Lecture 21:

Domain-specific Programming on Graphs

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2017

Tunes

The Love Me Nots

Make Up Your Mind

(The Demon and the Devotee)

"Music is meant to inspire, and I'm inspiring the 418 students to pick their projects."

- Nicole Laurenne

Last time: Increasing acceptance of domain-specific programming systems

- Challenge to programmers: modern computers are parallel, heterogeneous machines (HW architects striving for high area and power efficiency)

- Trend: domain-specific programming system design: give up generality in space of programs that can be expressed in exchange for achieving high productivity and high performance

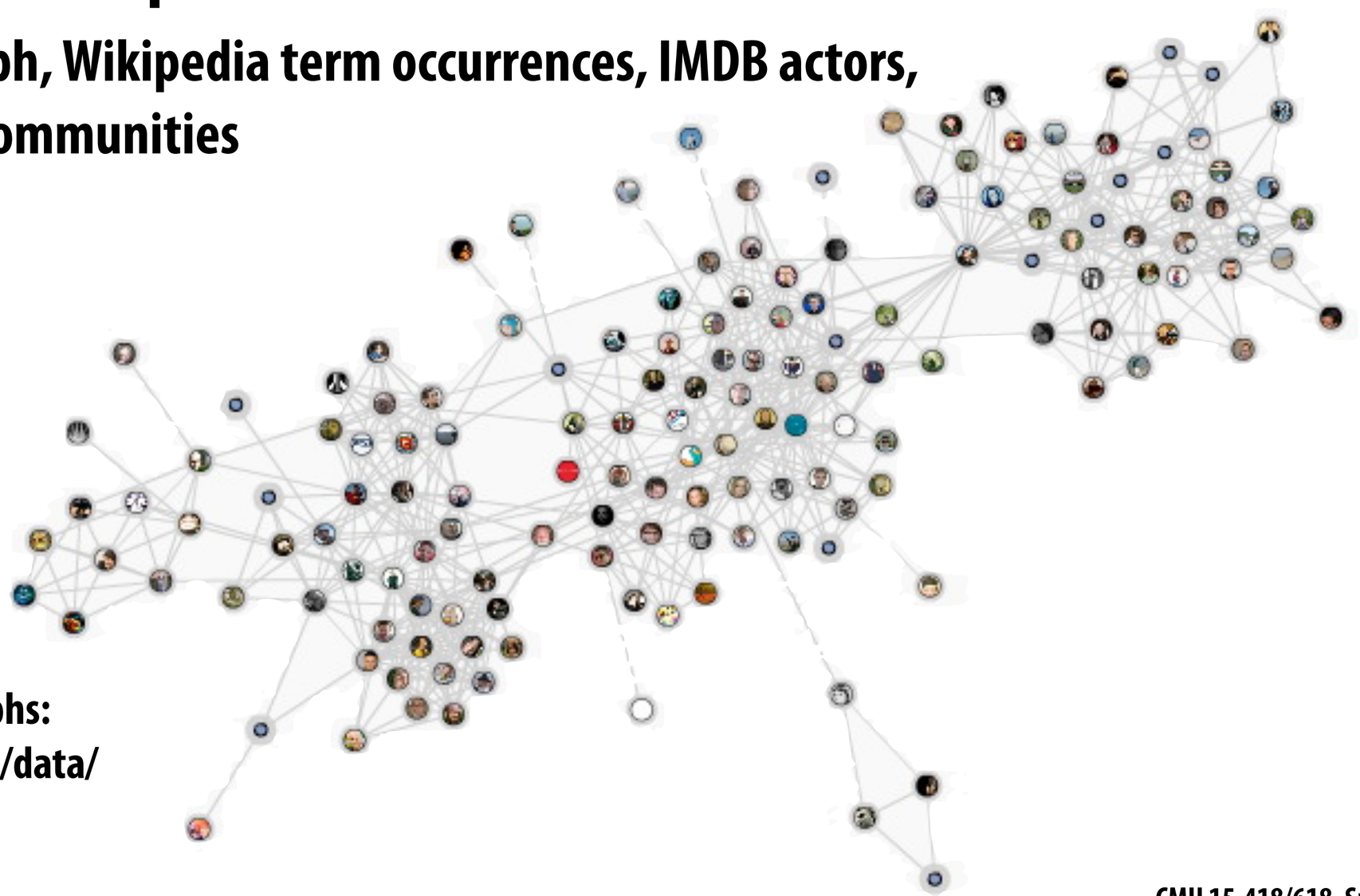
- "Performance portability" is a key goal: programs should execute efficiently on a variety of parallel platforms
  - Good implementations of same program for different systems require different data structures, algorithms, and approaches to parallelization — not just differences in low-level code generation (optimization is not only a matter of generating SSE vs. AVX vs ARM Neon vs. NVIDIA PTX instructions)

Today's topic: analyzing big graphs

- **Many modern applications:**
  - Web search results, recommender systems, influence determination, advertising, anomaly detection, etc.

- **Public dataset examples:**

  Twitter social graph, Wikipedia term occurrences, IMDB actors, Netflix, Amazon communities



Good source of public graphs:
https://snap.stanford.edu/data/

Thought experiment: if we wanted to design a programming system for computing on graphs, where might we begin?

What abstractions do we need?

Whenever I'm trying to assess the importance of a new programming system, I ask two questions:

"What tasks/problems does the system take off the programmer's hands? (are these problems challenging or tedious enough that I feel the system is adding sufficient value for me to want to use it?)"

"What problems does the system leave as the responsibility for the programmer?" (likely because the programmer is better at these tasks)

Liszt (recall last class):

Programmer's responsibility:

- Describe mesh connectivity and fields defined on mesh
- Describe operations on mesh structure and fields

Liszt system's responsibility:

- Parallelize operations without violating dependencies or creating data races (uses different algorithms to parallelize application on different platforms)
- Choose graph data structure / layout, partition graph across parallel machine, manage low-level communication (MPI send), allocate ghost cells, etc.

Halide (recall last class):

Programmer's responsibility:

- Describing image processing algorithm as pipeline of operations on images
- Describing the schedule for executing the pipeline (e.g., "block this loop, "parallelize this loop", "fuse these stages")

Halide system's responsibility:

- Implementing the schedule using mechanisms available on the target machine (spawning pthreads, allocating temp buffers, emitting vector instructions, loop indexing code)

A good exercise: carry out this evaluation for another programming system: like OpenGL, SQL, MapReduce, etc.

Programming system design questions:

- What are the fundamental operations we want to be easy to express and efficient to execute?

- What are the key optimizations used when authoring the best implementations of these operations?

(high-level abstractions provided by a programming system should not stand in the way of these optimizations, and maybe even allow the system to perform them for the application)

Example graph computation: Page Rank

Page Rank: iterative graph algorithm

Graph nodes = web pages

Graph edges = links between pages

discount

$$R[i] = \frac{1-\alpha}{N} + \alpha \sum_{j \text{ links to } i} \frac{R[j]}{\text{Outlinks}[j]}$$

Rank of page *i*

Weighted combination of rank of pages that link to it

GraphLab

Low et al. 2010
www.graphlab.org

- A system for describing iterative computations on graphs
- Implemented as a C++ library
- Runs on shared memory machines or distributed across clusters
  - GraphLab runtime takes responsibility for scheduling work in parallel, partitioning graphs across clusters of machines, communication between master, etc.

