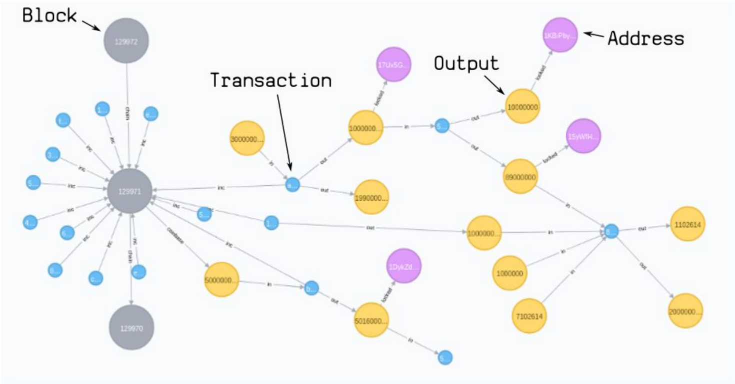
The More the Merrier: Efficient Multi-Source Graph Traversal

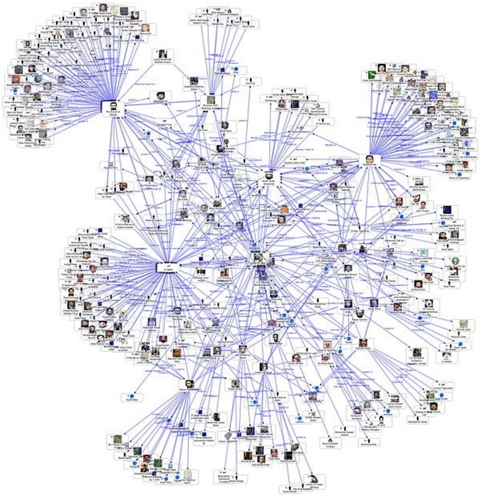
Then et al.

Presenter: Sai Sameer Pusapaty

Background

- Lot of information held via graphs
  - Social networks
  - Road Networks
  - Comp bio
- Graph analytics to comprehend relationships
  - Often requires multiple graph traversals (BFS)
  - Graph centrality
  - All pairs shortest paths
- Usually do a BFS from each vertex on the graph





Small World Assumption

- “Distance between any two vertices is very small compared to size of the graph”
- “Number of vertices discovered in each iteration grows rapidly”
- I.e. only a few iterations of BFS to traverse the entire graph
- This assumption is not unreasonable:
    - 92% of Facebook users are connected by only 5 steps
    - (“Four Degrees of Separation”, L. Backstrom et al. 2012)
    - 4.74 (2012), 3.57 (2016)
- Wikis, WWW, gene networks, electrical power grids

BFS (Single Traversal)

- Single source
- Keep track of unexplored neighbors
- Maintain levels of exploration
  - Max num of levels = diameter
  - Small-world assumption
- Optimizations:
  - Parallel BFS
  - Bottom up approach (direction optimized)

Work = Θ(n+m)

Depth = O(Δ log m)

