A Framework for Processing Large Graphs in Shared Memory

Julian Shun

Based on joint work with Guy Blelloch and Laxman Dhulipala
What are graphs?

- Can contain up to billions of vertices and edges
- Need simple, efficient, and scalable ways to analyze them

Graph Data is Everywhere!
Efficient Graph Processing

• Use parallelism

• Design efficient algorithms

  Breadth-first search
  Betweenness centrality
  Connected components
  ...

  Single-source shortest paths
  Eccentricity estimation
  (Personalized) PageRank
  ...

• Write/optimize code for each application
• Build a general framework
Ligra Graph Processing Framework

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

**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, Pegasus, Knowledge Discovery Toolbox, GraphChi, etc.

• 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 $r$
- Can process each frontier in parallel
- Race conditions, load balancing

Applications:
- Betweenness centrality
- Eccentricity estimation
- Maximum flow
- Web crawlers
- Network broadcasting
- Cycle detection
- ...
Steps for Graph Traversal

- Operate on a subset of vertices
- Map computation over subset of edges in parallel
- Return new subset of vertices
- Map computation over subset of vertices in parallel

We built the Ligra abstraction for these kinds of computations

Think with flat data-parallel operators

Abstraction enables optimizations (hybrid traversal and 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);
Actual BFS code in Ligra

```c
#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);
}
Sparse or Dense EdgeMap?

- 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]

Limited to BFS?
procedure \textsc{Edgemap}(G, frontier, Update, Cond):
    if (size(frontier) + sum of out-degrees > threshold) then:
        return \textsc{Edgemap\_Dense}(G, frontier, Update, Cond);
    else:
        return \textsc{Edgemap\_Sparse}(G, frontier, Update, Cond);

Loop through outgoing edges of frontier vertices in parallel

- More general than just BFS!
- Generalized to many other problems
  - For example, betweenness centrality, connected components, sparse PageRank, shortest paths, eccentricity estimation, graph clustering, k-core decomposition, set cover, etc.
- Users need not worry about this
Frontier-based approach enables hybrid traversal

Twitter graph (41M vertices, 1.5B edges)

<table>
<thead>
<tr>
<th></th>
<th>Dense</th>
<th>Sparse</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>1</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>Betweenness Centrality</td>
<td>30.7</td>
<td>10.5</td>
<td>5.5</td>
</tr>
<tr>
<td>Connected Components</td>
<td>20.7</td>
<td>3.5</td>
<td>2.5</td>
</tr>
<tr>
<td>Eccentricity Estimation</td>
<td>8</td>
<td>8.5</td>
<td>2.5</td>
</tr>
</tbody>
</table>

(Switching between sparse and dense using default threshold of |E|/20)
PageRank

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

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

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

procedure **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;
    return 1;

procedure **PageRank**(G, \alpha, \epsilon):
    frontier = \{0, \ldots, |V|-1\};
    while (error > \epsilon):
        frontier = \text{EDGEMAP}(G, \text{frontier}, \text{UPDATE}, \text{COND_true});
        frontier = \text{VERTEXMAP}(\text{frontier}, \text{COMPUTE});
        error = \text{sum of diff entries};
        swap(p_{\text{curr}}, p_{\text{next}})
    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 PowerGraph and McSherry WWW ‘05)
PageRank-Delta in Ligra

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

**procedure** \text{UPDATE}(s, d): \quad \text{//passed to EdgeMap}
\[
\text{atomic\_increment(\text{nghSum}[d], \text{Change}[s] / \text{degree}(s))};
\text{return 1;}
\]

**procedure** \text{COMPUTE}(i): \quad \text{//passed to VertexMap}
\[
\text{Change}[i] = \alpha \cdot \text{nghSum}[i]; \\
\text{PR}[i] = \text{PR}[i] + \text{Change}[i]; \\
\text{return (abs(Change}[i]) > \Delta); \quad \text{//check if absolute value of change is big enough}
\]
Performance of Ligra
Ligra BFS Performance

- Comparing against hybrid traversal BFS code by Beamer et al.
Ligra PageRank Performance

Twitter graph (41M vertices, 1.5B edges)

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

- Easy to implement “sparse” version of PageRank in Ligra
Largest publicly available graph

- Ligra’s performance is close to hand-written code
- Faster than best existing system
- Subsequent systems have used Ligra’s abstraction and hybrid traversal idea, e.g., Galois [SOSP ‘13], Polymer [PPoPP ’15], Gunrock [PPoPP ’16], Gemini [OSDI ’16], GraphGrind [ICS ‘17], Grazelle [PPoPP ‘18]