The More the Merrier:
Efficient Multi-Source Graph Traversal

Manuel Then*, Moritz Kaufmann*, Fernando Chirigati†, Tuan-Anh Hoang-Vu†, Kien Pham†, Alfons Kemper*, Thomas Neumann*, Huy T. Vo†

* Technische Universität München, † New York University

6.506 Algorithm Engineering – Paper Presentation

Presenter: Joseph Zhang

Problem and Background

Motivation

Graph analytics are becoming essential as more and more information is represented and manipulated as graphs

- social network analysis
- road network analysis
- web mining
- computational biology

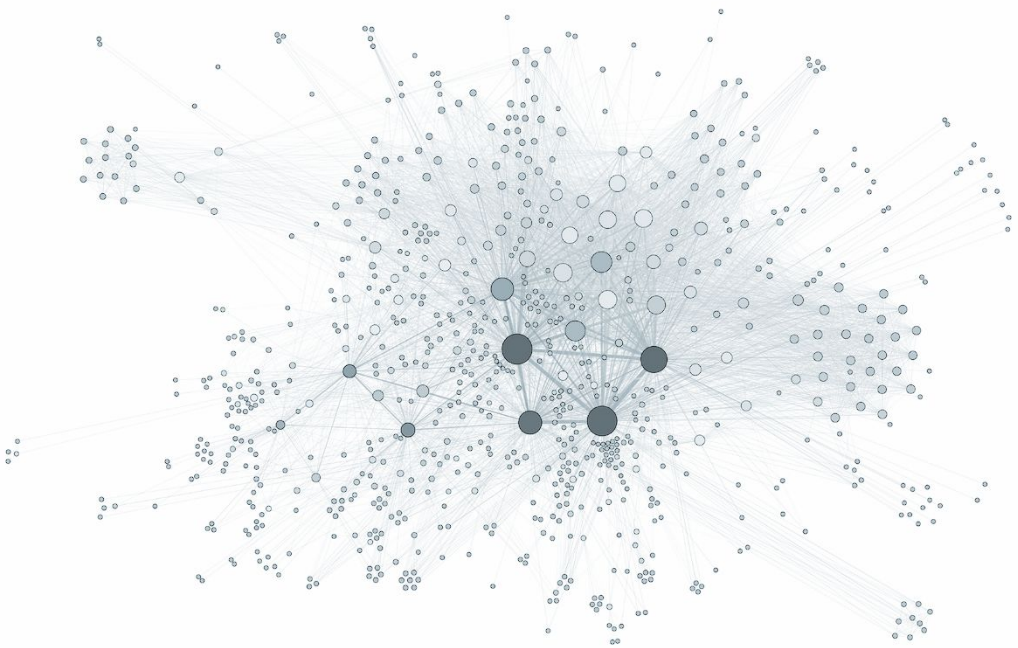
Breadth-First Search

BFS-based graph transversal is an important part of many graph analysis algorithms

- shortest path computation
- graph centrality calculation
- k-hop neighborhood detection

Often **computationally expensive ;-;**

- volume and nature of the data
- large datasets commonplace



Prior Work – Speeding Up BFS

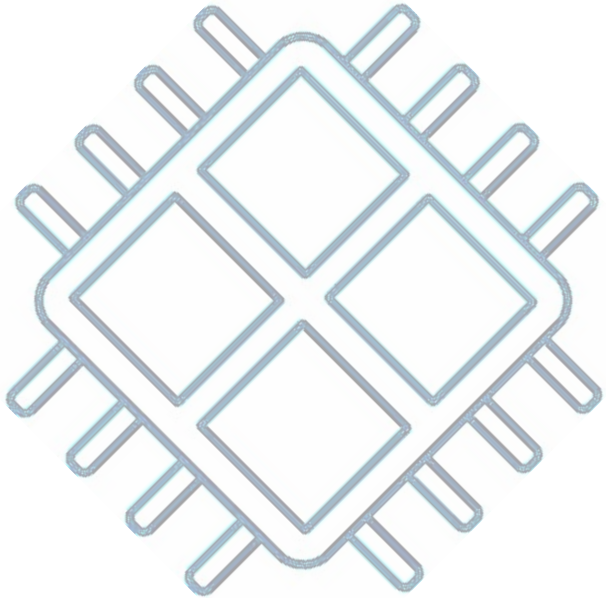
Taking advantage of parallelism from modern **multicore** systems

Focused on optimizing execution of single traversal (so single-source BFS)

Based around **exploring vertices in parallel** – issues:

- thread synchronization
- workload imbalance
- poor spatial and temporal locality of memory accesses

Distributed graph processing to span parallel graph traversals over multiple machines



Prior Work – Areas for Improvement

Many applications require **many BFSs** over same graph, e.g. one BFS from each vertex

- calculating graph centralities
- enumerating neighborhoods for all vertices
- solving all-pairs distance problem

Previous parallel BFS approaches are **inefficient for large graphs**

- they execute multiple single-thread BFSs in parallel, instead of parallel BFSs sequentially, to avoid synchronization and data transfer costs

Could instead **share computation** across multiple BFSs

- same vertex could be visited by various transversals!

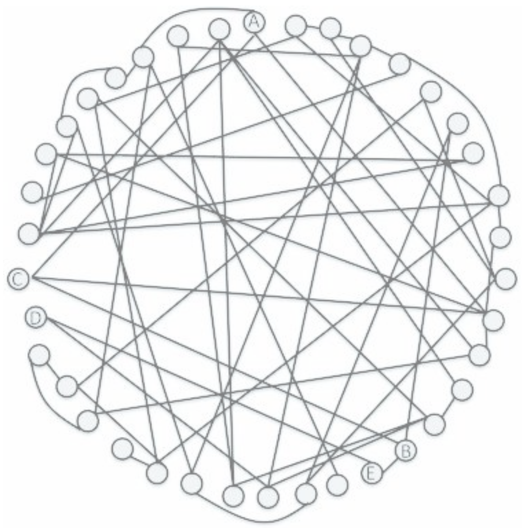
Small-World Networks

Distance between any two vertices small compared to size of graph (average **geodesic distance** increases logarithmically with graph size)

Few vertices have very many neighbors, most have few connections (scale-free networks)

Small-world networks **common in real-world graphs**: social networks, gene networks, neural networks, electrical power grids, and Web connectivity graphs, which can need graph analytics

Example: six degrees of separation theory – suggests everyone is only six or fewer steps away from each other, e.g. one study of 720 million facebook users showed 92% connected by just 5 steps



BFS Overview

BFS Algorithm (Single-Source)

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 ← ∅
```

Table 1: Number of newly discovered vertices in each BFS level for a small-world network.

Level	Discovered Vertices	≈ Fraction (%)
0	1	< 0.01
1	90	< 0.01
2	12,516	1.39
3	371,638	41.16
4	492,876	54.58
5	25,825	2.86
6	42	< 0.01

Vertex states during traversal:

- discovered = visited
- explored = edges and neighbors also visited

visit only contains vertices with same geodesic distance from source, i.e. in same **BFS level**, maximum level is diameter of graph (which is low in small-world networks)

- all vertices discovered in **few iterations**
- number of vertices discovered per level grows fast
- concurrent BFSs have high chance of discovering **common vertices** in same iteration

BFS Optimizations

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 ← ∅
```

Table 1: Number of newly discovered vertices in each BFS level for a small-world network.

