CSE332: Data Abstractions
Final Review

Nicholas Shahan
Winter 2015

Adapted from slides by Hye In Kim
Final Logistics

• Final on Wednesday, March 18th
  – Time: 12:30-2:20pm in Kane 220
    • No notes or no books
  – Info on website under “Final Exam”
Topics (short list)

• Sorting
• Graphs
• Parallelization
• Concurrency
• Amortized Analysis
• P, NP, NP-completeness
• Material in Midterm
  (fair game but not the focus)
Preparing for the Exam

• Written Homework a good indication of what could be on exam

• Check out previous quarters’ exams
  – 332 exams
  – 326 exams differ quite a bit
  – Final info site has links

• Make sure you:
  – Understand the key concepts
  – Can perform the key algorithms
Sorting Topics

• Know
  – Insertion & Selection sorts - O(n^2)
  – Heap Sort - O(n log n)
  – Merge Sort - O(n log n)
  – Quick Sort - O(n log n) on average
  – Bucket Sort & Radix Sort

• Know run-times
• Know how to carry out the sort
• Lower Bound for Comparison Sort
  – Cannot do better than O(n log n)
  – Won’t be ask to give full proof
  – But may be asked to use similar techniques
  – Be familiar with the ideas
Mergesort example: Merge as we return from recursive calls

We need another array in which to do each merging step. Merge results into there, then copy back to original array.
Graph Topics

• Graph Basics
  – Definition, weights, directedness, degree
  – Paths, cycles
  – Connectedness (directed vs undirected)
  – ‘Tree’ in a graph sense
  – DAGs

• Graph Representations
  – Adjacency List
  – Adjacency Matrix
  – What each is, how to use it

• Graph Traversals
  – Breadth-First
  – Depth-First
  – What data structures are associated with each?
Graph Topics

• Topological Sort
• Dijkstra’s Algorithm
  – Doesn’t play nice with negative weights
• Minimum Spanning Trees
  – Prim’s Algorithm
  – Kruskal’s Algorithm
• Know algorithms
• Know run-times
Dijkstra’s Algorithm Overview

Given a weighted graph and a vertex in the graph (call it A), find the shortest path from A to each other vertex

- Cost of path defined as sum of weights of edges
- Negative edges not allowed

The algorithm:
- Create a table like this:
- Init A’s cost to 0, others infinity (or just ‘??’)
- While there are unknown vertices:
  - Select unknown vertex w/ lowest cost (A initially)
  - Mark it as known
  - Update cost and path to all unknown vertices adjacent to that vertex

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>??</td>
<td></td>
</tr>
</tbody>
</table>
Parallelism

• Fork-join parallelism
  – Know the concept; diff. from making lots of threads
  – Be able to write pseudo-code
  – Reduce: parallel sum, multiply, min, find, etc.
  – Map: bit vector, string length, etc.

• Work & span definitions
• Speed-up & parallelism definitions
• Justification for run-time, given tree
• Justification for ‘halving’ each step
• Amdahl’s Law
• Parallel Prefix
  – Technique
  – Span
  – Uses: Parallel prefix sum, filter, etc.
• Parallel Sorting
Parallelism Overview

• We say it takes time $T_p$ to complete a task with $P$ processors.

• Adding together an array of $n$ elements would take $O(n)$ time, when done sequentially (that is, $P=1$)
  - Called the **work**; $T_1$

• If we have ‘enough’ processors, we can do it much faster; $O(\log n)$ time
  - Called the **span**; $T_\infty$
Considering Parallel Run-time

Our **fork** and **join** frequently look like this:

- Each node takes $O(1)$ time
  - Even the base cases, as they are at the cut-off
- Sequentially, we can do this in $O(n)$ time; $O(1)$ for each node, $\sim 3n$ nodes, if there were no cut-off (linear # on base case row, halved each row up/down)
- Carrying this out in (perfect) parallel will take the time of the longest branch; $\sim 2\log n$, if we halve each time
Some Parallelism Definitions

• **Speed-up** on \( P \) processors: \( \frac{T_1}{T_P} \)

