Lecture 23:

Domain-specific programming on graphs

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2015

Tunes

Alt-J

Tessellate

(An Awesome Wave)

“We wrote the lyrics to Tessellate while waiting on our GraphLab code to finish cranking on the Twitter graph.”

- Joe Newman

Recall from last time

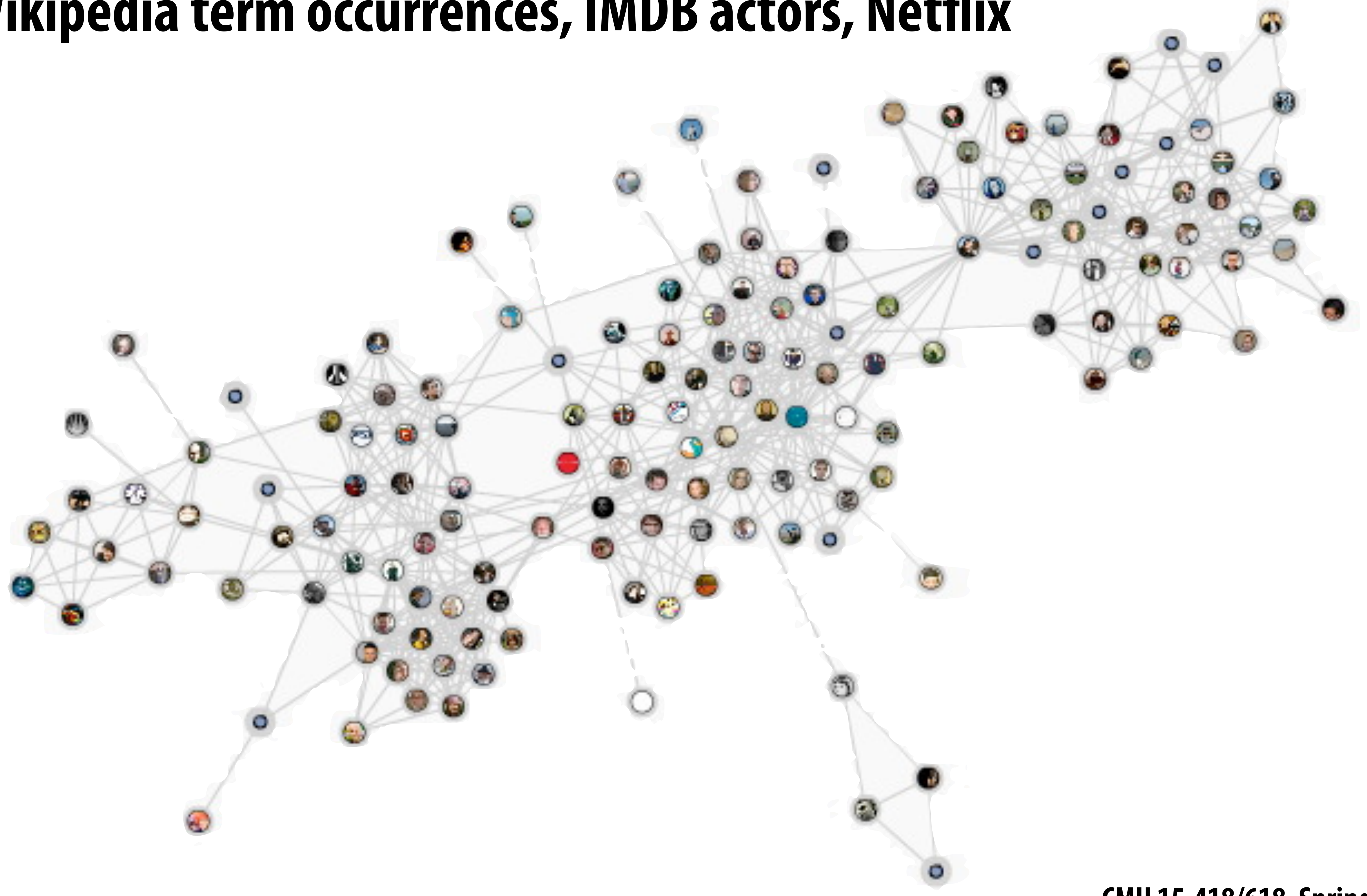
- Increasing acceptance of domain-specific programming systems
  - Challenge: modern computers are parallel, heterogeneous machines (HW architects striving for high area and power efficiency)
  - Programming trend: give up generality in what types of programs can be expressed in exchange for achieving both high programmer productivity and high performance
  - "Performance portability" is a key goal: want programs to execute efficiently on a variety of complex parallel platforms
    - Doing so can require different system implementations to use different data structures and algorithms, not just differ in low-level code generation (e.g., 4-wide vs. 8-wide SIMD)

Analyzing big graphs

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

- **Many public dataset examples:**

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



Today

- Three modern systems for expressing computations on graphs
- We'll use these systems as examples of making design choices when architecting programming systems

GraphLab

Ligra

Green-Marl

(Interested students may also want to take a look at: GraphX and Pregel)

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

What abstractions do we need?

Programming system design questions:

- **What are the fundamental operations do we want to be easy to express and efficient to execute?**
- **What are the key optimizations performed by best-known implementations of these operations? (system should not prevent these optimizations)**

Common illustrative example:
Page Rank

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

Map-Reduce (Hadoop) abstraction was the wrong tool for the job

(Iterative in-memory graph computations did not map well to "map" of independent computations followed by a dump of intermediate results to HDFS)

Whenever I'm trying to understand a new programming system, I ask two questions:

"What problems does the system take off the hands of the programmer? (and are these problems challenging/tedious enough for me to feel like 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)

Halide programming system (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, emitting loop indexing code)

Liszt programming system (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 parallelism algorithms on different platforms)
- Choose graph data structure / layout, partition graph across parallel machine, emit communication (MPI send), allocate ghost cells, etc.

