Principles of Software Construction: Objects, Design, and Concurrency

Part 6: Concurrency and distributed systems

The Perils of Concurrency

Can't live with it...
Can't live without it...

Jonathan Aldrich   Charlie Garrod
Administrivia

- 2\textsuperscript{nd} midterm exam Thursday
  - Exam review session tonight, 7:30 p.m. in DH 1112
- Homework 5 framework design advice
Key concepts from last Thursday
API: Application Programming Interface

- An API defines the boundary between components/modules in a programmatic system
An API design process

• Define the scope of the API
  – Collect use-case stories, define requirements
  – Be skeptical
    • Distinguish true requirements from so-called solutions
    • "When in doubt, leave it out."

• Draft a specification, gather feedback, revise, and repeat
  – Keep it simple, short

• Code early, code often
  – Write *client code* before you implement the API
Key design principle: Information hiding

- "When in doubt, leave it out."
Minimize mutability

- Immutable objects are:
  - Inherently thread-safe
  - Freely shared without concern for side effects
  - Convenient building blocks for other objects
  - Can share internal implementation among instances
    - See java.lang.String

- Mutable objects require careful management of visibility and side effects
  - e.g. Component getSize() returns a mutable Dimension

- Document mutability
  - Carefully describe state space
Use consistent parameter ordering

- An egregious example from C:
  ```c
  char* strncpy(char* dest, char* src, size_t n);
  void bcopy(void* src, void* dest, size_t n);
  ```
Avoid long lists of parameters

• Especially avoid parameter lists with repeated parameters of the same type

  HWND CreateWindow(LPCSTR lpClassName, LPCSTR lpWindowName, 
          DWORD dwStyle, int x, int y, int nWidth, int nHeight, 
          HWND hWndParent, HMENU hMenu, HINSTANCE hInstance, 
          LPVOID lpParam);

• Break up the method or use a helper class to hold parameters instead
Fail fast

- Report errors as soon as they are detectable
  - Check preconditions at the beginning of each method
  - Avoid dynamic type casts, run-time type-checking

```java
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);

    // Throws ClassCastException if this instance
    // contains any keys or values that are not Strings
    public void save(OutputStream out, String comments);
}
```
Avoid behavior that demands special processing

- Do not return `null` to indicate an empty value
  - e.g., Use an empty Collection or array instead
- Do not return `null` to indicate an error
  - Use an exception instead
- Do not return a `String` if a better type exists
- Do not use exceptions for normal behavior
- Avoid checked exceptions if possible

```java
try {
    Foo f = (Foo) g.clone();
} catch (CloneNotSupportedException e) {
    // Do nothing. This exception can't happen.
}
```
Don't let your output become your de facto API

- Document the fact that output formats may evolve in the future
- Provide programmatic access to all data available in string form

```java
public class Throwable {
    public void printStackTrace(PrintStream s);
}
```

```
at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
at com.ibm.rmi.io.ValueHandlerImpl.readValue(Array(ValueHandlerImpl.java:625)
at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:273)
at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:189)
at com.ibm.rmi.io.CDRInputStream.readValue(CDRInputStream.java:1429)
at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_Tie.java)
at com.ibm.CORBA.iioip.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.java:515)
at com.ibm.CORBA.orb.ORB.process(ORB.java:2377)
at com.ibm.CORBA.orb.OrbWorker.run(OrbWorker.java:186)
at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)
at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```
Don't let your output become your de facto API

- Document the fact that output formats may evolve in the future
- Provide programmatic access to all data available in string form

