LECTURE 1
Introduction

Julian Shun
February 5, 2019
What is Algorithm Engineering?

- Algorithm design
- Algorithm analysis
- Algorithm implementation
- Optimization
- Profiling
- Experimental evaluation

Theory

Practice

\[ O(n \log n) \]
\[ O(n) \]
\[ O(\log n) \]
Good empirical performance
Confidence that algorithms will perform well in many different settings
Ability to predict performance (e.g. in real-time applications)
Important to develop theoretical models to capture properties of technologies

Use theory to inform practice and practice to inform theory.
• In early days, implementing algorithms designed was standard practice
• 1970s–1980s: Algorithm theory is a subdiscipline in CS mostly devoted to ”paper and pencil” work
• Late 1980s–1990s: Researchers began noticing gaps between theory and practice
• 1997: First Workshop on Algorithm Engineering (WAE) by P. Italiano (now part of ESA)
• 1998: Meeting on Algorithm Engineering & Experiments (ALENEX)
• 2003: annual Workshop on Experimental Algorithms (WEA), now Symposium on Experimental Algorithms (SEA)
• Nowadays many conferences have papers on algorithm engineering
What is Algorithm Engineering?

Source: “Algorithm Engineering – An Attempt at a Definition”, Peter Sanders
Models of Computation

• **Random–Access Machine (RAM)**
  - Infinite memory
  - Arithmetic operations, logical operations, and memory accesses take $O(1)$ time
  - Most sequential algorithms are designed using this model (6.006/6.046)

• **Nowadays computers are much more complex**
  - Deep cache hierarchies
  - Instruction level parallelism
  - Multiple cores
  - Disk if input doesn’t fit in memory
Algorithm Design & Analysis

- Constant factors matter!
- Avoid unnecessary computations
- Simplicity improves applicability and can lead to better performance
- Think about locality and parallelism
- Think both about worst-case and real-world inputs
- Use theory as a guide to find practical algorithms
- Time vs. space tradeoffs

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Algorithm 1</th>
<th>Algorithm 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>N log₂ N</td>
<td>1000 N</td>
<td></td>
</tr>
</tbody>
</table>
Implementation

• Write clean, modular code
  • Easier to experiment with different methods, and can save a lot of development time

• Write correctness checkers
  • Especially important in numerical and geometric applications due to floating-point arithmetic, possibly leading to different results

• Save previous versions of your code!
  • Version control helps with this
Experimentation

- Instrument code with timers and use performance profilers (e.g., perf, gprof, valgrind)
- Use large variety of inputs (both real-world and synthetic)
  - Use different sizes
  - Use worst-case inputs to identify correctness or performance issues
- Reproducibility
  - Document environmental setup
  - Fix random seeds if needed
- Run multiple timings to deal with variance
• For parallel code, test on varying number of processors to study scalability
• Compare with best serial code for problem
• For reproducibility, write deterministic code if possible
  • Or make it easy to turn off non-determinism
• Use `numactl` to control NUMA effects on multi-socket machines
• Useful tools: Cilkscale, Cilksan
Libraries and Frameworks

- Use efficient building blocks from existing library/frameworks when appropriate
- Develop your own to help others and improve applicability
COURSE INFORMATION
Course Information

- Graduate-level class
  - Undergraduates who have taken 6.046 and 6.172 are welcome
- Lectures: Tuesday and Thursday 2:30–4pm
- Instructor: Julian Shun
- Units: 3–0–9
- We will use Piazza for communication

This course will cover various ideas in algorithm engineering, with an emphasis on parallelism and graph problems
## Schedule (tentative)

