THE MORE THE MERRIER

EFFICIENT MULTI-SOURCE GRAPH TRAVERSAL

Then et Al.

Jeremy Bogle

BACKGROUND

- Graph analytics
- Multi core machines
- Graph traversal on same graph from different sources
  - Calculating graph centralities
  - Enumerating neighborhoods for all vertices
  - All-pairs shortest distance problem

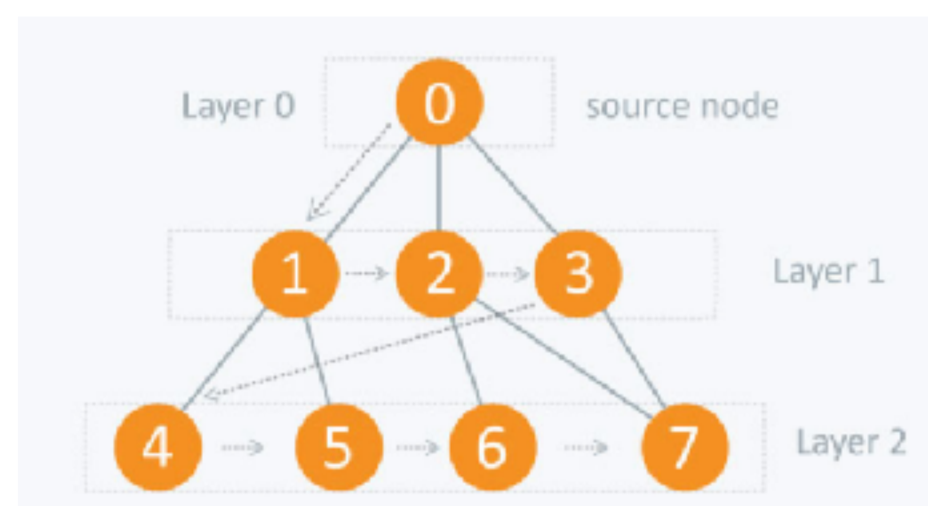
BFS

- Textbook BFS
- Building block for other graph traversals
- Max levels - diameter(G)
- Random memory accesses every time it checks if a neighbor has been visited

Listing 1: Textbook BFS algorithm.

```
Input: G, s
seen ← {s}
visit ← {s}
visitNext ← ∅

while visit ≠ ∅
    for each v ∈ visit
        for each n ∈ neighbors_v
            if n ∉ seen
                seen ← seen ∪ {n}
                visitNext ← visitNext ∪ {n}
                do BFS computation on n
    visit ← visitNext
    visitNext ← ∅
```



OPTIMIZING TEXTBOOK BFS

- Level by level parallelization

Work: $T_1(n) = \Theta(m+n)$

Span: $T_\infty(n) = \Theta(d)$

- Beamer et. All
  - Bottom up approach - Explores based on unvisited nodes
  - Hybrid approach - Uses bottom up for large frontiers, top up otherwise

Variable	Description
alpha	Tuning parameter
beta	Tuning parameter
m_f	# Edges in frontier
m_u	# Unexplored vertices
n_f	# Vertices in frontier
n	# Vertices

$$m_f > \frac{m_u}{\alpha} = C_{TB}$$

$$n_f < \frac{n}{\beta} = C_{BT}$$

MOTIVATION

- Large graphs often must be searched from various starting nodes
- Lots of overlap when executing BFS from multiple nodes
- Small world graphs have even more overlap - large fanout, each level grows rapidly

