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.

```
1 Input: G, s
2 seen ← {s}
3 visit ← {s}
4 visitNext ← ∅
5
6 while visit ≠ ∅
7   for each v ∈ visit
8       for each n ∈ neighbors_v
9         if n ∉ seen
10            seen ← seen ∪ {n}
11            visitNext ← visitNext ∪ {n}
12            do BFS computation on n
13       visit ← visitNext
14     visitNext ← ∅
```

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
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

### Listing 1: Textbook BFS algorithm.

```
1 Input: G, s
2 seen ← \{s\}
3 visit ← \{s\}
4 visitNext ← ∅
5
6 while visit ≠ ∅
7     for each v ∈ visit
8         for each n ∈ neighbors\_v
9             if n ∉ seen
10                seen ← seen ∪ \{n\}
11                visitNext ← visitNext ∪ \{n\}
12             do BFS computation on n
13     visit ← visitNext
14     visitNext ← ∅
```

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

<table>
<thead>
<tr>
<th>Level</th>
<th>Discovered Vertices</th>
<th>≈ Fraction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>1</td>
<td>90</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>2</td>
<td>12,516</td>
<td>1.39</td>
</tr>
<tr>
<td>3</td>
<td>371,638</td>
<td>41.16</td>
</tr>
<tr>
<td>4</td>
<td>492,876</td>
<td>54.58</td>
</tr>
<tr>
<td>5</td>
<td>25,825</td>
<td>2.86</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
<td>&lt; 0.01</td>
</tr>
</tbody>
</table>
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

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!

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

![Graph showing percentage of vertices at different BFS levels with concurrent BFS traversals.]
MS-BFS Algorithm

Listing 2: The MS-BFS algorithm.

1 Input: $G, \mathbb{B}, S$
2 $seen_i \leftarrow \{b_i\}$ for all $b_i \in \mathbb{B}$
3 $visit \leftarrow \bigcup_{b_i \in \mathbb{B}} \{(s_i, \{b_i\})\}$
4 $visitNext \leftarrow \emptyset$
5
6 while $visit \neq \emptyset$
7   for each $v$ in $visit$
8     $\mathbb{B}'_v \leftarrow \emptyset$
9     for each $(v', \mathbb{B}') \in visit$ where $v' = v$
10        $\mathbb{B}'_v \leftarrow \mathbb{B}'_v \cup \mathbb{B}'$
11     for each $n \in \text{neighbors}_v$
12        $D \leftarrow \mathbb{B}'_v \setminus seen_n$
13        if $D \neq \emptyset$
14           $visitNext \leftarrow visitNext \cup \{(n, D)\}$
15           $seen_n \leftarrow seen_n \cup D$
16           do BFS computation on $n$
17     $visit \leftarrow visitNext$
18     $visitNext \leftarrow \emptyset$

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
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

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.

Listing 3: MS-BFS using bit operations.

```
1  Input: G, B, S
2  for each b_i \in B
3      seen[ s_i ] \leftarrow 1 \ll b_i
4      visit[ s_i ] \leftarrow 1 \ll b_i
5  reset visitNext
6
7  while visit \neq \emptyset
8      for i = 1, \ldots, N
9          if visit[v_i] = B_\emptyset, skip
10         for each n \in neighbors[v_i]
11            D \leftarrow visit[v_i] \& \sim seen[n]
12            if D \neq B_\emptyset
13               visitNext[n] \leftarrow visitNext[n] | D
14               seen[n] \leftarrow seen[n] | D
15               do BFS computation on n
16      visit \leftarrow visitNext
17  reset visitNext
```
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.

\[
\begin{align*}
\mathbb{B} &= \{b_1, b_2\} \\
S &= \{1, 2\}
\end{align*}
\]
Algorithm Tuning
Algorithm Tuning
Memory Access Tuning
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.**

```plaintext
1 Input: G, B, S
2 for each \( b_i \in B \)
3   \( \text{seen}[s_i] \leftarrow 1 \ll b_i \)
4   \( \text{visit}[s_i] \leftarrow 1 \ll b_i \)
5 reset \( \text{visitNext} \)
6
7 while \( \text{visit} \neq \emptyset \)
8   for \( i = 1, \ldots, N \)
9      if \( \text{visit}[v_i] = B_{\emptyset}, \text{skip} \)
10     for each \( n \in \text{neighbors}[v_i] \)
11        \( \text{visitNext}[n] \leftarrow \text{visitNext}[n] | \text{visit}[v_i] \)
12
13   for \( i = 1, \ldots, N \)
14      if \( \text{visitNext}[v_i] = B_{\emptyset}, \text{skip} \)
15      \( \text{visitNext}[v_i] \leftarrow \text{visitNext}[v_i] \& \sim \text{seen}[v_i] \)
16      \( \text{seen}[v_i] \leftarrow \text{seen}[v_i] | \text{visitNext}[v_i] \)
17      if \( \text{visitNext}[v_i] \neq B_{\emptyset} \)
18         do BFS computation on \( v_i \)
19         \( \text{visit} \leftarrow \text{visitNext} \)
20      reset \( \text{visitNext} \)
```

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

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

**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*

---

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

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.

Figure 7: Speedup achieved by cumulatively applying different tuning techniques to MS-BFS.
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>
<thead>
<tr>
<th></th>
<th>$N$</th>
<th>$\omega$</th>
<th>$P$</th>
<th>Concurrent BFSs</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>64</td>
<td>1</td>
<td>64</td>
<td>22.8 MB</td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>64</td>
<td>16</td>
<td>1,024</td>
<td>366.2 MB</td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>64</td>
<td>64</td>
<td>4,096</td>
<td>1.4 GB</td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>512</td>
<td>1</td>
<td>512</td>
<td>183.1 MB</td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>512</td>
<td>16</td>
<td>8,192</td>
<td>2.9 GB</td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>512</td>
<td>64</td>
<td>32,768</td>
<td>11.4 GB</td>
<td></td>
</tr>
<tr>
<td>50,000,000</td>
<td>64</td>
<td>64</td>
<td>4,096</td>
<td>71.5 GB</td>
<td></td>
</tr>
<tr>
<td>50,000,000</td>
<td>512</td>
<td>64</td>
<td>32,768</td>
<td>572.2 GB</td>
<td></td>
</tr>
</tbody>
</table>
Maximum Sharing Heuristic

Recall that MS-BFS becomes faster as more BFSs explore the 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.
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
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.

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

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

*Execution aborted after 8 hours; runtime estimated.
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?