Algorithm Engineering Reading Notes

6.6050 Algorithm Engineering

October 1, 2025

Contents

A Theoretician's Guide to the Experimental Analysis of Algorithms

Author: David S. Johnson (2001)

1.1 General Notes

1.1.1 Governing Principles

1. **Perform newsworthy experiments**: Problems should have direct applications. Where do you get your test set from if it has no application? Also, algorithms should be somewhat competitive with the ones that are used in practice. Have it be relevant, general, and credible.
2. **Tie paper to the literature**: What actually are the interesting questions? What behaviour needs explaining, and what algorithms seem open to improvement? Try use implementations from previous papers.
3. **Use instance testbeds that can support general conclusions**: You have the choice between (a) instances from real-world/pseudo-real-world applications and (b) randomly generated instances (ideally to be structured like real-world instances). Random instance generators should be able to generate instances of arbitrarily large size.
4. **Use efficient and effective experimental designs**: e.g. variance reduction techniques (use the same set of randomly generated instances for all the algorithms you're testing), bootstrapping to evaluate multiple-run heuristics, use self-documenting programs (save data in descriptively named files)
5. **Use reasonably efficient implementations**: Efficiency comes at a cost in effort. Don't cut corners that would prevent you from making a fair comparison but also don't overdo it.
6. **Ensure reproducibility**: If you run the same code on the same instances of machine/compiler/OS/system load combination you should get the same runtime, operation count, solution quality. More broadly, if someone uses the same method will they be able to draw the same conclusions.
7. **Ensure compatibility**: Write your papers so that future researchers can compare their algorithms/instances to your results
8. **Report the full story**: Don't be overly selective with how you present your data. Maybe include tables in the appendix, but definitely include them.
9. **Draw well-justified conclusions and look for explanations**: What did you learn from your experiments?
10. **Present your data in informative ways**: Use good display techniques!

1.1.2 Possible Questions

1. How do implementation details, parameter settings, heuristics, data structure choices affect runtime of algorithm?
2. How does runtime of algorithm scale with instance size? How does this depend on instance structure?

3. What algorithmic operation counts best to help explain runtime?
4. What in practice are the computational bottlenecks? How do they depend on instance size and structure? How does this differ from predictions of worst-case analysis?
5. How is runtime (and runtime growth rate) affected by machine architecture? Can detailed profiling help explain it?
6. Given that one is running on the same/similar instances and on a fixed machine, how predictable are runtimes?
7. How does runtime compare to top competitors? How are comparisons affected by instance size/structure/machine architecture? Can differences be explained in terms of operation counts?
8. What are the answers to the above questions when "runtime" is replaced by "memory usage"/usage of some other computational resource?
9. What are answers to 1, 2, 6, 7 when one deals with approximation algorithms and "runtime" is replaced with "solution quality"?
10. Given a new class of instances you've identified, does it cause significant changes in the behaviour of previously studied algorithms?

1.1.3 Whom can you trust?

- Never trust a random number generator
- Never trust your code to be correct
- Never trust a previous author to have known all the literature
- Never trust your memory as to where you put that data (and how it was generated)
- Never trust your computer to remain unchanged
- Never trust backup media or websites to remain readable indefinitely
- Never trust a so-called expert on experimental analysis

1.2 Paper Review

1.2.1 Motivation

Describes issues that arise when algorithms are analyzed experimentally (challenges with rigorous analysis led to the emphasis on theoretical worst-/average-case analysis w.r.t. asymptotic behavior in the first place)

1.2.2 Key Ideas/Results

- Key metrics are resource usage (time/memory) and quality of output solution
- Reasons for implementing an algorithm:
  - **Application paper**: Using code in a particular application, often to prove mathematical conjectures. Here, result is more important than efficiency
  - **Horse-race paper**: Evidence of superiority of algorithmic ideas
  - **Experimental analysis paper**: Understand strengths, weaknesses, operation of interesting algorithmic ideas in practice
  - **Experimental average-case paper**: Understand average-case behaviour of algorithms where direct probabilistic analysis is too hard

- Rules for governing the writing of experimental papers:
  - Perform newsworthy experiments
  - Tie paper to literature
  - Use instance testbeds that can support general conclusions
  - Use efficient and effective experimental designs
  - Use reasonably efficient implementations
  - Ensure reproducibility
  - Ensure comparability
  - Report the full story
  - Draw well-justified conclusions and look for explanations
  - Present your data in informative ways

1.2.3 Pet Peeves

- Authors/referees who don't do their homework (make sure your algorithm isn't dominated)
- Focusing on unstructured random instances that don't reflect real data OR on exclusively real data that may be outdated
- In the millisecond testbed, runtime is somewhat irrelevant
- When evaluating approximation algorithms, don't limit yourself to algorithms where the solution/optimal value is known (in this case, using an optimization algorithm is reasonable, so why use approximation)
- Claiming "inadequate programming time/ability" as an excuse
- Supplying code that doesn't match a paper's description of it
- Irreproducible standards of comparison (e.g. only reporting the solution value, only reporting the percentage excess over the best solution currently known, reporting percentage excess over an estimate of expected optimal for randomly generated instances, reporting percentage excess over a well-defined lower bound, reporting percentage excess/improvement over some other heuristic)
- Using runtime as a stopping criterion (it's not a cake that you're baking)
- Using optimal solution value as a stopping criterion for algorithms that have no way of verifying optimality
- Hand-tuned algorithm parameters
- One-run study (unless study covers a wide range of instances)
- Using the best result found as an evaluation criterion (is often from the tail of the distribution)
- Uncalibrated machine (include processor speed, operating system, language/compiler)
- The lost testbed (make sure future researchers have access to your tests)
- False precision (more digits of accuracy than justified by the data)
- Unremarked anomalies
- Ex post facto stopping criterion (total running time is not reported, just the time taken to find the answer)
- Failure to report overall runtimes (even if your main focus is not runtime)

- Data without interpretation (at least be able to summarize patterns in the data)
- Conclusions without support
- Myopic approaches to asymptopia (medium/large instance behaviour doesn't necessarily translate to VERY large instance behaviour)
- Tables without pictures
- Pictures without tables
- Pictures with too little insight
- Inadequately/confusingly labeled pictures
- Pictures with too much information (best when clear and uncluttered)
- Confusing pictorial metaphors
- Spurious trend lines (drawing lines between each point)
- Poorly structured tables (order rows and columns so that they highlight important information)
- Undefined metric (cryptic labels)
- Comparing apples to oranges (algorithms tested on different instances, or on different machines)
- Detailed stats on unimportant questions
- Comparing approximation algorithms as to how often they find optima (restricts attention to test instances for which optimal solutions are known, ignores question of how near to optimal algorithm gets when it does not find the optimum)
- Too much data! Replace raw data with averages and other summary stats

1.2.4 Pitfalls

- Dealing with dominated (always slower than status quo) algorithms
- Devoting too much computation to the wrong questions (overstudying results, running full experimental suites before algorithm is efficient or you've decided what data to collect)
- Getting into an endless loop in the experimentation (draw the line on future research)
- Using randomly generated instances to evaluate behaviour of algorithm ends up exploring properties of randomly generated instances
- Too much code tuning (don't overthink, concentrate programming effort where it will be the most useful)
- Lost code/data (don't modify code without saving version of original, don't forget to keep backup copies, organize your directories)

1.2.5 Suggestions

- Think before you compute. Have you implemented it correctly? What do you want to study? What are your experiments addressing?
- Use exploratory experimentation to find good questions
- Use benchmark codes to calibrate machine speeds (some portable source code that other researchers can use to calibrate in the future)
- Use profiling to understand runtimes (number of calls to various subroutines, time spent in subroutines etc. can point out higher-exponent components of runtime earlier). Also offers a mode of explaining the results.
- Display normalized runtimes

1.2.6 Novelty

Very experienced researcher discussing the state of the art of paper writing in this (somewhat niche?) field.

1.2.7 Strengths/Weaknesses

Personal/objective opinion. Otherwise very thorough and goes through a whole series of do's/don'ts that are akin to the advice a PI might give when instructing students on how to write a paper in that field.

1.2.8 Ideas for Improving Techniques/Evaluation

1.2.9 Open Problems/Directions for Future Work

Would be nice to have some sense of the appropriate structure.

Algorithm Engineering: An Attempt at Definition

Author: Peter Sanders

2.1 General Notes

2.1.1 Introduction

- Fast search (Google) makes large quantities of text searchable
- Algorithms offer efficiency + performance guarantees
- Gap between theory and practice:
  - Traditional machine models for theorizing about algorithms don't match hardware parallelism, memory hierarchies etc.
  - Theoretically designed algorithms often not implementable
- Algorithmic engineering process similar to scientific method:
  - Falsifiable hypothesis that can be (in)validated by experiments
  - Reproducibility
  - Application-oriented design supplies realistic inputs, constraints, and design parameters
- Distinct from application engineering (which sits at the intersection of algorithm engineering and software engineering/developing production quality code):
  - Algorithm engineering emphasises fast dev, efficiency, instrumentation for experiments
  - Application engineering emphasises simplicity, thorough testing, maintainability, simplicity, tuning for particular inputs
  - Link is often created through algorithm libraries

2.1.2 History

- 1970s/80s: Algorithm theory become subdiscipline of CS devoted to "paper and pencil" work
- 1986: "Algorithm engineering" is term but not discussed
- 1997: "Workshop in Algorithm Engineering" developed

2.1.3 Models

2.1.4 External Memory Model (I/O Model)

- Two levels of memory (c.f. uniform memory): fast memory of limited size M and slow memory accessed in blocks of size B
- Cost to doing internal work (cost ratio between disk memory and RAM is $\sim$ 1:200)

2.1.5 Design

- Efficient algorithms must also consider constant factors
  - e.g. Maximum Flow algorithms: asymptotically best algorithm performs much worse than theoretically inferior algorithms

2.1.6 MST Example

- Undirected connected graph G with n nodes and m edges (edges with nonnegative weights)
- MST of G is subset of edges with minimum total weight that forms a spanning tree of G
- Can be solved in O(sort(m)) expected I/O steps where sort(N) = O(N/B $\log_{M/B}$ N/B) and denotes the number of I/O steps required for external sorting.
- New algorithm – semiexternal variant of Kruskal's algorithm:
  - Semiexternal graph is allowed O(n) words of fast memory
  - Edge accepted into MST if connecting two components of forest defined by previously found MST edges (does not create a cycle). Uses union-find (disjoint set union, DSU) data structure to store connected components
  - m ¿ what can fit into memory but n can. Edges are sorted externally and then streams them into RAM to apply Kruskal's procedure there. Using path compression, only log(n) bits are needed per node.