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Speaker/Required Reading</th>
<th>Optional Reading</th>
</tr>
</thead>
</table>
| Tuesday 2/5| Course Introduction        | Julian Shun  
  *Algorithm Engineering - An Attempt at a Definition*  
  *A Theoretician's Guide to the Experimental Analysis of Algorithms* |  
  *Algorithm Engineering: Bridging the Gap Between Algorithm Theory and Practice*  
  *A Guide to Experimental Algorithmics*  
  *Algorithm engineering: an attempt at a definition using sorting as an example*  
  *Algorithm Engineering for Parallel Computation*  
  *Distributed Algorithm Engineering*  
  *Experimental algorithmics*  
  *Programming Pearls*  
  *Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time* |
| Thursday 2/7| Parallel Algorithms        | Julian Shun  
  *Parallel Algorithms*  
  *Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques (Chapters 4-8)*  
  *CLRS Chapter 27* |  
  *Prefix Sums and Their Applications*  
  *Algorithm Design: Parallel and Sequential*  
  *Introduction to Parallel Algorithms* |
| Tuesday 2/12| Parallel Graph Traversal  |  
  *Direction-Optimizing Breadth-First Search*  
  *A Faster Algorithm for Betweenness Centrality*  
  *The More the Merrier: Efficient Multi-Source Graph Traversal* |  
  *A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers)*  
  *Internally Deterministic Parallel Algorithms Can Be Fast*  
  *SlimSell: A Vectorizable Graph Representation for Breadth-First Search*  
  *Chapter 3.6 of Networks, Crowds, and Markets* (describes Betweenness Centrality with an example)  
  *Better Approximation of Betweenness Centrality*  
  *ABRA: Approximating Betweenness Centrality in Static and Dynamic Graphs with Rademacher Averages*  
  *KADABRA is an ADaptive Algorithm for* |
Grading Breakdown

<table>
<thead>
<tr>
<th>Activity</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paper Reviews</td>
<td>15%</td>
</tr>
<tr>
<td>Paper Presentations</td>
<td>20%</td>
</tr>
<tr>
<td>Research Project</td>
<td>60%</td>
</tr>
<tr>
<td>Class Participation</td>
<td>5%</td>
</tr>
</tbody>
</table>

You must complete all assignments to pass the class.
Paper Presentations

• This is a research-oriented course
• Cover content from 2–3 research papers each lecture
• 25–30 minute student presentation per paper
  • Discuss motivation for the problem solved
  • Key technical ideas
  • Theoretical/experimental results
  • Related work
  • Strengths/weaknesses
  • Directions for future work
  • Include several questions for discussion
  • Presentation should cover necessary background to understand paper (you may have to read related papers)
  • Make slides but may use the board for theoretical proofs
• Sign up for presentations this week in Google doc
• Would be helpful to sign up even if listening
Paper Reviews

• Submit one paper review each week on a paper that will be covered that week
  • Cover motivation, key ideas, results, novelty, strengths/weaknesses, your ideas for improving the techniques or evaluation, any open problems or directions for further work
  • Submit on Learning Modules by Monday 11:59pm each week (before we cover the papers)
  • Reviews will be made viewable to class (anonymously)
  • Read them before the lecture to help prepare for the discussions
Research Project

• Open-ended research project to be done in groups of 1–3 people

• Some ideas
  • Implementation of non-trivial algorithms
  • Analyzing/optimizing performance of existing algorithms
  • Designing new theoretically and/or practically efficient algorithms
  • Applying algorithms in the context of larger applications
  • Improving or designing new algorithm frameworks or libraries
  • Any topic may involve parallelism, cache-efficiency, I/O-efficiency, and memory-efficiency

• Must contain an implementation component

• Can be related to research that you are doing

• On Tuesday 3/5, you can pitch any project ideas you have and find group members
### Project Timeline

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Due Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project idea pitches</td>
<td>3/5</td>
</tr>
<tr>
<td>Pre–proposal meeting</td>
<td>3/12</td>
</tr>
<tr>
<td>Proposal*</td>
<td>3/15</td>
</tr>
<tr>
<td>Weekly progress reports</td>
<td>3/22, 4/5, 4/12, 4/26, 5/3, 5/10</td>
</tr>
<tr>
<td>Mid–term report*</td>
<td>4/19</td>
</tr>
<tr>
<td>Project presentations</td>
<td>5/16</td>
</tr>
<tr>
<td>Final report</td>
<td>5/16</td>
</tr>
</tbody>
</table>

- **Pre–proposal meeting**
  - 15–minute meeting to run idea by instructor
- **Talk to instructors if you need computing resources for the project**
  - We may have some AWS credits

*You can submit it 3 days later if you go to the Comm Lab at least one day before original deadline.*
PARALLELISM
Parallelism

Data is becoming very large!

- Twitter: 41 million vertices, 1.5 billion edges (6.3 GB)
- Yahoo!: 1.4 billion vertices, 6.6 billion edges (38 GB)
- Common Crawl: 3.5 billion vertices, 128 billion edges (540 GB)

Parallel machines are everywhere!

