Parallel Array-Based Single- and Multi-Source Breadth First Searches on Large Dense Graphs

Moritz Kaufmann
Technical University of Munich
kaufmanm@in.tum.de

Manuel Then
Technical University of Munich
then@in.tum.de

Alfons Kemper
Technical University of Munich
kemper@in.tum.de

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

ABSTRACT

One of the fundamental algorithms in analytical graph databases is breadth-first search (BFS). It is the basis of reachability queries, centrality computations, neighborhood enumeration, and many other commonly-used algorithms.

We take the idea of purely array-based BFSs introduced in the *sequential* multi-source MS-BFS algorithm and extend this approach to *multi-threaded single-* and *multi-source* BFSs. Replacing the typically used queues with fixed-sized arrays, we eliminate major points of contention which other BFS algorithms experience. To ensure equal work distribution between threads, we co-optimize work stealing parallelization with a novel vertex labeling. Our BFS algorithms have excellent scaling behavior and take advantage of multi-core NUMA architectures.

We evaluate our proposed algorithms using real-world and synthetic graphs with up to 68 billion edges. Our evaluation shows that the proposed multi-threaded single- and multi-source algorithms scale well and provide significantly better performance than other state-of-the-art BFS algorithms.

1. INTRODUCTION

Graphs are a natural abstraction for various common concepts like communication, interactions as well as friendships. Thus, graphs are a good way of representing social networks, web graphs, and communication networks. To extract structural information and business insights, a plethora of graph algorithms have been developed in multiple research communities.

At the core of many analytical graph algorithms are breadth first searches (BFSs). During a BFS, the vertices of a graph are traversed in order of their distance—measured in hops—from a source vertex. This traversal pattern can for example be used to do shortest path computations, pattern matchings, neighborhood enumerations, and centrality calculations. While all these algorithms are BFS-based, many different BFS variants have been published. Important aspects that differentiate BFS variants are their *degree of parallelism*, the *number of sources* they consider, and the *type of graph* they are suited for.

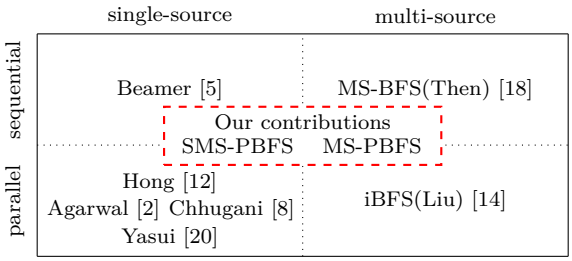

Figure 1: State-of-the-art single-server breadth-first-search publications

Possible degrees of parallelism include single-threaded and multi-threaded execution as well as distributed processing. Many emerging systems, e.g., Pregel [16], Spark [21], and GraphLab [15], focus heavily on distributed processing, but often neglect to optimize for the single-machine use case. However, especially for graph analytics, distributed processing is a very hard problem. The main reason for this is the high communication cost between compute nodes which is directly influenced by the inherent complexity of graph partitioning [4]. While there are cases in which distribution cannot be avoided, we argue that in graph analytics it is often done unnecessarily, leading to diminished performance. Actually, most—even large-scale—real-world graphs easily fit into the main memory of modern server-class machines [11]. Thus, we only consider the single-node scenario but differentiate between single and multi-threaded processing.

In Figure 1 we give an overview of the multi-threaded single-node state-of-the-art BFS algorithms.

The figure also includes the second important aspect of a BFS variant: its number of sources. Traditionally, the BFS problem is stated as traversing the graph from a single source vertex. While this single-source model can be applied to any BFS-based algorithm, it hampers inter-BFS optimizations. Specialized multi-source BFS algorithms like MS-BFS [18] and the GPU based iBFS [14] concurrently traverse the graph from multiple source vertices and try to share common work between the BFSs. This is, for example beneficial when the all pairs shortest path (APSP) problem needs to be solved as it is the case for the closeness centrality metric.

For this computation, a full BFS is necessary from every vertex in the graph. Considering that small-world networks often consist of a single large connected component, a single-source BFS would visit every vertex in each traversal while a multi-source-optimized BFS batches visits where possible.

One central limitation of current multi-source BFS algorithms is their limited ability to analyze large graphs efficiently. The GPU-based iBFS is limited to the memory available on GPU cards which is over an order of magnitude less than what is available in modern servers. The CPU-based MS-BFS on the other hand is sequential; utilizing all cores would require an separate BFS instance for each CPU core. It can only speed up analysis when a huge number of sources is analyzed and it requires much more memory due to the separate BFS states.

In this paper we propose two breadth-first search algorithms that are optimized for modern massively-parallel multi-socket machines: SMS-PBFS and MS-PBFS. *SMS-PBFS* is a parallelized single-source BFS, while *MS-PBFS* is a parallelized multi-source BFS. Both algorithms are based on the approaches introduced by the sequential MS-BFS. By adding scalable parallelization we enable massive speedups especially when working with a limited number of sources. Our approach also significantly reduces memory requirements for parallelized multi-source BFSs.

Our evaluation using real-world graphs as well as artificial graphs, including the industry-standard benchmark Graph500 [1], shows that SMS-PBFS and MS-PBFS greatly outperform the existing state-of-the-art BFS algorithms. Because the overhead for parallelization is negligible, our parallelized algorithms can be efficiently used for sequential BFS traversals without modifications.

Specifically, the contributions of this paper are as follows:

- We present the *MS-PBFS* algorithm, a multi-core NUMA-aware multi-source BFS that ensures full machine utilization even for a limited number of sources. We also introduce *SMS-PBFS*, a multi-core NUMA-aware single-source BFS based on MS-PBFS that shows better performance than existing single-source BFS algorithms.
- We introduce a new vertex labeling scheme that is both cache-friendly as well as skew-avoiding.
- We propose a parallel low-overhead work stealing scheduling scheme that preserves NUMA locality in BFS workloads.

The latter two contributions can also boost the performance of existing BFS algorithms as well as other graph algorithms.

The paper is structured as follows. In Section 2 we describe the state-of-the-art BFS algorithms for the sequential and parallel single-source case as well as for the sequential multi-source case and summarize their limitations. Afterward, in Section 3 we present our novel algorithms MS-PBFS and SMS-PBFS. In Section 4 we describe our optimized scheduling algorithm, vertex labeling scheme, and memory-layout for modern NUMA architectures. Section 5 contains the evaluation of our algorithms. We give an overview over the related work in Section 6. Section 7 summarizes our findings.

2. BACKGROUND

In this section we describe the current state-of-the-art BFS algorithm variants. We focus on algorithms that operate on undirected, unweighted graphs. Such a graph is represented by a tuple $G = \{V, E\}$, where V is the set of vertices and $E = \{neighbors_v | v \in V\}$ where $neighbors_v$ is the set of neighbors of v. Additionally, we assume that the graphs of interest are *small-world networks* [3], i.e., that they are strongly connected and their number of neighbors per vertex follows a power law distribution. This is the case for most real-world graphs; examples include social networks, communication graphs and web graphs.

Given a graph G and a source vertex s, a BFS traverses the graph from s until all reachable vertices have been visited. During this process, vertices with a one-hop distance from s are visited first, then all vertices with distance two and so on. Each distance corresponds to one iteration. While executing an iteration the neighbors of vertices that were newly discovered in the previous iteration are checked to see if they have not yet been discovered. If so, they are marked as newly seen and enqueued for the next iteration. Consequently, the basic data structures during execution are a queue of vertices that were discovered in the previous iteration and must be processed in the current iteration, called *frontier*, a mapping *seen* that allows checking if a vertex has already been visited, and a queue *next* of vertices that were newly discovered in the current iteration. The latter queue is used as input for the next iteration. The number of BFS iterations corresponds to the maximum distance of any vertex from the source. It is bound by the diameter of the graph, i.e., the greatest shortest distance between any two vertices.