2.1.7 Analysis

- Difficult to analyze even some simple/proven algorithms
- Simplex algorithm is exponential (theoretically) but in practice, and especially with the addition of random permutations, linear/polynomial

2.1.8 Implementation

- Essentially describes several algorithms that were too complex/required too difficult data structures to implement efficiently until many years later

2.1.9 Experiments

- Benchmarking the falsifiable hypothesis
- Difficult to test experiments with external memory algorithms (large inputs, huge runtime)

2.1.10 Algorithm Libraries

- Challenging to design since applications are "unknown" at implementation time
- Interface needs to be distinct from implementation
- Examples:
  - LEDA (Library of Efficient Data Types and Algorithms) for C++
  - Boost: forum for library designers that ensures quality and maybe to become part of STL
  - CGAL (Computational Geometry Algorithms Library): sophisticated example of C++ template programming

2.1.11 Instances

- Collections of realistic problem instances for benchmarking NP-hard problems e.g. traveling salesman, Steiner tree, satisfiability, graph partitioning
- More difficult to obtain polynomial ones e.g. route planning, flows etc. (which usually use random inputs instead)

2.2 Paper Review

2.2.1 Motivation

"Algorithmic engineering" lacks a formal definition but has a clear use-case. Paper is attempt at defining the field.

2.2.2 Key Ideas

- Algorithm engineering is the experimental branch of algorithmic theory
  - P. Italiano "Workshop in Algorithm Engineering" (1997) describes algorithm engineering as "concerned with the design, analysis, implementation, tuning, debugging, and experimental evaluation of computer programs for solving algorithmic problems... provides methodologies and tools for developing and engineering efficient algorithmic codes and aims at integrating and reinforcing traditional theoretical approaches for the design and analysis of algorithms and data structures"
- We need new memory models to capture the internal work done by an algorithm

2.2.3 Results

Uses Minimum Spanning Tree (MST) algorithm as an example.

2.2.4 Novelty

2.2.5 Strengths/Weaknesses

2.2.6 Ideas for Improving Techniques/Evaluation

2.2.7 Open Problems/Directions for Future Work

Parallel Algorithms

Authors: Guy E. Blelloch and Bruce M. Maggs

3.1 General Notes

- Parallelism in an algorithm $\neq$ ability of a computer to perform operations in parallel

3.2 Modeling Parallel Computations

- Sequential algorithms formulated using RAM (random access machine) model i.e. one processor connected to a memory system
  - Each basic CPU operation (arithmetic, logic, memory access) takes one time step
- Parallel computers have more complex organization

3.2.1 Multiprocessor Models

- Sequential RAM with more than one processor
- Types: local memory machine models, modular memory machine models, parallel random-access machine (PRAM) models
  - **LMM**: One interconnection network with many processors. Each processor has its own local memory. Local operations (e.g. local memory access) take unit time. Time taken to access another processor's memory is larger.
  - **MMM**: Interconnection network has many processors and many memories. Processors are linked to memory via interconnection network, and access it using memory requests through the network. Time for any processor to reach any given memory module is approximately uniform.
  - **PRAM**: One shared memory with many processors. A processor can access any "word" of memory in one step. Accesses from multiple processors can happen in parallel. "Ideal" case, not realistic.

3.2.2 Network topology

Definition 1. Network: collection of switches connected by communication channels. Processor/memory module has 1+ communication ports connected to the switches.

Definition 2. Network topology: pattern of interconnection of switches.

Definition 3. 2-dimensional mesh: network that can be laid out in a rectangular fashion. Each switch has label (x, y) i.e. its coordinates within X, Y bounds.

Algorithms have been designed to work with specific types of memory. May not work well on other networks and more complex than those designed for abstract models e.g. PRAM.

- Bus: simplest network topology
  - Local and modular memory machine models
  - All processors/memory modules are connected to a single bus

  - At most one piece of data can be written to the bus per step (request from processor to read/write or response from processor/memory module that holds the value)
  - Pros: easy to build, all processors/memory modules can observe traffic on the bus so easy to develop protocols for local cacheing
  - Cons: processors have to take turns using the bus (especially a problem with many processors)
- Mesh:
  - Number of switches = $X \cdot Y$
  - Often in local memory machine models (each processor with its local memory is connected to each switch)
  - Remote memory accesses done by routing messages through the mesh
  - Can also have 3D meshes, torus meshes, hypercubes
- Multistage network:
  - Used to connect one set of input switches to another set of output switches through a sequence of stages of switches
  - Stages numbered 1-L (L = depth of network). Input switches are on stage 1, output on L.
  - Often used in modular memory computers (processors attached to input, memory to output). Word of memory accessed by injecting memory access request message into network.
  - 2-stage network connecting 4 processors to 16 memory modules. Each swtich has two channels at bottom and four at the top. More memory than processors because processors can generate memory access requests faster than memory modules can service them
- Fat-tree:
  - Structured like a tree. Each edge can represent many communication channels/each node can represent many switches.
  - Memory requests travel down the tree to least-common ancestor then go back up.

Alternate modeling technique: latency and bandwidth

- Latency: time taken for message to traverse the network (topology dependent)
- Bandwidth: rate at which data can be injected into the network (topology dependent); minimum gap g between successive injections of messages into the network.
- Models include Postal Model (model described by single parameter L), Bulk-Synchronous Parallel model (L and g), LogP model (L, g, and o (overhead/wasted time on sending/receiving message))

3.2.3 Primitive operations

What kinds of operations are the processors/network able to perform?

- Assume all porcessors are allowed to perform in same local instructions as single processor in RAM
- Special instructions for non-local memory requests, sending messages to other processors, global operations e.g. synchronization, restrictions to avoid processors from interfering with each other (e.g. writing to same memory locaiton)
- Nonlocal instruction types:
  - Instructions performing concurrent accesses to same shared memory locaiton
  - Instructions for synchronization
  - Instructions performing global operations on data

- What happens when processors try to read/write to same resource (processor, memroy module, memory location) at the same time?
  - Exclusive access to resource: forbid the action
  - Concurrent access to resource: unlimited access
  - Queued access: time for a step is proportional to the max number of accesses to a resource
- Other primitives support synchronization, combining arithmetic operations with communication etc.

3.2.4 Work-Depth Models

Definition 4. Work depth model: the cost of an algorithm is determined by examining the total number of operations it performs (W), and the dependencies among those operations (D).

Definition 5. W work: Number of operations an algorithm performs D depth: longest chain of dependencies among operations P paralellism: W/D

If the sequential cost of an algorithm is W, then the ideal parallel time with P processors is $\approx T_P \approx W/P + D$ where we can evenly spread work amongst processors but cannot go faster than the critical path.

- Pros: no machine-dependent details. Often lead to realistic implementation
- Classes:
  - Circuit models: most abstract. Nodes and directed arcs, where node is a basic operation. Number of incoming arcs = fan-in, outgoing arcs = fan-out. Input arcs provide input to the whole circuit. No directed cycle permitted.
  - Vector machine models: algorithm is a sequence of steps which perform operations on a vector of input values, producing an output vector.
  - Language-based models: work-depth cost is associated with each programming language construct (e.g. work of calling two functions in parallel == work of two calls)

3.2.5 Costs

Work W = number of processes $\times$ time required for algorithm to complete execution Depth D = total time required to execute the algorithm

3.2.6 Emulations

An algorithm designed for one parallel model can often be translated into algorithms that work for another (work-preserving, i.e. work performed by both algorithms is approximately the same)

This section goes through a proof showing that a PRAM processor can be "emulated" by a more realistic multiprocessor processor with a butterfly network with only logarithmic slowdowns.

3.3 Parallel algorithmic techniques

3.3.1 Divide-and-conquer

Split the problem into subproblems that are easier to solve than the original, solve the subproblem, and merge the solutions. Often inherently parallelizable as operations are typically independent.

Mergesort

- Sorts n keys by splitting keys into two sequences of n/2 keys, recursively sorting each sequence, merging two sorted sequences of n/2 keys into sorted sequence of n keys.
- Each half can be sent to a parallel process

3.3.2 Randomization

- Used in parallel algorithms to ensure that processors can make global decisions which very probably lead to good global decisions. E.g. selecting a representative sampling, breaking symmetry, load balancing (dividing data into evenly sized subsets)

3.3.3 Parallel pointer manipulation

Many operations for lists, trees, graphs (e.g. traversing linked list, visiting tree nodes, DFS) are inherently sequential. Parallel alternatives include:

- **Pointer jumping** for lists and trees. Each node in parallel replaces its pointer with that of its successor/parent. After (at most log n) steps every node points to the root of the tree
- **Euler tour** computing subtrees, tree depths, or node levels that would originally require sequential traversal can be done all at the same time
- **Graph contraction** a graph is reduced in size while maintaining some of its original structure. Problem is solved on contracted graph, then used for final solution.
- **Ear decomposition** Partition of graph edges into ordered collection of paths. First is a cycle, others are ears. Replaces depth first search.
- Other: small grpah seperators, hashing for load balancing and mapping addresses to memory

3.4 Basic operations on sequences, lists, trees

```
1 ALGORITHM: sum(A)
2 if (A.size() = 1) return A[0]
3 else return sum({A[2i] + A[2i + 1] : i \in [0...A.size()/2]})
```

This can also be used to calculate max etc.

```
1 ALGORITHM: scan(A)
2 if (A.size() == 1) return [0]
3 else
4     S = scan({A[2i] + A[2i + 1] : i in [0..A.size()/2]})
5     R = {if (i mod 2 == 0) then S[i/2] else S[(i-1)/2] + A[i-1] : i in [0...A.size
          ()]}
```

Element-wise adding even elements of A to the odd elements, recursively solving the problem on the resulting sequence.

Multiprefix generalizes the scan operation to multiple independent scans, where for [(1,5), (0,2), (0,3), (1,4), (0,1), (2,2)] each position receives the sum of all the elements that have the same key to yield [0, 0, 2, 5, 5, 0].

Fetch and add is multiprefix but the order of input elements for the scan is not necessarily the same as the order in input sequence A.

3.4.1 Pointer jumping

Each node i replaces pointer P[i] with pointer of the node it points to, P[P[i]]. Can compute a pointer to the end of the list/root of tree for each node.

```
1 ALGORITHM point_to_root(P)
2 for j from 1 to [log(P.size())]
3     P := {P[P[i]] : i \in [0...P.size()]}
```

3.4.2 List ranking

Computing distance from each node to the end of a linked list.

```
ALGORITHM list_rank(P)
V = {if P[i] = i then 0 else 1 : i \in [0...P.size()]}
for j from 1 to [log(P.size())]
    V := {V[i] + V[P[i]] : i \in [0...P.size()]}
    P := {P[P[i]] : i \in [0...P.size()]}
```

Where V[i] is distance spanned by pointer P[i] w.r.t. original list.

