Analysis of Subgraph-centric Distributed Shortest Path Algorithm

Ravikant Dindokar
Neel Choudhury  Yogesh Simmhan

ParLearning Workshop, 2015
Introduction

- Massive graphs with m/billions of nodes/edges
  - Shared-memory platforms on HPC machines
  - Distributed memory platforms on commodity hardware

- Path-based graph algorithms are key for ML
  - Link prediction, Spatial mining, Path ranking

- **SSSP** captures a common class of problems
  - But it can be costly on a single machine!

- We focus on SSSP on distributed platforms
Vertex-centric Model

- Logic written for a single vertex
- Execution as series of synchronized supersteps
- Vertices partitioned across multiple hosts
- Message passing between vertices. Messages delivered at superstep boundaries.
- Parallelism at vertex level
- E.g. Google Pregel, Apache Giraph

→ But, large communication cost, more time to converge

Subgraph-centric Model

- **Subgraph**: Weakly Connected Component (WCC) within a partition
- Logic written for a subgraph
  - Message passing between subgraphs
  - Parallelism at subgraph level
- Less communication cost, ~faster convergence
- *E.g.* GoFFish, Blogel, Giraph++

But, behavior of such distributed algos less understood

Need for algorithmic analysis

- Bounds on compute, communication and I/O complexity
- Predict behaviour of distributed graph algorithms before implementation
- Develop optimizations to graph runtime framework

Contributions

1. **Algorithmic analysis of subgraph-centric SSSP**
   - Communication & Compute Complexity of single superstep
   - Compute Complexity across multiple supersteps

2. **Co-relate empirical performance against expected analytic behaviour**
   - Benchmark of SSSP for three different real world graphs
Pseudocode: SSSP (S0)

Compute (<Messages> M_arr){
    if Superstep == 0
        dist[v] = ∞ ∀ v
        if source is present
            Rootset <- {source}
            dist[source] = 0
    else
        for each message in M_arr
            if dist[m.vertex] < m.value
                dist[m.vertex] = m.value
                Rootset <- Updated vertices

    Run Dijkstra’s on RootSet
    Send Messages to Remote Vertices
    VoteToHalt()
}

Note: Edges are Unweighted

Pseudocode: SSSP (S1)

Compute (<Messages> M_arr) {
    if Superstep == 0
        dist[v] = ∞ ∀ v
        if source is present
            Rootset <- {source}
            dist[source] = 0
    else
        for each message in M_arr
            if dist[m.vertex] < m.value
                dist[m.vertex] = m.value
                Rootset <- Updated vertices

    Run Dijkstra’s on RootSet
    Send Messages to Remote Vertices
    VoteToHalt()
}

Note: Edges are Unweighted
Pseudocode: SSSP (S2)

Compute (<Messages> M_arr){
  if Superstep == 0
    dist[v] = ∞ ∀ v
  if source is present
    Rootset <- {source}
    dist[source] = 0
  else
    for each message in M_arr
      if dist[m.vertex] < m.value
        dist[m.vertex] = m.value
        Rootset <- Updated vertices

  Run Dijkstra’s on RootSet
  Send Messages to Remote Vertices
  VoteToHalt()
}

Note: Edges are Unweighted
Algorithmic Analysis
Algorithmic Notation

- **Subgraph**: $SG = \langle V, E, R \rangle$
  - $|V| = v$ vertices
  - $|E| = e$ edges
  - $|R| = r$ remote edges

- For a superstep $s$
  - Compute complexity: $O_p$
  - Communication complexity: $O_c$
  - Total Complexity for subgraph $SG_k$: $T_k^s$
Algorithmic Analysis

From the perspective of a single subgraph, there are two phases:

1. Dijkstra's is called on a subgraph for first time
   a) Dijkstra's on SG with source vertex
   b) Dijkstra's on SG without source vertex

2. Dijkstra's is called on a subgraph for revisit update
1(a). Superstep 0 on source SG

Happens in **Superstep 0**

- Initial Scan for source vertex takes 
  \[ O_p (v_{source}) \]
- RootSet contains single entry, *source vertex*. So complexity of *Dijkstra’s* is
  \[ O_p (e_{source} \log v_{source}) \]
- Messages sent is bounded by remote edges. So communication complexity for source subgraph is
  \[ O_c (r_{source}) \]
- Total complexity for source subgraph is
  \[ T_{source}^0 = O_p (v_{source} + e_{source} \log v_{source}) + O_c (r_{source}) \]
1(a) Superstep 0 on non-source SG

