Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Concurrency

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

Josh Bloch  Charlie Garrod
Administrivia

• Homework 5 Best Frameworks available today
• Homework 5c due Tuesday, 11:59 p.m.
Key concepts from Tuesday
Policies for thread safety

1. **Thread-confined state** – mutate but don’t share
2. **Shared read-only state** – share but don’t mutate
3. **Shared thread-safe** – object synchronizes itself internally
4. **Shared guarded** – client synchronizes object(s) externally
3. Shared thread-safe state

- Thread-safe objects that perform internal synchronization
- You can build your own, but **not for the faint of heart**
- You’re better off using ones from `java.util.concurrent`
- `j.u.c` also provides skeletal implementations
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**
The fork-join pattern

```java
if (my portion of the work is small)
do the work directly
else
    split my work into pieces
    recursively process the pieces
```
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 span?
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 \( x \)
  
  \[
  \begin{array}{l}
  \text{sum}(x[0...0]), \\
  \text{sum}(x[0...1]), \\
  \text{sum}(x[0...2]), \\
  \vdots \\
  \text{sum}(x[0...n-1])
  \end{array}
  \]

• 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 $\text{sum}(x[0...3])$ 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, *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

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

Now unwind to calculate the other sums

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

Now unwind to calculate the other sums

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

- Recall, we started with:

\[
\begin{bmatrix}
13, & 9, & -4, & 19, & -6, & 2, & 6, & 3 \\
\end{bmatrix}
\]
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: $O(n)$
  – Span: $O(\lg n)$

• 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 the previous lecture: 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: $O(n)$
  – Span: $O(\lg n)$

• See PrefixSumsParallelArrays.java
Parallel prefix sums algorithm

- How good is this?
  - Work: $O(n)$
  - Span: $O(\log n)$

- See PrefixSumsParallelArrays.java
- See PrefixSumsSequential.java
Parallel prefix sums algorithm

- How good is this?
  - Work: $O(n)$
  - Span: $O(\lg n)$

- 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