```java
public class Throwable {
    public void printStackTrace(PrintStream s);
    public StackTraceElement[] getStackTrace();
}

public final class StackTraceElement {
    public String getFileName();
    public int getLineNumber();
    public String getClassName();
    public String getMethodName();
    public boolean isNativeMethod();
}
```
Course themes

• Code-level design
  – Process – how to start
  – Patterns – re-use conceptual solutions
  – Criteria – e.g. evolveability, understandability

• Analysis and modeling
  – Practical specification techniques and verification tools

• Object-oriented programming
  – Evolveability, reuse
  – Industry use – basis for frameworks
  – Vehicle is Java – industry, upper-division courses

Threads and Concurrency
  – System abstraction – background computing
  – Performance
  – Our focus: explicit, application-level concurrency
    • Cf. functional parallelism (150, 210) and systems concurrency (213)
Today: Concurrency, part 1

- The backstory
  - Motivation, goals, problems, ...

- Basic concurrency in Java
  - Synchronization

- Coming soon (but not today):
  - Higher-level abstractions for concurrency
    - Data structures
    - Computational frameworks
Learning goals

• Understand concurrency as a source of complexity in software
• Know common abstractions for parallelism and concurrency, and the trade-offs among them
  – Explicit concurrency
    • Write thread-safe concurrent programs in Java
    • Recognize data race conditions
  – Know common thread-safe data structures, including high-level details of their implementation
  – Understand trade-offs between mutable and immutable data structures
  – Know common uses of concurrency in software design
Processor speeds over time
Power requirements of a CPU

• Approx.: Capacitance * Voltage$^2$ * Frequency

• To increase performance:
  – More transistors, thinner wires
    • More power leakage: increase V
  – Increase clock frequency F
    • Change electrical state faster: increase V

• *Dennard scaling*: As transistors get smaller, power density is approximately constant...
  – ...until early 2000s

• Now: Power requirements are super-linear to CPU performance
  – Heat output is proportional to power input
One option: fix the symptom

- Dissipate the heat
One option: fix the symptom

• Better: Dissipate the heat with liquid nitrogen
  – Overclocking by Tom's Hardware's 5 GHz project

http://www.tomshardware.com/reviews/5-ghz-project,731-8.html
Another option: fix the underlying problem

• Reduce heat by limiting power input
  – Adding processors increases power requirements linearly with performance
    • Reduce power requirement by reducing the frequency and voltage
    • Problem: requires concurrent processing
Aside: Three sources of disruptive innovation

• Growth crosses some threshold
  – e.g., Concurrency: ability to add transistors exceeded ability to dissipate heat

• Colliding growth curves
  – Rapid design change forced by jump from one curve onto another

• Network effects
  – Amplification of small triggers leads to rapid change
Aside: The threshold for distributed computing

- Too big for a single computer?
  - Forces use of distributed architecture
    - Shifts responsibility for reliability from hardware to software
      - Allows you to buy larger cluster of cheap flaky machines instead of expensive slightly-less-flaky machines
        » Revolutionizes data center design
Aside: Colliding growth curves

- From http://www.genome.gov/sequencingcosts/
Aside: Network effects

- Metcalfe's rule: network value grows quadratically in the number of nodes
  - a.k.a. Why my mom has a Facebook account
  - $n(n-1)/2$ potential connections for $n$ nodes

- Creates a strong imperative to merge networks
  - Communication standards, media formats, ...
Concurrency

• Simply: doing more than one thing at a time
  – In software: more than one point of control
    • Threads, processes
• Resources simultaneously accessed by more than one thread or process
Concurrency then and now

• In the past multi-threading was just a convenient abstraction
  – GUI design: event threads
  – Server design: isolate each client's work
  – Workflow design: producers and consumers

• Now: must use concurrency for scalability and performance
Problems of concurrency

• Realizing the potential
  – Keeping all threads busy doing useful work

• Delivering the right language abstractions
  – How do programmers think about concurrency?
  – Aside: parallelism vs. concurrency

• Non-determinism
  – Repeating the same input can yield different results
Realizing the potential

• Possible metrics of success
  – Breadth: extent of simultaneous activity
    • width of the shape
  – Depth (or span): length of longest computation
    • height of the shape
  – Work: total effort required
    • area of the shape

• What are the typical goals in parallel algorithm design?
Amdahl’s law: How good can the depth get?

- Ideal **parallelism** with $N$ processors:
  - Speedup = $N$

- In reality, some work is always inherently sequential
  - Let $F$ be the portion of the total task time that is inherently sequential
  - Speedup =
    \[
    \frac{1}{F + \frac{1 - F}{N}}
    \]
    
  - Suppose $F = 10\%$. What is the max speedup? (you choose $N$)
Amdahl’s law: How good can the depth get?

- **Ideal parallelism** with $N$ processors:
  - Speedup = $N$

- **In reality, some work is always inherently sequential**
  - Let $F$ be the portion of the total task time that is inherently sequential
  - Speedup = \[ \frac{1}{F + \frac{(1 - F)}{N}} \]

- Suppose $F = 10\%$. What is the max speedup? (you choose $N$)
  - As $N$ approaches $\infty$, \(1/(0.1 + 0.9/N)\) approaches 10.
Using Amdahl’s law as a design guide

• For a given algorithm, suppose
  – N processors
  – Problem size M
  – Sequential portion $F$

• An obvious question:
  – What happens to speedup as N scales?

• A less obvious, important question:
  – What happens to $F$ as problem size M scales?

"For the past 30 years, computer performance has been driven by Moore’s Law; from now on, it will be driven by Amdahl’s Law."

— Doron Rajwan, Intel Corp
Abstractions of concurrency

- **Processes**
  - Execution environment is isolated
    - Processor, in-memory state, files, ...
  - Inter-process communication typically slow, via message passing
    - Sockets, pipes, ...

- **Threads**
  - Execution environment is shared
  - Inter-thread communication typically fast, via shared state
Aside: Abstractions of concurrency

• What you see:
  – State is all shared

• A (slightly) more accurate view of the hardware:
  – Separate state stored in registers and caches
  – Shared state stored in caches and memory
Basic concurrency in Java

- **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();
  ```