• For subgraphs without source vertex, linear scan for source vertex takes

\[ T^0_k = O_p(v_k) \quad \forall \ SG_k \ | \ source \ v \not\in SG_k \]

• Total Time Complexity for superstep 0 is given by:

\[ T^0 = \max ( T^0_{source}, T^0_k ) \]
1(b) Dijkstra’s on non-source SG

- Max RootSet: \( \min(v_k, r_k) \)
- Dijkstra’s complexity \( O_p(e_k \log v_k) \)
- Insertion in priority queue \( O_p(r_k \log v_k) \)
- Communication complexity \( O_c(r_k) \)
- Total complexity

\[ T_{s_k} = O_p((r_k + e_k) \log v_k) + O_c(r_k) \]
2. Analysis across supersteps: **Meta-graph sketch**

- Each **subgraph** is a **meta-vertex**
- Directed **meta-edges** depending on presence of **remote edges**
- Motivation: Reduces problem of analyzing millions of vertices to few meta-vertices.
- Estimate **number of supersteps** from structure of meta-graph.
- Behaviour across supersteps
- Useful to express bounds on superstep

**Metagraph for CA Road Network**

\( v = 1.96M, e = 2.76M \)
2. Analysis across Supersteps ..

- For convergence, each SG must be visited at least once
- Perform BFS from SG with source
- BFS depth corresponds to Superstep execution

“The height of the BFS tree in the meta-graph rooted at the source subgraph (meta-vertex) gives the lower bound on the number of supersteps required for SSSP to converge”.

“Depth of the first BFS visit on a meta-vertex (subgraph) gives the superstep at which Dijkstra’s will be called on that subgraph”
2. Analysis across supersteps...

- Dijkstra's performed when boundary vertices gets an update.
- Update received, only when SG visited in BFS traversal.

“Depth of BFS re-visits on a meta-vertex (subgraph) gives the superstep at which Dijkstra’s may be repeated on that subgraph”
Empirical Analysis
Experimental setup & Dataset

- Commodity cluster of 6 machines
- Each machine has AMD 3380 (8 cores, 2.6 GHz) CPU, 32GB RAM, Gigabit Ethernet
- GoFFish v2.6 with OpenJDK v1.7
- Graphs have directed edges.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>Dia.</th>
</tr>
</thead>
<tbody>
<tr>
<td>California Road Network (CARN)</td>
<td>1,965,206</td>
<td>2,766,607</td>
<td>849</td>
</tr>
<tr>
<td>Google Web Graph (WEBG)</td>
<td>875,713</td>
<td>5,105,039</td>
<td>21</td>
</tr>
<tr>
<td>Wikipedia Talk Network (WIKI)</td>
<td>2,394,385</td>
<td>5,021,410</td>
<td>9</td>
</tr>
</tbody>
</table>
1. Initial Dijkstra's: Expected vs. Actual

- **Dijkstra’s called exactly at superstep corresponding to traversal depth.**
- **Expected and observed time complexity matches closely.**
- **Outliers:** Subgraph with source and subgraph with large number of incoming messages

*Expected time is normalized by multiplying it by a constant $\alpha$  
*Plot showing only non tiny subgraphs ( $|V| > 100$ )
2. Revisits normalized to Initial Dijkstra’s

- Uniform decay in number of vertices that get update across subsequent visits.
- Dijkstra’s takes place only in subset of the depths of BFS.
- Outlier: Due to directed nature of graph & subgraph defined as WCC

*Plot showing only non tiny subgraphs ( |V| > 100 )
Conclusions

- We propose an analytical model for behaviour of subgraph-centric SSSP
  - Using both computational & communication cost within a superstep, progress across the superstep
- We offer a novel meta-graph sketch based approach to model algorithmic behaviour
- We have shown correlation between expected & actual time complexity for initial Dijkstra's
Future Work

- Model the decay in computational complexity in successive visits
- Offer tighter upper bounds on total number of supersteps to converge
- One SG dominates Scope to improve utilization ... *paper under review*
Thank You!

Contact Us: {ravikant7, neel}@ssl.serc.iisc.in
Download GoFFish: dream-lab.serc.iisc.in/software

©DREAM:Lab, 2015
This work is licensed under a Creative Commons Attribution 4.0 International License