Our novel MS-PBFS and SMS-PBFS algorithms build on multiple existing techniques which we introduce in the following. We categorize the presented algorithms as either parallel or sequential, and as either single-source or multi-source as shown in Figure 1. To the best of our knowledge, each of the presented algorithms in this chapter is the current single-server state-of-the art in its category.

2.1 Sequential and Parallel Single-Source BFS

The fastest sequential single source BFS algorithm for dense graphs was presented by Beamer et al. [5]. It breaks up the algorithmic structure of the traditional BFS to especially reduce the amount of work required to analyze small-world networks. In such graphs most vertices are reached within few iterations [18]. This has the effect that at the end of this "hot phase" the frontier for the next iteration contains many more vertices than there are unseen vertices in the graph, as most were already discovered. At this point the classical *top-down* approach — trying to find unseen vertices by processing the *frontier* — becomes inefficient. Most vertices' neighbors will already have been seen but would still need to be checked. The ratio of vertices discovered per traversed edge becomes very low. For these cases Beamer et al. propose to use a *bottom-up* approach and iterate over the vertices that were not yet seen in order to try to find an already seen vertex in their neighbor lists. Even though the result of the BFS is not changed, this approach significantly reduces the number of neighbors that have to be checked. This translates into better traversal performance, thus, this approach is often used in more specialized BFS algorithms [2, 18, 20].

Those algorithmic changes also have implications on the BFS data structures that can be used. Typically, the queues in a BFS are implemented using either a dense bitset or a sparse vector. The bottom-up phase, though, requires efficient lookups of vertices in the queue, thus, it can not be used

efficiently with a sparse vector. The original authors solve this by converting the data structures from bitset to sparse vector when switching from top-down to bottom-up or vice versa.

For *parallelization*, this approach can be combined with existing work on scalable queues for BFSs [2, 12, 8, 5, 20] and on scalability on multi-socket NUMA architectures [19, 8] through static partitioning of vertices and data across NUMA nodes. Many of these techniques are combined in the BFS algorithm proposed by Yasui et al. [20, 19] which is the fastest multi-threaded single-source BFS.

2.2 Sequential Multi-Source BFS

The MS-BFS algorithm [18] is targeted towards multi-source traversal and further reduces the total number of neighbor lookups across all sources compared to Beamer et al. It is based on two important observations about BFS algorithms. Firstly, regardless the data structure used, for sufficiently large graphs it is expensive to check whether a vertex is already contained in *seen* as CPU cache hit rates decrease. For this very frequent operation even arrays with their containment check bound of $O(1)$ are bound by memory latency. This problem is further exacerbated on non-uniform memory access (NUMA) architectures that are common in modern server-class machines. Secondly, when multiple BFSs are run in the same connected component, every vertex of this component is visited *separately* in each BFS. This leads to redundant computations, because whenever two or more BFS traversals in the same component find a vertex v in the same distance d from their respective source vertices, the remainder of those BFS traversals from v will likely be very similar, i.e., visit most remaining vertices in the same distance.

MS-BFS alleviates some of these issues by optimizing for the case of executing multiple independent BFSs from different sources in the same graph. It uses three k-wide bitsets to encode the state of each vertex during k concurrent BFSs:

1. $seen[v]$, where the bit at position i indicates whether v was already seen during the BFS i,
2. $frontier[v]$, determining if v must be visited in the current iteration for the BFSs, and
3. $next[v]$, with each set bit marking that the vertex v needs to be visited in the following iteration for the respective BFS.

For example given $k = 4$ concurrent BFSs, the bitset $seen[v] = (1, 0, 0, 1)$ indicates that vertex v is already discovered in BFSs 0 and 3 but not in BFSs 1 and 2. Using this information, a BFS step to determine *seen* and *next* for all neighbors n of v can be executed using the bitwise operations *and* (&), *or* (|), and *negation* ($\sim$):

for each $n \in neighbors[v]$
$next[n] \leftarrow next[n] \mid (frontier[v] \ \& \ \sim seen[n])$
$seen[n] \leftarrow seen[n] \mid frontier[v]$

Here, if n is not yet marked seen for a BFS and this BFS's respective bit is set in $frontier[v]$, then the vertex n is marked as seen and must be visited in the next iteration. The bitwise operations calculate *seen* and *next* for k BFSs at the same time and can be computed efficiently by leveraging the wide registers of modern CPUs. A full MS-BFS iteration consists of executing these operations for all vertices v in the graph. Note that all k BFSs are run concurrently on a single CPU core with their traversals implicitly merged whenever possible.

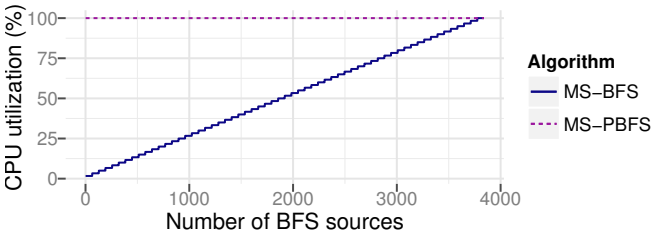

Figure 2: CPU utilization of MS-BFS and MS-PBFS as the number of sources increases.

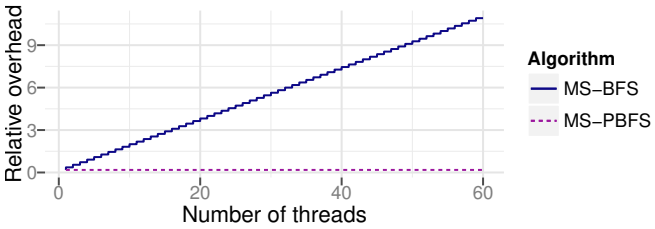

Figure 3: Relative memory overhead compared to graph size as number of threads increases.

MS-BFS works for any number of concurrent BFSs using bitset sizes chosen accordingly. However, it is especially efficient when the vertex bitsets have a width for which the target machine natively supports bit operations. Modern 64-bit x86 CPUs do not only have registers and instructions that support 64 bit wide values, but also ones for 128 bit and 256 bit using the SSE and AVX-2 extensions, respectively. The original publication elaborates on the trade-offs of various bitset widths and how they influence the algorithm's performance.

In Section 3 we show how MS-BFS can be efficiently parallelized and present an optimized variant that is highly efficient for single source traversals.

2.3 Limitations of Existing Algorithms

The MS-BFS algorithm is limited to sequential execution. The only way to saturate a multi-core system is to run a separate MS-BFS instance on each core. However, if the number of BFS sources is limited, e.g., to only 64 as in the Graph500 benchmark, MS-BFS can only run single-threaded or, at best, on few cores. In such cases, the capabilities of a multi-core system cannot be fully utilized. Figure 2 analyzes this problem using a 60-core machine and 64 concurrent BFSs per MS-BFS. Every 64 sources one more thread can be used. Hence, only with 3840 or more sources all cores are utilized. Also, by running multiple sequential instances simultaneously, the memory requirements rise drastically to the point that the dynamic state of the BFSs require much more memory than the graph itself. This is demonstrated in Figure 3. It compares the memory required for the MS-BFS and our proposed MS-PBFS data structures to the size of the analyzed graph. We calculated the memory requirement based on 16 edges per vertex like the Kronecker graphs in the Graph500 benchmark. While traditional BFSs only require a fraction of the graph memory for their working set, MS-BFS

Listing 1: Top-down MS-BFS algorithm from [18].

```
1  for each v ∈ V
2     if frontier[v] = ∅, skip
3     for each n ∈ neighbors_v
4        next[n] ← next[n] | frontier[v]
5
6  for each v ∈ V
7     if next[v] = ∅, skip
8     next[v] ← next[v] & ∼seen[v]
9     seen[v] ← seen[v] | next[v]
10    if next[v] ≠ 𝔹_∅
11       v is found by BFSs in next[v]
```

already requires more memory than the graph using only 6 threads. With 60 threads it requires over 10 times more memory! Hence, more than one terabyte of main memory would be needed to analyze a 100GB graph using all cores. An alternative could be to use smaller batch sizes, thus, requiring fewer sources and memory to take advantage of all cores. However, that would decrease the traversal performance as less work can be shared between the BFSs. In contrast, the parallel multi-source algorithm MS-PBFS proposed in this paper can use all cores at 64 BFSs and only consumes as much memory as a single MS-BFS.