These are not work efficient since it takes O(n log n) work vs sequential algorithms that can do it in O(n).

3.4.3 Removing duplicates

Input and output are both sequences. Order doesn't matter.

1. Using an array of flags: initialize a second array that keeps track of the initial appearance of each value. Only add the ones that do not repeat more than once. Explodes for longer lists.

2. Hashing: create a hash table containing a prime number of entries, where prime is 2x as big as the number of items. If multiple items are attempted to be written into the hash table, only one will succeed. May not work the first iteration because values are defeated by values that are different. Needs several iterations with different hash functions.

```
ALGORITHM remove_duplicates(V)
m := next_prime(2 * V.size())
table := distribute(-1, m)
i := 0 // different hash function used for each iteration
result := {}
while V.size() > 0
    table := table <- {(hash(V[j], m, i), j) : j \in [0...V.size()]}
    winners := {V[j] : j \in [0...V.size()] | table[hash(V[j], m, i)] = j}
    result := result ++ winners
    table := table <- {(hash(k, m, i), k) : k \in winners}
    V := {k \in V | table[hash(k, m, i)] != k}
    i := i + 1
return result
```

3.5 Graphs

Most graph problems do not parallelize well.

Definition 6. Sparse graph: m (number of edges) $\ll n^2$ (number of nodes)

Definition 7. Diameter D(G): maximum, over all pairs of vertices (u,v), of the minimum number of edges that need to be traversed from u to v

Edge lists, adjacency lists, adjacency matrices used to represent graphs. For parallel algorithms, linked lists are reprsented with arrays (e.g. edge-list = array of edges, adjacency-list = array of arrays).

3.5.1 BFS

Parallel similar to sequential version. Start with source vertex s, traverse each level of the graph to find vertices that have not yet been visited. Each level is visited in parallel and no queue is required.

Maintain a set of frontier vertices to keep track of current level and produce new frontier on next step. Collect all neighbours of current frontier vertices in parallel and remove any that have not been visited. Multiple vertices may have the same uncollected vertex, so we need to remove duplicates.

```
ALGORITHM: bfs(s, G)
front := [s]
tree := distribute(-1, G.size())
tree[s] := s
while (front.size() != 0)
    E := flatten({{(u,v) : u \in G[v]} : v \in front})
    E' := {(u,v) : E | tree[u] = -1}
    tree := tree <- E'
    front := {u : (u,v) \in E' | v = tree[u]}
return tree
```

Where front is the frontier vertices, tree contains the current BFS tree. Iterations terminate when no more vertices in frontier. Each vertex and edges are only visited once, so O(m+n).

Can generate trees that cannot be generated with sequential BFS.

3.5.2 Connected components

How to label connected components of an undirected graph, such that two vertices u, v have the same label if and only if there is a path between them. BFS and DFS are very inefficient.

Graph contraction helps. Contract vertices of a subgraph into a single vertex.

1. Random mate graph contraction

- Form a set of star subgraphs (tree depth 1), contract the seperators. Merge child into parent
- Randomly decide if vertex is parent or child. Neighboring parent vertex is identified and made the child's root.
- Repeat until all components have size 1

How many contraction steps we need depends on how many vertices are removed in each step. Only children will be removed (P(child) = 1/2), and only if there's a neighbouring parent (P(has neighbouring parent) ¿= 1/2). The probability that a vertex is removed is P = 1/2 * 1/2 = 1/4.

```
ALGORITHM cc_random_mate(labels, E)
if E.size() == 0 return labels
else
    child := {rand_bit() : v \in [1...n]} // randomly become child/parent
    hooks := {(u,v) \in E | labels[u] != labels[v]} // edges with different labels
    labels := labels <- hooks // children adopt parent label through hooks
    E' := {(labels[u],labels[v]) : (u,v) \in E | labels[u] != labels[v]} // create
        smaller graph
    labels' := cc_random_mate(labels, E') // recursively solve on smaller graph
    labels' := labels <- {(u,labels'[v]) : (u,v) \in hooks} // propagate labels
        back through hook
return labels'
```

The re-expansion of the graph passes labels from each root of a contracted star to its children. The graph is contracted while going down recursion and re-expanded coming back up. Each child can have multiple hook edges but only one parent. For each hook edge, the parent's label is written into the star.

Coins are flipped on even already-contracted vertices.

Possible improvements: don't use all the edges for hooking on each step, just use a sample.

2. Deterministic graph contraction Form a set of disjoint subgraphs (each is a tree). Use point-to-root algorithm to contract subgraphs to a single vertex.

```
ALGOIRTHM cc_tree_contract(labels, E)
if (E.size() == 0) return labels
else
    hooks := {(u,v) \in E | u > v}
    labels := labels <- hooks
```

```
    labels := point_to_root(labels)
    E' := {(labels[u], labels[v]) : (u,v) \in E | labels[u] != labels[v]}
    return cc_tree_contract(labels, E')
```

Instead of parent/child, the hooks are selected by using pointer jumping. Point goes from larger numbered vertices to smaller numbered vertices. Worst case behaviour occurs when the maximum index is at the center of a star and all of its children are smaller than it is.

Possible improvements: interleave hooking steps with pointer-jumping steps. Tree is only partially contracted when executing each hooking step.

3.5.3 Spanning trees and minimum spanning trees

Definition 8. Spanning tree of connected graph G = (V,E) is connected graph T = (V,E') s.t. E' is a subset of E, and E'.size() = V.size() - 1. Cannot have any cycles and forms a tree.

When components are hooked together algorithm can keep track of which edges were used. Collection of all edges used in hooking (since only used once) will form a spanning tree.

For minimum spanning tree, random mate algorithm makes sure that it selects the minimum-weight edge. If it doesn't lead to a parent, the child does not connect and is orphaned.

3.6 Sorting

Focusing on two algorithsm: QuickSort and radix sort.

3.6.1 QuickSort

```
ALGORITHM: quicksort(A)
if A.size() == 1 then return A
i := rand_int(A.size())
p := A[i]
in parallel do
    L := quicksort({a : a \in A | a < p}) \\ less than
    E := {a : a \in A | a == p} \\ equal to
    G := quicksort({a : a \in A | a > p}) \\ greater than
return L ++ E ++ G
```

Can be further parallelized by selecting more than one partition element. With P processors, choosing P-1 partition elements divides keys into P sets, each of which can be sorted. Make sure to assign same number of keys per processor.

3.6.2 Radix sort

Not a comparison sort (does not compare keys directly to find relative ordering); instead, reprsents keys as b-bit integers.

Examines keys to be sorted one digit position at a time, starting with the least significant digit in each key. The output ordering of this sorrt must preserve the input order of any two keys with identical digit values in the position being examined.

Counting sort finds the rank of each key (position in the output order) then permutes the keys.

```
ALGORITHM radix_sort(A,b)
for i from 0 to b-1
    flags := {(a >> i) mod 2 : a \in A}
    notflags := {1-b : b \in flags}
    R0 := scan(notflags)
    s0 := sum(notflags)
    R1 := scan(flags)
    R := {if flags[j] == 0 then R0[j] else R1[j] + s0 : j \in [0...A.size()]}
```

```
    A := A <- {(R[j],A[j]) : j \in [0...A.size()]}
return A
```

Can also handle floating point numbers.

3.7 Computational geometry

Calculating properties of sets of objects in k-dimensional space (e.g. closest-pair of points, convex-hull, line/polygon intersections).

Parallel solutions often use divide-and-conquer or plane sweeping.

3.7.1 Closest pair

Set of points in k dimensions, returns two points that are closest to each other (Euclidean). Uses divide and conquer to split points along lines parallel to the y-axis.

```
ALGORITHM closest_pair(P)
if (P.size() < 2) return (P, inf)
x_m := median({x : (x,y) \in P})
L := {(x,y) \in P | x < x_m}
R := {(x,y) \in P | x > x_m}
in parallel do
    (L',delta_l) := closest_pair(L)
    (R',delta_r) := closest_pair(R)
P' := merge_by_y(L',R')
delta_p := boundary_merge(P',delta_l,delta_r,x_m)
return (P',delta_p)
```

where merge_by_y merges L' and R' along y axis, and boundary_merge does the following. Inputs are original points P sorted along y, closest distance within L and R, and median point x_m. CLosest distance in P must be eitehr delta_l, delta_r, or a distance between a point in L and R. The two points must lie within delta = min(delta_l, delta_r) of x = x_m, which defines a region in which the points must lie.

```
ALGORITHM boundary_merge(P, delta_l, delta_r, x_m)
delta := min(delta_l, delta_r)
M := {(x,y) \in P | (x >= x_m - delta) and (x <= x_m + delta)}
delta_m := min({min({distance(M[i], M[i+j]) : j \in [1...7]}) : i \in [0...P.size
    ()-7]})
```

3.7.2 Convex hull

1. Quickhull Quickhull does something similar to QuickSort in that it picks a "pivot" element, splits the data around the pivot, and recurses on each split set. Pivot element not guaranteed to split the data into equal sized sets, but is often quite effective.

Take points p in plane and p1, p2 that are known to lie on convex hull (e.g. x, y extrema) and return all points that lie clockwise from p1 to p2. Algorithm removes points that cannot be on hull because they are right of teh line from p1 to p2.

```
ALGORITHM subhull(P, p1, p1)
P' := {p \in P | left_of?(p,(p1,p2))}
if (P'.size() < 2) return [p1] ++ P'
else
    i = max_index({distance(p,(p1,p2)) : p \in P'})
    p_m := P'[i]
    in parallel do
        Hl := subhull(P', p1, pm)
        Hr := subhull(P', p, p2)
    return Hl ++ Hr
```

2. Mergehull

Assumes P is presorted according to x coordinates of points. Hl must be a convex hull on the left and Hr must be a convex hull on the right.

```
ALGORITHM mergehull(P)
if P.size() < 3 return P
else
    in parallel do
        Hl = mergehull(P[0...P.size()/2])
        Hr = mergehull(P[P.size()/2...P.size()])
    return join_hulls(Hl, Hr)
```

Can be improved by modifying the searhc for bridge points so they run in constant depth with linear work. Alternatively, use divide and conquer to separate point set into sqrt(n) regions, solving convex hull on each region recursively, then merging all pairs of those regions using binary search.

3.8 Numerical algorithms

3.8.1 Matrix operations

Matrix multiplication e.g. is highly parallelizable because each loop can be done simultaneously.

```
ALGORITHM matrix_multiply(A,B)
(l,m) := dimensions(A)
(m,n) := dimensions(B)
in parallel for i \in [0...l] do
    in parallel for j \in [0...n] do
        R_ij := sum({A_ik * B_kj : k \in [0...m]})
```

Arguably too parallel – a lot of research focuses on what subset of parallelization is actually needed. Matrix inversion is more difficult to parallelize, but can be done.