Can rent machines on AWS with 72 cores (144 hyper-threads) and 4TB of RAM
Parallelism Models

- **Work** = number of vertices in graph (number of operations)
- **Depth (Span)** = longest directed path in graph (dependence length)
- **Parallelism** = Work / Depth

**Goal 1:** work-efficient and low (polylogarithmic) depth algorithms

**Goal 2:** simple, practical, and cache-friendy
GRAPHS
What is a graph?

- **Vertices** model objects
- **Edges** model relationships between objects

![Graph Diagram](https://commons.wikimedia.org/wiki/File:Protein_Interaction_Network_for_TMEM8A.png)
Graph Representations

- Vertices labeled from 0 to n–1

```
0 1 2 3 4
0 1 0 0 0
1 0 0 1 1
0 0 0 1 0
0 1 1 0 0
0 1 0 0 0
```

Adjacency matrix
(“1” if edge exists, “0” otherwise)

- O(n^2) space for adjacency matrix
- O(m) space for edge list
Graph Representations

- **Adjacency list**
  - Array of pointers (one per vertex)
  - Each vertex has an unordered list of its edges

  ![Adjacency list diagram]

- Space requirement is $O(n+m)$
- Can substitute linked lists with arrays for better cache performance
  - Tradeoff: more expensive to update graph
Graph Representations

- Compressed sparse row (CSR)
  - Two arrays: Offsets and Edges
  - Offsets[i] stores the offset of where vertex i’s edges start in Edges

<table>
<thead>
<tr>
<th>Vertex IDs</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offsets</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>Edges</td>
<td>2</td>
<td>7</td>
<td>9</td>
<td>16</td>
</tr>
</tbody>
</table>

- How do we know the degree of a vertex?
- Space usage is $O(n+m)$
- Can also store values on the edges with an additional array or interleaved with Edges
### Tradeoffs in Graph Representations

- What is the cost of different operations?

<table>
<thead>
<tr>
<th></th>
<th>Adjacency matrix</th>
<th>Edge list</th>
<th>Adjacency list (linked list)</th>
<th>Compressed sparse row</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage cost / scanning whole graph</td>
<td>O(n^2)</td>
<td>O(m)</td>
<td>O(m+n)</td>
<td>O(m+n)</td>
</tr>
<tr>
<td>Add edge</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(m+n)</td>
</tr>
<tr>
<td>Delete edge from vertex v</td>
<td>O(1)</td>
<td>O(m)</td>
<td>O(deg(v))</td>
<td>O(m+n)</td>
</tr>
<tr>
<td>Finding all neighbors of a vertex v</td>
<td>O(n)</td>
<td>O(m)</td>
<td>O(deg(v))</td>
<td>O(deg(v))</td>
</tr>
<tr>
<td>Finding if w is a neighbor of v</td>
<td>O(1)</td>
<td>O(m)</td>
<td>O(deg(v))</td>
<td>O(deg(v))</td>
</tr>
</tbody>
</table>

- There are variants/combinations of these representations
BREADTH-FIRST SEARCH
Breadth-First Search (BFS)

- Given a source vertex $s$, visit the vertices in order of distance from $s$.

Possible outputs:
- Vertices in the order they were visited:
  - D, B, C, E, A
- The distance from each vertex to $s$:
  
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- A BFS tree, where each vertex has a parent to a neighbor in the previous level.

Applications:
- Betweenness centrality
- Eccentricity estimation
- Maximum flow
- Web crawlers
- Network broadcasting
- Cycle detection
- ...

© 2019 Julian Shun
Sequential BFS Algorithm

Breadth-First-Search(Graph, root):

for each node n in Graph:
    n.distance = INFINITY
    n.parent = NIL

• **BFS requires O(n+m) work on n vertices and m edges**

Source: https://en.wikipedia.org/wiki/Breadth-first_search
Sequential BFS Algorithm

• Assume graph is given in compressed sparse row format
  • Two arrays: Offsets and Edges
  • \( n \) vertices and \( m \) edges (assume \( \text{Offsets}[n] = m \))

```c
int* parent = (int*) malloc(sizeof(int)*n);
int* queue = (int*) malloc(sizeof(int)*n);

for(int i=0; i<n; i++) {
    parent[i] = -1;
}

queue[0] = source;
parent[source] = source;

int q_front = 0, q_back = 1;
```

// while queue not empty
while(q_front != q_back) {
    int current = queue[q_front++]; // dequeue
    int degree =
        Offsets[current+1]-Offsets[current];
    for(int i=0; i<degree; i++) {
        int ngh = Edges[Offsets[current]+i];
        // check if neighbor has been visited
        if(parent[ngh] == -1) {
            parent[ngh] = current;
            // enqueue neighbor
            queue[q_back++] = ngh;
        }
    }
}

• What is the most expensive part of the code?
  • Random accesses cost more than sequential accesses
Depth-first search
Depth–First Search (DFS)

- Explores edges out of the most recently discovered vertex
- Possible outputs:
  - Depth–first forest
  - Vertices in the order they were first visited (preordering)
  - Vertices in the order they were last visited (postordering)
  - Reverse postordering

Applications
- Topological sort
- Solving mazes
- Biconnected components
- Strongly connected components
- Cycle detection
  ...

