LECTURE 1
Introduction

Julian Shun
February 6, 2024
What is Algorithm Engineering?

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

Theory

Practice

© 2018–2024 MIT Algorithm Engineering Instructors
Bridging Theory and Practice

- 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.
Brief History

• 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

© 2018–2024 MIT Algorithm Engineering Instructors
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
  • Read and write costs are not necessary the same
<table>
<thead>
<tr>
<th>Complexity</th>
<th>Algorithm 1</th>
<th>Algorithm 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N \log_2 N$</td>
<td>$1000N$</td>
<td></td>
</tr>
</tbody>
</table>

- 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
- Work vs. parallelism tradeoffs
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 times to deal with variance
Experimentation II

- For parallel code, test on varying number of processors to study scalability
- Compare with best serial code for problem
- For reproducibility, write deterministic parallel 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 libraries/frameworks when appropriate
- Contribute to existing libraries/frameworks or develop your own to help others and improve applicability
COURSE INFORMATION
Course Information

• Graduate-level class
  • Undergraduates who have taken 6.122 (6.046) and 6.106 (6.172) are welcome

• Lectures: Tuesday/Thursday 11am–12:30pm ET in 3–370

• Instructor:
  • Julian Shun (jshun@mit.edu)

• TA
  • Amy Hu (amyhu@mit.edu)

• Units: 3–0–9

• We will use Piazza for communication

• Office hours by appointment
<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Speaker</th>
<th>Required Reading</th>
<th>Optional Reading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuesday 2/6</td>
<td>Course Introduction</td>
<td>Julian Shun</td>
<td>Algorithm Engineering - An Attempt at a Definition</td>
<td>Algorithm Engineering: Bridging the Gap Between Algorithm Theory and Practice</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A Theoretician's Guide to the Experimental Analysis of Algorithms</td>
<td>A Guide to Experimental Algorithmics</td>
</tr>
<tr>
<td>Thursday 2/8</td>
<td>Parallel Algorithms</td>
<td>Julian Shun</td>
<td>Parallel Algorithms</td>
<td>Algorithm engineering: an attempt at a definition using sorting as an example</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques (Chapters 4-8)</td>
<td>Algorithm Engineering for Parallel Computation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CLRS Chapter 26 (Parallel Algorithms)</td>
<td>Distributed Algorithm Engineering</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Experimental algorithmics</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Programming Pearls</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time</td>
</tr>
<tr>
<td>Tuesday 2/13</td>
<td>Parallel Graph Traversal</td>
<td></td>
<td>Direction-Optimizing Breadth-First Search</td>
<td>Prefix Sums and Their Applications</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The More the Merrier: Efficient Multi-Source Graph Traversal</td>
<td>Algorithm Design: Parallel and Sequential</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Introduction to Parallel Algorithms</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Scheduling Multithreaded Computations by Work Stealing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Thread Scheduling for Multiprogrammed Multiprocessors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Multidimensional Included and Excluded Sums</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Work-Efficient Parallel Algorithms for Accurate Floating-Point Prefix Sums</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Problem Based Benchmark Suite</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Internally Deterministic Parallel Algorithms Can Be Fast</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SlimSell: A Vectorizable Graph Representation for Breadth-First Search</td>
</tr>
</tbody>
</table>
Grading

<table>
<thead>
<tr>
<th>Grading Breakdown</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Paper Reviews</td>
<td>20%</td>
</tr>
<tr>
<td>Problem Set</td>
<td>10%</td>
</tr>
<tr>
<td>Paper Presentations</td>
<td>15%</td>
</tr>
<tr>
<td>Research Project</td>
<td>45%</td>
</tr>
<tr>
<td>Class Participation</td>
<td>10%</td>
</tr>
</tbody>
</table>

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

• Cover content from 2 research papers each lecture
• 25–30 minute student presentation + Q&A 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 whiteboard for theory