• We often assume perfect linear speed-up
  – That is, \( \frac{T_1}{T_P} = P \); w/ 2x processors, it’s twice as fast
  – ‘Perfect linear speed-up ’usually our goal; hard to get in practice

• **Parallelism** is the maximum possible speed-up: \( \frac{T_1}{T_\infty} \)
  – At some point, adding processors won’t help
  – What that point is depends on the span
The ForkJoin Framework Expected Performance

If you write your program well, you can get the following expected performance:

$$T_P \leq \left( \frac{T_1}{P} \right) + O(T_\infty)$$

- $T_1/P$ for the overall work split between $P$ processors
  - $P=4$? Each processor takes 1/4 of the total work
- $O(T_\infty)$ for merging results
  - Even if $P=\infty$, then we still need to do $O(T_\infty)$ to merge results

• What does it mean??
  - We can get decent benefit for adding more processors; effectively linear speed-up at first (expected)
  - With a large # of processors, we’re still bounded by $T_\infty$; that term becomes dominant
Amdahl’s Law

Let the **work** (time to run on 1 processor) be 1 unit time

Let $S$ be the portion of the execution that **cannot** be parallelized

Then: $T_1 = S + (1-S) = 1$

Then: $T_P = S + (1-S)/P$

Amdahl’s Law: The overall **speedup** with $P$ processors is:

$$T_1 / T_P = 1 / (S + (1-S)/P)$$

And the **parallelism** (infinite processors) is:

$$T_1 / T_\infty = 1 / S$$
Parallel Prefix Sum

• Given an array of numbers, compute an array of their running sums in $O(\log n)$ span

• Requires 2 passes (each a parallel traversal)
  – First is to gather information
  – Second figures out output

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>4</th>
<th>16</th>
<th>10</th>
<th>16</th>
<th>14</th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>6</td>
<td>10</td>
<td>26</td>
<td>36</td>
<td>52</td>
<td>66</td>
<td>68</td>
<td>76</td>
</tr>
</tbody>
</table>
## Parallel Prefix Sum

Two passes:
1) Compute ‘sum’
2) Compute ‘fromleft’

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>26</td>
</tr>
<tr>
<td>10</td>
<td>36</td>
</tr>
<tr>
<td>16</td>
<td>52</td>
</tr>
<tr>
<td>14</td>
<td>66</td>
</tr>
<tr>
<td>2</td>
<td>68</td>
</tr>
<tr>
<td>8</td>
<td>76</td>
</tr>
</tbody>
</table>

### Diagram

```
  range 0,8
  fromleft 0

  range 4,8
  fromleft 36

  range 0,4
  sum 36
  fromleft 0

  range 2,4
  sum 26
  fromleft 10

  range 4,6
  sum 52
  fromleft 36

  range 6,8
  sum 68
  fromleft 66

  range 0,2
  sum 10
  fromleft 0

  range 2,3
  sum 16
  fromleft 0

  range 3,4
  sum 26
  fromleft 10

  range 4,5
  sum 36
  fromleft 16

  range 5,6
  sum 52
  fromleft 14

  range 6,7
  sum 66
  fromleft 2

  range 7,8
  sum 68
  fromleft 8
```

### Two passes:
1) Compute ‘sum’
2) Compute ‘fromleft’
Parallel Quicksort

2 optimizations:

1. Do the two recursive calls in parallel
   - Now recurrence takes the form:
     \[ O(n) + 1T(n/2) \]
   - So \( O(n) \) span

2. Parallelize the partitioning step
   - Partitioning normally \( O(n) \) time
   - Recall that we can use Parallel Prefix Sum to ‘filter’ with \( O(\log n) \) span
   - Partitioning can be done with 2 filters, so \( O(\log n) \) span for each partitioning step

These two parallel optimizations bring parallel quicksort to a span of \( O(\log^2 n) \)
Concurrency

• Race conditions
• Data races
• Synchronizing your code
  – Locks, Reentrant locks
  – Java’s ‘synchronize’ statement
  – Readers/writer locks
  – Deadlock
  – Issues of critical section size
  – Issues of lock scheme granularity – coarse vs fine
• Knowledge of bad interleavings
• Be able to write pseudo-code for Java threads and, locks
Race Conditions

