The More the Merrier: Efficient Multi-Source Graph Traversal

Manuel Then* then@in.tum.de
Moritz Kaufmann* kaufmanm@in.tum.de
Fernando Chirigati† fchirigati@nyu.edu
Tuan-Anh Hoang-Vu† tuananh@nyu.edu
Kien Pham† kien.pham@nyu.edu
Alfons Kemper* kemper@in.tum.de
Thomas Neumann* neumann@in.tum.de
Huy T. Vo† huy.vo@nyu.edu

* Technische Universität München
† New York University

ABSTRACT

Graph analytics on social networks, Web data, and communication networks has been widely used in a plethora of applications. Many graph analytics algorithms are based on breadth-first search (BFS) graph traversal, which is not only time-consuming for large datasets but also involves much redundant computation when executed multiple times from different start vertices. In this paper, we propose *Multi-Source BFS* (MS-BFS), an algorithm that is designed to run multiple concurrent BFSs over the same graph on a single CPU core while scaling up as the number of cores increases. MS-BFS leverages the properties of *small-world networks*, which apply to many real-world graphs, and enables efficient graph traversal that: (i) shares common computation across concurrent BFSs; (ii) greatly reduces the number of random memory accesses; and (iii) does not incur synchronization costs. We demonstrate how a real graph analytics application—all-vertices closeness centrality—can be efficiently solved with MS-BFS. Furthermore, we present an extensive experimental evaluation with both synthetic and real datasets, including Twitter and Wikipedia, showing that MS-BFS provides almost linear scalability with respect to the number of cores and excellent scalability for increasing graph sizes, outperforming state-of-the-art BFS algorithms by more than one order of magnitude when running a large number of BFSs.

1. INTRODUCTION

An ever-growing amount of information has been stored and manipulated as graphs. To better comprehend and assess the relationships between entities in this data, as well as to uncover patterns and new insights, graph analytics has become essential. Numerous applications have been extensively using graph analytics, including social network analysis, road network analysis, Web mining, and computational biology. A typical example in the field of social networks is identifying the most central entities, as these potentially have influence on others and, as a consequence, are of great importance to spread information, e.g., for marketing purposes [20].

Proceedings of the VLDB Endowment, Vol. 8, No. 4

In a wide range of graph analytics algorithms, including shortest path computation [13], graph centrality calculation [9, 27], and k-hop neighborhood detection [12], *breadth-first search* (BFS)-based *graph traversal* is an elementary building block used to systematically *traverse* a graph, i.e., to visit all reachable vertices and edges of the graph from a given start vertex. Because of the volume and nature of the data, BFS is a computationally expensive operation, leading to long processing times, in particular when executed in large datasets that are commonplace in the aforementioned fields.

To speed up BFS-based graph analytics, significant research has been done to develop efficient BFS algorithms that can take advantage of the parallelism provided by modern multi-core systems [2, 5, 7, 14, 18]. They optimize the execution of a single traversal, i.e., a single BFS, mostly by visiting and exploring vertices in a parallel fashion. Hence, previous work had to address not only parallelization-specific issues, such as thread synchronization and the presence of workload imbalance caused by skew, but also fundamental challenges in graph processing, including poor spatial and temporal locality in the memory access pattern [24]. Recent work on processing graphs in distributed environments—including scalable traversal techniques [10, 11, 30], graph databases [26, 33], and platforms for distributed analytics [15, 23, 25, 31]—can be used to span the execution of parallel graph traversals to multiple machines, improving the overall performance and coping with data that is partitioned across different nodes.

Although many graph analytics algorithms (e.g., shortest path computation on unweighted graphs) involve executing single BFSs and can make good use of the existing parallel implementations, there is a plethora of other applications that require hundreds (or even millions) of BFSs over the same graph—in many cases, one BFS is needed from each vertex of the graph. Examples of such applications include calculating graph centralities, enumerating the neighborhoods for all vertices, and solving the all-pairs shortest distance problem. These scenarios do not fully benefit from current parallel BFS approaches: often, the best one can do in order to reduce the overall runtime is to execute multiple single-threaded BFSs in parallel, instead of running parallel BFSs sequentially, because the former avoids synchronization and data transfer costs, as we discuss in Section 6. By doing so, however, one misses opportunities for sharing computation across multiple BFSs when the same vertex is visited by various traversals. This hampers scalability, making single BFS traversals inefficient for large graphs.

In this paper, we propose *Multi-Source BFS* (MS-BFS), a new BFS algorithm for modern multi-core CPU architectures designed for graph analytics applications that run a large number of BFSs from multiple vertices of the same graph. MS-BFS takes an approach that is orthogonal to previous work: instead of parallelizing a single BFS, we focus on processing a large number of BFSs *concurrently in a single core*, while still allowing to scale up to multiple cores. This approach allows us to share computation between different BFSs without incurring synchronization cost.

This work leverages properties of *small-world networks* [3]: 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. Concretely, this means that a BFS in such a network discovers most of the vertices in few iterations, and concurrent BFSs over the same graph have a high chance of having overlapping sets of discovered vertices in the same iteration. Based on these properties, in MS-BFS, we *combine* accesses to the same vertex across multiple BFSs. This amortizes cache miss costs, improves cache locality, avoids redundant computation, and reduces the overall runtime. Note that small-world networks are commonplace as these properties apply to many real-world graphs, including social networks, gene networks, and Web connectivity graphs, which are of interest for many graph analytics applications.

MS-BFS executes concurrent BFSs in a data-parallel fashion that can be efficiently implemented using bit fields in wide CPU registers and requires neither locks nor atomic operations. We assume that the graph fits in main memory, which is a realistic assumption for many real-world graphs and applications (e.g., the Who to Follow service at Twitter [17]) as modern servers can store up to hundreds of billions of edges in memory. This design choice avoids overheads from disk accesses and network roundtrips, allowing unprecedented analytics performance. Nevertheless, the optimized variants of our algorithm, described in Sections 3.2 and 4.1.1, allow data to be scanned sequentially and thus can be easily adapted to provide good performance for disk-resident graphs as well.

MS-BFS is a generic BFS algorithm that can be applied to many graph problems that run multiple traversals from different start vertices. As an example of a real application, we demonstrate how it can be used to efficiently solve the computationally expensive problem of calculating the closeness centrality values for all vertices in a graph. We also present an extensive experimental evaluation using synthetic datasets generated with the LDBC Social Network Data Generator [8, 21], as well as real-world graphs from Twitter and Wikipedia, showing that MS-BFS (i) outperforms existing BFS algorithms by more than one order of magnitude when running multiple BFSs, and (ii) scales almost linearly with respect to graph size and number of cores. It is worth noting that the approach presented here was successfully used by the two leading teams—first and second place—in the SIGMOD 2014 Programming Contest.

Overall, our contributions are as follows:

- We propose Multi-Source BFS (MS-BFS), a graph traversal algorithm that can efficiently execute multiple concurrent BFSs over the same graph using a single CPU core (Section 3). MS-BFS is most efficient in small-world networks, where it combines the execution of multiple BFSs in a synchronization-free manner, improves the memory access pattern, and avoids redundant computation. We also discuss tuning strategies to further improve the performance of the algorithm (Section 4).
- To demonstrate the feasibility of our approach, we show how a real graph analytics application—the all-vertices closeness centrality computation—can be efficiently implemented using MS-BFS (Section 5).
- We present an extensive experimental evaluation[1] with synthetic and real datasets, showing that MS-BFS scales almost linearly with an increasing number of CPU cores and that it exhibits excellent scalability under changes to the input graph size. We further show that MS-BFS greatly outperforms existing state-of-the-art BFS algorithms when running multiple BFSs (Section 6).