State-of-the-art parallel single-source algorithms are limited by either locality and scalability issues associated with the sparse queues. Even if partitioned at NUMA socket granularity there can be a lot of contention and the trend of having more cores per CPU socket does not work in such approaches favor.

3. PARALLEL SINGLE- AND MULTI-SOURCE BFS

In this section we present our *parallelized* multi-source BFS algorithm as well as a single-source BFS variant, both designed to avoid those problems.

3.1 MS-PBFS

In the following we introduce MS-PBFS, a parallel multi-source BFS algorithm that ensures full machine utilization even for a single multi-source BFS.

MS-PBFS is based on MS-BFS and parallelizes both its top-down (Section 3.1.1) and its bottom-up (Section 3.1.2) variant. Our basic strategy is to parallelize all loops over the vertices by partitioning them into disjunct subsets and processing those in parallel. State that is modified and accessed in parallel then has to be synchronized to ensure correctness.

3.1.1 Top-down MS-PBFS

MS-BFS uses a two-phase top-down variant, shown in Listing 1. As described in the Section 2, each value in *seen*, *frontier* and *next* is not a single boolean value but a bitset. The first phase, lines 1 through 4, aggregates information about which vertices are reachable in the current iteration. After it finishes, the second phase, the loop in lines 6 through 11, identifies which of the reachable vertices are newly discovered and processes them.

Our strategy is to parallelize both of these loops and separate them using a barrier. During this parallel processing, the first loop accesses the fixed-size *frontier*, *neighbors* and *next* data structures. As the former two are constant during

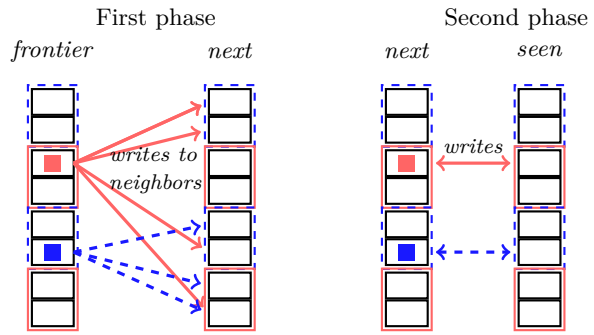

Figure 4: Concurrent top-down memory accesses

the loop accesses, they do not require any synchronization. The *next* data structure on the other hand is updated for each neighbor n by combining n's *next* bitset with the currently visited v's *frontier*. As vertices in general can be reached via multiple edges from different vertices, different threads might update *next* simultaneously for a vertex. To avoid losing information in this situation, we use an atomic compare and swap (CAS) instruction, replacing line 4 with the following:

```
do
   oldNext ← next[n]
   newNext ← oldNext | frontier[v]
while atomic_cas(next[n], oldNext, newNext)
```

Bitsets wider than the CPU's largest atomic operation value type can be supported by implementing the update operation as a series of independent atomic CAS updates of each sub-part of the bitset. For example a 512-bit bitset could be updated using eight 64-bit CAS as described above. This retains the desired semantics as the operation can only add bits but never unset them. It is also not required to track which thread first added which bit as the updates of the newly discovered vertices is only done in the second phase.

The second phase iterates over all vertices in the graph and updates *next* and *seen*. In contrast to the first phase, no two worker threads can access the same entry in the data structures. Regardless of how the vertex ranges are partitioned, there is a bijective mapping between a vertex, the accessed data entries, and the worker that processes it. Consequently, there cannot be any conflicts, thus, no synchronization is necessary.

Figure 4 visualizes the memory access patterns for all writes in the first and second phase of the top-down MS-PBFS algorithm. The example shows a configuration with two parallel workers and a task size of two. Squares show the currently active vertices and arrows point to entries that are modified. The linestyle of the square shows the association to the different workers. We come back to this figure in Section 4.4 to further explain the linestyles.

To reduce the time spent between iterations, we directly clear each *frontier* entry inside the second parallelized loop. This allows MS-PBFS to re-use the memory of the current *frontier* for *next* in the subsequent iteration without having to clear the memory separately. Thus, we reduce the algorithm's memory bandwidth consumption.

Furthermore, we only update *next* entries if the computation results in changes to the bitset. This avoids unnecessary writes and cache line invalidations on other CPUs [2].

Listing 2: Bottom-up MS-BFS traversal from [18].

```
1 for each u ∈ V
2   if |seen[u]| = |S|, skip
3   for each v ∈ neighbors_u
4     next[u] ← next[u] | frontier[v]
5   next[u] ← next[u] & ∼seen[u]
6   seen[u] ← seen[u] | next[u]
7   if |next[u]| ≠ 0
8     u is found by BFSs in next[u]
```

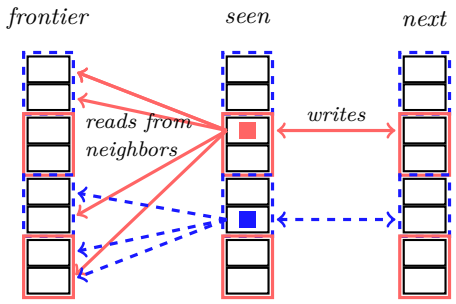

Figure 5: Concurrent bottom-up memory accesses

3.1.2 Bottom-up MS-PBFS

As explained in Section 2.1, a BFS's bottom-up variant linearly traverses the *seen* data structure to find vertices that have not been marked yet. For every vertex v that is not yet seen in all concurrent BFSs, MS-BFS's bottom-up variant checks whether any of its neighbors was already seen in the respective BFS. If so, v is marked as seen and is visited in the next iteration. We show the full bottom-up loop in Listing 2.

MS-PBFS parallelizes this loop by splitting the graph into distinct vertex ranges which are then processed by worker threads. Inside the loop, the current iteration's *frontier* is only read. Both *seen* and *next* are read as well as updated. Similar to the second phase described in the previous section, there is a bijective mapping between each updated entry and the worker that processes the respective vertex. Consequently, there cannot be any read-write or write-write conflicts and, thus, no synchronization is required within the ranges. Figure 5 depicts the bottom-up variant's memory access pattern, again for two parallel workers.

Once all active BFSs bits are set in *next* we stop checking further neighbors to avoid unnecessary read operations. This check is also used in the original bottom-up algorithm by Beamer et al.