- **See IncrementTest.java**
Principles of Software Construction: Objects, Design, and Concurrency

Part 6: Concurrency and distributed systems

The Perils of Concurrency, Part 2
Can't live with it...
Can't live without it...

Jonathan Aldrich   Charlie Garrod
Administirvia

• Homework 5a due tomorrow 9 a.m.
• 2nd midterm exam returned today at end of class
• Do you want to be a software engineer?
The foundations of the Software Engineering minor

- Core computer science fundamentals
- Building good software
- Organizing a software project
  - Development teams, customers, and users
  - Process, requirements, estimation, management, and methods
- The larger context of software
  - Business, society, policy
- Engineering experience
- Communication skills
  - Written and oral
SE minor requirements

• Prerequisite: 15-214
• Two core courses
  – 15-313 Foundations of SE (fall semesters)
  – 15-413 SE Practicum (spring semesters)
• Three electives
  – Technical
  – Engineering
  – Business or policy
• Software engineering internship + reflection
  – 8+ weeks in an industrial setting, then
  – 17-413
To apply to be a Software Engineering minor

- Email clegoues@cs.cmu.edu
  - Your name, Andrew ID, class year, QPA, and minor/majors
  - Why you want to be a SE minor
  - Proposed schedule of coursework

- Spring applications due by Friday, 13 Nov 2015
  - Only 15 SE minors accepted per graduating class

- More information at:
  - http://isri.cmu.edu/education/undergrad/
Key concepts from last Tuesday
Today: Concurrency, part 2

• The backstory
  – Motivation, goals, problems, ...