2. BACKGROUND

In this paper, we consider an unweighted graph $G = \{V, E\}$, where $V = \{v_1, \ldots, v_N\}$ is the set of vertices, $E = \{neighbors_{v_i}|_{i=1}^{N}\}$ is the set of edges, $neighbors_v$ is the set of vertices to which v connects (neighbor vertices of v), and N is the number of vertices in the graph. We further assume that G exhibits properties of *small-world networks* (Section 2.1), and that the graph analytics algorithms to be used over G is based on BFS (Section 2.2).

2.1 Small-World Networks

In small-world networks, as the graph size increases, the average geodesic distance—the number of edges—between vertices increases logarithmically. In other words, we say that a graph G has *small-world properties* if its diameter, i.e., the longest distance between any two vertices in G, is low even for a large N [3]. Another property of many small-world graphs is that their distribution of *vertex degree*, i.e., the number of neighbors of a vertex, follows a power law relationship: few vertices have a very high number of neighbors, while most of the vertices have few connections. Graphs exhibiting the latter property are also known as *scale-free networks* [3].

A famous example of these properties is the six degrees of separation theory, suggesting that everyone is only six or fewer steps away from each other. A recent study showed that 92% of Facebook users ($N \approx 720$ million) are connected by only 5 steps, and that the average distance between users is 4.74 [4]. In fact, besides social networks, many other real-world graphs that are of critical interest for both academia and industry—including gene and neural networks, the world-wide web, wikis, movie-actor and scientific collaboration networks, and electrical power grids—exhibit small-world properties [3].

2.2 Breadth-First Search Overview

Breadth-first search (BFS) is an algorithm to systematically traverse a graph G from a given start vertex, or *source*, $s \in V$. We present the original BFS algorithm, to which we refer as *textbook BFS*, in Listing 1. There are two main states for a vertex during a BFS traversal: *discovered*, which means

[1] Code available at `https://github.com/mtodat/ms-bfs`

that the BFS has already visited the vertex, and *explored*, which means that not only the vertex but also its edges and neighbors have been visited. The algorithm starts by adding s to *seen*, which is the set of vertices that have been discovered. It also adds the source vertex to *visit*, which is the set of vertices yet to be explored. By iterating over *visit* in Line 7, vertices in *visit* are explored to find new reachable vertices. Vertices connected to v (Line 8) that have not been discovered yet (Line 9) are added to both *seen* and *visitNext*. Furthermore, newly discovered vertices are processed by the graph analytics application that uses the BFS (Line 12), e.g., a shortest path algorithm stores the distance between s and n. The *visitNext* set becomes the next *visit* set after all the vertices in the current one have been explored (Lines 13 and 14).

Note that for every iteration in Line 6, *visit* only contains vertices that have the same geodesic distance from the source s: we say that these vertices are in the same *BFS level*. The maximum number of levels that any BFS can have in G is equivalent to the diameter of G. Since G is a small-world network, its diameter is low, which means that a BFS will have a small number of levels as well: all vertices are discovered in few iterations, and the number of vertices discovered in each level grows rapidly. Table 1 shows this behavior in a synthetic dataset of 1 million vertices (generated with the data generator from LDBC), where a BFS is run over a connected component that comprises more than 90% of the vertices of the graph. Note that the number of BFS levels is small compared to the size of the graph, and that nearly 95% of the vertices are discovered in BFS levels 3 and 4.

In a traditional implementation of the BFS algorithm, list or queue data structures are often used for *visit* and *visitNext*, while *seen* is represented by either a list or a hash set. The set E of edges is usually implemented as an adjacency list, where each vertex has its own list of neighbors, i.e., $neighbors_v$ is a list containing all the neighbor vertices of v.

Optimizing BFS. Small-world graphs tend to have few connected components. Often, the entire graph is a single component, which means that every vertex is reachable from every other vertex. As a consequence, the larger the traversed graph, the more vertices and edges need to be visited by the BFS, which becomes a significantly time-consuming operation. This issue is exacerbated by BFS's lack of memory locality, as shown in the random accesses to *seen* and to the adjacency list (Lines 8 and 9), reducing the usefulness of CPU caches. Furthermore, towards the end of the BFS execution, most of the vertices will have been already discovered (see Table 1), and there will be much fewer non-discovered vertices than vertices in the *visit* set. As a consequence, there will be a significant number of failed checks to *seen* (Line 9) that consume resources unnecessarily [7].

Optimizing the execution of the BFS algorithm for large datasets is essential for graph analytics, and there has been substantial work in this direction. Most of this work is focused on implementing a *single parallel BFS*, i.e., parallelizing a single BFS execution, mainly by making use of the *level-synchronous* approach: vertices are explored and discovered in parallel for each BFS level, i.e., the loops in Lines 7 and 8 are executed in parallel for each level. The main idea of this approach is to divide the work across multiple cores and thus speed up the execution of one BFS. However, this comes at a cost: *visit* and *visitNext* must be synchronized at the end of each BFS level (before a new iteration at Line 6 starts), and race conditions must be avoided 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 ← ∅
```

For shared-memory and multi-core CPUs, numerous optimizations have been proposed to efficiently implement the level-synchronous approach and to address the foregoing challenges [2, 5, 6, 7, 14, 18], including the use of more efficient data structures, mechanisms to improve memory locality, e.g., sequential access to data structures [18], and further optimizations specific to certain hardware architectures. Notably, Beamer et al. [7] propose a bottom-up approach to avoid many unnecessary checks to *seen* as mentioned before: instead of visiting new vertices by looking at the outgoing edges of discovered vertices, the approach iterates over non-discovered vertices, looking for edges that connect them to vertices that have already been discovered (i.e., that are in *visit*). The authors combine the textbook BFS with the bottom-up approach in a hybrid *direction-optimized* technique. Although their implementation is for single parallel BFSs, the optimization is mostly orthogonal to parallelization and can be used for sequential execution. We further use this technique to optimize our algorithm (Section 4.1.2) and for comparison purposes (Section 6).

Executing the BFS algorithm in distributed memory systems has also been extensively studied before [10, 11, 30], as this raises a new range of issues, including the need to manage communication between CPUs and to partition the graph among processors, which is challenging and can deeply impact performance. Frameworks such as Parallel BGL [16], Pregel [25], Trinity [31], GraphLab [23], and Teradata Aster [32] provide APIs to facilitate the scale-out of graph traversal and other graph algorithms to multiple nodes. Distributed graph databases, e.g., Titan [33], allow

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

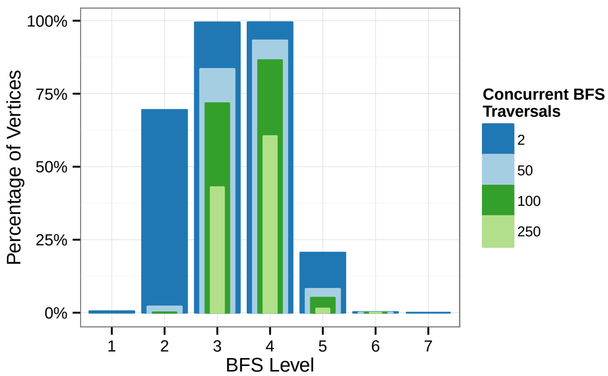


Figure 1: Percentage of vertex explorations that can be shared per level across 512 concurrent BFSs.

users to store and query graphs that are partitioned in multi-machine clusters, and engines such as Faunus [15] can be used on top of these databases to optimize the performance when doing large-scale graph analytics. Recently, there has also been an increasing interest in traversing and processing graphs using MapReduce [28].

3. MULTI-SOURCE BFS

As mentioned before, numerous graph analytics algorithms rely on executing multiple independent BFS traversals over the same graph from different sources. Often, a BFS traversal is run from every vertex in the graph. Clearly, this is very expensive, in particular for large real-world graphs that often have millions of vertices, and hence, require the execution of millions of BFSs. Our prime goal is to optimize the execution of *multiple* independent BFSs on the same graph in order to improve the performance of such graph analytics applications. We focus on a non-distributed environment—a single server—and in-memory processing to exploit the capabilities of modern multi-core servers with large memories. This is reasonable even for graphs with hundreds of billions of edges, as shown by Gupta et al. [17], and provides a better performance per dollar for workloads that process multi-gigabyte datasets [29]. Note that these are not limitations of our algorithm, which can be extended to handle disk-resident graphs.

To the best of our knowledge, Multi-Source BFS (MS-BFS) is the first approach for efficiently executing a large number of BFSs over the same graph. Most of the existing approaches for multi-core CPUs, as presented in the previous section, are orthogonal to our goal: they optimize the execution of a *single* BFS, while we optimize the execution for *multiple* BFSs. This introduces a new range of issues and requirements: (i) executing multiple traversals over the same graph exacerbates memory locality issues because the same vertices need to be discovered and explored for multiple BFSs, resulting in a higher number of cache misses; (ii) resource usage should be kept to a minimum to make the approach scalable for a large number of BFSs and as the number of cores increases; and (iii) synchronization costs of any kind should be avoided as their overheads become especially apparent when executing vast amounts of BFSs.

To address these requirements, we (i) *share* computation across *concurrent* BFSs, exploiting the properties of small-world networks; (ii) execute hundreds of BFSs *concurrently* in a *single CPU core*; and (iii) use *neither locking nor atomic operations*, which makes the execution more efficient and also improves its scalability on multi-core systems.

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 ← ∅
```