Level	Discovered Vertices	≈ Fraction (%)
0	1	< 0.01
1	90	< 0.01
2	12,516	1.39
3	371,638	41.16
4	492,876	54.58
5	25,825	2.86
6	42	< 0.01

Small-world graphs tend to have few connected components (often just one), larger graph means many more vertices to see

BFS as shown currently has some potential issues:

- Lack of **memory locality** (many random accesses to *seen* and adjacency list)
- Later in traversal, most vertices **already discovered**, so many failed checks to *seen*
- **Bottom-up approach** can help, by iterating over non-discovered vertices and looking for edges to connect them to discovered ones

Prior work mainly focused on parallelizing a single BFS, using a **level-synchronous** approach

- requires synchronization of *visit* and *visitNext*
- race conditions when multiple threads access *seen*

Multi-Source BFS (MS-BFS)

MS-BFS Overview

Main goal: optimize execution of **multiple independent BFSs** on same graph, focused on non-distributed environment and in-memory processes, introduces new issues:

- memory locality issues from multiple traversals over same graph
- scalability would require very minimal resource usage
- avoid synchronization overheads which are high with many BFSs

Solutions:

- **share computation** across concurrent BFSs (small-world networks!)
- hundreds of BFSs executed in single CPU core
- use no locking nor atomic operations

MS-BFS Reasoning

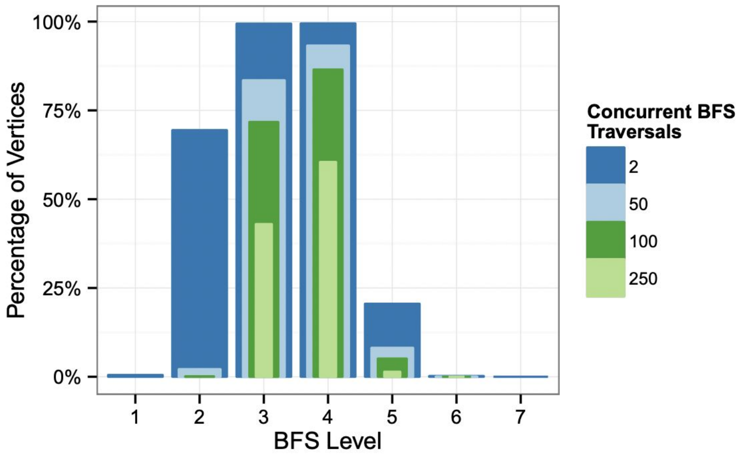


Figure 1: Percentage of vertex explorations that can be shared per level across 512 concurrent BFSs.

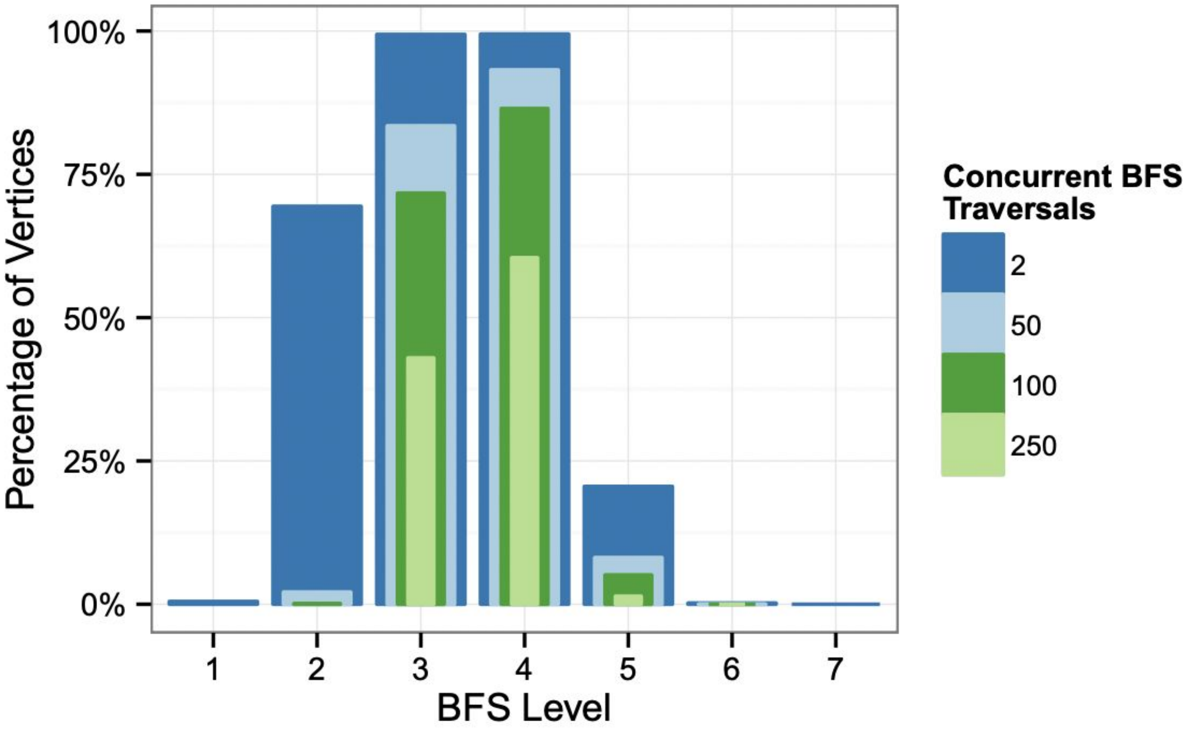
Idea: **combine accesses to same vertex** across multiple BFSs

