iBFS: Concurrent Breadth-First Search on GPUs

Hang Liu   H. Howie Huang   Yang Hu
Department of Electrical and Computer Engineering
George Washington University
{asherliu, howie, huyang}@gwu.edu

ABSTRACT

Breadth-First Search (BFS) is a key graph algorithm with many important applications. In this work, we focus on a special class of graph traversal algorithm - concurrent BFS - where multiple breadth-first traversals are performed simultaneously on the same graph. We have designed and developed a new approach called *iBFS* that is able to run i concurrent BFSes from i distinct source vertices, very efficiently on Graphics Processing Units (GPUs). iBFS consists of three novel designs. First, iBFS develops a single GPU kernel for *joint traversal* of concurrent BFS to take advantage of shared frontiers across different instances. Second, *outdegree-based GroupBy rules* enables iBFS to selectively run a group of BFS instances which further maximizes the frontier sharing within such a group. Third, iBFS brings additional performance benefit by utilizing highly optimized *bitwise operations* on GPUs, which allows a single GPU thread to inspect a vertex for concurrent BFS instances. The evaluation on a wide spectrum of graph benchmarks shows that iBFS on one GPU runs up to 30× faster than executing BFS instances sequentially, and on 112 GPUs achieves near linear speedup with the maximum performance of 57,267 billion traversed edges per second (TEPS).

1. INTRODUCTION

Graph-based representations are ubiquitous in many applications such as social networks [1–3], metabolic networks [4], and computer networking [5]. As a result, graph algorithms have been an vital research topic in the era of big data, among which Breadth-First Search (BFS) draws a significant amount of interests due to its importance.

In this work, we focus on a special class of BFS algorithm - concurrent BFS - where multiple breadth-first traversals are performed simultaneously on the same graph. We refer to our solution to this problem as iBFS that is able to perform i multiple breadth-first traversals in parallel on GPUs, each from a distinct source vertex. Here i is between 1 and $|V|$, i.e., the total number of vertices in the graph.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2882959

Depending on the value of i, iBFS actually becomes a number of different problems. Formally, in a graph with $|V|$ vertices, iBFS is

- single source shortest path (SSSP) *if* $i = 1$ [6];
- multi-source shortest path (MSSP) *if* $i \in (1, |V|)$ [7,8];
- all-pairs shortest path (APSP) *if* $i = |V|$ [9, 10].

Moreover, iBFS can be utilized in many other graph algorithms such as betweenness centrality [11, 12] and closeness centrality [13]. For example, one can leverage iBFS to construct the index for answering graph reachability queries, that is, whether there exists a path from vertex s to t with the number of edges in-between less than k [14–16]. We will show in this paper this step can be an order of magnitude faster with iBFS. In all, a wide variety of applications, e.g., network routing [17–19], network attack detection [20], route planning [21–23], web crawling [24,25], etc., can benefit from high-performance iBFS.

This work proposes a new approach for running iBFS on GPUs that consists of three novel techniques: joint traversal, GroupBy, and bitwise optimization. Prior work has proposed to combine the execution of different BFS instances mostly on multi-core CPUs [26–28]. The performance improvement, however, is limited. For one, none of early projects has attempted to group the BFS instances to improve the frontier sharing during the traversal. Further, bottom-up BFS provides new additional challenges. For example, while MS-BFS [26] supports bottom-up, it does not provide early termination which iBFS leverages for faster traversal. In essence, BFS is a memory-intensive workload that matches well with thousands of lightweight threads provided by GPUs. Prior work such as [6,29–31] has shown great success of using GPUs for single-source BFS. To the best of our knowledge, this is the first work that supports concurrent BFS on GPUs.

The first technique of iBFS is motivated by the observation that a naive implementation that simply runs multiple BFS instances sequentially or in parallel would not be able to achieve high parallelism on GPUs. To address this challenge, iBFS proposes the technique of *joint traversal* to leverage the shared frontiers among different concurrent BFS instances, as they can account for as high as 48.6% of the vertices in some of the graphs we have evaluated. In particular, iBFS executes different traversals within a single GPU kernel. That is, all concurrent BFS instances share a joint frontier queue and a joint status array.

Second, to achieve the maximum benefit of joint traversal, iBFS shall execute all BFS instances together. However, this is impossible due to limited hardware resources, e.g., global memory size and thread count, on GPUs. Fortunately, we

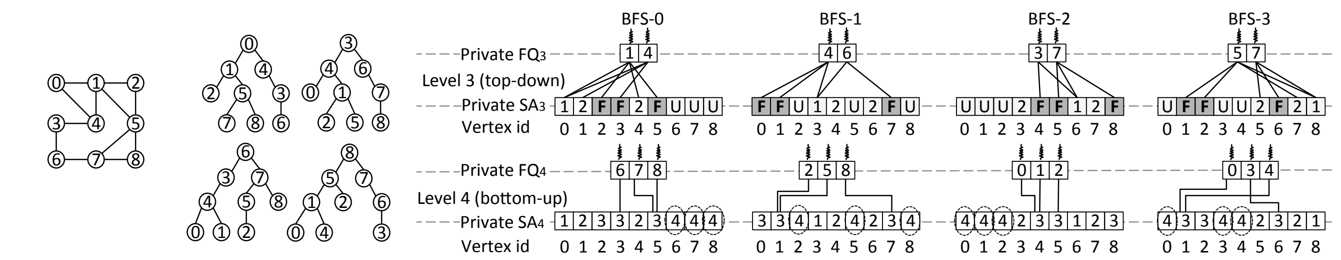


Figure 1: (a) An example graph used throughout this paper (b) Four valid BFS traversal trees from different source vertices, i.e., BFS-0, BFS-1, BFS-2, and BFS-3 starting from vertices 0, 3, 6, and 8, respectively. (c), (d), (e) and (f) are two levels traversal of the corresponding four valid trees, where the top half represents top-down traversal and the bottom half bottom-up. In all examples, FQ and SA stand for frontier queue and status array, respectively. The frontier queue stores frontiers while the status array indicates the status of each frontier in the current level where "F", "U" and numbers represent "Frontier", "Unvisited" and its depth (visited), respectively. The dotted circles indicate the updated depth for each frontier.

discover that grouping BFS instances can be optimized to ensure a high ratio of frontier sharing among different instances. Guided by a theorem on inter-group sharing ratio, iBFS develops the second technique of *GroupBy* that selectively combines BFS instances into optimized batches, which results in 2× speedup of the overall performance.

A typical BFS algorithm starts from the source vertex in a top-down fashion, inspects the frontiers at each level, and switches to bottom-up to avoid inspect too many edges unnecessarily. GroupBy improves iBFS performance in both top-down and bottom-up, although in different fashions. In top-down, higher sharing ratio directly reduces the number of memory access for inspection and expansion. In contrast, through sharing, GroupBy allows bottom-up traversals to complete in approximately the same amount of time, minimizing workload imbalance across multiple BFSes.

Third, iBFS would need to inspect a considerable amount of frontiers at multiple levels, in some case, upto 15× more than a single BFS. Although for each individual BFS not every vertex is a frontier at every level, concurrent BFS significantly increases the number of frontiers at each level. While GPUs offer thousands of hardware threads, traversing millions of the vertices of large graphs in parallel remains challenging. To this end, iBFS utilizes a *bitwise status array* that uses one bit to represent the status of the vertex for each BFS instance. This reduces the size of data fetched during inspection, and more importantly through bitwise operations reduces the number of threads needed for inspection, together accelerating the graph traversal by 11×.

In summary, the contributions of this paper are:

- iBFS combines joint traversal and GroupBy to allow both top-down and bottom-up BFS to run very efficiently within each group. This is achieved via utilizing a set of outdegree-based GroupBy rules to form good groups of concurrent BFS instances that are able to enjoy a high sharing ratio of frontiers among them. A theoretical study on frontier sharing and GroupBy is also presented.
- iBFS also develops bitwise-operations based inspection and frontier identification, which further simplifies the traversal on GPUs and delivers another key benefit of allowing early termination for bottom-up BFS. As a result, iBFS achieves up to 2.6× speedup compared to the state of the art [26].
- We have implemented and evaluated iBFS on a wide range of graphs, as well as on both a single GPU and a cluster. To the best of our knowledge, this is the first in both running GPU-based concurrent BFS on joint data structures, i.e., joint frontier queue and bitwise status array, and scaling to $\mathcal{O}(100)$ GPUs with unprecedented traversal rate, i.e., delivering 57,267 billion traversed edges per second (TEPS) on 112 GPUs.

The rest of the paper is organized as follows: Section 2 introduces the background on BFS, concurrent BFS, and the motivations. Section 4 presents the design of single-kernel and joint traversal. Section 5 discusses our GroupBy technique. Section 6 presents the bitwise optimization. Section 8 presents the evaluations on our iBFS system. Section 9 discusses the related work and Section 10 concludes.

2. BACKGROUND

BFS typically starts the traversal in top-down and switches to bottom-up in a later stage [32, 33]. Both directions perform three tasks at each level, namely, *expansion*, *inspection*, and *frontier queue generation.* To start, *frontier queue* (FQ) is initialized with the source vertex and will always contain the vertices (frontiers) to be expanded from at the next level. From the frontiers, *expansion* explores their edges to the adjacent vertices, and *inspection* updates the statuses of those neighbors in the *status array* (SA) which maintains a record (unvisited, frontier, or visited) for each vertex.

In top-down BFS, expansion and inspection aim to identify unvisited neighbors of each frontier. Figure 1 presents an example of breadth-first traversal. Given the graph in Figure 1(a), at level 3, a top-down BFS shown in the top half of Figure 1(c) (BFS-0 starting from vertex 0) will expand from the queue {1, 4} and mark their unvisited neighbors{2, 3, 5} as frontiers in the status array. Although vertices {0, 1, 4} have been visited before, they will still be inspected at this level, but their statuses will not be updated.

In contrast, bottom-up BFS tries to find a parent for unvisited vertices in expansion and inspection, and in this case, stores unvisited vertices in the frontier queue. In the bottom half of Figure 1(c), BFS-0 at level 4, {6, 7, 8} are unvisited vertices, i.e., treated as frontiers here. For vertex 6, since its first neighbor 3 is visited, bottom-up BFS will mark the depth of vertex 6 as 4, and there is no need to check additional neighbors. This is what we refer to as *early termination* in bottom-up BFS, which can potentially lead to significant performance gains. The newly proposed bitwise optimization in iBFS can further expedite this process for concurrent traversals and lead to even higher performance.