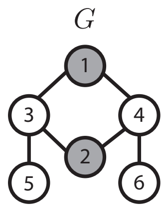
In this section, we describe in detail our novel approach. We begin by introducing the algorithm in Section 3.1. Section 3.2 then shows how this algorithm can be mapped to efficient bit operations.

3.1 The MS-BFS Algorithm

An important observation about running multiple BFSs from different sources in the same graph is that every vertex is discovered multiple times—once for every BFS if we assume the graph has a single connected component—and every time the vertex is explored, its set of neighbors must be traversed, checking if each of them has already been discovered. This leads to many random memory accesses and potentially incurs a large number of cache misses.

To decrease the amortized processing time per vertex and to reduce the number of memory accesses, we propose an approach to *concurrently* run multiple BFSs and to *share* the exploration of vertices across these BFSs by leveraging the properties of small-world networks. Recall that the diameter of such graphs is low—which means that the number of BFS levels is also small compared to the size of the graph—and that the number of discovered vertices in each level grows rapidly. Since in few steps, all the vertices of the graph are discovered, we expect the likelihood of multiple concurrent BFSs having to explore the same vertices in the same level to be high. For a concrete example of this behavior, we analyze the LDBC graph with 1 million vertices introduced in the previous section. Figure 1 depicts for every BFS level the percentage of vertex explorations that can be shared by *at least* 2, 50, 100, and 250 BFSs out of 512 concurrent BFSs as indicated by the bar height and color. Note that the exploration of more than 70% of the vertices in levels 3 and 4 can be shared among *at least* 100 BFSs, and in level 4 more than 60% of them can be shared by *at least* 250 BFSs. Concretely, this means that in level 4 it is possible to explore more than 60% of the vertices only once for 250 or more BFSs, instead of exploring them individually for each BFS. This significantly reduces the number of memory accesses and speeds up the overall processing.

We present the MS-BFS algorithm in Listing 2. In addition to the graph G, MS-BFS also receives as input the set



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

Initial State

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

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

1st BFS Level

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

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

2nd BFS Level

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

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

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

of BFSs $\mathbb{B} = \{b_1, \ldots, b_\omega\}$, where each b_i represents a BFS, and ω is the total number of BFSs to be executed. Another input is the set $S = \{s_1, \ldots, s_\omega\}$ that contains the source vertex s_i for each BFS b_i.

MS-BFS marks discovered vertices and vertices yet to be explored differently from the textbook BFS (Listing 1). Instead of having a single set *seen* of discovered vertices, each vertex v has its own set $seen_v \subseteq \mathbb{B}$ of BFSs that have already discovered it. Furthermore, the sets *visit* and *visitNext* comprise tuples $(v', \mathbb{B}')$, where the set $\mathbb{B}' \subseteq \mathbb{B}$ contains the BFSs that must explore v'.

Similar to the textbook BFS, MS-BFS initially marks the source of each BFS as discovered (Line 2), and includes in *visit* each source with its corresponding BFS identifier to indicate which vertex needs to be explored for which BFS (Line 3). Next, for each BFS level (Line 6), the algorithm repeatedly selects a vertex v to be explored (Line 7) and merges all BFS sets from *visit* that refer to v into a set $\mathbb{B}'_v$ (Line 10). Thus, $\mathbb{B}'_v$ contains all BFSs that explore v in the current level, so that v is explored only *once* for all of them *at the same time*.

This shared exploration process is shown in Lines 11–16. For each neighbor n of v, the algorithm first identifies the set $\mathbb{D}$ of all the BFSs that need to explore n in the next level (Line 12). A BFS b_i must explore n if it explores v in the current level ($b_i \in \mathbb{B}'_v$) and if it has not discovered n yet ($b_i \notin seen_n$). If $\mathbb{D}$ is not empty, there is at least one BFS that needs to explore n in the next level. Hence, we add a tuple $(n, \mathbb{D})$ to *visitNext* (Line 14). Furthermore, we mark n as discovered (Line 15) and process it in the BFS computation (Line 16) for all BFSs in $\mathbb{D}$. Note that $neighbors_v$ is traversed only *once* for all $b_i \in \mathbb{D}$, and in the next level, each vertex n will be explored only *once* for these BFSs. This significantly reduces the number of memory accesses when running a large number of BFSs. Similar to the textbook BFS, *visitNext* is used as the *visit* set for the next BFS level (Lines 17 and 18).

Figure 2 shows an example of running MS-BFS for two BFSs, b_1 and b_2, starting from vertices 1 and 2 of graph G, respectively. At the beginning of the algorithm all *seen* sets are initialized (we omit the empty ones) and *visit* contains the information that b_1 needs to explore vertex 1 and that b_2 needs to explore vertex 2. In the first BFS level, vertices 1 and 2 are explored. Since they are both adjacent to vertices 3 and 4, the *visit* set for the next level contains tuples for vertices 3 and 4 in BFSs b_1 and b_2. In the second BFS level, vertices 3 and 4 are explored. When picking vertex 3, Line 10 of the algorithm *merges* the sets of BFSs that need to explore 3 (see the highlighted section in Figure 2), resulting in $\mathbb{B}'_3 = \{b_1, b_2\}$. Therefore, 3 is explored only once for b_1 and b_2, and vertex 5 is discovered simultaneously by both BFSs as shown by the tuple $(5, \{b_1, b_2\})$ in *visit*. Similarly, vertex 4 is explored for both BFSs ($\mathbb{B}'_4 = \{b_1, b_2\}$), and vertex 6 is discovered simultaneously for both of them, adding the tuple $(6, \{b_1, b_2\})$ to the next *visit*. During the second BFS level, vertex 1 is also discovered for b_2, and vertex 2 for b_1. Since all vertices are already discovered at this point (i.e., all the *seen* sets contain both BFSs), no tuples are added to *visit* in the third level, and the algorithm finishes.

Note that this approach differs from parallelizing a single BFS since MS-BFS still discovers and explores vertices sequentially. However, with MS-BFS, multiple BFSs are executed concurrently and share the exploration of vertices.

3.2 Leveraging Bit Operations

In practice, it is inefficient to run MS-BFS using set data structures as presented in Listing 2 because the union and set difference operations on *seen* and *visitNext* are expensive when applied to large numbers of concurrent BFSs. In addition, it is prohibitively expensive for large graphs to scan *visit* every time a vertex v is picked in order to merge the BFS sets corresponding to this vertex.

