform as necessary
  – Drop the lock

• If you must do something slow, move it outside the synchronized region
Example: adding concurrency to the observer pattern

```java
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!
    }
```

One solution: *snapshot iteration*

```java
private void notifyOf(E element) {
    List<Observer<E>> snapshot = null;

    synchronized(Observers) {
        snapshot = new ArrayList<>(observers);
    }

    for (Observer<E> observer : snapshot) {
        observer.notify(this, element); // Safe
    }
}
```
private final List<Observer<E>> observers =
    new CopyOnWriteArrayList<>();

public void addObserver(Observer<E> observer) {
    observers.add(observer);
}

public boolean removeObserver(Observer<E> observer) {
    return observers.remove(observer);
}

private void notifyOf(E element) {
    for (Observer<E> observer : observers)
        observer.notify(this, element);
}
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
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
  void execute(Runnable task);
  Future submit(Runnable task);
  Future<V> submit(Callable<V> task);
  List<Future<V>> invokeAll(Collection<Callable<V>> tasks);
  Future<V> invokeAny(Collection<Callable<V>> tasks);
  ```
Today

• Concurrency in practice: In the trenches of parallelism
Concurrency at the language level

• Consider:
  ```java
  Collection<Integer> collection = ...;
  int sum = 0;
  for (int i : collection) {
    sum += i;
  }
  ```

• In python:
  ```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 $x$

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

• 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: Partial sums can be efficiently combined to form much larger partial sums. E.g., if we know \( \text{sum}(x[0\ldots3]) \) and \( \text{sum}(x[4\ldots7]) \), then we can easily compute \( \text{sum}(x[0\ldots7]) \)

• 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

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

\[
\begin{align*}
[ &13, &9, &-4, &19, &-6, &2, &6, &3 ] \\
[ &13, &22, &-4, &15, &-6, &-4, &6, &9 ] \\
[ &13, &22, &-4, &37, &-6, &-4, &6, &5 ] \\
[ &13, &22, &-4, &37, &-6, &-4, &6, &42 ]
\end{align*}
\]
Parallel prefix sums algorithm, **downsweep**

Now unwind to calculate the other sums

\[
[ 13, \quad 22, \quad -4, \quad 37, \quad -6, \quad -4, \quad 6, \quad 42 ]
\]

\[
[ 13, \quad 22, \quad -4, \quad 37, \quad -6, \quad 33, \quad 6, \quad 42 ]
\]
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
        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 from Thursday: 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

```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(...), // 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: $O(\log n)$

- 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. Know the libraries
  – Cache and constants matter
In-class example for parallel prefix sums

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