GraphLab programs: state

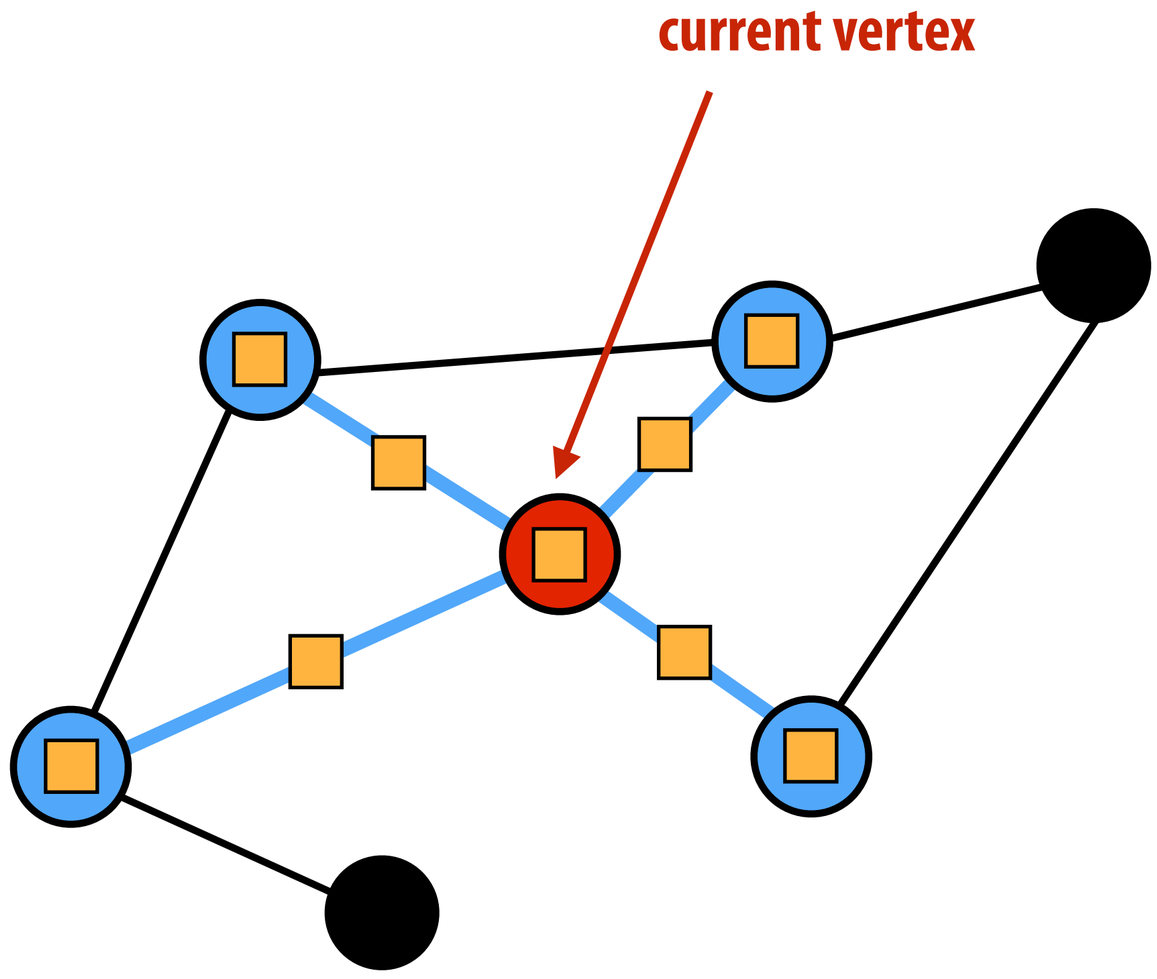
- **The graph: G = (V, E)**
  - Application defines data blocks on each vertex and directed edge
  - D_v = data associated with vertex v
  - $D_{u \to v}$ = data associated with directed edge $u \to v$
- **Read-only global data**
  - Can think of this as per-graph data, rather than per vertex or per-edge data)

Notice: I always first describe program state

And then describe what operations are available to manipulate this state

GraphLab operations: the "vertex program"

- Defines per-vertex operations on the vertex's local neighborhood
- Neighborhood (aka "scope") of vertex:
  - The current vertex
  - Adjacent edges
  - Adjacent vertices



Recall: in the previous lecture the "in scope" data for a loop iteration was called the "stencil"

= vertex or edge data "in scope" of red vertex (graph data that can be accessed when executing a vertex program at the current (red) vertex)

Simple example: PageRank *

$$R[i] = \frac{1-\alpha}{N} + \alpha \sum_{j \text{ links to } i} \frac{R[j]}{\text{Outlinks}[j]}$$

```
PageRank_vertex_program(vertex i) {

  // (Gather phase) compute the sum of my neighbors rank
  double sum = 0;
  foreach(vertex j : in_neighbors(i)) {
    sum = sum + j.rank / num_out_neighbors(j);
  }

  // (Apply phase) Update my rank (i)
  i.rank = (1-0.85)/num_graph_vertices() + 0.85*sum;
}
```

Let alpha = 0.85

Programming in GraphLab amounts to defining how to update graph state at each vertex. The system takes responsibility for scheduling and parallelization.

* This is made up syntax for slide simplicity: actual syntax is C++, as we'll see on the next slide

GraphLab: data access

- The application's vertex program executes per-vertex
- The vertex program defines:
  - What adjacent edges are inputs to the computation
  - What computation to perform per edge
  - How to update the vertex's value
  - What adjacent edges are modified by the computation
  - How to update these output edge values

- Note how GraphLab requires the program to tell it all data that will be accessed, and whether it is read or write access

PageRank: GraphLab vertex program (C++ code)

```
struct web_page {
  std::string pagename;
  double      pagerank;
  web_page(): pagerank(0.0) { }
}

typedef graphlab::distributed_graph<web_page, graphlab::empty> graph_type;

class pagerank_program : public graphlab::ivertex_program …
{
public:
  // we are going to gather over all the in-edges
  edge_dir_type gather_edges(icontext_type& context,
                             const vertex_type& vertex) const {
    return graphlab::IN_EDGES;
  }

  // for each in-edge gather the weighted sum of the edge.
  double gather(icontext_type& context, const vertex_type& vertex,
                edge_type& edge) const {
    return edge.source().data().pagerank / edge.source().num_out_edges();
  }

  // Use the total rank of adjacent pages to update this page
  void apply(icontext_type& context, vertex_type& vertex,
             const gather_type& total) {
    double newval = total * 0.85 + 0.15;
    vertex.data().pagerank = newval;
  }

  // No scatter needed. Return NO_EDGES
  edge_dir_type scatter_edges(icontext_type& context,
                              const vertex_type& vertex) const {
    return graphlab::NO_EDGES;
  }
};
```

Graph has record of type web_page per vertex, and no data on edges

Define edges to gather over in "gather phase"

Compute value to accumulate for each edge

Update vertex rank

PageRank example performs no scatter

Running the program

```
graphlab::omni_engine<pagerank_program> engine(dc, graph, "sync");
engine.signal_all();
engine.start();
```

GraphLab runtime provides "engines" that manage scheduling of vertex programs

`engine.signal_all()` marks all vertices for execution

You can think of the GraphLab runtime as a work queue scheduler.
And invoking a vertex program on a vertex as a task that is placed in the work queue.

So it's reasonable to read the code above as: "place all vertices into the work queue"

Or as: "foreach vertex" run the vertex program.

Vertex signaling: GraphLab's mechanism for generating new work

$$R[i] = \frac{1-\alpha}{N} + \alpha \sum_{j \text{ links to } i} \frac{R[j]}{\text{Outlinks}[j]}$$

Iteratively update of all R[i]'s 10 times

Uses generic "signal" primitive (could also wrap code on previous slide in a for loop)

```
struct web_page {
  std::string pagename;
  double      pagerank;
  int         counter;
  web_page(): pagerank(0.0),counter(0) { }
}

 // Use the total rank of adjacent pages to update this page
  void apply(icontext_type& context, vertex_type& vertex,
             const gather_type& total) {
    double newval = total * 0.85 + 0.15;
    vertex.data().pagerank = newval;
    vertex.data().counter++;
    if (vertex.data().counter < 10)
       vertex.signal();
  }
```

Per-vertex "counter"

If counter < 10, signal to scheduler to run the vertex program on the vertex again at some point in the future

Signal: general primitive for scheduling work

Parts of graph may converge at different rates
(iterate PageRank until convergence, but only for vertices that need it)

```
class pagerank_program: public graphlab::ivertex_program

private:
  bool perform_scatter;

public:

   // Use the total rank of adjacent pages to update this page
  void apply(icontext_type& context, vertex_type& vertex,
             const gather_type& total) {
    double newval = total * 0.85 + 0.15;
    double oldval = vertex.data().pagerank;
    vertex.data().pagerank = newval;
    perform_scatter = (std::fabs(prevval - newval) > 1E-3);
  }

  // Scatter now needed if algorithm has not converged
  edge_dir_type scatter_edges(icontext_type& context,
                              const vertex_type& vertex) const {
    if (perform_scatter) return graphlab::OUT_EDGES;
    else return graphlab::NO_EDGES;
  }

   // Make sure surrounding vertices are scheduled
   void scatter(icontext_type& context, const vertex_type& vertex,
                edge_type& edge) const {
    context.signal(edge.target());
  }
};
```