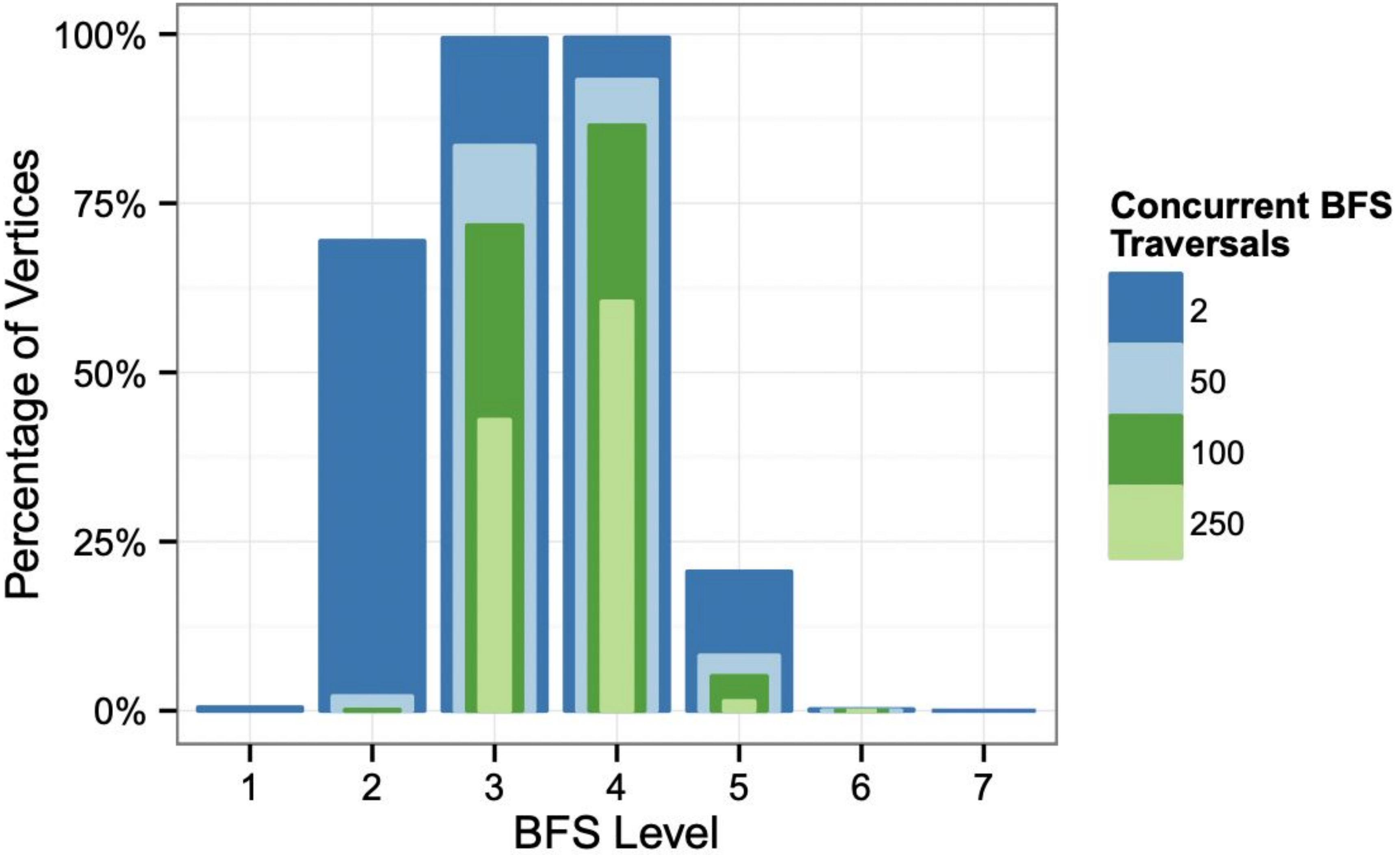
- amortize cache miss costs
- improve cache locality
- avoid redundant computation

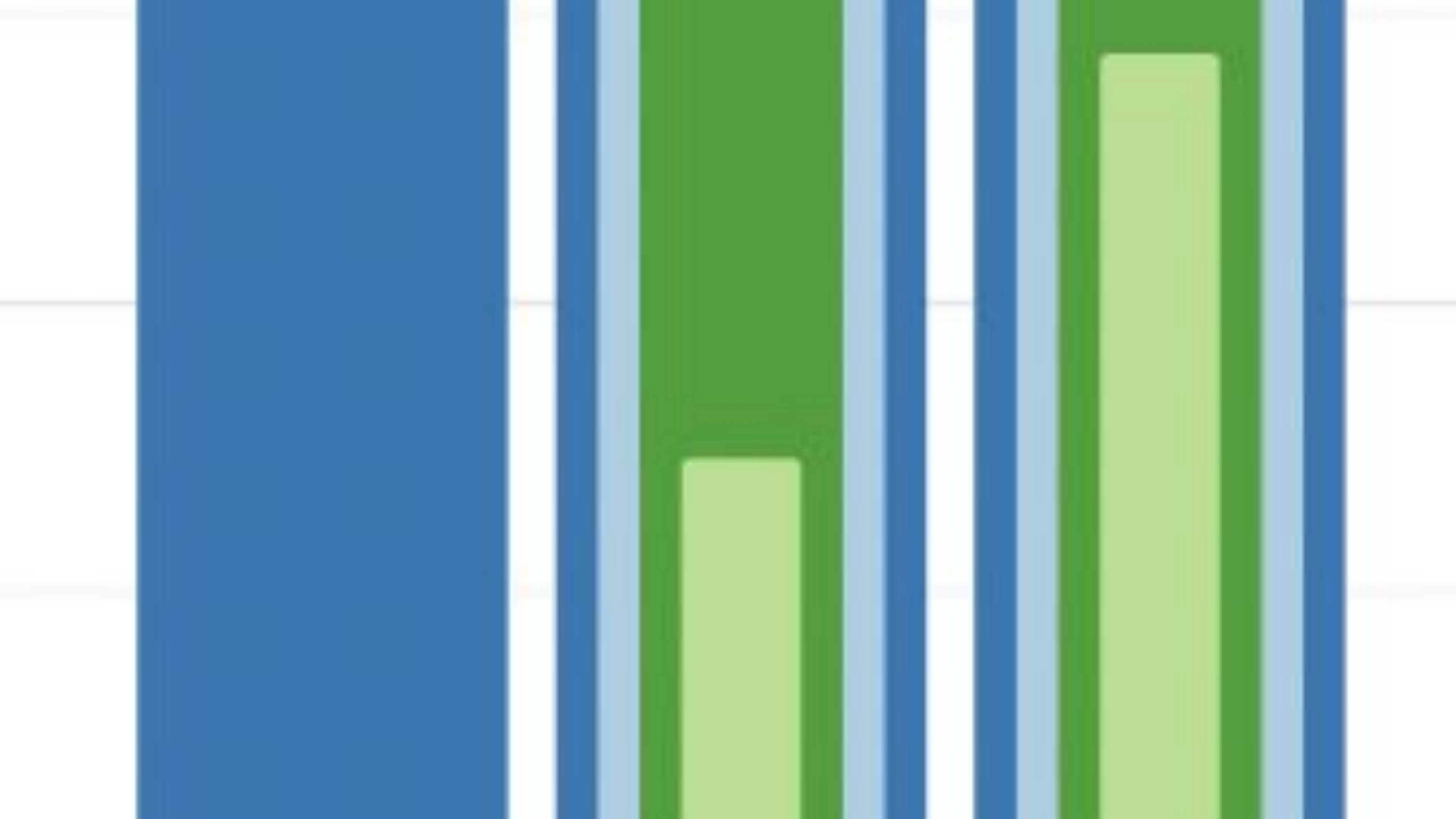
Analysis on LDBC graph with 1 million vertices shown in chart

For example, in level 4, we can explore more than 60% of vertices only once for 250 or more BFSs, instead of once for each BFS – **reduces memory accesses** significantly!