To solve these issues, we leverage *bit operations* to design a more efficient version of MS-BFS. We fix the maximum number of concurrent BFSs ω to a machine-specific parameter, as elaborated in Section 4.2.1. This allows us to represent a set $\mathbb{B}' \subseteq \mathbb{B}$ of BFSs as a fixed-size bit field $f_1 \ldots f_\omega$ where $f_i = \mathtt{1}$ if $b_i \in \mathbb{B}'$, and $f_i = \mathtt{0}$ otherwise. Thus, we represent *seen* for a vertex v as $seen_v = f_1 \ldots f_\omega$, where $f_i = \mathtt{1}$ if b_i has discovered v. Furthermore, *visit* for v is represented by $visit_v = g_1 \ldots g_\omega$, where $g_i = \mathtt{1}$ if v needs to be explored by b_i. The same applies for *visitNext*.

The main advantage of representing BFS sets as bit fields is that MS-BFS can use efficient bit operations instead of complex set operations. Set unions $A \cup B$ become *binary or* operations $A \mid B$. Similarly, a set difference $A \setminus B$ becomes a *binary and* operation of A with the *negation* of B, i.e., $A \ \& \sim B$. We further denote an empty bit field as $\mathbb{B}_\varnothing$ and a bit field that only contains BFS b_i as $\mathtt{1} << b_i$.

To provide constant-time access to the bit fields of *visit*, *visitNext*, and *seen*, we store them in arrays with one bit field for each vertex in the graph. Thus, we have: $visit_v = visit[v]$, $visitNext_v = visitNext[v]$, and $seen_v = seen[v]$. In addition, we write $visit = \varnothing$ to represent the fact that $visit[v_i] = \mathbb{B}_\varnothing$ for all $i = 1, \ldots, N$.

Listing 3 presents the MS-BFS algorithm from Listing 2 using bit operations and array data structures. The overall logic of MS-BFS does not change: Lines 7, 11, 13, and 14

Listing 3: MS-BFS using bit operations.

```
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             𝔻 ← visit[v_i] & ∼seen[n]
12             if 𝔻 ≠ 𝔹_∅
13                 visitNext[n] ← visitNext[n] | 𝔻
14                 seen[n] ← seen[n] | 𝔻
15                 do BFS computation on n
16     visit ← visitNext
17     reset visitNext
```

from Listing 3 are equivalent to Lines 6, 12, 14, and 15 from Listing 2, respectively. Note that it is a significant improvement to use bit fields to represent BFS sets and arrays for *visit* because it avoids the expensive merging loop of Lines 9 and 10 from Listing 2.

We assume that the *neighbors* adjacency list is implemented as a single array that contains all the neighbors for all vertices, and that *neighbors*$[v_i]$ points to the memory block in *neighbors* that encompasses the neighbors of v_i. Also, these memory blocks are stored in order, i.e., *neighbors*$[v_{i-1}]$ precedes *neighbors*$[v_i]$ for all $i = 2, \ldots, N$. This representation improves memory locality for the algorithm: vertices are explored in order (Line 8), and as a consequence, the *neighbors* array can be retrieved in order as well (Line 10), which maximizes opportunities for sequential reads and makes better use of caching [18].

Figure 3 shows the example presented in Figure 2 using arrays of bit fields for *visit* and *seen*. As in the previous figure, the *visitNext* array is similar to the *visit* array of the next BFS level and is omitted for clarity. Each row represents the bit field for a vertex, and each column corresponds to one BFS. The symbol X indicates that the bit value is `1`; otherwise, the value is `0`.

By processing the *visit* array in the first BFS level, vertices 3 and 4 are discovered for both BFSs, since both of them are neighbors of the sources 1 and 2. Hence, *seen*[3] and *seen*[4] have a bit field of value `11`, indicating that both BFSs have discovered these vertices. The bit fields *visit*[3] and *visit*[4] have this value as well, indicating that these vertices need to be explored for both BFSs in the next level. During the second BFS level, vertices 3 and 4 are explored only once for both b_1 and b_2 (since *visit*[3] = *visit*[4] = `11` at the end of the first level), leading to the discovery of vertices 5 and 6 for both BFSs. As the *seen* array does not contain bits of value `0` anymore, no new vertices are discovered in the third BFS level.

Note that our algorithm can leverage efficient native bit operations, in particular when ω is a multiple of the machine's register width. Furthermore, because we process concurrent BFSs in a single CPU core, we do not need to synchronize memory accesses between them. This allows our approach to scale nearly linear when multiple cores are used for MS-BFS runs. We elaborate on both these points in Section 4.2.1.

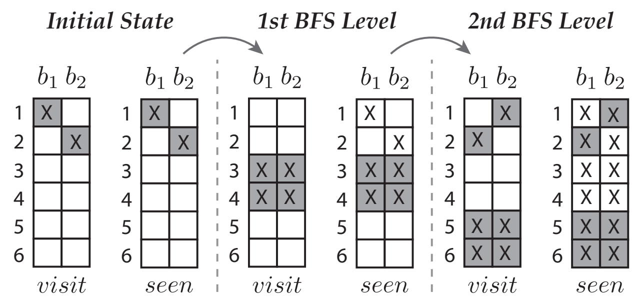


Figure 3: An example showing the steps of MS-BFS when using bit operations. Each row represents the bit field for a vertex, and each column corresponds to one BFS. The symbol X indicates that the value of the bit is `1`.

4. ALGORITHM TUNING

In this section, we discuss techniques to further improve the performance of MS-BFS, including techniques to improve the algorithm's memory locality and to avoid—even more—the impact of random memory accesses (Section 4.1). Furthermore, we describe efficient strategies to execute a number of BFSs greater than the size ω of the used bit fields (Section 4.2). We evaluate the impact of the presented techniques in Section 6.2.

4.1 Memory Access Tuning

4.1.1 Aggregated Neighbor Processing

As mentioned earlier, sequentially checking the elements in the *visit* array in Listing 3 improves the memory locality of MS-BFS and results in fewer caches misses. However, there are still random accesses to the *visitNext* and *seen* arrays (Lines 11, 13, and 14) as the same neighbor vertex n can be discovered by different vertices and BFSs in the same level, i.e., *visitNext*$[n]$ and *seen*$[n]$ may be accessed in different iterations of the loop in Line 8. In addition, the application-specific BFS computation for n (Line 15) may have to be executed multiple times as well, which worsens the issue.

To provide further improvements in memory locality, we propose the *aggregated neighbor processing* (ANP) technique. The main idea of ANP is to reduce the number of both BFS computation calls and random memory accesses to *seen* by first collecting all the vertices that need to be explored in the next BFS level, and then processing in batch the remainder of the algorithm. This removes the dependency between *visit* and both *seen* and the BFS computation.

Listing 4 shows the MS-BFS algorithm using ANP. Concretely, when using ANP, we process a BFS level in two stages. In the first stage (Lines 8–11), we sequentially explore all vertices in *visit* to determine in which BFSs their neighbors should be visited and write this information to *visitNext*. In the second stage (Lines 13–18), we sequentially iterate over *visitNext*, update its bit fields based on *seen*, and execute the BFS computation. Note that we only perform these steps *once* for every newly discovered vertex, and thus we *aggregate* the neighbor processing.

ANP leverages the fact that *visitNext* is reset for every new BFS level. In addition, Lines 11 and 15 in Listing 4 are equivalent to Lines 11 and 13 in Listing 3 by means of

Listing 4: MS-BFS algorithm using ANP.

