A Framework for Processing Large Graphs in Shared Memory

Julian Shun

Based on joint work with Guy Blelloch and Laxman Dhulipala
(Work done at Carnegie Mellon University)
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, 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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Betweenness centrality</td>
<td>Eccentricity estimation</td>
<td>Maximum flow</td>
</tr>
<tr>
<td>Web crawlers</td>
<td>Network broadcasting</td>
<td>Cycle detection</td>
</tr>
</tbody>
</table>

- Can process each frontier in parallel
- Race conditions, load balancing
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()){
        vertexSubset output = edgeMap(GA, Frontier, BFS_F(Parents));
        Frontier.del();
        Frontier = output; //set new frontier
    }
    Frontier.del();
    free(Parents);
}
```
Sparse or Dense Edge Map?

- 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 **EDGEMAP**\((G, \text{frontier}, \text{Update}, \text{Cond})\):

\[
\begin{align*}
\text{if } & (\text{size(frontier)} + \text{sum of out-degrees} > \text{threshold}) \text{ then:} \\
& \quad \text{return } \text{EDGEMAP\_DENSE}(G, \text{frontier}, \text{Update}, \text{Cond}); \\
\text{else:} \\
& \quad \text{return } \text{EDGEMAP\_SPARSE}(G, \text{frontier}, \text{Update}, \text{Cond});
\end{align*}
\]

Loop through outgoing edges of frontier vertices in parallel

Loop through incoming edges of “unexplored” vertices (in parallel), breaking early if possible

• 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>0.1</td>
<td>1.1</td>
<td>0.1</td>
</tr>
<tr>
<td>Betweenness Centrality</td>
<td>30.7</td>
<td>6.7</td>
<td>11.2</td>
</tr>
<tr>
<td>Connected Components</td>
<td>20.7</td>
<td>4.7</td>
<td>11.2</td>
</tr>
<tr>
<td>Eccentricity Estimation</td>
<td>8.7</td>
<td>11.2</td>
<td>11.2</td>
</tr>
</tbody>
</table>

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

\[
PR[v] = \frac{1 - \gamma}{|V|} + \gamma \sum_{u \in N^-(v)} \frac{PR[u]}{\text{deg}^+(u)}
\]
VertexMap

bool f(v){
    data[v] = data[v] + 1;
    return (data[v] == 1);
}

VertexSubset

VertexMap

VertexSubset
PageRank in Ligra

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

\begin{verbatim}
procedure UPDATE(s, d):
    atomic_increment(p_{\text{next}}[d], p_{\text{curr}}[s] / \text{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, \varepsilon):
    \text{frontier} = \{0, \ldots, |V|-1\};
    while (error > \varepsilon):
        \text{frontier} = \text{EDGEMAP}(G, \text{frontier}, \text{UPDATE}, \text{COND}_{\text{true}});
        \text{frontier} = \text{VERTEXMAP}(\text{frontier}, \text{COMPUTE});
        \text{error} = \text{sum of diff entries};
        \text{swap}(p_{\text{curr}}, p_{\text{next}})
    return p_{\text{curr}};
\end{verbatim}
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 **UPDATE** (s, d): \hspace{1em} \text{//passed to EdgeMap}
\[ \text{atomic increment} (\text{nghSum}[d], \text{Change}[s] / \text{degree}(s)); \]
return 1;

procedure **COMPUTE** (i): \hspace{1em} \text{//passed to VertexMap}
\[ \text{Change}[i] = \alpha \cdot \text{nghSum}[i]; \]
\[ \text{PR}[i] = \text{PR}[i] + \text{Change}[i]; \]
return (abs(\text{Change}[i]) > \Delta); \hspace{1em} \text{//check if absolute value of change is big enough}
Performance of Ligra
Comparing against hybrid traversal BFS code by Beamer et al.
• 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]
## 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>4</td>
<td>12</td>
<td>122</td>
<td>1 x 120 SSD</td>
<td>$0.834 per Hour</td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td>244</td>
<td>1 x 240 SSD</td>
<td>$1.668 per Hour</td>
</tr>
<tr>
<td>16</td>
<td>47</td>
<td>488</td>
<td>1 x 480 SSD</td>
<td>$3.336 per Hour</td>
</tr>
<tr>
<td>32</td>
<td>91</td>
<td>976</td>
<td>1 x 960</td>
<td>$6.672 per Hour</td>
</tr>
<tr>
<td>64</td>
<td>179</td>
<td>1952</td>
<td>1 x 1920 SSD</td>
<td>$13.344 per Hour</td>
</tr>
<tr>
<td>128</td>
<td>340</td>
<td>3904</td>
<td>2 x 1920 SSD</td>
<td>$26.688 per Hour</td>
</tr>
</tbody>
</table>

- Most can fit on commodity shared memory machine

**Example**

Dell PowerEdge R930:
Up to 96 cores and 6 TB of RAM
What if you don’t have or can’t afford that much memory?

**Graph Compression**
Ligra+: Adding Graph Compression to Ligra
Ligra+: Adding Graph Compression to Ligra

- Same interface as Ligra
- All changes hidden from the user!
Graph reordering to improve locality
- Goal: give neighbors IDs close to vertex ID
- BFS, DFS, METIS, our own separator-based algorithm
Variable-length codes

- k-bit codes
  - Encode value in chunks of k bits
  - Use k-1 bits for data, and 1 bit as the “continue” bit
- Example: encode “401” using 8-bit (byte) code
- In binary: 

  1 1 0 0 1 0 0 0 1

  7 bits for data

  0 0 0 0 0 0 0 1 1

  “continue” bit
Another idea: get rid of “continue” bits

<table>
<thead>
<tr>
<th>x₁</th>
<th>x₂</th>
<th>x₃</th>
<th>x₄</th>
<th>x₅</th>
<th>x₆</th>
<th>x₇</th>
<th>x₈</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Number of bytes required to encode each integer

Use run-length encoding

Header

Integers in group encoded in byte chunks

Number of bytes per integer

Size of group (max 64)

• Increases space, but makes decoding cheaper (no branch misprediction from checking “continue” bit)
Ligra+: Adding Graph Compression to Ligra

- Same interface as Ligra
- All changes hidden from the user!
Modifying EdgeMap

- Processes outgoing edges of a subset of vertices

<table>
<thead>
<tr>
<th>VertexSubset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 7 16 25 44</td>
</tr>
</tbody>
</table>

| 2 | 5 | 2 | 7 | 9 | 2 | 1 | 3 | 3 |

| -4 | 6 | 3 | 1 | 3 | 5 | 6 | 2 |

| 5 | 10 | 2 |

| 30 | 5 |

| -16 | 2 | 19 | 1 | 4 | 2 | 5 | 3 |

All vertices processed in parallel

What about high-degree vertices?
Handling high-degree vertices

High-degree vertex

Chunks of size $T$

Encode first entry relative to source vertex

All chunks can be decoded in parallel!

- We chose $T=1000$
- Similar performance and space usage for a wide range of $T$
Ligra+ Space Savings

- Space savings of about 1.3—3x
- Could use more sophisticated schemes to further reduce space, but more expensive to decode
- Cost of decoding on-the-fly?
Ligra+ Performance

- Cost of decoding on-the-fly?
- Memory subsystem is a scalability bottleneck in parallel as these graph algorithms are memory-bound
- Ligra+ decoding gets better parallel speed up
Ligra Summary

**VertexSubset**

**VertexMap**

**EdgeMap**

**Optimizations: Hybrid traversal and graph compression**

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

**Simplicity, Performance, Scalability**