• Basic concurrency in Java
  – Synchronization

• Coming soon:
  – Higher-level abstractions for concurrency
    • Data structures
    • Computational frameworks
Basic concurrency in Java

- **The java.langRunnable 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();
  ```

- **See IncrementTest.java**
Atomicity

- An action is \textit{atomic} if it is indivisible
  - Effectively, it happens all at once
    - No effects of the action are visible until it is complete
    - No other actions have an effect during the action
- In Java, integer increment is not atomic

\begin{itemize}
\item[i++]\quad is actually\quad \begin{itemize}
\item Load data from variable \texttt{i}
\item Increment data by \texttt{1}
\item Store data to variable \texttt{i}
\end{itemize}
\end{itemize}
One concurrency problem: race conditions

- A race condition is when multiple threads access shared data and unexpected results occur depending on the order of their actions.
- E.g., from IncrementTest.java:
  - Suppose `classData` starts with the value 41:
    
    **Thread A:**
    ```java
    classData++;
    ```
    
    **Thread B:**
    ```java
    classData++;
    ```
    
    **One possible interleaving of actions:**
    
    1A. Load data(41) from `classData`
    1B. Load data(41) from `classData`
    2A. Increment data(41) by 1 -> 42
    2B. Increment data(41) by 1 -> 42
    3A. Store data(42) to `classData`
    3B. Store data(42) to `classData`
Race conditions in real life

• E.g., check-then-act on the highway
Race conditions in real life

- E.g., check-then-act at the bank
  - The "debit-credit problem"

### Alice, Bob, Bill, and the Bank

- **A. Alice to pay Bob $30**
  - Bank actions
    1. Does Alice have $30?
    2. Give $30 to Bob
    3. Take $30 from Alice

- **B. Alice to pay Bill $30**
  - Bank actions
    1. Does Alice have $30?
    2. Give $30 to Bill
    3. Take $30 from Alice

- If Alice starts with $40, can Bob and Bill both get $30?
Race conditions in real life

• E.g., check-then-act at the bank
  – The "debit-credit problem"

**Alice, Bob, Bill, and the Bank**

• **A. Alice to pay Bob $30**
  - Bank actions
    1. Does Alice have $30?
    2. Give $30 to *Bob*
    3. Take $30 from *Alice*

• **B. Alice to pay Bill $30**
  - Bank actions
    1. Does Alice have $30?
    2. Give $30 to *Bill*
    3. Take $30 from *Alice*

• If *Alice* starts with $40, can *Bob* and *Bill* both get $30?
Race conditions in your life

- E.g., check-then-act in simple code

```java
public class StringConverter {
    private Object o;
    public void set(Object o) {
        this.o = o;
    }
    public String get() {
        if (o == null) return "null";
        return o.toString();
    }
}
```

- See StringConverter.java, Getter.java, Setter.java
Some actions are atomic

Precondition: int $i = 7$;

Thread A: $i = 42$;

Thread B: ans = $i$;

• What are the possible values for ans?
Some actions are atomic

Precondition:  Thread A:  Thread B:
\[
\text{int } i = 7; \quad i = 42; \quad \text{ans} = i;
\]

• What are the possible values for \( \text{ans} \)?

\[
\begin{align*}
\text{i: } & \quad \begin{array}{c}
00000\ldots00000111 \\
\vdots \\
00000\ldots00101010 
\end{array} \\
\text{i: } & \quad \begin{array}{c}
00000\ldots00101010 
\end{array}
\end{align*}
\]
Some actions are atomic

Precondition:  
Thread A:  
Thread B:  

- What are the possible values for \( \text{ans} \)?

- In Java:
  - Reading an int variable is atomic
  - Writing an int variable is atomic

- Thankfully, \( \text{ans: } 00000...00101111 \) is not possible
Bad news: some simple actions are not atomic

- Consider a single 64-bit long value

- Concurrently:
  - Thread A writing high bits and low bits
  - Thread B reading high bits and low bits

Precondition: Thread A: Thread B:

```
long i = 10000000000;
i = 42;
ans = i;
```

```
ans: 01001...00000000

ans: 00000...00101010

ans: 01001...00101010

(10000000000)

(42)