1 **Input:** $G, \mathbb{B}, S$
2 **for each** $b_i \in \mathbb{B}$
3     $\mathit{seen}[\ s_i\ ] \leftarrow \mathtt{1} << b_i$
4     $\mathit{visit}[\ s_i\ ] \leftarrow \mathtt{1} << b_i$
5 reset *visitNext*
6
7 **while** $\mathit{visit} \neq \varnothing$
8   **for** $i = 1, \ldots, N$
9   **if** $\mathit{visit}[v_i] = \mathbb{B}_\varnothing$, **skip**
10   **for each** $n \in \mathit{neighbors}[v_i]$
11             $\mathit{visitNext}[n] \leftarrow \mathit{visitNext}[n] \mid \mathit{visit}[v_i]$
12
13   **for** $i = 1, \ldots, N$
14   **if** $\mathit{visitNext}[v_i] = \mathbb{B}_\varnothing$, **skip**
15         $\mathit{visitNext}[v_i] \leftarrow \mathit{visitNext}[v_i]\ \&\ {\sim}\mathit{seen}[v_i]$
16         $\mathit{seen}[v_i] \leftarrow \mathit{seen}[v_i] \mid \mathit{visitNext}[v_i]$
17   **if** $\mathit{visitNext}[v_i] \neq \mathbb{B}_\varnothing$
18   do BFS computation on v_i
19     $\mathit{visit} \leftarrow \mathit{visitNext}$
20   reset *visitNext*

the distributive property of binary operations; for a vertex n and vertices $v_1, \ldots, v_k$ of which n is a neighbor:

$$\Big(\mathit{visit}[v_1] \mid \ldots \mid \mathit{visit}[v_k]\Big)\ \&\ {\sim}\mathit{seen}[n] \equiv$$
$$\Big(\mathit{visit}[v_1]\ \&\ {\sim}\mathit{seen}[n]\Big) \mid \ldots \mid \Big(\mathit{visit}[v_k]\ \&\ {\sim}\mathit{seen}[n]\Big)$$

Note that the random memory accesses to *visitNext* in Line 11 are inevitable. We discuss how to mitigate this issue in Section 4.1.3. Nevertheless, ANP brings a range of advantages. Notably, it: (i) reduces the number of memory accesses to *seen*, since the array is only retrieved once for every discovered vertex v, independent of the number of vertices of which v is a neighbor; (ii) replaces random access with sequential access to *seen*, which improves memory locality; and (iii) reduces the number of times that the BFS computation is executed. With these advantages, ANP improves the usage of low cache levels, prevents stalls caused by cache misses, and thus, reduces the overall execution time of MS-BFS. As reported in Section 6.2, ANP speeds up MS-BFS by 60 to 110%.

4.1.2 Direction-Optimized Traversal

As we focus on small-world graphs, it is further beneficial to apply the *direction-optimized* BFS technique, introduced by Beamer et al. [7], to MS-BFS. The technique chooses at runtime between two BFS strategies: *top-down* and *bottom-up*. The former strategy is a conventional BFS, discovering new vertices by exploring the ones found in the previous level, i.e., by exploring the *visit* array in Lines 8–10 in Listing 3. In contrast, the bottom-up approach, when applied to MS-BFS, avoids traversing the *visit* array, and instead, scans the *seen* array for vertices that have not yet been discovered by all BFSs. When such a vertex v is found, the approach traverses its edges and processes the *visit* entries of the neighbor vertices n that are adjacent to v:

$$\mathit{visitNext}[v] \leftarrow \mathit{visitNext}[v] \mid \mathit{visit}[n]$$

Note that, as suggested by the technique name, the two strategies work in different directions: the top-down one goes from discovered to non-discovered vertices, while the bottom-up one goes in the opposite direction. Direction-optimized traversal uses a heuristic based on the number of non-traversed edges during the BFS and a threshold to perform either the top-down or the bottom-up strategy. Concretely, the heuristic often chooses the top-down approach for the initial BFS levels, and the bottom-up approach for the final steps, where most of the vertices have already been discovered. The reader is referred to Beamer et al. [7] for further details.

Our experiments (Section 6.2) show that with both this hybrid approach and ANP, MS-BFS can significantly reduce the number of random accesses to *visit* and *visitNext*, further improving the performance by up to 30%.

4.1.3 Neighbor Prefetching

Recall that the ANP technique reduces the number of random accesses to the *seen* array. However, many random accesses are still unavoidable when updating *visitNext* (Line 11 in Listing 4).

To mitigate the high latency of these memory accesses, it is beneficial to use *prefetching*: once the vertex v_i is picked from *visit*, we detect its neighbors and the memory addresses of their entries in *visitNext*. We can then explicitly *prefetch* some of these entries *before* computing *visitNext* for them. As a consequence, the iteration in Line 10 is less prone to execution stalls because the prefetched *visitNext* entries are likely to be in the CPU cache when they are required, which provides an additional speedup to MS-BFS. Instead of doing the prefetching interleaved with the algorithm execution, it is also beneficial to do it asynchronously in simultaneous multithreading cores [22]. We identified experimentally that by prefetching tens or even hundreds of neighbors the performance of MS-BFS can be improved by up to 25%, as elaborated in Section 6.2.

4.2 Execution Strategies

4.2.1 How Many BFSs?

Conceptually, the MS-BFS algorithm can be used to run any number of concurrent BFSs by using bit fields of arbitrary sizes. However, our approach is more efficient when the bit operations are implemented using native machine instructions. Thus, to achieve optimal performance, ω should be set according to the register and instruction width of the used CPU. As an example, on modern Intel CPUs, there are instructions and registers with a width of up to 256 bits, thus allowing MS-BFS to efficiently execute 256 concurrent BFSs; in CPUs supporting the AVX-512 extension, there are instructions that operate on 512 bits, which doubles the number of concurrent BFSs that can be executed using a single CPU register per vertex and data structure.

Nevertheless, it is often the case that applications need to run BFSs for a number of sources greater than the size of any CPU-optimized ω. In this case, there are three different strategies that can be used: (1) increase ω by using multiple CPU registers for the bit fields, (2) execute multiple MS-BFS runs in parallel, and (3) execute multiple MS-BFS runs sequentially. In the first approach, multiple CPU registers are used to represent the bit fields in *seen*, *visit*, and *visitNext*, i.e., ω is set to a multiple of the register width. As an example, we can leverage two 128-bit registers to have 256-bit fields, which, in turn, enable us to run 256 BFSs concurrently. The main advantage of this approach is that,

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

clearly, the graph needs to be traversed less often as more BFSs are executed simultaneously, thus allowing additional sharing of vertex explorations. Moreover, when these registers are stored adjacent in memory, they can be aligned to cache line boundaries so that accessing part of the bit field ensures that its other parts are in the CPU cache as well. Hence, we further reduce the number of main memory accesses during a MS-BFS run. In Section 6.2, we show that having bit fields that are exactly sized to fit a cache line exhibits the best performance. On current Intel CPUs, cache lines are 64 bytes wide, which allows efficient processing of 512 sources per MS-BFS run.

A second approach to execute a larger number of BFSs is to make use of parallelism. While the presented MS-BFS algorithm runs in a single core, multiple cores can be used to execute multiple MS-BFS runs in parallel since these runs are independent from each other. As a result, MS-BFS scales almost linearly with an increasing number of cores.

The drawback of the first two approaches is their potentially high memory consumption. For P parallel runs and N vertices, MS-BFS requires $P \times 3 \times N \times \omega$ bits of memory to store the fields for *seen*, *visit* and *visitNext*. Table 2 gives some examples of the total memory consumption for different graph sizes and numbers of parallel runs, running from hundreds to tens of thousands of BFSs concurrently.

Last, it is possible to execute multiple MS-BFS runs sequentially, since, again, runs are independent. This is particularly interesting when memory becomes an issue. Based on the available memory and to adapt to different situations, we can choose the best strategy and *combine* the three approaches. For instance, multiple threads, each using bit fields that are several registers wide, can be used to execute sequences of MS-BFS runs.

4.2.2 Heuristic for Maximum Sharing

When executing a number of BFSs greater than ω, it is useful to group BFSs in the same MS-BFS run, i.e., in the same set $\mathbb{B}$, that will share most computations at each level. Recall that the main idea of MS-BFS is to share vertex explorations across concurrent BFSs. As a consequence, the more BFSs explore the same vertex v in a given level, the less often v will have to be explored again in the same run, and the faster MS-BFS becomes.

