The More the Merrier: Efficient Multi-Source Graph Traversal

Authors: : Manuel Then, Moritz Kaufmann, et al

6.827 Paper Presentation

Presenter: Edmund WIlliams

February 2022

Table of Contents

Problem: How To Make BFS Faster

Previous Ideas

- Parallel BFS implementations
- Bottom-Up approach (Beamer et al.)

New Idea

- Most applications require more than a single BFS traversal
- Instead of making one BFS faster, can we make batches of BFS traversals run faster?

Problem: How To Make BFS Faster

Previous Ideas

- Parallel BFS implementations
- Bottom-Up approach (Beamer et al.)

New Idea

- Most applications require more than a single BFS traversal
- Instead of making one BFS faster, can we make batches of BFS traversals run faster?

Problem: How To Make BFS Faster

Previous Ideas

- Parallel BFS implementations
- Bottom-Up approach (Beamer et al.)

New Idea

- Most applications require more than a single BFS traversal
- Instead of making one BFS faster, can we make batches of BFS traversals run faster?

Problem: How To Make BFS Faster

Previous Ideas

- Parallel BFS implementations
- Bottom-Up approach (Beamer et al.)

New Idea

- Most applications require more than a single BFS traversal
- Instead of making one BFS faster, can we make batches of BFS traversals run faster?

Problem: How To Make BFS Faster

Previous Ideas

- Parallel BFS implementations
- Bottom-Up approach (Beamer et al.)

New Idea

- Most applications require more than a single BFS traversal
- Instead of making one BFS faster, can we make batches of BFS traversals run faster?

Table of Contents

Intuition

- Due to the small-world principle most real large graphs have a relatively small diameter compared to their size. Because of this most vertices are explored within a few steps of the BFS traversal.
- Concurrent BFS traversals are likely have a large overlap of what vertices they are exploring within a single step of a BFS traversal.
- Is there a way to efficiently store this overlap instead of each BFS maintaining their own data structures?

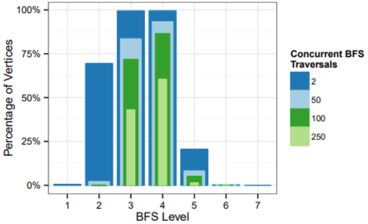


Figure 1: Percentage of vertex explorations that can be shared per level across 512 concurrent BFSs.

Table of Contents

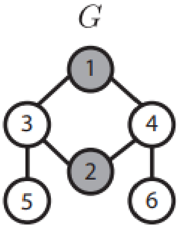
Multi-Source BFS Algorithm

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
    $visit \leftarrow visitNext$
    $visitNext \leftarrow \varnothing$

Walk-Through



$$\mathbb{B} = \{b_1, b_2\}$$
$$S = \{1, 2\}$$

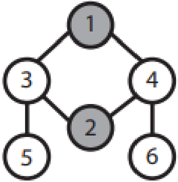
Initial State

$$seen_1 = \{b_1\}$$
$$seen_2 = \{b_2\}$$

$$visit = \left\{ \begin{matrix} (1, \{b_1\}) \\ (2, \{b_2\}) \end{matrix} \right\}$$

Walk-Through

G



$\mathbb{B} = \{b_1, b_2\}$

$S = \{1, 2\}$

1st BFS Level

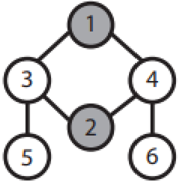
$seen_1 = \{b_1\} \quad seen_3 = \{b_1, b_2\}$

$seen_2 = \{b_2\} \quad seen_4 = \{b_1, b_2\}$

$$visit = \left\{ \begin{array}{l} (3, \{b_1\}) \\ (3, \{b_2\}) \\ (4, \{b_1\}) \\ (4, \{b_2\}) \end{array} \right\} \begin{array}{l} \mathbb{B}'_3 = \{b_1, b_2\} \\ \\ \mathbb{B}'_4 = \{b_1, b_2\} \end{array}$$

Walk-Through

G



$\mathbb{B} = \{b_1, b_2\}$

$S = \{1, 2\}$

2nd BFS Level

$$seen_1 = \{b_1, b_2\} \quad seen_4 = \{b_1, b_2\}$$
$$seen_2 = \{b_1, b_2\} \quad seen_5 = \{b_1, b_2\}$$
$$seen_3 = \{b_1, b_2\} \quad seen_6 = \{b_1, b_2\}$$

$$visit = \begin{Bmatrix} (5, \{b_1, b_2\}) \\ (6, \{b_1, b_2\}) \\ (1, \{b_2\}) \\ (2, \{b_1\}) \end{Bmatrix}$$

Table of Contents

A short coming of Iteration 1 was the overhead of runtime in maintaining the traversal sets ($b_i \in \beta$) when doing set operations. A solution to this was to this was the usage of bitmaps that have constant time operations.