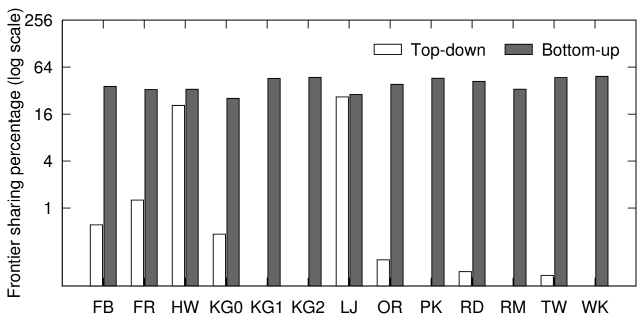
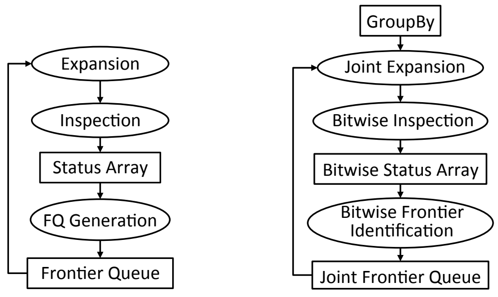


Figure 2: Average frontier sharing percentage between two different BFS instances.



(a) A Single BFS (b) iBFS

Figure 3: The flow charts of (a) BFS, (b) iBFS.

Among three tasks at each level, inspecting adjacent vertices of the frontiers involves a lot of random memory accesses (pointer-chasing), accounting for most of the runtime. This can be observed on four BFS traversals for both top-down and bottom-up in Figure 1.

Concurrent BFS executes multiple BFS instances from different source vertices. Using the example in Figure 1, four BFS instances start from vertex 0, 3, 6, and 8, respectively. A naive implementation of concurrent BFS will run all BFS instances separately and keep its own private frontier queue and status array. On a GPU device, each individual subroutine is defined as a *Kernel*. Therefore, in the aforementioned example, four kernels will run four BFS instances in parallel from four source vertices. NVIDIA Kepler provides *Hyper-Q* to support concurrent execution of multiple kernels, which dramatically increases the GPU utilization especially when a single kernel cannot fully utilize the GPU [34].

Unfortunately, this naive implementation of concurrent BFS takes approximately the same amount of time as running these BFS instances sequentially, as we will show later in Section 8. For example, for all the graphs evaluated in this paper, sequential and naive implementation of concurrent BFS take average 52 ms and 48 ms, respectively, with a difference in traversal rate of 500 million TEPS. The main reason for such a small benefit is because simply running multiple BFS instances in parallel would overwhelm the GPU, especially at the *direction-switching level* when a BFS goes from top-down to bottom-up. At that moment each individual BFS would require a large number of threads for their workloads. As a result, such a naive implementation may even underperform a sequential execution of all BFS instances.

Opportunity of Frontier Sharing: iBFS aims to address this problem by leveraging the existence of frontiers shared among different BFS instances. Figure 2 presents the average percentage of shared frontiers per level between two instances. The graphs used in this paper are presented in Section 8. Top-down levels have smaller number of shared frontiers (close to 4% on average) whereas bottom-up levels have much more as high as 48.6%. This is because bottom-up traversals often start from a large number of unvisited vertices (frontiers in this case) and search for their parents. The proposed GroupBy technique can improve the sharing for both directions to 10× and 1.7×, respectively.

Potentially, the shared frontiers can yield three benefits in concurrent BFS: (1). These frontiers need to be enqueued only once into the frontier queue. (2). The neighbors of shared frontiers need to be loaded in-core only once during expansion. (3). Memory accesses to the statuses of those neighbors for different BFSes can be coalesced. It is important to note that each BFS still has to inspect the statuses independently, because not all BFSes will have the same statuses for their neighbors. In other words, shared frontiers do not reduce the overall workload. Nevertheless, this work proposes that shared frontiers can be utilized to offer faster data access and saving in memory usage, both of which are critical on GPUs. This is achieved through a combination of bitwise, joint traversals and GroupBy rules that guide the selection of groups of BFSes for parallel execution.

3. iBFS OVERVIEW

In a nutshell, iBFS as shown in Figure 3, consists of three unique techniques, namely, joint traversal, GroupBy, and bitwise optimization, which will be discussed in Section 4, 5, and 6, respectively.

Ideally the best performance for iBFS would be achieved by running all i BFS instances together without GroupBy. Unfortunately, ever-growing graph sizes, combined with limited GPU hardware resources, puts a cap on the number of concurrent BFS instances. In particular, we have found that GPU global memory is the dominant factor, e.g., 12GB on K40 GPUs compared to many TB-scale graphs.

Let M be GPU memory size and N the maximum number of concurrent BFS instances in one group (i.e., the group size). If the whole graph requires S storage, a single BFS instance needs $|SA|$ to store its data structures (e.g., the status array for all the vertices), and for a joint traversal each group requires at least $|JFQ|$ for joint data structures (e.g., the joint frontier queue), then $N \leqslant \frac{M-S-|JFQ|}{|SA|}$. In most cases N satisfies $1 < N \ll i \leqslant |V|$. In this paper, we use a value of 128 for N by default.

Unfortunately, randomly grouping N different BFS instances is unlikely to produce the optimal performance. Care has to be taken to ensure a good grouping strategy. To illustrate this problem, for a group A with two BFS instances BFS-s and BFS-t, let $JFQ_A(k)$ be the joint frontier queue of group A at level k, $FQ_s(k)$ the individual frontier queue for BFS-s, and FQ_t for BFS-t. Thus, $|JFQ_A(k)| = |FQ_s(k)| \cup |FQ_t(k)| - |FQ_s(k)| \cap |FQ_t(k)|$, where $|FQ_s(k)| \cap |FQ_t(k)|$ represents the shared frontiers between two BFS instances. Clearly, the more shared frontiers each group has, the higher performance iBFS will be able to achieve. Before we describe the GroupBy technique in Section 5 that aims to maximize such sharing within each group, we will first introduce how iBFS achieves joint traversal in the next section which makes parallel execution possible.

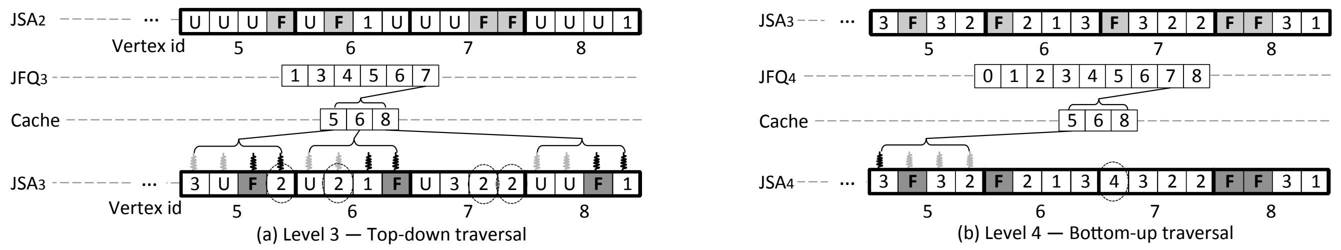


Figure 5: iBFS traversal on joint status array (a) Level 3 – top-down traversal and (b) Level 4 – bottom-up traversal using the example graph and four BFS instances from Figure 1, here i=4 and black threads are active while gray are not. While this figure only presents the inspection of one vertex 7, we keep JSA updated with the latest depth represented as dotted circles. Note that JSA_3 is different between (a) top-down and (b) bottom-up as frontiers are identified differently.

4. JOINT TRAVERSAL

In this work, we propose to implement iBFS in a single GPU kernel, different from prior GPU-based work [35]. This way, iBFS will be able to exploit the sharing across different BFS instances, e.g., if two threads of the same kernel are scheduled to work on a shared frontier, iBFS only needs to load adjacent vertices from global memory once. This benefit would be, otherwise, not possible on GPUs with a multi-kernel implementation which would be used for the aforementioned naive implementation.

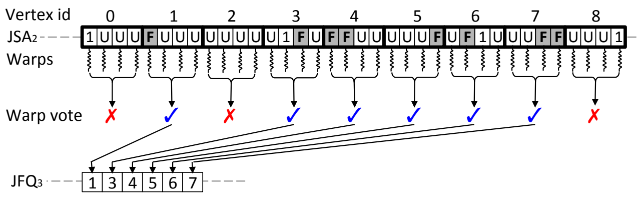


Figure 4: Generate joint frontier queue (JFQ_3) from joint status array (JSA_2). Assuming we are executing the four BFS traversals of Figure 1 in a single kernel.

The joint traversal of iBFS uses two joint data structures for all concurrent BFS instances: (1) **Joint Status Array (JSA)** is used to keep the status of each vertex for all instances. For each vertex, iBFS puts its statuses for different BFS instances sequentially. For example, in Figure 4, as we run four concurrent BFSes, four bytes are used for each vertex. For vertex 0, the first byte of 1 indicates that vertex 0 has been visited with the depth of 1 in BFS-0, and the next three bytes of U indicate that the same vertex 0 has yet been visited for all three other BFS instances. (2) **Joint Frontier Queue (JFQ)** is a set of all the frontiers from concurrent BFS instances, where any shared frontier appears only once. Thus, this queue requires the maximum size of $|V|$ to store the frontiers for all i BFS instances. In comparison, using a private frontier queue would require a much bigger queue with the size of $i\times|V|$.

A GPU uses single instruction multiple thread (SIMT) model and schedules threads in a *warp* that consists of 32 threads [36]. Several warps can form a thread block called *Cooperative Thread Array* (CTA). The threads within a block can communicate with each other quickly with *shared memory* and built-in primitives.

To generate the JFQ, iBFS assigns one warp to scan the status of each vertex as in Figure 4. If this vertex happens to be a frontier for any BFS instance, iBFS needs to store it in the joint frontier queue, e.g., vertex 1 is put in JFQ_3 because it is a frontier for BFS-0 at level 3. In contrast, vertex 0 is not considered a frontier for all four BFS instances.