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

GraphLab

Low et al. 2010
www.graphlab.org

- A system for describing iterative computations on graphs
- Implemented as a C++ runtime
- 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

Low et al. 2010
www.graphlab.org

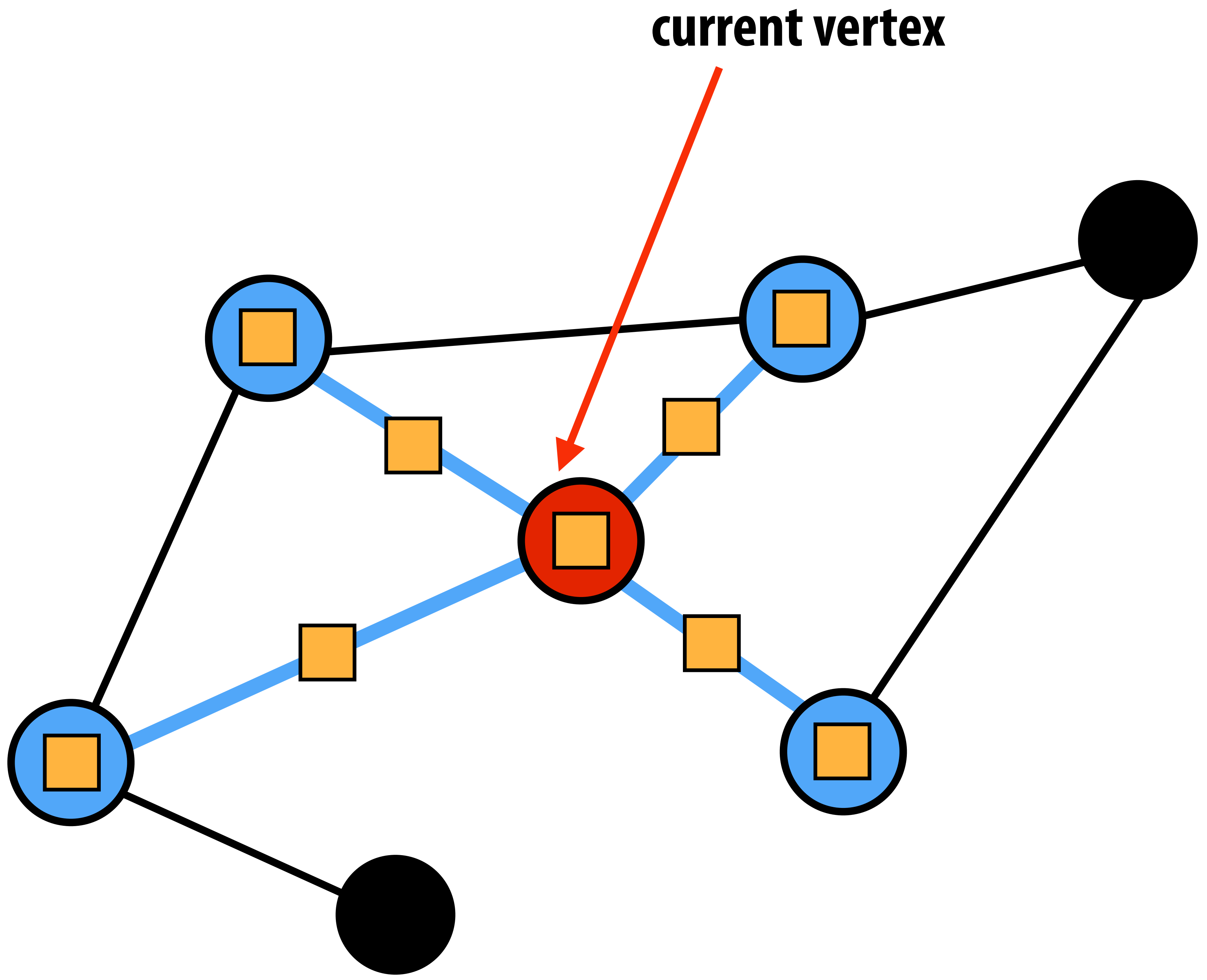
- **Application state:**
  - **The graph: G = (V, E)**
    - **Application defines data blocks on each vertex and directed edge**
    - D_v **= data associated with vertex** v
    - $D_{u \rightarrow v}$ **= data associated with directed edge** $u \rightarrow v$
  - **Read-only global data (can think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

Notice: I always first describe program state

And then describe what operations are available to manipulate state

Key concept: GraphLab vertex program

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



🟧 = 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 state at each vertex
Systems handles scheduling and parallelization.

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

Actual graphLab code (C++)

```cpp
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<graph_type, double>,
            public graphlab::IS_POD_TYPE {
public:
  // we are going to gather on 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.

Generating new work by signaling

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

- **Iterate 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 is a 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<graph_type, double>,
           public graphlab::IS_POD_TYPE {

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());
  }
};
```