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 ← ∅
```

Motivation for MS-BFS

- Goal is to optimize execution of multiple independent BFSs
  - Common graph analytics would benefit from this
- Related work just on improving single execution of BFS
- Compared to old method of repetitive BFSs traversals…
  - We want better memory locality since the same vertices are discovered and explored -> tldr; fewer cache misses
  - We want better resource management as the # of BFSs increase and # of cores increase
  - We want to avoid synchronization due to its overhead

Overview of MS-BFS

- Very similar to a normal BFS
- Each vertex maintains *seen,* holds BFSs that have already visited it
- *Visit* contains a tuple of the vertex and BFS that is currently visiting it -- unioned together
- Neighbors not seen before are explored
- **Main Idea:** BFS that share common sub-traversal travel together

Listing 2: The MS-BFS algorithm.

Input: $G, \mathbb{B}, S$
$seen_{s_i} \leftarrow \{b_i\}$ for all $b_i \in \mathbb{B}$
$visit \leftarrow \bigcup_{b_i \in \mathbb{B}} \{(s_i, \{b_i\})\}$
$visitNext \leftarrow \varnothing$

while $visit \neq \varnothing$
    for each v **in** $visit$
        $\mathbb{B}'_v \leftarrow \varnothing$
        for each $(v', \mathbb{B}') \in visit$ **where** $v' = v$
            $\mathbb{B}'_v \leftarrow \mathbb{B}'_v \cup \mathbb{B}'$
        for each $n \in neighbors_v$
            $\mathbb{D} \leftarrow \mathbb{B}'_v \setminus seen_n$
            if $\mathbb{D} \neq \varnothing$
                $visitNext \leftarrow visitNext \cup \{(n, \mathbb{D})\}$
                $seen_n \leftarrow seen_n \cup \mathbb{D}$
                do BFS computation on n
    $visit \leftarrow visitNext$
    $visitNext \leftarrow \varnothing$

An Example

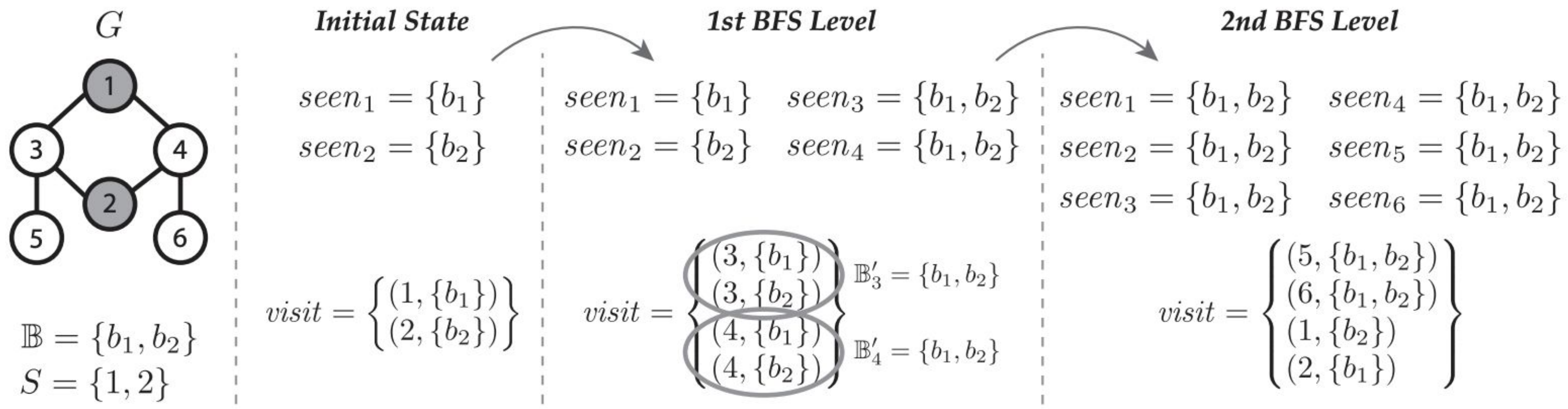


Figure 2: An example of the MS-BFS algorithm, where vertices 3 and 4 are explored once for two BFSs.

Improving with Bit Operations

- In practice set unions and differences are expensive
- Use bit operations instead
- *seen* is a bit-field for each vertex v, such that if the ith bit is 1, that means the ith BFS has already seen v
- Similarly *visit* and *visitNext* are set up s.t. If the ith bit is 1, then v still needs to be explored by the ith BFS
- Set operations become binary operations
- Store these three bitfields in arrays for constant time access -> visit_v = visit[v]

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
```