• Student presentations begin next Tuesday
• Sign-up sheet will be released soon
• Please sign up even if you are a listener
**Paper Reviews**

- **Submit one paper review for each lecture**
  - Starting next 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 Canvas by 10am ET on the day of each lecture (before we cover the papers)
Problem Set

- Complete a problem set on parallel algorithms
  - To be released this week and due on Monday 3/4
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 algorithm frameworks or libraries, parallel runtime systems, or software productivity tools
  • 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
## Project Timeline

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Due Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-proposal meeting</td>
<td>3/5</td>
</tr>
<tr>
<td>Proposal</td>
<td>3/15</td>
</tr>
<tr>
<td>Weekly progress reports</td>
<td>3/22, 4/5, 4/12, 4/19, 4/26, 5/3, 5/10</td>
</tr>
<tr>
<td>Mid-term report</td>
<td>4/16</td>
</tr>
<tr>
<td>Project presentations</td>
<td>5/14</td>
</tr>
<tr>
<td>Final report</td>
<td>5/14</td>
</tr>
</tbody>
</table>

- **Pre-proposal meeting**
  - 15-minute meeting to run ideas by instructors
- **Computing resources for the project**
  - Sign up for Google Cloud Platform for free cloud computing credits
  - Talk to instructor if you need additional credits
PARALLELISM
Parallelism

Data is becoming very large!

- Twitter: 41 million vertices, 1.5 billion edges (6.3 GB)
- Yahoo! Finance: 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 up to 224 cores (448 hyper-threads) and 24TB of RAM
Parallelism Models

- **Work** = number of vertices in graph (number of operations)
- **Span** (depth) = longest directed path in graph (dependence length)
- **Running time** \( \leq \frac{\text{Work}}{\#\text{processors}} + O(\text{Span}) \)
- A **work-efficient** parallel algorithm has work that asymptotically matches that of the best sequential algorithm for the problem

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

Goal 2: simple, practical, and cache-friendly
Cilk 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?
Graphs
What is a graph?

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

Alice → Bob → Eve → Fred → Greg → Hannah
Carol → David
Graph Representations

- Graph has $n$ vertices and $m$ edges
- Vertices labeled from 0 to $n-1$

Adjacency matrix
("1" if edge exists, "0" otherwise)

Edge list

- $O(n^2)$ 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

- 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 weights 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

© 2018–2024 MIT Algorithm Engineering Instructors
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></th>
<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>
<td></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

...
Sequential BFS Algorithm

1. procedure BFS(G, root) is
   2.     let Q be a queue
   3.     label root as explored
   4.     Q.enqueue(root)
   5.     while Q is not empty do
   6.         v := Q.dequeue()
   9.         for all edges from v to w in G.adjacentEdges(v) do
   10.        if w is not labeled as explored then
   11.           label w as explored
   12.           Q.enqueue(w)

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

• What is the running time of BFS?
  • Each node is enqueued and dequeued once: O(n)
  • Each edge is visited once in each direction: O(m)
  • Total: O(n+m)
Sequential BFS Algorithm

- Assume graph is given in compressed sparse row format
  - Two arrays: Offsets and Edges
- n vertices and m edges (assume 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

Total of m random accesses
Analyzing the program

What if we can fit a bitvector of size $n$ in cache?

- Might reduce the number of cache misses
- More computation to do bit manipulation

```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(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;
        }
    }
}
```

Check bitvector first before accessing parent array

$n$ cache misses instead of $m$
BFS with bitvector

```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;
}

for(int i=0; i<n; i++) {
    visited[i] = 0;
}

queue[0] = source;
pARENT[source] = source;
visited[source/32] = (1 << (source % 32));

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(!((1 << ngh%32) & visited[ngh/32])){
            visited[ngh/32] |= (1 << (ngh%32));
            parent[ngh] = current;
            // enqueue neighbor
            queue[q_back++] = ngh;
        }
    }
}
```

- Bitvector version is faster for large enough values of m

© 2018–2024 MIT Algorithm Engineering Instructors
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
- ...