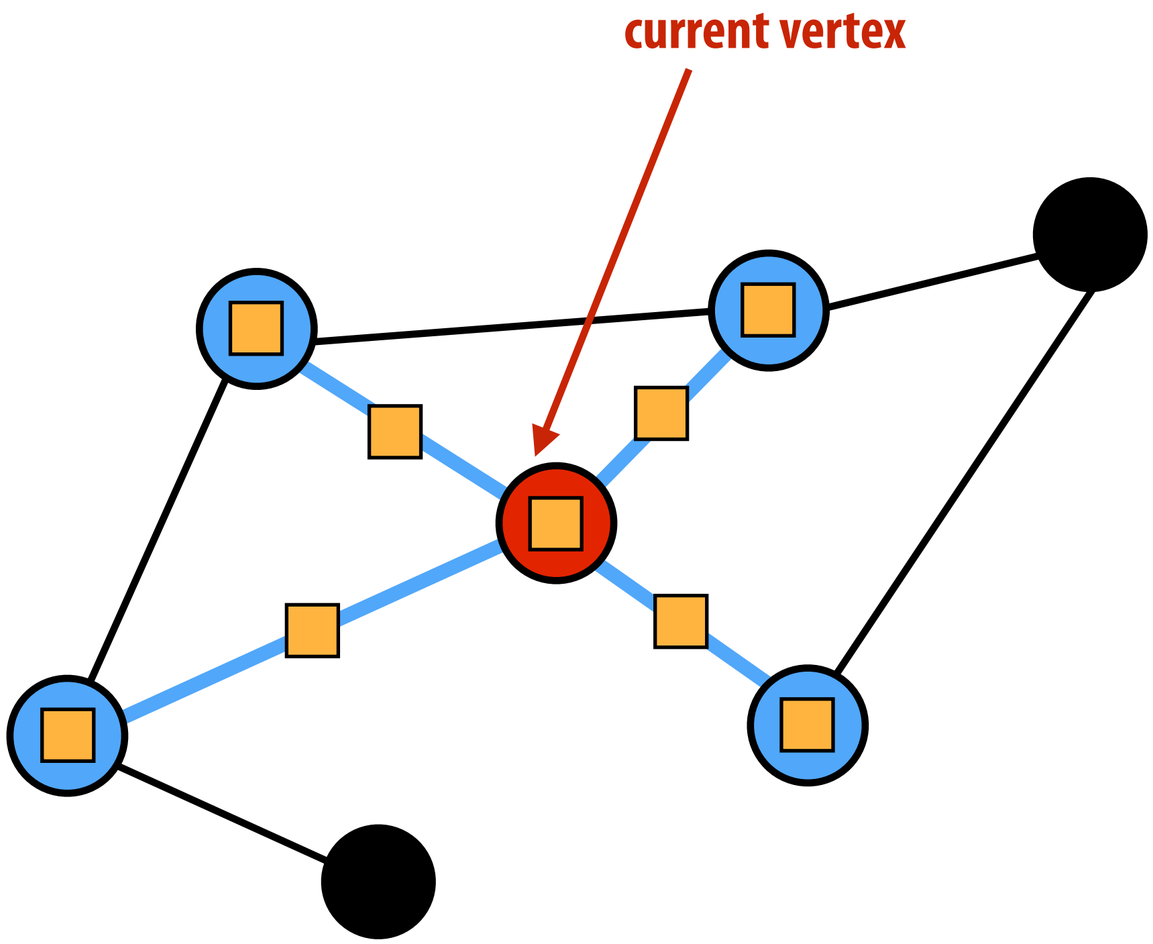
Private variable set during apply phase, used during scatter phase

Check for convergence

Schedule update of neighbor vertices

Synchronizing parallel execution

Local neighborhood of vertex (vertex's "scope") can be read and written to by a vertex program



= vertex or edge data in scope of red vertex

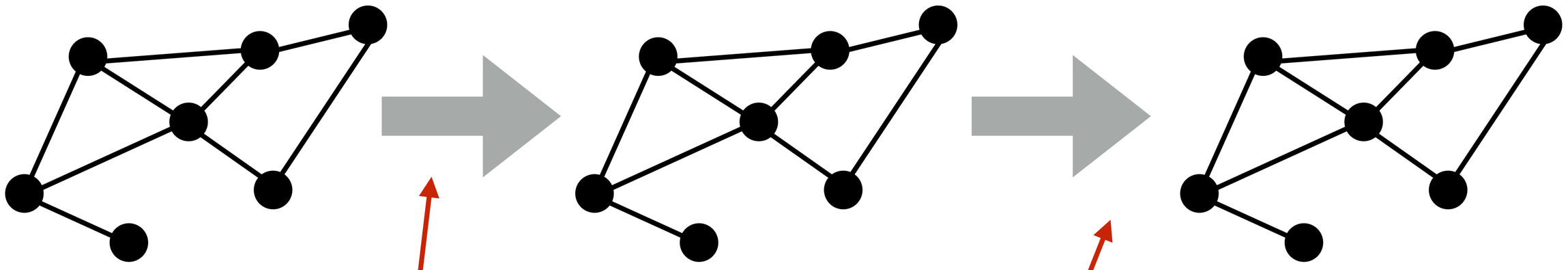
Programs specify what granularity of atomicity ("called consistency by GraphLab") they want GraphLab to provide: this determines amount of available parallelism

- **"Full consistency": implementation ensures no other execution reads or writes to data in scope of v when vertex program for v is running.**
- **"Edge consistency": no other execution reads or writes any data in v or in edges adjacent to v**
- **"Vertex consistency": no other execution reads or writes to data in v ...**

GraphLab: job scheduling order

GraphLab implements several work scheduling policies

- Synchronous: update all scheduled vertices "simultaneously" (vertex programs observe no updates from programs run on other vertices in same "round")



Run vertex programs for all scheduled vertices.
(output to copy of graph structure)

Run vertex programs for all scheduled vertices.
(output to copy of graph structure)

GraphLab: job scheduling order

- GraphLab implements several work scheduling policies
  - Synchronous: update all vertices simultaneously (vertex programs observe no updates from programs run on other vertices in same "round")
  - Round-robin: vertex programs observe most recent updates
  - Graph coloring
  - Dynamic: based on new work created by `signal`
    - Several implementations: fifo, priority-based, "splash" ...

- Application developer has flexibility for choosing consistency guarantee and scheduling policy
  - Implication: choice of schedule impacts program's correctness/output
  - Kayvon's opinion: this seems like a weird design at first glance, but this is common (and necessary) in the design of efficient graph algorithms

Summary: GraphLab concepts

- Program state: data on graph vertices and edges + globals

- Operations: per-vertex update programs and global reduction functions (reductions not discussed today)
  - Simple, intuitive description of work (follows mathematical formulation)
  - Graph restricts data access in vertex program to local neighborhood
  - Asynchronous execution model: application creates work dynamically by "signaling vertices" (enable lazy execution, work efficiency on real graphs)

- Choice of scheduler and consistency implementation
  - In this domain, the order in which nodes are processed can be critical property for both performance and quality of result
  - Application responsible for choosing right scheduler for its needs

Ligra

[Shun 2013]

- A simple framework for parallel graph operations
- Motivating example: breadth-first search

```
parents = {-1, ..., -1}

// d = dst: vertex to “update” (just encountered)
// s = src: vertex on frontier with edge to d
procedure UPDATE(s, d)
   return compare-and-swap(parents[d], -1, s);
```

```
procedure COND(i)
   return parents[i] == -1;

procedure BFS(G, r)
   parents[r] = r;
   frontier = {r};
   while (size(frontier) != 0) do:
       frontier = EDGEMAP(G, frontier, UPDATE, COND);
```

Semantics of EDGEMAP:
foreach vertex i in frontier, call UPDATE for all neighboring vertices j for which COND(j) is true. Add j to returned set if UPDATE(i, j) returns true

Implementing edgemap

- Assume vertex subset U (`frontier` in previous example) is represented sparsely:
  - e.g., three vertex subset U of 10 vertex graph G=(E,V): $U \subset V = \{0, 4, 9\}$

graph

set of vertices

```
procedure EDGEMAP_SPARSE(G, U, F, C):
   result = {}
   parallel foreach v in U do:
      parallel foreach v2 in out_neighbors(v) do:
         if (C(v2) == 1 and F(v,v2) == 1) then
            add v2 to result
   remove duplicates from result
   return result;
```

condition check on neighbor vertex

update function on neighbor vertex

Cost of EDGEMAP_SPARSE?