3.8.2 Fourier transform

Discrete Fourier Transform often solved using the Fast Fourier Transform algorithm, which is similarly easy to parallelize because of its loops. So, most work has gone into reducing the communication costs (butterfly network topology is called FFT network since the FFT has the same communication pattern as the network.)

```
ALGORITHM fft(A)
n := A.size()
if n == 1 return A
else
    in parallel do
        even := fft({A[2i] : i \in [0...n/2]})
        odd := fft({A[2i + 1] : i \in [0...n/2]})
    return {even[j] + odd[j]e^(2pi * i * j / n) : j \in [0...n/2]} ++ {even[j] -
        odd[j]e^(2pi * i * j / n) : j \in [0...n/2]}
```

3.9 Research issues and summary

Research recently on pattern matching, data structures, sorting, computational geometry, combinatorial optimization, linear algebra, linear and integer programming.

3.10 Paper Review: Parallel Algorithms

3.10.1 Motivation

Introduction to the design and analysis of parallel algorithms: algorithms that specify multiple operations on each step.

3.10.2 Key Ideas

There are several ways of modeling parallel algorithms. These models are necessary because, in the context of having multiple processors and memory modules, the assumption made in sequential models such as random-access machines (RAM) that basic CPU operations (arithmetic, logic, memory access etc.) only require one time step is no longer valid.

The first kind of model is called a multiprocessor model. This is closely related to the underlying hardware of a parallel processor, and takes into consideration the connectivity between processors and memory modules (e.g. is there an interconnection network? If so, what is its topology? Do processors have local memory modules or are they connected to memory through the network?) With the exception of parallel random-access machines (PRAM), a somewhat hypothetical form of parallel processor that assumes one shared memory that each processor can access in one time step, the organization of processors, memory modules, and network determine the cost of each CPU operation.

The second way of modeling parallel algorithms is called work-depth. This is more abstract, and studies an algorithm's potential to be parallelized. Under the work-depth model, W work is considered to be the total cost of the operations an algorithm performs. The maximum length of dependencies between these operations is called the depth D. The ideal parallel time that can be achieved with P processors for a given algorithm is therefeore $T_P \approx W/P + D$ where work is evenly spread out amongst the available processors, but cannot go faster than longest path of interdependent operations.

An algorithm that is designed for ideal conditions using work-depth model can be translated into a more hardware-conscious multiprocessor model in a way that preserves work and does not incur substantial additional slowdowns.

3.10.3 Results

The article describes several algorithms that are common in sequential form in parallel form. Oftentimes, parallelization is used when there are many simple and/or independent operations that can be performed at the same time (e.g. moving pointers in a linked list, or summing adjacent values in a list).

More complex are graph problems. Traditional BFS and DFS can be converted into parallel form but are not necessarily more efficient. Consequently, a variety of techniques such as graph contraction have been developed. The chapter focuses on two such algorithms, a random-mate and deterministic graph contraction. It then expounds on some of the applications of these algorithms to related problems such as finding the minimum spanning tree.

The author also discusses computational geometry and numerical problems. I'm more familiar with these kinds of algorithms so seeing the parallel versions was both interesting and seemed intuitive. I found it interesting that most of the work in the matrix operations world is focused not on finding clever ways of parallelizing, since those algorithms naturally lend themselves to parallel implementaitons, but on optimizing the communication costs i.e. figuring out when it's actually smarter not to parallelize. The little snippet linking the butterfly network architecture in hardware to the fast fourier transform was also quite beautiful.

3.10.4 Novelty

I don't know if anything here was particularly novel but it was certainly educational, and since many of these concepts are novel to me, arguably a success.

Thinking in Parallel

Author(s): [To be filled]

4.1 General Notes

4.2 Paper Review

4.2.1 Motivation

4.2.2 Key Ideas

4.2.3 Results

4.2.4 Novelty

4.2.5 Strengths/Weaknesses

4.2.6 Ideas for Improving Techniques/Evaluation

4.2.7 Open Problems/Directions for Future Work

Parallel Breadth-First Search on Distributed Memory Systems

Author(s): Aydın Buluç Kamesh Madduri

5.1 General Notes

- asymptotic complexity of parallel "level synchronous" BFS (O(D) where D is diameter of graph) is the same as serial algorithm because PRAM does not account for synchronization costs
- Key optimization directions:
  - parallelize edge visit steps, make sure parallelization is load-balanced
  - mitigate synchronization costs due to atomic updates/barrier synchronization after each level
  - improve locality of memory references by modifying graph layout and/or BFS data structures
- Multithreaded systems:
  - Bader and Madduri: ensure graph traversal is load-balanced to run on thousands of hardware threads (MTA-2 system has no cache)
  - GPGPUs rely on large scale multithreading to hide memory latency. High memory bandwidth, outperform CPU on low-diameter high-vertex/edge families.
- Multicore CPU systems:
  - Performance is dependent on graph size
  - From Agarwal et al. : atomic intrinsics are a problem. Partition vertices of the graph and their edges among multiple sockets. Local vertices are updated atomically, nonlocal vertices are held back to avoid coherence traffic.
  - Xia and Prasanna : low-overhead "adaptive barrier" adjusting number of threads participating in traversal based on estimated amount of work to be performed
  - Leiserson and Schardl : shared queue replaced with "bag" data structure
- Distributed memory systems
  - Use level-synchronous approach
  - 'visited' checks replaced by edge-aggregation-based strategies (processor cannot tell if nonlocal vertex has been visited or not. Accumulate all edges corresponding to nonlocal vertices and send to owner processor at end of local traversal.)
  - Scarpazza et al. : 2d graph partitioning scheme limits key collective communication phases to at most sqrt(p) processors. Assume regular degree distribution. Compute time increases up to 10x with increasing processor counts (sequential kernels, data structures not work-efficient)
- External memory algorithms: random access is expensive. External memory algs use I/O-optimal strategies for sorting and scanning
- Other parallel BFS: fastest known algorithm in PRAM complexity model repeatedly squares adjacency matrix of graph, requires $O(n^3)$ processors

- Other related work: graph partitioning intrinsic to memory graph algorithm design (bounds inter-processor communication traffic). Sparse graph can be viewed as sparse matrix, can use linear algebra methods.

5.2 Paper Review Notes

5.2.1 Motivation

- efficient RAM algorithms don't necessarily translate to modern (parallel) architectures
- current architectures – prioritize efficient computation of regular computations with low memory footprints, penalize memory-intensive code with irregular memory accesses
- BFS algorithm is irregular because graph structure changes
- design and parallel performance of data-intensive graph algorithms is understudied

5.2.2 Key Ideas

1D partitioning

- Each processor P owns n/p vertices and their outgoing vertices
- $\approx$ 1d partitioning of adjacency matrix
- multiple threads can list the neighbours of the frontier vertices, so they have to send newly discovered vertices to the owner process (all-to-all communication step)
- c.f. serial version that only needs local compute – distributed requires message buffers of size O(m), and all-to-all communication needs to be done at each BFS level
- Partitioning of frontier vectors follows the vertices

2D partitioning

- each BFS iteration is computationally equivalent to sparse matrix-sparse vector multiplication
- A: adjacency matrix (sparse boolean); x_k: k-th frontier (sparse boolean vector with integer variables)
- Vertex ownership is more flexible than in 1d
  - Distribute vector entries over only one processor direction (rows, cols), e.g. diagonals, or first processor in row
  - Let each processor have same number of vertices (2d vector distribution matches matrix distribution), and each processor row is responsible for $t = [n/p_r]$ elements. Last processor row gets remaining $n - [n/p_r](p_r - 1)$ elements i.e. i'th row is responsible for vertices numbered v_{ip_r+1} to v_{ip_r}.

5.2.3 Results

Method

- graphs are sparse
- average path length is small constant value compared to n vertices (upper bounded by log n)
- Use CSR representation (all adjacencies of vertex sorted and stored in contiguous chunk of memory, contiguously to the next vertex). Too wasteful for storing sub-matrices after 2D partitioning, however, so use Doubly-Compressed Sparse Columns (DCSC) for hypersparse matrices after 2D partitioning that uses column pointers and column IDs to index an array of m row ids.

5.2.4 Novelty

- Two approaches to distributed-memory BFS with skewed degree distribution:
  1. one-dimensional distributed adjacency arrays for representing graph
  2. sparse matrix representation and 2-d partitioning among processors
- 2d partitioning approach + intranode multithreading reduces communication overhead by 3.5x
- single-node performance of approach ¿= single-node shared memory results
- updating multicore systems with modest levels of thread-level parallelism:
  1. thread local stacks store newly visited vertices and are merged to form next frontier (memory requirement bounded by O(n)); copying does not have a large overhead
  2. cache coherence ensures that when distance value is written at given level the correct value propagates ("benign races" for insertions)

5.2.5 Strengths/Weaknesses

5.2.6 Ideas for Improving Techniques/Evaluation

5.2.7 Open Problems/Directions for Future Work

5.3 Paper Review

5.3.1 Parallel Breadth-First Search on Distributed Memory Systems

This paper is generally motivated by the fact that parallel BFS and its implementations on modern computer architectures is understudied, particularly in applications that involve irregular graphs. RAM (random access machine) models for algorithm design neglect the computational overhead of synchronizing and passing data between multiple processors.

As BFS is a key fundamental algorithm for graph operations, particularly in parallel systems where the naturally serial DFS is not as eligible for parallelization, much work has been invested into improving the efficiency of BFS. There have been meaningful advances in designing BFS for specific hardware systems (e.g. the massively parallel MTA-2 system used by Bader and Madduri), as well as for distributed memory systems that require intelligent partitioning of the graph to avoid coherence traffic and duplicate work (Scarpazza et al.). The most effective of these methods, however, operate on graphs with constant degrees, which do not reflect the typical distribution of graphs found in the wild. Another outstanding challenge is all-to-all communication at each level in BFS in order to synchronize knowledge about newly found vertices amongst the processors, which is especially expensive on distributed systems.

The novel contributions of this paper focus on improving distributed-memory BFS on graphs with irregular degree distributions, and are twofold. First, the authors present a one-dimensional distributed adjacency array for representing graphs. Second, they offer an alternative 2-dimensional partitioning that can be used amongst multiple processes.

I thought the discussion about potential bottlenecks in section 4.2 was very interesting. I would have expected that the synchronization and copying of the new frontier stack FS (i.e. merging thread-local stacks at every level) would have induced greater overhead than 3% of execution time, and that cache coherence would have been a greater issue for updating the distance array. However, the barrier and memory fence ensure that the correct distance is written and propagated to all the cores, making these "benign races" on insertions (into per-thread new frontier stacks NS_i) indeed benign. In the 2D case, use of the sparse accumulator in forming $\bigcup A_{ij}(:, k)$ for all k where $f_i(k)$ exists (i.e. unioning neighbour columns of A for the frontier to build the new frontier) is also fascinating, since I would not have anticipated a dense vector of values to perform better than a method that works better than the multiway merge, which, given the previous readings about parallel algorithms, seems like it would have been the preferable option (more memory efficient, sorted output).