It is worthy to note that iBFS uses a CUDA vote instruction, i.e., `__any()`, to communicate among different threads in the same warp and schedules one thread to enqueue the frontier. Furthermore, iBFS uses another CUDA feature **`__ballot(parameter)`** to generate a separate variable to indicate which BFS instances share this frontier. This is important for shared frontiers, e.g., while vertex 7 is a frontier in BFS-2 and BFS-3, it would appear once in the joint frontier queue. Removing such redundant frontiers from the queue substantially reduces the number of costly global memory updates, which contributes partly to performance gains obtained by iBFS.

iBFS's joint traversal introduces two unique memory optimizations to reduce the number of expensive global memory transactions: (1) during expansion, iBFS uses a new cache presented in Figure 5 to load the adjacent vertices of a frontier from GPU's global memory to its shared memory to feed all BFS instances. This way the neighbors from each frontier will only be loaded from global memory once, although numerous requests may come from multiple BFS instances. This benefit is not limited just to shared frontiers, rather every frontier in the queue. And (2) during inspection, iBFS schedules multiple threads with contiguous thread IDs to work on each frontier. Here the number of threads is the same as the that of concurrently executed BFS instances. This is because on GPUs one global memory transaction typically fetches 16 contiguous data entries from an array and only continuous threads can share the retrieved data. On the other hand, if continuous threads write to the same memory block, such writes are coalesced into one global memory transaction as well.

Now we will use an example to show how iBFS utilizes these new structures in both top-down and bottom-up traversal. Figure 5(a) exhibits the top-down traversal of vertex 7 which is a frontier in the proceeding level (i.e., level 2). During expansion, the neighbors {5, 6, 8} of vertex 7 are loaded in the cache. During inspection, BFS instances that do not have this frontier will not inspect the neighbors, specifically, the first and second threads in this figure. On the other hand, when the third and fourth threads access the statuses of three neighbors, it is performed in a single global memory transaction since these statuses are stored side by side and accessed by contiguous threads. At this level, vertex 7 will also have its status updated with the depth of 2.

The bottom-up traversal is performed in a different manner as shown in Figure 5(b). For the frontier vertex 7, iBFS will similarly load its adjacent vertices {5, 6, 8} into the cache. The difference here is that iBFS will check if {5, 6, 8} are already visited, if so, mark the depth of 7 as 4.

Note that a vertex can be a frontier at both top-down and bottom-up levels, for example, vertex 7 in Figure 5. In Figure 5(a), vertex 7 is a frontier for top-down traversal of the third and fourth BFSes, and in Figure 5(b) bottom-up of the first BFS. Clearly, as long as one BFS considers a vertex as a frontier at a particular level, this vertex shall be enqueued in iBFS.

5. GROUPBY

In this section, we will introduce the concept of sharing ratio, and propose several outdegree-based GroupBy rules. We will first use top-down traversal for the discussion of GroupBy and later present the impacts on bottom-up BFS.

5.1 Frontier Sharing Degree and Ratio

To a great extent, the performance of iBFS is determined by how many frontiers are shared at each level during the joint traversal of each group. This is because if multiple BFS instances have a common frontier in one level, all its edges will be checked only once, thus leading to overall performance improvement. In this work we define for any group, say group A with the size N, the Sharing Degree (SD$_A$) as the degree of sharing that exists in the joint frontier queue:

$$SD_A = \frac{\sum_k \sum_{j=1}^{N} |FQ_j(k)|}{\sum_k |JFQ_A(k)|} \tag{1}$$

where $j \in$ [1, N] and represents the j-th BFS instance in the group A, and k represents the level (or depth) for the traversal. In essence, SD$_A$ shows on average each joint frontier is shared by how many BFS instances in a group. Thus, the Sharing Ratio can be easily calculated as SD divided by the total number of instances in the group.

Traditionally, the time complexity of a BFS is calculated by the number of inspections performed, that is, the edge count $|E|$. Thus the time of a sequential execution of group A will be N $\cdot$ $|E|$.

In this work, We use $T_A(k)$ to represent the time of the joint execution of group A at the level k, that is,

$$T_A(k) = \sum_{v \in JFQ_A(k)} outdegree(v)$$

where v stands for each frontier in the JFQ at the level k. And the total runtime T_A for group A can be calculated by summarizing the runtime of each level, which is:

$$T_A = \sum_k T_A(k) = \sum_k \sum_{v \in JFQ_A(k)} outdegree(v) \tag{2}$$

According to the Amdahl's Law, the speedup $Speedup_A$ of joint traversal for group A over sequential execution is:

$$Speedup_A = \frac{N \cdot |E|}{T_A} \tag{3}$$

Let $\bar{d}$ be the average outdegree, we can obtain the expected value of $Speedup_A$ using equations (2) and (3). Thus,

$$\begin{aligned}\mathbb{E}[Speedup_A] &= \mathbb{E}[\frac{N \cdot \bar{d} \cdot |V|}{\bar{d} \cdot \sum_k |JFQ_A(k)|}] \\ &= \mathbb{E}[\frac{N \cdot |V|}{\sum_k |JFQ_A(k)|}]\end{aligned} \tag{4}$$

Lemma 1. *For any BFS group, say group A, its sharing ratio is equal to the expected value of $Speedup_A$, that is,*

$$SD_A = \mathbb{E}[Speedup_A] \tag{5}$$

Proof. Because for the j-th BFS every vertex will become a frontier at one of the levels, the sum of $|FQ_j(k)|$ across all levels is equal to the total number of the vertices, that is, $\sum_k |FQ_j(k)| = |V|$.

$$\begin{aligned}SD_A &= \frac{\sum_k \sum_{j=1}^{N} |FQ_j(k)|}{\sum_k |JFQ_A(k)|} = \frac{\sum_{j=1}^{N} |V|}{\sum_k |JFQ_A(k)|} \\ &= \mathbb{E}[Speedup_A]\end{aligned} \tag{6}$$

□

Lemma 1 demonstrates that the sharing ratio reflects the performance of the joint execution when it is compared to the sequential execution of such a group. However, for any group, since the size of JFQ at each level is not known until runtime, we will not able to calculate a priori the sharing ratio as well as the expected performance. Fortunately, we discover that the sharing ratios at the first few levels can be used as a good metric, and there also exists an important correlation for sharing ratios at consecutive levels.

Theorem 1. *Given any two BFS groups A and B with the same number of BFS instances, if at the level k their sharing ratios obeys $SD_A(k) > SD_B(k)$, at the level $k+1$ the following relationship, $\mathbb{E}[SD_A(k+1)] > \mathbb{E}[SD_B(k+1)]$, will hold.*

Proof. Let us start with one group A. At the level k, the j-th BFS performs two main tasks for a frontier vertex v, i.e., expansion and inspection. In the first task, the j-th BFS fetches the neighbor list of frontier v, and in the second task, it checks all neighbors. The number of inspections is equal to the outdegree of v.

We define *null inspections* as those that do not lead to an increase in the size of the frontier queue at a particular level. Note that as vertices are shared, null inspections do not necessarily mean such inspections have not found a frontier at the next level. There are three cases that render a neighbor check as a null inspection. The first two cases are relevant to a single BFS, while all three are applicable to iBFS. For a neighbor w of the frontier v,

Case 1 : the neighbor w is visited.

Case 2 : the neighbor w has already been marked as a new frontier by other inspections at the level k. This check will be discarded, and will not increase the size of the frontier queue. This happens because w may have additional parents other than v.

Case 3 : the neighbor w has already been marked as a new frontier by other concurrent BFS instances. Similarly, in joint traversal, this check will be discarded, and will not increase the size of the frontier queue, because w may be shared by concurrent BFS instances.

We use three values to represent the occurring percentages of three cases: α accounts for case 1 and 2 for a single BFS, β for case 1 and 2 for iBFS, and γ for case 3 for iBFS alone. Thus, the probability of the complement of case 1 and 2 for a single BFS can be represented by $(1-\alpha)$, and so on.

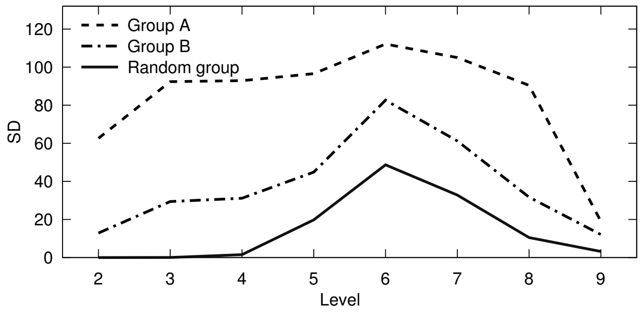


Figure 6: Sharing ratio trend of Facebook graph (FB).

Now for the level $k+1$, the size of JFQ is equal to the sum of the number of frontiers shared by different BFS instances within group A, from one to N instances. Thus, let $s_j(k)$ be the number of vertices that are shared by exactly j BFS instances, we have

$$SD_A(k+1) = \frac{\sum_{j=1}^{N} |FQ_j(k+1)|}{|JFQ_A(k+1)|} = \frac{\sum_{j=1}^{N} j \cdot s_j(k+1)}{\sum_{j=1}^{N} s_j(k+1)} = \frac{\sum_{j=1}^{N} j \cdot s_j(k) \cdot d_j \cdot (1-\alpha_j)}{\sum_{j=1}^{N} s_j(k) \cdot d_j \cdot (1-\beta_j-\gamma_j)} \quad (7)$$

where d_j denotes the average out-degree of these frontiers.