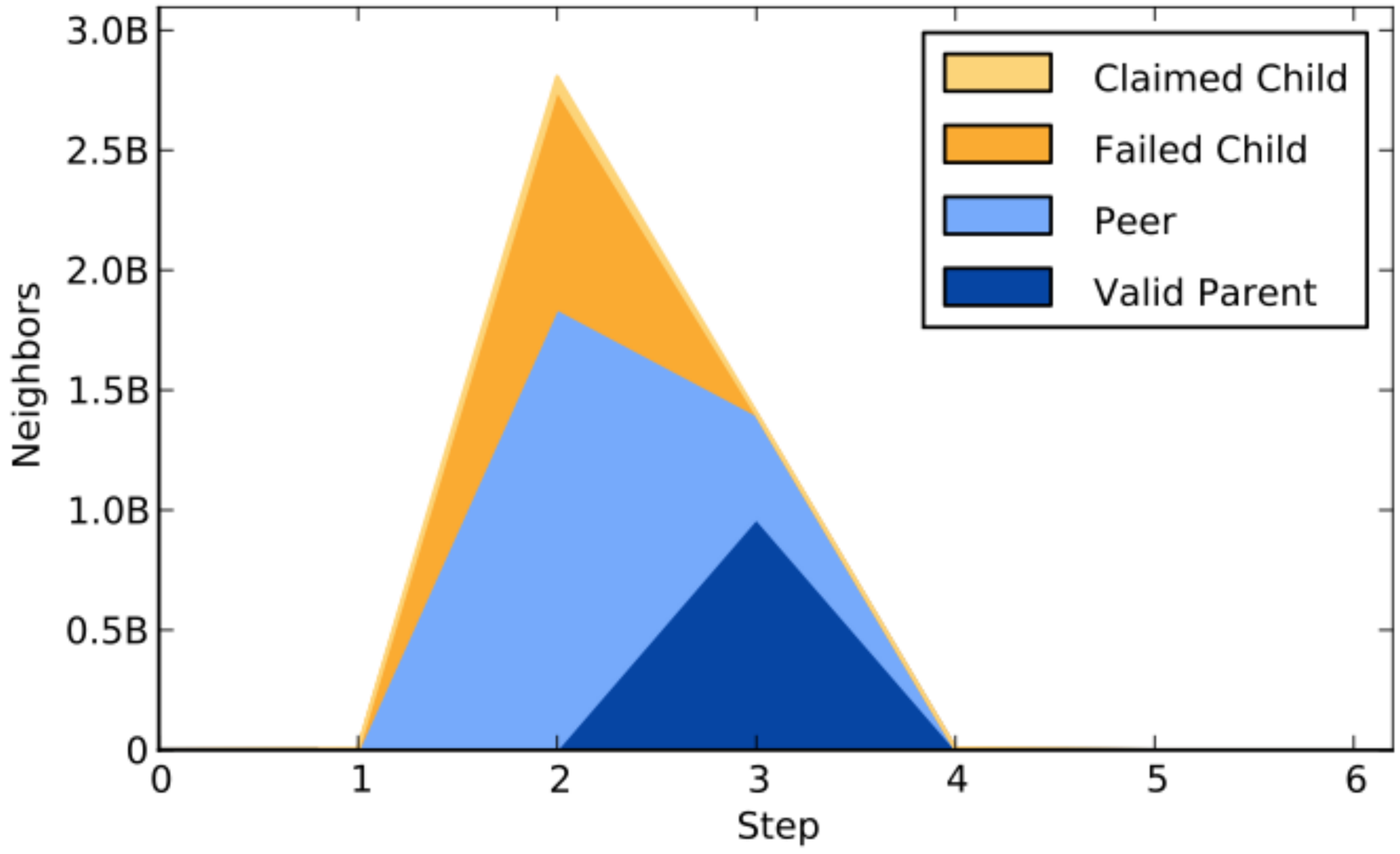
O(|U| + sum of outgoing edges from U)

```
parents = {-1, ..., -1}

procedure UPDATE(s, d)
   return compare-and-swap(parents[d], -1, s);

procedure COND(i)
   return parents[i] == -1;

procedure BFS(G, r)
   parents[r] = r;
   frontier = {r};
   while (size(frontier) != 0) do:
       frontier = EDGEMAP(G, frontier, UPDATE, COND);
```

Visiting every edge on frontier can be wasteful

- **Each step of BFS, every edge on frontier is visited**
  - **Frontier can grow quickly for social graphs (few steps to visit all nodes)**
  - **Most edge visits are wasteful! (they don't lead to a successful "update")**

- **claimed child: edge points to unvisited node (useful work)**
- **failed child: edge points to node found in this step via another edge**
- **peer: edge points to a vertex that was added to frontier in same step as current vertex**
- **valid parent: edge points to vertex found in previous step**



[Credit: Beamer et al. SC12]

Implementing edgemap for dense vertex subsets

- Assume vertex subset (`frontier` in previous example) is represented densely with a bitvector:
  - e.g., vertex subset U of 10 vertex graph G=(E,V): $U \subset V = \{1,0,0,0,1,0,0,0,0,1\}$

```
procedure EDGEMAP_SPARSE(G, U, F, C):
   result = {}
   parallel foreach v in U do:
      parallel foreach v2 in out_neighbors(v) do:
         if (C(v2) == 1 and F(v,v2) == 1) then
            add v2 to result
remove duplicates from result
return result;
```

```
procedure EDGEMAP_DENSE(G, U, F, C):
   result = {}
   parallel for i in {0,...,|V|-1} do:
      if (C(i) == 1) then:
         foreach v in in_neighbors(i) do:
            if v ∈ U and F(v, i) == 1 then:
               add i to result;
               if (C(i) == 0)
                  break;
return result;
```

Cost of EDGEMAP_DENSE?

For each unvisited vertex, quit searching as soon as some parent is found

Could be as low as O(|V|)

Also no synchronization needed ("gather" results rather than "scatter")

Ligra on one slide

- **Entities:**
  - **Graphs**
  - **Vertex subsets (represented sparsely or densely by system)**
  - **EDGEMAP and VERTEXMAP functions**

```
procedure EDGEMAP(G, U, F, C):
   if (|U| + sum of out degrees > threshold)
     return EDGEMAP_DENSE(G, U, F, C);
   else
     return EDGEMAP_SPARSE(G, U, F, C);
```

Iterate over all vertices adjacent to vertices in set U
Choose right algorithm for the job

```
procedure VERTEXMAP(U, F):
   result = {}
   parallel for u ∈ U do:
      if (F(u) == 1) then:
         add u to result;
   return result;
```

Iterate over all vertices in set U

Page rank in Ligra

```
r_cur  = {1/|V|, ... 1/|V|};
r_next = {0,...,0};
diff = {}

procedure PRUPDATE(s, d):
   atomicIncrement(&r_next[d], r_cur[s] / vertex_degree(s));

procedure PRLOCALCOMPUTE(i):
  r_next[i] = alpha * r_next[i] + (1 - alpha) / |V|;
  diff[i] = |r_next[i] - r_cur[i]|;
  r_cur[i] = 0;
  return 1;

procedure COND(i):
  return 1;

procedure PAGERANK(G, alpha, eps):
   frontier = {0, ... , |V|-1}
   error = HUGE;
   while (error > eps) do:
      frontier = EDGEMAP(G, frontier, PRUPDATE, COND);
      frontier = VERTEXMAP(frontier, PRLOCALCOMPUTE);
      error = sum of per-vertex diffs  // this is a parallel reduce
      swap(r_cur, r_next);
   return err
```

Question: can you implement the iterate until convergence optimization we previously discussed in GraphLab?

(if so, what GraphLab scheduler implementation is the result equivalent to?)

Ligra summary

- System abstracts graph operations as data-parallel operations over vertices and edges
  - Emphasizes graph traversal (potentially small subset of vertices operated on in a data parallel step)

- These basic operations permit a surprisingly wide space of graph algorithms:
  - Betweenness centrality
  - Connected components
  - Shortest paths

See Ligra: a Lightweight Framework for Graph Processing for Shared Memory [Shun and Blelloch 2013]

Elements of good domain-specific programming system design

#1: good systems identify the most important cases, and provide most benefit in these situations

- Structure of code mimics the natural structure of problems in the domain
  - Halide: pixel-wise view of filters: pixel(x,y) computed as expression of these input pixel values
  - Graph processing algorithms are designed in terms of per-vertex operations)

- Efficient expression: common operations are easy and intuitive to express

- Efficient implementation: the most important optimizations in the domain are performed by the system for the programmer
  - My experience: a parallel programming system with "convenient" abstractions that precludes best-known implementation strategies will almost always fail

#2: good systems are usually simple systems

- They have a small number of key primitives and operations
  - Ligra: only two operations! (vertexmap and edgemap)
  - GraphLab: run computation per vertex, trigger new work by signaling
    - But GraphLab gets messy with all the scheduling options
  - Halide: only a few scheduling primitives
  - Hadoop: map + reduce

- Allows compiler/runtime to focus on optimizing these primitives
  - Provide parallel implementations, utilize appropriate hardware

- Common question that good architects ask: "do we really need that?" (can this concept be reduced to a primitive we already have?)
  - For every domain-specific primitive in the system: there better be a strong performance or expressivity justification for its existence

#3: good primitives compose

- Composition of primitives allows for wide application scope, even if scope remains limited to a domain
  - e.g., frameworks discussed today support a wide variety of graph algorithms
  - Halide's loop ordering + loop interleaving schedule primitives allow for expression of wide range of schedules

- Composition often allows optimization to generalizable
  - If system can optimize A and optimize B, then it can optimize programs that combine A and B

- Sign of a good design:
  - System ultimately is used for applications original designers never anticipated

- Sign that a feature <u>should not</u> be added (or added in a different way):
  - The new feature does not compose with all existing features in the system