DFS requires $O(n+m)$ work on $n$ vertices and $m$ edges.

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

```
function Dijkstra(Graph, source):
    dist[source] ← 0 // Initialization
    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):
ShortestPaths = {∞, ∞, ..., ∞} //size n; stores shortest path distances
ShortestPaths[source] = 0
for i=1 to n:
    for each vertex v in G:
        for each w in neighbors(v):
            if(ShortestPaths[v] + weight(v,w) < ShortestPaths[w]):
                ShortestPaths[w] = ShortestPaths[v] + weight(v,w)
if no shortest paths changed:
    return ShortestPaths
report “negative cycle”

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

• We will study algorithms for particular problems
  • Parallelism, cache–efficiency, dynamic updates

<table>
<thead>
<tr>
<th>Breadth–first search</th>
<th>Betweenness centrality</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank</td>
<td>Spanning forest</td>
</tr>
<tr>
<td>Low–diameter decomposition</td>
<td>Maximal independent set</td>
</tr>
<tr>
<td>Connected components</td>
<td>Graph clustering</td>
</tr>
<tr>
<td>Graph neural networks</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, Distinger, Hornet, GraphIn, Tornado, Bagel, KickStarter, 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, CellIQ, AsyncStripe, Cgraph, GraphD, GraphH, ASAP, RStream, and many others…
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
Large Graphs

• What if you cannot fit a graph on your machine?
• Cost of machines increases with memory size

Graph Compression
Graph Compression on CSR

<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edges</th>
<th>2</th>
<th>7</th>
<th>9</th>
<th>16</th>
<th>0</th>
<th>1</th>
<th>6</th>
<th>9</th>
<th>12</th>
</tr>
</thead>
</table>

| Compressed Edges | 2 | 5 | 2 | 7 | -1 | -1 | 5 | 3 | 3 |

- For each vertex v:
  - First edge: difference is $\text{Edges}[\text{Offsets}[v]] - v$
  - $i$’th edge ($i > 1$): difference is $\text{Edges}[\text{Offsets}[v] + i] - \text{Edges}[\text{Offsets}[v] + i - 1]$

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

![Graph before reordering](image1)

![Graph after reordering](image2)

- Can improve compression rate due to smaller “differences”
- Can improve performance due to higher cache hit rate
- Various methods: BFS, DFS, METIS, degree, etc.
Cache-Efficiency and I/O-Efficiency
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>
### I/O Efficiency

- Need to read input from disk at least once
- May 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>

SORTING ALGORITHMS
• Lots of research on engineering sorting algorithms
• Will study parallel comparison sorting and integer sorting algorithms
• [http://sortbenchmark.org/](http://sortbenchmark.org/)
JOINS AND AGGREGATION
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)
Stencil Computations
Stencil Computations

- Computations that iteratively update data based on a fixed pattern (stencil)
  - For example, can be used to approximately solve heat equation
- We will study algorithms for stencil computations that improve on work, parallelism, and cache-efficiency over standard approaches

Source: https://en.wikipedia.org/wiki/Iterative_Stencil_Loops
CLUSTERING
Clustering

- Group “similar” objects together, and separate “dissimilar” objects
- Can be applied to graph data and spatial data
- Applications: Community detection, bioinformatics, parallel/distributed processing, visualization, image segmentation, anomaly detection, document analysis, machine learning, etc.
**Graph Neural Networks**
Graph Neural Networks (GNNs)


- Traditional neural networks have a fixed topology, but in GNNs the topology is the graph
  - Repeatedly pass messages to neighbors, and aggregate messages received to update node
  - Each node has a different computation graph!
  - Many different graph neural networks, based on how they pass and aggregate messages
  - We’ll study some high-performance GNN approaches
Summary

- Lots of exciting research going on in algorithm engineering!
- Take this course to learn about latest results and try out research in the area