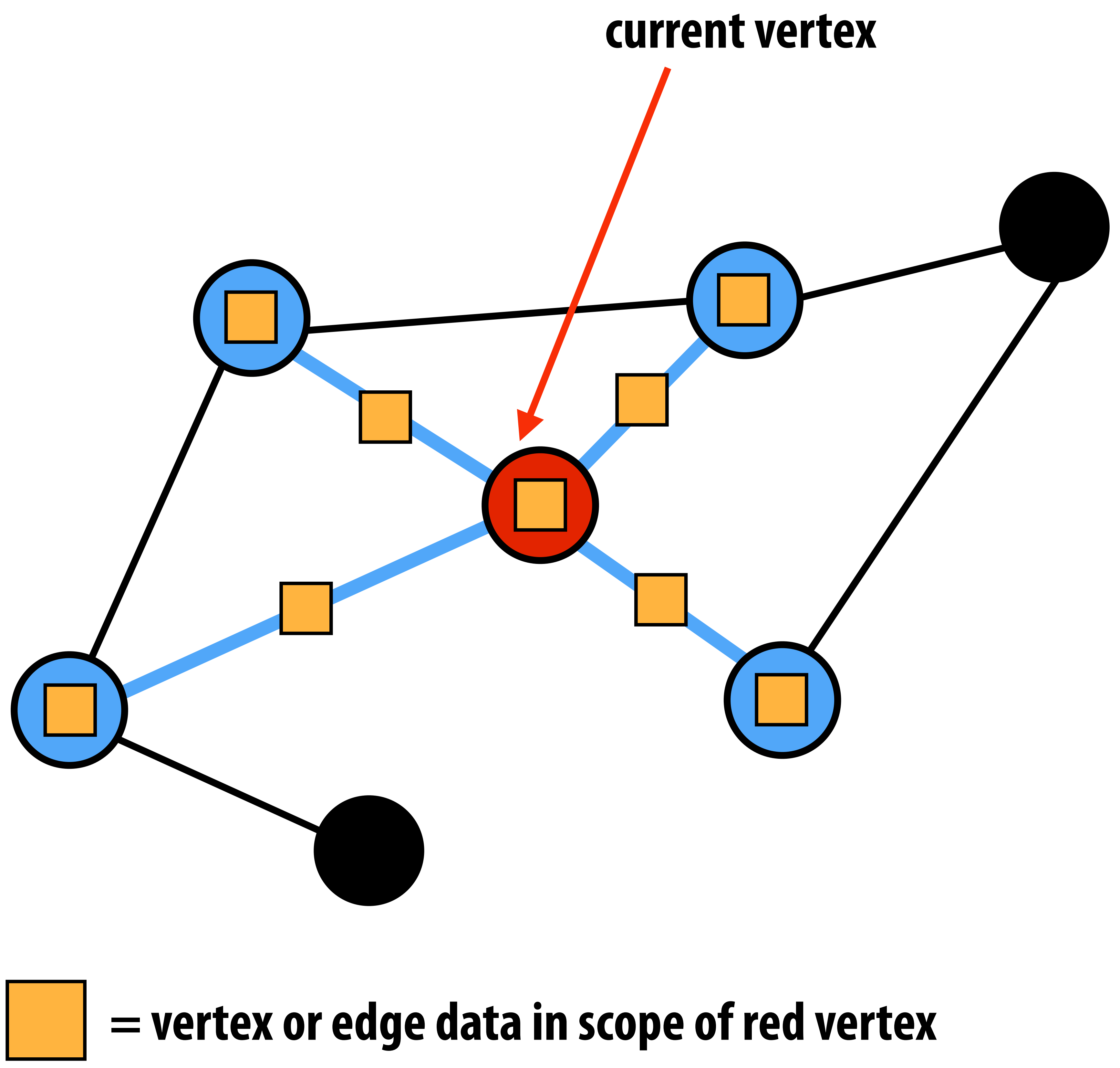
Private variable set during apply phase, used during scatter phase

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



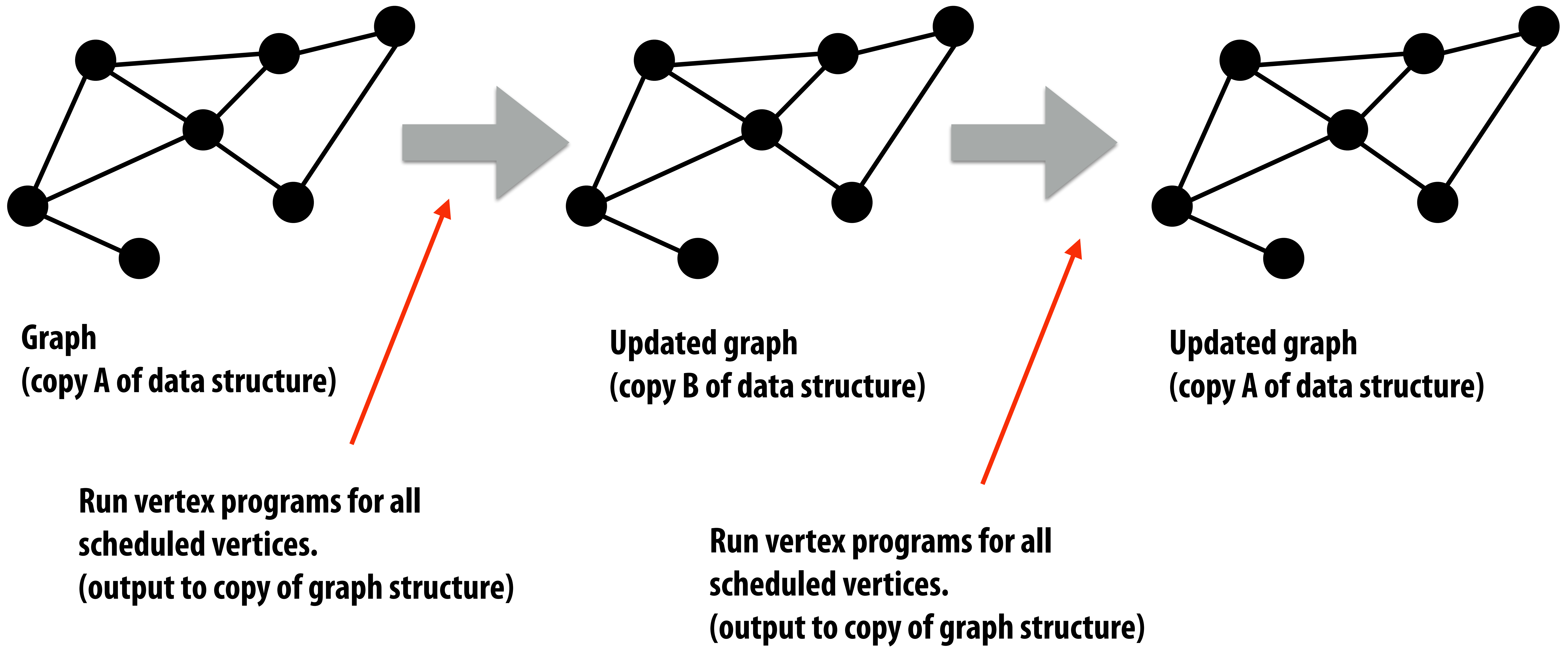
🟧 = vertex or edge data in scope of red vertex