A race condition occurs when the computation result depends on scheduling (how threads are interleaved)

• If T1 and T2 happened to get scheduled in a certain way, things go wrong
• We, as programmers, cannot control scheduling of threads; result is that we need to write programs that work independent of scheduling

Race conditions are bugs that exist only due to concurrency

• No interleaved scheduling with 1 thread

Typically, problem is that some intermediate state can be seen by another thread; screws up other thread

• Consider a ‘partial’ insert in a linked list; say, a new node has been added to the end, but ‘back’ and ‘count’ haven’t been updated
Data Races

• A *data race* is a specific type of *race condition* that can happen in 2 ways:
  – Two different threads can *potentially* write a variable at the same time
  – One thread can *potentially* write a variable while another reads the variable
  – Simultaneous reads are fine; not a data race, and nothing bad would happen
  – ‘Potentially’ is important; we say the code itself has a data race – it is independent of an actual execution

• Data races are bad, but we can still have a race condition, and bad behavior, when no data races are present
Readers/writer locks

A new synchronization ADT: The **readers/writer lock**

- **Idea:** Allow any number of readers OR one writer
- **This allows more concurrent access (multiple readers)**
- **A lock’s states fall into three categories:**
  - “not held”
  - “held for writing” by one thread
  - “held for reading” by one or more threads
- **new:** make a new lock, initially “not held”
- **acquire_write:** block if currently “held for reading” or “held for writing”, else make “held for writing”
- **release_write:** make “not held”
- **acquire_read:** block if currently “held for writing”, else make/keep “held for reading” and increment **readers count**
- **release_read:** decrement readers count, if 0, make “not held”
Deadlock

• As illustrated by the ‘The Dining Philosophers’ problem
• A deadlock occurs when there are threads $T_1$
  – Each is waiting for a lock held by the next
  – $T_n$ is waiting for a resource held by $T_1$
• In other words, there is a cycle of waiting

```java
class BankAccount {
    ...
    synchronized void withdraw(int amt) {...}
    synchronized void deposit(int amt) {...}
    synchronized void transferTo(int amt, BankAccount a) {
        this.withdraw(amt);
        a.deposit(amt);
    }
}
```

Consider simultaneous transfers from account $x$ to account $y$, and $y$ to $x$
Amortized Analysis

• To have an Amortized Bound of $O(f(n))$:
  – *There does not exist a series of $M$ operations with run-time worse than $O(M*f(n))*

• Amortized vs average case
• To prove: prove that no series of operations can do worse than $O(M*f(n))$
• To disprove: find a series of operations that’s worse
P, NP, NP Completeness

- **P**: set of all problems that can be solved in polynomial time
  - sorting, shortest path, Euler circuit, etc.
- **NP**: set of all problems for which a given candidate solution can be tested in polynomial time
  - Hamiltonian Circuit, Vertex Cover, etc.
P=NP ???

- Currently no proof.
- It is generally believed that $P \neq NP$
- Prove it for fame, fortune and a Turing Award!
NP-Complete

• Set of problems in NP that (we are pretty sure) cannot be solved in polynomial time.
• These are thought of as the hardest problems in the class NP.
• Interesting fact: If any one NP-Complete problem could be solved in polynomial time, then all NP-Complete problems could be solved in polynomial time.
• Even more: If any NP-Complete problem is in P, then all of NP is in P
Is my problem in P or NP?

• Reduce a known NP-Complete problem into your problem
• (not the other way around) via a transformation
  – The transformation must take polynomial time
• Now you can say your problem is at least as hard as a known NP-Complete problem
Working with NP Problems

• Approximation Algorithm
  – Can we get an efficient algorithm that guarantees something close to optimal? (e.g. Answer is guaranteed to be within 1.5x of Optimal, but solved in polynomial time).

• Restrictions
  – Many hard problems are easy for restricted inputs (e.g. graph is always a tree, degree of vertices is always 3 or less).

• Heuristics
  – Can we get something that seems to work well (good approximation/fast enough) most of the time? (e.g. In practice, n is small-ish)