The following sections talk more about memory efficiency, particularly focusing on the distinction between local and network memory accesses. Since, in the distributed approach, the array size for distance array checks (which make up a large fraction of the runtime) are reduced by a factor of p processors, multithreading has meaningful performance benefits. Similar tenets hold for the 2D case, but the working sets are larger (p_c and p_r are both smaller than the total p).

The experimental studies seem to align with the requests that Johnson makes in A Theoretician's Guide to the Experimental Analysis of Algorithms, 2001, in that the hardware and experimental conditions are clearly described and likely replicable. A comparison is made to previous implementations in order to situate the work in the literature, and are open about failures (inability to compile on Cray machines). They also reveal unexpected findings, for instance that on some architectures, the 1D algorithm perform 1.5-1.8x as fast as the 2D (Franklin), but that when scaling up, the 2D performs faster due to reduced time in communication (Hopper).

If we go up to section 2.2, the authors state that prior work has primarily optimized by ensuring the parallelization of edge visit steps is load balanced, synchronization costs are mitigated, and that locality of memory references is improved. Although the authors are implementing a novel approach, they do pay attention to load balancing (randomly relabeling vertices), reducing synchronization overhead (the "benign races" as a mechanism for merging local thread frontier stacks), and improving locality of memory references (CSR for 1D case, DCSC for sparse 2D matrices).

It would be interesting to see what happens when higher-diameter graphs are run on this algorithm, and whether any of these methods can be further optimized to create hardware specific solutions.

A Simple and Practical Linear-Work Parallel Algorithm for Connectivity

Author(s): Julian Shun, Laxman Dhulipala, Guy Blelloch

6.1 General Notes

6.2 Paper Review

6.2.1 Motivation

Graph connectivity is an important problem. Sequentially, it's manageable in linear work. So far, parallel algorithms either (a) require super-linear work or (b) are very complicated (poly-logarithmic-depth).

Graph connectivity : labelling vertices in a graph to distinguish between those that belong to one connected component or other.

Past approaches

- Sequential: BFS and DFS have linear work
- Simple but super-linear work parallel algorithms:
  - Shiloach and Vishkin, Awerbuch and Shiloach, combine vertices into trees so that at end of algorithm vertices in the same component belong to the same tree. Number of trees decreases by constant factor each iteration, but constant fraction of edges is not guaranteed to be removed. $O(m \log n)$ work.
  - Reif and Phillips contract vertices in same component together, constant fraction of vertices decrease per iteration, do not guarantee that constant fraction of edges are removed. Also $O(m \log n)$ expected work.
- Parallel BFS: linear work, but depth is proportional to sum of diameters of connected components. Not efficient except for low-diameter graphs with few connected components.

6.2.2 Key Ideas

First work-efficient parallel graph connectivity algorithm with an implementation

- Based on parallel algorithm for generating low-diameter decompositions of graphs (Miller et al.). N edges between partitions is also small. Diameter is $O(\log n/\beta)$, n edges between partitions is $O(\beta m)$ for $0 \leq \beta \leq$. Linear work, $O(\log^2 n/\beta)$ depth with high probability. Performs BFS from multiple sources in parallel, start times drawn from exponential distribution. BFS searches need to be run for at most $O(\log n/\beta)$ iterations before all vertices are visited.
- To label all components, decomposition algorithm is called recursively. β is constant.
  - Each call contracts partitions into single vertex
  - Vertices and edges between partitions are relabeled

Method: Concurrent-read concurrent-write (CRCW) PRAM

- Work: equal to number of operations required

- Depth: number of time steps needed

Decomposition:

- The (β, d)-decomposition algorithm by Miller et al. is a parallel decomposition based on parallel BFS. Resulting partitions V have small diameter (any two vertices are at hop distance $\leq d$) and the number of edges with endpoints in different clusters is $\leq \beta m$ i.e. β is how many edges we want to cut.
- Every vertex v has shift value δ_v drawn from exponential distribution. This random value determines the start time of that vertex's BFS
- v is assigned parition S_u that minimizes shifted distance dist(u, v) - δ_u
  - Multiple BFS
  - At each iteration t (starting with $t = 0$) start BFS's in parallel from unvisited vertices v such that $\delta_v \in [t, t + 1]$.

Algorithm 1: CC using DECOMP: i.e. connected components using decomposition

- Input undirected graph $G = (V, E)$
- Output labels of connected components
  - Parameter β controls how aggressively edges are cut
  - L = DECOMP$(G(V, E)\beta)$ to partition V into hop-diameter partitions using random exponential shifts; each cluster gets labelled
  - Contract graph so that each cluster becomes a super-vertex in V'. If an edge in G crossed clusters, then it becomes an edge in E'.
  - If $E' == \emptyset$ then there are no edges in clusters and each cluster is fully connected
  - Run CC recursively on smaller graph $G'(V', E')$ and returns labels L' for super-vertices.
  - Each vertex in cluster C inherits cluster's super-label
- Option of using arbitrary decomp which rounds down the δ_v values and makes arbitrary tiebreaking decisions
- When BFS examines edge, check on-the-fly if intra-component or inter-component. Ignore intra-component, only carry inter-component forward

Implementation:

- Three versions: Decomp-min, decomp-arb, decomp-arb-hybrid
-

6.2.3 Results

- Over 40 cores, implementations are 18-39x faster over same implementation on single thread

6.2.4 Novelty

6.2.5 Strengths/Weaknesses

6.2.6 Ideas for Improving Techniques/Evaluation

6.2.7 Open Problems/Directions for Future Work

6.3 Paper Review

6.3.1 A Simple and Practical Linear-Work Parallel Algorithm for Connectivity

This paper describes the first parallel algorithm for identifying connected components in a graph that is work-efficient and can be reasonably implemented. It builds off literature that describes (a) parallel algorithms that are not theoretically work-efficient or (b) highly efficient theoretical algorithms that would be very challenging to practically implement. As such, the paper fits perfectly into the corpus of Algorithm Engineering, a positioning which is further reinforced by the blend of theory and experimental results presented in the work.

The idea underpinning the algorithm is quite elegant. It uses Miller et al.'s (β, d)-decomposition as a subroutine to partition the graph into small diameter clusters (any two vertices in the cluster are at most hop distance d away from each other). The number of edges connecting clusters is less than βm. In other words, β, a fraction between 0 and 1, tells us what proportion of edges we are willing to cut in order to create distinct clusters. A graph so decomposed is then collapsed into a smaller graph $G'(V', E')$ where each V' represents a cluster, and each E' represents one of the at most βm edges in between the clusters. The goal, to be accomplished through recursive applications of (β, d)-decomposition (DECOMP) to the graph, is to fully collapse the graph so that individual connected components are represented by only one super-vertex V''. The label L'' given to V'' is finally applied to all of the vertices that have been collapsed into that super-vertex.

An important practicality consideration that the authors make is using arbitrary decomposition, which uses rounded δ_v instead of fractional tie-breaking. Although it has the same asymptotic guarantees as regular DECOMP, the expectation of cut edges doubles to $2\beta m$. Other optimizations include in-place/on-the-fly filtering of edges that will be carried forward during the BFS. Intra-component edges are no longer needed by the algorithm and therefore culled. Memory is also saved by using a single structure C for storing both the component ID and resolving conflicts. There is some inefficiency inherent in parallel BFS as multiple nodes might write to the same child node at the same time, leading to a conflict that is resolved at the next barrier. The barrier effectively maintains synchronicity and also allows intra-component edges to be dropped in the min-decomp version of the code. The arbitrary version of the decomposition algorithm avoids the synchronization barrier and only stores one integer per vertex (its component ID).

A last note on implementation is made with respect to hybrid BFS, which implements bottom-up search per Beamer et al. when the frontier is large. Although the connected cluster algorithm must inspect every edge in order to determine if it is inter- or intra-cluster, using cache-friendly read-based computation when the frontier is large does improve performance.

The algorithms were tested on six graphs (random 5-degree graph, two power-law degree distribution graphs of differing densities, a 3D-grid graph structured like a lattice, a straight line, and a social network graph com-Orkut). The performance of the serial versions of these algorithms varies depending on the graph they are applied to. In general, arbitrary decomposition which requires only one pass over the edges of the frontier wins out over regular decomposition. When the frontier grows large the hybrid algorithm is beneficial, as expected. Parallelizing the algorithms achieved up to a 13x speedup, and is robust across graph types.

One of the clear strengths of this paper is its novelty. It is the theoretical work-efficient algorithm with polylogarithmic-depth and an implementation of its kind. The paper also provides a simple, comprehensible version of the algorithm alongside some clever optimizations that improve its practical runtime. The results suggest, as might be expected, that this algorithm works much better on small-diameter graphs than on large-diameter graphs, and I wonder what its performance on more real datasets (e.g. Wikipedia, or the Flickr dataset we discussed last week) would be. Also, the choice of β is and remains a little arbitrary.

I think future work could lead in the direction of additional experiments with real-world graphs. A way that one might be able to build on this paper is finding some novel implementation (what real-world problems are there that would benefit from finding disconnected elements quickly? I imagine there are some interesting applications in finite element analysis, which suffers from numerical instability when there are disconnected mesh elements.)

Shared Memory Parallelism can be Fast and Scalable

Chapters 7-8

Author(s): Julian Shun

7.1 Chapter 7: Ligra

A Lightweight Graph Processing Framework for Shared Memory

Origins

- Many packages developed for processing large graphs in parallel (including distributed)
- Ligra is especially good at graph traversal
- Based on hybrid BFS by Beamer et al. (2011, 2012); sparse representation of vertices when frontier is small and dense representation when large
- Ligra gets close to same memory/time efficiency and is simpler than Beamer et al.