Optimizing graph computations

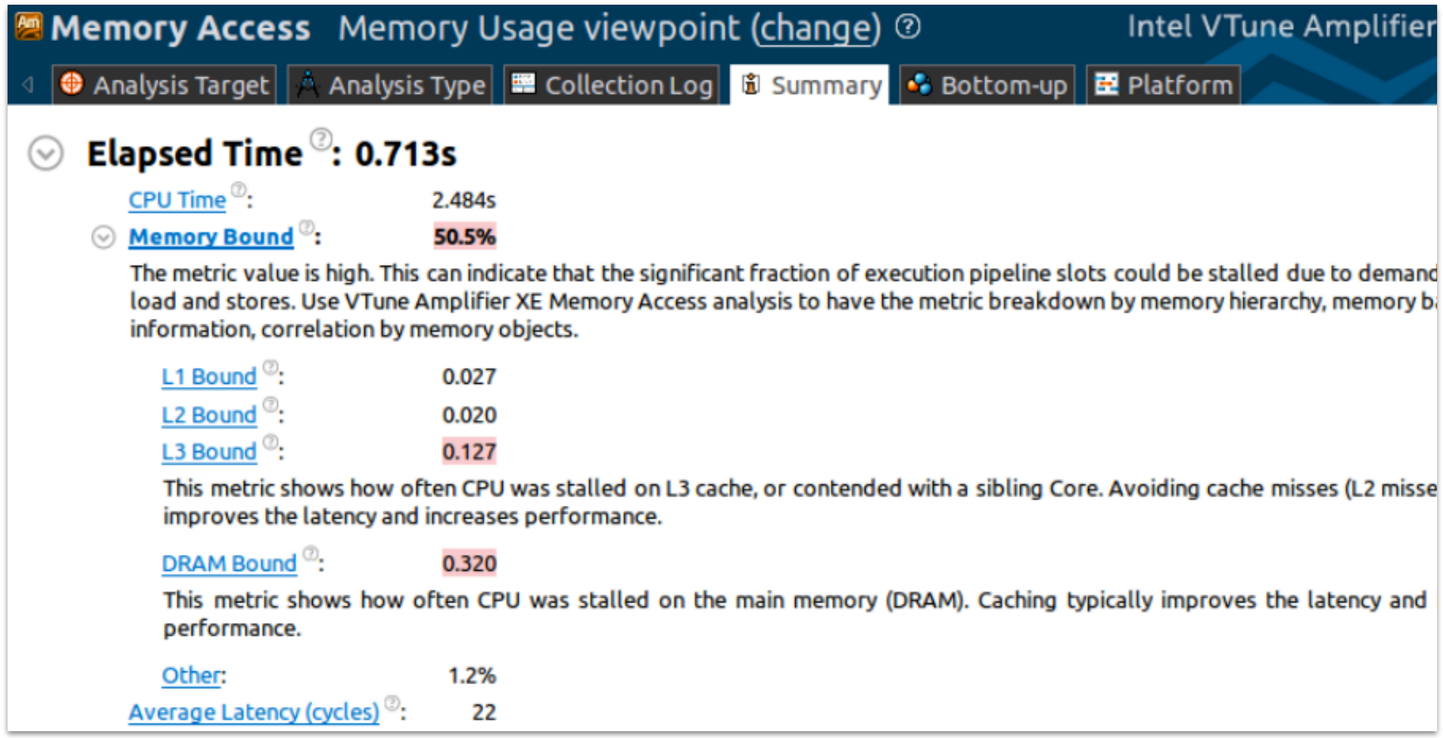
(now we are talking about implementation)

Wait a minute…

- So far in this lecture, we've discussed issues such as parallelism, synchronization …
- But you may recall from Assignment 3 that graph processing is typically has low arithmetic intensity

Walking over edges accesses information from "random" graph vertices

VTune profiling results from Asst 3: Memory bandwidth bound!



Or just consider PageRank: ~ 1 multiply-accumulate per iteration of summation loop

$$R[i] = \frac{1-\alpha}{N} + \alpha \sum_{j \text{ links to } i} \frac{R[j]}{\text{Outlinks}[j]}$$

Two ideas to increase the performance of operations on large graphs *

1. Reorganize graph structure to increase locality

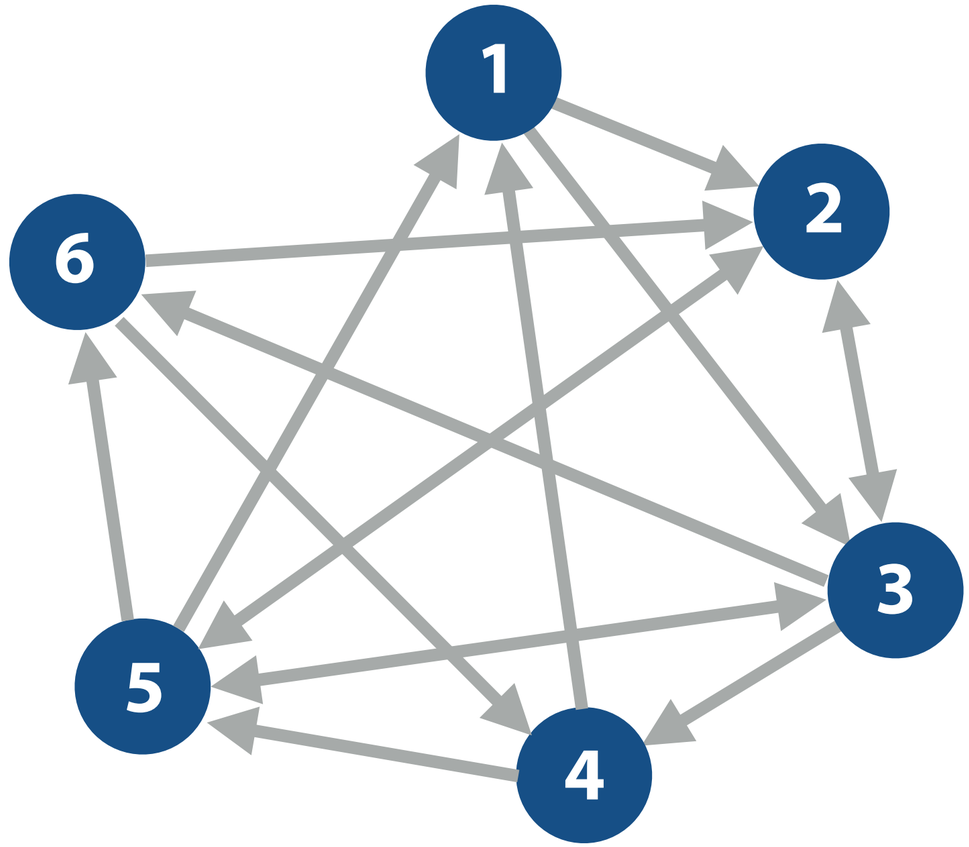
2. Compress the graph

* Both optimizations might be performed by a framework without application knowledge

Recall: directed graph representation

Vertex Id	1	2	3	4	5	6
Outgoing Edges	2 3	3 5	2 4 5 6	1 5	1 2 3 6	2 4

Vertex Id	1	2	3	4	5	6
Incoming Edges	4 5	1 3 5 6	1 2 5	3 6	2 3 4	3 6



Memory footprint challenge of large graphs

- **Challenge: cannot fit all edges in memory for large graphs (graph vertices may fit)**
  - Consider representation of graph from Assignment 3:
    - Each edge represented twice in graph structure (as incoming/outgoing edge)
    - 8 bytes per edge to represent adjacency
  - May also need to store per-edge values (e.g., 4 bytes for a per-edge weight)
  - 1 billion edges (modest): ~12 GB of memory for edge information
  - Algorithm may need multiple copies of per-edge structures (current, prev data, etc.)