Preorder: D, B, A, C, E
Postorder: C, A, B, E, D
Reverse postorder: D, E, B, A, C

DFS requires $O(n+m)$ work on $n$ vertices and $m$ edges
T O P O L O G I C A L S O R T
Topological Sort

- Given a directed acyclic graph, output the vertices in an order such that all predecessors of a vertex appear before it.
  - Application: scheduling tasks with dependencies (e.g. parallel computing, Makefile).
  - Solution: output vertices in reverse postorder in DFS.

Reverse postorder: D, E, B, A, C.
SHORTEST PATHS
Single-Source Shortest Paths

• Given a weighted graph and a source vertex, output the distance from the source vertex to every vertex

• Non-negative weights
  • Dijkstra’s algorithm
  • $O(m + n \log n)$ work using Fibonacci heap

• General weights
  • Bellman–Ford algorithm
  • $O(mn)$ work
Dijkstra’s Algorithm

1. `function Dijkstra(Graph, source):`
2. \[ \text{dist}[source] \leftarrow 0 \] // Initialization
3. create vertex set Q

- \( O((m+n) \log n) \) work using normal heap
- \( O(m + n \log n) \) work using Fibonacci heap
  - Extract-min takes \( O(\log n) \) work but decreasing priority only takes \( O(1) \) work (amortized)
Bellman–Ford Algorithm

Bellman–Ford(G, source):

1. \text{ShortestPaths} = \{\infty, \infty, ..., \infty\}  \quad //\text{size } n; \text{ stores shortest path distances}
2. \text{ShortestPaths}[\text{source}] = 0
3. for \(i = 1\) to \(n-1\):
   for each vertex \(v\) in \(G\):
     for each \(w\) in \text{neighbors}(v):
       if(\text{ShortestPaths}[v] + \text{weight}(v,w) < \text{ShortestPaths}[w]):
         \text{ShortestPaths}[w] = \text{ShortestPaths}[v] + \text{weight}(v,w)

4. if no shortest paths changed:
   return \text{ShortestPaths}
5. report “negative cycle”

- At most \(n\) rounds, each doing \(O(n+m)\) work
- Total work = \(O(mn)\)
More Graph Algorithms

- We will study algorithms for particular problems
  - Parallelism, cache–efficiency, I/O–efficiency, dynamic updates

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-first search</td>
<td>Betweenness centrality</td>
</tr>
<tr>
<td>PageRank</td>
<td>Triangle Computations</td>
</tr>
<tr>
<td>Low-diameter decomposition</td>
<td>SSSP</td>
</tr>
<tr>
<td>Connected components</td>
<td>Maximal independent set</td>
</tr>
<tr>
<td>K-core decomposition</td>
<td>Multi-BFS</td>
</tr>
<tr>
<td>Minimum spanning forest</td>
<td>Spanning forest</td>
</tr>
<tr>
<td>Maximal matching</td>
<td>Set cover</td>
</tr>
<tr>
<td>Eccentricity estimation</td>
<td>Subgraph matching</td>
</tr>
</tbody>
</table>
Graph processing frameworks
Graph Processing Frameworks

- Provides high level primitives for graph algorithms
- Reduce programming effort of writing efficient parallel graph programs