The first clear approach to allow sharing of vertex explorations is to group BFSs based on their connected components: BFSs running in the same component should be in the same MS-BFS run, as different components do not share any vertices or edges. To optimize the sharing in a single connected component, we propose a heuristic to group BFSs based on ordering their corresponding source vertices *by degree*. Recall that small-world networks are often scale-free as well, which means that there are few vertices with a high degree, while most of the vertices have a significantly smaller number of neighbors. Based on this property and the fact that small-world networks have a low diameter, our intuition is that vertices with higher degrees will have a significant number of neighbors in common. Therefore, this heuristic comprises sorting the source vertices by descending degree, and then grouping BFSs according to this order to improve the sharing of vertex explorations. We expect that a MS-BFS run starting from the highest degree vertices will have a very efficient execution. In fact, our evaluation in Section 6.2 shows that, compared with a random assignment of BFSs to MS-BFS runs, this heuristic can improve the performance by up to 20%.

5. APPLICATION: COMPUTING CLOSENESS CENTRALITY

Since our approach executes multiple concurrent BFSs, MS-BFS must efficiently handle the application-specific computation for multiple BFSs (Line 15 in Listing 3). As it is common for graph algorithms to determine the number of neighbors found in a BFS level, we elaborate on how this can be done in MS-BFS. Based on this, we describe how MS-BFS can be used to solve the graph analytics problem of *all-vertices closeness centrality*, which comprises calculating the closeness centrality value for all vertices in a graph.

Closeness Centrality. Computing vertex centrality metrics is an important application in graph analytics. Centrality metrics can be used, for instance, to gain knowledge about how central persons are distributed in a social network, which can, in turn, be used for marketing purposes or research of the network structure. In the literature, many centrality metrics have been proposed, including closeness centrality [27] and betweenness centrality [9]. For the purpose of this discussion, we focus on the former.

The closeness centrality value of a vertex v measures how close v is, in terms of shortest path, from all other vertices in the graph. Essentially, it is based on the inverse of the sum of the distances between v and all other vertices. From Wasserman and Faust [34]:

$$\mathit{ClosenessCentrality}_v = \frac{(C_v - 1)^2}{(N-1) * \Sigma_{u \in V} d(v,u)}$$

where C_v is the number of vertices in the connect component of v, and $d(v,u)$ denotes the geodesic distance (i.e., the length of the shortest path) between v and u. To compute $d(v,u)$ for all $u \in V$ in an unweighted graph, a BFS-based traversal from v is required to calculate and maintain the geodesic distance in the BFS computation. For the *all-vertices closeness centrality* problem, a BFS must be run for every vertex in the graph, which makes this computationally expensive. We use this problem as an example of the applicability of MS-BFS in a real-world application.

Using MS-BFS. We can leverage MS-BFS to efficiently solve the all-vertices closeness centrality problem by running hundreds of concurrent BFSs per core to calculate the required geodesic distances. Note that the algorithm does not need to store each distance $d(v,u)$ since it suffices to find the sum of all distances computed during the BFSs.

An efficient way to find this sum is to count the number of discovered vertices for each BFS level, multiply this number by the current distance from the source, and then finally sum all obtained multiplication results. Note that the number of discovered vertices may be different for each concurrent BFS, so they need to be counted separately. For every discovered vertex v, each BFS can determine whether this vertex belongs to it by detecting if the bit field $visitNext[v]$ contains a bit of value `1` for it. If so, a BFS-local counter is incremented. This naive approach takes $O(\omega)$ time for each discovered vertex in the BFS level.

An alternative is to use hardware operations that count the number of leading or trailing bits `0` in a bit field: we continuously find the position of a bit `1` using such an operation, increase the respective BFS-local counter, and set the bit to `0`, until there are no more bits `1` in the bit field. This approach takes $O(m)$ time, where $m \leq \omega$ is the number of BFSs that discovered the vertex in that level. However, this is still inefficient when most of the bits in a field have value `1`, and also because this operation is sensitive to branch mispredictions, since the CPU cannot predict well whether there will be another iteration.

In order to provide a more efficient solution, we designed an amortized $O(1)$ algorithm that efficiently updates the BFS-local neighbor counters in a branch-free manner. Our general idea is to use a space-efficient 1 byte wide neighbor counter for every BFS. We update these counters every time the BFS computation is executed and copy their information to wider fields once they overflow. The main difference to the previous approaches is that we update the 1-byte counters using SIMD instructions available in modern CPUs: by masking the $visitNext[v]$ bit field, we increment every counter belonging to a BFS that discovered v in the current level and leave other counters unmodified. Note that, since the runtime of this BFS computation step is independent from the number of BFSs that discover v, it optimally leverages the benefits of ANP as presented in Section 4.1.1, which reduces the number of executions of the BFS computation. Due to the lack of space, we omit further details of our BFS-computation approach.

6. EXPERIMENTAL EVALUATION

To assess the efficiency of our approach and the presented optimizations, we studied the performance of MS-BFS using both synthetic and real datasets. Notably, we report experiments on scalability with respect to input size, number of cores and number of BFSs (source vertices), and discuss the impact of the tuning techniques introduced in Section 4. Also, we compare the performance of MS-BFS with the textbook BFS and a state-of-the-art BFS algorithm.

6.1 Experimental Setup

BFS Algorithms. In our experimental evaluation, we used a number of different BFS implementations for comparison purposes: (i) MS-BFS for the CPU register widths of 64, 128, and 256 bits; (ii) a non-parallel version of the Direction-Optimized BFS (DO-BFS) [7], a state-of-the-art BFS algorithm; and (iii) Textbook BFS (T-BFS) as shown in Listing 1.

For each MS-BFS variant, we evaluated the performance when using a single register as well as when using multiple registers for a single bit field to fill an entire cache line.

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

We indicate the latter using the suffix *CL* and follow the approach explained in Section 4.2.1. We also enabled all other optimizations from Section 4 in our experiments with MS-BFS unless otherwise noted.

We performed the comparisons by using each of the algorithms to compute all-vertices closeness centrality, which, as described before, is a computationally expensive graph analytics problem that uses BFS-based graph traversal. It is worth mentioning that the overheads for computing the closeness centrality values were similar among the BFS algorithms in order to provide a fair comparison.

Further Algorithms. Note that we do not compare our approach with parallel BFS implementations, since they are not optimized for the efficient execution of a large number of BFSs. To understand this behavior, we implemented and experimented with the single-socket version of the parallel BFS introduced by Agarwal et al. [2], where the authors propose a parallel BFS implementation that uses a bitmap for the *seen* data structure, as well as efficient atomic operations to amortize the synchronization costs of the level-synchronous parallelization. When varying the number of cores, running single-threaded T-BFSs or DO-BFSs showed a significantly better BFS throughput than sequentially executing the same number of parallel BFSs to solve the all-vertices closeness centrality problem. The main reason for these results is that, although highly optimized, the synchronization costs of parallel BFSs hamper good scalability for running a large number of traversals. Due to the lack of space, we do not elaborate on this experiment.

We also experimented with the well-known open-source graph database Neo4j [26]. We used their integrated closeness centrality computation function and benchmarked its hot-cache runtime. On a graph with 50,000 vertices and 1.5 million edges from the LDBC generator, the all-vertices closeness centrality computation took 23 hours. As Neo4j was running in a single CPU core, and assuming perfect scalability, we estimated that its parallelized runtime on our evaluation machine would be 23 minutes, which is still more than two orders of magnitude slower than our T-BFS implementation. Compared to MS-BFS, it is nearly four orders of magnitude slower. Therefore, we do not include Neo4j in our comparison.

