Lecture 23:

Domain-specific programming on graphs
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 last time

- Domain-specific programming systems
  - Idea: give up generality in what types of programs can be expressed in exchange for achieving programmer productivity and high performance
  - “Performance portability” is a key goal: want programs to execute efficiently on a variety of complex parallel platforms (recall: wide diversity in modern platforms)
  - Doing so can require different data structure and algorithmic choices: not just good low-level code generation
Today

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

GraphLab

Ligra

Green-Marl
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
Designing a framework for writing programs that graph operations

- What operations do we want to make easy to express and efficient?
- What are the key optimizations performed by best-known implementations of these operations?

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
GraphLab

- 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 graph partitioning across cluster machines, communication between master, work scheduling, etc.
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

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
Simple example: PageRank *

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

* Let \( \alpha = 0.85 \)

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;
}

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

Programming in GraphLab amounts to defining how to update state at each vertex
Systems handles scheduling/parallelization
Actual graphLab code (C++)

```
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) {
        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

```cpp
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)

```cpp
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();
}
```

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)

```cpp
class pagerank_program:
    public graphlab::ivertex_program<
graph_type,	
    double>,
    public graphlab::IS_POD_TYPE {

private:
    bool perform_scatter;  // Private variable set during apply phase, used during scatter phase

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);  // Check for convergence
    }

    // 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());
    }
};
```
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 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)
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
  - Implication: programs make assumptions about the schedule (unlike Lizst)
  - Kayvon’s opinion: 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

- 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 $G=(E,V)$: $U \subset V = \{0, 4, 9\}$

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 $\{0,\ldots,|V|-1\}$ do:
        if (C($i$) == 1) then:
            foreach $v$ in in_neighbors($i$) do:
                if $v \in U$ and F($v$, $i$) == 1 then:
                    add $i$ to result;
        if (C($i$) == 0) break;
    return result;
```

```plaintext
procedure EDGEMAP_SPARSE(G, U, F, C):
    result = {}
    parallel foreach $v$ in U do:
        parallel foreach $v_2$ in out_neighbors($v$) do:
            if (C($v_2$) == 1 and F($v$, $v_2$) == 1) then:
                add $v_2$ to result
                remove duplicates from result
    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

```javascript
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

\[
\begin{align*}
    r_{cur} &= \{1/|V|, \ldots, 1/|V|\}; \\
    r_{next} &= \{0, \ldots, 0\}; \\
    \text{diff} &= \{} \\
\end{align*}
\]

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

procedure PRLOCALCOMPUTE(i):
    r_{next}[i] = alpha \cdot r_{next}[i] + (1 - alpha) / |V|;
    \text{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, \ldots, |V|\} - 1
    error = HUGE;
    while (error > eps) do:
        frontier = EDGEMAP(G, frontier, PRUPDATE, COND);
        frontier = VERTEXMAP(frontier, PRLOCALCOMPUTE);
        error = \text{sum of diffs} // this is a parallel reduce
        \text{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

- 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)

- These basic operations permit surprising 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

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.InNBrs) (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

```plaintext
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

**Not discussed today:**
Google’s Pregel
Elements of good programming system design

- **Simple:**
  - A small number of key primitives and operations
    - Ligra: only two operations!
    - GraphLab: run computation per vertex, force neighbors to run using signaling
      - 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 by system
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 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

- 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 incoming edges 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 incoming edges to these vertices are stored together in a shard
- Sort edges in a shard by source vertex id

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 incoming edges to these vertices are stored together in a shard
- Sort edges in a shard by source vertex id

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**

```plaintext
1 typedef: VertexType float
2 Update(vertex) begin
3     var sum ← 0
4     for e in vertex.inEdges() do
5         sum += e.weight * neighborRank(e)
6     end
7     vertex.setValue(0.15 + 0.85 * sum)
8     broadcast(vertex)
9 end
```

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

```plaintext
1 typedef: EdgeType { float weight; }
2 float[] in_mem_vert
3 neighborRank(edge) begin
4     return edge.weight * in_mem_vert[edge.vertex_id]
5 end
```

Take per-vertex rank and distribute to all outbound edges (memory inefficient: replicates per-vertex rank to all edges)
Performance on a Mac mini (8 GB RAM)

- Performance remains stable as graph size is increased