Graph processing frameworks/libraries

Pregel, Giraph, GPS, GraphLab, PowerGraph, PRISM, Pegasus, Knowledge Discovery Toolbox, CombBLAS, GraphChi, GraphX, Galois, X-Stream, Gunrock, GraphMat, Ringo, TurboGraph, TurboGraph++, FlashGraph, Grace, PathGraph, Polymer, GPSA, GoFFish, Blogel, LightGraph, MapGraph, PowerLyra, PowerSwitch, Imitator, XDGP, Signal/Collect, PrefEdge, EmptyHeaded, Gemini, Wukong, Parallel BGL, KLA, Grappa, Chronos, Green-Marl, GraphHP, P++, LLAMA, Venus, Cyclops, Medusa, NScale, Neo4J, Trinity, GBase, HyperGraphDB, Horton, GSPARQL, Titan, ZipG, Cagra, Milk, Ligra, Ligra+, Julienne, GraphPad, Mosaic, BigSparse, Graphene, Mizan, Green-Marl, PGX, PGX.D, Wukong+S, Stinger, cuStinger, Distingier, Hornet, GraphIn, Tornado, Bagel, Kick Starter, Naiad, Kineograph, GraphMap, Presto, Cube, Giraph++, Photon, TuX2, GRAPE, GraM, Congra, MTGL, GridGraph, NXgraph, Chaos, Mmap, Clip, Floe, GraphGrind, DualSim, ScaleMine, Arabesque, GraMi, SAHAD, Facebook TAO, Weaver, G-SQL, G-SPARQL, gStore, Horton+, S2RDF, Quegel, EAGRE, Shape, RDF-3X, CuSha, Garaph, Totem, GTS, Frog, GBTL-CUDA, Graphulo, Zorro, Coral, GraphTau, Wonderland, GraphP, GraphIt, GraPu, GraphJet, ImmortalGraph, LA3, CellliQ, AsyncStripe, Cgraph, GraphD, GraphH, ASAP, RStream, and many others…

© 2019 Julian Shun
Dynamic Graphs
Dynamic Graphs

- Many graphs are changing over time
  - Adding/deleting connections on social networks
  - Traffic conditions changing
  - Communication networks (email, IMs)
  - World Wide Web
  - Content sharing (Youtube, Flickr, Pinterest)

- Need graph data structures that allow for efficient updates (in parallel)
- Need (parallel) algorithms that respond to changes without re-computing from scratch
COMPRESSION
• What if you cannot fit a graph on your machine?
• Cost of machines increases with memory size

Graph Compression
Graph Compression on CSR

- **Vertex IDs**: 0, 1, 2, 3
- **Offsets**: 0, 4, 5, 11
- **Edges**: 2, 7, 9, 16, 0, 1, 6, 9, 12
- **Compressed Edges**: 2, 5, 2, 7, -1, -1, 5, 3, 3

**Sort edges and encode differences**

- **2 - 0 = 2**
- **7 - 2 = 5**
- **1 - 2 = -1**

**For each vertex v:**
- First edge: difference is `Edges[Offsets[v]] - v`
- i’th edge (`i > 1`): difference is `Edges[Offsets[v] + i] - Edges[Offsets[v] + i - 1]`

- Want to use fewer than 32 or 64 bits per value
- Compression can improve running time

© 2019 Julian Shun
Fast Compression Schemes

- Study speed and space tradeoffs in compression schemes for integer sequences
- Study how compression has been used to speed up sparse matrix–vector multiplication and graph processing
- Also useful in storing inverted lists for information retrieval
Graph Reordering

- Reassign IDs to vertices to improve locality
  - Goal: Make vertex IDs close to their neighbors’ IDs and neighbors’ IDs close to each other

```
4  1  3
0  2  5
```

```
0  1  3  4
1  2  5
```

- Can improve compression rate due to smaller “differences”
- Can improve performance due to higher cache hit rate
- Various methods: BFS, DFS, METIS, degree, etc.
CACHING AND NON–UNIFORM MEMORY ACCESS
Cache Hierarchies

Design cache-efficient and cache-oblivious algorithms to improve locality

<table>
<thead>
<tr>
<th>Memory level</th>
<th>Approx latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td>1–2ns</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>3–5ns</td>
</tr>
<tr>
<td>L3 cache</td>
<td>12–40ns</td>
</tr>
<tr>
<td>DRAM</td>
<td>60–100ns</td>
</tr>
</tbody>
</table>
Non-uniform Memory Access (NUMA)

- Accessing remote memory is more expensive than accessing local memory of a socket
  - Latency depends on the number of hops

Design NUMA–aware algorithms to improve locality
I/O Efficiency
I/O Efficiency

• Need to read input from disk at least once
• Need to read many more times if input doesn’t fit in memory

<table>
<thead>
<tr>
<th>Memory</th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>60–100 ns</td>
<td>Tens of GB/s</td>
</tr>
<tr>
<td>SSD</td>
<td>Tens of µs</td>
<td>500 MB–2 GB/s (seq), 50–200 MB/s (rand)</td>
</tr>
<tr>
<td>HDD</td>
<td>Tens of ms</td>
<td>200 MB/s (seq), 1 MB/s (rand)</td>
</tr>
</tbody>
</table>

I/O Efficiency

- For graphs larger than main memory, disk-based computing can be competitive with distributed clusters
- GraphChi: Large-Scale Graph Computation on Just a PC (OSDI 2012)