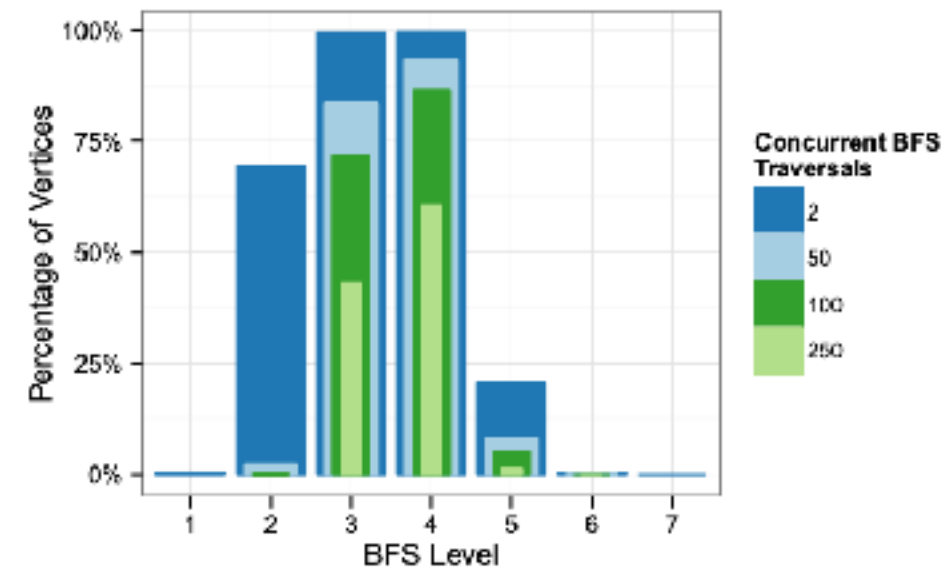
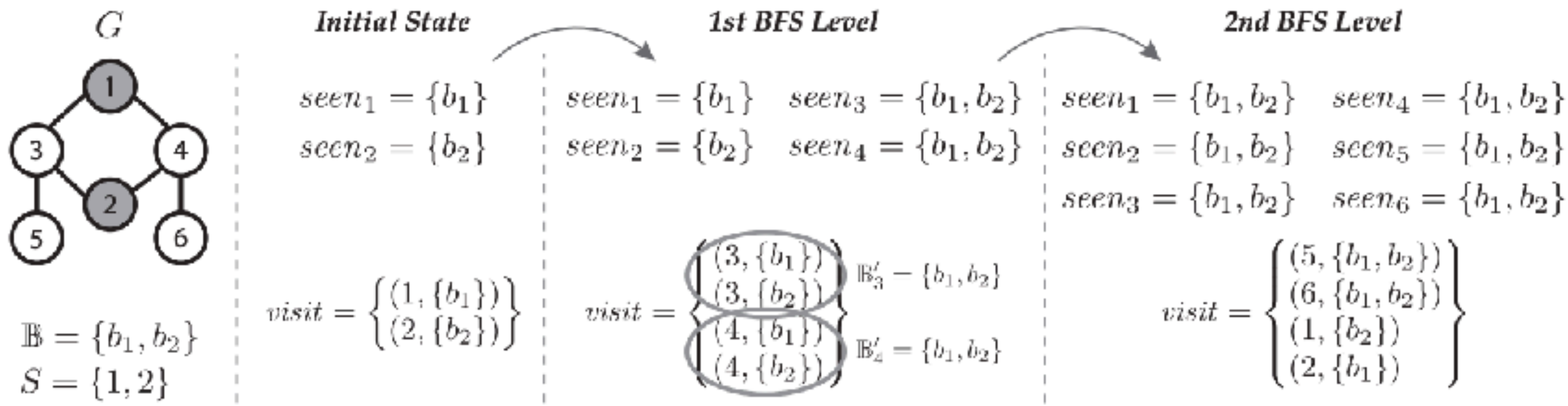


Figure 1: Percentage of vertex explorations that can be shared per level across 512 concurrent BFSs.

SMALL WORLD GRAPHS

The distance between any two vertices is very small compared to the size of the graph, and the number of vertices discovered in each iteration of the BFS algorithm grows rapidly.

MS-BFS: EXAMPLE



- Increase the dimensionality of textbook BFS to allow for multiple BFS at once
- Shared exploration of nodes
- Finish all BFSs executions in parallel

Listing 2: The MS-BFS algorithm.

```
Input: G, 𝔹, S
seen_{s_i} ← {b_i} for all b_i ∈ 𝔹
visit ← ⋃_{b_i∈𝔹} {(s_i, {b_i})}
visitNext ← ∅

while visit ≠ ∅
    for each v in visit
        𝔹′_v ← ∅
        for each (v′, 𝔹′) ∈ visit where v′ = v
            𝔹′_v ← 𝔹′_v ∪ 𝔹′
        for each n ∈ neighbors_v
            𝔻 ← 𝔹′_v \ seen_n
            if 𝔻 ≠ ∅
                visitNext ← visitNext ∪ {(n, 𝔻)}
                seen_n ← seen_n ∪ 𝔻
                do BFS computation on n
    visit ← visitNext
    visitNext ← ∅
```

OPTIMIZATIONS FOR MS-BFS

- Bit operations
- Aggregated neighbor processing
- Direction optimized
- Neighbor prefetching
- Sharing heuristic

OPTIMIZATIONS FOR MS-BFS

Listing 2: The MS-BFS algorithm.

```
Input: G, 𝔹, S
seen_{s_i} ← {b_i} for all b_i ∈ 𝔹
visit ← ⋃_{b_i∈𝔹} {(s_i, {b_i})}
visitNext ← ∅

while visit ≠ ∅
    for each v in visit
        𝔹'_v ← ∅
        for each (v', 𝔹') ∈ visit where v' = v
            𝔹'_v ← 𝔹'_v ∪ 𝔹'
        for each n ∈ neighbors_v
            𝔻 ← 𝔹'_v \ seen_n
            if 𝔻 ≠ ∅
                visitNext ← visitNext ∪ {(n, 𝔻)}
                seen_n ← seen_n ∪ 𝔻
                do BFS computation on n
    visit ← visitNext
    visitNext ← ∅
```

Listing 3: MS-BFS using bit operations.

```
Input: G, 𝔹, S
for each b_i ∈ 𝔹
    seen[ s_i ] ← 1 << b_i
    visit[ s_i ] ← 1 << b_i
reset visitNext

while visit ≠ ∅
    for i = 1, . . . , N
        if visit[v_i] = 𝔹_∅, skip
        for each n ∈ neighbors[v_i]
            𝔻 ← visit[v_i] & ∼seen[n]
            if 𝔻 ≠ 𝔹_∅
                visitNext[n] ← visitNext[n] | 𝔻
                seen[n] ← seen[n] | 𝔻
                do BFS computation on n
    visit ← visitNext
    reset visitNext
```