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

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

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")


Graph
(copy A of data structure)
Updated graph
(copy B of data structure)
Updated graph
(copy A of data structure)
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)

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: programs make assumptions about the schedule for correctness (unlike Halide, Lizst, many other parallel systems we have studied)
  - 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 ⊂ V = {0, 4, 9}

graph

set of vertices

condition check on neighbor vertex

update function on neighbor vertex

```
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;
```

```
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);
```

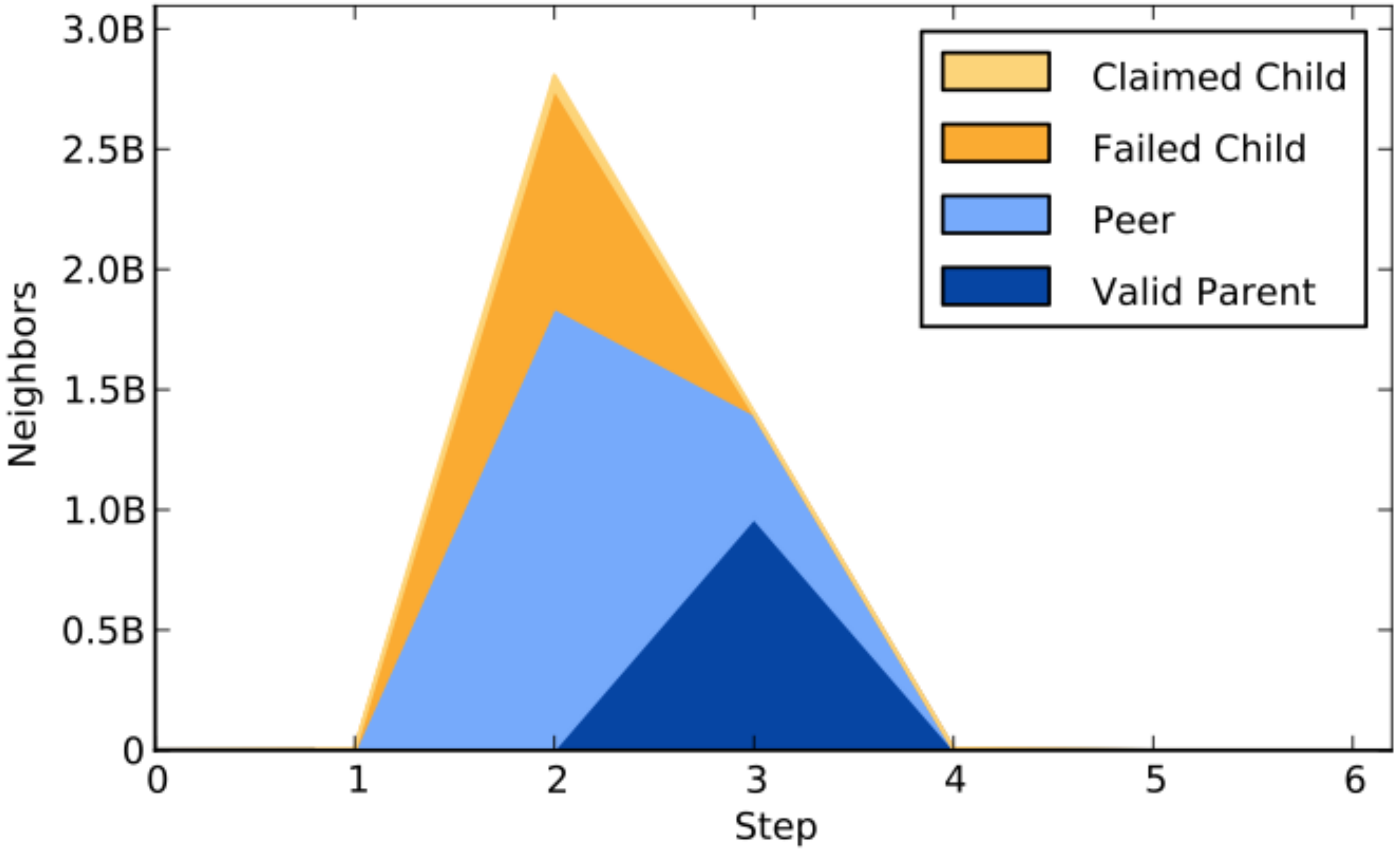
Cost of EDGEMAP_SPARSE?

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

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 <u>some</u> 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 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]

Green-Marl

[Hong 2012]
https://github.com/stanford-ppl/Green-Marl

A domain-specific language for computations on graphs

```
Procedure PageRank(G: Graph, thresh,
                   alpha: Double,
                   max_iter: Int,
                   PR: Node_Prop<Double>(G))
{
  Double diff = 0;
  Int cnt = 0;
  Double N = G.NumNodes();
  G.PR = 1 / N:
  Do {
     diff = 0.0;
     Foreach (n1: G.nodes) {
        Double val = (1 - alpha) / N  + alpha * sum(n2: n1.InNBrs) (n2.PR / n2.outDegree());
        n1.PR <= val @ n1;  // modification not visible until end of n1 loop
        diff += |val - n1.PR|;
     }
     cnt++;
  }  While (diff > thresh && cnt < max_iter);
}
```

Unlike GraphLab and Ligra, The Green-Marl programmer explicitly describes iteration (syntax looks more like Liszt)