Since we do not know a priori the values for d_j, α_j, β_j, and γ_j, if replacing them with average values $\bar{d}$, $\bar{\alpha}(k)$, $\bar{\beta}(k)$, and $\bar{\gamma}(k)$, the expected value of $SD_A(k+1)$ is

$$\mathbb{E}[SD_A(k+1)] = \frac{\sum_{j=1}^{N} j \cdot s_j(k) \cdot \bar{d} \cdot (1-\bar{\alpha}(k))}{\sum_{j=1}^{N} s_j(k) \cdot \bar{d} \cdot (1-(\bar{\beta}(k)) - \bar{\gamma}(k))} \quad (8)$$

We assume that each single BFS has the same probability $\bar{\alpha}(k)$, so for iBFS, we have $\bar{\beta}(k) = \bar{\alpha}^j(k)$, and

$$\mathbb{E}[SD_A(k+1)] = \frac{\sum_{j=1}^{N} j \cdot s_j(k) \cdot \bar{d} \cdot (1-\bar{\alpha}(k))}{\sum_{j=1}^{N} s_j(k) \cdot \bar{d} \cdot (1-(\bar{\alpha}^j(k)) - \bar{\gamma}(k))} \approx SD_A(k) \cdot \frac{1-\bar{\alpha}(k)}{1-\bar{\gamma}(k)} \quad (9)$$

Now we consider two groups A and B. If $SD_A(k) > SD_B(k)$, as $\bar{\alpha}(k)$ and $\bar{\gamma}(k)$ are independent of groups, we get $\mathbb{E}[SD_A(k+1)] > \mathbb{E}[SD_B(k+1)]$ from equation (9). □

Figure 6 exhibits average sharing ratios for three different groups. We start from the second level as no BFS instances share the source vertices, and the maximum SD is equal to N, that is, 128. Since group A has a higher SD at the second level than group B, it always has higher ratios in the following levels. Similarly, group B has higher ratios than random. Clearly, as shown in Figure 6, sharing ratios would not increase monotonically for a group, which tend to peak at the first several bottom-up levels. Nevertheless, Theorem 1 implies that a higher sharing ratio of the initial levels can lead to a higher expected sharing ratio in later levels. Combining Lemma 1 and Theorem 1, we can see that a GroupBy strategy can achieve good speedup by focusing on the first several levels as described in Lemma 2.

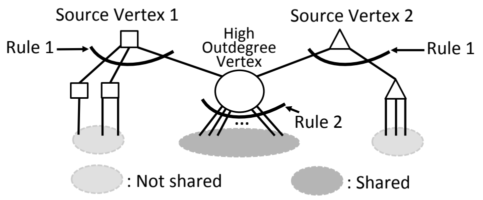


Figure 7: An illustration of a power-law graph.

LEMMA 2. *For two BFS groups A and B, for a small number k, if their initial sharing ratios obey $SD_A(k) > SD_B(k)$ at the level k, the expected values for the speedups will follow $\mathbb{E}[speedup_A] > \mathbb{E}[speedup_B]$.*

5.2 Outdegree-based GroupBy Rules

Lemma 2 states that a good GroupBy rule can be obtained by focusing on increasing the BFS sharing in the first several levels. Fortunately, an easy analysis on the outdegrees of the frontiers at these levels, coupled with a quick determination on the connectivity to *high-outdegree vertices*, can lead to two simple yet powerful **outdegree-based rules**:

Rule 1 The out-degrees of these two source vertices are less than p.

Rule 2 Two source vertices connect to at least one common vertex whose outdegree is greater than q.

Both rules are complementary to each other. Since the first rule ensures small outdegrees of the source vertices, other non-shared neighbors will not amortize the sharing ratio contributed by the shared vertex with high outdegrees from the second rule. With these two GroupBy rules, two BFS instances will likely get high sharing ratio.

Figure 7 shows an example of a power-law graph which is the focus of this work. In this example, many vertices are connected to a high-outdegree vertex, and the source vertices for different BFSes are not exception. When they share a common high-outdegree vertex, high sharing ratio across multiple BFSes can be easily achieved. It is not required that the source vertex directly connects to a high-outdegree vertex, as long as within the first several levels. Simply put, for two BFSes, it is beneficial to group them together if their source vertices have relatively small number of edges and also connect to high-outdegree vertices.

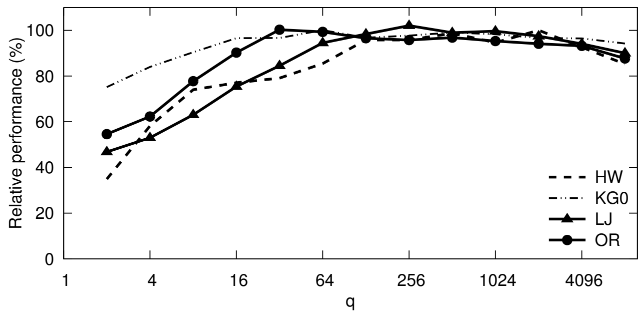


Figure 8: Performance of *GroupBy* for different q.

Selection of p and q. Figure 8 plots the GroupBy performance for different q values. One can see that the performance rises initially and reaches the peak, typically around the range of 128 – 1024. The lower performance is observed for both smaller and larger q. For smaller q, the groups would likely have small sharing ratios. On the other hand, for larger q, very few BFS instances would satisfy the GroupBy rules. In this paper q is 128 by default. Once q is decided, p is selected in the ascending order from a sequence of numbers that are the power of 2. Here we select p from a sequence of $4, 16, 64$, and 128.

The rules are applied as follows. First, iBFS selects all groups that satisfy both rule 1 and 2 with pre-determined p and q. iBFS will run such groups directly when their sizes are larger than N, the maximum group size described in Section 3. Otherwise, several small groups, likely using different values of p, will be combined and run together. Second, iBFS will try to combine the groups with different high-outdegree vertices. Last, when no BFS satisfies both rules, iBFS will group the remaining them in a random manner.

Sharing ratio improvement. Figure 9 plots the improvement on the sharing ratio using outdegree-based rules for both top-down and bottom-up. Specifically, for top-down, our GroupBy rules improves the sharing ratio by 10×, which increases from average 3.9% to 39.3% for 128 BFS instances. For bottom-up, although the relative improvement is smaller, the sharing ratio is greatly improved to average 66.1% which we will discuss shortly.

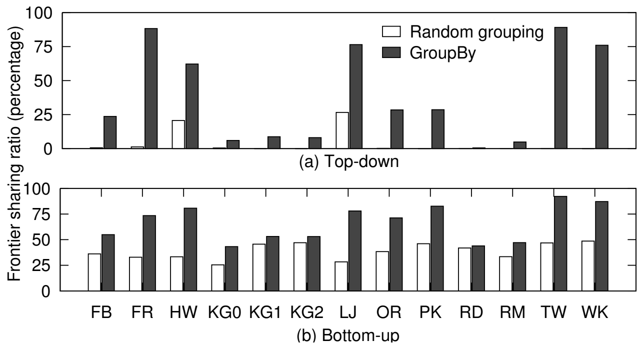


Figure 9: Frontier sharing ratio comparison between random and GroupBy.

For random graph that has a relatively uniform outdegree distribution, iBFS can adopt a slightly different rule. Since Theorem 1 and Lemma 2 still apply, iBFS can select a group of BFS instances if they share some common vertices from the sources. Our evaluation shows that such a rule may obtain 3.5× and 5% improvement in top-down and bottom-up respectively on a random graph, albeit much smaller compared to other graphs in our test suite.

5.3 GroupBy on Bottom-Up BFS

So far we have focused on GroupBy during the top-down stage of BFS. Surprisingly, good GroupBy rules for top-down will lend themselves to achieving great speedups during the bottom-up stage. Together, GroupBy further accelerates iBFS performance by increasing frontier sharing and more importantly balancing the workload across various BFS instances. These two effects can be best explained using an example of two BFS instances, say BFS-s and BFS-t, within a group. In Figure 10, Area I represents shared, visited vertices between two BFSes whereas Area III shared, unvisited vertices. When BFS-s and BFS-t share more frontiers at

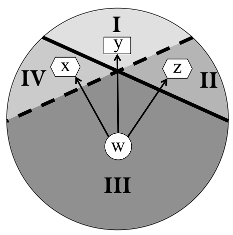


Figure 10: The circle represents all the vertices in a graph. There are two instances BFS-s and BFS-t. Area I and II represent the visited vertices for BFS-s (frontiers at top-down), and Area III and IV the unvisited vertices for BFS-s (frontiers at bottom-up). Similarly, Area I and IV represent the visited vertices for BFS-t, and Area II and III the unvisited vertices for BFS-t.

top-down, that is, bigger Area I, they share more frontiers at bottom-up, that is, bigger Area III. From Figure 9(b), one can see that for bottom-up, all the graphs still achieve 66.1% on average, close to 2 times improvement. This is significant because in bottom-up frontier sharing ratio is already high (38.7%) to begin with.

The most significant benefit of GroupBy when it comes to bottom up lies on the fact that it helps balance the workload. For a shared vertex w in Area III shown in Figure 10, depending on which BFS it belongs to and who is its parent, it may need to search all three other areas. Again, When BFS-s and BFS-t share more frontiers at top-down, that is, bigger Area I, this also increases the likelihood that both BFS instances discover the shared parent y in Area I. As bottom-up inspection terminates as soon as a parent is found, such sharing will further lead to similar runtimes across different BFSes, that is, reducing the variance in runtimes. To demonstrate this, Figure 11 shows the standard deviation for the number of inspections during bottom-up before and after GroupBy. Since GroupBy combines the BFS instances that would find their parent with a similar runtime, it lowers the standard deviation by 13×. Among all the graphs, it helps the TW graph the most – by 66×, reducing the number of inspections from 744 to 11.3. Clearly, GroupBy helps to transform a highly imbalanced workload to a much more balanced one.

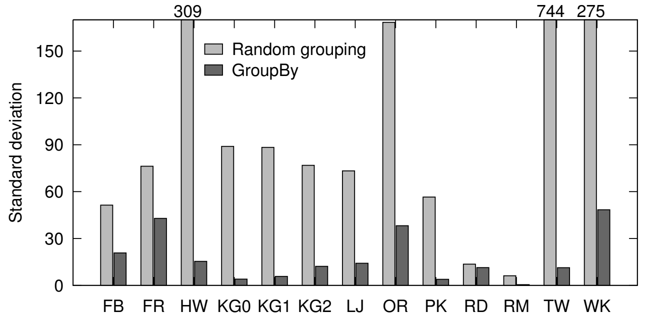


Figure 11: Standard deviation of the distribution for number of inspections during bottom-up before and after GroupBy.

