Recall last time

- Domain-specific programming systems
  - Idea: give up generality in exchange for achieving programmer productivity and high performance
  - **Performance portability** as a key goal: want ability to run programs efficiently across a variety of complex parallel platforms (wide diversity in modern platforms)
  - Doing so requires different data structure and algorithmic choices: not just good low-level code generation
Today

- Three recent systems for expressing operations on graphs
- We’ll use these system as examples of making design choices when architecting a system
Analyzing big graphs

- Public dataset examples:
  - Twitter social graph, Wikipedia term occurrences, IMDB actors, Netflix

MapReduce (Hadoop) abstraction was the wrong tool for the job
(e.g., iterative graph computations did not map well to “map” of independent computations)

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

Illustrative example:
Page rank
Designing a framework for graph computations

- What operations do we want to make easy to express and efficient?
- What are key optimizations?

Illustrative example:
Page rank

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

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

Application state:
- The graph: $G = (V, E)$
  - Application defines data blocks on each vertex and directed edge
  - $D_v = \text{data associated with vertex } v$
  - $D_{u \rightarrow v} = \text{data associated with directed edge } u \rightarrow v$
- Read-only global data
Simple example: Page Rank *

- **Per-vertex update functions**
  - Read and write data associated with vertex’s local neighborhood
  - Neighborhood = local vertex data + adjacent (inbound/outbound) edges + neighboring vertices

- **Built in global reductions (not shown in this example)**

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

```
PageRank_vertex_program(vertex i) {

  // (Gather) 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) Update my rank (i)
  i.rank = (1-ALPHA)/num_vertices + ALPHA*sum;
}
```

*My slides show GraphLab 1.0 style program for simplicity
(pseudocode given for clarity: actual syntax is C++)*
Iterate until convergence

```
PageRank_vertex_program(vertex i) {
  // (Gather) 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) Update my Rank (i)
  double old_rank = i.rank;
  i.rank = (1-ALPHA)/num_vertices + ALPHA*sum;
  i.change = fabs(i.rank - old_rank);
}
```

**Main program:**

```
// setup graph and vertex program here

double global_change;

while (!done);
  signalall(); // execute program for all vertices
  global_change = reduce(change); // graphLab reduction “sync” operation
  if (global_change < threshold)
    done = true;
}
```
Dynamic scheduling

- Different parts of graph may converge at different rates
  - For large graph, processing all vertices every iteration is wasteful

```java
PageRank_vertex_program(vertex i) {
  // (Gather) 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) Update my Rank (i)
  i.old_rank = i.rank;
  i.rank = (1-ALPHA)/num_vertices + ALPHA*sum;

  // (Scatter) If necessary, signal my neighbors to recompute their rank
  if (abs(i.old_rank - i.rank) > EPSILON) {
    foreach (vertex j : out_neighbors(i))
      signal(j);
  }
}
```

- Can consider GraphLab runtime engine as processing a work queue of jobs (executing a vertex program is a job)
  - Departure from bulk, data-parallel model of execution
Synchronizing parallel execution

- Local neighborhood of vertex (vertex’s “scope”) can be read and written to by a vertex program.

Programs specify what granularity of atomicity (“consistency”) they want GraphLab runtime to provide: determines amount of 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 \( v \) ...
Job scheduling order

- **GraphLab supports a collection of 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**
  - With edge consistency, program may be correct with any scheduler
  - With graph coloring scheduler, can be correct with only vertex consistency
  - Implication: programs make assumptions about the schedule (unlike Lizst)
    - Seems weird at first, but this is true of graph algorithms
GraphLab summary

- **State:** data on graph vertices and edges + globals
- **Operations:** per-vertex update programs and reduction (“sync”) functions
  - 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

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

```plaintext
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);

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

```plaintext
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;
```

Cost of EDGEMAP_SPARSE?
$O(|U| + \text{sum of outgoing edges from } U)$
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!

- **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): \( U \subset V = \{1,0,0,0,1,0,0,0,0,1\} \)

```plaintext
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")

```plaintext
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;
```
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);

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

\[ r_{\text{cur}} = \{1/|V|, \ldots, 1/|V|\}; \]
\[ r_{\text{next}} = \{0, \ldots, 0\}; \]
\[ \text{diff} = \{\} \]

procedure PRUPDATE(s, d):
    atomicIncrement(&r_{\text{next}}[d], r_{\text{cur}}[s] / \text{vertex}_{\text{degree}}(s))

procedure PRLocalCompute(i):
    r_{\text{next}}[i] = \alpha \times r_{\text{next}}[i] = (1 - \alpha) / |V|;
    \text{diff}[i] = |r_{\text{next}}[i] - r_{\text{cur}}[i]|;
    r_{\text{cur}}[i] = 0;
    return 1;

procedure COND(i):
    return 1;

procedure PAGERANK(G, \alpha, \epsilon):
    \text{frontier} = \{0, \ldots, |V|-1\}
    \text{error} = \text{HUGE};
    \text{while (error > \epsilon) do:}
        \text{frontier} = \text{EDGEMAP}(G, \text{frontier}, \text{PRUPDATE}, \text{COND});
        \text{frontier} = \text{VERTEXMAP}(\text{frontier}, \text{PRLocalCompute});
        \text{error} = \text{sum of diffs} \quad // \text{this is a parallel reduce}
        \text{swap}(r_{\text{cur}}, r_{\text{next}});
    \text{return err}

Question: can you implement GraphLab’s sparse update optimization?

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

- Abstract 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)

- Further algorithm scope:
  - Betweenness centrality
  - Connected components
  - Shortest paths

See Ligra: a Lightweight Framework for Graph Processing for Shared Memory [Shun and Blelloch 2013]
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 (t: G.nodes) {
      Double val = (1 - alpha) / N + alpha * sum(w: t.InNBrss) (w.PR / w.outDegree());
      t.PR <= val & t; // modification not visible until end of t loop
      diff += |val - t.PR|;
    }
    cnt++;
  } While (diff > thresh && cnt < max_iter);
}
Graph-specific iteration

- Betweenness-centrality example:

- Iteration over sets

- BFS/DFS iteration over graphs

```c
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 more 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
Elements of a good programming system design

- **Simple:**
  - A small number of key primitives and operations
  - Ligra: only two operations!
  - GraphLab: run computation on vertex, force neighbors to run
    - Design gets messy with all the scheduling options
  - Few primitives = focus optimization on these primitives
    - Ligra example: sparse vs. dense optimization (developed for BFS) but is applied all algorithms written using EDGEMAP/VERTEXMAP

- **Expressive:**
  - Composition/use of primitives allows for broad space of uses (wide application scope, even if it is limited to a domain)

- **Optimized for the common case**
  - Expression: common operations are easy to express, intuitive, and efficient
  - Most important optimizations can be performed