3.2 Parallel Single-Source: SMS-PBFS

In order to also apply our novel algorithm to BFS that traverse the graph from only a single source, we derive a single-source variant: SMS-PBFS. SMS-PBFS contains two main changes: the values in each array are represented by boolean values instead of bitsets, and checks that are only required when multiple BFS are bundled can be replaced by constants. This allows us to simplify the atomic update in the top-down algorithm, as a single atomic write is sufficient, instead of a compare and swap loop. The SMS-PBFS top-down and bottom-up algorithms are shown in Listing 3 and 4, respectively. Parallel coordination is only required when scheduling the vertex-parallel loops and during the single

Listing 3: Single-source parallel top-down algorithm

```
1  parallel for each v ∈ V
2    if not(frontier[v]), skip
3    for each n ∈ neighbors_v
4      if not(next[n]), atomic(next[n] ← true)
5    frontier[v] ← false
6
7  parallel for each v ∈ V
8    if not(next[v]),  skip
9    next[v] ← not(seen[v])
10   if not(seen[v])
11     seen[v] ← true
12     v is found
```

Listing 4: Single-source parallel bottom-up algorithm

```
1  parallel for each u ∈ V
2    if seen[u]
3      next[u] ← false
4    else
5      for each v ∈ neighbors_u
6        if frontier[v]
7          next[u] ← true
8          break
9      if next[v]
10           seen[u] ← true
11           u is found
```

atomic update in the first top-down loop.

While MS-PBFS always has to use an array of bitsets to implement *next*, *frontier* and *seen*, there is more freedom when implementing SMS-PBFS. It is still restricted to using dense arrays, but each entry can either be a bit, a byte or a wider data type. In the parallel case, where the state of 512 vertices fits into one 64-byte CPU cache line using bit representation, the chance of concurrent modification is very high. Choosing a larger data type allows to balance cache efficiency and reduced contention between workers. We demonstrate these effects in our evaluation. To reduce the number of branches, we try to detect when a consecutive range of vertices is not active in the current iteration and skip it. Instead of checking each vertex individually we check ranges of size 8 bytes, which can efficiently be implemented on 64-bit CPUs. Using a bit representation, each such range contains the status of 64 vertices. If no bit is set, we directly jump to the next chunk and save a large number of individual bit checks. Otherwise, each vertex is processed individually. This is similar to the *Bitsets-and-summary* optimization [19] but does not require an explicit summary bit.

4. SCHEDULING AND PARALLEL GRAPH DATA STRUCTURES

The parallelized algorithms' descriptions in Section 3 focus on how to provide semantical correctness. It leaves out the implementation details of how to actually partition the work to workers and how to store the BFS data structures as well as the graph. As shown by existing work on parallel single-source BFSs, these implementation choices can have a huge influence on the performance of algorithms that are intended to run on multi-socket machines with a large number of cores. In this section we describe the data structures and memory

Figure 6: Visited neighbors per worker during a BFS using static partitioning on a social network graph with different vertex labelings

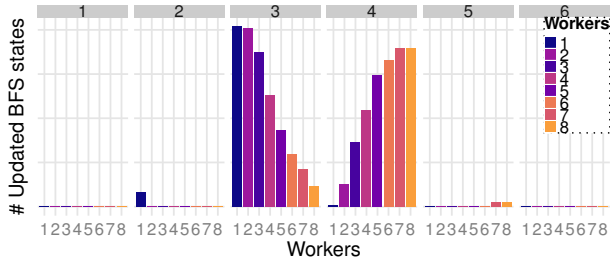

Figure 7: Updated BFS vertex states per worker per iteration during a BFS using static partitioning on a social network graph with ordered vertex labeling

organization to efficiently scale MS-PBFS and SMS-PBFS (together abbreviated as (S)MS-BFS) on such machines.

4.1 Parallelization Strategies

The initial version of our parallelized algorithms used popular techniques from state-of-the-art parallel single-source implementations. Specifically, it used static partitioning of vertices to workers, and degree-ordered vertex labeling[19]. With this labeling scheme, we re-labeled the graph's vertices and assigned dense ids in the order of the vertices' degrees, with the highest-degree vertex getting the smallest id. That way, the states of high degree vertices are located close together which improves cache hit rates. This first implementation showed very good performance for a small number of workers. However, at higher thread counts, the overall scalability was severely limited.

Together with static partitioning, degree based labeling has the effect that due to the power-law distribution of vertex degrees in many real world graphs, the vertices in the first partitions have orders of magnitude more neighbors than those in later partitions. We visualize this effect in Figure 6. In that experiment, the first worker processes the first $\frac{1}{8}$th of the vertices in the graph, the second worker the second $\frac{1}{8}$th, and so on. As the amount of work per partition increases with the number of neighbors that need to be visited, the described skew directly affects the workers' runtime, as it is one of the most costly operations besides updating the BFS vertex state. While it may be possible to create balanced static partitions such that each worker has to do the same amount of work across all BFS iterations, it is not enough to significantly increase utilization. The problem would then be that in different iterations different parts of the graph are active, thus, there would still be a large runtime skew in each iteration. The different workload for workers across iterations is shown in Figure 7 using the number of updated BFS vertex states as an indicator for the actual amount of work.

Additionally, this figure gives an indication why dynamic work assignment does not ensure full utilization on its own. Intuitively, in a small-world network an average BFS traverses the graph starting from the source vertex to the vertices with the highest degrees, because these are well-connected, and from there to the remainder of the graph. For such a BFS, the high-degree vertices are typically discovered after two to three iterations as shown in Figure 7. There in iteration two only a tiny fraction of vertices is updated. On the other hand as these are the high-degree vertices a lot of undiscovered vertices are reachable from them, resulting in a huge number of updates in iteration three. The updates themselves, which are processed in the second phase of the top-down algorithm could be well distributed across workers. Identifying the newly reachable vertices, which is done in the first phase, by searching the neighbors of the high-degree vertices is more challenging to schedule because there are only few and due to the labeling they are all clustered together. In combination, this iteration is very expensive but a large part of the work is spent when processing very few high-degree vertices. To achieve even utilization, tiny task sizes would be required. Such tiny tasks mean, however, that the scheduling overhead would become so significant that the overall performance would not improve.

Instead, our design relies on two strategies. We use fine-granular tasks together with work-stealing to significantly reduce the number of vertices that are assigned at once and enable load-balancing between the cores. We also use a novel vertex labeling scheme that is scheduling-aware and distributes high-degree vertices in such a way that they are both clustered but also spread across multiple tasks. This allows us to avoid the use of tiny task ranges.

4.2 Parallel Vertex Processing

In this section we focus on providing a parallelization scheme that minimizes synchronization between threads and balances work between nodes to achieve full utilization of all cores during the whole algorithm.

Our concept allows load balancing through work stealing with negligible synchronization overhead.

4.2.1 Task creation and assignment

Efficient load balancing requires tasks to have two related properties: there need to be many tasks and their runtime needs to be relatively short compared to the overall runtime. If there were only two tasks on average per thread, a scenario is very probable where a slow thread only starts its last task when all other threads are already close to being finished with their work. This creates potential to have all other threads idling until this last thread is finished. Given a fixed overall runtime, the shorter the runtime of each work unit and the more tasks are available, the easier it becomes to get all threads to finish at approximately the same time. On the other hand, if the ranges are very small, the threads have to request tasks more often from the task pool. This can lead to contention and, thus, decrease efficiency because more processing time is spent in scheduling instead of doing actual work.