(10000000042 or ...)
```
Primitive concurrency control in Java

• Each Java object has an associated intrinsic lock
  – All locks are initially unowned
  – Each lock is exclusive: it can be owned by at most one thread at a time

• The synchronized keyword forces the current thread to obtain an object's intrinsic lock
  – E.g.,
    
    synchronized void foo() { ... } // locks "this"

    synchronized(fromAcct) {
        if (fromAcct.getBalance() >= 30) {
            toAcct.deposit(30);
            fromAcct.withdrawal(30);
        }
    }

• See SynchronizedIncrementTest.java
Primitive concurrency control in Java

• `java.lang.Object` allows some coordination via the intrinsic lock:
  ```java
  void wait();
  void wait(long timeout);
  void wait(long timeout, int nanos);
  void notify();
  void notifyAll();
  ```

• See `Blocker.java`, `Notifier.java`, `NotifyExample.java`
Primitive concurrency control in Java

- Locks are *exclusive* and *reentrant*:
  - Each lock can be owned by only one thread at a time
  - If a thread owns a lock, it can lock the lock multiple times
- A thread can own multiple locks

```java
synchronized(lock1) {
    // do stuff that requires lock1

    synchronized(lock2) {
        // do stuff that requires both locks
    }

    // ...
}
```
Another concurrency problem: deadlock

- E.g., Alice and Bob, unaware of each other, both need file A and network connection B
  - Alice gets lock for file A
  - Bob gets lock for network connection B
  - Alice tries to get lock for network connection B, and waits...
  - Bob tries to get lock for file A, and waits...

- See Counter.java and DeadlockExample.java
Dealing with deadlock (abstractly, not with Java)

• Detect deadlock
  – Statically?
  – Dynamically at run time?

• Avoid deadlock

• Alternative approaches
  – Automatic restarts
  – Optimistic concurrency control
Detecting deadlock with the waits-for graph

- The *waits-for graph* represents dependencies between threads
  - Each node in the graph represents a thread
  - A directed edge $T_1 \rightarrow T_2$ represents that thread $T_1$ is waiting for a lock that $T_2$ owns
- Deadlock has occurred iff the waits-for graph contains a cycle
Deadlock avoidance algorithms

• Prevent deadlock instead of detecting it
  – E.g., impose total order on all locks, require locks acquisition to satisfy that order
  • Thread:
    acquire(lock1)
    acquire(lock2)
    acquire(lock9)
    acquire(lock42)  // now can't acquire lock30, etc...
Avoiding deadlock with restarts

• One option: If thread needs a lock out of order, restart the thread
  – Get the new lock in order this time

• Another option: Arbitrarily kill and restart long-running threads
Avoiding deadlock with restarts

• One option: If thread needs a lock out of order, restart the thread
  – Get the new lock in order this time
• Another option: Arbitrarily kill and restart long-running threads
• Optimistic concurrency control
  – e.g., with a copy-on-write system
  – Don't lock, just detect conflicts later
    • Restart a thread if a conflict occurs
Another concurrency problem: livelock

- In systems involving restarts, *livelock* can occur
  - Lack of progress due to repeated restarts
- *Starvation*: when some task(s) is(are) repeatedly restarted because of other tasks
Concurrency control in Java

• Using primitive synchronization, you are responsible for correctness:
  – Avoiding race conditions
  – Progress (avoiding deadlock)

• Java provides tools to help:
  – java.util.concurrent.atomic
  – java.util.concurrent
The power of immutability

• Recall: Data is *mutable* if it can change over time. Otherwise it is *immutable*.
  – Primitive data declared as `final` is always immutable
• After immutable data is initialized, it is immune from race conditions
The `java.util.concurrent.atomic` package

- Concrete classes supporting atomic operations
  - AtomicInteger
    ```java
    int get();
    void set(int newValue);
    int getAndSet(int newValue);
    int getAndAdd(int delta);
    boolean compareAndSet(int expectedValue, int newValue);
    ```

  ...

  - AtomicIntegerArray
  - AtomicBoolean
  - AtomicLong
  - ...
The `java.util.concurrent` package

- Interfaces and concrete thread-safe data structure implementations
  - `ConcurrentHashMap`
  - `BlockingQueue`
    - `ArrayBlockingQueue`
    - `SynchronousQueue`
  - `CopyOnWriteArrayList`
  - ...
- Other tools for high-performance multi-threading
  - `ThreadPools and Executor services`
  - `Locks and Latches`
java.util.concurrent.ConcurrentHashMap

• Implements java.util.Map<K,V>
  – High concurrency lock striping
    • Internally uses multiple locks, each dedicated to a region of the hash table
    • Locks just the part of the table you actually use
    • You use the ConcurrentHashMap like any other map...

![Diagram showing ConcurrentHashMap and its locks and hash table]
java.util.concurrent.BlockingQueue

• Implements java.util.Queue<E>
• java.util.concurrent.SynchronousQueue
  – Each put directly waits for a corresponding poll
  – Internally uses wait/notify
• java.util.concurrent.ArrayBlockingQueue
  – put blocks if the queue is full
  – poll blocks if the queue is empty
  – Internally uses wait/notify
The CopyOnWriteArrayList

• Implements java.util.List<E>
• All writes to the list copy the array storing the list elements
Concurrent at the language level

• Consider:
  ```java
  int sum = 0;
  Iterator i = coll.iterator();
  while (i.hasNext()) {
    sum += i.next();
  }
  ```

• In python:
  ```python
  sum = 0;
  for item in coll:
    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
• What is the total work? What is the depth?
  – What assumptions do you have to make?
Prefix sums (a.k.a. inclusive scan)