Graph-specific iteration constructs

- **Betweenness-centrality example:**

- **Iteration over sets**
- **BFS/DFS iteration over graphs**

```
Procedure Compute_BC(
 G: Graph, BC: Node_Prop<Float>(G)) {
  G.BC = 0;        // initialize BC
  Foreach(s: G.Nodes) {
   // define temporary properties
    Node_Prop<Float>(G) Sigma;
    Node_Prop<Float>(G) Delta;
    s.Sigma = 1; // Initialize Sigma for root
   // Traverse graph in BFS-order from s
    InBFS(v: G.Nodes From s)(v!=s) {
      // sum over BFS-parents
      v.Sigma = Sum(w: v.UpNbrs) {w.Sigma};
    }
   // Traverse graph in reverse BFS-order
    InRBFS(v!=s) {
      // sum over BFS-children
      v.Delta = Sum (w:v.DownNbrs) {
         v.Sigma / w.Sigma * (1+ w.Delta)
      };
      v.BC += v.Delta @s; //accumulate BC
} } }
```

Summary: three domain-specific systems for expressing operations on graphs

- GraphLab
  - Programmer "thinks" about vertices exchanging data
  - Asynchronous execution as a key feature (exact results not needed in many ML algorithms)
- Ligra
  - Programmer "thinks" about graph traversal (computation happens when code "traverses" to node)
  - Traversal expressed using flat data-parallel operators
- Green-Marl
  - Add graph-specific iteration concepts to a language
  - Programmer "thinks" about traversal, but codes it up him/herself
  - Compiler smarts are largely in optimizing application-specified iteration

Not discussed today:
Pregel, GraphX

Elements of good domain-specific programming system design

#1: good systems identify and “win” in the most important cases

- **Efficient thinking: code written in a manner that mimics the fundamental structure of the problem**

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

- **Efficient implementation: most important optimizations are performed by the system for the programmer**
  - My experience: a parallel programming system with “convenient” abstractions that precludes best-known implementation strategies 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
    - Option: but GraphLab's design gets messy with all the scheduling options
  - Halide: only a few scheduling primitives
  - Hadoop: map + reduce

- Allows focus on optimizing these primitives
  - Ligra example: sparse vs. dense optimization (developed for BFS) but is applied all algorithms written using EDGEMAP/VERTEXMAP

- Do we really need that? (can it be reduced to a primitive we already have)
  - Want a performance or expressivity justification for every primitive

#3: good primitives compose

- **Composition/use 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

- **Composition often allows for generalizable optimization**

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

- **Sign that a new feature <u>should not</u> be added (or added in a better way):**
  - It does not compose with all existing features

Streaming graph computations

(now we are talking about implementation)

Streaming graph computations

- Would like to process large graphs on a single machine
  - Managing clusters of machines is difficult
  - Partitioning graphs is expensive and difficult
- Challenge: cannot fit all edges in memory for large graphs (although all vertices may fit)
  - Example: 1 billion edge graph
  - Consider sparse representation of graph from Assignment 3: each edge represented twice in structure (incoming/outgoing): 8 bytes per edge for adjacency structure
  - Must also store per-edge values (e.g., 4 bytes for a per-edge weight)
  - ~12 GB of memory for edge information
  - Graph algorithms traverse edges, which is random access (single operation on a graph might require billions of tiny loads from disk)

