Principles of Software Construction: Objects, Design and Concurrency

The Perils of Concurrency, Part 4
(Can't live with it, can't live without it.)

Christian Kästner
Charlie Garrod
Administrivia

- Homework 5: The Framework Strikes Back
  - 5a presentations yesterday!
Key topics from Tuesday
Today: In the trenches of parallelism

- A concurrent implementation of prefix-sums
Today: In the trenches of parallelism

- A concurrent implementation of prefix-sums
- Also: Java I/O fundamentals
Recall: work, breadth, and depth

- **Work**: total effort required
  - area of the shape

- **Breadth**: extent of simultaneous activity
  - width of the shape

- **Depth (or span)**: length of longest computation
  - height of the shape
Prefix sums (a.k.a. inclusive scan)

- **Goal:** given array $x[0...n-1]$, compute array of the sum of each prefix of $x$

$$[\text{sum}(x[0...0]), \text{sum}(x[0...1]), \text{sum}(x[0...2]), \ldots, \text{sum}(x[0...n-1])]$$

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

- **Code:**

  
  ```python
  def prefix_sums(x):
      for d in 0 to (\log n) - 1:  // d is depth
          parallel
          for i in 2^d to n-1:
              newx[i] = x[i-2^d] + x[i]
          x = newx
  ```

- **How good is this?**
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 \( \text{sum}(x[0...7]) \).

• **Code:**

```python
prefix_sums(x):
    for d in 0 to (\lg n)-1:
        // d is depth
        parallel for i in 2^d to n-1:
            newx[i] = x[i-2^d] + x[i]
        x = newx
```

• **How good is this?**
  - **Work:** \( O(n \lg n) \)
  - **Depth** \( O(\lg n) \)
A better parallel prefix sums algorithm

• 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 \( \text{sum}(x[0...7]) \)

• Code:

```python
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]

  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]
```

• e.g., \( x = [13, 9, -4, 19, -6, 2, 6, 3] \)
A better parallel prefix sums algorithm, part 1

- Computes the partial sums in a more useful manner

\[ [13, \quad 9, \quad -4, \quad 19, \quad -6, \quad 2, \quad 6, \quad 3] \]

\[ [13, \quad 22, \quad -4, \quad 15, \quad -6, \quad -4, \quad 6, \quad 9] \]
A better parallel prefix sums algorithm, part 1

- Computes the partial sums in a more useful manner

\[
\begin{align*}
[13, & 9, -4, 19, -6, 2, 6, 3] \\
[13, & 22, -4, 15, -6, -4, 6, 9] \\
[13, & 22, -4, 37, -6, -4, 6, 5]
\end{align*}
\]
A better parallel prefix sums algorithm, part 1

- Computes the partial sums in a more useful manner

\[
\begin{array}{cccccccc}
13 & 9 & -4 & 19 & -6 & 2 & 6 & 3 \\
13 & 22 & -4 & 15 & -6 & -4 & 6 & 9 \\
13 & 22 & -4 & 37 & -6 & -4 & 6 & 5 \\
13 & 22 & -4 & 37 & -6 & -4 & 6 & 42 \\
\end{array}
\]
A better parallel prefix sums algorithm, part 2

- Now unwinds to calculate the other sums

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

\[ [13, 22, -4, 37, -6, 33, 6, 42] \]
• Now unwinds 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:

\[ [ 13, \ 9, \ -4, \ 19, \ -6, \ 2, \ 6, \ 3 ] \]
A better parallel prefix sums algorithm, in code

- **An iterative Java-esque implementation:**

```java
void computePrefixSums(long[] a) {
    for (int gap = 1; 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 (int gap = a.length/2; gap > 0; gap /= 2) {
        parfor(int i=gap-1; i+gap<a.length; i += 2*gap) {
            a[i] = a[i] + ((i-gap >= 0) ? a[i-gap] : 0);
        }
    }
}
```
A better parallel prefix sums algorithm, in code

- **A recursive Java-esque implementation:**

  ```java
  void computePrefixSumsRecursive(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];
    }

    computePrefixSumsRecursive(a, gap*2);

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

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

- See Main.java, PrefixSumsNonSequentialImpl.java
Goal: parallelize PrefixSumsNonSequentialImpl

- Specifically, parallelize the parallelizable loops
  \[
  \text{parfor}(\text{int } i=\text{gap}-1; \ i+\text{gap}<\text{a.length}; \ i += 2*\text{gap}) \{ \\
  \hspace{1em} \text{a}[i+\text{gap}] = \text{a}[i] + \text{a}[i+\text{gap}]; \\
  \}
  \]

- Partition into multiple segments, run in different threads
  \[
  \text{for}(\text{int } i=\text{left}+\text{gap}-1; \ i+\text{gap}<\text{right}; \ i += 2*\text{gap}) \{ \\
  \hspace{1em} \text{a}[i+\text{gap}] = \text{a}[i] + \text{a}[i+\text{gap}]; \\
  \}
  \]
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();
  ```