Listing 4: MS-BFS algorithm using ANP.

```
Input: G, 𝔹, S
for each b_i ∈ 𝔹
    seen[ s_i ] ← 1 << b_i
    visit[ s_i ] ← 1 << b_i
reset visitNext

while visit ≠ ∅
    for i = 1, . . . , N
        if visit[v_i] = 𝔹_∅, skip
        for each n ∈ neighbors[v_i]
            visitNext[n] ← visitNext[n] | visit[v_i]

    for i = 1, . . . , N
        if visitNext[v_i] = 𝔹_∅, skip
        visitNext[v_i] ← visitNext[v_i] & ∼seen[v_i]
        seen[v_i] ← seen[v_i] | visitNext[v_i]
        if visitNext[v_i] ≠ 𝔹_∅
            do BFS computation on v_i
    visit ← visitNext
    reset visitNext
```

EVALUATION AND RESULTS

- Running BFS from all nodes as number of vertices increases
- Traversed edges per second
- Improvement benefits from various optimizations

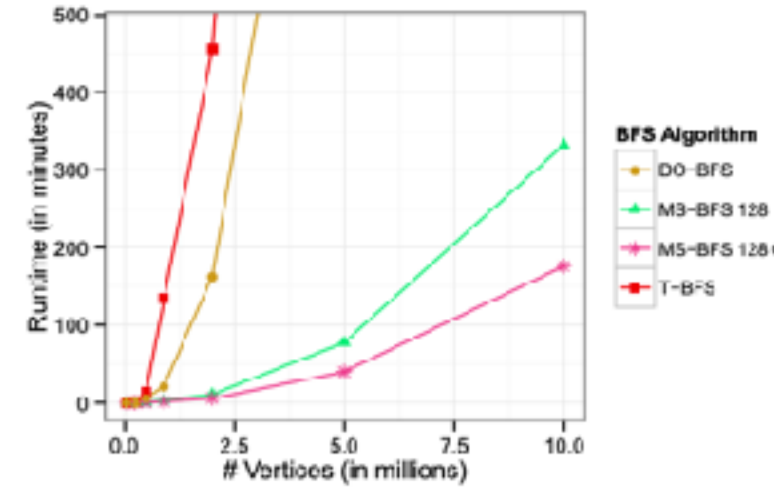


Figure 4: Data size scalability results.

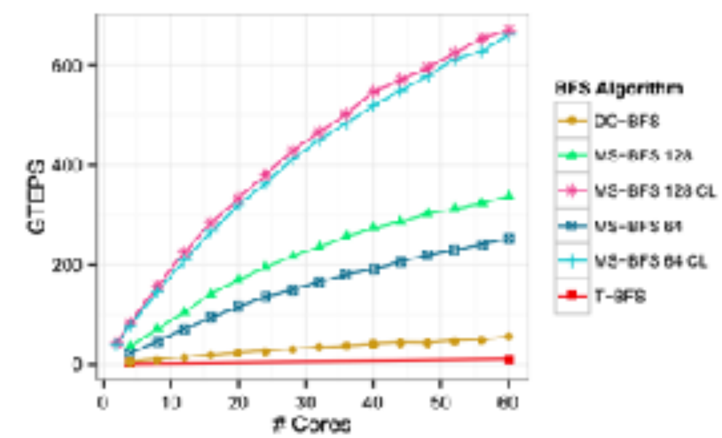


Figure 5: Multi-core scalability results.

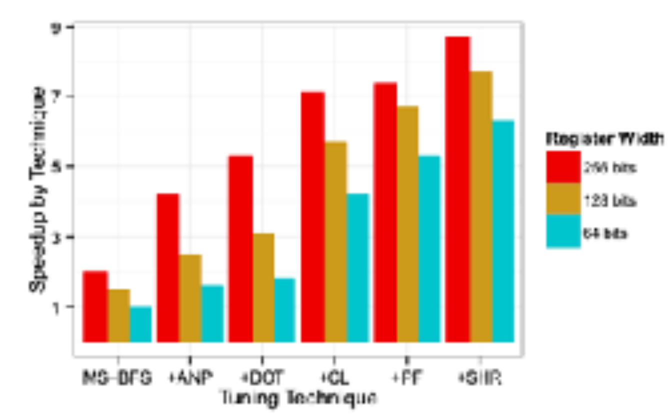


Figure 7: Speedup achieved by cumulatively applying different tuning techniques to MS-BFS.

STRENGTHS

- Comparison with existing approaches
- Leverage existing optimizations
- Large scale evaluations

WEAKNESSES

- Must be overlapping during the same iterations
  - No “memory” of previously searched nodes
- Evaluation on non “small-world” graphs
- Perform optimizations independently
- Evaluation in a distributed system
- MS-BFS with parallelization at each level

DISCUSSION

- What did you guys think were the strengths and weaknesses?

- On what types of graphs is MS-BFS NOT useful
  - How could it be improved to be useful on these graphs?
  - How does MS-BFS perform compared to textbook BFS in these scenarios