An Example

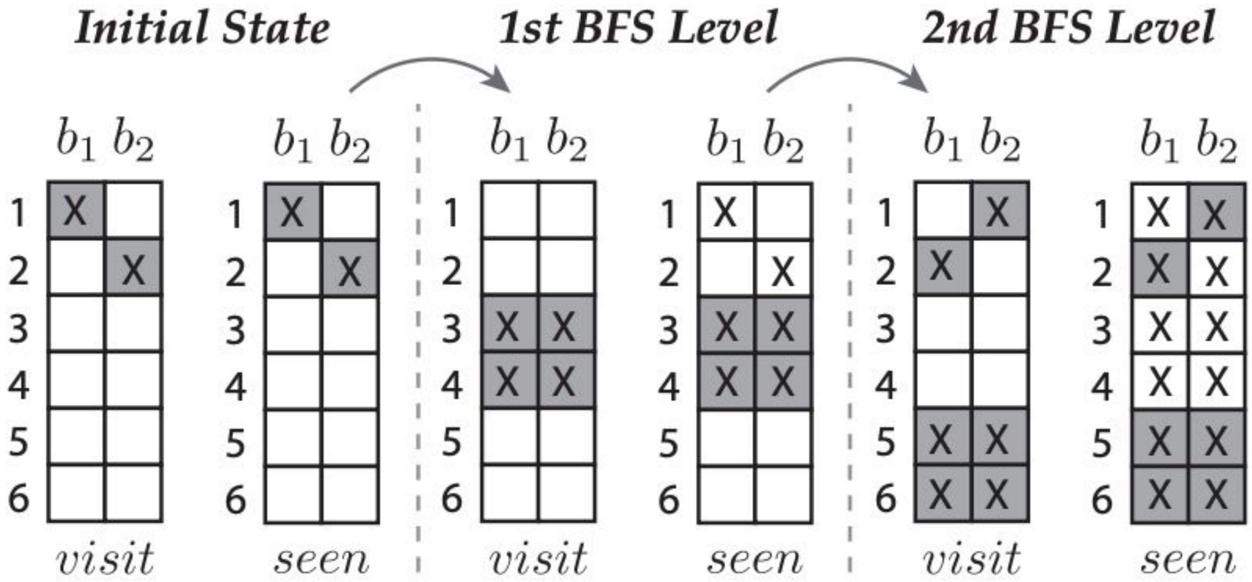


Figure 3: An example showing the steps of MS-BFS when using bit operations. Each row represents the bit field for a vertex, and each column corresponds to one BFS. The symbol X indicates that the value of the bit is 1.

Aggregated Neighbor Processing

- Still some bad memory access
- *seen* has a lot of random accesses which lead to cache misses
- ANP first collects all the vertices needed to be explored in the next level (lines 8-11)
- *seen* is updated in batch
- Improvement include
  - Fewer calls to *seen* (once per discovered vertex)
  - Thus, fewer iterations of BFS computation
  - Memory access is sequential
- Direction Optimized Traversal, prefetching, max sharing heuristic

Listing 4: MS-BFS algorithm using ANP.

```
1  Input: G, 𝔹, S
2  for each b_i ∈ 𝔹
3      seen[ s_i ] ← 1 << b_i
4      visit[ s_i ] ← 1 << b_i
5  reset visitNext
6
7  while visit ≠ ∅
8      for i = 1, . . . , N
9          if visit[v_i] = 𝔹_∅, skip
10         for each n ∈ neighbors[v_i]
11             visitNext[n] ← visitNext[n] | visit[v_i]
12
13     for i = 1, . . . , N
14         if visitNext[v_i] = 𝔹_∅, skip
15         visitNext[v_i] ← visitNext[v_i] & ∼seen[v_i]
16         seen[v_i] ← seen[v_i] | visitNext[v_i]
17         if visitNext[v_i] ≠ 𝔹_∅
18             do BFS computation on v_i
19     visit ← visitNext
20     reset visitNext
```