MS-BFS Reasoning




Percentage of Vertices
100%
75%
50%
25%
0%
1
2
3
4
5
6
7
BFS Level
Concurrent BFS Traversals
2
50
100
250



MS-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
    $visit \leftarrow visitNext$
    $visitNext \leftarrow \varnothing$

Additional inputs are sets of BFSs and their corresponding source vertices

Instead of single seen set, each vertex has its **own seen set** of BFSs that already discovered it

visit and *visitNext* contain tuples of vertices and **set of BFSs** that must explore them

For iterations in each BFS level, all BFS sets from *visit* that refer to selected vertex are **merged** into a set which now contains all BFSs that explore it in the level

MS-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
    $visit \leftarrow visitNext$
    $visitNext \leftarrow \varnothing$

For each neighbor *n* of *v*, we have set *D* of BFSs to explore it in the next level

If a BFS explores *v* in current level, and it has not discovered *n* yet, it must then explore *n*, so we update *visitNext* and seen set for *n* accordingly

Neighbors for *v* **traversed only once** for all BFSs in *D*, and each vertex *n* explored only once for them, significantly reducing memory accesses!

MS-BFS Example

Multiple BFSs executed concurrently and **share their explorations**, but vertices are still discovered and explored **sequentially** – different from parallel single BFS!

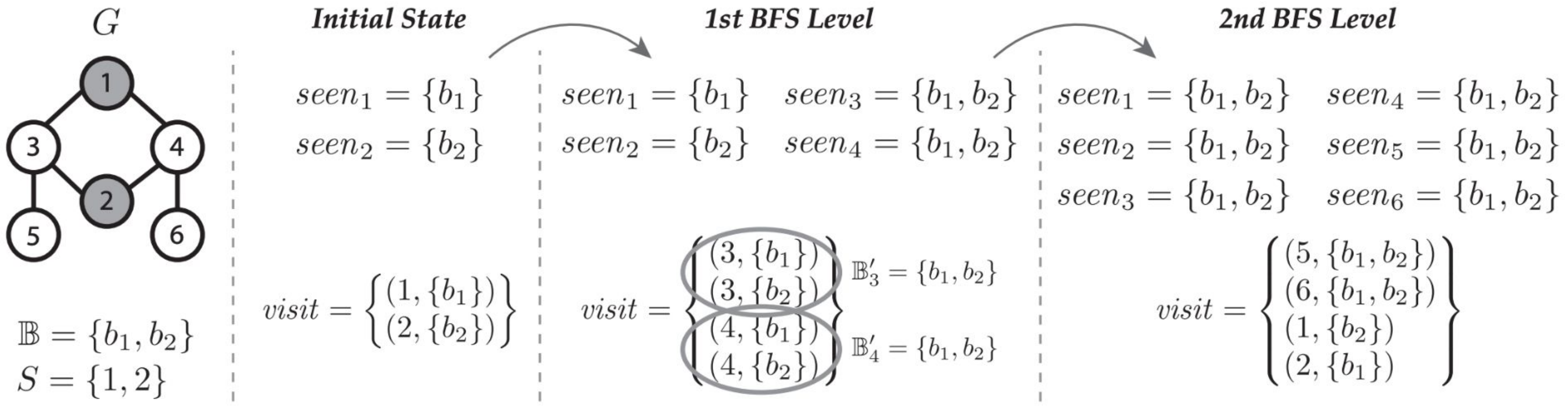


Figure 2: An example of the MS-BFS algorithm, where vertices 3 and 4 are explored once for two BFSs.

MS-BFS Bit Operations Optimizations

Listing 3: MS-BFS using bit operations.

Input: $G, \mathbb{B}, S$
for each $b_i \in \mathbb{B}$
    $seen[\ s_i\ ] \leftarrow 1 << b_i$
    $visit[\ s_i\ ] \leftarrow 1 << b_i$
reset $visitNext$

while $visit \neq \varnothing$
    for $i = 1, \ldots, N$
        if $visit[v_i] = \mathbb{B}_\varnothing$, **skip**
        for each $n \in neighbors[v_i]$
            $\mathbb{D} \leftarrow visit[v_i]\ \&\ {\sim}seen[n]$
            if $\mathbb{D} \neq \mathbb{B}_\varnothing$
                $visitNext[n] \leftarrow visitNext[n] \mid \mathbb{D}$
                $seen[n] \leftarrow seen[n] \mid \mathbb{D}$
                do BFS computation on n
    $visit \leftarrow visitNext$
    reset $visitNext$

In practice, all those union and set difference operations, and scans of visit, become **prohibitively expensive** for many concurrent BFSs

Idea: use efficient bit operations!

Represent the sets as **fixed-size bit fields**, fixing maximum concurrent BFSs to machine-specific parameter, such as multiple of register width

MS-BFS Bit Operations Example

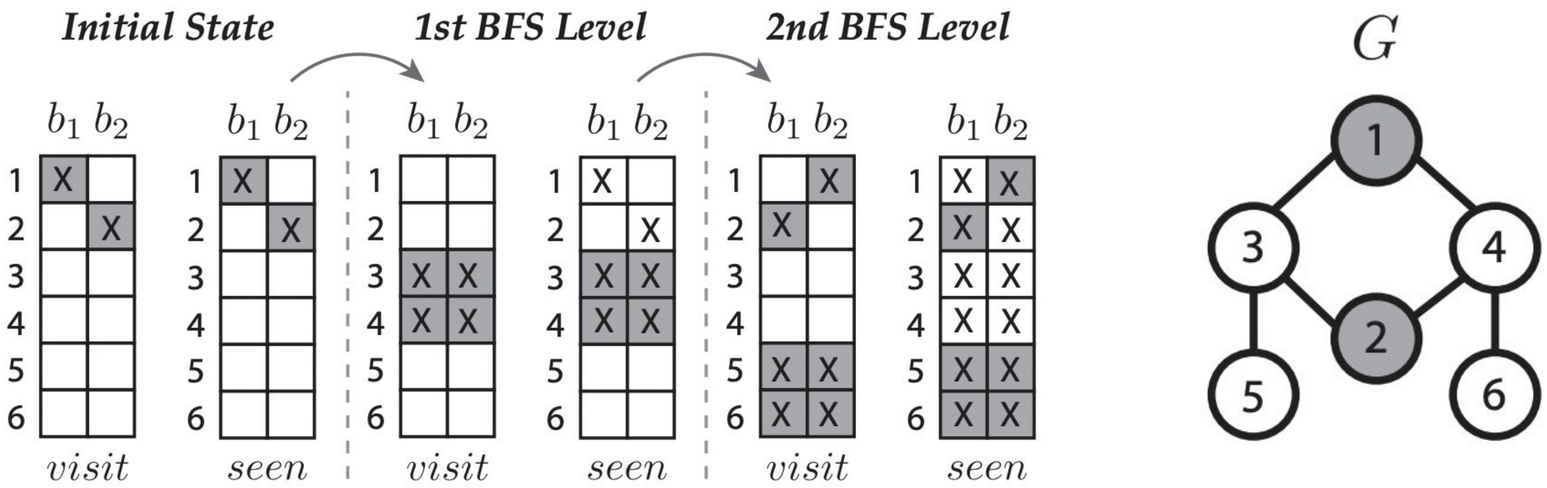


Figure 3: An example showing the steps of MS-BFS when using bit operations. Each row represents the bit field for a vertex, and each column corresponds to one BFS. The symbol X indicates that the value of the bit is 1.