- Goal: given array $x[0...n-1]$, compute array of the sum of each prefix of $x$
  
  \[
  \begin{array}{c}
  \text{sum}(x[0...0]), \\
  \text{sum}(x[0...1]), \\
  \text{sum}(x[0...2]), \\
  \vdots \\
  \text{sum}(x[0...n-1])
  \end{array}
  \]

- 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: 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, winding

- Computes the partial sums in a more useful manner

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

- Computes the partial sums in a more useful manner

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

- Computes the partial sums in a more useful manner

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

- Now unwinds to calculate the other sums

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

- 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:

\[ [13, 9, -4, 19, -6, 2, 6, 3] \]
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., \(x = [13, 9, -4, 19, -6, 2, 6, 3]\)
• Pseudocode:

\[
\text{prefix sums}(x):
\]

\[
\text{for } d \text{ in } 0 \text{ to } (\log_2 n) - 1:\quad // \text{d is depth}
\]

\[
\text{parallel for } i \text{ in } 2^d - 1 \text{ to } n-1, \text{ by } 2^{d+1}:
\]

\[
\text{x[i+2}^d\text{]} = \text{x[i]} + \text{x[i+2}^d\text{]}
\]

\[
\text{for } d \text{ in } (\log_2 n) - 1 \text{ to } 0:
\]

\[
\text{parallel for } i \text{ in } 2^d - 1 \text{ to } n-1-2^d, \text{ by } 2^{d+1}:
\]

\[
\text{if } (i-2^d >= 0):
\]

\[
\text{x[i]} = \text{x[i]} + \text{x[i-2}^d\text{]}
\]
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);
        }
    }
}
```
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);
    }
}
```
Parallel prefix sums algorithm

- How good is this?
Parallel prefix sums algorithm

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

• See Main.java, PrefixSumsNonconcurrentParallelWorkImpl.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<Callable<V>> tasks);
  Future<V> invokeAny(Collection<Callable<V>> tasks);
  ```
Executors for common computational patterns

• From the `java.util.concurrent.Executors` class:
  ```java
  static ExecutorService newSingleThreadExecutor();
  static ExecutorService newFixedThreadPool(int n);
  static ExecutorService newCachedThreadPool();
  static ExecutorService newScheduledThreadPool(int n);
  ```

• Aside: see `NetworkServer.java` (later)
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!
Parallel prefix sums algorithm

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

• See PrefixSumsSequentialImpl.java
Parallel prefix sums algorithm

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

• 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.
Next week...

- Introduction to distributed systems
In-class example for parallel prefix sums

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