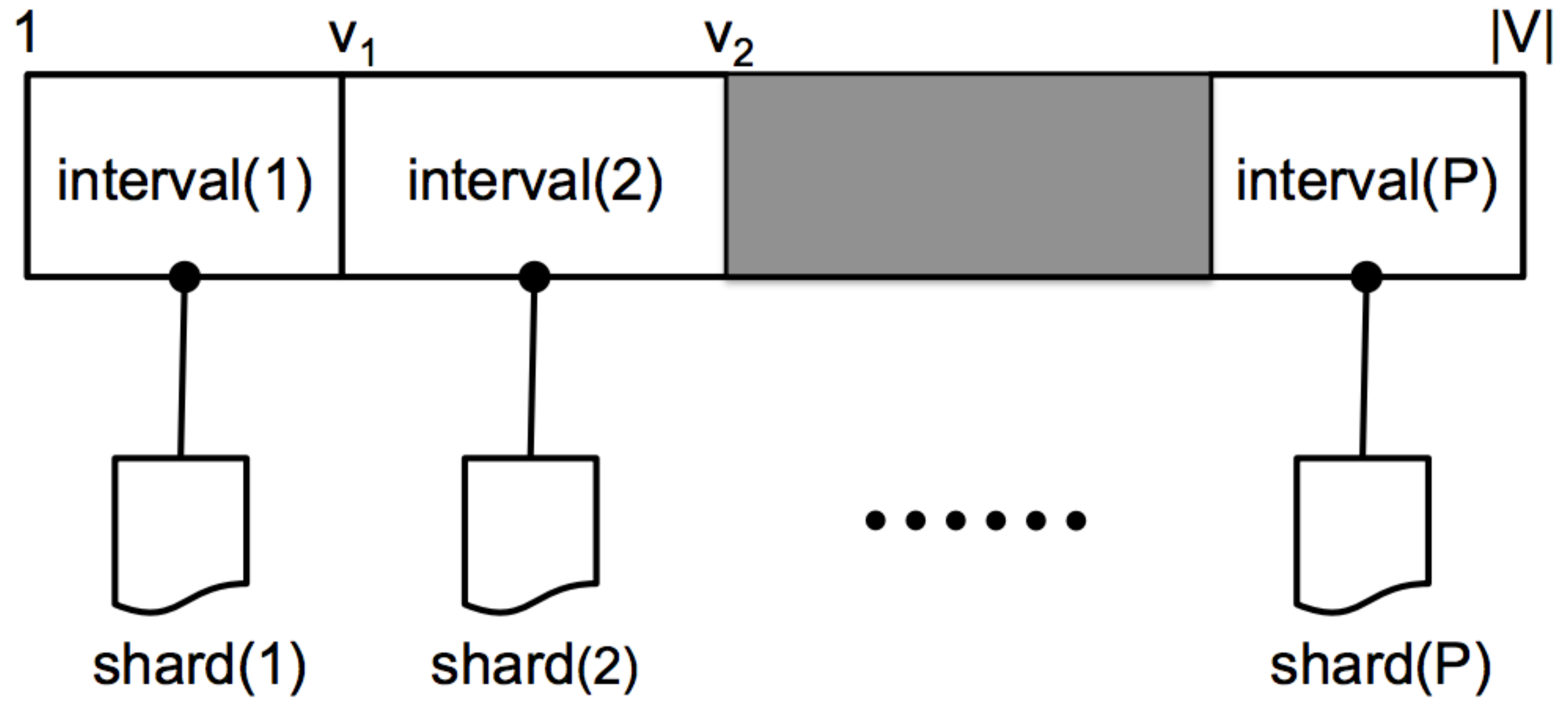
Streaming graph computations

- Would like to process large graphs on a single machine
- Challenge: cannot fit all edges in memory for large graphs (although all vertices may fit)
- Caching subset of vertices in memory is unpredictable/difficult, clustering/partitioning also requires significant amounts of memory as well…

Parallel sliding window approach

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

- Assumption: graph operations to update a vertex require only immediate neighboring information in the graph
- Main idea: organize the graph data structure so that graph operations require only a small number of large, bulk loads/stores to disk
  - Partition graph vertices into P intervals
  - Store vertices and only <u>incoming edges</u> to these vertices are stored together in a shard (total of P shards)



Sharded graph representation

- Partition graph vertices into intervals (sized so that each shard fits in memory)
- Store vertices and only <u>incoming edges</u> to these vertices are stored together in a shard
- Sort edges in a shard by source vertex id

Interval 1:
vertices 1, 2

Shard 1

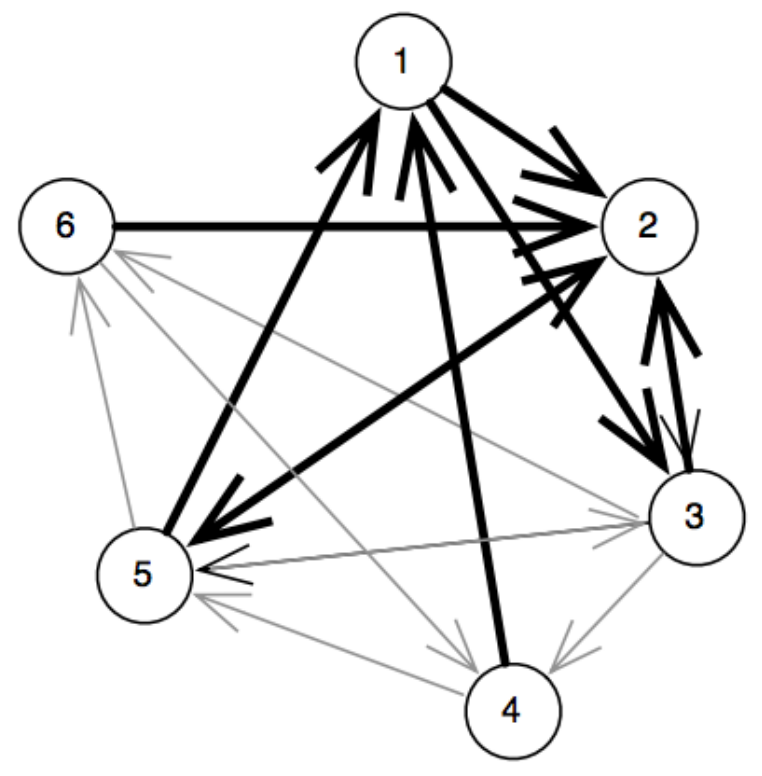
src	dst	value
1		
	2	0.3
3		
	2	0.2
4		
	1	1.4
5		
	1	0.5
	2	0.6
6		
	2	0.8

Shard 2

src	dst	value
1		
	3	0.4
2		
	3	0.3
3		
	4	0.8
5		
	3	0.2
6		
	4	1.9

Shard 3

src	dst	value
2		
	5	0.6
3		
	5	0.9
	6	1.2
4		
	5	0.3
5		
	6	1.1



Data required to process vertices in interval 1

Notice: to construct subgraph containing vertices in interval 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

- Partition graph vertices into intervals (sized so that each shard fits in memory)
- Store vertices and only <u>incoming edges</u> to these vertices are stored together in a shard
- Sort edges in a shard by source vertex id

Interval 2:
vertices 3, 4

Shard 1

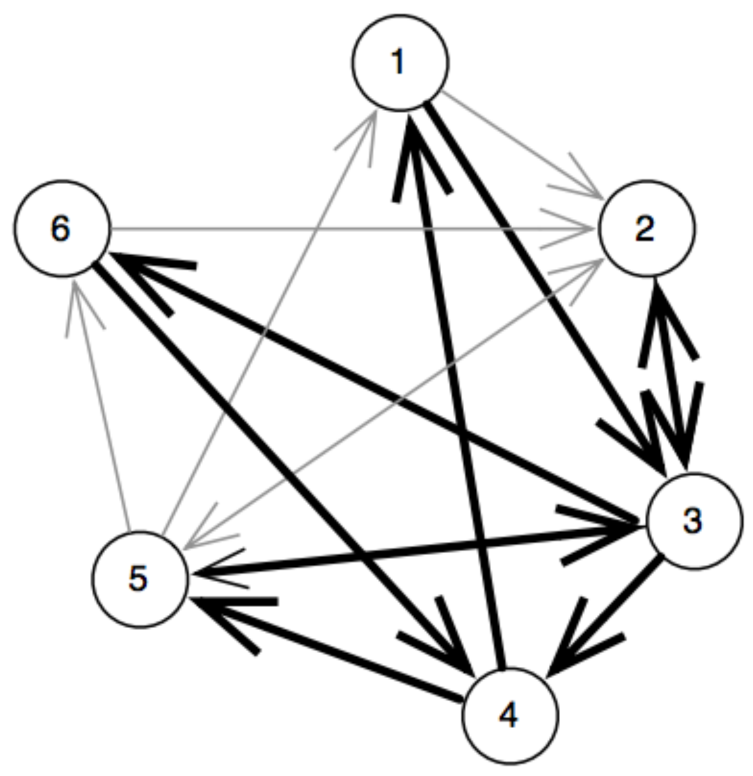
src	dst	value
1		
	2	0.273
3		
	2	0.22
4		
	1	1.54
5		
	1	0.55
	2	0.66
6		
	2	0.88