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

Algorithm Tuning

Algorithm Tuning

Memory Access Tuning

Aggregated Neighbor Processing

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
```

Still have random accesses to *visitNext* and *seen* arrays, as well as possible repeated application-specific BFS computation

Idea: we can further reduce number of BFS computations and random accesses by first collecting then processing all vertices to be explored in next BFS level in **batch**

Removes dependency between *visit* and *seen* and BFS computation

Using ANP in MS-BFS

Listing 4: MS-BFS algorithm using ANP.

Input: $G, \mathbb{B}, S$
for each $b_i \in \mathbb{B}$
    $seen[\ s_i\ ] \leftarrow 1 << b_i$
    $visit[\ s_i\ ] \leftarrow 1 << b_i$
reset $visitNext$

while $visit \neq \varnothing$
    for $i = 1, \ldots, N$
        if $visit[v_i] = \mathbb{B}_{\varnothing}$, **skip**
        for each $n \in neighbors[v_i]$
            $visitNext[n] \leftarrow visitNext[n] \mid visit[v_i]$

    for $i = 1, \ldots, N$
        if $visitNext[v_i] = \mathbb{B}_{\varnothing}$, **skip**
        $visitNext[v_i] \leftarrow visitNext[v_i]\ \&\ {\sim}seen[v_i]$
        $seen[v_i] \leftarrow seen[v_i] \mid visitNext[v_i]$
        if $visitNext[v_i] \neq \mathbb{B}_{\varnothing}$
            do BFS computation on v_i
    $visit \leftarrow visitNext$
    reset $visitNext$

Process BFS level in **two stages**:

- Explore all vertices in *visit* to determine in which BFSs neighbors to be visited
- Sequentially iterate over these neighbors in *visitNext* and perform bit fields updates and BFS computations

For each discovered vertex, these steps are **only done once**, aggregating neighbor processing

Distributive property of binary operations

Using ANP in MS-BFS

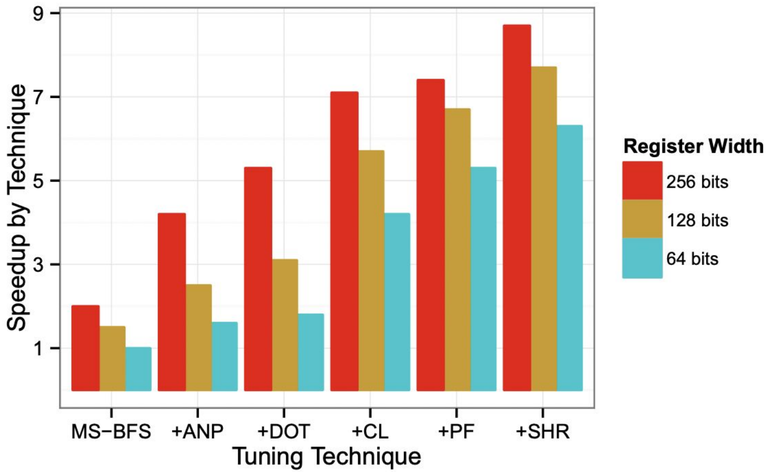


Figure 7: Speedup achieved by cumulatively applying different tuning techniques to MS-BFS.

Advantages of ANP:

- reduces memory accesses to seen
- sequential instead of random access to seen – better **memory locality**
- reduces BFS computation executions

Some effects of the advantages:

- improves **low-level cache** usage
- reduces cache misses

ANP speeds up MS-BFS by 60-110 %

Direction-Optimized Traversal

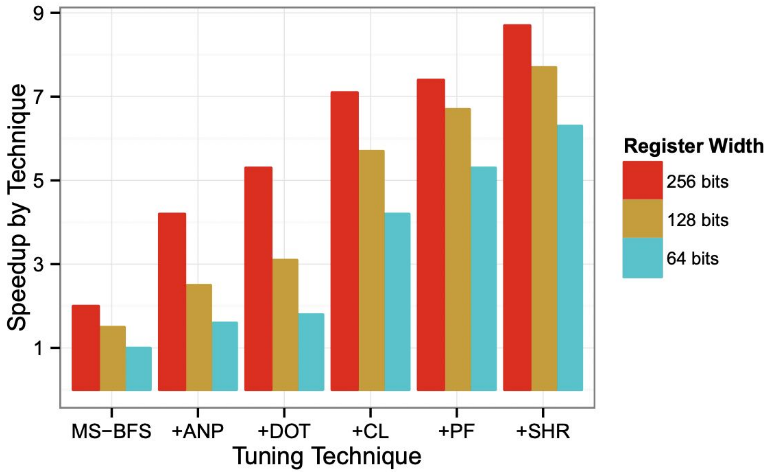


Figure 7: Speedup achieved by cumulatively applying different tuning techniques to MS-BFS.

Top-down – conventional BFS, go from discovered to non-discovered vertices

Bottom-up – opposite direction, explore non-discovered vertices

Heuristic based on number of non-traversed edges to choose strategy

Often top-down near beginning and bottom-up near end of search

Helps **reduce random accesses**

Neighbor Prefetching

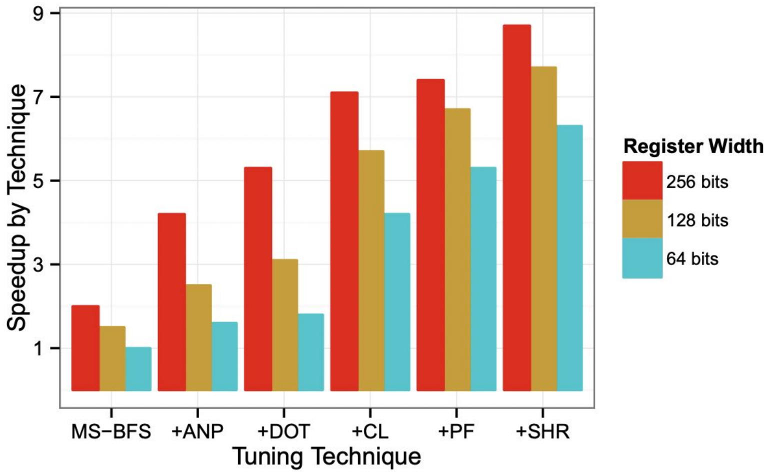


Figure 7: Speedup achieved by cumulatively applying different tuning techniques to MS-BFS.

ANP reduces random accesses to seen array, but we still have *visitNext* updates

Detect neighbors and **explicitly prefetch** some of these memory addresses, so that they are likely in **cache** when computing *visitNext* for them

Prefetching tens or hundreds of neighbors seemed to show some improvements in experiments

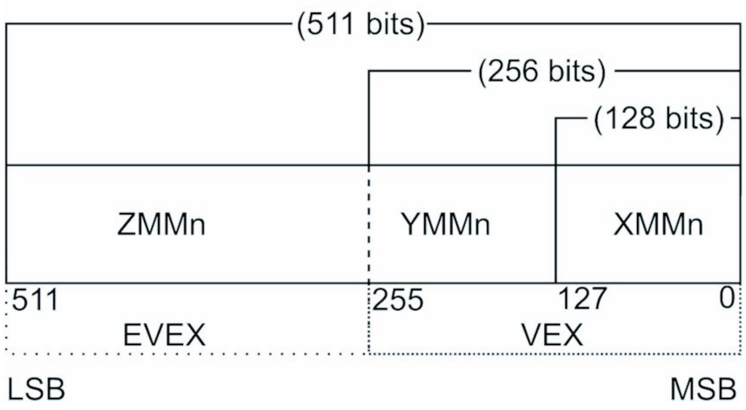
Algorithm Tuning

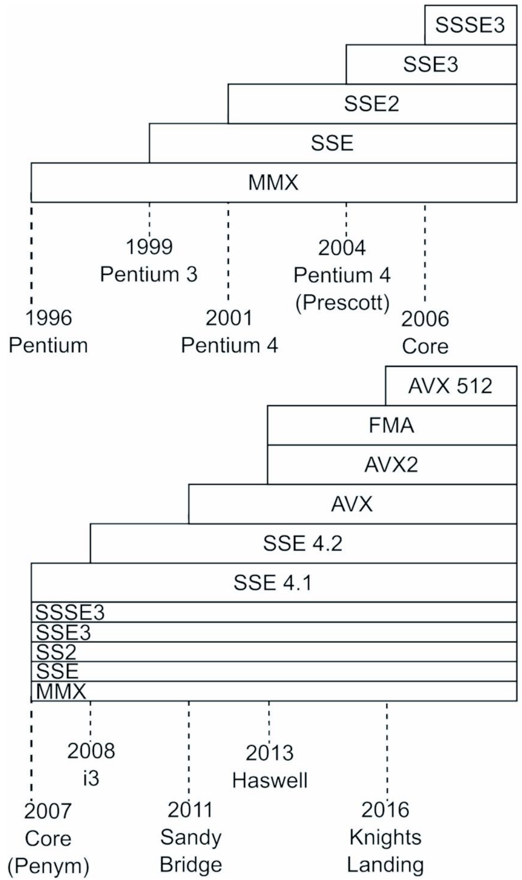
Execution Strategies

How Many BFSs?

MS-BFS bit operations more efficient using **native machine instructions**

Should set number of BFSs based on register and instruction width of CPU





Even More BFSs?

What if CPU-optimized number of BFSs just isn't enough?

Use **multiple registers** for the bit fields

- more shared vertex exploration
- can align to cache line boundaries

Execute multiple MS-BFS in **parallel**

- scales almost linearly with cores

Execute multiple MS-BFS **sequentially**

- lower memory requirements

We can also combine the three approaches!

Table 2: Memory consumption of MS-BFS for N vertices, ω-sized bit fields, and P parallel runs.

N	ω	P	Concurrent BFSs	Memory
1,000,000	64	1	64	22.8 MB
1,000,000	64	16	1,024	366.2 MB
1,000,000	64	64	4,096	1.4 GB
1,000,000	512	1	512	183.1 MB
1,000,000	512	16	8,192	2.9 GB
1,000,000	512	64	32,768	11.4 GB
50,000,000	64	64	4,096	71.5 GB
50,000,000	512	64	32,768	572.2 GB

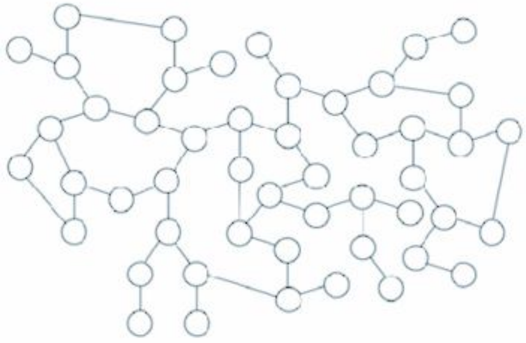