Experimental Results

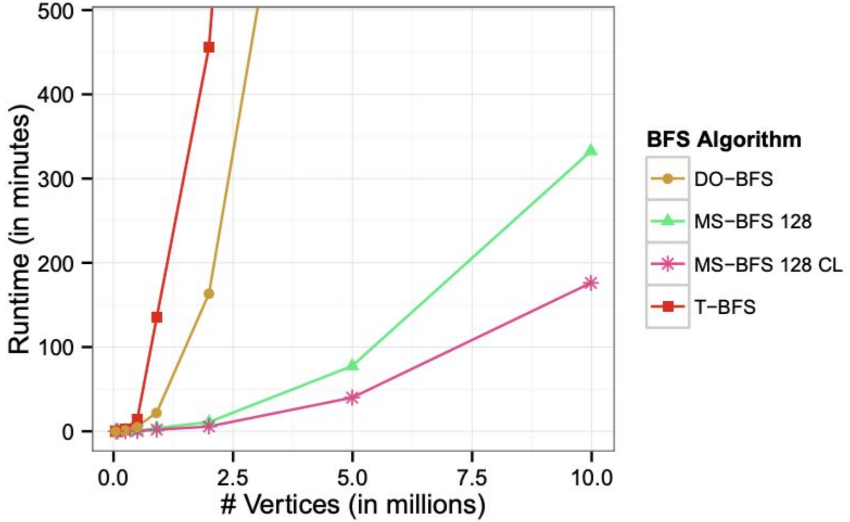


Figure 4: Data size scalability results.

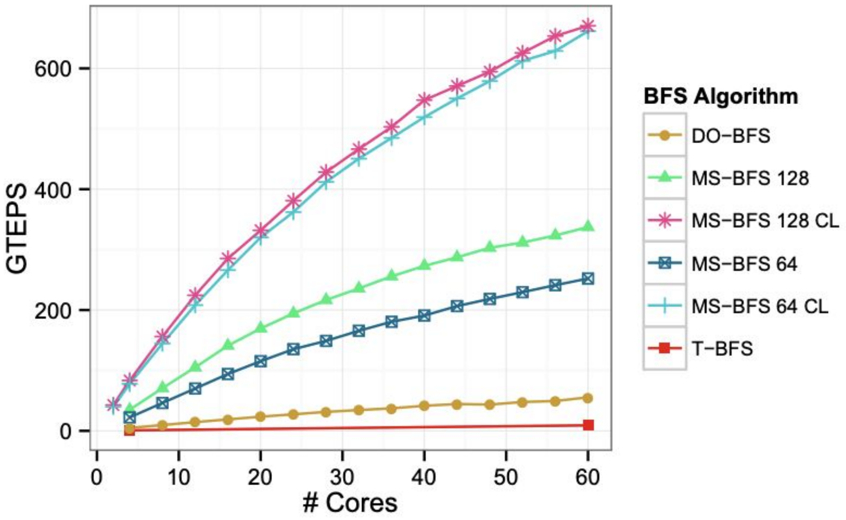


Figure 5: Multi-core scalability results.