Shard 2

src	dst	value
1		
	3	0.364
2		
	3	0.273
3		
	4	0.8
5		
	3	0.2
6		
	4	1.9

Shard 3

src	dst	value
2		
	5	0.545
3		
	5	0.9
	6	1.2
4		
	5	0.3
5		
	6	1.1



Data required to process vertices in interval 2

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

PageRank in GraphChi

- **GraphChi is a system that implements the out-of-core sliding window approach**

PageRank in graphChi

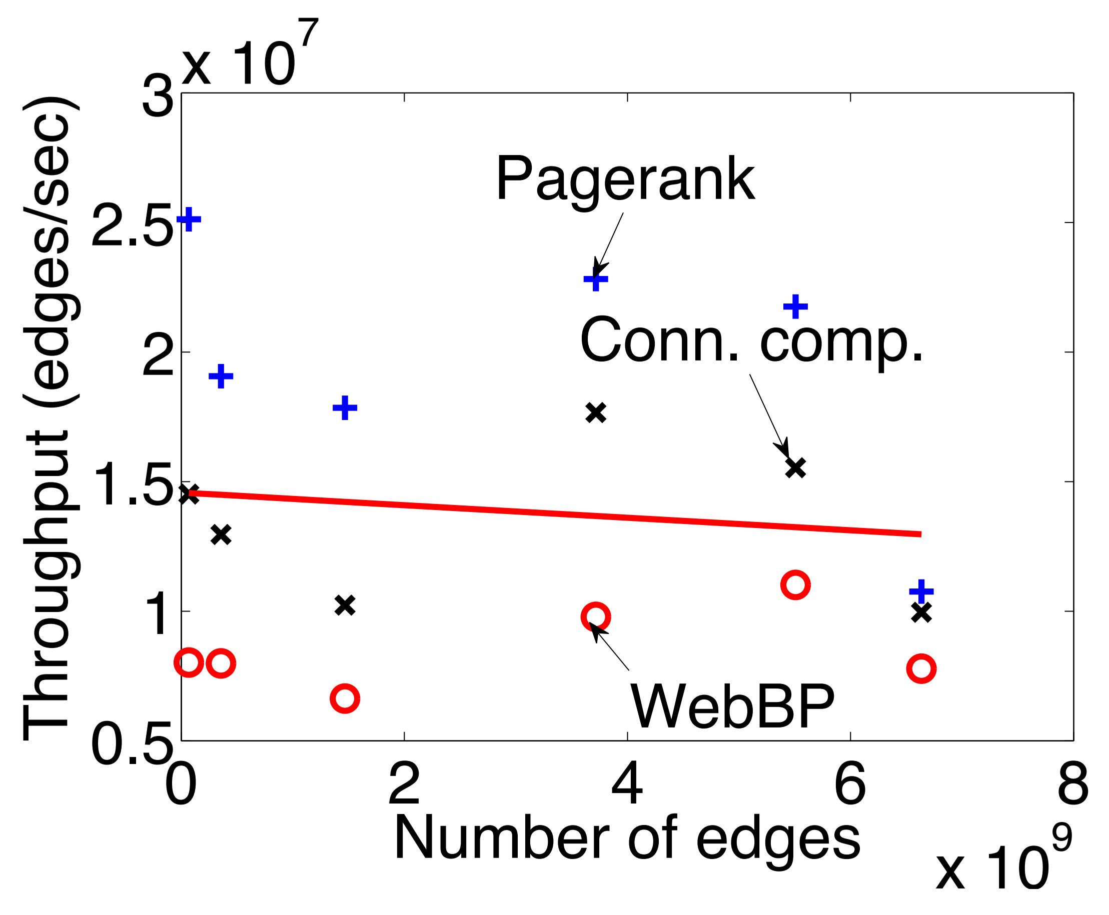
```
typedef: VertexType float
Update(vertex) begin
    var sum ← 0
    for e in vertex.inEdges() do
        sum += e.weight * neighborRank(e)
    end
    vertex.setValue(0.15 + 0.85 * sum)
    broadcast(vertex)
end
```

Take per-vertex rank and distribute to all outbound edges (memory inefficient: replicates per-vertex rank to all edges)