Maximum Sharing Heuristic

Recall that MS-BFS becomes faster as more BFSs explore same vertex in a given level

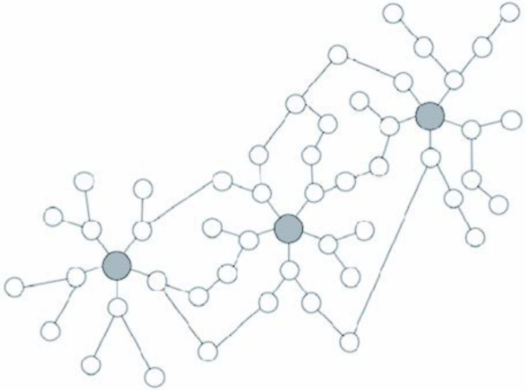
Group BFSs based on **connected components**, since if they're not running in the same one they can't share vertices or edges

Heuristic to group BFSs by their **source vertex degrees**

- small-world networks have low diameter and often few vertices with high degree (scale-free)
- intuition: vertices with higher degrees should have **many common neighbors**
- group BFSs based on sorting their source vertices by descending degree



(A) Random network



(B) Scale-free network

Application

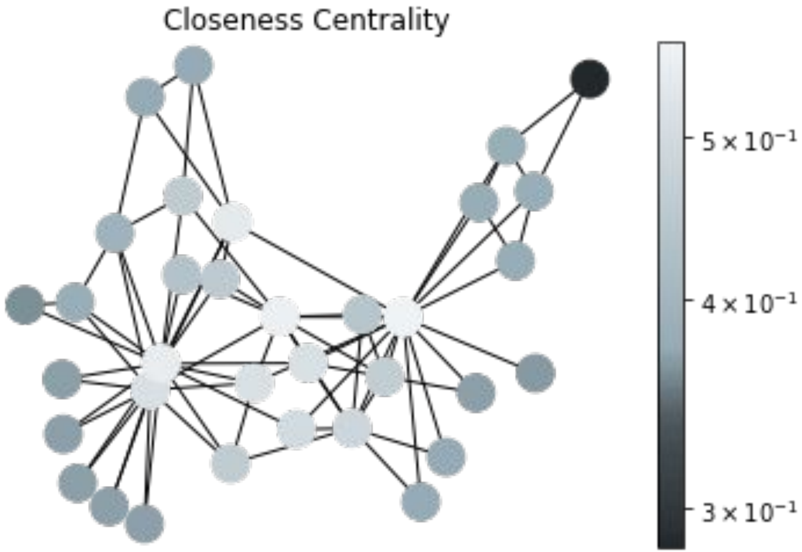
Closeness Centrality Computation

All-Vertices Closeness Centrality

Closeness centrality measures how close a vertex is to the rest of the vertices in the graph

To compute for all vertices, running a **BFS from each vertex** is needed!

Some further optimizations of the BFS computations can also be done to count discovered vertices per level efficiently



Experimental Evaluation

Experimental Evaluation

Experiment Setup

Algorithms and Datasets

Different BFS implementations:

- MS-BFS with various register widths, and also single vs. multiple registers per bit field
- non-parallel direction optimized BFS (DO-BFS)
- state-of-the-art BFS algorithm
- textbook BFS (T-BFS)

Table 3: Properties of the evaluated datasets.

Graph	Vertices (k)	Edges (k)	Diameter	Memory
LDBC 50k	50	1,447	10	5.7 MB
LDBC 100k	100	5,252	6	20.4 MB
LDBC 250k	250	7,219	10	28.5 MB
LDBC 500k	500	14,419	11	56.9 MB
LDBC 1M	1,000	81,363	8	314 MB
LDBC 2M	2,000	57,659	13	228 MB
LDBC 5M	5,000	144,149	13	569 MB
LDBC 10M	10,000	288,260	15	1.14 GB
Wikipedia	4,314	112,643	17	446 MB
Twitter	41,652	2,405,026	19	9.3 GB

Experimental Evaluation

Experiment Results

Scalability Results – Data Size

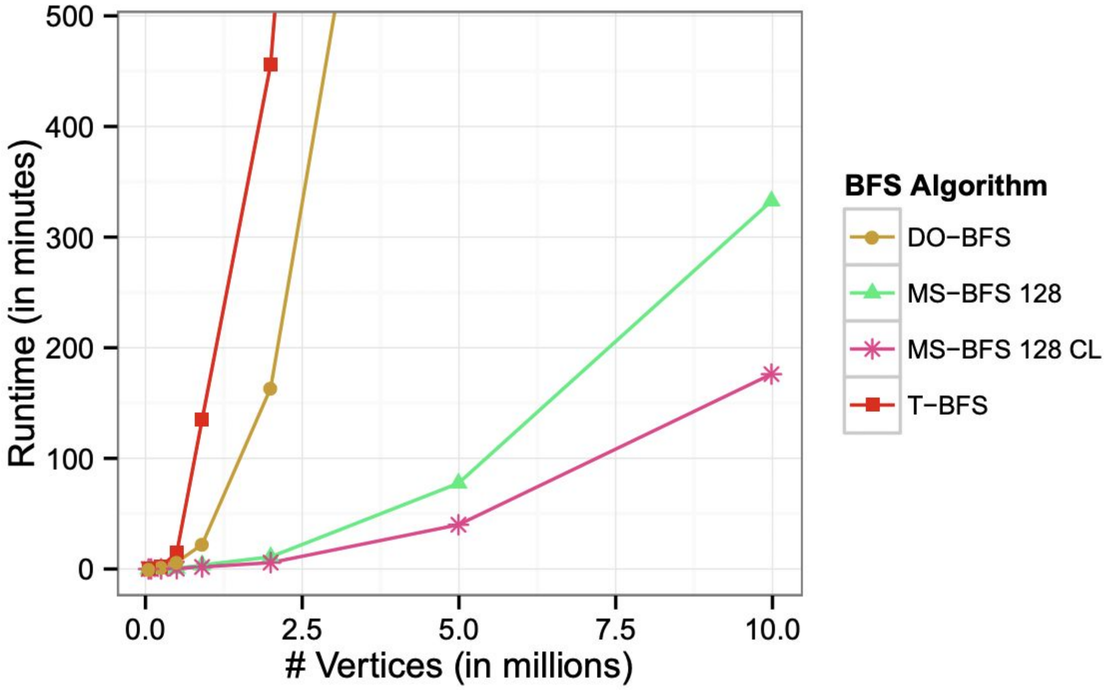


Figure 4: Data size scalability results.

CL indicates using multiple registers for single bit field to fill entire cache line

Scalability Results – Multicore

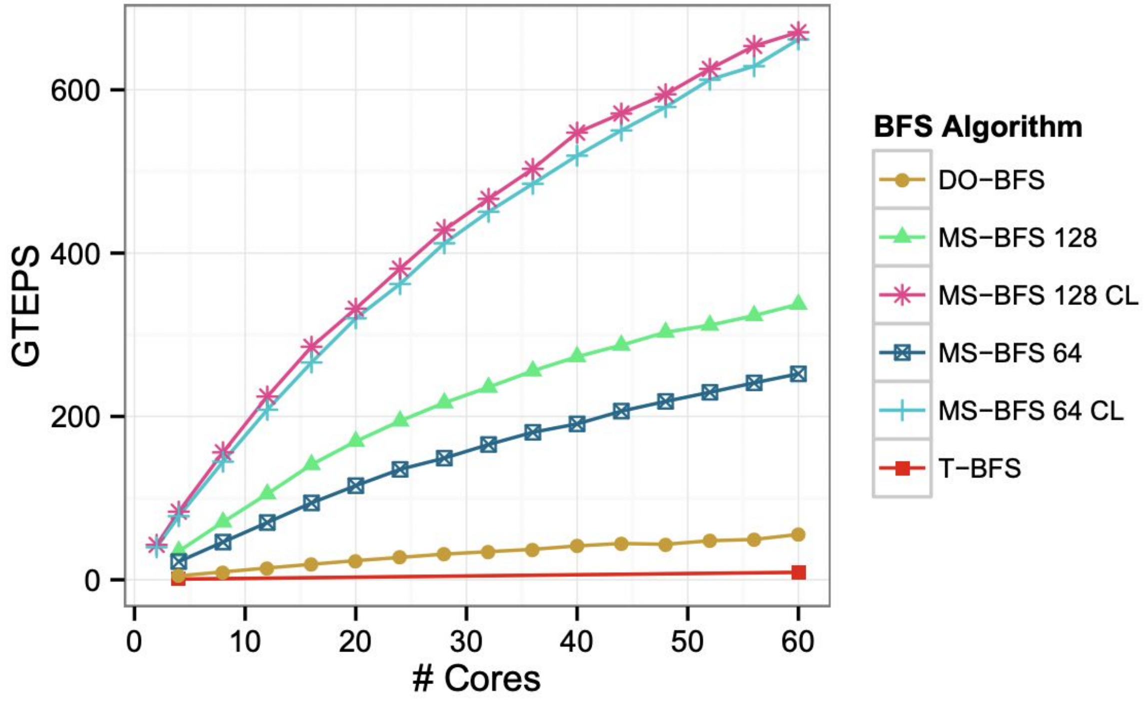


Figure 5: Multi-core scalability results.