6. GPU-BASED BITWISE OPERATIONS

Although joint status array removes random inspections on shared frontiers, assigning a warp of threads to work on each frontier is not feasible when a graph has millions of frontiers. While modern GPUs like NVIDIA K40 provide thousands of hardware threads, clearly we still need to find a smart way to utilize threads effectively when dealing with several millions of vertices in a graph. To this end, we propose a novel concept of **Bitwise Status Array (BSA)** in iBFS that uses a single bit to represent the status of each vertex for different BFS instances. And all bits of one vertex

Algorithm 1 Bitwise iBFS at Level $k+1$

```
1: BSA_{k+1} ← BSA_k
2: forall frontier $f$ in parallel do
3:     foreach neighbor $v$ of $f$ do
4:         if Top-Down then
5:             $BSA_{k+1}[v] = BSA_{k+1}[v]$ OR_atomic $BSA_k[f]$
6:         else // Bottom-up
7:             if $BSA_{k+1}[f]$==0xff...f then
8:                 break; // Early termination
9:             end if
10:            $BSA_{k+1}[f] = BSA_{k+1}[f]$ OR $BSA_k[v]$
11:        end if
12:    end for
13: end for
```

Algorithm 2 Frontier Identification at Level $k+1$

```
1: forall vertex $v$ in parallel do
2:     if Top-down then
3:         if $BSA_{k+1}[v]$ XOR $BSA_k[v]$ then
4:             JFQ.enqueue($v$)
5:         end if
6:     else // Bottom-up
7:         if NOT $BSA_{k+1}[v]$ then
8:             JFQ.enqueue($v$)
9:         end if
10:    end if
11: end for
```

are kept in a single variable. If this vertex is visited, we set it as 1, otherwise 0.

Figure 12 presents the mapping between joint and bitwise status array. Here for any vertex, while JSA uses the first four variables (3, U, 3, 2) to store the status for four BFS instances, BSA only needs a single variable (1011).

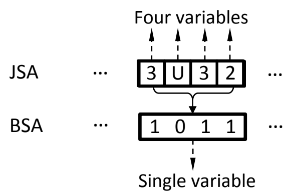


Figure 12: Mapping from joint status array (JSA) to bitwise status array (BSA) for one vertex.

With the bitwise status array, iBFS only needs one expansion thread to fetch the statuses of each vertex for all concurrent BFS. In addition, the inspection of concurrent BFS that will be described shortly can be executed by a single bitwise operation, which can be easily done with a single thread. In summary, this design frees up a substantial number of threads and further reduces the number of global memory access.

Bitwise Inspection. At level $k+1$, iBFS keeps a copy of bitwise status array – BSA_k. During traversal, the statuses for the most recently visited vertices are also marked as 1 in BSA_{k+1}. At the end of each level, the differences between BSA_k and BSA_{k+1} are used to identify the just visited vertices. Algorithm 1 shows the pseudo-code.

For top-down traversal, iBFS assigns a single thread to work on frontier f. This thread loads frontier's status (i.e., $\text{BSA}_k[\text{f}]$) from BSA_k and subsequently sets all neighbors' status in $\text{BSA}_{k+1}[\text{v}]$ via a bitwise `OR` operation. For each neighbor, this `OR` operation only affects the bits for the corresponding BFS instances that share this frontier, because only these bits in $\text{BSA}_k[f]$ are recently updated as 1.

Using frontier 7 in Figure 13(a) as an example, vertex 7 is a frontier shared by the third and fourth BFS instances in Figure 1. During the inspection of vertex 7's neighbor – {5, 6, 8}, iBFS uses the bitwise `OR` operation between 7's status and each neighbor's status. Specifically, for vertex 5, it sets the third bit in $\text{BSA}_3[5]$, and for vertex 6, the fourth bit in $\text{BSA}_3[6]$. One may also notice that, the `OR` operation does not affect the already set bits, like the fourth bit of vertex 5 in $\text{BSA}_3[5]$. The reason is that the fourth BFS instance has already visited vertex 5 in prior levels.

Because multiple BFS instances may want to set different bits of the same vertex in the bitwise status array concurrently, iBFS needs atomic operations to avoid overwrites of the updates to BSA_{k+1}, another difference from [26].

Bottom-up traversal, similarly assigns a single thread to work on each frontier. However, the frontier's status is updated by the neighbors' statuses, that is, iBFS uses `OR` between $\text{BSA}_{k+1}[\text{f}]$ and $\text{BSA}_k[\text{v}]$ and stores the result into $\text{BSA}_{k+1}[\text{f}]$. Using frontier 7 of Figure 13(b) as an example, its neighbors are {5, 6, 8}. iBFS executes `OR` between vertex 5's and 7's statuses and stores the result in $\text{BSA}_4[7]$.

Early Termination. During bottom-up traversal for some frontiers, like frontier 7 in Figure 13(b), iBFS does not necessarily need to check all its neighbors because it is possible that some of its neighbors (e.g., vertex 5 for frontier 7) can set all bits of this frontier in the bitwise status array. When this happens, iBFS will terminate the inspection on this frontier, eliminating the need to further examine other neighbors (e.g., vertices 6 and 8), which we call early termination in this work. This newly freed-up thread will then be scheduled to work on other frontiers. This is not possible unless the bitwise operations already record all visited vertices as '1' in BSA, in this case before this level $\text{BSA}_4[7]$ already has three bits set. In short, the early termination allows a great reduction in the traversal time compared to prior work such as [26].

Bitwise Frontier Identification. iBFS also needs a different approach for frontier identification that works efficiently with the bitwise status array. Algorithm 2 shows the pseudocode for bitwise frontier identification.

Top-down traversal executes the `XOR` operation between $\text{BSA}_k[\text{v}]$ and $\text{BSA}_{k+1}[\text{v}]$. If *true* is returned, it means some BFS instances have just changed the corresponding bit of v and stored in $\text{BSA}_{k+1}[\text{v}]$. Since top-down treats most recently visited vertices as frontiers, iBFS hence stores the identified frontiers in the joint frontier queue. In contrast, bottom-up traversal, is much easier since it treats unvisited vertices as frontiers. Therefore, iBFS simply performs a `NOT` operation on $\text{BSA}_{k+1}[\text{v}]$. If it is evaluated as *true*, that is, some bits of $\text{BSA}_{k+1}[\text{v}]$ are not set, then this vertex is a frontier and iBFS puts it in the joint frontier queue.

Vector data types on GPUs are also leveraged in iBFS. Clearly, the number of bits in each variable affects the number of concurrent BFS, e.g., if BSA is implemented with `int` type, one variable can represent the statuses for 32 BFS instances. In CUDA, the basic data are `char`, `int` and `long`, and on the other hand `vector` packs multiple basic types into one, e.g., `int4` contains four `ints`, similarly `char4`, `long4` etc. Using the vector data type in iBFS can further reduce the memory access time as it fetches four basic data elements together at one time.

Summary. Our bit-wise approach is novel in a number of ways, which leads to upto 2.6× speedup over [26]. First, [26]

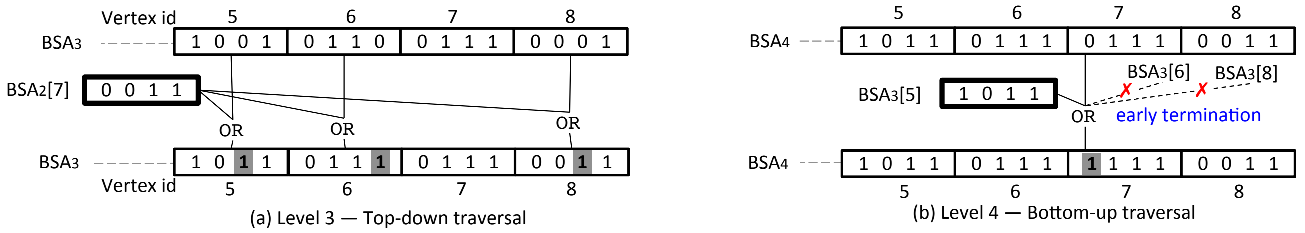

Vertex id
BSA3
BSA2[7]
OR
(a) Level 3 — Top-down traversal
BSA4
BSA3[5]
BSA3[6]
BSA3[8]
early termination
(b) Level 4 — Bottom-up traversal

Figure 13: Traversing four BFS instances of the example graph from Figure 1 with bitwise status array: (a) Level 3 – Top-down traversal. (b) Level 4 – Bottom-up traversal. Shadow bits stand for updated status.

resets the bitwise status array at each level and only records the current level frontiers as 1. In comparison, iBFS records every visited vertex as 1 regardless of which level it is visited. This way our bitwise status array remembers all visited vertices, for which we also introduce a new frontier identification technique and early termination during bottom-up traversal as described earlier. Second, [26] is based on single thread BFS implementation, that is, each thread works on one BFS instance. In contrast, iBFS supports multi-threaded bitwise operation, that is, all available threads will work on a group of BFS instances. Inter-thread synchronization shall be carefully managed, because in iBFS multiple threads will need to update different bits of the same vertex simultaneously. In top-down, iBFS uses (manually controllable) shared memory on GPUs to cache and merge the updates in the same CTA (typically 256 threads), which avoids the overhead of atomic operations at this step. Next, iBFS has to rely on atomic operations to push the combined updates to the global memory. In bottom-up, iBFS performs multi-step tree-based merging of the updates within threads in a warp or CTA, again avoiding atomic operations.

7. iBFS ON CPUS

In principal iBFS can be implemented on CPUs. Specifically, *joint traversal* and *GroupBy* can follow the same design on GPUs. One notable difference is that iBFS would need atomic operation on CPUs for the multi-thread *bitwise operation*. Note that [26] does not need atomic operations because it is based on single-thread BFS. Generally speaking, concurrent BFS is an I/O intensive application, as it always has many frontiers to be processed. As modern CPUs provide tens of cores and thousands of registers [37,38], issuing a large number of CPU threads may improve memory throughput but inevitably would incur high overhead of context switches. On the other hand, GPUs not only provide a large quantity of small cores coupled with huge register files, e.g., 2,880 cores and 983,040 registers on NVIDIA Kepler K40 GPUs, but also support zero-overhead context switch [36]. As we will present shortly, compared to the CPU-based implementation, GPU-based iBFS runs 2× faster on average on various graphs.