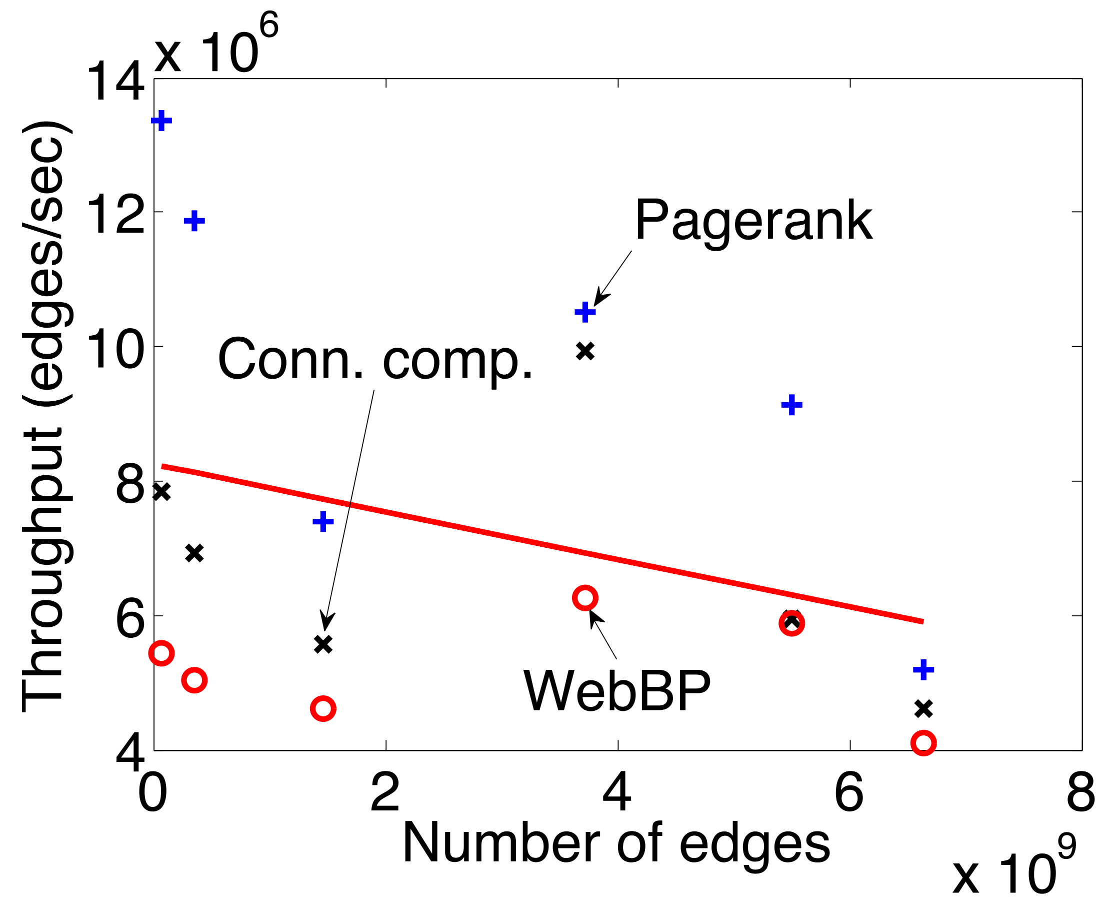
Alternative model: assume vertex data can be kept in memory and redefine neighborRank() function

```
typedef: EdgeType { float weight; }
float[] in_mem_vert
neighborRank(edge) begin
    return edge.weight * in_mem_vert[edge.vertex_id]
end
```

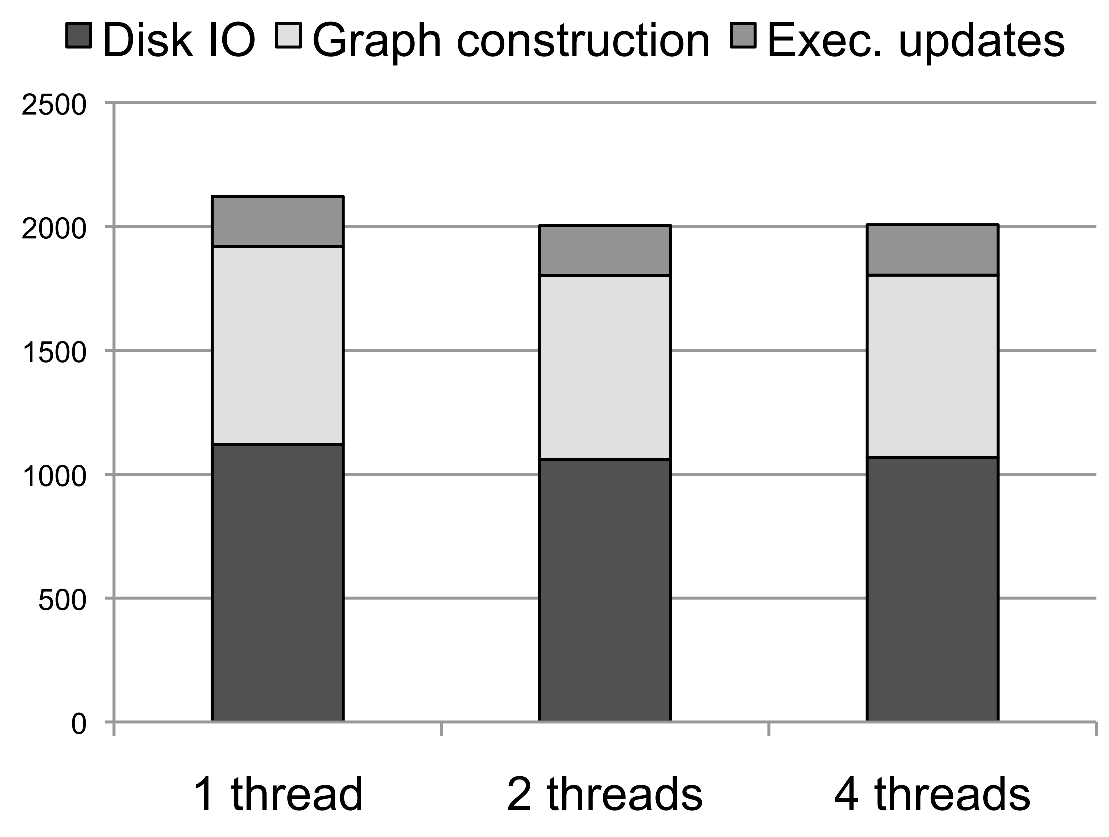
Performance on a Mac mini (8 GB RAM)



(a) Performance: SSD



(b) Performance : Hard drive



(c) Runtime breakdown

- **Performance remains stable as graph size is increased**
  - **Desirable property: throughput largely invariant of dataset size.**