Software and Hardware Configuration. We ran most of our experiments on a 4-socket server machine with Intel Xeon E7-4870v2 CPUs, which has a total of 60 cores at 2.30 GHz and with a Turbo Boost frequency of 2.90 GHz. The server is equipped with 1 TB of main memory equally divided among four NUMA regions. The experiments for

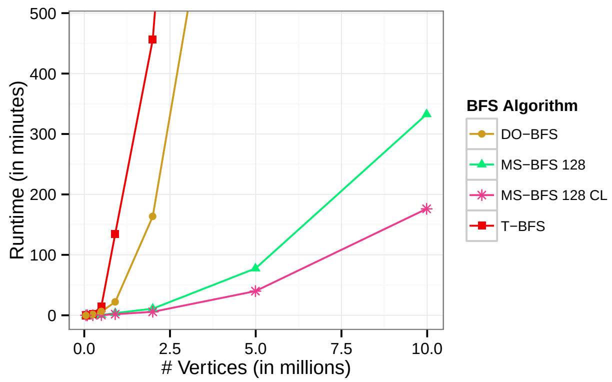


Figure 4: Data size scalability results.

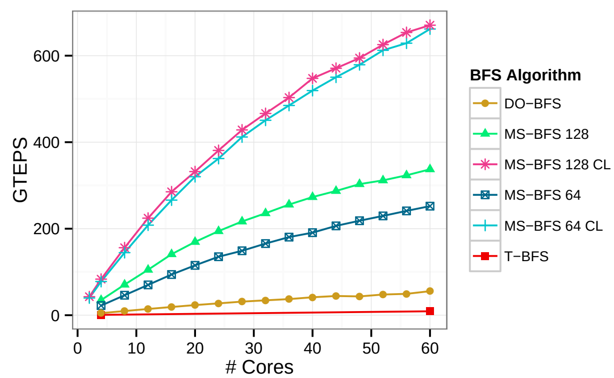


Figure 5: Multi-core scalability results.

Figures 6 and 7 were run on a machine equipped with an Intel Core i7-4770K CPU at 3.50 GHz with a Turbo Boost frequency of 3.9 GHz; we use this CPU as it supports the AVX-2 instruction set for bit operations on 256-bit wide registers. All algorithms were implemented in C++ 11, compiled with GCC 4.8.2, and executed on Ubuntu 14.04 with kernel version 3.13.0-32.

Datasets. In our evaluation, we experimented with both synthetic and real datasets. For the former, we used the LDBC Social Network Data Generator [8, 21], which was designed to produce graphs with properties similar to real-world social networks. With this generator, we created synthetic graphs of various sizes—from 50,000 to 10 million vertices, and with up to 288 million edges. Additionally, we evaluated the performance of MS-BFS using two real-world datasets from Twitter and Wikipedia. The Twitter dataset[2] contains 2.4 billion edges following the relationships of about 41 million users, while the Wikipedia dataset[3] represents a snapshot of the data as of July 2014, consisting of articles and links connecting them. Note that we considered the edges in all datasets as undirected. Table 3 shows the properties of the graphs that we used in our evaluation, including their number of vertices and edges, diameter, and the memory consumption in our graph data structure.

6.2 Experimental Results

Data Size Scalability. To understand how MS-BFS scales as the size of the graph increases, we measured its runtime for different synthetic datasets, containing up to 10 million vertices and 288 million edges. Figure 4 shows the scalability of the BFS algorithms for all LDBC datasets we introduced before. The runtimes are measured in minutes and do not include loading times. The algorithms were executed using 60 cores, i.e., multiple runs of each algorithm were executed in parallel using all the cores available in the machine.

From the results, it is clear that T-BFS and DO-BFS do not scale well as the data size increases when running multiple BFSs. As an example, T-BFS and DO-BFS took 135 minutes and 22 minutes, respectively, to process the LDBC graph with 1 million vertices, while MS-BFS took only 1.75 minutes.

MS-BFS showed excellent scalability for all the presented graphs. In fact, it makes time-consuming computations feasible to run on a single machine, which is not possible to achieve using traditional approaches: computations that formerly took hours are sped up to minutes.

Our experiments show the benefits of sharing computation among multiple concurrent BFSs. Even for large graphs, MS-BFS has a good runtime as a significant amount of vertex explorations is shared. Also, our use of bit operations provides very efficient concurrent BFS execution in a single core. MS-BFS runs many concurrent BFSs, while T-BFS and DO-BFS can only perform one traversal per execution. In Figure 4, we also show that using an entire cache line for bit fields in MS-BFS significantly improves the performance of the algorithm. Our evaluation machine uses 512 bit wide cache lines, which we filled using the data from 4 128-bit registers, thus executing up to 512 concurrent BFSs in a single core.

Multi-Core Scalability. We studied the scalability of MS-BFS, T-BFS, and DO-BFS with an increasing number of CPU cores. Instead of showing the execution runtime, we measured the performance in *traversed edges per second* (TEPS), i.e., the total number of edges traversed by the algorithm divided by the runtime [1], when running all-vertices closeness centrality for LDBC 1M. We report the results in GTEPS, i.e., billion TEPS.

Figure 5 depicts the nearly linear scalability of MS-BFS: by keeping the resource usage low for a large number of concurrent BFSs, the approach can execute more traversals as the number of cores increases. Notably, by using 128-bit registers and the entire cache line, MS-BFS could reach 670 GTEPS using 60 cores. T-BFS and DO-BFS showed a significantly lower performance.

MS-BFS did not exhibit an exact linear scale-up due to the activated Turbo Boost functionality in recent Intel CPUs. Turbo Boost increases the clock frequency of the CPUs when fewer cores are used, i.e., the more cores used, the lower the clock rate. We chose not to disable this feature to show how the algorithm would behave under real-world conditions.

BFS Count Scalability. The main goal of MS-BFS is to execute a large number of BFSs efficiently. To evaluate this property, we studied the scalability of MS-BFS as the number of BFSs increases. In Figure 6, we show the scalability from 1 to 2,000 closeness centrality computations. Again, we used the LDBC 1M dataset and report the results in GTEPS. In contrast to the previous experiment, we only used a single CPU core in order to make the results clearer. DO-BFS is omitted in this experiment as it showed the same

[2] http://konect.uni-koblenz.de/networks/twitter
[3] http://dumps.wikimedia.org/enwiki/20140707

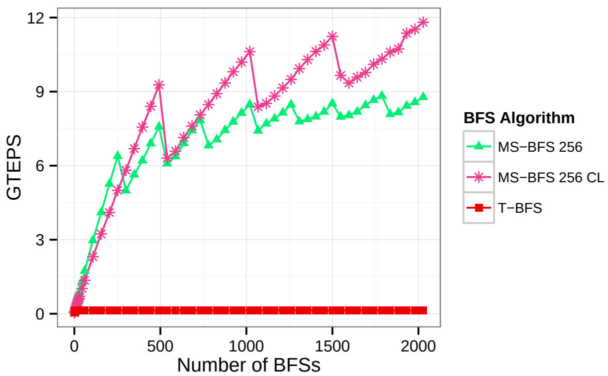


Figure 6: BFS count scalability results.

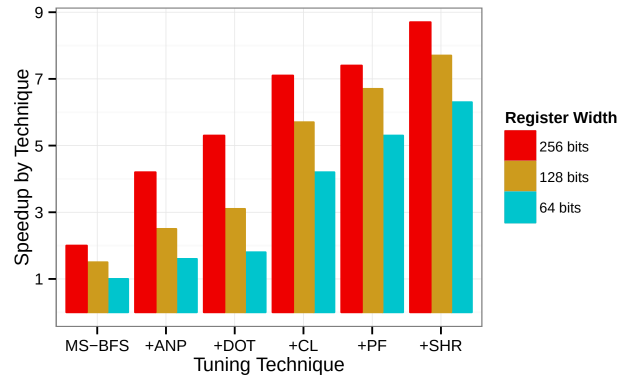


Figure 7: Speedup achieved by cumulatively applying different tuning techniques to MS-BFS.

behavior as T-BFS. Furthermore, we note that we used the same source vertices when comparing the algorithms.