8. EXPERIMENTS

iBFS is implemented in 4,000 lines of CUDA and C++ codes, extending a GPU-based high-performance BFS implementation – Enterprise [33]. Since it supports both top-down and bottom-up BFS, our iBFS can be easily configured to support conventional top-down BFS and traverse weighted graphs. iBFS is compiled using g++ 4.4.7, MPI (MVAPICH2) and NVIDIA CUDA 6.0 with the *-O3* flag. Our system is evaluated on NVIDIA Kepler K40 GPUs on our local cluster with Intel Xeon E5-2683 CPUs, and later on the Stampede supercomputer on NSF Extreme Science and Engineering Discovery Environment (XSEDE) program, where iBFS runs from 1 to 112 machines each of which is equipped with one NVIDIA Kepler K20 GPU.

We measure the execution time from when all the data are loaded in GPU memory to the traversal is completed and the results are stored in GPU memory. All the execution uses uint64 data type. We use the metric of *traversed edges per second* (TEPS) to measure the performance, which is calculated by the ratio of the number of directed edges traversed by the search, counting any multiple edges and self-loops, and the time elapsed during iBFS execution. In the tests, iBFS performs breadth-first search from all the vertices.

8.1 Graph Benchmarks

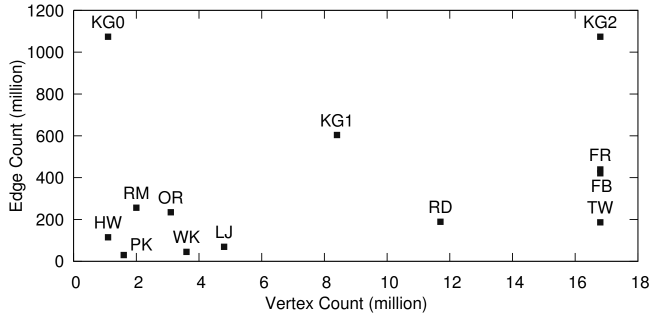


Figure 14: Graph benchmarks.

In this work we use total 13 graph benchmarks to evaluate iBFS, which as summarized in Figure 14 have upto 17 million vertices and 1 billion edges. In particular, there are seven real-world graphs of well-known online social networks. Facebook (FB) [39], a user to friend connection graph, contains 16,777,216 vertices and 420,914,604 edges. Twitter (TW) [40] is a follower connection graph, that is, if user v follows user u, (v, u) is considered as an edge. It also has 16,777,216 vertices and 196,427,854 different edges. Wikipedia (WK) [41] is a inter-website hyper-link graph, which consists of 3,566,908 vertices and 45,030,389 edges. We also obtain four popular online social network graphs, i.e., LiveJournal (LJ), Orkut (OR), Friendster (FR), and Pokec (PK), from Stanford Large Network Dataset Collection [42]. Specifically, LJ contains 4,847,571 vertices and 137,987,546 edges. OR has 3,072,627 vertices with an average outdegree of 75.27. FR contains 16,777,212 vertices and 439,147,122 edges. PK is the smallest graph with 1,632,804 vertices and 30,622,564 edges.

In addition, we generate three types of synthetic graphs with Graph 500 generator [43–45], namely, KG0, KG1 and

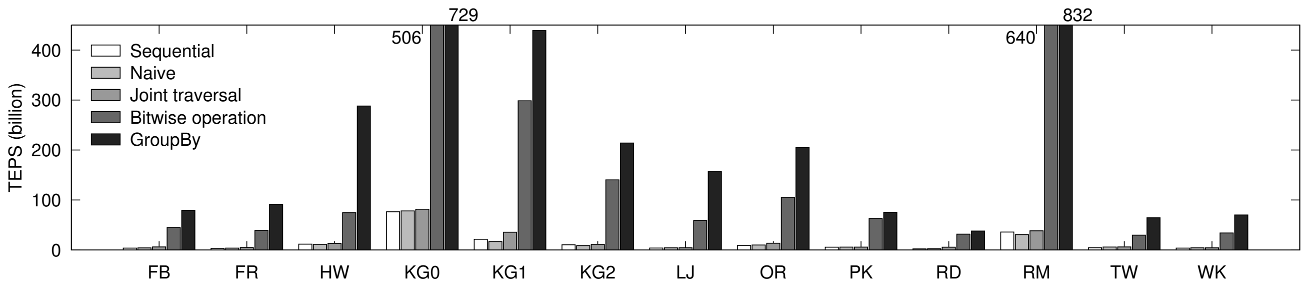


Figure 15: Traversal rate comparison between Sequential BFS, Concurrent BFS, Joint Traversal, Bitwise Optimization, and GroupBy.

KG2. The default value of (A, B, C) parameter is (0.57, 0.19, 0.19) per the requirement of Graph 500 [44]. Specifically, KG0 stands for the high average outdegree graphs, i.e., its average outdegree is 1024 and vertex count is 1,048,576. KG2 serves as the biggest graph in this paper, i.e., with both biggest vertex and edge count – 16,777,216 vertices and 1,073,741,824 edges. KG1 has 8,388,608 vertices and 603,979,776 edges. We also use the DIMACS graph generator [46] to generate the RM and RD graphs in this paper. RM follows the same theory from Graph 500 [43–45] but with a different (A, B, C) set of (0.45, 0.15, 0.15). RM has 2,097,152 vertices and 268,435,456 edges. RD [47] graph has uniform outdegree distribution, i.e., each vertex has roughly the same outdegree. RD contains 11,796,480 vertices and 188,743,680 edges.

All these graphs are stored in the *Compressed Sparse Row* (CSR) format. For graphs that are provided in the edge list format, we translate them into CSR while preserving the edge sequence. For undirected graphs, each edge is considered as two directed edges. For directed graphs, we also store the reversed edges to support the bottom-up traversal. The size of these graph ranges from 478 MB (PK) to 8.2 GB (KG2) when using *long integer* (8 bytes) to represent the vertex id.

8.2 iBFS Performance

We evaluate three techniques, *joint traversal*, *bitwise operation*, and *GroupBy*, and compare against running all BFS instances sequentially (*sequential*) or in parallel without any optimization (*naive*), both of which are based on state-of-the-art BFS implementation such as Enterprise [33]. In this test, we run APSP on all the graphs. As shown in Figure 15, sequential and naive implementation perform roughly the same. On average, the later traverses 1.05× faster, but a worse performance is observed in HW, KG1, KG2 and RM graphs, with only 78% of sequential performance on KG1.

In iBFS, joint traversal delivers 1.4× speedup compared to sequential implementation. The biggest performance gain of 2.6× speedup comes for RD, from 4.4 to 11.3 billion TEPS while the smallest is 1.03× speedup for PK . For other graphs, the performance improvement is more than 10%. On the other hand, bitwise iBFS, on average speedup the traversal by 11×. While the smallest improvement is 6.4× on HW from 11.7 to 75 billion TEPS, the biggest improvement is observed on RM – 18 × speedup from 36 to 640 billion TEPS. Astoundingly, **Groupby** improves iBFS by additional 2× on average beyond bitwise optimization, with the highest traversal rate of 832 billion TEPS on RM.

In this test, iBFS greatly reduces the run time of APSP on all graphs from average 123 hours of sequential traversal to 5.5 hours, where joint traversal helps to reduce to 87.8 hours, bitwise optimization 11.2 hours, and GroupBy the rest. GroupBy incurs a minimal processing time of less than 0.1 second on average.

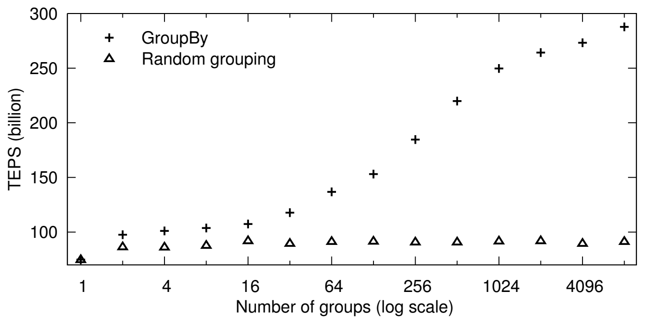


Figure 16: Traversal performance when running different number of BFS groups on HW.

We further evaluate the traversal performance of iBFS as the number of i varies, that is, running MSSP with varied number of source vertices. We observe similar performance on all the graph benchmarks. Figure 16 presents the TEPS when running different number of BFS groups on the HW graph, where the total number of BFS instances equals to the multiply of the number of groups and the group size. Clearly, as there are more BFS instances to run, the benefit of GroupBy – the gap between GroupBy and random grouping – increases because better groups can be formed. Specifically, with randomly formed groups, the traversal rate fluctuates in the range of 75 and 90 billion TEPS, which is raised to 288 billion TEPS with the help of GroupBy.

8.3 iBFS Scalability

In this test, we aim to evaluate the scalability of iBFS on a large number of distributed GPUs on the Stampede supercomputer at TACC. Clearly, as long as different GPUs work on independent BFSes, there is no need for inter-GPU communication. Therefore, the key challenge here is achieving workload balance on GPUs, especially when each individual BFS may inspect different number of edges during bottom-up. The longest time consumption of all the GPUs is reported in this test.

For the five graphs tested, iBFS achieves good speedup from 1 to 112 GPUs (the total number of GPUs on Stampede)[1]. As shown in Figure 17, from one to two GPUs, the biggest speedup, as expected, is from RD of 1.97× because RD graph has the most balanced workload. And even the

[1]As it took all the GPUs on Stampede, we were only given a small time window to conduct this test.

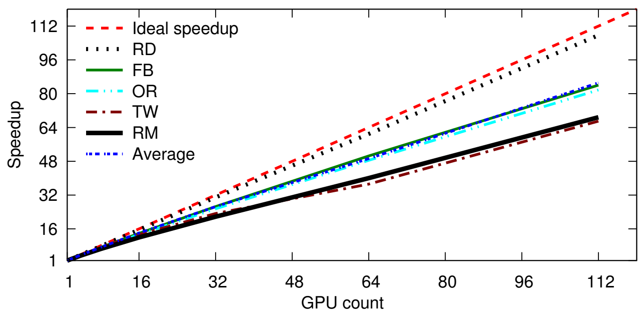


Figure 17: Scalability of bitwise iBFS from 1 to 112 GPUs.

smallest speedup from OR is 1.9×. From one to four GPUs, the average speedup is 3.8× with the smallest speedup from OR of 3.6× while the biggest from RD graph – 3.9×. As the GPU count increases, workload imbalance slowly emerges and begins to negatively affect the performance. Specifically, iBFS achieves an average of 85× speedup for 112 GPUs. Again, RD gets the biggest speedup, i.e., 108×.

