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 \( r \)

<table>
<thead>
<tr>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Betweenness centrality</td>
</tr>
<tr>
<td>Eccentricity estimation</td>
</tr>
<tr>
<td>Maximum flow</td>
</tr>
<tr>
<td>Web crawlers</td>
</tr>
<tr>
<td>Network broadcasting</td>
</tr>
<tr>
<td>Cycle detection</td>
</tr>
<tr>
<td>...</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);
#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]
procedure **EDGEMAP**\( (G, \text{frontier}, \text{Update}, \text{Cond}) \):

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

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)

40-core running time (seconds)

- BFS
- Betweenness Centrality
- Connected Components
- Eccentricity Estimation

Dense
Sparse
Hybrid

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

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

VertexSubset

```
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\((p_{\text{next}}[d], \frac{p_{\text{curr}}[s]}{\text{degree}(s)})\);
    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, \(\alpha\), \(\epsilon\)):
    \text{frontier} = \{0, \ldots, |V| - 1\};
    while (\text{error} > \epsilon):
        \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}})
    return \text{p\_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

| PR[i] = \{1/|V|, \ldots, 1/|V|\}; |
| nghSum = \{0, \ldots, 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] = \alpha \cdot nghSum[i];
PR[i] = PR[i] + Change[i];

return (abs(Change[i]) > \Delta); //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
Connected Components Performance

Twitter graph (41M vertices, 1.5B edges)

- 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>x1e.xlarge</td>
<td>4</td>
<td>122</td>
<td>1 x 120 SSD</td>
<td>$0.834 per Hour</td>
</tr>
<tr>
<td>x1e.2xlarge</td>
<td>8</td>
<td>244</td>
<td>1 x 240 SSD</td>
<td>$1.668 per Hour</td>
</tr>
<tr>
<td>x1e.4xlarge</td>
<td>16</td>
<td>488</td>
<td>1 x 480 SSD</td>
<td>$3.336 per Hour</td>
</tr>
<tr>
<td>x1e.8xlarge</td>
<td>32</td>
<td>976</td>
<td>1 x 960</td>
<td>$6.672 per Hour</td>
</tr>
<tr>
<td>x1e.16xlarge</td>
<td>64</td>
<td>1952</td>
<td>1 x 1920 SSD</td>
<td>$13.344 per Hour</td>
</tr>
<tr>
<td>x1e.32xlarge</td>
<td>128</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!

**Interface**

- Graph
- VertexSubset
- EdgeMap
- VertexMap

- Use compressed representation
- Same as before
- Decode edges on-the-fly
- Same as before
### Graph representation

#### Vertex IDs
- 0
- 1
- 2
- 3

#### Offsets
- 0
- 4
- 5
- 11
- ...

#### Edges
- 2
- 7
- 9
- 16
- 0
- 1
- 6
- 9
- 12
- ...

#### Compressed Edges
- 2
- 5
- 2
- 7
- -1
- -1
- 5
- 3
- 3
- ...

---

**Sort edges and encode differences**

- $2 - 0 = 2$
- $7 - 2 = 5$
- $1 - 2 = -1$
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

“continue” bit
**Encoding optimization**

- Another idea: get rid of “continue” bits

\[
\begin{array}{cccccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\
  1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
\end{array}
\]

Number of bytes required to encode each integer

Use run-length encoding

```
0 1 0 1 1 0 0 1
```

Header

Integers in group encoded in byte chunks

- 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</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>2</th>
<th>5</th>
<th>2</th>
<th>7</th>
<th>9</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td>2</td>
<td>19</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
```

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?
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

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
Thank you!


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