Ligra

- Designed for shared-memory machines (comm. costs are cheaper than distributed memory). Less capacity but big enough for most relevant graphs (100 billion edges)
- Atomic compare-and-swap instruction deals with race conditions
- Two data types:
  1. Graph $G = (V, E)$
  2. Subsets of vertices v (vertexSubset)
- Two functions (that can also be used on subsets):
  1. mapping over vertices (vertexMap)
  2. mapping over edges (edgeMap)
- BFS made using increasing vertexSubsets that represent the frontier
  - Allow edges to be processed in different orders depending on need
  - Edge traversal includes looping over each vertex' out-edges, or looping over destination vertices sequentially/in parallel and checking if in-edges are in frontier
- Between-ness centrality (what is the "importance" of a vertex) is BFS that accumulates statistics along the way and needs both forward and backward traversal. Easy in Ligra because vertexSubsets are stored each iteration during forward traversal
- In-edges and out-edges are stored as individual arrays. When flipping traversal direction, just switch the role of the arrays. When undirected or symmetric graph, just store one copy of the arrays. When weighted, array is interleaved edge target and weights for cache efficiency [N1, W1, N2, W2, ..., Nm, Wm]

- EdgeMapSparse vs EdgeMapDense
  - EdgeMapDense: (bottom-up) called when number of outgoing edges of vertexSubset $U \in V$ is greater than some threshold (default is $m/20$). Loops through all $v \in V$ in parallel and for each vertex applies $F(ngh, v)$ for each of v's incoming neighbors ngh that are in U until $C(u)$ returns false. Returns dense representation of vertexSubset. Good for large vertexSubsets. Since F is applied serially, EdgeMapDense is not atopic w.r.t. the target vertex. Read-based.
  - EdgeMapSparse: (top-down) loops through all vertices in U in parallel, for given $u \in U$ applies $F(u, ngh)$ to all outgoing neighbors in G in parallel. Returns sparsely represented vertex subset. Work is proportional to $|U| + \sum_{u \in U}(\text{out-degree}(u))$. Good for small vertexSubsets.

Applications of Ligra

- BFS
- Between-ness centrality
  - Tells us relative importance of vertices in a graph (e.g. betweenness centrality index)
  - For graph $G = (V, E)$ and $s, t \in V$ let $\sigma_{st}(v)$ be n shortest paths from s to t that pass through v. Therefore $\delta_{st}(v) = \sigma_{st}(v)/\sigma_{st}$ is the pair-dependency on v.
  -

A functional approach to external graph algorithms

Author(s): J. Abello, A. L. Buchsbaum, J. R. Westbrook

8.1 General Notes

8.2 Paper Review

8.2.1 Motivation

- Classical algorithms do not scale when data exceeds main memory limits
- Need to extend to external memory but not always easy with RAM model
- Graphs do not have data locality required for efficient external-memory extensions

8.2.2 Previous approaches

PRAM simulation (Chiang et al.)

- Simulate CRCW PRAM algorithm using one processor and external disk to give general method for constructing external graph algorithms from PRAM graph algorithms
- Simulation maintains copy of main memory in array A. Array sorted by memory address. Also state array T of N elements. Location T[i] contains current state of processor i
- PRAM step:
  1. D = list of tuples (d(i), i) where d(i) is memory address and i is processor number. Sort D by memory address (group all processors that want to read from the same location)
  2. Scan and create read list R, list of current values. R is tuples (r(i), i) where r(i) is value and i is processor number
  3. Sort R by processor number i (prioritize processor 1 over 2, e.g.)
  4. Scan through R. For each processor i T[i] ¡- r(i) i.e. the value it read in this step
  5. Results get back to the right processors in order. Converts parallel operations into sequential operatons that can be run when there is no parallel hardware available.

Buffering data structures

- Buffer trees: sequences of insert, delete, deletemin operations on N elements in $O((1/B)\log_{M/B}(\mathrm{N/B}))$. Better than O(N) I/Os for individual processing.
  - Internal node has a buffer storing pending operations (buffer size cM for some 0 ¡ c ¡1)
  - Leaves contain data
  - When buffer fills up, operations are pushed down to child nodes
  - Operations are processed in batches, trickle down tree as buffers fill up
- Tournament trees: maintain elements 1 to N subject to delete, deletemin, and update. Update takes (x,k) and sets key of x to minkey(x), k

  - Internal nodes represent tournaments between its children (winner is promoted)
  - Deleting a node replays tournaments along path from deleted leaf back to root
  - Good for algorithms that need to find and remove minimum elements when data is toolarge for main memory (updates only require traversing one path in the tree)
- Not functional: tree-structures require in situ replacement of nodes on disk
- Could copy, but significant I/O overhead to change pointers to point to new complexity

8.2.3 Key Ideas

- Divide-and-conquer approach for designing external graph algorithms without special data structures
- Functional: each algorithm is a sequence of functions applied to input data and producing output data. Information remains unchanged.
- I/O model
  - M = number of items that can fit in main memory (10e9)
  - B = number of items per disk block (10e3); number of items that can be read/written in single I/O operation; O($Nlog^i(N)$)
  - N = number of items in instance
  - In general, 1 ¡ B ¡ M/2 ¡ M ¡ N
  - Functional I/O only makes functional transformations (no transformations to data)
- Primitives (scanning, sorting)
  - scan(N) = ceil(N/B) – reading N contigiuous items from disk, cost is ceil(N/B) I/O operations (need to read that many blocks to get all N items)
  - sort(N) = $\Theta(\text{N/B} \cdot \log_{M/B}(\text{N/B}))$ – sorting N items using external memory. N/B is number of blocks of data, M/B is number of blocks that fit in memory simultaneously.
  - Once data is in RAM (main memory) computation is free. Only disk operations matter for complexity.
  - Goal is to replace traditional N with N/B (process whole blocks at once), $\log_{M/B}$ instead of $\log_2$ i.e. dividing problems into M/B subproblems at the same time (grows slower than log)

Functional graph transformations *Connected components*

- Graph G of rooted stars, if root of v $r(v) == r(u)$ then v and u are in the same
- Contraction is performed using new supervertex to delete x, y that absorbs all of their edges.
- Lemma 3.1: if each pair in $E\prime$ contains two vertices in the same connected component in G, for any $u, v \in V$, u and v are in the same connected component in G if and only if $s(u)$ and $s(v)$ are in the same connected component in $G\prime$ (which is $= G/E\prime$) i.e. contraction preserves connectivity information and only safe contractions are allowed.
- Divide-and-conquer reduces problem size at each step through contraction, to ultimately create a problem that can be solved in RAM/fit in memory PLUS random-access graph problem is turned into a structured problem using stars

Functional approach to designing external graph algorithms

1. Apply selection function to G to get S(G) to get G1 *subset of half the edges of G*
2. Combine G and solution fp(G1) *forest of rooted stars corresponding to connected components of G* using transformation T1 to get subgraph G2 *contraction of G with respect to connected components of S(G)*

3. Map solution from fp(G1) and fp(G2) to solution in G to get fp(G) using T2 *re-expansion of rooted stars*

- Transformations (e.g. contractions) must preserve solutions and non-solutions. You can't have non-planarity in a contracted graph
- Functional if S, T1, T2 can be implemented without side effects on inputs

Deterministic algorithms

8.2.4 Results

-

8.2.5 Novelty

8.2.6 Strengths/Weaknesses

8.2.7 Ideas for Improving Techniques/Evaluation

8.2.8 Open Problems/Directions for Future Work

8.3 Paper Review

8.3.1 A functional approach to external graph algorithms

Abello, Buchsbaum, and Westbrook's paper presents, as the title helpfully indicates, a functional approach to external graph algorithms. It is characterized by the concision and power of the functions that are exposed to the user, including primitives (scan, sort, bucket) and transformations (contract, relabel, select). This approach differs from previous methods, which typically operate using in-place modifications of the graph, by completely avoiding mutation of the input data. This property is desirable in external memory algorithms because the bulk of running time is consumed by I/O operations that read and write to disk (in fact, they contribute so much to complexity that the authors approximate operations on RAM as "free" relative to the cost of I/O operations.)

Central to the challenge that the paper addresses is how primitives batch the data so that B items exist in a single disk block, that is, the number of items that are written or read per I/O operation. In the use-cases this method is designed for, value B is generally less than the number of items in main memory M, which in turn is less than the total number of items in an instance N. For example, the simplest exposed primitive `scan(N)` takes $\lceil N/B \rceil$ I/O operations to read all N items, as it reads from disk in batches of size B. In general, these methods allow us to reduce N I/O cost to N/B, and to reduce the number of I/O passes to $\log_{M/B} N/B$ instead of $log_2 N$. [1]

The algorithms presented in this paper work by recursively reducing the size of the problem until it fits in main memory and computations can be efficiently performed on the reduced graph using RAM. Once performed, the graph is expanded out again. This divide-and-conquer approach can be applied to many algorithms in the literature, including connected components, minimum spanning forests, bottleneck minimum spanning forests (first implementation for external memory), maximal matching (first implementation for external memory), and maximal independent sets (first implementation for external memory). All of these algorithms operate on properties that are retained over the course of recursive contraction. The authors give the example of an algorithm that requires information about graph planarity as something that could not be processed by their method, as local differences in planarity would be lost as the graph is contracted.

In addition to being novel, the implementations are competitive with other examples in the literature. Deterministic connected components and maximum spanning forests slightly improve over the asymptotic I/O bounds of earlier work, and the randomized algorithms at least match earlier I/O bounds. The semi-external model, which assumes that (much like social networks and other real-world graphs) the number of edges can sometimes far exceed the number of vertices, so stores vertices in local memory but does not store edges, further improves on these bounds.

I appreciate the intuition of the paper's solution – take existing graph problems and circumvent I/O overhead by doing as little external memory work as possible, and batching the data that you do read/write so that you perform as few I/O operations as possible. The authors also present a compelling suite of applications that demonstrate how rich their implementation is.

The method is tantalizing and introduces many avenues for future work around the implementation of other algorithms (shortest path etc.) in a functional framework. I would also love to see the semi-external applications applied to real-world, large-scale, small-world networks. It seems as though there could be demonstrable and dramatic improvements to previous external-memory solutions to parsing these large networks.

[1] The authors have an interesting discussion about the way parallel algorithms can be simulated on serial hardware using buffering data structures.

The Input/Output complexity of sorting and related problems

Author(s): Alok Aggarwal and Jeffrey Scott Vitter

9.1 Paper notes

The motivation for this paper is drawn from the real-world need for extremely fast sorting algorithms over extremely large datasets (on the order of tens of millions of records). 1/4 of all computer cycles are taken up by sorting, and most of those cycles are taken up by I/O between internal memory and secondary storage.

The problem can be solved by relaxng the program requirements and introducing parallel/distributed processing, or to improve our understanding of the actual, theoretical limits underpinning sorting. The authors take the latter approach and set up the problem with the following parameters: N records $(R_1, R_2, ..., R_N)$ to sort, M records that can fit into internal memory, B records per block, and P blocs that can be transferred simultaneously. With this setup, records can be transferred in parallel within each block, and blocks can also be transferred in parallel.

This setup is used in order to find theoretically optimal bounds for five problems: sorting, FFT, permutation networks, permuting, matrix transposition.