In all, iBFS achieves the average traversal rate of 16,509 billion TEPS across all the tested graphs with the maximum 57,267 billion TEPS on RM.

8.4 Joint Traversal and GroupBy

This section uses the NVIDIA profiler [48] to measure the impact of joint traversal and GroupBy on memory accesses. Having a joint frontier queue with only one copy of shared frontiers reduces the potentially large number of global memory writes, compared to having private frontier queues for each BFS instance. Figure 18 shows the number of global store transactions during the frontier queue generation of 1,024 BFS instances. Using private frontier queue, i.e., *private FQ*, executes 4 billion transactions on average across all the graphs, while joint frontier queue with random groups, i.e. *random FQ*, only needs one fourth transactions. The biggest reduction of over 11× is observed in the KG2 graph, from 8.3 billion to 700 million, the smallest saving is 1.2× on HW, i.e., 720 million to 580 million. With *GroupBy*, iBFS further saves the global stores by 2.6× with the largest from HW of 4×.

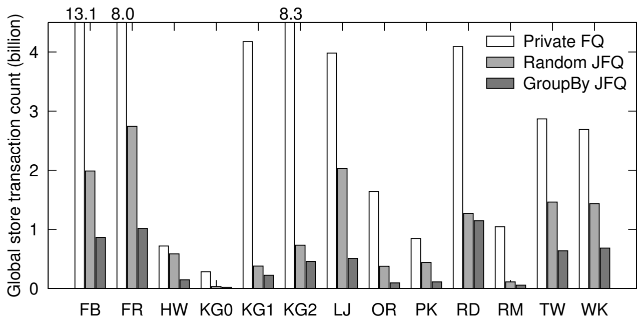


Figure 18: Global store transaction count during the generation of private frontier queue, random joint frontier queue and GroupBy joint frontier queue.

Combining joint status array with careful thread scheduling introduces significant performance benefits, as iBFS minimizes costly memory operations. Figure 19 exhibits global load transactions per request during traversal before and after this optimization. Note that global store transactions per request exhibits similar trend. Since joint status array of iBFS always coalesces the inspections and updates from contiguous threads into a single global memory transaction, our tests across 1,024 BFS instances show that we are able to reduce on average from four loads to a single load. The benefits add up quickly considering a large number of memory operations in concurrent BFS. On average each BFS instance executes 50 million of global load transactions.

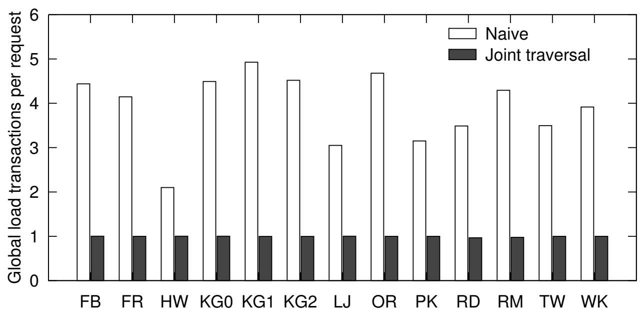


Figure 19: Load transaction count per request.

8.5 Bitwise Operation

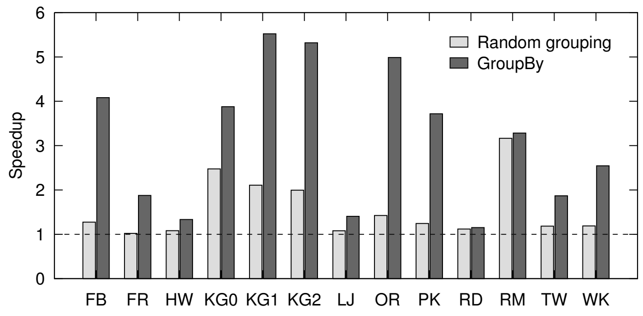


Figure 20: The speedup of our bitwise operation.

We implement the bitwise operation as in [26] and use it as baseline running all the benchmarks in our paper. Figure 20 plots the speedup of our bitwise operation. Even with random groups, we achieve 40% speedup, and our bitwise operation enjoys a better speedup with the outdegree-based GroupBy rules, i.e., 2.6×. Specifically, in random grouping, the maximum and minimum speedup is 2.5× of KG2 and 2% of FR. In comparison, the maximum and minimum speedup in GroupBy is 5.5× of KG1 and 15% of RD graph. Additional improvement from Groupby comes from the combined effect of high sharing ratio across grouped BFS instances and early termination enabled by bitwise traversal, which as a result allows concurrent instances to complete together and as early as possible.

In addition, we evaluate the impacts on the total number of global load transactions by bitwise traversal. Because bitwise status array consolidates the statuses of multiple (128 in our case) vertices into a single variable, we reduce the global load transactions of 1,024 BFS instances by 40%, i.e., from 53 to 38 million on average presented in Figure 21.

8.6 Comparison of State of the Art

We have implemented two CPU-based concurrent BFS, namely MS-BFS [26] and our own iBFS, both of which run 64 threads in total. In addition, we compare our GPU-based iBFS with B40C [29] and SpMM-BC [27]. B40C runs a single BFS instance on GPUs and has similar performance as the

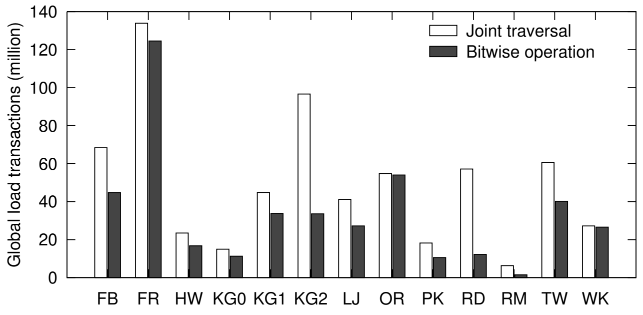


Figure 21: Total number of load transactions.

sequential or naive implementation presented in Figure 15, and SpMM-BC uses a simple GPU-based concurrent BFS to calculate betweenness centrality. Figure 22 presents the performance of all five implementations when running APSP on six different graphs.

CPU-based iBFS is significantly faster than MS-BFS thanks to the techniques such as GroupBy and early termination. The biggest speedup is achieved for the KG0 graph, where MS-BFS obtains 120 billion TEPS while our CPU-based iBFS reaches 397 billion TEPS. iBFS also achieves an average improvement of 45% on other five graphs. On GPUs, iBFS traverses on average 2× faster than SpMM-BC, and 19.3× than B40C. For graphs KG0 and HW, iBFS delivers about 700 and 300 TEPS respectively, greatly outperforming the other two implementations. Compared to the CPU-based implementation, GPU-based iBFS runs 2× faster on average across six different graphs.

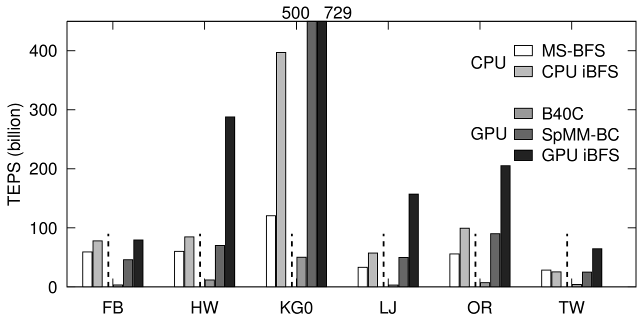


Figure 22: Comparison of CPU and GPU implementations.

8.7 Application: Reachability Index

To illustrate the broader application of iBFS on graph algorithms, in this paper we evaluate the benefits of using iBFS to construct the index for answering graph reachability queries, which computes the first k levels BFS for a large amount of selected vertices. Table 1 lists the runtimes of various implementations for constructing the index for 3-hop reachability. Clearly, GPU-based iBFS outperforms other concurrent BFS systems on four different graphs. Specifically, it is 21×, 3.3× and 2.2× faster than B40C, MS-BFS and CPU-based iBFS, respectively.

Dataset	CPU		GPU	
	MS-BFS	CPU-iBFS	B40C	GPU-iBFS
FB	19.2	16.5	302.8	14.3
KG0	1.85	0.56	2.9	0.31
OR	4.1	3.22	25.3	1.1
TW	2.1	2.7	27.8	0.9

Table 1: Runtime (hours) of 3-hop reachability index.

9. RELATED WORK

Our iBFS is closely related to CPU-based MS-BFS [26], which runs concurrent BFS by extending a single-threaded BFS, and compared to iBFS, underperforms on large graphs, e.g., it only achieves 10 billion TEPS on a graph with a billion of edges. In contrast, iBFS achieves more than 500 billion TEPS on a graph with billions of edges. Furthermore, the bitwise operation from MS-BFS cannot support early termination due to that it requires to reset the status array at each level, which leads to a much slower performance. Most notably, iBFS introduces a novel GroupBy strategy that improves the frontier sharing ratio dramatically and increases the benefit of concurrent traversal by another 2×.

Recent work has demonstrated that GPUs have great potentials in delivering high-performance breadth-first graph traversal [29,49]. One project SpMM-BC on regularized centrality [27] also extends the GPU-based BFS to concurrent BFS, but it does not support bottom-up BFS. On the other hand, the work [35] executes concurrent BFS to calculate the betweenness centrality [11] of a graph. However, each GPU in this work only executes a single BFS which is similar to the *naive* implementation in our work. In comparison, iBFS runs hundreds of BFS instances with one kernel and achieves on average 22× speedup compared to executing a single BFS on one GPU. Also, iBFS supports $\mathcal{O}(100)$ GPUs and achieves around 200 billion TEPS for graphs such as LJ and OR, significantly outperforming prior work [27,35].