• The `java.util.concurrent.ExecutorService` interface
  
  ```java
  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);
  ```
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);

• See NetworkServer.java
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 PrefixSumsParallelImpl.java, PrefixSumsParallelLoop1.java, and PrefixSumsParallelLoop2.java

• See the processor go, go go!
A better parallel prefix sums algorithm

• How good is this?
  ▪ Work: $O(n)$
  ▪ Depth: $O(\log n)$

• See PrefixSumsSequentialImpl.java
A better parallel prefix sums algorithm

• How good is this?
  ▪ Work: O(n)
  ▪ Depth: O(lg n)

• See PrefixSumsSequentialImpl.java
  ▪ n-1 additions
  ▪ Memory access is sequential

• For PrefixSumsNonsequentialImpl.java
  ▪ About 2n useful additions, plus extra additions for the loop indexes
  ▪ Memory access is non-sequential

• The punchline: Constants matter.
Today: In the trenches of parallelism

- A concurrent implementation of prefix-sums
- Also: Java I/O fundamentals
System.out is a java.io.PrintStream

- java.io.PrintStream: Allows you to conveniently print common types of data
  - void close();
  - void flush();
  - void print(String s);
  - void print(int i);
  - void print(boolean b);
  - void print(Object o);
  ...
  - void println(String s);
  - void println(int i);
  - void println(boolean b);
  - void println(Object o);
  ...

The fundamental I/O abstraction: a stream of data

- **java.io.InputStream**
  ```java
  void close();
  abstract int read();
  int read(byte[] b);
  ```

- **java.io.OutputStream**
  ```java
  void close();
  void flush();
  abstract void write(int b);
  void write(byte[] b);
  ```

- **Aside: If you have an OutputStream you can construct a PrintStream:**
  ```java
  PrintStream(OutputStream out);
  PrintStream(File file);
  PrintStream(String filename);
  ```
We typically want structured input, too

- **e.g., java.util.Scanner**
  - Scanner(InputStream source);
  - Scanner(File source);
  - void close();
  - boolean hasNextInt();
  - int nextInt();
  - boolean hasNextDouble();
  - double nextDouble();
  - boolean hasNextLine();
  - String nextLine();
  - boolean hasNext(Pattern p);
  - String next(Pattern p);
  - ...
See the FileExample.java demo

- Note the output format
To read and write arbitrary objects

- Your object must implement the `java.io.Serializable` interface
  - Methods: none!
  - If all of your data fields are themselves `Serializable`, Java can automatically serialize your class
    - If not, will get runtime `NotSerializableException`

- See `QABean.java` and `FileObjectExample.java`
Distributed systems

- Multiple system components (computers) communicating via some medium (the network)

- Challenges:
  - Heterogeneity
  - Scale
  - Geography
  - Security
  - Concurrency
  - Failures

(courtesy of http://www.cs.cmu.edu/~dga/15-440/F12/lectures/02-internet1.pdf)
Communication protocols

- Agreement between parties for how communication should take place
  - e.g., buying an airline ticket through a travel agent

(courtesy of http://www.cs.cmu.edu/~dga/15-440/F12/lectures/02-internet1.pdf)
Abstractions of a network connection

HTML | Text | JPG | GIF | PDF | ...
HTTP | FTP | ...
TCP | UDP | ...
IP

data link layer

physical layer
Packet-oriented and stream-oriented connections

- **UDP**: User Datagram Protocol
  - Unreliable, discrete packets of data

- **TCP**: Transmission Control Protocol
  - Reliable data stream
Internet addresses and sockets

- For IP version 4 (IPv4) host address is a 4-byte number
  - e.g. 127.0.0.1
  - Hostnames mapped to host IP addresses via DNS
  - ~4 billion distinct addresses

- Port is a 16-bit number (0-65535)
  - e.g. 80
  - Assigned conventionally

- In Java:
  - `java.net.InetAddress`
  - `java.net.Inet4Address`
  - `java.net.Inet6Address`
  - `java.net.Socket`
  - `java.net.InetSocketAddress`
Networking in Java

• The java.net.InetAddress:
  static InetAddress getByName(String host);
  static InetAddress getByAddress(byte[] b);
  static InetAddress getLocalHost();

• The java.net.Socket:
  Socket(InetAddress addr, int port);
  boolean isConnected();
  boolean isClosed();
  void close();
  InputStream getInputStream();
  OutputStream getOutputStream();

• The java.net.ServerSocket:
  ServerSocket(int port);
  Socket accept();
  void close();
...
A simple Sockets demo

- TextSocketClient.java
- TextSocketServer.java
- TransferThread.java
What do you want to do with your distributed system today?
Higher levels of abstraction

• Application-level communication protocols

• Frameworks for simple distributed computation
  • Remote Procedure Call (RPC)
  • Today: Java Remote Method Invocation (RMI)

• Complex computational frameworks
  • e.g., distributed map-reduce
Next week:

- Distributed systems