Sorting: internal memory is empty

Cache oblivious algorithms

Author(s): Matteo Frigo, Charles Leiserson, Harald Prokop, Sridhar Ramachandran

10.1 General Notes

Cache oblivious algorithms

- some algorithms are designed for fixed cache sizes; cache-oblivious algorithms are optimal for caches even with unknown parameters M and B
- M is the cache size and B is the block size (how long is each line of cache)
- We assume a tall cache (that is, $M = \Omega(B^2)$) where big-Ω is the asymptotic lower bound.

Example: cache-aware matrix multiplication

- Ordinary matrix multiplication takes $O(n^3)$ work and scans across whole rows and whole columns
- We split each matrix into $s \times s$ blocks and multiply the corresponding blocks with each other $(A_{ik}, B_kj, C_{ij}, s)$ where C is the matrix we're writing into. We want to tune s so that all three $s \times s$ blocks of A, B, C can fit into cache.
- One tile needs (tight asymptotic bound) $\Theta(s^2/B)$ lines of cache = $\Theta(3s^2/B)$ for all three tiles. We have M memory to store $3s^2$ elements so $\Theta(s^2) <= M$.
- There are $\Theta(s^2/B)$ cache misses required to load all the tiles and $(n/s)^3$ calls to get data (three-layer for loop). Total cache misses = $\Theta((\frac{n}{s})^3 \times \frac{s^2}{B}) = \Theta(\frac{n^3}{sB})$. If $s = \sqrt{M}$ then $\Theta(\frac{n^3}{B\sqrt{M}})$ total cache misses, which is $< \Theta(n^3/B)$.
- Many multiplications reuse the cache block before it is removed from cache.

Example: cache-oblivious rect-mult

- $C \leftarrow C + AB$ where $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, $C \in \mathbb{R}^{m \times p}$. If $C == 0$ then the output is a matrix product AB.
  1. $m \geq max(n, p)$: $\begin{pmatrix} C_1 \\ C_2 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$ (2 recursions)
  2. $n \geq max(m, p)$: $C = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2$ (first $C \leftarrow C + A_1 B_1$ then $C \leftarrow C + A_2 B_2$)
  3. $p \geq max(m, n)$: $\begin{pmatrix} C_1 & C_2 \end{pmatrix} = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix}$ (2 recursions)
- Once a subproblem fits into a cache, it can be solved without any more cache misses
- Base case: recursive tiling where the largest of m, n, p gets halved and loaded. Has same cache complexity bound as before $s = sqrtM$. No need to specify s manually.
- Ideal case: $m, n, p \leq \alpha\sqrt{M}$. Matrices are stored as $\Theta(1 + mn/B + np/B + mp/B)$ cache lines. Cache complexity $\mathrm{Q} = 1 + (mn + np + mp)/B$.

10.2 Paper Review

10.2.1 Cache oblivious algorithms

This paper builds on the I/O discussion we began last week. The motivation driving both of last week's papers was that I/O operations cost many orders of magnitude more than internal memory operations, a constraint that inspired the development of several algorithms that efficiently batch read/write operations in cases that require a lot of storage to external memory (e.g. graph operations on graphs with billions of edges).

Cache-oblivious algorithms by Frigo, Leiserson, Prokop, and Ramachandran from MIT's Laboratory for Computer Science (published 2012) propose a new approach toward designing algorithms that that efficiently modulate between cache and main memory. Rather than limiting themselves to cache-aware algorithms that are manually fine-tuned to the parameters of a given cache (M size of cache and B block size, or length of any line of cache that is written/read from main memory at the same time), the authors propose cache-oblivious algorithms that are provably efficient on any cache size. They assume that the cache is tall $M = \Omega(B^2)$.

The primary mechanism which drives many of the cache-oblivious algorithms is a dynamic partitioning of input matrices to fit in memory combined with divide-and-conquer solving. Typically, the same amount of arithmetic work is performed as in the naïve case (matrix multiplication still requires $O(n^3)$ work), but minimize the I/O operations by placing data contiguously into main memory in such a way that it can be re-used for multiple iterations of the algorithm (e.g. multiplications).

My favorite section of the paper describes funnelsort, a new type of sorting algorithm that is cache-oblivious (mergesort is criticised for being, in the classical case, oblivious to cache misses, and in the cache-optimized case introduced by Aggarwal and Vitter, cache-aware).

Beginning with regular mergesort to describe the difference, each of the two arrays being sorted is scanned once (n items fetched, so $O(n/B)$ cache misses in the worst case). Subsequent levels of merge also have $O(n/B)$ cache misses since the total elements merged at each level remains n. Over $\log n$ levels of merging, this gives a total $O(\frac{n}{B} \log n)$ cache misses. In the more optimal cache-aware multiway-merge sort, the cache complexity is reduced to $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$.

Cache-oblivious funnelsort works similarly but utilizes a k-merger to perform the merge operation. Rather than iteratively merging k sorted sequences, the k-merger begins with smaller $sqrt(k)$ sequences and recursively builds longer runs. To give an example over two levels, $sqrt(k)$ sequences are sorted and channeled into output buffers. Once an output buffer is sufficiently full ($2k^{3/2}$ elements), the k-merger suspends operations on that run, allowing data to remain in the cache and be reused. A final k-merge is performed over the $sqrt(k)$ buffer outputs, and this is the final result. This algorithm is work efficient ($O(n \log n)$), and after a lot of algebra, can be proved to be cache-efficient in accordance with Aggarwal and Vitter's result as well.

The paper ends with a number of rebuttals to anticipated challenges to its fundamental assumptions. This includes a justification for using LRU as a tactic for optimal replacement (it's competitive!) The authors also argue that the two-level cache structure is a reasonable model even for multi-level caches. I want to spend some time on this because, as I was reading the article, this appeared to be one of its major shortcomings in a world where real hardware *does* consist of multi-level caches. The authors argue that a multi-level cache using LRU replacement (a) has the property that each level of cache is a subset of the memory stored in the next higher level of cache (inclusion) and (b) will respond to cache misses in level i cache with a hit in higher levels of cache using a sequence of memory accesses comparable to a single-level cache. This is theoretically convincing but probably not how it practically works out. Although the authors show that LRU can be implemented, if a CPU uses pseudo-LRU, random, or tree-based replacement the assumption about inclusion breaks. Additionally, access costs in terms of latency and bandwidth to different levels of cache are not uniform and not discussed.[1]

The empirical results are definitely compelling, but I would like to see a comparison to cache-aware algorithms as well. Comparing to the naïve implementation shows that the cache-oblivious algorithms work well, but it would also be great to show the comparison to tuned cache-aware algorithms (both on the hardware they were tuned for and that which they were not tuned for), which could be a robust argument in favor of the generality of the authors' method.

Last, it would be interesting to learn whether the algorithms work on SSDs, which experience write amplification and have a limited amount of write cycles before they wear.

Engineering a cache-oblivious sorting algorithm

Author(s): Gerth Stolting Brodal, Rolf Fagerberg, Kristoffer Vinther

11.1 General Notes

Many results have built on Friggo et al.

- Often proven under the tall-cache assumption $M \geq B^2$
- Many theoretical results but few empirical (those that do show that cache-oblivious algorithms ¿ classic RAM and are competitive with cache-aware algorithms)
- k-merger does not need a specific memory layout. Sizes of buffers not layout in memory are critical feature

This paper

- Cache-oblivious sorting algorithm (faster than Quicksort for input sizes that fit into memory)

11.2 Paper Review

This paper presents an implemented version of Frigo et al.'s cache-oblivious search. One of the first points that they address is the theoretical optimality of using a two-level I/O model for multilevel memory hierarchies. They argue that if I/O operations are made by optimal cache replacement strategies, then the analysis holds for all levels of multilevel memory, and the algorithm will be optimized to all levels of cache.

Engineering in-place shared-memory sorting algorithms

Author(s): Michael Axtmann, Sascha Witt, Daniel Ferizovic, Peter Sanders (Karlsruhe)

12.1 General Notes

- Want to write a sorting algorithm quicker than quicksort, that works for arbitrary data types, is portable to different processor architectures and operating systems
- Tested over 500,000 different configurations
- Goal to do in-place (use constant space in addition to input)

Super-scalar samplesort

- Two temporary arrays of size n : one to store buckets, one *oracle array* to store bucket indices where input elements are placed
- Sampling phase sorts $\alpha k - 1$ randomly sampled input elements where oversampling factor α is tunable
- Splitters $S = [s_0 \dots s_{k-2}]$ are picked equidistantly from sorted sample. Classification stores target bucket indices in oracle array, increases size of bucket. Start with bucket bounds $-\inf$ *and* $\inf$
- Use prefix sum to calculate bucket boundaries
- Use decision tree for element classification to elminate branch mispredictions (binary search tree)

12.2 Paper Review

12.2.1 Engineering in-place shared-memory sorting algorithms

As this paper makes very clear at the beginning, it is very easy (and apparently quite common) to claim you have the "best" sorting algorithm but difficult to definitively prove that this is the case. The authors of this paper not only claim that their in-place shared-memory sorting algorithm, which is based on superscalar (out-of-place) samplesort, but perform 500,000 tests on different configurations of processor architecture and operating system to establish its performance on a wide range of problems compared to other, established sorting methods. The solution is also remarkable for being an implementation of a strictly in-place sorting algorithm, to which the authors point out many theoretical solutions, but of which most are quite challenging to transfer into practice.

The algorithm's antecedents include quicksort, samplesort, and superscalar samplesort, all of which fall in the same lineage and have been incrementally improved over time. Samplesort is an improvement over quicksort because, instead of relying on a single partition and recursing, it randomly samples $k-1$ elements from the list to be sorted and uses these as pivots for k buckets, which are then recursed over until the bucket size is small enough to use insertion or heap sort. This has better guarantees for consistent bucket size than quicksort, but can be inefficient when it mispredicts branches (for each element, the algorithm has to find the appropriate bucket, which is done with binary search and can lead to breaks in parallel processing if incorrect), has a cache miss when writing to buckets, or if the input array has many duplicates (which can be problematic for bucket size). Superscalar samplesort is, in turn, an improvement for samplesort because it implements a binary tree for the splitter array (branchless decision tree, i.e. we can perform arithmetic to manage control flow so that are no conditional jumps. Since k is always a power of two, we can unroll the entire decision tree traversal into fast, straight-line code *and* parallelize it. Pretty cool). The author's algorithm further improves on superscalar samplesort by sorting the buckets in-place.

In-place sorting is achieved by treating the entire input array as a contiguous array of buckets. A set of buffers, whose size can be controlled, writes blocks of elements that belong to distinct buckets to The rule for navigating the branchless decision tree in the authors' algorithm is $i \leftarrow 2i + \mathbb{1}_{a_i<e}$. If e is larger than the splitter a_i then the index is incremented by 1, else we walk left. They remove the equality bucket from StringPS^{4}o and just use ¡, which is made possible by the use of the compare function and the leaf-level computation $2i + 1 - \mathbb{1}_{e<s_i}$ (except in the critical case where there are many duplicate values, in which case equality buckets allow bucket sizes to remain approximately the same). Additionally, to augment parallelization, each level of the decision tree is applied to a block of elements before moving the shared pointers to the next block of elements.

