A Framework for Processing Large Graphs in Shared Memory

Julian Shun
UC Berkeley (Miller Institute, AMPLab)

Based on joint work with Guy Blelloch and Laxman Dhulipala, Kimon Fountoulakis, Michael Mahoney, and Farbod Roosta-Khorasani
Graphs are everywhere!

- Can contain up to billions of vertices and edges
- Need simple, efficient, and scalable way to analyze them
Ligra Graph Processing Framework

**EdgeMap**
- Breadth-first search
- Betweenness centrality
- Connected components
- Triangle counting
- K-core decomposition
- Maximal independent set
  ...

**VertexMap**
- Single-source shortest paths
- Eccentricity estimation
- (Personalized) PageRank
- Local graph clustering
- Biconnected components
- Collaborative filtering
  ...

*Simplicity, Performance, Scalability*
Graph Processing Systems

• Existing: Pregel/Giraph/GPS, GraphLab, PRISM, Pegasus, Knowledge Discovery Toolbox, GraphChi, GraphX, and many others…

• Our system: Ligra - Lightweight graph processing system for shared memory

Takes advantage of “frontier-based” nature of many algorithms (active set is dynamic and often small)
Breadth-first Search (BFS)

- Compute a BFS tree rooted at source $r$ containing all vertices reachable from $r$

Applications

- Betweenness centrality
- Eccentricity estimation
- Maximum flow
- Web crawlers
- Network broadcasting
- Cycle detection

- Can process each level in parallel
- Race conditions, load balancing
Steps for Graph Traversal

- Operate on a subset of vertices
- Map computation over subset of edges \textit{in parallel}
- Return new subset of vertices
- Map computation over subset of vertices \textit{in parallel}

\textbf{We built the Ligra abstraction for these kinds of computations}

Think with flat data-parallel operators

Optimizations:
- hybrid traversal
- graph compression
Breadth-first Search in Ligra

parents = {-1, ..., -1};  /*-1 indicates “unexplored”*/

procedure UPDATE(s, d):
  return compare_and_swap(parents[d], -1, s);

procedure COND(v):
  return parents[v] == -1;  /*checks if “unexplored”*/

procedure BFS(G, r):
  parents[r] = r;
  frontier = {r};  /*VertexSubset*/
  while (size(frontier) > 0):
    frontier = EDGEMAP(G, frontier, UPDATE, COND);
#include "ligra.h"

struct BFS_F {
    intT* Parents;
    BFS_F(intT* _Parents) : Parents(_Parents) {} 
    inline bool update (intT s, intT d) { //Update 
        if(Parents[d] == -1) { Parents[d] = s; return 1; }
        else return 0;
    }
    inline bool updateAtomic (intT s, intT d){ //atomic version of Update 
        return (CAS(&Parents[d],(intT)-1,s));
    }
    //cond function checks if vertex has been visited yet
    inline bool cond (intT d) { return (Parents[d] == -1); }
};

template <class vertex>
void Compute(graph<vertex> GA, intT start) {
    intT n = GA.n;
    //creates Parents array, initialized to all -1, except for start
    intT* Parents = newA(intT,GA.n);
    parallel_for(intT i=0;i<GA.n;i++) Parents[i] = -1;
    Parents[start] = start;

    vertexSubset Frontier(n,start); //creates initial frontier

    while(!Frontier.isEmpty()){
        //loop until frontier is empty
        vertexSubset output = edgeMap(GA, Frontier, BFS_F(Parents));
        Frontier.del();
        Frontier = output; //set new frontier
    }
    Frontier.del();
    free(Parents);
}
• Dense method better when frontier is large and many vertices have been visited

• Sparse (traditional) method better for small frontiers

• Switch between the two methods based on frontier size [Beamer et al. SC ’12]
EdgeMap

procedure **EDGEMAP**(*G*, frontier, Update, Cond):
    if (size(frontier) + sum of out-degrees > threshold) then:
        return **EDGEMAP_DENSE**(*G*, frontier, Update, Cond);
    else:
        return **EDGEMAP_SPARSE**(*G*, frontier, Update, Cond);

- More general than just BFS!
- Generalized to many other problems
- Users need not worry about this
Frontier-based approach enables sparse/dense traversal

Twitter graph (41M vertices, 1.5B edges)

- BFS
- Betweenness Centrality (30.7)
- Connected Components (20.7)
- Eccentricity Estimation

- Dense
- Sparse
- Sparse/Dense

40-core running time (seconds)