CL indicates using multiple registers for single bit field to fill entire cache line

Scalability Results – BFS Count

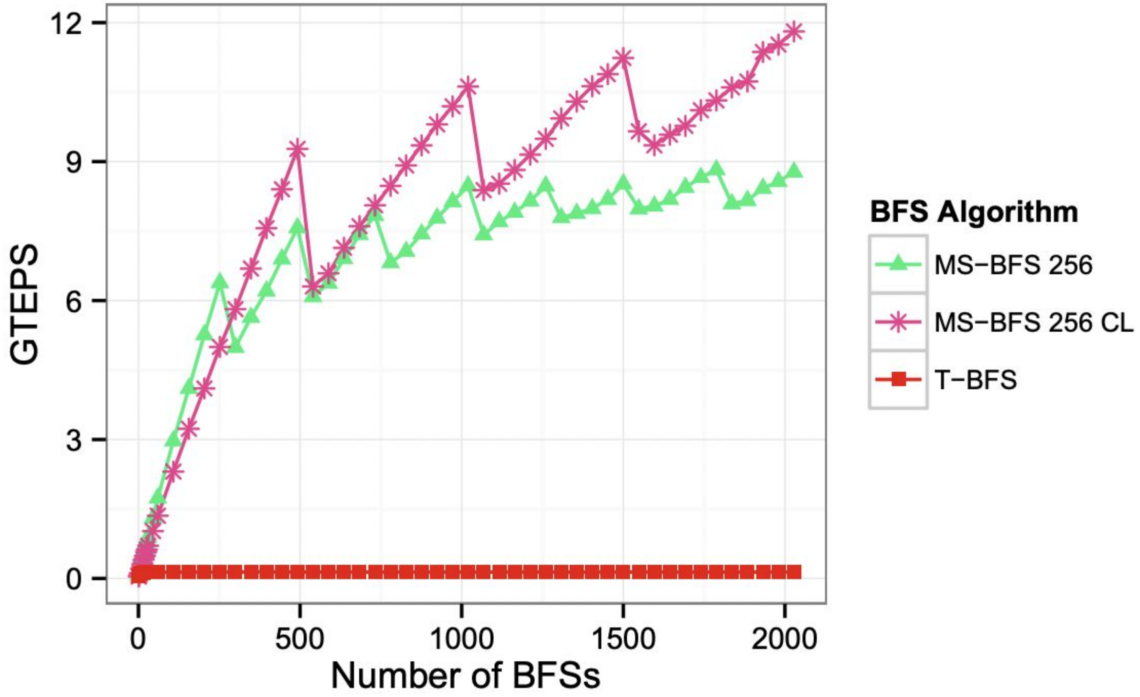


Figure 6: BFS count scalability results.

CL indicates using multiple registers for single bit field to fill entire cache line

Impact of Algorithm Tuning

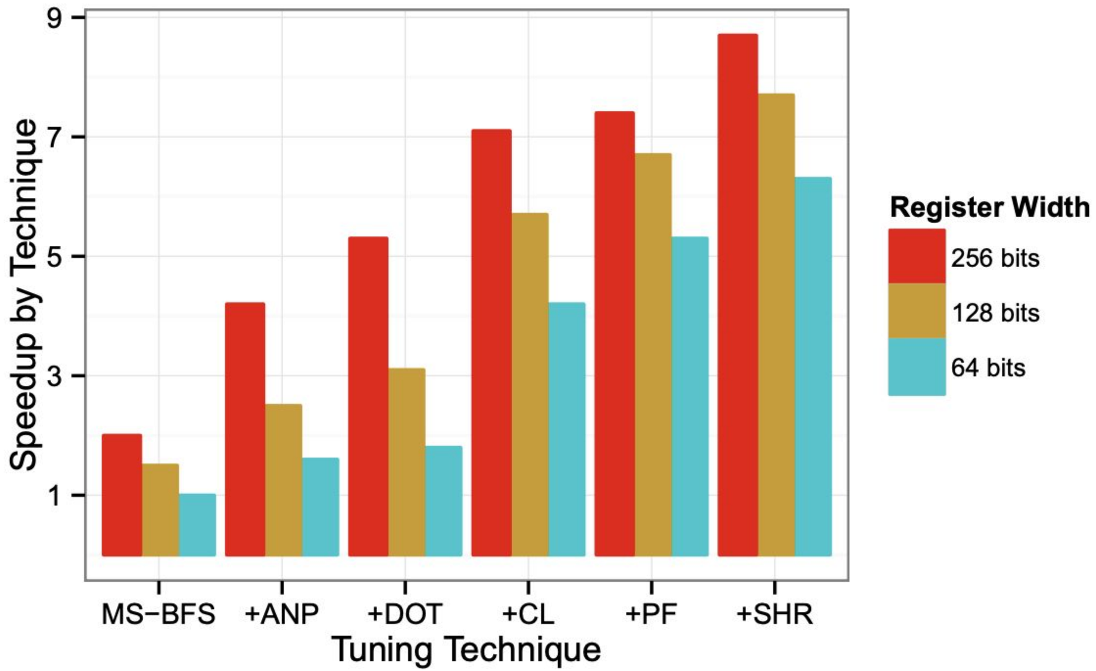


Figure 7: Speedup achieved by cumulatively applying different tuning techniques to MS-BFS.

ANP – aggregated neighbor processing

DOT – direction optimized traversal

CL – use of entire cache lines

PF – neighbor prefetching

SHR – heuristic for maximum sharing

Performance Summary

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.

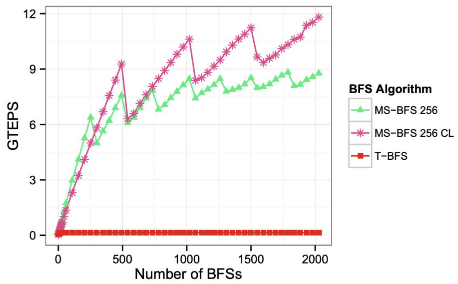


Figure 6: BFS count scalability results.

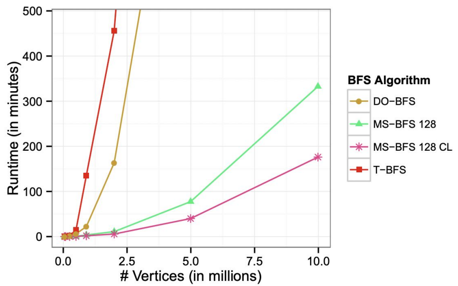


Figure 4: Data size scalability results.

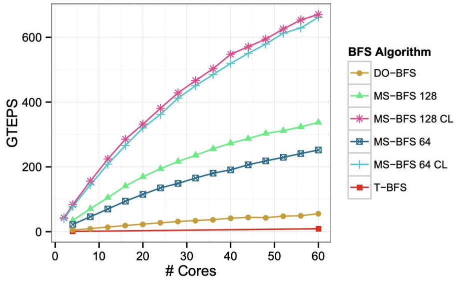


Figure 5: Multi-core scalability results.

Summary and Discussion

Summary

MS-BFS leverages **small-world network properties** to run multiple independent BFSs concurrently, with further algorithm, memory, and tuning optimizations to

- reduce random memory accesses
- amortize expensive cache misses
- utilize wide registers and efficient bit operations

Experimental results show MS-BFS outperforming existing solutions at running many BFSs on the same graph in terms of data and multicore **scalability** as well as **performance**

Possible directions for future work, such as

- combine approach with existing parallel BFS algorithms
- adapt MS-BFS for distributed environments and GPUS
- developing better heuristics for maximizing sharing
- applying MS-BFS to other analytics algorithms
- assessing MS-BFS on other types of graphs

Discussion

Some possible questions to consider:

What are some possible limitations of MS-BFS, and maybe possible directions we could explore to try to address them?

Thoughts on possible generalizations or extensions of some kind for the approaches given in the paper, for future work?

Some potential strengths and/or weaknesses of the work presented in the paper?