<table>
<thead>
<tr>
<th>Application &amp; Graph</th>
<th>Iter.</th>
<th>Comparative result</th>
<th>GraphChi (Mac Mini)</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pagerank &amp; domain</td>
<td>3</td>
<td>GraphLab[30] on AMD server (8 CPUs) 87 s</td>
<td>132 s</td>
<td>-</td>
</tr>
<tr>
<td>Pagerank &amp; twitter-2010</td>
<td>5</td>
<td>Spark [45] with 50 nodes (100 CPUs): 486.6 s</td>
<td>790 s</td>
<td>[38]</td>
</tr>
<tr>
<td>Pagerank &amp; V=105M, E=3.7B</td>
<td>100</td>
<td>Stanford GPS, 30 EC2 nodes (60 virt. cores), 144 min</td>
<td>approx. 581 min</td>
<td>[37]</td>
</tr>
<tr>
<td>Pagerank &amp; V=1.0B, E=18.5B</td>
<td>1</td>
<td>Piccolo, 100 EC2 instances (200 cores) 70 s</td>
<td>approx. 26 min</td>
<td>[36]</td>
</tr>
<tr>
<td>Webgraph-BP &amp; yahoo-web</td>
<td>1</td>
<td>Pegasus (Hadoop) on 100 machines: 22 min</td>
<td>27 min</td>
<td>[22]</td>
</tr>
<tr>
<td>ALS &amp; netflix-mm, D=20</td>
<td>10</td>
<td>GraphLab on AMD server: 4.7 min</td>
<td>9.8 min (in-mem)</td>
<td>[30]</td>
</tr>
<tr>
<td>Triangle-count &amp; twitter-2010</td>
<td>-</td>
<td>Hadoop, 1636 nodes: 423 min</td>
<td>40 min (edge-repl.)</td>
<td>[39]</td>
</tr>
<tr>
<td>Pagerank &amp; twitter-2010</td>
<td>1</td>
<td>PowerGraph, 64 x 8 cores: 3.6 s</td>
<td>158 s</td>
<td>[20]</td>
</tr>
<tr>
<td>Triangle-count &amp; twitter- 2010</td>
<td>1</td>
<td>PowerGraph, 64 x 8 cores: 1.5 min</td>
<td>60 min</td>
<td>[20]</td>
</tr>
</tbody>
</table>

- Lots of follow-up work on disk-based computing that we will study
- External-memory algorithms to minimize I/O’s
SORTING ALGORITHMS
• Lots of research on engineering sorting algorithms
• Will study parallel comparison sorting and radix sorting algorithms
• [http://sortbenchmark.org/](http://sortbenchmark.org/)
STRING ALGORITHMS
• We will study algorithms for efficiently constructing suffix arrays and suffix trees
• Many other interesting problems (edit distance, Lempel–Ziv compression, approximate string matching, alignment, etc.)
GEOMETRY ALGORITHMS
We will study how to efficiently triangulate a mesh (Delaunay triangulation).

Many other interesting problems (convex hull, linear programming, segment intersection, point location, space partitions, etc.)

Be careful with numerical issues.
Binary Search Trees
Binary Search Trees

- We will study different types of binary trees
- We will study how to efficiently construct and update binary search trees in parallel
- We will look at applications such as range trees, interval trees, segment and rectangle queries
JOINS AND AGGREGATION
• JOIN and GROUPBY are two of the most expensive operations in database systems
• We will study algorithms and optimizations for these operations (in main-memory)
WRITE-EFFICIENT ALGORITHMS
Emerging Memory Technologies

- Non-volatile memories projected to become a dominant form of main memory
- Significant gap in cost for reads vs. writes (energy and latency)
- Need to design models and algorithms that take read-write asymmetry into account
Parallel Scheduling
Parallel Scheduling

• Manually scheduling threads is difficult

• Cilk work-stealing scheduler
  • How can we translate work and depth bounds into efficient parallel running times in theory and practice?

• Space-bounded scheduler
  • How can we get efficient running times and cache-efficiency?
Relevant Topics Not Covered

- GPUs, other accelerators, and special-purpose hardware
- Networking
- Matrix computations
- Linear and integer programming
- Optimizing NP-hard problems
- Succinct data structures
- Concurrent data structures
- Transactional memory
- Performance of different programming languages
- Deep learning
• Lots of exciting research going on in algorithm engineering!
• Take this course to learn about latest results and try out research in the area