For T-BFS, the number of traversed edges per second was constant for any number of BFSs, which is the expected result for an algorithm that runs BFSs sequentially. For MS-BFS, we saw a different behavior: as more BFSs were executed, the GTEPS increased, since multiple BFSs could run concurrently in a single core. The peaks in the performance correspond to when the number of BFSs is a multiple of the bit field width of the MS-BFS run: 256 for MS-BFS 256, and 512 for MS-BFS 256 CL. The performance decays are related to the sequential execution of multiple MS-BFS runs as the bit fields became entirely filled. Nevertheless, because of the source vertex re-ordering (Section 4.2.2), the performance kept increasing as more BFSs were executed, which shows that MS-BFS provides good scalability with respect to the number of traversals.

Speedup. Table 4 shows the speedup of MS-BFS compared to T-BFS and DO-BFS when running all-vertices closeness centrality for two synthetic datasets, as well as for the Wikipedia and Twitter datasets. In the Twitter dataset, we randomly selected 1 million vertices and computed the closeness centrality values for only these vertices. In these experiments, 60 cores were used. Some runs, indicated by an asterisk, were aborted after executing for more than eight hours; the runtimes were then estimated by extrapolating the obtained results. We can see that MS-BFS outperformed both T-BFS and DO-BFS by up to almost 2 orders of magnitude, between 12.1x and 88.5x.

Impact of Algorithm Tuning. To analyze the performance gains obtained by using each tuning technique described in Section 4, we evaluated their impacts by means of speedup. As the baseline, we used the MS-BFS algorithm as described in Section 3.2 using 64-bit registers. We then varied the register size and the techniques applied to the algorithm *cumulatively* and in the following order: aggregated neighbor processing (ANP), direction-optimized traversal (DOT), use of entire cache lines of 512 bits (CL), neighbor prefetching (PF) and heuristic for maximum sharing (SHR). The results are shown in Figure 7.

Using wider registers was beneficial for all optimizations, as more BFSs could be run concurrently. From the figure, we can see that CL provided the most significant speedup. ANP also showed a substantial speedup, in particular when using wide registers, which demonstrates the impact of improving the memory locality for graph applications. Prefetching (PF) only showed noticeable speedup for smaller register sizes; it exhibited nearly no improvement when applied to MS-BFS using wide registers. Together, the tuning techniques improved the overall performance of MS-BFS by more than a factor of 8 over the baseline.

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.

7. CONCLUSION

In this paper, we addressed the challenge of efficiently running a large number of BFS-based graph traversals in graph analytics applications. By leveraging the properties of small-world networks, we proposed MS-BFS, an algorithm that can run multiple independent BFSs concurrently in a single core. MS-BFS reduces the number of random memory accesses, amortizes the high cost of cache misses, and takes advantage of wide registers as well as efficient bit operations in modern CPUs. We demonstrated how MS-BFS can be used to improve the performance of solving the all-vertices closeness centrality problem, and we are confident that the principles behind our algorithm can significantly help speed up a wide variety of other graph analytics algorithms as well. Our experiments reveal that MS-BFS outperforms state-of-the-art algorithms for running a large number of BFSs, and that our approach, combined with the proposed tuning techniques, provides excellent scalability with respect to data size, number of available CPUs, and number of BFSs.

There are numerous interesting directions for future work, remarkably: combining our approach with existing parallel BFS algorithms; adapting MS-BFS for distributed environments and GPUs; analyzing how MS-BFS can be applied to other analytics algorithms; assessing its behavior on different types of graphs; designing new heuristics to maximize the sharing among BFSs; and using MS-BFS in query optimizers to improve the throughput of graph databases. Furthermore, we plan to integrate MS-BFS with the graph analytics capabilities of HyPer [19].

8. ACKNOWLEDGMENTS

We thank the organizers of the SIGMOD 2014 Programming Contest for running the competition. The TUM team was sponsored in part by the EU FP7 project LDBC and the German Federal Ministry of Education and Research (grant HDBC 01IS12026). The NYU team was supported in part by NSF (award CNS-1229185), the Sloan Foundation, and DARPA under agreement number FA8750-14-2-0236.

9. REFERENCES

[1] Graph 500 Benchmark, 2014. `http://www.graph500.org/`.

[2] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable Graph Exploration on Multicore Processors. In *SC '10*, pages 1–11, 2010.

[3] L. A. N. Amaral, A. Scala, M. Barthélémy, and H. E. Stanley. Classes of Small-World Networks. *PNAS*, 97(21):11149–11152, 2000.

[4] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. Four Degrees of Separation. In *WebSci '12*, pages 33–42, 2012.

[5] D. A. Bader and K. Madduri. Designing Multithreaded Algorithms for Breadth-First Search and St-connectivity on the Cray MTA-2. In *ICPP '06*, pages 523–530, 2006.

[6] D. A. Bader and K. Madduri. SNAP, Small-world Network Analysis and Partitioning: An Open-source Parallel Graph Framework for the Exploration of Large-scale Networks. In *IPDPS*, pages 1–12, 2008.

[7] S. Beamer, K. Asanović, and D. Patterson. Direction-Optimizing Breadth-First Search. In *SC '12*, pages 12:1–12:10, 2012.

[8] P. Boncz. LDBC: Benchmarks for Graph and RDF Data Management. In *IDEAS '13*, pages 1–2, 2013.

[9] U. Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[10] A. Buluç and K. Madduri. Parallel Breadth-First Search on Distributed Memory Systems. In *SC '11*, pages 65:1–65:12, 2011.

[11] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal. Breaking the Speed and Scalability Barriers for Graph Exploration on Distributed-Memory Machines. In *SC '12*, pages 13:1–13:12, 2012.

[12] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu. K-Reach: Who is in Your Small World. *PVLDB*, 5(11):1292–1303, July 2012.

[13] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

[14] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In *IPDPS '12*, pages 378–389, May 2012.

[15] Faunus – Graph Analytics Engine, 2014. `http://thinkaurelius.github.io/faunus/`.

[16] D. Gregor and A. Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In *POOSC*, 2005.

[17] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. WTF: The Who to Follow Service at Twitter. In *WWW '13*, pages 505–514, 2013.

[18] S. Hong, T. Oguntebi, and K. Olukotun. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *PACT '11*, pages 78–88, Oct 2011.

[19] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE'11*, pages 195–206, 2011.

[20] A. Landherr, B. Friedl, and J. Heidemann. A Critical Review of Centrality Measures in Social Networks. *Business & Information Systems Engineering*, 2(6):371–385, 2010.

[21] LDBC Social Network Data Generator, 2014. `https://github.com/ldbc/ldbc_snb_datagen`.

[22] J. Lee, C. Jung, D. Lim, and Y. Solihin. Prefetching with Helper Threads for Loosely Coupled Multiprocessor Systems. *TPDS*, 20(9):1309–1324, 2009.

[23] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 5(8):716–727, April 2012.

[24] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD '10*, pages 135–146, 2010.

[26] Neo4j, 2014. `http://neo4j.com/`.

[27] P. Olsen, A. Labouseur, and J.-H. Hwang. Efficient Top-K Closeness Centrality Search. In *ICDE '14*, pages 196–207, March 2014.

[28] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin. Scalable Big Graph Processing in MapReduce. In *SIGMOD '14*, pages 827–838, 2014.

[29] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody Ever Got Fired for Using Hadoop on a Cluster. In *HotCDP '12*, pages 2:1–2:5, 2012.

[30] N. Satish, C. Kim, J. Chhugani, and P. Dubey. Large-Scale Energy-efficient Graph Traversal: A Path to Efficient Data-Intensive Supercomputing. In *SC '12*, pages 14:1–14:11, 2012.

[31] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD '13*, pages 505–516, 2013.

[32] D. Simmen, K. Schnaitter, J. Davis, Y. He, S. Lohariwala, A. Mysore, V. Shenoi, M. Tan, and X. Yu. Large-Scale Graph Analytics in Aster 6: Bringing Context to Big Data Discovery. *PVLDB*, 7(13):1405–1416, August 2014.

[33] Titan – Distributed Graph Database, 2014. `http://thinkaurelius.github.io/titan/`.

[34] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*, volume 8. Cambridge University Press, 1994.