(using default threshold of |E|/20)
PageRank

\[ \text{PR}[v] = \frac{1 - \gamma}{|V|} + \gamma \sum_{u \in N^-(v)} \frac{\text{PR}[u]}{\text{deg}^+(u)} \]
bool f(v) {
    data[v] = data[v] + 1;
    return (data[v] == 1);
}
PageRank in Ligra

\[
p_{\text{curr}} = \{1/|V|, ..., 1/|V|\}; \quad p_{\text{next}} = \{0, ..., 0\}; \quad \text{diff} = \{\}; \quad \text{error} = \infty;
\]

procedure \textbf{UPDATE}(s, d):
\[
\text{atomic\_increment}(p_{\text{next}}[d], p_{\text{curr}}[s] / \text{degree}(s));
\]
\[
\text{return} 1;
\]

procedure \textbf{COMPUTE}(i):
\[
p_{\text{next}}[i] = \alpha \cdot p_{\text{next}}[i] + (1 - \alpha) \cdot (1/|V|);
\]
\[
\text{diff}[i] = \text{abs}(p_{\text{next}}[i] - p_{\text{curr}}[i]);
\]
\[
p_{\text{curr}}[i] = 0;
\]
\[
\text{return} 1;
\]

procedure \textbf{PageRank}(G, \alpha, \varepsilon):
\[
\text{frontier} = \{0, ..., |V| - 1\};
\]
while (error > \varepsilon):
\[
\text{frontier} = \text{EDGEMAP}(G, \text{frontier, UPDATE, COND}_{\text{true}});
\]
\[
\text{frontier} = \text{VERTEXMAP}(\text{frontier, COMPUTE});
\]
\[
\text{error} = \text{sum\ of\ diff\ entries};
\]
\[
\text{swap}(p_{\text{curr}}, p_{\text{next}})
\]
\[
\text{return} p_{\text{curr}};
\]
PageRank

• *Sparse version?*
  
  • PageRank-Delta: Only update vertices whose PageRank value has changed by more than some $\Delta$-fraction (discussed in GraphLab and McSherry WWW ‘05)
PageRank-Delta in Ligra

PR[i] = {1/|V|, ..., 1/|V|};
nghSum = {0, ..., 0};
Change = {};  //store changes in PageRank values

procedure UPDATE(s, d):  //passed to EdgeMap
    atomic_increment(nghSum[d], Change[s] / degree(s));
    return 1;

procedure COMPUTE(i):  //passed to VertexMap
    Change[i] = α · nghSum[i];
    PR[i] = PR[i] + Change[i];
    return (abs(Change[i]) > Δ);  //check if absolute value of change is big enough
Performance of Ligra
Ligra BFS Performance

• Comparing against direction-optimizing code by Beamer et al.
Ligra PageRank Performance

Twitter graph (41M vertices, 1.5B edges)

- GraphLab (64 x 8-cores)
- GraphLab (40-core machine)
- Ligra (40-core machine)
- Hand-written Cilk/OpenMP (40-core machine)

- Easy to implement “sparse” version of PageRank in Ligra
Ligra Connected Components Performance

Twitter graph (41M vertices, 1.5B edges)

- Performance close to hand-written code
- Faster than existing high-level frameworks at the time
- Shared-memory graph processing can be very efficient
  - Requires graph to fit on a single machine
### Large Graphs

**Amazon EC2**

<table>
<thead>
<tr>
<th>vCPU</th>
<th>ECU</th>
<th>Memory (GiB)</th>
<th>Instance Storage (GB)</th>
<th>Linux/UNIX Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory Optimized - Current Generation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1.32xlarge</td>
<td>128</td>
<td>349</td>
<td>1952</td>
<td>2 x 1920 SSD</td>
</tr>
<tr>
<td>r3.large</td>
<td>2</td>
<td>6.5</td>
<td>15</td>
<td>1 x 32 SSD</td>
</tr>
<tr>
<td>r3.xlarge</td>
<td>4</td>
<td>13</td>
<td>30.5</td>
<td>1 x 80 SSD</td>
</tr>
<tr>
<td>r3.2xlarge</td>
<td>8</td>
<td>26</td>
<td>61</td>
<td>1 x 160 SSD</td>
</tr>
<tr>
<td>r3.4xlarge</td>
<td>16</td>
<td>52</td>
<td>122</td>
<td>1 x 320 SSD</td>
</tr>
<tr>
<td>r3.8xlarge</td>
<td>32</td>
<td>104</td>
<td>244</td>
<td>2 x 320 SSD</td>
</tr>
</tbody>
</table>

- Most can fit on commodity shared memory machine
Ongoing Work: Local Graph Clustering

- Significant savings if output cluster is much smaller than entire graph
- We parallelized local algorithms using Ligra
  - Algorithms are frontier-based, processing small active sets on each iteration
- Ligra is crucial to getting local running time
Ligra Summary

VertexSubset  VertexMap  EdgeMap

Optimizations: Hybrid traversal and graph compression

Breadth-first search  Single-source shortest paths
Betweenness centrality  Eccentricity estimation
Connected components  (Personalized) PageRank
Triangle counting  Local graph clustering
K-core decomposition  Biconnected components
Maximal independent set  Collaborative filtering
...

Simplicity, Performance, Scalability
Thank you!

Code: https://github.com/jshun/ligra/