Due to the fixed-size *frontier* and *next* arrays which span

Listing 5: Task creation algorithm: create_tasks

```
1  Input: queueSize, splitSize, numThreads
2  workerTasks ← ∅
3  curWorker ← 0
4  for(offset = 0; offset < queueSize; offset+ = splitSize)
5      wId ← curWorker mod numThreads
6      range ← {offset, min(offset + splitSize, queueSize)}
7      workerTasks[wId] ← workerTasks[wId] ∪ range
8      curWorker ← curWorker + 1
9  taskQueues ← ∅
10 for i = 1, . . . , num_threads
11     taskQueues[i] ← {|workerTasks[i]|, workerTasks[i]}
12 return taskQueues
```

Listing 6: Task retrieval algorithm: fetch_task

```
1  Input: taskQueues, workerId
2  offset ← 0
3  do
4      i ← (threadId + offset )mod |taskQueues|
5      taskId ← fetch_add_task_ix(taskQueues[i], 1)
6      if taskId < num_tasks(taskQueues[i])
7          return get_task(taskQueues)[i]
8      else
9          offset ← offset + 1
10 while offset < |taskQueues|
11 return empty_range()
```

all the vertices no matter how many of them are actually enqueued, all parallel loops of (S)MS-PBFS can follow the same pattern: a given operation has to be executed for all vertices in the graph. To create the tasks we divide the list of vertices into small ranges. In our experiments we found that task range sizes of 256 or more vertices do not have any significant scheduling overhead (below 1% of total runtime) for a graph with more than one million vertices. With about 3900 tasks in such a graph there are enough tasks to load balance even machines with hundreds of cores.

For work assignment we do not use one central task queue, but similar to static partitioning give each worker its own queue. When a parallelized loop over the vertices of the graph is executed, the queues are initialized using the *create_tasks* function shown in Listing 5. Each task queue $taskQueues[i] = \{curTaskIx, queuedTasks\}$ belonging to worker i consists of an index *curTaskIx* pointing to the next task and a list of tasks *queuedTasks*. The number of vertices per task is controlled by the parameter *splitSize*. We use a round-robin distribution scheme, so the difference in queue sizes can be at most one task.

4.2.2 Work stealing scheduling

The coordination of workers during task execution is handled by the lock-free function *fetch_task*, which is shown in Listing 6. The function can be kept simple due to the fact that during a phase of parallel processing no new tasks need to be added. Only after all tasks have been completed the next round of tasks is processed—e.g., a new iteration or the second phase of the top-down algorithm.

Initially, each worker fetches tasks from its own, local queue which is identified using the *workerId* parameter. It atomically fetches and increments the current value of *curTaskIx* as shown in line 5. Using modern CPU instructions, this can be done without explicit locking. If the task id is within the bounds of the current task queue (line 7), the corresponding task range is processed by the worker. Otherwise, it switches to the next worker's task queue by incrementing the task queue offset, and tries again to fetch a task. This is repeated until either a task is found or, alternatively, after all queues have been checked, an empty task range is returned to the worker to signal that no task is available anymore. Further optimizations like remembering the task queue index where the current task was found and resuming from that offset when the next task is fetched, guarantee that every worker skips each queue exactly once. Incrementing the *curTaskIx* only if the queue is not empty avoids atomic writes which could lead to cache misses when other workers are visiting

Listing 7: Parallelized *for* loop

```
1  tasks ← create_tasks(|V|, splitSize, |workers|)
2  run on each w ∈ workers
3      workerId ← getWorkerId()
4      while((range ← fetch_task(tasks, workerId)) ≠ ∅)
5          for each v ∈ range
6              {Loop body}
7  wait_until_all_finished(workers)
```

that queue.

Listing 7 shows how the task creation and fetching algorithms can be combined to implement the parallel *for* loop which is used to replace the original sequential loops in the top-down and bottom-up traversals. Here, *workers* is the set of parallel processors. In line 2 all workers are notified that new work is available and given the current *task* queues. Each worker fetches tasks and loops over the contained vertices until all tasks are finished. Once a worker can not retrieve further tasks it signals the main thread, which waits until all workers are finished.

As long as a worker only fetches from its own queue, the task retrieval cost is minimal—mostly only an atomic increment which is barely more expensive than a non-atomic increment on the x86 architecture[17]. Even when reading from remote queues, our scheduling has only minimal cost that mostly results from reading one value if the respective queue is already finished, and one write when fetching a task. This is negligible compared to the normal BFS workload of at least one atomic write per vertex in the graph. The construction cost of the initial queues in the *create_tasks* function is also barely measurable and could be easily parallelized if required.

4.3 Striped vertex order

In the introduction of Section 4.1, we discussed that the combination of multi-threading, degree ordered labeling and array-based BFS processing leads to large skew between worker runtimes. As (S)MS-PBFS's top-down algorithms is designed for multi-threading and requires efficient random-access to the frontier, neither the threading model, nor the backing data structure must be changed. Thus, though our single-threaded benchmarks confirmed that the increased cache locality achieved through degree-ordered labeling leads to significantly shorter runtimes compared to random vertex labeling, we cannot use degree-ordered labeling.

Instead, we propose a cache-friendly semi-ordered approach: We distribute degree-ordered vertices in a round robin fashion across the workers' task ranges. The highest-

degree vertex is labeled such that it comes at the start of the first task of worker one. The second-highest degree vertex is labeled such that it comes at the start of the first task of worker two, etc. This round robin distribution is continued until all the first tasks for the workers are filled. Then, all the second tasks, and so on, until all vertices are assigned to the workers. Using this approach we still cannot guarantee that all task ranges have the same cost, but we can control that the cost of the ranges in each task queue are approximately the same per worker. Also, because the highest degree vertices are assigned first, the most expensive tasks will be executed first. Having small task sizes at the end has the advantage of reducing wait times towards the end of processing when no tasks are available for stealing anymore. The pre-computation cost of this *striped vertex labeling* is similar to that of degree-ordered labeling.

4.4 NUMA Optimizations

Our (S)MS-PBFS algorithms, as described above, scale well on single-socket machines. When running on multi-socket machines, however, the performance does only improve insignificantly, even though the additional CPUs' processing power is used. The main problems causing this are twofold. First, if all data is located in a single socket's local memory, i.e., in a single NUMA region, reading it from other sockets can expose memory bandwidth limitations. Second, writing data in a remote NUMA region can be very expensive [8]. This leads to a situation where the scalability of seemingly perfectly parallelizable algorithms is limited to a single socket. In the following, we describe (S)MS-PBFS optimizations that substantially improve the algorithms' scaling on NUMA architectures.

In our (S)MS-PBFS algorithms it is very predictable which data is read, particularly within a task range. Consider a bottom-up iteration as described in Section 3.1.2. When processing a task, it updates only the information of vertices inside that task. We designed our algorithm with the goals of not only providing NUMA scalability but *also* of avoiding any overhead from providing this scalability. We deterministically place memory pages for all BFS data structures, e.g., for *seen*, in the NUMA region of the worker that is assigned to vertices contained in the memory page. Further, we pin each worker thread to a specific CPU core so that it is not migrated during traversals. The desired result of this pinning is visualized in Figures 4 and 5. In addition to the figures' already discussed elements, we use the linestyle to encode the NUMA region of both the data and the workers. The memory pages backing the arrays are interleaved across the NUMA nodes at exactly the task range borders—for the example shown in the figures there are two vertices per task—and the workers only process vertices with data on the same NUMA node except for stolen tasks.

We calculate the mapping of vertices to memory pages and the size of task ranges as follows. Consider that a 64 bit wide bitset is used per *seen* entry, and memory pages have a common size of 4096 bytes. In this example, the task range size should be a multiple of $\frac{pageSize}{bitsetSize/8} = 512$ vertices. Given a task range, it is straightforward to calculate the memory range of the data belonging to the associated vertices.

Because we initialize large data structures like *seen*, *frontier*, and *next* only once at the beginning of the BFS and use them across iterations, we need to make sure that the data is placed deterministically, and that tasks accessing the same vertices are scheduled accordingly in all iterations. Thus, work stealing must not occur during the parallel initialization of the data structures to ensure proper initial NUMA region assignments of the memory pages.