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
```

Table of Contents

Cache Optimization

Many neighbors of a single vertex in the *visit* set are the neighbors of other vertices in the *visit* set. To avoid exploring the same neighbors multiple times (and possibly multiple cache misses for the same vertex), all neighbors are accumulated first before exploring them and adding them to the *visitNext* set

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
```

Table of Contents

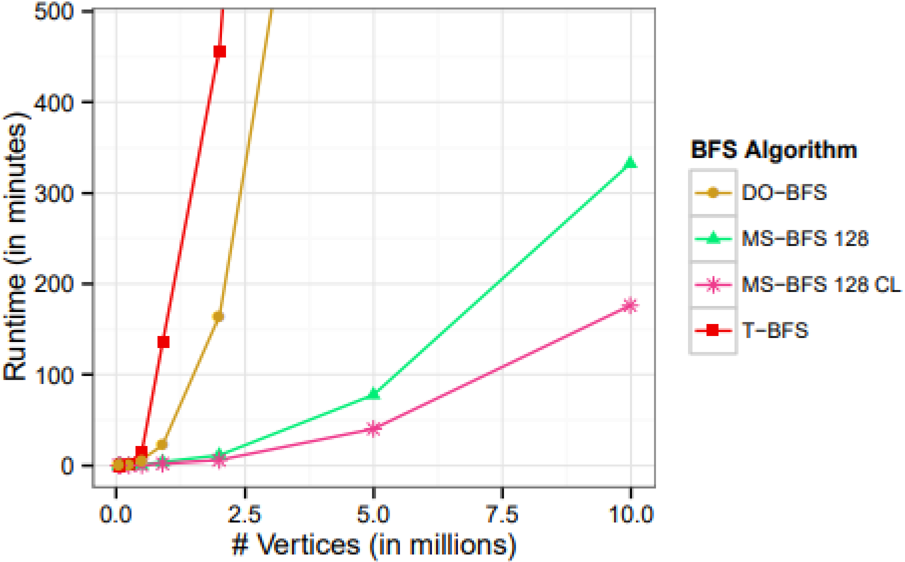


Figure 4: Data size scalability results.

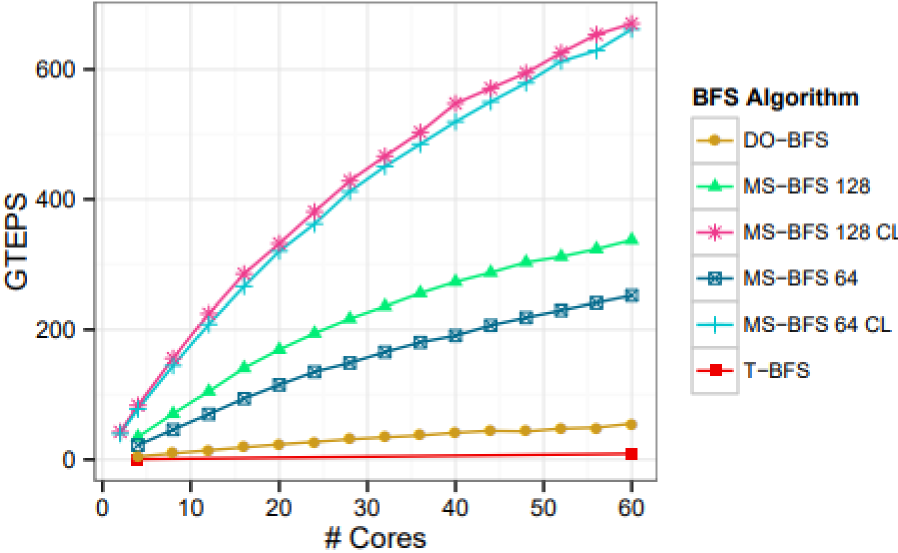


Figure 5: Multi-core scalability results.

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.