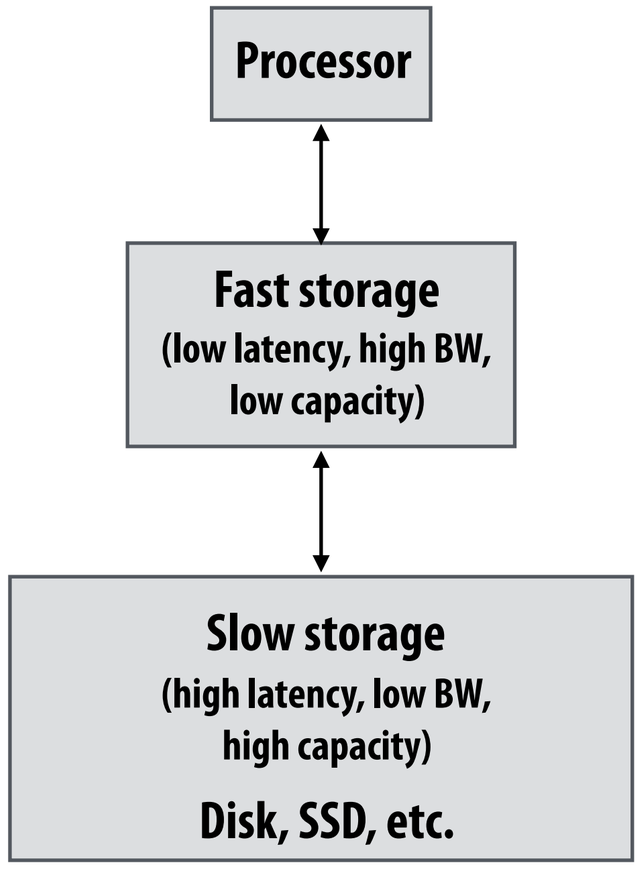
- **Could employ cluster of machines to store graph in memory**
  - Rather than store graph on disk

- **Would prefer to process large graphs on a single machine**
  - Managing clusters of machines is difficult
  - Partitioning graphs is expensive (also needs a lot of memory) and difficult

"Streaming" graph computations

- Graph operations make "random" access to graph data (edges adjacent to vertex *v* may distributed arbitrarily throughout storage)
  - Single pass over graph's edges might make billions of fine-grained accesses to disk

- Streaming data access pattern
  - Make large, predictable data accesses to slow storage (achieve high bandwidth data transfer)
  - Load data from slow storage into fast storage*, then reuse it as much as possible before discarding it (achieve high arithmetic intensity)
  - Can we modify the graph data structure so that data access requires only a small number of efficient bulk loads/stores from slow storage?



* By fast storage, in this context I mean DRAM. However, techniques for streaming from disk into memory would also apply to streaming from memory into a processor's cache

Sharded graph representation

GraphChi: Large-scale graph computation on just a PC [Kryola et al. 2013]

- Partition graph vertices into intervals (sized so that subgraph for interval fits in memory)
- Vertices and only incoming edges to these vertices are stored together in a shard
- Sort edges in a shard by source vertex id

Shard 1: vertices (1-2)

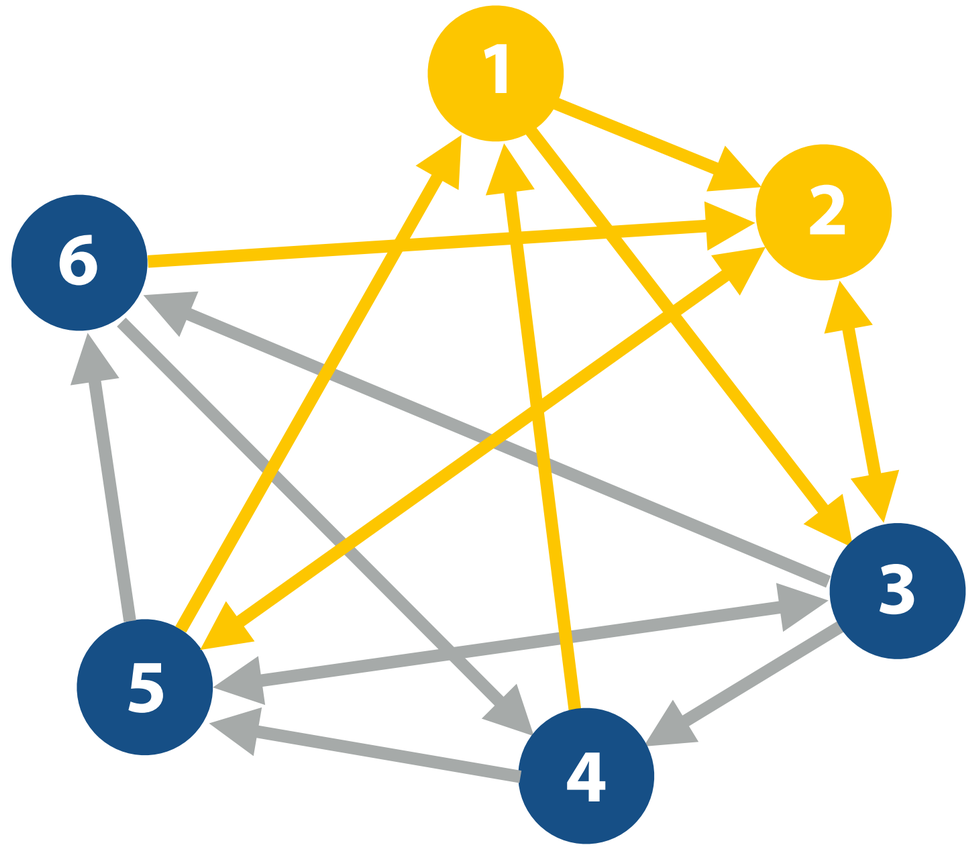
src	dst	value
1	2	0.3
3	2	0.2
4	1	0.8
5	1	0.25
	2	0.6
6	2	0.1

Shard 2: vertices (3-4)

src	dst	value
1	3	0.4
2	3	0.9
3	4	0.15
5	3	0.2
6	4	0.9

Shard 3: vertices (5-6)

src	dst	value
2	5	0.6
3	5	0.9
	6	0.85
4	5	0.3
5	6	0.2



Yellow = data required to process subgraph containing vertices in shard 1

Notice: to construct subgraph containing vertices in shard 1 and their incoming and outgoing edges, only need to load contiguous information from other P-1 shards

Writes to updated outgoing edges require P-1 bulk writes

Sharded graph representation

GraphChi: Large-scale graph computation on just a PC [Kryola et al. 2013]

- Partition graph vertices into intervals (sized so that subgraph for interval fits in memory)
- Vertices and only incoming edges to these vertices are stored together in a shard
- Sort edges in a shard by source vertex id

Shard 1: vertices (1-2)

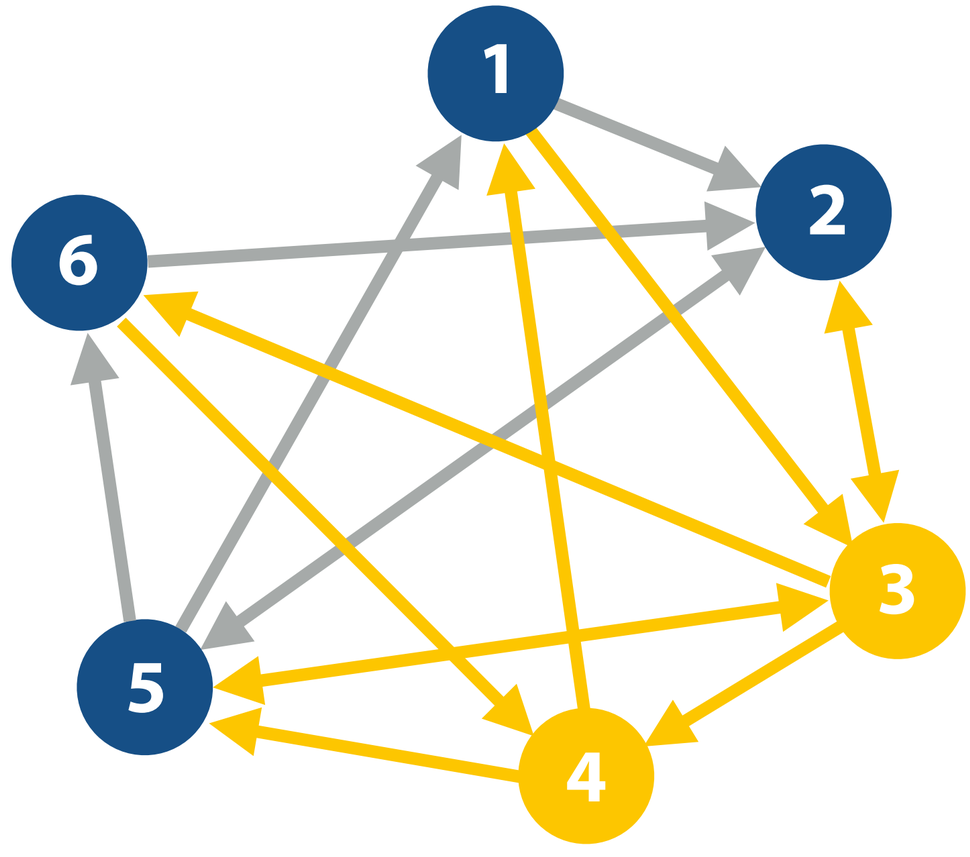
src	dst	value
1	2	0.3
3	2	0.2
4	1	0.8
5	1	0.25
	2	0.6
6	2	0.1

Shard 2: vertices (3-4)

src	dst	value
1	3	0.4
2	3	0.9
3	4	0.15
5	3	0.2
6	4	0.9

Shard 3: vertices (5-6)

src	dst	value
2	5	0.6
3	5	0.9
	6	0.85
4	5	0.3
5	6	0.2



Yellow = data required to process subgraph containing vertices in shard 2

Sharded graph representation

GraphChi: Large-scale graph computation on just a PC [Kryola et al. 2013]