More Results

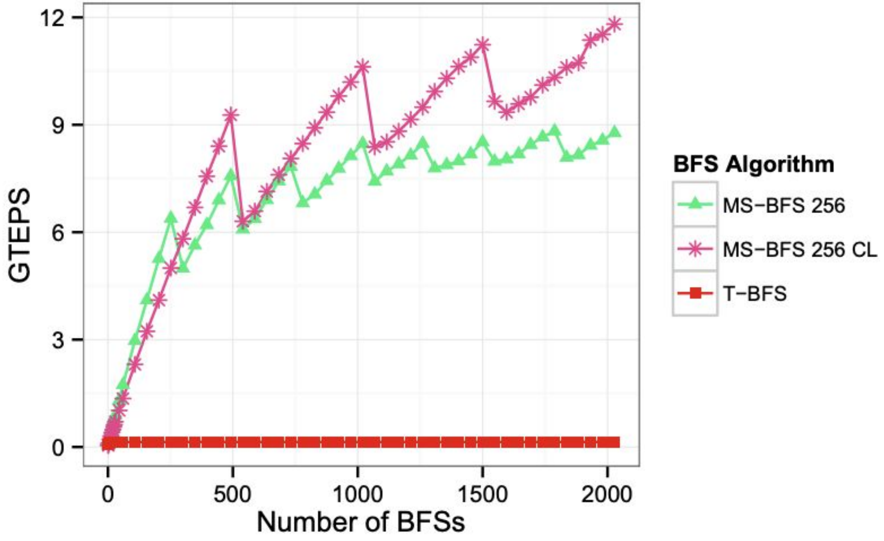


Figure 6: BFS count scalability results.

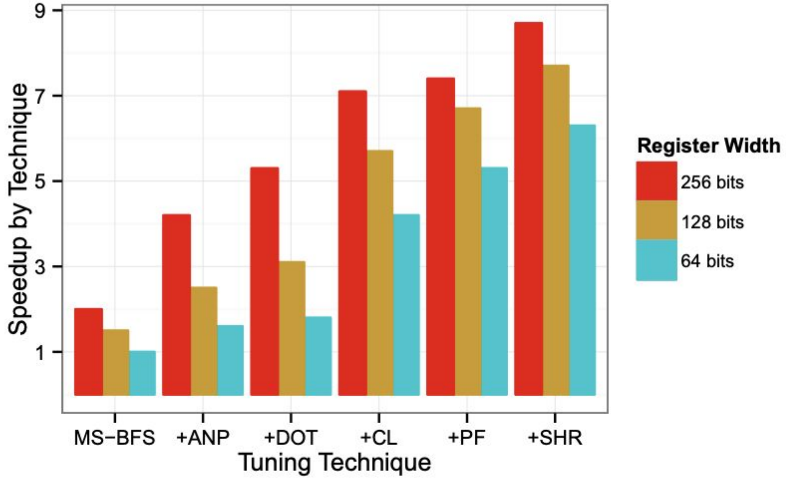


Figure 7: Speedup achieved by cumulatively applying different tuning techniques to MS-BFS.

Relative Speedups

Table 4: Runtime and speedup of MS-BFS compared to T-BFS and DO-BFS.

Graph	T-BFS	DO-BFS	MS-BFS	Speedup
LDBC 1M	2:15h	0:22h	0:02h	73.8x, 12.1x
LDBC 10M	*259:42h	*84:13h	2:56h	88.5x, 28.7x
Wikipedia	*32:48h	*12:50h	0:26h	75.4x, 29.5x
Twitter (1M)	*156:06h	*36:23h	2:52h	54.6x, 12.7x

*Execution aborted after 8 hours; runtime estimated.

Strengths

- Outperforms T-BFS and DO-BFS on a single core (main goal)
- Scales well with an increasing number of cores
- Generally scales well even as the number of BFSs increase
- Paper provides further improvements which experimentally did well
- Works well on real life graphs
- Can be parallelized naturally (no immediate barriers -- kind of)

Weaknesses

- Some limitations if the number of BFSs increase past register width
  - Paper proposes some alternatives
  - Parallelizing, Using multiple registers, running many instances
  - Performance/Memory tradeoff
- Graph set is limited to those that follow the small world assumption
- Memory overhead with large graphs to store BFS states at each vertex
- Provides the benefit to vertices that multiple BFSs access on the same level
  - No memory if the vertices have already been accessed before
  - Potential for further decrease in computation

Future Work

- Look at parallelizing at the frontier level
- Adapting MS-BFS for distributed environments and GPUs
- Apply it to other graph analytics algorithms
- Testing MS-BFS on various graph types
- New heuristics to maximize sharing

Discussion

- What are some other strengths and weaknesses you see in MS-BFS?
- Can MS-BFS generalize to other graphs besides small-world graphs?
- Do you have your own thoughts for improvement of future extensions?