When the BFS tasks only update memory regions that were initialized by themselves, we achieve NUMA locality. Note that even though our work stealing scheduling approach results in additional flexibility regarding task assignment, most tasks are still executed by their originally assigned workers when the total runtime for the tasks in each queue is balanced.

While this goal is perfectly attainable for the bottom-up variant and the second loop of top-down processing, we cannot efficiently predict which vertex information is updated in the first top-down loop. Besides processing stolen tasks, this is the *only* part of our algorithm in which non-local writes can happen.

In summary, given that each worker thread initializes the memory pages that correspond to the vertex ranges it is assigned to, nearly all write accesses, except for the first top-down loop and the work stolen from other threads, are NUMA-local. (S)MS-PBFS further guarantee that the share of memory located in each NUMA region is proportional to the share of threads that belong to that NUMA region. If, for example, 8 threads are located in NUMA region 0 and 2 threads are located in NUMA region 1, 80% of the memory required for the BFS data structures are located in region 0 and 20% will be in region 1.

Similar to the NUMA optimizations of the BFS data structures, also the graph storage can be optimized. We minimize cross-NUMA accesses by allocating the neighbor lists of the vertices processed in each task range on the same NUMA node as the worker which the task is assigned to. By using the same vertex range assignment while loading the neighbor lists and during BFS traversal, we ensure that all the data entries for each vertex are co-located. This principle is similar to the G^B partitioning described by Yasui et al. [19], which, however, uses static partitioning with one partition per NUMA node.

5. EVALUATION

In our evaluation we analyze four key aspects:

- What influence do the different labeling schemes have?
- How does the sequential performance of SMS-PBFS's algorithmic approach compare to Beamer's direction-optimizing BFS?
- How effectively does it scale both in terms of number of cores and dataset size?
- How does it compare to MS-BFS and the parallel single-source BFS by Yasui et al.?

In addition to the MS-PBFS and SMS-PBFS algorithms described before, we test two more variants. *MS-PBFS (sequential)* is our novel MS-PBFS algorithm run the same way as MS-BFS: a single instance per core requiring multiple batches to be evaluated in parallel. This tests the impact of the early exit optimization in the bottom-up phase, as well as our optimized data-structures. Another variant, *MS-PBFS (one per socket)* runs one parallel multi-source BFS per CPU socket using MS-PBFS. We use the performance of this variant to determine the cost of parallelization across all NUMA

nodes when using MS-PBFS. Furthermore, SMS-PBFS is run in two variants: *SMS-PBFS (byte)* uses an array of bytes for *seen*, *frontier*, and *next*, and *SMS-PBFS (bit)* uses an array of bits.

Our test machine is a 4-socket NUMA system with 4x Intel Xeon E7-4870 v2 CPUs @ 2.3 GHz with one terabyte of main memory. Across all four CPUs, the system has 60 hardware threads. In the experiments we also used all Hyper-Threads.

We used both synthetic as well as real-world graphs. The synthetic graphs are Kronecker graphs [13] with the same parameters that are used in the Graph500 benchmark. They exhibit a structure similar to many large social networks. Additionally, for validation we also use artificial graphs generated by the LDBC data generator [9]. The generated LDBC graphs are designed to match the characteristics of real social networks very closely. Our used real-world graphs are chosen to cover different domains with various characteristic: the twitter follower graph, the uk-2005 web crawl graph and the hollywood-2011 graph describing who acted together in movies. The uk-2005 and hollywood-2011 graph were provided by the WebGraph project [7]. Table 1 lists the properties of all graphs used in our experiments. For the Kronecker graph we omit some of the in-between sizes as they always grow by a factor of 2. The vertex counts only consider vertices that have at least one neighbor. KG0 is a special variation of the Kronecker graph that was used in the evaluation of [14]; it was generated using an average out-degree of 1024.

The listed memory size is based on using 32-bit vertex identifiers and requiring $2 * vertex\_size = 8$ bytes per edge. To measure the performance of MS-BFS we use the source code published on github[1]. For comparison with Beamer, we used their implementation provided as part of the GAP Benchmark Suite (GAPBS) [6]. We did not have access to an implementation of Yasui et al. or iBFS; instead, we compare to published numbers on a similar machine using the same graph.

Our basic metric for comparison is the edge traversal rate (GTEPS). The Graph500 specification defines the number of traversed edges per source as number of input edges contained in the connected component which the source belongs to. Compared to the runtime which increases linearly with the graph size, this metric it is more suitable to compare performance across different graphs. In the MS-BFS paper, the number of edges was calculated by summing up the degrees of all vertices in the connected component. In the official benchmark, however, each undirected edge is only counted once. We use this method in our new measurements. To compare the numbers in this paper with the number of the MS-BFS paper, the other numbers have to be divided by two. In order to give an intuition about the effort required to analyze a specific graph we also show the time MS-PBFS requires for processing 64 sources in Table 1.

5.1 Labeling Comparison

To evaluate the different labeling approaches, we ran both MS-PBFS and SMS-PBFS using 120 threads on a scale 27 Kronecker graph with work stealing scheduling. The three tested variants are random vertex labeling (random), degree-ordered labeling (ordered), and our proposed striped vertex labeling (striped). The average runtime per BFS iteration for each scheme and algorithm is shown in Figure 8. Our results show that degree-ordered labeling exhibits significantly better runtimes that random labeling for the MS-PBFS algorithm. Especially in the most expensive third iteration, the difference between the approaches is close to a factor of two. This supports the results of existing work about graph re-labeling[19].

[1]https://github.com/mtodat/ms-bfs

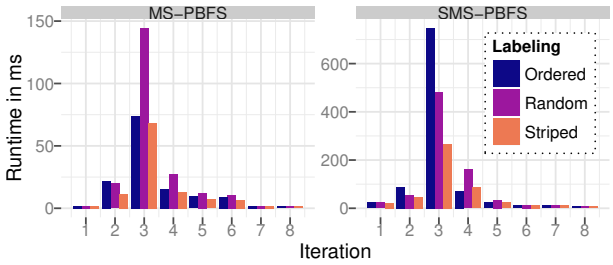

Figure 8: Runtime of BFS iterations under different vertex labeling strategies using SMS-PBFS and MS-PBFS.

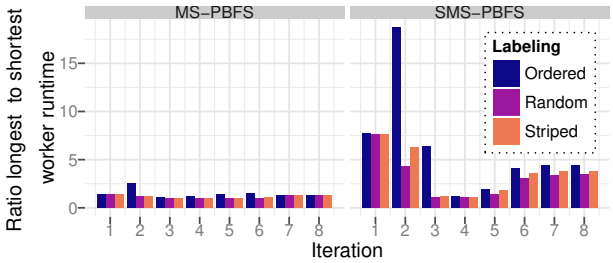

Figure 9: Skew in worker runtimes per iteration when running MS-PBFS and SMS-PBFS with different vertex labelings.

In contrast, for our array-based parallel single-source SMS-PBFS, random ordering exhibits better runtimes. Here, the skew, described in Section 4, and its related problems show their full impact. We evaluated this further in Figure 9 which shows the runtime difference between the longest to the shortest worker per iteration for all three labeling approaches. We see that skew is a much larger problem for SMS-PBFS than for MS-PBFS. Especially in the costly third iteration, there is a significant difference—more than factor 15 for degree-ordered—between *worker* runtimes per iteration. In MS-PBFS skew is a smaller problem as a much larger number of vertices is active in each iteration as there are so many BFSs active at once. Our novel striped vertex ordering shows the best overall runtimes and also balances the workload well. It combines the benefits of degree-based and random ordering in SMS-PBFS and MS-PBFS, while avoiding the other labelings' disadvantages. Similar to random labeling, striped vertex ordering provides good skew resilience, and like degree-ordering, it achieves very good cache locality. Using SMS-BFS the overall runtimes per BFS were: 42ms (striped), 86ms (ordered), 68ms (random).