In addition, our work is related to three types of shortest path algorithms [50–56], namely, Dijkstra, Bellman-Ford and Floyd-Warshall. The first two algorithms focus on SSSP while the later APSP. Specifically, Dijkstra [57] applies to weighted graph where weight must be positive, with the complexity of $\mathcal{O}(|E| + |V|\log|V|)$. Bellman-Ford [58] extends Dijkstra by allowing negative weighted edges but non-negative cycles. Floyd-Warshall [59] works for APSP [60] with the same constraint as Bellman-Ford algorithm. In contrast, our iBFS applies to all types of shortest path problems on a unweighted graph with the time complexity of $\mathcal{O}(i|V|) \sim \mathcal{O}(i|E|)$.

PHAST [61] runs multiple SSSP on GPU concurrently. However, this work only applies to road network graphs and suffers from small-world graphs which do not have good separators as mentioned in [62]. Two MSSP algorithms [63,64] mainly focus on special graphs, i.e., planar and embedded graphs, respectively. In contrast, our iBFS applies to both small-world and random graphs, and leverages the opportunity of frontier sharing to accelerate concurrent BFS.

10. CONCLUSION

In this work, we present iBFS, a new GPU-based concurrent BFS system which leverages a novel GroupBy strategy, combined with joint frontier queue and bitwise operations, to achieve high-performance concurrent breadth-first traversals. iBFS achieves unprecedented performance of over 57,000 billion TEPS on over 100 GPUs. As part of future work, we plan to explore additional optimizations for iBFS and study its application on a wide range of domains.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments and suggestions. This work was supported by NVIDIA and the National Science Foundation.

11. REFERENCES

[1] Stanley Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[2] David Knoke and Song Yang. *Social network analysis*, volume 154. Sage, 2008.

[3] Juanzi Li, Jie Tang, Jing Zhang, Qiong Luo, Yunhao Liu, and Mingcai Hong. Eos: expertise oriented search using social networks. In *Proceedings of the 16th international conference on World Wide Web*, pages 1271–1272. ACM, 2007.

[4] Hawoong Jeong, Bálint Tombor, Réka Albert, Zoltan N Oltvai, and A-L Barabási. The large-scale organization of metabolic networks. *Nature*, 407(6804):651–654, 2000.

[5] David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[6] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *High performance computing–HiPC*, pages 197–208. Springer, 2007.

[7] Hiroki Yanagisawa. A multi-source label-correcting algorithm for the all-pairs shortest paths problem. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10. IEEE, 2010.

[8] I-Lin Wang, Ellis L Johnson, and Joel S Sokol. A multiple pairs shortest path algorithm. *Transportation science*, 39(4):465–476, 2005.

[9] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of computer and system sciences*, 51(3):400–403, 1995.

[10] Abhinav Sarje and Srinivas Aluru. All-pairs computations on many-core graphics processors. *Parallel Computing*, 39(2):79–93, 2013.

[11] Ulrik Brandes. A faster algorithm for betweenness centrality*. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[12] Kamesh Madduri, David Ediger, Karl Jiang, David A Bader, and Daniel Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–8. IEEE, 2009.

[13] Paul W Olsen, Alan G Labouseur, and Jeong-Hyon Hwang. Efficient top-k closeness centrality search. In *International Conference on Data Engineering (ICDE)*, pages 196–207. IEEE, 2014.

[14] Hilmi Yildirim, Vineet Chaoji, and Mohammed J Zaki. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):276–284, 2010.

[15] James Cheng, Zechao Shang, Hong Cheng, Haixun Wang, and Jeffrey Xu Yu. K-reach: who is in your small world. *Proceedings of the VLDB Endowment*, 5(11):1292–1303, 2012.

[16] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *Proceedings of the SIGMOD International Conference on Management of data*, pages 813–826. ACM, 2009.

[17] Shlomi Dolev, Yuval Elovici, and Rami Puzis. Routing betweenness centrality. *Journal of the ACM (JACM)*, 57(4):25, 2010.

[18] Shilpa Mahajan and Jyoteesh Malhotra. Energy efficient path determination in wireless sensor network using bfs approach. *Wireless Sensor Network*, 3(11):351, 2011.

[19] Vladimir Ufimtsev and Sanjukta Bhowmick. Application of group testing in identifying high betweenness centrality vertices in complex networks. In *11th Workshop on Machine Learning with Graphs, KDD*, 2013.

[20] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Error and attack tolerance of complex networks. *nature*, 406(6794):378–382, 2000.

[21] Michael Bosse, Paul Newman, John Leonard, and Seth Teller. Simultaneous localization and map building in large-scale cyclic environments using the atlas framework. *The International Journal of Robotics Research*, 23(12):1113–1139, 2004.

[22] Shigeru Ichikawa. Navigation system and method for calculating a guide route, February 26 2002. US Patent 6,351,707.

[23] Angela Bonifati, Radu Ciucanu, and Aurélien Lemay. Learning path queries on graph databases. In *18th International Conference on Extending Database Technology (EDBT)*, 2014.

[24] Marc Najork and Janet L Wiener. Breadth-first crawling yields high-quality pages. In *Proceedings of the 10th international conference on World Wide Web*, pages 114–118. ACM, 2001.

[25] Xuesong Lu, Tuan Quang Phan, and Stéphane Bressan. Incremental algorithms for sampling dynamic graphs. In *Database and Expert Systems Applications*, pages 327–341. Springer, 2013.

[26] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment*, 8(4), 2014.

[27] Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. Regularizing graph centrality computations. *Journal of Parallel and Distributed Computing*, 2014.

[28] Ahmet Erdem Sariyuce, Erik Saule, Kamer Kaya, and Umit V Catalyurek. Hardware/software vectorization for closeness centrality on multi-/many-core architectures. In *International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 1386–1395. IEEE, 2014.

[29] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *ACM PPoPP*, 2012.

[30] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-driven versus topology-driven irregular computations on gpus. In *27th International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 463–474. IEEE, 2013.

[31] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *International Conference on Parallel*

Architectures and Compilation Techniques (PACT), pages 78–88. IEEE, 2011.

[32] S Beamer, K Asanovic, and D Patterson. Direction-optimizing breadth-first search. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10. IEEE, 2012.

[33] Hang Liu and H. Howie Huang. Enterprise: Breadth-first graph traversal on gpus. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[34] Nvidia. Nvidia kepler gk110 architecture whitepaper. 2013.

[35] Adam McLaughlin and David A Bader. Scalable and high performance betweenness centrality on the gpu. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 572–583. IEEE, 2014.

[36] Nvidia cuda c programming guide. *NVIDIA Corporation*, 2011.

[37] Intel Inc. Product brief intel xeon processor e7-8800/4800/2800 v2 product families. 2014.

[38] Intel Inc. Hpc code modernization workshop at lrz. 2015.

[39] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. Practical Recommendations on Crawling Online Social Networks. *JSAC*, 2011.

[40] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *WWW*, 2010.

[41] The University of Florida: Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[42] SNAP: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/.

[43] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.

[44] Graph500. http://www.graph500.org/.

[45] Green Graph500. http://green.graph500.org/.

[46] GTgraph: A suite of synthetic random graph generators. http://www.cse.psu.edu/~madduri/software/gtgraph/.

[47] Stéphane Bressan, Alfredo Cuzzocrea, Panagiotis Karras, Xuesong Lu, and Sadegh Heyrani Nobari. An effective and efficient parallel approach for random graph generation over gpus. *Journal of Parallel and Distributed Computing*, 73(3):303–316, 2013.

[48] Nvidia Profiler Tools. http://docs.nvidia.com/cuda/profiler-users-guide/.

[49] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *PPoPP*, 2011.

[50] Philipp Klodt, Gerhard Weikum, Srikanta Bedathur, and Stephan Seufert. *Indexing Strategies for Constrained Shortest Paths over Large Social Networks*. PhD thesis, 2011.

[51] Huiping Cao, K Selçuk Candan, and Maria Luisa Sapino. Skynets: Searching for minimum trees in graphs with incomparable edge weights. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1775–1784. ACM, 2011.

[52] Jun Gao, Jeffrey Xu Yu, Ruoming Jin, Jiashuai Zhou, Tengjiao Wang, and Dongqing Yang. Neighborhood-privacy protected shortest distance computing in cloud. In *Proceedings of the SIGMOD International Conference on Management of data*, pages 409–420. ACM, 2011.

[53] Weimin Yu, XINGQIN LIN, Wensheng Zhang, Julie McCann, et al. Fast all-pairs simrank assessment on large graphs and bipartite domains. 2014.

[54] Andy Diwen Zhu, Xiaokui Xiao, Sibo Wang, and Wenqing Lin. Efficient single-source shortest path and distance queries on large graphs. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 998–1006. ACM, 2013.

[55] Jianlong Zhong and Bingsheng He. Parallel graph processing on graphics processors made easy. *Proceedings of the VLDB Endowment*, 6(12):1270–1273, 2013.

[56] Jianlong Zhong and Bingsheng He. Medusa: A parallel graph processing system on graphics processors. *ACM SIGMOD Record*, 43(2):35–40, 2014.

[57] Roger Pearce, Maya Gokhale, and Nancy M Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE Computer Society, 2010.

[58] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *28th International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 349–359. IEEE, 2014.

[59] Hristo Djidjev, Sunil Thulasidasan, Guillaume Chapuis, Rumen Andonov, and Dominique Lavenier. Efficient multi-gpu computation of all-pairs shortest paths. In *28th International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 360–369. IEEE, 2014.

[60] Foto Afrati and Jeffrey Ullman. Matching bounds for the all-pairs mapreduce problem. In *Proceedings of the 17th International Database Engineering & Applications Symposium*, pages 3–4. ACM, 2013.

[61] Daniel Delling, Andrew V Goldberg, Andreas Nowatzyk, and Renato F Werneck. Phast: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.

[62] Edgar Solomonik, Aydin Buluc, and James Demmel. Minimizing communication in all-pairs shortest paths. In *27th International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 548–559. IEEE, 2013.

[63] Philip N Klein. Multiple-source shortest paths in planar graphs. In *SODA*, volume 5, pages 146–155, 2005.

[64] Sergio Cabello, Erin W Chambers, and Jeff Erickson. Multiple-source shortest paths in embedded graphs. *SIAM Journal on Computing*, 42(4):1542–1571, 2013.