- Partition graph vertices into intervals (sized so that subgraph for interval fits in memory)
- Vertices and only incoming edges to these vertices are stored together in a shard
- Sort edges in a shard by source vertex id

Shard 1: vertices (1-2)

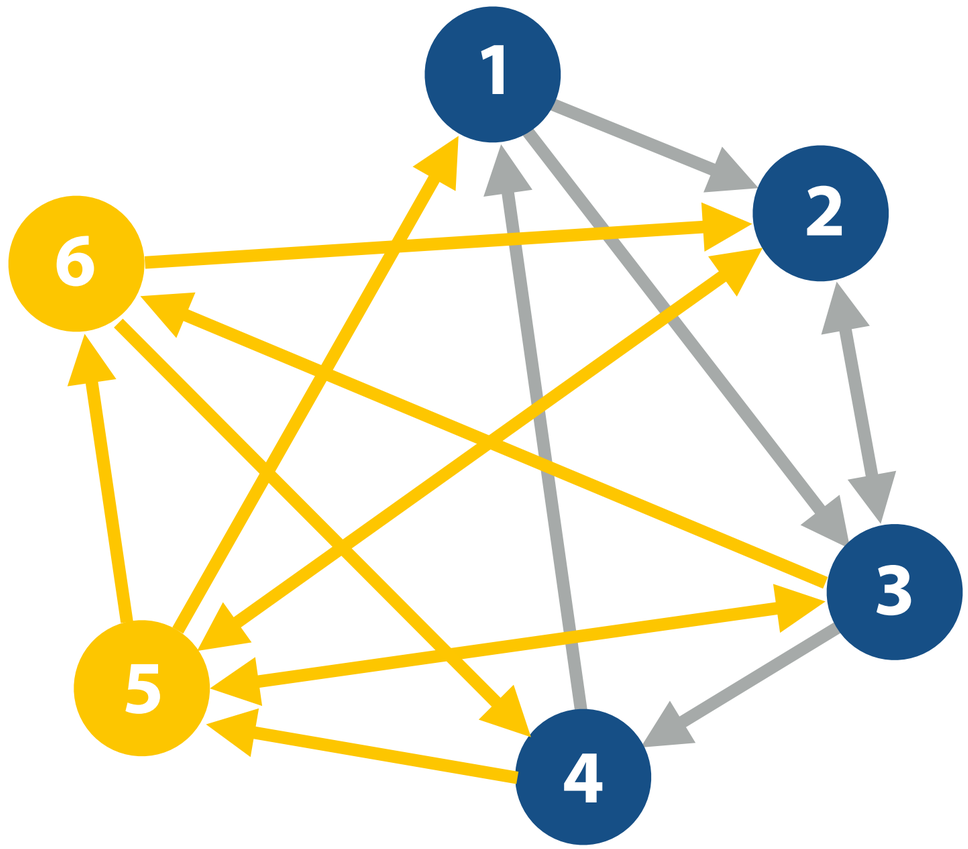
src	dst	value
1	2	0.3
3	2	0.2
4	1	0.8
5	1	0.25
	2	0.6
6	2	0.1

Shard 2: vertices (3-4)

src	dst	value
1	3	0.4
2	3	0.9
3	4	0.15
5	3	0.2
6	4	0.9

Shard 3: vertices (5-6)

src	dst	value
2	5	0.6
3	5	0.9
	6	0.85
4	5	0.3
5	6	0.2



Yellow = data required to process subgraph containing vertices in shard 3

Observe: due to sort of incoming edges, iterating over all intervals results in contiguous sliding window over the shards

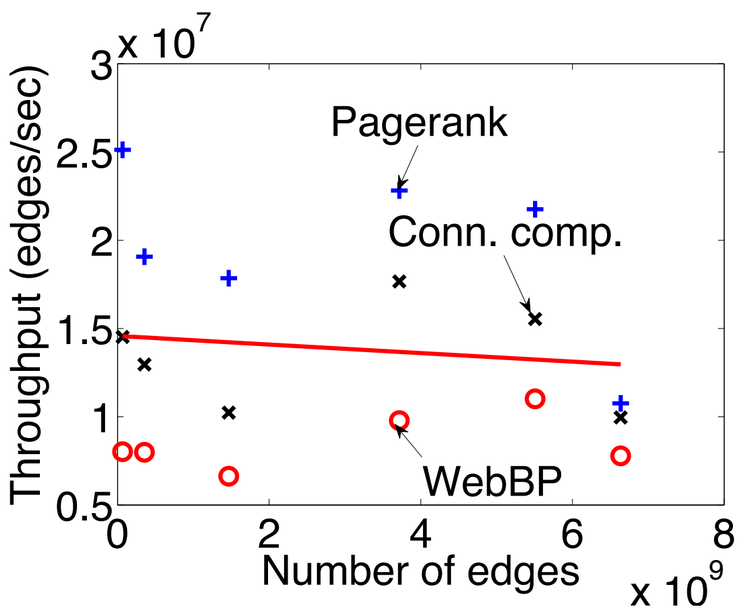
Putting it all together: looping over all graph edges

For each partition *i* of vertices:

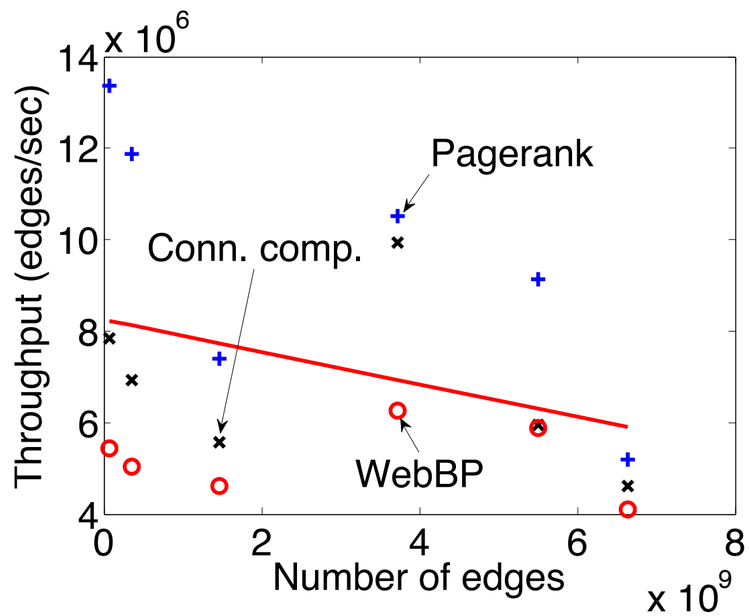
- **Load shard *i* (contains all incoming edges)**
- **For each other shard *s***
  - **Load section of *s* containing data for edges leaving *i* and entering *s***
- **Construct subgraph in memory**
- **Do processing on subgraph**

Note: a good implementation could hide disk IO by prefetching data for next iteration of loop

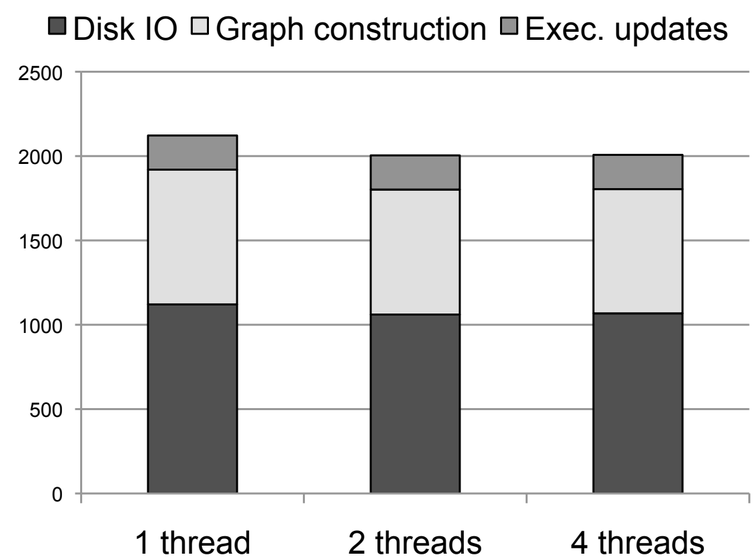
Performance on a Mac mini (8 GB RAM)



(a) Performance: SSD



(b) Performance : Hard drive



(c) Runtime breakdown

Throughput (edges/sec) remains stable as graph size is increased

- Desirable property: throughput (edges/sec) largely invariant of dataset size

Graph compression

- Recall: graph operations are often BW-bound

- Implication: using CPU instructions to reduce BW requirements can benefit overall performance (the processor would be waiting on memory anyway, so use it to decompress data!)