5.2 Sequential Comparison

In this section, we analyze SMS-PBFS in a sequential setting and compare it against Beamer et al.'s state-of-the-art in sequential single-source BFSs. In addition to Beamer's GAPBS implementation, we also implemented two variants of their BFS that use the same graph, data structure and

Name	Nodes ($x10^6$)	Edges ($x10^6$)	Memory size (GB)	MS-PBFS (runtime per 64)	MS-PBFS (GTEPS)	MS-BFS (GTEPS)	MS-BFS 64 (GTEPS)	SMS-PBFS (GTEPS)
Kronecker 20	2^{20}	15.7	0.119	3.27 ms	**307**	160	4.44	56.2 (bit)
Kronecker 26	2^{26}	1,050	7.96	246 ms	**274**	65.8	2.23	58.9 (bit)
Kronecker 32	2^{32}	68,300	5q5	39,700 ms	**110**	*failed (OOM)*	0.845	76.7 (bit)
KG0	0.982	364	2.72	12.5ms	**1860**	241	11.2	110 (bit)
LDBC 100	1.61	102	0.764	24.4 ms	**267**	76.6	3.01	39.2 (byte)
LDBC 1000	12.4	1010	7.61	551 ms	**118**	45.5	1.30	83.2 (byte)
Hollywood-2011	1.99	114	0.860	49.8 ms	**147**	59.6	2.19	26.5 (byte)
UK-2005	39.5	783	5.98	2220 ms	**22.6**	13.2	0.773	4.96 (bit)
Twitter	41.7	1,200	9.11	934 ms	**82.4**	32.5	1.13	21.0 (bit)

Table 1: Graphs description and algorithm performance in GTEPS using 60 threads.

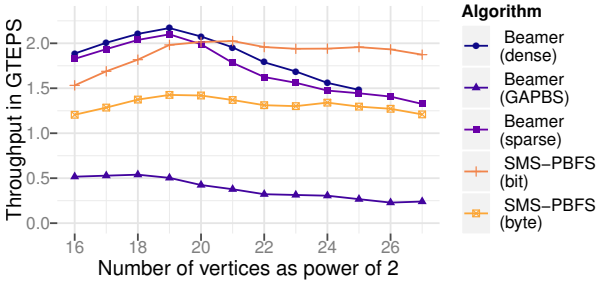

Figure 10: Performance of single-threaded BFS runs over varying graph sizes

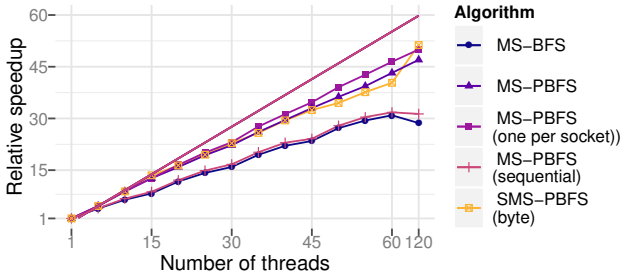

Figure 11: Relative speedup as number of threads increase in a 2^{26} vertices Kronecker graph

chunk skipping optimizations which we use for SMS-PBFS (bit). In the first variant, the queues in the top-down phase are backed by a sparse vector, and in the second variant we used a dense bit array. Both variants use the same bottom-up implementation.

Figure 10 shows the single-source BFSs throughput on a range of Kronecker graphs. The measurements show that for graphs with as few as 2^{20} vertices, our SMS-PBFS is already faster than Beamer et al.'s BFS. As the graph size increases, the probability that the data associated with a vertex is in the CPU cache decreases. There, our top-down approach benefits from having fewer non-sequential data accesses. On the other hand, our BFS has to iterate over all vertices twice. At small graph sizes this overhead can not be recouped as the reduction of random writes does not pay off when the write locations are in the CPU cache. For larger graph sizes, the improvement of SMS-PBFS over our Beamer implementation is limited, as the algorithms only differ in the top-down algorithms but a majority of the runtime in each BFS is spent in the bottom-up phase.

5.3 Parallel Comparison

In this section we compare our (S)MS-PBFS algorithms against the MS-BFS algorithm in a multi-threaded scenario. Inspired by the Graph500 benchmark, we fix the size of a batch for all algorithms to at most 64 sources. The MS-BFS algorithm is sequential and can only utilize all cores by running one algorithm instance per core. Thus, it requires at least $batch\_size * num\_threads = 7,680$ sources to fully utilize the machine. To minimize the influence of straggling threads when executing MS-BFS we used three times as many sources for all measurements. All algorithms have to analyze the same set of source vertices that were randomly selected from the graph. For MS-BFS, the sources are processed one 64-vertex batch at a time per CPU core. MS-PBFS can saturate all compute resources with a single 64-vertex batch; it, thus, analyzes one batch at a time. SMS-PBFS analyzes all sources one single source at a time, utilizing all cores.

5.3.1 *Thread Count Scaling*

To ensure that the amount of work is constant in the CPU scaling experiments, we kept the number of sources fixed even when running with fewer cores. The first 15 cores are located on the first CPU socket, 16–30 on the second socket, 31–45 on the third and 46–60 on the fourth. Figure 11 shows that MS-PBFS scales better than MS-BFS even though the latter has no synchronization between the threads. MS-PBFS (sequential) which uses the same optimizations as MS-PBFS but is executed like MS-BFS with one BFS batch per core exhibits the same limited scaling behavior for large graphs. This contradicts the MS-BFS paper's hypothesis that multiple sequential instances always beat a parallel algorithm as no synchronization is required. The explanation for this can be found when analyzing the cache hit rates. With our (S)MS-PBFS algorithms, the different CPU cores share large portions of their working set, and, thus, can take advantage of the sockets' shared last level caches. In contrast, each sequential MS-BFS mostly uses local data structures; only the graph is shared. This diminishes CPU caches' efficiency.

The scalability of around factor 45 for MS-PBFS and factor 35 for SMS-PBFS using 60 threads is comparable to the results reported by Yasui et al. [20] for their parallel single-source BFS. This is a very good result especially as our multi-source algorithm operates at a much higher throughput of 274 GTEPS compared to their best reported result of around 60 GTEPS on a similar machine in the Graph500 benchmark. The close results between the MS-PBFS (one per socket) variant, where all data except for the graph is completely local, and MS-PBFS show that our algorithm is mostly resilient to NUMA effects and is not limited by contention.

When analyzing the performance gains achieved by the

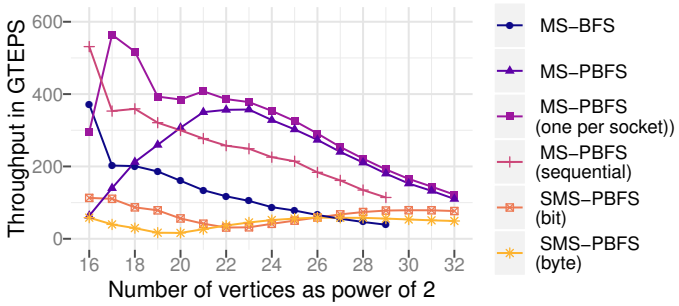

Figure 12: Throughput using 60 cores as graph size increases

additional 60 Hyper-Threads, the difference between multi-source and single-source processing is clearly visible. SMS-PBFS is memory latency-bound and does not saturate the memory bandwidth; thus, it can gain additional performance by having more threads. The multi-source algorithms on the other hand are already mostly memory-bound, and, thus, they do not benefit from the additional threads.