Given this setup, and the dependencies between serial and parallel processing for individual threads based on whether an equality bucket or not (interestingly, the conference portion of the article did not contain the sophisticated scheduling algorithm presented here, and just executed tasks with $> n/t$ elements with all t threads), the actual implementation is critical to the success of this algorithm. The parameters tuned by the authors include $k = 256$, base case size $n_0 = 16$, among others. They report that the buffer blocks and swap buffers (required for permuting the locations of the blocks within individual buckets) take up the majority of additional memory. Since both of these sizes can be modulated by the programmer and are largely independent of size n, this algorithm is considered to be in-place. On a secondary point, I appreciate that this algorithm works with the C++ standard library threads invocation. It is also notable that, for the purpose of comparison, the authors reimplement superscalar samplesort and make their own optimizations.

The experimentation portion of the paper evaluates a number of algorithms, input distributions, and hardware. The evaluation is performed as a grid search so that all permutations are explored, and comparisons between algorithms reported as average slowdown with respect to the set of all algorithms evaluated ($A \in \mathcal{A}$). They conclude that the in-place parallel superscalar samplesort is extremely competitive, if not the fastest, among peer algorithms both on sequential as well as parallel machines. Despite this success, the authors still recommend several potential avenues for future work. These include improving the smaller sorting algorithms for the base case, and using vector instructions for the branchless decision tree.

Honestly, compared to many of the previous papers we have read for this class, I thought there were no glaring omissions (certainly it seems to cover most of the pet peeves listed out by "A Theoretician's Guide to the Experimental Analysis of Algorithms"). The theory is laid out and followed by experiments that are clearly explained and replicable. I do wonder, since the authors emphasize that this is a prototypical

codebase, what the conversion to production code (slash integration into standard libraries) might look like and whether this "practical" implementaiton is also practical when deployed as an alternative to std::sort.

While this is less of an accomplishment, I want to note that the authors also have a very pleasant writing style that very much enhanced the experience of reading this paper.

Pregel: A system for large-scale graph processing

Author(s): Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski

This paper differs from the previous texts we have read, which focus on individual algorithms optimized for parallel or distributed frameworks, in describing a computational model that can efficiently run various algorithms on distributed systems. The authors' motivation is Google's cluster architecture, where geographically interconnected racks of individual PCs have high intra-rack bandwidth but still need to be able to communicate to the other racks to perform large-scale computations on web graphs (sites, social networks etc.)

The authors address the limitations that, in 2010, existed around large graph processing. Users would have to choose between developing/adopting a custom infrastructure for the algorithm or graph topology of choice, or rely in an off-the shelf distributed computing platform unsuitable for graph processing, limiting themselves ot a single-computer graph algorithm library, or using a parallel graph system that was not fault tolerant. Their vision was to create an API that would be scalable, fault-tolerant (i.e. incorporated useful checkpointing), and could run any arbitrary graph algorithm.

A defining feature of the Pregel model is that vertices are the primary agents ("first-class citizens") in performing calculations, voting to terminate the algorithms, changing graph topology, and sending messages to other vertices. Indeed, this agency is so pronounced that the authors note that explicitly listed edges may not be necessary in cases where there is a register of all vertices and the partition of the graph they are responsible for. In addition to local operations performed by individual vertices, the authors introduce the concept of a "superstep", during which vertices can send messages (to be received in the next superstep), modify their state, or modify the topology of the graph. This message-based system was chosen over emulating a shared memory system because it is faster and also compatible with many graph algorithms.

The message-based system does, however, come with some issues. How does one indicate to the process that it should terminate? How to resolve message conflicts (e.g. creating a vertex with conflicting starting values), how to minimize message overhead, and how to keep global statistics up-to-date? The authors address these problems with a range of solutions that naturally emerge from the proposed architecture of Pregel. Terminating the process, for example, is accomplished by having vertices "vote" to terminate (e.g. if performing a max-value operation, once a vertex receives a message from a vertex with a value lower than its own, it votes to terminate. Once all vertices have voted to terminate, they must share the maximum value.) Additionally, Pregel uses constructs called combiners (joining multiple messages into one through some function) and aggregators (global variables that can be updated by vertices at each superstep) to optimize and manage coordination over the distributed system.

The paper is relatively concise but expressive. After laying out the overall framework for the Pregel API, the authors discuss its application to relevant graph algorithms (PageRank and shortest path are naturally important to Google's business). Some of the more interesting comments in this section address work efficiency (the wavefront-based approach to shortest path visits more vertices than sequential versions like Dijkstra, but is massively scaleable in a way that Dijkstra is not.) and the new approaches to standard algorithms made possible by Pregel (e.g. using votes for termination to coordinate bipartite matching.)

The greatest weakness of the paper is maybe the fact that its experiments demonstrate limited comparison to other solutions. Perhaps the argument is that the qualitative statements about the shortcomings of contemporary alternatives suffices to convince the reader, but it would have been nice to see the claim that MapReduce does not perform well on graph algorithms demonstrated as a contrast to Pregel, or even Parallel Boost Graph and CCMgraph, which are discussed in the following section.

Since this is an industry paper, I imagine most of the improvements building on Pregel will be internal to Google, which means that in addition to improving efficiency and building algorithms on top of the framework, the authors will be concerned about longevity of the system, documentation, and usability (e.g. the HTTP server that produces a user interface for the master.) The authors' note at the end is telling. They are no longer "at liberty ot change the API without considering compatibility", indicating that the process of developing this framework had to be thorough enough that an inability to update the underlying system would not be overly detrimental.

GraphChi: Large-Scale Graph Computation on Just a PC

Author(s): Aapo Kyrola, Guy Blelloch, Carlos Guestrin

This paper is an effort at democratizing large-graph processing by developing an algorithm that allows for sophisticated graph computing for graphs on the scale of modern internet graphs to be processed on a single, memory-limited (DRAM) computer instead of being dependent on large, expensive distributed networks.

At the core of the authors' solution, which they call the Parallel Sliding Windows method for processing large graphs from disk (SSD, hard drive), lies a gripe with the bulk-synchronous parallel (BSP) model used in (at the time, the most cutting-edge) vertex-centric graph processing systems such as Pregel and GraphLab (which we discussed in the Tuesday class). The BSP method is subject to bottlenecks and costly synchronization steps that may prevent it from converging at all, issues that could be avoided by using asynchronous methods that allow each update function to use the *most recent* version of the edge and vertex values rather than those from the *previous state*. These asynchronous methods open up a world of possibilities, including selective updating of vertices.

The first issue the authors tackle is one of data representation. CSR (Compressed Sparse Row) format allows for rapid access of out-neighbors, and CSC (Compressed Sparse Column, the CSR of the transposed graph) allows for rapid access of in-neighbors. When vertex values update, however, random-read and random-write are required to ensure alignment between the vertex value and that which its neighbors can access. This is not an easy problem, since real-world graphs are highly interconnected. One way of getting around this would be by simply using SSD, which, as a dedicated heap space, would support faster random-read and -write than a hard disk. Unfortunately its capacity is likely to be insufficient for storing the graph. Other solutions, like graph compression, are effective but less useful when there is data associated with vertices and edges.

The updating solution presented in this paper is neither to "simply have more/faster RAM" nor to compress the graph, but rather to process computations and update the graph by partition. The structure of each partition is as follows: the vertices V of graph $G = (V, E)$ are split into P disjoint intervals. The edges of the graph E are subdivided amongst these intervals according to their destination vertex, and ordered by their source vertex (e.g. if vertex n is in interval p, edge (m, n) will be allocated to a shard in p, and be ranked behind edge (l, n), which precedes it). Each interval constitutes a subgraph of G which can be fully loaded into memory. Thanks to the sorting, the source vertices of adjacent edges (say, (m, n) and (l, n)) will often be adjacent in neighboring intervals, so $P - 1$ block reads are required to fully update the graph given changes in p, or, alternately, P reads to fully process interval p. This significantly reduces the random-reads and random-writes to disk. It also allows for clever sequencing of the operations (e.g. if we know that m and l are in the same interval, PSW will process them in sequence to avoid race conditions, but k, in a different interval, may be updated in parallel).

Dynamic graphs with updating edges are also supported in a memory-access efficient way using edge-buffers. The edge buffers are written to disk once the buffer is full, and if writing the buffer overfills a shard beyond the local memory limit (a requirement for any shard), the shard is split in two.

For a full iteration of PSW, the number of non-sequential disk seeks is $O(P^2)$ as opposed to a possible $O(V^2)$ (where $V >> P$, since if the graph is fully connected, we would require V writes per vertex operation).

Among the implementation details, which are numerous, a few that interested me were:

- The time-intensity of dynamic allocation v.s. predesignating memory to an array (something I've thought about in the past but isn't as natural to encode in in Julia as it is in C++, but that I have increasingly being more mindful of).

- Building and maintaining auxiliary data structures as part of a pre-processing and dynamic-updating procedure.
- Selective scheduling that not only allows some parts of the graph to compute ahead of others, but to organize PSW as a whole. I don't think I fully understand the asynchronous scheduling mechanism, but it seems to me like it would limit the types of iterative calculations to ones that *do* work in a vertex-centric model (i.e. a vertex depends on neighbors to update, so maybe it only gets scheduled once the neighboring vertices report that they are done with their tasks? What prevents one side of the graph from completely running away with the computation and processing too fast relative to another part of the graph?)
- This hypothesis seems at least partially ratified by the types of algorithms the authors present as examples of PSW implementation.
- (My mom also had a Mac Mini running MACOS Lion in 2012, RIP little computer you stored so many CFA flashcards).

I think by modern expectations of processing times the reported values are not particularly impressive, but it is genuinely remarkable that the authors could get a runtime in the same order of magnitude as distributed graph processing algorithms like GraphLab on an 8GB-main-memory local machine. Clearly, the goal of democratizing large-graph processing is accomplished using GraphChi.

Some of the weaknesses the paper admits includes inefficiency for key algorithms such as graph traversals (BFS, etc.) since the shard-based access makes it necessary to load the entire shard and scan across it to find the out-neighbors of a given vertex (i.e. limited priority scheduling). Additionally, one may not want each shards to be written and read to disk every time it is processed if one has sufficient memory to load multiple shards, and the time complexity, while far better than it could be, is still limited by the size of P (which has to be pretty large to fit big graphs into memory). I mentioned my questions about the types of algorithms this method can process earlier (limitation to iterative algorithms). It would also have been nice to see experiments on other machines (e.g. a laptop) to see how performance drops off with increased P etc.