- Idea: store graph compressed in memory, decompress on-the-fly when operation wants to read data

Compressing an edge list

Vertex Id 32

Outgoing Edges 1001 10 5 30 6 1025 200000 1010 1024 100000 1030 275000

---

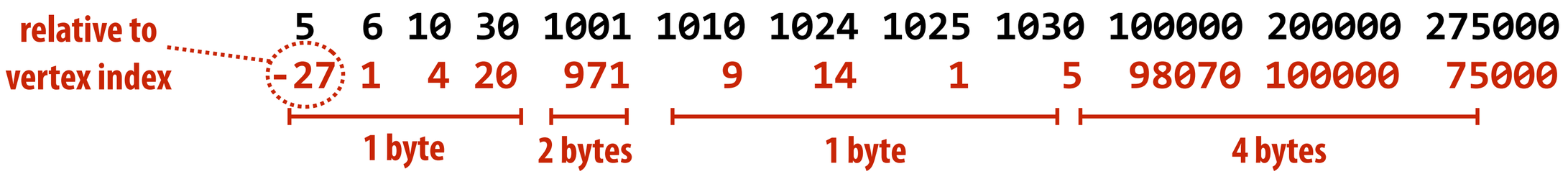
1. **Sort edges for each vertex**

5 6 10 30 1001 1010 1024 1025 1030 100000 200000 275000

2. **Compute differences**

5 6 10 30 1001 1010 1024 1025 1030 100000 200000 275000
0 1 4 20 971 9 14 1 5 98070 100000 75000

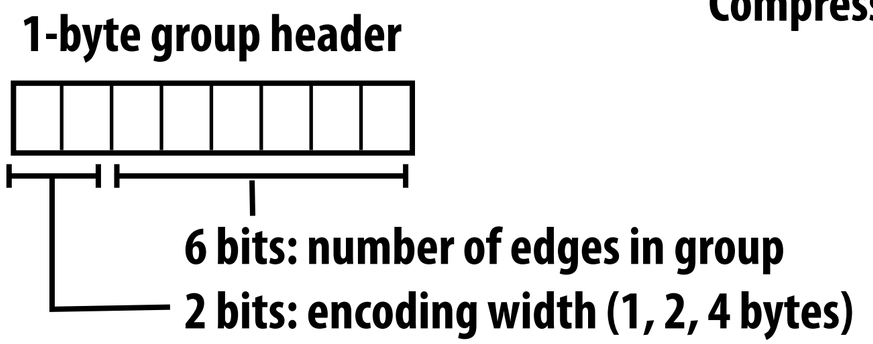
3. **Group into sections requiring same number of bytes**



4. **Encode deltas**

Uncompressed encoding: 12 edges x 4 bytes = 48 bytes

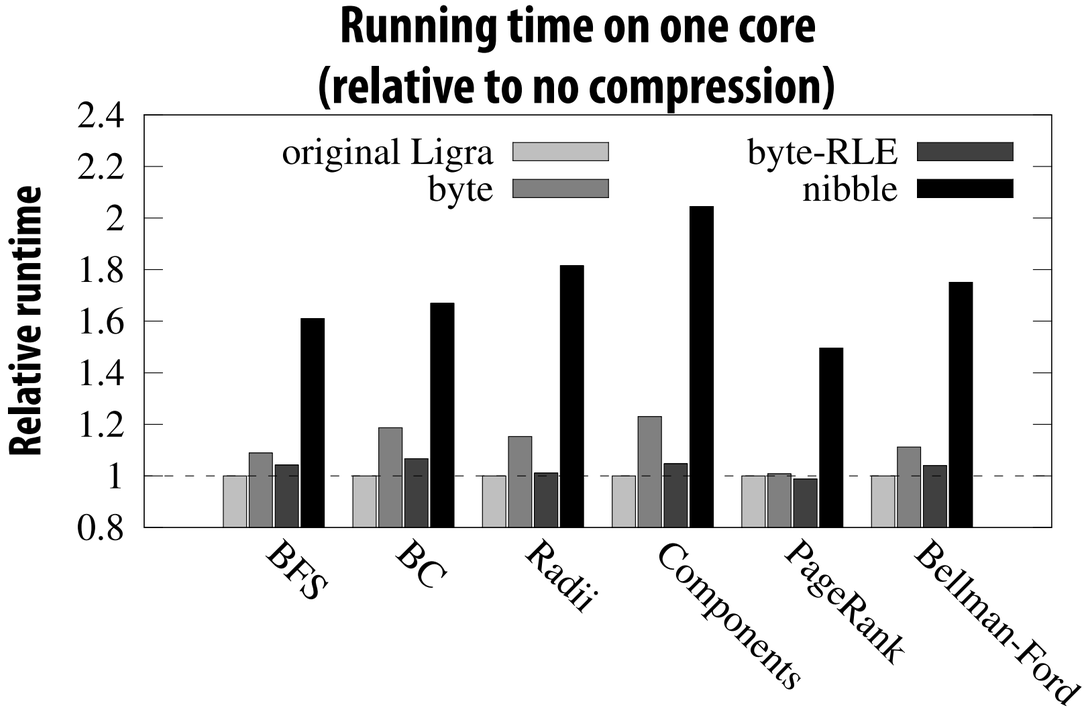
Compressed encoding: 26 bytes

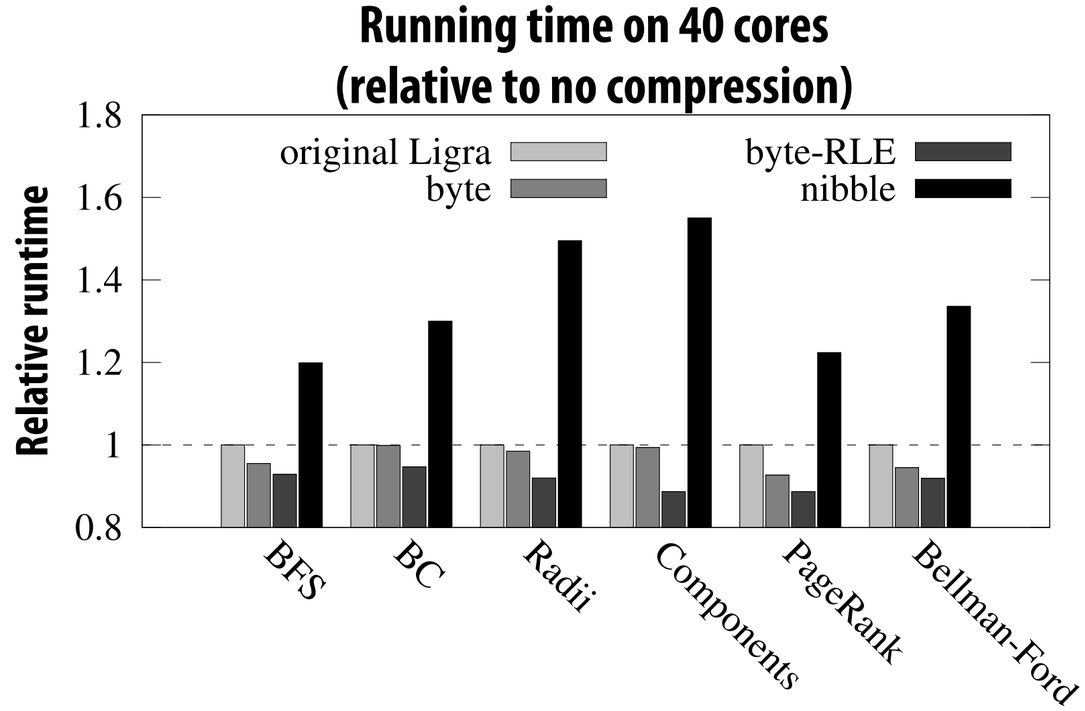



```
[ONE_BYTE, 4], -27, 1, 4, 20                (5 bytes)
[TWO_BYTE, 1], 971                          (3 bytes)
[ONE_BYTE, 4], 9, 14, 1, 5                  (5 bytes)
[FOUR_BYTE, 3], 98070, 100000, 75000        (13 bytes)
```

Performance impact of graph compression

[Shun et al. DDC 2015]





- Benefit of graph compression increases with higher core count, since computation is increasingly bandwidth bound
- Performance improves even if graphs already fit in memory
  - Added benefit is that compression enables larger graphs to fit in memory

* Different data points on graphs are different compression schemes (byte-RLE is the scheme on the previous slide)

Summary

- Analyzing large graphs is a workload of high interest
- High performance execution requires
  - Parallelism (complexity emerges from need to synchronize updates to shared vertices or edges)
  - Locality optimizations (restructure graph for efficient I/O)
  - Graph compression (reduce amount memory BW or disk I/O)
- Graph-processing frameworks handle many of these details, while presenting the application programmer with domain-specific abstractions that make it easy to express graph analysis operations