5.3.2 *Graph Size Scaling*

Orthogonal to the thread count scaling experiment, we also measured how the algorithms behave for various graph sizes using Kronecker graphs. We use graph sizes from approximately 2^{16} to 2^{32} vertices and 1 million to 68 billion edges, respectively. As the traversal speed should be independent of the graph size, an ideal result would have constant throughput. Our measured results are shown in Figure 12. MS-BFS as well as the sequential MS-PBFS variant show a continuous decline in performance as the graph size increases. This can be explained with memory bottlenecks, as for larger graph sizes a smaller faction of the graph resides in the CPU cache, and more data has to be fetched from main memory.

In contrast, the parallel BFSs struggle at small graph sizes. Their two biggest problems are contention and that there is only very little work per iteration (on average less than 1 ms runtime). The reason for the contention in very small graphs is the high probability that in the top-down phase multiple threads will try to update the same entry. Furthermore, for small graphs, the constant overheads for task creation, memory allocation, etc., have a relatively high impact on the overall result.

Parallelization is much more important for large graph sizes. Starting at 2^{20} (around 1 million) vertices, MS-PBFS manages to beat the MS-PBFS (sequential) implementation. At this size, MS-PBFS requires 3.27ms for one batch of 64 sources. At larger sizes a decline in performance can be measured again, caused by a reduction in cache hit rates resulting in memory bandwidth bottlenecks. SMS-PBFS maintains its peak performance for a larger range of graph sizes, though at a lower level. As it only operates on a single BFS, it is more bound by memory latency in case of a cache miss than by memory bandwidth. Other BFS approaches [2, 20] also exhibit a similar drop in performance at larger scales. The measurement for MS-BFS and MS-PBFS (sequential) only include graphs up to scale 29, as at larger graph sizes the available one terabyte of memory did not suffice to run 120 instances of the algorithms, demonstrating the severe limitations of MS-BFS in a multi-threaded scenario.

In Table 1 we summarize the algorithms' performance for real world datasets. Additionally, we show the performance when MS-BFS is only limited to processing 64 sources at a time (MS-BFS 64) like MS-PBFS. The results show that in this kind of use case the performance of MS-BFS is very low as it can only utilize one CPU. Overall, our measurements show that even if MS-BFS is given enough sources to utilize all cores, MS-PBFS performs significantly better on large graphs.

We also wanted to compare to the currently fastest parallel single-source BFS by Yasui and fastest parallel multi-source iBFS but did not have access to their implementations. As we could evaluate our algorithms on the same synthetic graphs, we instead compare to their published numbers. For Yasui et al. we compare our SMS-PBFS to their most recent numbers published on the Graph500 ranking. Their results in the Graph500 ranking are based on a CPU that is about 20% faster than ours but from the same CPU generation so they should be comparable. Their result places them 67st overall, 1st single-machine (CPU-only), on the June 2016 ranking and they achieve a throughput of 59.9 GTEPS compared to the 76.7 GTEPS demonstrated by our single-source SMS-PBFS on the same scale 32 graph. For iBFS we use the numbers from their paper, they do not use the default kronecker graph generator settings but test on graphs with larger degrees. We compare MS-PBFS against their algorithm on the KG0 graph where they report their best performance. Using 64 threads, the iBFS CPU implementation reaches 397 GTEPS, and their GPU implementation 729 GTEPS. MS-PBFS reaches 1860 GTEPS on 120 threads showing a significant improvements even when accounting for number of threads.

6. RELATED WORK

The closest work in the area of multi-source BFS algorithms are MS-BFS [18] and the iBFS[14] which is designed for GPUs. Compared to the first algorithm, our parallelized approach using striped labeling significantly improves the performance, and reduces the memory requirements. Furthermore, MS-PBFS enables the use of multi-source BFS in a wider setting by providing full performance also with a limited number of sources. iBFS describes a parallelized approach for GPUs which uses a sparse joint frontier queue (JFQ) containing the active vertices for a iteration. By using special GPU voting instructions, it manages to avoid queue contention on the GPU. However, those instructions don't have equivalents on mainstream CPU architectures. Consequently, the CPU adaption of their algorithm exhibits significantly lower performance than ours.

The work on parallel single-source algorithms primarily focuses on how to reduce the cost of insertion into the *frontier* and *next* queues. Existing approaches span from using multi-socket-optimized queues like FastForward [10], to batch insertions and deletions [2], as well as to having a single queue per NUMA node as it is used by the parallel Yasui BFS [20]. Yet, all of these approaches have in common that they share a single insertion point either at the global level or per NUMA node. Even if organized at NUMA socket granularity there is potentially a lot of contention, and the trend of having more cores per CPU does not work in such approaches' favor. The work of Chhugani et al. [8] which also focuses on dynamic load balancing has similar limitations as it only focuses on distributing work inside each NUMA socket. Our analysis shows that while this may be sufficient for sparse queue-based

algorithms, it would not provide scalability in array-based algorithms.

7. CONCLUSION

In our work we presented the MS-PBFS and SMS-PBFS algorithms that improve on the state-of-the art BFS algorithms in several dimensions.

MS-PBFS is a parallel multi-source breadth-first search that builds on MS-BFS's principles of sharing redundant traversals in concurrent BFSs in the same graph. In contrast to MS-BFS, our novel algorithm provides CPU scalability even for a limited number of source vertices, fully utilizing large NUMA systems with many CPU cores, while consuming significantly less memory, and providing better single-threaded performance. Our parallelization and NUMA optimizations come at minimal runtime costs so that no separate algorithms are necessary for sequential and parallel processing, neither for NUMA and non-NUMA systems.

SMS-PBFS is a parallel single-source BFS that builds on the ideas of MS-PBFS. Compared to existing state-of-the-art single-source BFSs, our proposed SMS-PBFS algorithm provides comparable scalability at much higher absolute performance. Unlike other BFS algorithms, SMS-PBFS has a simple algorithmic structure, requiring very few atomic instructions and no complex lock or queue implementations. Our novel striped vertex labeling allows more coarse-grained task sizes while limiting the skew between task runtimes. Striped vertex labeling can also be used to improve the performance of other BFS algorithms.

8. ACKNOWLEDGMENTS

Manuel Then is a recipient of the Oracle External Research Fellowship.

9. REFERENCES

[1] The Graph 500 Benchmark. http://www.graph500.org/specifications. Accessed: 2016-09-09.

[2] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proc. of the 22nd IEEE and ACM Supercomputing Conference (SC10)*, SC '10, pages 1–11. IEEE, 2010.

[3] L. A. N. Amaral, A. Scala, M. Barthélémy, and H. E. Stanley. Classes of small-world networks. *PNAS*, 97(21), 2000.

[4] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013.

[5] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[6] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[7] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW '04*, pages 595–601, 2004.

[8] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 378–389, May 2012.

[9] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630. ACM, 2015.

[10] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 43–52. ACM, 2008.

[11] 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.

[12] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 78–88, Oct 2011.

[13] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, Mar. 2010.

[14] H. Liu, H. H. Huang, and Y. Hu. iBFS: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 403–416. ACM, 2016.

[15] 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. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[16] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146. ACM, 2010.

[17] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456, Oct 2015.

[18] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment*, 8(4), 2014.

[19] Y. Yasui and K. Fujisawa. Fast and scalable NUMA-based thread parallel breadth-first search. In *High Performance Computing & Simulation (HPCS)*, pages 377–385. IEEE, 2015.

[20] Y. Yasui, K. Fujisawa, and Y. Sato. Fast and energy-efficient breadth-first search on a single NUMA system. In *Proceedings of the 29th International Conference on Supercomputing*, ISC 2014, pages 365–381. Springer-Verlag New York, Inc., 2014.

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2. USENIX Association, 2012.